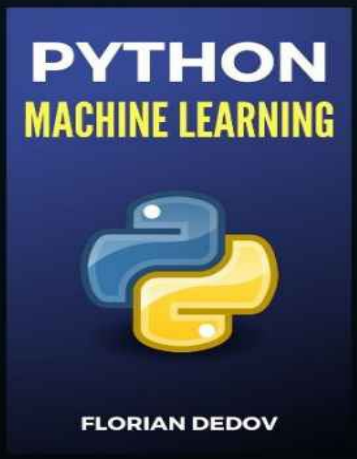
PYTHON

7 IN 1



FLORIAN DEDOV

PYTHON PROGRAMMIEREN

7 IN 1

Von

FLORIAN DEDOV

Dieses Buch ist eine 7-in-1 Version der Serie „Python Programmieren – Der schnelle Einstieg“

Lesen Sie daher am besten alle Bücher in der richtigen Reihenfolge, da das Wissen aufeinander aufbaut!

Falls Sie nach dem Lesen dieses Buches, der Meinung sind, dass es positiv zu Ihrer Programmierkarriere beigetragen hat, würde es mich sehr freuen, wenn Sie eine Rezension auf Amazon hinterlassen.

Danke!

PYTHON PROGRAMMIEREN FÜR ANFÄNGER

FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

Python gewinnt in den letzten Jahren massiv an Popularität und wird wirtschaftlich immer relevanter. Die Programmiersprache findet ihre Anwendung in verschiedensten Bereichen, wie Web Development, Data Science, Neuronalen Netzen, Machine Learning, Robotik und vielen mehr. Alles Bereiche, welche unsere Zukunft maßgeblich prägen werden. Es sind bereits jetzt zahlreiche Errungenschaften und Fortschritte gelungen und es werden mit Sicherheit noch sehr viele folgen. Wenn Sie ein Teil dieser Entwicklung sein wollen, ist es definitiv eine gute Idee sich näher mit Python zu befassen. Die Sprache ist sehr simpel zu erlernen und hat eine einfache Syntax. Mittlerweile gehört sie zu den bedeutsamsten Sprachen unserer Zeit und die Chancen stehen außerordentlich gut, dass sie noch relevanter wird.

In diesem Buch werden Ihnen die fundamentalen Grundkenntnisse der Programmierung mit Python nähergebracht. Dabei werden keine Vorkenntnisse im Bereich der Programmierung oder der Informatik im Allgemeinen benötigt. Dieses Buch richtet sich an absolute Einsteiger. Es wird also alles von der Pike auf gelernt. Dennoch können Sie mit Sicherheit auch als Fortgeschrittener Coder hiervon profitieren, wenn Sie damit ihr Wissen auffrischen möchten.

Nachdem Sie dieses Buch gelesen haben und das Gelernte auch nebenbei angewandt haben, werden Sie in der Lage sein, grundlegende Konzepte der Programmierung zu verstehen und diese in der Skriptsprache Python anzuwenden. Kurzgesagt: Sie haben eine solide Basis für eine erfolgreiche Programmierkarriere.

Sie sollten dieses Buch jedoch nicht nur schnell durchlesen und versuchen oberflächlich zu verstehen, worum es geht, sondern auch aktiv nebenbei mitprogrammieren. Dadurch lernen Sie am meisten und das Wissen wird am besten gefestigt. In den folgenden Kapiteln werden Sie zahlreiche Codebeispiele finden, die Sie nachprogrammieren können. Sie können jedoch auch Ihren eigenen Code schreiben, mithilfe der Dinge die Sie bereits gelernt haben, zu dem Zeitpunkt.

Im Grunde genommen bleibt es Ihnen selbst überlassen, wie Sie diese Lektüre handhaben möchten. Das Buch ist kompakt und simpel gehalten, sodass Sie es rein theoretisch innerhalb von wenigen Stunden, aber auch innerhalb von mehreren Tagen lesen können. In jedem Fall wünsche ich Ihnen maximalen Erfolg beim Programmieren und viel Spaß beim Lernen von Python.

1 – INSTALLATION VON PYTHON

Da Python eine interpretierte, höhere Programmiersprache ist brauchen wir bestimmte Werkzeuge um unsere Programme auszuführen. Wie bei jeder anderen Programmiersprache, kann der Code selbst in jedem beliebigen Editor (z.B. Notepad++, Atom, Sublime…) geschrieben werden. Um jedoch den Programmcode einer interpretierten Sprache auszuführen, benötigt man einen sogenannten Interpreter.

INSTALLATION DER IDLE

Auf https://www.python.org/downloads finden Sie die verschiedenen Versionen von Python zum Herunterladen. Wir werden in diesem Buch mit der dritten Version arbeiten. Dabei ist es nicht entscheidend welche genau, sondern einfach, dass sie das Schema 3.x.x hat. Wenn Sie Python zusammen mit der IDLE erfolgreich installiert haben, so können sie diese, unter Windows, über die Suchfunktion finden. Geben sie dazu einfach *IDLE* ein.

WAS IST DIE IDLE?

IDLE ist eine englische Abkürzung und bedeutet *Integrated Development Environment.* Die IDLE können Sie (müssen jedoch nicht) benutzen, um Python Scripts auszuführen und/oder zu schreiben. Ich persönlich benutze zum schreiben meines Codes den Open-Source Editor Atom und führe dann meine Scripts über die Konsole (CMD oder PowerShell) aus. Wie Sie es machen, bleibt Ihnen überlassen.

2 – DAS ERSTE PROGRAMM

Wie in jeder anderen Programmiersprache, beginnt man, auch in Python, typischerweise mit einer klassischen *Hello World Anwendung*. Dabei handelt es sich um ein Programm, welches nichts anderes tut, als den Text „Hello World“ auszugeben.

HELLO WORLD

Der folgende Code ist ein Hello World Programm:

```
print("Hello World")
```

Wenn dieser Code ausgeführt wird (wie das geht finden wir gleich heraus), bekommt man also auf die Konsole den Text „Hello World“ ausgegeben. Die sogenannte Funktion *print*, welche hier aufgerufen wird, gibt den ihr übergebenen Text aus. Dieser befindet sich in den runden Klammern und wird durch Anführungszeichen als String definiert. Wenn die Anführungszeichen fehlen denkt der Interpreter, dass es sich um eine Variable handelt, welche, in unserem Fall, nicht existiert.

AUSFÜHREN VON PYTHON SCRIPTS

Es gibt viele Wege um ein Python Skript auszuführen, sogar sehr viele. Wir werden in diesem Buch unsere Programme über die Kommandozeile ausführen. Dazu öffnet man unter Windows die Eingabeaufforderung (CMD) oder die PowerShell. Dort navigiert man mit dem Befehl – **cd <pfad>** - zu dem Skript und führt es mit dem Befehl – **python <scriptname.py>** aus.

3 – VARIABLEN

Sie werden Variablen wahrscheinlich bereits aus der Mathematik kennen, wo sie Platzhalter für Zahlenwerte oder komplexere Strukturen (z.B. Matrizen, Funktionen) darstellen. Auch in der Programmierung erfüllen sie dieselbe Funktion. Sie reservieren einen bestimmten Speicherbereich im RAM und werden im Code dafür benutzt um Werte zu speichern. Anders als in der Mathematik, kann man in Programmiersprachen auch Texte, boolesche Werte und eigens definierte Objekte speichern.

DATENTYPEN

Werfen wir erst mal einen Blick auf die verschiedenen Datentypen welche Python von Haus aus mitliefert. Jeder Datentyp hat in Python ein bestimmtes Schlüsselwort (Keyword) mit welchem man diesen definieren kann.

NUMERISCHE DATENTYPEN

Numerische Datentypen		
Datentyp	Keyword	Beschreibung
Integer	int	Eine Ganzzahl
Float	float	Eine Fließkommazahl
Complex	complex	Eine komplexe Zahl

STRINGS

Strings sind einfach nur Zeichenketten, sprich, Text. Mit einem String kann man nicht rechnen. Er ist nur eine Ansammlung von Zeichen. Es gibt jedoch String-Funktionen zu welchen wir später noch kommen werden.

SEQUENZEN

Sequenzen werden wir in einem späteren Kapitel noch näher beleuchten. Derzeit ist es nur wichtig einen Überblick darüber zu bekommen, welche es gibt.

Sequenzarten		
Datentyp	Keyword	Beschreibung
List	list	Ansammlungen von vielen Variablen
Tuple	tuple	Unveränderbare Listen
Dictionary	dict	Listen mit Key-Value Pairs

BOOLEAN

Ein Boolean ist ein Wert welcher immer nur entweder *wahr* oder *falsch* sein kann. Die Ergebnisse von Vergleichen (Kapitel 4) sind immer Booleans. Das Keyword lautet *bool*.

ERSTELLEN VON VARIABLEN

In Python erstellt man Variablen wie folgt:

```
variablenName = 10
variablenName2 = "Hallo"
```

Man schreibt als erstes den Namen der Variable hin, gefolgt von einem Ist-Gleich-Zeichen und dem Wert welchen man der Variable zuweisen möchte. Der Variablenname kann grundsätzlich beliebig gewählt werden, jedoch darf er nicht mit einer Zahl oder einem Sonderzeichen (außer dem Unterstrich) beginnen. In Python muss man den Datentypen vorerst nicht extra angeben, da der Interpreter selbst erkennt um welchen es sich handelt. Eine Angabe des Datentyps ist bei einer Konversion erforderlich.

KONVERTIEREN VON DATENTYPEN

Wenn wir eine Variable von einem Datentyp haben, welchen wir gerne in einen anderen konvertieren möchten, so müssen wir die Variable *typecasten.*

```
eingabe = "20"
zahl = int(eingabe) # Speichert die Texteingabe als Zahl
text = str(zahl) # Speichert die Zahl wieder als Text
```

Um in Python Variablen zu konvertieren, müssen wir die Funktion des jeweiligen Datentyps nutzen. Diese Funktion ist der Name des Keywords. In die Klammern kommt dann die Variable, welche konvertiert werden soll.

Warum man das macht? Nehmen wir mal an, der User gibt, durch eine Benutzereingabe, eine Zahl ein. Diese Zahl ist dann als String abgespeichert, weil Eingaben standardmäßig Texte sind. Wenn wir mit dieser Eingabe rechnen möchten müssen wir sie zuvor in eine Zahl konvertieren. Dies geht jedoch nur, wenn in dem Text ausschließlich eine Zahl enthalten ist. Ansonsten bekommen Sie eine Fehlermeldung und das Skript schmiert ab.

4 – OPERATOREN

Um mit den Variablen bestimmte Operationen durchführen zu können, benötigen wir sogenannte Operatoren. Es gibt zahlreiche Operatoren, aber da es sich hier um ein Buch für Anfänger handelt werden wir nur auf die fundamentalen eingehen, die wir für dieses Buch benötigen.

ARITHMETISCHE OPERATOREN

Arithmetische Operatoren		
Operator	Name	Beschreibung
+	Addition	Addiert zwei Variablen
-	Subtraktion	Subtrahiert eine Variable von der anderen
*	Multiplikation	Multipliziert zwei Variablen
/	Division	Dividiert eine Variable durch die Andere
%	Modulo	Gibt den Rest einer Division von zwei Variablen zurück
**	Exponent	Potenziert eine Zahl mit der anderen

```
a = 8
b = 2
x = a + b # = 10
x = a - b # = 6
x = a * b # = 16
x = a / b # = 4
x = a % b # = 0
x = a ** b # = 64
```

Am besten Sie experimentieren selbst ein wenig mit den Operatoren herum indem sie den *print* Befehl nutzen um das Ergebnis einer Rechnung auszugeben. Sie können auch Variablen ausgeben lassen (auch Zahlen).

VERGLEICHSOPERATOREN

Vergleiche liefern immer einen Boolean als Ergebnis. Entweder stimmt der Vergleich (wahr) oder er stimmt nicht (falsch).

Vergleichsoperatoren		
Operator	Name	Beschreibung
==	Gleich	Zwei Variablen sind gleich
!=	Ungleich	Zwei Variablen sind nicht gleich
>	Größer	Eine Variable ist größer als die andere
<	Kleiner	Eine Variable ist kleiner als die andere
>=	Größer-Gleich	Eine Variable ist größer oder gleich der anderen
<=	Kleiner-Gleich	Eine Variable ist kleiner oder gleich der anderen

```
a, b = 20, 10
print(a == b) # FALSE
print(a != b) # TRUE
print(a > b) # FALSE
print(a < b) # TRUE
print(a >= b) # FALSE
print(a <= b) # TRUE
```

Sobald man einen Vergleich ausgeben lässt, erhält man den Wahrheitswert als Ausgabe. Die erste Zeile soll Sie nicht verwirren. Es ist einfach eine Schreibweise, in welcher man mehreren Variablen gleichzeitig einen Wert zuweisen kann, innerhalb einer Zeile.

ZUWEISUNGSOPERATOREN

Neben dem Ist-Gleich-Zeichen gibt es auch noch einige andere Zuweisungsoperatoren welche man benutzen kann um Variablen einen bestimmten Wert zuzuweisen.

Zuweisungsoperatoren	
Operator	Beschreibung
=	Einer Variable wird der Wert zugewiesen
+=	Eine Variable wird um den Wert erhöht
-=	Eine Variable wird um den Wert verringert
***=**	Eine Variable wird mit dem Wert multipliziert
/=	Eine Variable wird durch den Wert dividiert
%=	Eine Variable wird zum Restwert der Division durch den Wert
****=**	Eine Variable wird zum Ergebnis des Potenzierens mit dem Wert

```
a = 10
a += 10 # a ist 20
b = 10
b *= 5 # b ist 50
```

LOGISCHE OPERATOREN

Logische Operatoren		
Operator	Name	Beispiel
or	Oder	True **or** False = True
and	Und	True **and** False = False True **and** True = True
not	Nicht	**not** True = False **not** False = True

Logische Operatoren werden, genau wie Vergleichsoperatoren bei Bedingungen (Kapitel 5) sehr wichtig sein. Man kann logische Operatoren gut mit Vergleichsoperatoren verbinden.

5 – BEDINGUNGEN

Wenn wir in einem Skript Verzweigungen haben möchten, sprich, wir je nach Sachverhalt entscheiden wollen, was getan werden soll brauchen wir Bedingungen. In Programmiersprachen nennen wir diese *If-Statements* und *Else-Statements*.

IF-STATEMENTS

```
a, b = 20, 10

if(a > b):
    print("a ist größer!")
```

In die Klammern einer If-Abfrage befindet sich immer ein boolescher Ausdruck, d.h. ein Ausdruck welcher entweder wahr oder falsch zurückgibt. Hier prüfen wir ob a (20) größer ist als b (10), was den Wert *True* zurückliefert. Nach der Abfrage folgt ein Doppelpunkt welcher anzeigt, dass jetzt der Code folgt, welcher ausgeführt wird, wenn die Bedingung wahr ist. Hierbei ist es außerordentlich wichtig, dass ordnungsgemäß eingerückt wird. Wenn in Python der Code unter einem Doppelpunkt nicht eingerückt ist, wird er nicht damit in Verbindung gesetzt.

ELSE-STATEMENTS

In dem oberen Codebeispiel wird die Bildschirmausgabe durchgeführt, wenn die Bedingung zutrifft und andernfalls läuft das Skript ganz normal weiter. Wollen wir jedoch selber entscheiden was passiert, wenn der Ausdruck *False* zurückliefert, so benötigen wir Else-Statements.

```
a, b = 20, 10

if(a > b):
    print("a ist größer!")
else:
    print("a ist nicht größer!")
```

Wie Sie sehen können, wird auch das Else-Statement gefolgt von einem Doppelpunkt. Wichtig ist hierbei, dass das *else* nicht eingerückt wird, da es einen eigenen Zweig außerhalb des *if* darstellt.

ELIF-STATEMENTS

Das Schlüsselwort *elif* ist eine Abkürzung für Else-If. Das Skript begibt sich genau dann in einen Elif-Zweig, wenn die If-Bedingung nicht erfüllt wurde, die Elif-Bedingung jedoch schon. Das heißt, dass der Code nicht ausgeführt wird, sollte die erste If-Bedingung schon erfüllt sein, wie es bei aneinander gereihten If-Statements der Fall wäre.

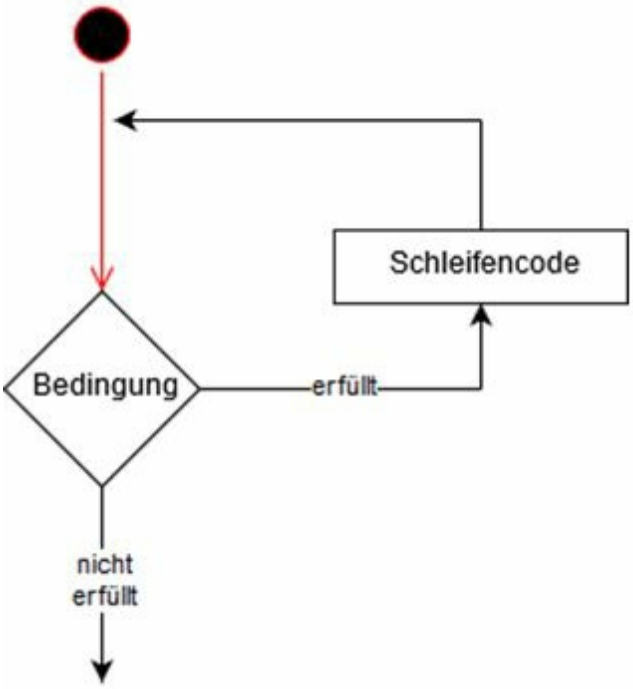
```
a, b = 10, 20

if(a > b):
    print("a ist größer als b!")
elif(a == b):
    print("a ist gleich b!")
elif(a < b):
    print("a ist kleiner als b!")
else:
    print("Etwas anderes ist der Fall!")
```

Wie in diesem Beispiel zu sehen ist, kann auch noch ein *else* am Ende angehängt werden für den Fall, dass keine Bedingung erfüllt wurde.

6 – SCHLEIFEN

Mit If- und Else-Statements können Sie den Verlauf des Skripts also, je nach Sachverhalt, in eine bestimmte Richtung lenken. Wenn Sie jedoch eine Aktion, bzw. einen Codeabschnitt, solange hintereinander ausführen wollen, solange eine bestimmte Bedingung erfüllt ist, benötigen Sie sogenannte Schleifen.



Hier sehen sie wie eine Schleife funktioniert. Jedes Mal, wenn die Schleife den Code ausgeführt hat, wird die Bedingung geprüft. Ist diese immer noch erfüllt, so wird das Ganze wiederholt. Dieser Prozess geht weiter bis sie irgendwann nicht mehr erfüllt ist.

WHILE-SCHLEIFEN

Die erste Schleifenart, welche wir uns ansehen werden, ist die sogenannte While-Schleife. While bedeutet auf Englisch *solange*. Das heißt, solange eine Bedingung gilt, führt diese Schleife den Code aus.

```
x = 0

while(x < 10):
    print(x)
    x += 1
else:
    print("x ist jetzt nicht mehr kleiner als 10")
```

In diesem Codebeispiel, wird die Variable x, solange ausgegeben und anschließend um Eins erhöht, solange sie kleiner ist als Zehn. Außerdem, kann an While-Schleifen auch ein *else* angehängt werden, welches den Code beinhaltet, welcher ausgeführt werden soll, falls die Bedingung nicht (mehr) zutrifft.

UNENDLICHE SCHLEIFE

Der folgende Code zeigt eine While-Schleife, welche nicht enden wird, bevor das Programm selbst dies tut.

```
while(True):
    print("Diese Schleife endet nicht!")
```

Da die Bedingung der Schleife konstant auf *True* gesetzt ist, ist sie immer erfüllt und der Code wird unendlich lange ausgeführt.

FOR-SCHLEIFEN

Das Schlüsselwort *for* bedeutet auf Englisch *für* und in diesem Zusammenhang bedeutet es *„für jedes Element in“*. Im Prinzip iteriert eine For-Schleife über eine Ansammlung von Dingen. Für jedes Element in dieser Ansammlung wird der Code der Schleife einmal ausgeführt.

```
for i in range(5, 10)
    print(i) # Gibt alle Zahlen zwischen 5 und 10 aus
```

Eine For-Schleife hat immer eine sogenannte Laufvariable, welche bei jedem Durchlauf einen anderen Wert hat. In dem oberen Beispiel iterieren wir über den Zahlenbereich von Fünf bis Zehn. In diesem Fall, ist die Laufvariable i, bei jedem Durchlauf, um Eins größer.

ITERIEREN ÜBER STRINGS

Folgendes Beispiel zeigt wie man eine For-Schleife auf einen String anwenden kann.

```
for l in "Hallo":
    print(l) # Gibt jeden Buchstaben einzeln aus
```

ITERIEREN ÜBER LISTEN

Auch wenn Sequenzen erst im nächsten Kapitel besprochen werden, sehen Sie im folgenden Beispiel, das Iterieren über eine Liste.

```
liste = [400, 23, 1, 43, "Hallo"]

for i in liste:
    print(i) # Gibt alle Elemente der Liste aus
```

STEUERUNG VON SCHLEIFEN

Wenn Sie an einem bestimmten Punkt, in einer Schleife, den Ablauf dieser manipulieren wollen, so benötigen Sie sogenannte *Loop-Control-Statements*.

BREAK-STATEMENT

Ein Break-Statement bewirkt, das sofortige Terminieren einer Schleife.

```
a = 0
while(a < 10):
   print(a)
   if(a == 5):
      break
```

In diesem Beispiel, hört die Schleife sofort auf, sobald a den Wert Fünf erreicht hat. Der restliche Code des Skripts wird normal weiter ausgeführt.

CONTINUE-STATEMENT

Ein Continue-Statement überspringt nur den aktuellen Durchlauf der Schleife. Danach wird diese normal weiter ausgeführt.

```
liste = [400, 23, 1, 43, "Hallo"]

for i in liste:
   if(i == 23):
      continue;
   print(i)
```

In diesem Beispiel werden Elemente der Liste mit dem Wert 23 nicht ausgegeben. Der Durchlauf wird übersprungen.

PASS-STATEMENT

Das Pass-Statement ist ein ganz besonderes Statement, denn es kann absolut gar nichts. Es wird meistens dann eingesetzt, wenn syntaktisch ein Code gefordert ist, der Entwickler diesen jedoch erst irgendwann später hinzufügen

möchte.

```
a = 0
while(a < 10):
    if(a == 5):
        pass
    print(a)
```

Das *pass* hat in diesem Codebeispiel absolut keine Wirkung. Es ist jedoch syntaktisch nicht zulässig, dass nach einer If-Bedingung kein Code kommt. Es wird quasi als Platzhalter für zukünftigen Code verwendet.

7 – SEQUENZEN

Eine Sequenz ist im Allgemeinen, und auch in der Programmierung, eine Folge von Objekten die in sich geschlossen ist. Sie stellen die simpelste aller Datenstrukturen in Python dar. Jedes Element hat eine eindeutige Nummer, nämlich die Position bzw. der Index. Die Indexierung beginnt mit der Nummer 0 und der letzte Index ist die Anzahl der Elemente minus Eins. Es gibt mehrere Arten von Sequenzen, doch einige Funktionen können auf alle davon angewandt werden.

LISTEN

Die Liste ist einer der vielseitigsten Datentypen in Python. Sie wird mittels eckigen Klammern deklariert und die Objekte innerhalb der Liste, müssen nicht vom selben Typen sein, wie es in anderen Programmiersprachen, wie Java, der Fall ist.

```
liste = [10, "Hans", 8.9, True, 29, False]
```

In dem oben angeführten Codebeispiel sehen Sie wie eine Liste mit verschiedensten Datentypen erstellt wird. Dabei hat der erste Wert (10) den Index 0 und der zweite Wert („Hans“) den Index 1 und das geht weiter bis zum Index 5. Wenn wir diese Werte auslesen wollen müssen wir die Liste mit dem entsprechenden Index aufrufen:

```
print(liste[0]) # Gibt das Element mit dem Index 0 aus
print(liste[3]) # Gibt das Element mit dem Index 3 aus
```

Natürlich ist es auch möglich alle Werte auf einmal mit einer For-Schleife auszulesen wie wir es im letzten Kapitel gesehen haben.

Die selbe Syntax können wir auch verwenden um Werte zu verändern.

```
liste[1] = "Peter" # Ändert den Wert bei dem Index 1
liste[3] = 102 # Ändert den Wert bei dem Index 3
```

Hier würde die Methode mit der For-Schleife nicht funktionieren weil diese nur liest und nicht schriebt. Jedoch wäre es möglich den Index durch eine Schleife immer um eins zu erhöhen und so jeden Wert zu verändern:

Die Funktion *len* welche hier aufgerufen wird gibt die Länge einer Liste zurück. In diesem Fall also Sechs.

```
for i in range(len(liste)):
    liste[i] = 10
```

LISTENOPERATIONEN

Sie können Listen miteinander addieren oder Listen mit Zahlen multiplizieren. Wenn Sie zwei Listen addieren, so erhalten Sie als Ergebnis eine neue Liste mit allen Elementen, beider Listen, aneinandergereiht. Sollten Sie eine Liste mit einer Zahl multiplizieren, so erhalten sie als Ergebnis nicht etwa die Werte in der Liste, multipliziert mit dem Faktor, sondern Sie erhalten eine Liste, mit denselben Werten, mehrmals aneinandergereiht.

Listenoperationen	
Operation	Ergebnis
[1, 2, 3] + [3, 8, 1]	[1, 2, 3, 3, 8, 1]
[12, „Hey] * 3	[12, „Hey“, 12, „Hey“, 12, „Hey“]

LISTENFUNKTIONEN

In Python gibt es folgende Listenfunktionen:

len(liste) Gibt die Länge einer Liste zurück.
max(liste) Gibt den höchsten Wert einer Liste zurück.
min(liste) Gibt den niedrigsten Wert einer Liste zurück.
list(sequenz) Konvertiert eine beliebige Sequenz in eine Liste.

LISTENMETHODEN

Zusätzlich gibt es folgende Listenmethoden:

list.append(objekt) Hängt ein Objekt an die Liste an.
list.count(objekt) Zählt wie oft ein Objekt in der Liste vorkommt.
list.extend(sequenz) Hängt eine beliebige Sequenz an die Liste an.
list.index(objekt) Gibt den Index zurück, bei dem ein Objekt als erstes vorkommt.
list.insert(index, objekt)

Fügt ein Objekt an einem bestimmten Index ein.
list.remove(objekt) Entfernt ein Objekt aus der Liste.
list.reverse() Kehrt die Reihenfolge der Objekte einer Liste um.
list.sort([funktion]) Sortiert eine Liste gemäß einer angegebenen Funktion.

ERWEITERTES INDEXIEREN

Neben dem einfachen Indexieren, indem man eine einfache Zahl in die eckigen Klammern schreibt, gibt es noch ein paar wenige andere Methoden.

```
print(liste[-2]) # Gibt das zweite Element von rechts aus
print[liste[:4]] # Gibt alle Elemente bis zum vierten Index aus
print(liste[1:3]) # Gibt alle Elemente von Index 1 bis 3 aus
print(liste[2:]) # Gibt alle Elemente ab Index 2 aus
```

TUPEL

Ein Tupel ist eine Sequenz, welche, im Gegensatz zu einer Liste, nicht manipuliert werden kann. Das bedeutet, dass es nicht möglich ist Elemente hinzuzufügen, zu entfernen oder zu bearbeiten.

```
tupel = (10, "Anton", 4.3, 29)
```

Während man Listen mittels eckigen Klammern deklariert, deklariert man Tupeln mit runden Klammern.

TUPELOPERATIONEN

Das Auslesen von Werten, so wie das Iterieren über Tupel, funktioniert genau wie bei einer Liste. Sie können Tupel addieren, multiplizieren, aus Tupeln Werte auslesen usw.

TUPELFUNKTIONEN

Alle Lesefunktionen, wie *len*, *max*, *min* usw., können auch auf Tupel angewandt werden, weil diese das Tupel nicht verändern. Es ist jedoch nicht möglich eine Schreibmethode auf diese anzuwenden, da Tupel statisch sind.

DICTIONARIES

Dictionary heißt übersetzt Wörterbuch, oder auch Lexikon, was darauf hindeutet, dass in dieser Datenstruktur ein Objekt auf ein anderes verweist. In anderen Programmiersprache findet man ähnliche Strukturen unter dem Namen *HashMap*. Im Grunde genommen, ist ein Dictionary eine Sequenz, welche pro Eintrag zwei Werte hat: Einen eindeutigen Key und einen Value, die zusammen ein sogenanntes Key-Value-Pair bilden.

```
dictionary = {"Name": "Alex", "Alter": 25, "Groesse": 1.80}
```

Dictionaries werden mit geschwungenen Klammern deklariert, innerhalb welcher wir immer einen Schlüssel, gefolgt von einem Doppelpunkt und dem jeweiligen Wert, finden. In dem oberen Codebeispiel, haben wir die Schlüssel *Name*, *Alter* und *Größe*, mit den dazugehörigen Werten, nach dem Doppelpunkt. Wichtig ist, dass jeder Key nur genau einen einzigen Value haben darf.

```
print(dictionary["Name"]) # Gibt "Alex" aus
print(dictionary["Alter"]) # Gibt 25 aus
dictionary["Alter"] = 30 # Ändert das Alter auf 30
```

Anders als bei Listen und Tupeln, rufen wir bei Dictionaries die einzelnen Werte durch die Angabe der jeweiligen Keys auf. Dementsprechend spielen Indizes hierbei also keine Rolle. Die Keys sind unveränderbare Werte und müssen entweder Strings, Zahlen oder Tupel sein. Die entsprechenden Werte, sind jedoch, im Gegensatz zu Werten in einem Tupel, modifizierbar.

DICTIONARYFUNKTIONEN

Ein Dictionary hat folgende Funktionen:

len(dict) Gibt die Länge eines Dictionarys zurück.
str(dict) Gibt eine String Darstellung des Dictionarys zurück.

DICTIONARYMETHODEN

Zusätzlich gibt es folgende Dictionarymethoden:

dict.clear() Entfernt alle Elemente eines Dictionarys.
dict.copy() Gibt eine Kopie des Dictionarys zurück.
dict.fromkeys(dict) Erstellt ein Dictionary mit denselben Keys und ‘None‘ Werten.
dict.get(key) Gibt den Wert des jeweiligen Key-Value-Pairs zurück.
dict.items() Gibt alle Elemente eines Dictionarys zurück.
dict.setdefault(key) Gibt den Value für einen Key zurück. Wenn dieser Key nicht vorhanden ist, setzt er ihn auf den Default-Wert ‘None‘.
dict.update(dict2) Fügt den Inhalt eines Dictionarys in den eines anderen ein.
dict.values() Gibt alle Werte eines Dictionarys zurück.

AUF WERTE IN SEQUENZEN PRÜFEN

Wenn Sie wissen wollen, ob ein bestimmter Wert in einer Sequenz enthalten ist, so können Sie das mit dem Schlüsselwort *in* machen.

```
liste = [1, 2, 3, 4, 5, 6]
print(2 in liste) # True
print(9 in liste) # False
```

Die Ausdrücke in der Klammer liefern beide einen Wahrheitswert zurück, jedoch nicht denselben, da die Zwei enthalten ist und die Neun nicht.

```
dictionary = {"A": 1, "B": 2, "C": 3}
print("A" in dictionary) # True
print(1 in dictionary) # False
print(2 in dictionary) # False
```

Bei Dictionaries sieht das Ganze etwas anders aus. Wenn wir hier das Schlüsselwort *in* benutzen, so bekommen wir nur dann *True* als Wert zurück, wenn das gesuchte Objekt als Key enthalten ist. Values werden nicht berücksichtigt.

8 – FUNKTIONEN

Funktionen sind Blöcke, welche organisierten, wiederverwendbaren Code beinhalten. Durch sie werden Anwendungen modularer und Sie erhalten einen hohen Grad an Wiederverwendung von Code.

FUNKTIONEN DEFINIEREN

Um eine Funktion zu definieren benötigen wir das Schlüsselwort *def*, gefolgt von runden Klammern, welche die sogenannten Parameter beinhalten oder leer sind. Nach dieser Klammer kommt wie bei Bedingungen und Schleifen ein Doppelpunkt. Alles was danach eingerückt, darunter steht gehört zu der Funktion.

```
def helloWorld():
    print("Hello World!")
```

In dem oberen Codebeispiel ist der Name der parameterlosen Funktion *helloWorld*. Die Funktion gibt bei jedem Aufruf den Text „Hello World!“ auf die Konsole aus.

```
def addieren(x, y):
    return x + y
```

Hier haben wir eine Funktion mit einem Rückgabewert, welcher nach dem Schlüsselwort *return* steht. Dieser wird bei Funktionsaufruf zurückgegeben und kann zum Beispiel ausgegeben oder in eine Variable gespeichert werden. Außerdem sind hier in der Klammer zwei Parameter, welche bei Funktionsaufruf eingegeben werden müssen.

FUNKTIONEN AUFRUFEN

Um eine aufzurufen muss man nur ihren Namen, gefolgt von Klammern (ggf. inkl. Parameter), angeben.

```
helloWorld() # Gibt "Hello World!" aus
print(addieren(5, 10)) # Gibt 15 aus
```

Die Funktion *addieren* sorgt für keine Ausgabe auf die Konsole, sie liefert lediglich einen Rückgabewert, welcher durch *print* ausgegeben werden muss.

STANDARDPARAMETER

Wenn Sie für den Aufruf Ihrer Funktion nicht immer alle Parameter extra angeben möchte, so können Sie einige davon mit Standardwerten versehen.

```
def person(name, alter = 21, groesse = 1.80):
    print(name, alter, groesse)
```

Beim Aufruf dieser Funktion sind die Parameter Alter und Größe nun optional. Wenn nur der Name angegeben wird, übernehmen diese die Standardwerte. Dennoch können sie manuell eingegeben werden.

VARIABLE PARAMETERANZAHL

Wenn Sie bei der Definition einer Funktion noch nicht wissen wie viele Parameter übergeben werden bzw. es mehrere Möglichkeiten gibt, können Sie auf Tupel als Parameter ausweichen.

```
def summe(*zahlen):
    summe = 0
    for i in zahlen:
        summe += i
    return summe

print(summe(1, 2, 3))
```

Hier sehen Sie ein Beispiel, wie das Ganze angewandt werden kann. Die Ausgabe dieses Skriptes ist die Zahl Sechs. Die Variable **zahlen* ist ein unveränderbares Tupel mit beliebig vielen Werten.

GÜLTIGKEITSBEREICHE

Es gibt, grob unterteilt, zwei Arten von Variablen. Die lokalen Variablen, welche nur in einem bestimmten Bereich gelten, und die globalen Variablen die überall gelten.

```
def funktion():
    variable = 10
    print(variable)

funktion()
print(variable) # Das wird nicht funktionieren
```

In diesem Fall zum Beispiel wird die Variable in der Funktion definiert und ist somit außerhalb nicht aufrufbar, weil sie nur eine lokale Gültigkeit hat. Sollte die Variable jedoch global, außerhalb von Funktionen, Klassen und Co. definiert werden, so kann Sie überall aufgerufen werden.

9 – STRINGFUNKTIONEN

Auch wenn Strings nur Texte sind, kann man auf diese eine sehr große Anzahl an Funktionen anwenden. Da sich dieses Buch an Anfänger richtet, werden wir uns nicht alle Stringfunktionen ansehen. Es ist jedoch sehr wichtig, dass Sie wissen, was Sie alles mit Strings machen können.

STRINGS ALS SEQUENZEN

Im Prinzip können Strings als Sequenzen von Buchstaben angesehen werden. Das heißt, dass das Indexieren und Iterieren von bzw. über Strings möglich ist. Außerdem können auf Strings miteinander addiert oder mit Zahlen multipliziert werden.

```
text = "HelloWorld"
print(text[1:5]) # Gibt "ello" aus
print(text + "Today") # Gibt "HelloWorldToday" aus

for s in text:
    print(s)
```

Hier sehen Sie, dass ein String wie eine Liste behandelt werden kann.

SPEZIELLE ZEICHEN

Einige Zeichen wie Zeilenumbrüche, Leerzeichen oder Backspace, können oder müssen in Python mit Backslash Notationen angegeben werden. Sie sind sehr zahlreich, aber wir beschäftigen uns fürs erste einmal nur mit den beiden wichtigsten.

```
print("Das ist ein\nZeilenumbruch") # \n = Zeilenumbruch
print("Das ist ein\tTab") # \t = Tab
```

Zeilenumbrüche werden mit \n gemacht und sind sehr wichtig um zum Beispiel zeilenweise aus Dateien zu lesen oder in diese zu schreiben. Tabs werden mit \t gemacht und haben eine Standardweite von vier Zeichen.

STRINGFORMATIERUNG

Das Formatieren von Strings ermöglicht den Gebrauch von sogenannten Platzhaltern innerhalb eines Strings. Es sind schlichtweg Werte welche erst im Nachhinein in den String geladen werden. Diese Werte werden immer mit einem Prozentzeichen eingeleitet.

```
print("Hello %s" % "World") # Gibt "Hello World" aus
```

In diesem simplen Beispiel wird der Text „World“ erst im Nachhinein in den Platzhalter *%s* hineingeladen. Statt einem statischen String („World“) kann auch eine Variable welche einen Text enthält angegeben werden. Nachdem Haupttext, welcher die Platzhalter enthält, folgt ein Prozentzeichen, welches die Werte einleitet. Es kann nicht für jede Art von Wert dieselbe Art von Platzhalter verwendet werden. In diesem Beispiel benutzen wir *%s* welches ausschließlich für Strings funktioniert. Für Zahlen müssen wir zum Beispiel einen anderen Platzhalter verwenden.

Platzhalter für Stringformatierung	
Platzhalter	Datentyp
%s	Text / Strings
%d oder %i	Ganzzahl / Integer
%c	Zeichen / Character
%f	Fließkommazahl / Float

TRIPLE QUOTES

Triple Quotes bedeutet im Englischen dreifache Anführungszeichen. Diese werden in Python benutzt um über mehrere Zeilen hinweg Texte zu schreiben. Dabei werden Zeilenumbrüche auch ohne \n gemacht, also einfach durch einen ordinären Zeilenumbruch im String.

```
text = """Hallo Welt! Das ist ein Text,
    das ist eine neue Zeile welche eingerückt ist

und das war eine leere Zeile."""
```

Geben wir diese Variable nun aus sind alle Leerzeichen, Tabs und Zeilenumbrüche im String enthalten.

STRINGFUNKTIONEN

Wie bereits anfangs erwähnt gibt es Unmengen an Stringfunktionen, welche wir in diesem Buch nicht alle abdecken werden. Wir werden uns die anschauen welche für Anfänger interessant sind und sollten Sie Interesse an weiteren Stringfunktionen haben, so können Sie diese sicher durch eine Suchmaschine ihrer Wahl finden.

LISTENFUNKTIONEN

Wie wir bereits wissen können Strings als Sequenzen angesehen werden. Deshalb können auch die grundlegenden Listenfunktionen wie *len*, *max* und *min* auf Strings angewandt werden.

```
len("HelloWorld") # Gibt 10 zurück
max("HelloWorld") # Gibt r zurück (ASCII-Code)
min("HelloWorld") # Gibt H zurück (ASCII-Code)
```

CASE-RELATED FUNKTIONEN

Unter Case-Related Funktionen ordnen wir in diesem Buch alle Funktionen ein, welche die Schrift dahingehend verändern, dass, gewisse Buchstaben, groß oder klein geschrieben werden. Diese Funktionen müssen an einen String mit einem Punkt angehängt werden.

Case-Related Funktionen	
Funktion	Beschreibung
capitalize()	Macht den Anfangsbuchstaben eines Strings zu einem Großbuchstaben.
upper()	Macht alle Buchstaben eines Strings zu Großbuchstaben.
lower()	Macht alle Buchstaben eines Strings zu Kleinbuchstaben.
swapcase()	Invertiert die Groß- und Kleinschreibung aller Buchstaben eines Strings.
title()	Konvertiert einen String in das Titelformat.

```
print("Hello World".swapcase()) # hELLO wORLD
```

```
print("Hello World".upper()) # HELLO WORLD
print("hello world".title()) # Hello World
```

REPLACE FUNKTION

Die Funktion *replace* ersetzt die Vorkommnisse eines bestimmten Strings, in einem bestimmten String, durch einen bestimmten String.

```
text = "Wie geht es Ihnen? Geht es Ihnen gut?"
text.replace("Ihnen", "dir")
```

In dem oberen Beispiel benutzen wir die Funktion, um das Wort „Ihnen“ bei jedem Vorkommen, durch das Wort „dir“ zu ersetzen.

FIND FUNKTION

Die Funktion *find* wird benutzt, um einen bestimmten String, in einem anderen bestimmten String zu finden.

```
alph = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(s.find("Q")) # Gibt 16 aus
print(s.find("A")) # Gibt 0 aus
```

In diesem Beispiel haben wir einen String *alph* welcher das Alphabet als Text enthält. Mit der Funktion finden wir heraus an welcher Stelle die jeweiligen Buchstaben sind. Achtung: Das Zählen beginnt bei Sequenzen immer mit Null.

COUNT FUNKTION

Die Funktion *count* wird benutzt, um herauszufinden wie oft ein bestimmter String, in einem anderen bestimmten String vorkommt.

```
text = "Hello World! Das ist ein Text!"
print(text.count("!")) # Gibt 2 aus
print(text.count("i")) # Gibt 2 aus
```

In diesem Beispiel finden wir heraus wie oft die Zeichen „!“ und „i“ vorkommen. Man kann jedoch auch ganze Wörter zählen bzw. suchen.

10 – MODULE

Module sind im Grunde genommen Python Dateien, welche, in andere Python Dateien, importiert werden. Module selbst werden nicht ausgeführt, sondern stellen zum Beispiel Funktionen und Klassen zur Verfügung. Sie werden benutzt um Anwendungen modularer, also übersichtlicher und somit leichter verständlich, zu machen. Das hat unter anderem auch den Vorteil, dass es leichter ist einzelne Segmente auszutauschen.

DAS IMPORT STATEMENT

Zum Importieren von Modulen in Python wird das sogenannte *import* Keyword benutzt. Module können, wie bereits erwähnt, selbst geschriebene Python Skripts oder von Python bereits zur Verfügung gestellte Module sein.

Der folgende Code befindet sich in der Datei **modul1.py**:

```
def hallo():
    print("Hallo!")

def bye():
    print("Bye!")
```

Der folgende Code befindet sich in der Datei **main.py:**

```
import modul1

modul1.hallo()
modul1.bye()
```

Da beide Dateien im selben Verzeichnis sind, kann die eine Datei in die andere importiert werden. Die Funktionen können dann, über den Verweis auf das Modul, benutzt werden.

DAS FROM STATEMENT

Das *from* Keyword wird in Kombination mit dem *import* Keyword benutzt um bestimmte Dinge aus einem Modul zu importieren, sodass kein Verweis auf dieses mehr notwendig ist.

```
from modul1 import hallo

hallo()
modul1.bye() # Wird nicht funktionieren
```

In diesem Beispiel importieren wir aus modul1 ausschließlich die Funktion *hallo* und müssen bzw. können, somit gar nicht auf das Modul verweisen. Die Funktion *bye* oder das Modul selbst werden nicht erkannt.

ALLES AUS EINEM MODUL IMPORTIEREN

Um alle Elemente aus einem Modul zu importieren, sodass anschließend nicht mehr darauf verwiesen werden muss, wird das Asterisk Symbol benötigt.

```
from modul1 import *

hallo()
bye()
```

Mit dieser Methode importieren Sie alle Elemente aus einem Modul. Dies können Sie auch auf die Module anwenden, die bereits von Python angeboten werden, was uns gleich zu unserem nächsten Kapitel bringt, dem Modul der Mathematik.

11 – MATHEMATIK

Das Modul der Mathematik ist nur eines der zahlreichen Module in Python. Wir schauen uns dieses Modul etwas genauer an, da es gerade für Informatiker, wozu Programmierer nun einmal gehören, wichtig ist Mathematik zu verstehen. Keine Sorge wir werden nicht allzu tief ins Detail gehen.

```
from math import *
```

Als erstes, sollten Sie, wie in dem Codebeispiel oben, alle Ressourcen aus dem Modul *math* importieren.

MATHEMATISCHE FUNKTIONEN

In der Mathematik gibt es einige Funktionen, welche man hin und wieder, in seiner Anwendung, benötigt um ein Problem in der Realität zu lösen. Wurzelziehen, Winkelfunktionen und vieles mehr kann hierbei von Bedeutung sein. Im Folgenden bekommen Sie einen Überblick über die grundlegendsten mathematischen Funktionen in Python.

WURZELZIEHEN (SQUAREROOT)

Die Funktion *sqrt* bedeutet abgekürzt Squareroot, was, zu Deutsch, Quadratwurzel bedeutet. Mit dieser Funktion lässt sich die Wurzel einer Zahl ermitteln.

```
print(sqrt(64)) # Gibt die Wurzel von 64 (8) aus
print(sqrt(2)) # Gibt die Wurzel von 2 (1.4142…) aus
```

TRIGONOMETRIE

In Python lassen sich die Winkelfunktionen, ganz simpel, durch die Funktionen *sin*, *cos* und *tan* benutzen. Außerdem gibt es auch die inversen Funktionen *asin*, *acos* und *atan*, welche hier jedoch nicht weiter erläutert werden.

```
print(sin(35)) # Berechnet den Sinus von 35
print(cos(35)) # Berechnet den Cosinus von 35
print(tan(35)) # Berechnet den Tangens von 35
```

Des Weiteren ist es möglich, das Resultat in Degree oder Radiant umzuwandeln, wie im folgenden Codebeispiel:

```
print(degrees(2)) # Wandelt 2 (Radiant) in Degrees um
print(radians(150)) # Wandelt 150 (Degrees) in Radiant um
```

Passend zu diesen beiden Funktionen, stellt das Modul auch noch die Konstante *pi* zur Verfügung.

```
print(pi) # Gibt 3.14159... aus
```

FAKULTÄT

Es ist auch möglich in Python die Fakultätsfunktion zu verwenden um das Faktorielle einer Zahl zu ermitteln wie im folgenden Beispiel:

```
print(factorial(5)) # Gibt 5! (120) aus
```

Diese Funktion ist auch oftmals das Paradebeispiel für rekursive Programmierung, auf welche wir vielleicht, in einem Buch für Fortgeschrittene, zu sprechen kommen werden.

LOGARITHMUSFUNKTIONEN

Um den Logarithmus einer Zahl, zu einer bestimmten Basis, zu ermitteln, gibt es in Python mehrere Funktionen. Handelt es sich dabei um die Basis Zehn, so gibt es die Funktion *log10*. Zusätzlich gibt es noch die Funktion *log* welche standardmäßig die Basis *e* benutzt, jedoch auch als zweiten Parameter eine beliebige Basis akzeptiert.

```
print(log10(10000)) # Logarithmus von 10000 Basis 10 (4)
print(log(10000)) # Logarithmus von 10000 Basis e
print(log(10000, 2)) # Logarithmus von 10000 Basis 2
```

Passend zu diesen Funktionen stellt die Bibliothek auch die Konstante *e* zur Verfügung.

```
print(e) # Gibt 2.71821... aus
```

Außerdem gibt es Funktionen, zum Potenzieren von Zahlen und das Modul bietet auch eine Funktion, zum Potenzieren von der Variable *e*.

```
print(pow(5, 2)) # Gibt 5 hoch 2 (25) aus
print(exp(5)) # Gibt e hoch 5 aus
```

So viel zur Mathematik. Vor allem im Bereich Scientific Computing, spielt Mathematik, in Programmiersprachen, eine große Rolle. Nun verstehen Sie die Grundlagen dieser.

12 – DATEIOPERATIONEN

In diesem Kapitel werden wir uns, in erster Linie, mit Streams auseinandersetzen. Das Kapitel heißt Dateioperationen, weil wir uns, mal abgesehen von ein paar wenigen Ausnahmen am Anfang, hauptsächlich mit den Streams von Dateien beschäftigen werden.

BENUTZEREINGABEN

Benutzereingaben in Python funktionieren mit der sogenannten *input* Funktion. Als Parameter übergibt man der Funktion den Text, welchen man gerne auf die Konsole ausgegeben haben möchte. In der Regel sagt dieser Text dem Bentuzer, welche Art von Eingabe von ihm erwartet wird. Die Eingabe des Benutzers wird mit *Enter* beendet und ist der Rückgabewert der Funktion. Dieser kann dann anschließend in eine Variable gespeichert werden. Wichtig: Auch Zahlen werden nur im Stringformat gespeichert, was bedeutet, dass diese konvertiert werden müssen, sollte man mit ihnen Rechenoperationen durchführen wollen.

```
eingabe = input("Bitte um eine Eingabe: ")
```

DATEIEN ÖFFNEN UND SCHLIESSEN

Bevor Sie in eine Datei schreiben oder aus dieser lesen können, müssen sie das File erst einmal öffnen. Für diesen Zweck gibt es die Funktion *open*. Diese liefert ein File als Objekt zurück, welches in eine Variable gespeichert werden kann.

```
datei = open("datei.txt", "r")
```

In diesem Fall öffnen wir die Datei *datei.txt* im Lesemodus.

Die Funktion *open* übernimmt bis zu drei Parametern, wovon nur der erste unbedingt benötigt wird. Der erste Parameter gibt an welche Datei geöffnet werden soll, der zweite gibt an in welchem Modus die Datei geöffnet werden soll (Lesen, Schreiben, Anhängen…) und der dritte gibt an ob Line-Buffering durchgeführt werden soll, was für uns derzeit weniger interessant ist.

ACCESS MODES

Access Modes sind die verschiedenen Modi, in welchen eine Datei geöffnet werden kann. Hier ein kurzer Überblick über die Modi, welche für uns als Anfänger interessant sind:

Zugriffsmodi	
Buchstabe	Modus
r	Lesemodus
w	Schreibmodus
a	Anhängemodus
r+	Lese- und Schreibmodus
w+	Lese- und Schreibmodus
a+	Lese- und Anhängemodus

Der Unterschied zwischen r+ und w+ ist, dass bei w+ eine neue Datei erstellt wird, sollte es keine Datei mit dem angegebenen Pfad geben. Bei r+ wird eine bereits bestehende Datei zum Lesen und Schreiben geöffnet.

DATEIEN SCHLIESSEN

Um ein geöffnetes File wieder zu schließen, zum Beispiel, weil dieser nicht mehr weiter benutzt wird, gibt es die *close* Funktion.

```
datei = open("datei.txt", "r")
datei.close()
```

ATTRIBUTE VON DATEIEN

Sobald Sie ein File geöffnet haben, können Sie, mit weiteren Funktionen, einige Attribute auslesen. Folgende Attribute sind für uns interessant:

```
print(datei.name) # Gibt den Dateinamen aus
print(datei.mode) # Gibt den Access Mode aus
print(datei.closed) # Gibt aus ob ein File geschlossen wurde
```

AUS DATEIEN LESEN

Um aus Dateien zu lesen, müssen diese, in einem Lesemodus, geöffnet werden. Auch wichtig zu beachten ist, dass, sobald eine Datei in eine Schreibmodus geöffnet wird, diese überschrieben wird, sollte eine Datei mit demselben Namen existieren.

```
datei = open("datei.txt", "r")
print(datei.read()) # Liest den ganzen Inhalt aus
print(datei.read(10)) # Liest die ersten 10 Zeichen aus
```

Der Funktion *read* kann, muss aber kein, Parameter übergeben werden. Der optionale Parameter gibt an wie viele Zeichen gelesen werden sollen.

IN DATEIEN SCHREIBEN

Um in Dateien zu schreiben, müssen diese, in einem Schreib- oder Anhängemodus geöffnet werden. Der Unterschied zwischen diesen ist, dass beim Schreibmodus, die Dateien komplett überschrieben werden und beim Anhängemodus nur mit Text ergänzt werden. Die Funktion, welche zum Schreiben benutzt wird, nennt sich *write*.

Im folgenden Beispiel erstellen wir eine neue Datei mit dem Namen *neueDatei.txt*, indem wir diese im Schreibmodus öffnen.

```
datei = open("neueDatei.txt", "w")
datei.write("Dieser Text kommt in die Datei!")
datei.close()
```

Im folgenden Beispiel hängen wir einen Text an eine bereits existierende Datei an, indem wie diese im Anhängemodus öffnen.

```
datei = open("datei.txt", "a")
datei.write("Dieser Text wird angehängt!")
datei.close()
```

Es ist sehr wichtig zu beachten, dass ein Text erst dann endgültig in eine Datei geschrieben wird, nachdem dieser in die Datei geflusht wurde. Dafür gibt es die Funktion *flush*, wobei es auch reicht, die Datei zu schließen, da dann jeglicher Text automatisch geflusht wird.

```
datei.flush()
```

WEITERE DATEIOPERATIONEN

Es gibt noch einige Dateioperationen, für welche jedoch System Call notwendig sind. System Calls rufen vom Betriebssystem bereitgestellte Funktionalitäten auf. Wir werden diese System Calls in Python benutzen um Ordner zu erstellen, Dateien umzubenennen und mehr. Um diese nutzen zu können müssen wir das Modul *os* importieren.

DATEIEN LÖSCHEN UND UMBENENNEN

Die beiden Funktionen *remove* und *rename* erlauben und das Löschen und das Umbenennen von Dateien. Um diese nutzen zu können, importieren wir erst einmal alle Elemente aus dem Modul *os*.

```
from os import *
rename("datei1.txt", "datei2.txt") # Benennt eine Datei um
remove("datei.txt") # Entfernt eine Datei
```

VERZEICHNISOPERATIONEN

Durch das *os* Modul, ist es uns auch möglich Verzeichnisse zu erstellen und zu löschen und durch diese zu navigieren.

```
mkdir("Ordner") # Erstellt ein neues Verzeichnis
chdir("C:\\Program Files") # Ändert aktuelles Verzeichnis
rmdir("Ordner") # Löscht Verzeichnis
```

Da ein Backslash, wie wir bei den Stringfunktionen bereits gelernt haben, für spezielle Zeichen verwendet werden, müssen, bei Pfadangaben, immer zwei Backslashs hintereinander gemacht werden, sodass es als eines erkannt wird.

13 – EXCEPTIONS

Wenn Sie, während sie dieses Buch bis zu diesem Punkt gelesen haben, nebenbei selber mitprogrammiert haben, so ist ihnen wahrscheinlich hin und wieder eine Exception erscheinen. Exception bedeutet übersetzt Ausnahme und kommt dann vor wenn das Programm aus irgendeinem Grund einen Fehler hat. Ein einfaches Beispiel ist der Versuch, eine Zahl durch Null zu dividieren. Wenn sie das versuchen bekommen Sie eine Exception. Diese sieht dann so aus:

```
>>> print(200/0)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print(200/0)
ZeroDivisionError: division by zero
>>>
```

In diesem Fall handelt es sich um einen *ZeroDivisionError*, da hier versucht wurde durch Null zu dividieren, was bekanntlich mathematisch nicht möglich ist.

ASSERTIONS

Assertion bedeutet übersetzt Behauptung. Behauptungen erlaubten uns in Python Exceptions zu werfen, wenn eine Bedingung nicht erfüllt wurde. Dazu gibt es die Funktion *assert.*

```
zahl = 10
assert(zahl == 10) # Keine Exception
assert(zahl == 20) # Wirft Exception
```

Der Ausdruck in Zeile Drei verursacht einen AssertionError.

EXCEPTION HANDLING

Sobald eine Exception geworfen und nicht vorher behandelt wird, ist das Programm zu Ende, es stürzt ab. Damit so etwas nicht passiert gibt es das sogenannte Exception Handling, welches sich darum dreht, wie man mit Exceptions umgeht, nachdem man diese abgefangen hat. Hierbei sind zwei Schlüsselwörter entscheidend, nämlich *try* und *except*.

```
try:
    print(10/0)
except ZeroDivisionError:
    print("Division durch 0 nicht möglich!")
```

In dem oberen Beispiel versuchen wir wieder eine Division durch Null durchzuführen, fangen dieses Mal jedoch die Exception ab, sodass das Programm weiterläuft. Das Schlüsselwort *try* initiiert einen Versuch. Sobald in diesem Bereich eine Exception, von der Art ZeroDivisionError, geworfen wird, begibt sich das Programm in den *except* Zweig. Sollte eine andere Art von Exception geworfen werden, so wird diese in diesem Beispiel nicht behandelt.

```
try:
    print(10/0)
    datei = open("datei.txt", "r")
except ZeroDivisionError:
    print("Division durch 0 nicht möglich!")
except IOError:
    print("Die Datei existiert nicht!")
except:
    print("Hier ist etwas anderes los!")
```

Es ist möglich mehrere *except* Zweige aneinander zu reihen. In dem Zweig selber stehen im Allgemeinen die Aktionen, welche durgeführt werden, sobald diese Art von Exception geworfen wurde. Außerdem muss nicht zwangsläufig eine bestimmte Art von Exception spezifiziert werden. Es kann genauso gut ein Zweig für alle Arten von Exceptions angehängt werden.

ELSE-STATEMENTS BEI EXCEPTIONS

Wie bei If-Verzweigungen, kann auch bei Try-Statements ein *else* angehängt werden. Der Code in diesem Else-Zweig wird dann ausgeführt, wenn keine Exception geworfen wurde.

```
try:
    print(10/0)
except ZeroDivisionError:
    print("Division durch 0 nicht möglich!")
else:
    print("Keine Exception!")
```

FINALLY-STATEMENTS

Der Code der nach dem Schlüsselwort *finally* steht, wird auf jeden Fall ausgeführt unabhängig von Exceptions.

```
try:
    print(10/0)
except ZeroDivisionError:
    print("Division durch 0 nicht möglich!")
finally:
    print("Das wird immer ausgeführt!")
```

14 – ANWENDUNGSBEISPIELE

Da Sie nun die Grundlagen der Programmierung mit Python beherrschen, ist es Zeit für ein paar Beispiele bzw. Aufgaben, welche Sie mit dem gelernten Wissen lösen können sollten.

ZEICHEN IN DOKUMENTEN ZÄHLEN

Angabe: Schreiben Sie ein Skript, welches die Vorkommnisse eines bestimmten Strings, in einem Dokument, zählt.

Lösung:

```
datei = open("datei.txt", "r")
inhalt = datei.read()

counter = 0
for s in inhalt:
    if(s == "H"):
        counter += 1

# Auch inhalt.count(„H“) wäre möglich
print(counter)
```

SORTIEREN EINER ZAHLENLISTE

Angabe: Schreiben Sie ein Skript, welches eine Liste, welche ausschließlich mit Zahlen gefüllt ist, der Größe nach sortiert. Sie dürfen dabei nicht die bereits vorhandene Sortierfunktion nutzen.

Tipp: Es gibt mehrere Sortieralgorithmen. Für Anfänger wird der BubbleSort empfohlen (Google).

Lösung:

```
liste = [10, 22, 12, 300, 44, 2, 19, 8]

sortiert = False

while not sortiert:
    sortiert = True
    for element in range(0, len(liste)-1):
        if(liste[element] > liste[element + 1]):
            sortiert = False
            halter = liste[element + 1]
            liste[element + 1] = liste[element]
            liste[element] = halter

print(liste)
```

TEXT UMSCHREIBEN

Angabe: Schreiben Sie ein Skript, welches folgenden Text formell macht:

„Hallo, wie geht es dir? Ich habe gestern deinen Bruder getroffen. Ich würde dich auch gerne Mal treffen.“

Lösung:

```
text = "Hallo, wie geht es dir? Ich habe gestern deinen Bruder getroffen. Ich würde dich auch gerne Mal treffen."

text = text.replace("dir", "Ihnen")
text = text.replace("deinen", "Ihren")
text = text.replace("dich", "Sie")
print(text)
```

TASCHENRECHNER

Angabe: Implementieren Sie einen Rechner, bei der der Anwender selbst, in der Konsole, die Zahlen eingeben kann.

Lösung:

```
op = input("Was wollen Sie tun? (+,-,*,/)")
z1 = int(input("Geben Sie die erste Zahl ein: "))
z2 = int(input("Geben Sie die zweite Zahl ein: "))

if(op == "+"):
    print(z1 + z2)
elif(op == "-"):
    print(z1 - z2)
elif(op == "*"):
    print(z1 * z2)
elif(op == "/"):
    print(z1 / z2)
else:
```

```
PRINT("UNGÜLTIGE EINGABE!")
```

PYTHON PROGRAMMIEREN

FÜR FORTGESCHRITTENE

FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

Python gewinnt in den letzten Jahren massiv an Popularität und wird wirtschaftlich immer relevanter. Die Programmiersprache findet ihre Anwendung in verschiedensten Bereichen, wie Web Development, Data Science, Neuronalen Netzen, Machine Learning, Robotik und vielen mehr. Alles Bereiche, welche unsere Zukunft maßgeblich prägen werden. Es sind bereits jetzt zahlreiche Errungenschaften und Fortschritte gelungen und es werden mit Sicherheit noch sehr viele folgen. Wenn Sie ein Teil dieser Entwicklung sein wollen, ist es definitiv eine gute Idee sich intensiver mit Python zu befassen. Die Sprache ist sehr simpel zu erlernen und hat eine einfache Syntax. Mittlerweile gehört sie zu den bedeutsamsten Sprachen unserer Zeit und die Chancen stehen außerordentlich gut, dass sie noch relevanter wird.

Wenn Sie dieses Buch lesen, haben Sie wahrscheinlich schon bereits Erfahrungen mit Python gemacht. Sollte dem nicht so sein, empfehle ich Ihnen mein Buch für Anfänger.

Dieses Buch setzt grundlegende Python-Kenntnisse voraus. Sie müssen definitiv kein Profi sein, aber die Themen, welche im ersten Buch abgehandelt wurden, sollten sitzen. Nachdem Sie dieses Buch gelesen haben, werden Sie in der Lage sein, selbstständig, fortgeschrittene Applikationen in Python zu schreiben. Sie werden erweiterte Konzepte dieser Sprache verstehen und ein solides Fundament für die Weiterbildung in noch interessantere Bereiche wie Künstliche Intelligenz oder Data Science haben.

Sie sollten dieses Buch jedoch nicht nur schnell durchlesen und versuchen oberflächlich zu verstehen, worum es geht, sondern auch aktiv nebenbei mitprogrammieren. Dadurch lernen Sie am meisten und das Wissen wird am besten gefestigt. In den folgenden Kapiteln werden Sie zahlreiche Codebeispiele finden, die Sie nachprogrammieren können. Sie können jedoch auch Ihren eigenen Code schreiben, mithilfe der Dinge die Sie bereits gelernt haben, zu dem Zeitpunkt.

Im Grunde genommen bleibt es Ihnen selbst überlassen, wie Sie diese Lektüre handhaben möchten. In jedem Fall wünsche ich Ihnen maximalen Erfolg beim Programmieren und viel Spaß beim Lernen von Python.

1 – PAKETVERWALTUNG MIT PIP

Wenn Sie mit Python programmieren und Module benutzen möchten, welche nicht bereits in Python integriert sind, so können Sie die Pakete über eine sogenannte Paketverwaltung installieren. Eines dieser Programme lautet *pip*, dessen Name ein rekursives Akronym ist und *„pip installs packages“*, also *„pip installiert Pakete“*, bedeutet. Dieser Paketmanager wird bei der Installation von Python bereits mitgeliefert.

Um pip nun zu benutzen benötigen Sie eine Kommandozeile. Unter Linux und Mac OS X wäre das das Terminal und unter Windows können Sie zwischen CMD und PowerShell wählen, oder Sie installieren die Git Bash, welche Sie hier herunterladen können.

Sie können dort nun folgenden Befehl eingeben um die Version von Ihrem pip zu überprüfen:

```
pip --version
```

PAKETE INSTALLIEREN

Um nun Pakete bzw. Module zu installieren, benötigen wir zunächst den Namen, des zu installierenden Moduls. Als Beispiel nehmen wir hier das beliebte Modul *numpy*, welches eine effiziente Verarbeitung von großen Listen ermöglicht und sich somit super für die Arbeit mit Big Data eignet. Wir geben also nun folgenden Befehl in unsere Kommandozeile ein, um *numpy* zu installieren.

pip install numpy

Das Modul können Sie beliebig austauschen. Es gibt unzählig viele nützliche und interessante Python Module, welche Sie auf diese Weise installieren können

Mit folgendem Befehl können Sie bereits installierte Pakete deinstallieren:

pip uninstall numpy

Wollen Sie nun alle bereits installieren Pakete einsehen, so geben Sie folgenden Befehl in die Kommandozeile ein:

pip list

Sie können jedoch mit folgendem Command auch Informationen über einzelne Module abrufen:

pip status <modulname>

Wenn Sie nicht wissen, welche Module es gibt und das Repository durchsuchen möchten, können Sie das wie folgt tun:

pip search <suchtext>

2 – KLASSEN UND OBJEKTE

Python ist, seit es existiert, eine objektorientierte Programmiersprache, was bedeutet, dass der Code in einzelne Einheiten, nämlich in Objekte, unterteilt werden kann. Jedes dieser Objekte gehört zu einer Klasse. Diese kann man sich wie den Bauplan für die Instanz vorstellen. Beispielsweise wäre der Plan für ein Auto die Klasse und jedes einzelne physische Auto ist dann ein Objekt bzw. eine Instanz. Eine Klasse hat also bestimmte Eigenschaften, wobei deren Werte von Objekt zu Objekt variieren.

KLASSEN ERSTELLEN

Um in Python eine Klasse zu erstellen benötigen wir das Schlüsselwort *class*, gefolgt von einem Klassennamen.

```
class Auto:
    'Dokumentationstext der Klasse'
    anzahlAutos = 0

    def __init__(self, hersteller, modell, ps):
        self.hersteller = hersteller
        self.modell = modell
        self.ps = ps
        Auto.anzahlAutos += 1

    def printAnzahl(self):
        print(Auto.anzahlAutos)

    def printInfo(self):
        print("Hersteller: %s, Modell: %s, PS: %s" % (self.hersteller, self.modell, self.ps))
```

Hier ist einiges neu, wie Sie sehen können. In dem oberen Beispiel erstellen wir eine Klasse, also einen Bauplan, für Autos. Direkt darunter können wir einen Dokumentationskommentar machen, welcher die Klasse beschreibt. Als nächstes wurde eine Variable definiert, welche für alle Autoobjekte global gilt. Das bedeutet: Wenn Sie zehn Autos erzeugen, gibt es immer noch nur diese eine Variable, welche sich diese Autos alle teilen. Ihr Wert wird entsprechend zehn sein.

Als nächstes haben wir hier drei Methoden definiert, wovon die __init__ Methode, die einzige ist, die wir wirklich benötigen, um das Objekt zu erzeugen. In dieser finden wir vier Parameter. Drei davon sind Eigenschaften, welche das Auto haben kann: Einen Hersteller, ein Modell und die Pferdestärke. Der erste Parameter *self*, verweist auf das jeweilige Objekt. Das bedeutet, dass wir es, jedes Mal wenn wir *self* aufrufen, mit den Attributen des aktuellen Objektes zu tun haben. Im Gegensatz zur Variable *anzahlAutos*, sind die anderen von Objekt zu Objekt unterschiedlich. Bei jeder Funktion in einer Klasse, muss als Parameter *self* übergeben werden.

Sie können natürlich so viele Attribute (egal ob geteilt oder private) und so

viele Funktionen pro Klasse definieren, wie Sie wollen.

OBJEKTE ERSTELLEN

Um eine Instanz einer Klasse, also ein Objekt, zu erstellen, müssen wir den Konstruktor der Klasse Auto, also die __init__ Methode, aufrufen und dieser die Werte für die Attribute übergeben. Das tun Sie wie folgt:

```
auto1 = Auto("Volkswagen", "Polo", 60)
auto2 = Auto("Mercedes", "Benz", 500)
```

Nach dem Ausführen des oberen Beispiels, haben wir zwei Instanzen der Klasse Auto, mit verschiedenen Herstellern, Modellen und Pferdestärken. Dennoch haben Sie dieselbe Autoanzahl, nämlich zwei.

Beim Erstellen eines Objektes, müssen alle Parameter, welche keine Standardwerte haben, übergeben werden.

Um jetzt die Funktionen der beiden Objekte aufzurufen, verweisen wir auf unsere Instanzen.

```
auto1.printInfo()
```

Sie können natürlich auch auf einzelne Attribute zugreifen.

```
print(auto1.modell)
```

In Python ist es möglich Objekten komplett neue Attribute hinzuzufügen, welche in der Klasse gar nicht definiert sind. Sie können jedoch auch bereits vorhandene Attribute mit dem Schlüsselwort *del* löschen.

```
auto1.neuesAttribut = "Test"
print(auto1.neuesAttribut)
del auto1.modell
# auto1.printInfo() -> Exception
```

VERERBUNG

Ein wichtiges Prinzip in der objektorientierten Programmierung ist die Vererbung. Mithilfe dieses Konzepts, können Sie bereits vorhandene Klassen nutzen, um auf diesen neue aufzubauen. Das funktioniert natürlich nur, wenn die neue Klasse eine Erweiterung der alten darstellt. Ein gutes Beispiel für eine Elternklasse wäre der Mensch, wobei die erbende Kinderklasse hierbei ein Sportler, ein Tänzer oder ein Programmierer sein könnte. Alle haben die Eigenschaften eines Menschen, doch manche haben auch für ihre Klasse spezifische Attribute.

```
class Mensch:

    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def altern(self, jahre):
        self.alter += jahre

class Programmierer(Mensch):

    def __init__(self, name, alter, lieblingssprache):
        super(Programmierer, self).__init__(name, alter)
        self.lieblingssprache = lieblingssprache

    def printSprache(self):
        print(self.lieblingssprache)

p1 = Programmierer("Max", 26, "Python")
p1.altern(2)
```

Wie Sie in dem oberen Beispiel sehen können, wird die Klasse, von welcher geerbt werden soll, bei der Definition der erbenden Klasse in die Klammern geschrieben. Die Klasse Programmierer hat nun automatisch alle Attribute und Funktionen der Klasse Mensch.

Um nun einen Programmierer erstellen zu können, müssen wir dennoch irgendwie die Parameter für Name und Alter dem Konstruktor übergeben. Um auf den Konstruktor der Elternklasse zuzugreifen, gibt es die Methode *super*. Dieser übergibt man als erstes die erbende Klasse und dann wieder

self. Damit greift man dann auf die Klasse zu, von welcher man erbt und kann deshalb auch auf ihre __init__ Methode zugreifen.

Wichtig ist auch noch zu erwähnen, dass Methoden beliebig überschrieben werden können. So könnten Sie zum Beispiel die Methode *altern* in der Klasse Programmierer neu definieren und eine andere Funktionalität implementieren.

OPERATOREN ÜBERLADEN

Auf den ersten Blick, macht es keinen Sinn, zwei Menschen zu addieren oder zu subtrahieren. Das liegt jedoch nur daran, dass wir nicht definiert haben was wir darunter verstehen. Wir könnten zum Beispiel definieren, dass wenn man zwei Menschen miteinander addiert, ein neuer Mensch zurückgegeben wird, welcher genauso alt ist wie beide zusammen. Was auch immer wir gerne für Aktionen hätten, wenn bestimmte Operatoren auf unsere Objekte angewandt werden, muss definiert werden, indem die Operatoren überladen werden. Das bedeutet, dass wir Funktionen schreiben, welche genau dann aufgerufen werden, wenn ein bestimmter Operator auf unser Objekt angewandt wird.

```
class Mensch:

    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def __add__(self, other):
        return Mensch("Kombi", self.alter + other.alter)

m1 = Mensch("Hans", 19)
m2 = Mensch("Peter", 30)
m3 = m1 + m2
print(m3.alter) # Gibt 49 aus
```

Hier auch noch ein etwas praktikableres und sinnvolleres Beispiel:

```
class Vektor():

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "X: %d, Y: %d" % (self.x, self.y)

    def __add__(self, other):
        return Vektor(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vektor(self.x - other.x, self.y - other.y)
```

```
v1 = Vektor(3, 5)
v2 = Vektor(6, 2)
v3 = v1 + v2
v4 = v1 - v2

print(v1)
print(v2)
print(v3)
print(v4)
```

Hierbei überladen wir die beiden arithmetischen Operatoren, um mit Vektoren arbeiten zu können. Die Methode __str__ beinhaltet den Programmcode, welcher ausgeführt wird, wenn das Objekt ausgegeben wird.

VERSTECKTE ATTRIBUTE

Sollten Sie aus irgendeinem Grund ein Attribut verstecken wollen, so müssen Sie bei der Definition, vor dem Namen, zwei Unterstriche einfügen.

```
class Mensch:

    __versteckt = "hidden"

    def __init__(self):
        self.__auchVersteckt = 10

    def test(self):
        print(self.__auchVersteckt) # Hier geht es
        print(Mensch.__versteckt) # Hier geht es

mensch = Mensch()
print(mensch.__auchversteckt) # Fehler
mensch.test() # Funktioniert
```

Wie Sie sehen können, sind die versteckten Variablen nur innerhalb der Klassendefinition aufrufbar. Wollen Sie dennoch auch von außerhalb darauf zugreifen, so können Sie das wie folgt:

```
print(mensch._Mensch__versteckt)
print(mensch._Mensch__auchVersteckt)
```

3 – MULTITHREADING

Threads sind leichtgewichtige Prozesse, welche bestimmte Aktionen in einem Programm ausführen, wobei sie selbst Teile eines Prozesses sind. Diese Threads können nun parallel arbeiten, so als würde man zwei Programme gleichzeitig laufen lassen. Beim sogenannten *Multithreading* haben Sie jedoch folgende Vorteile:

- Threads im selben Prozess teilen sich den Speicherbereich für die Daten und können daher viel einfacher Daten austauschen und miteinander kommunizieren.
- Threads brauchen weniger Ressourcen als Prozesse. Sie werden deshalb oft leichtgewichtige Prozesse genannt.

FUNKTIONSWEISE EINES THREADS

Ein Thread hat einen Anfang, eine Ausführsequenz und ein Ende und er kann unterbrochen oder angehalten werden. Letzteres bezeichnet man auch als Schlafenlegen, also *sleep*.

Es gibt zwei Arten von Threads:

- Kernel Threads -> Teil des Betriebssystems
- User Threads -> Vom Programmierer verwaltet

Wir befassen uns als Programmierer in erster Linie mit den User Threads.

Thread ist in erster Linie einmal eine Klasse, von welcher wir Instanzen erzeugen können. Jede dieser Instanzen stellt dann im Endeffekt einen eigenen Thread dar, welchen wir starten, anhalten, unterbrechen usw. können. Sie sind alle unabhängig voneinander und können gleichzeitig verschiedene Aktionen durchführen. Beispielsweise könnten Sie bei einem Videospiel einen Thread damit beauftragen die Grafiken zu laden und einen anderen damit die Tastatur- und Mauseingaben zu verarbeiten. In diesem Beispiel wäre es absolut undenkbar, die beiden Aktionen nacheinander, also seriell, auszuführen, wie wir es bisher immer getan haben.

STARTEN VON THREADS

Um überhaupt sinnvoll mit Threads zu arbeiten, sollten Sie folgende Module importieren:

```
import threading
import time
```

Das Modul *time* ist nicht für das Threading per se notwendig, wird uns jedoch bei anderen Dingen behilflich sein. Essentiell ist hierbei das *threading* Modul, welches das mittlerweile veraltete *thread* abgelöst hat.

```
import threading

def hallo():
   print("Hallo Welt!")

t1 = threading.Thread(target=hallo)
t1.start()
```

In diesem Beispiel definieren wir einen Thread, welcher als Zielfunktion *hallo()* hat. Die Klasse Thread hat zwei Funktionen zum Ausführen, nämlich *run()* und *start()*. Der Unterschied ist, dass wir bei der Funktion *run()* keinen separaten neuen Thread starten, sondern im aktuellen Thread eine Funktion gestartet wird, was bei *start()* jedoch anders aussieht.

Das bedeutet: Wenn Sie zwei Threads definieren und beide mit *run()* aufrufen, werden die Funktionen seriell, also nacheinander, ausgeführt. Wenn Sie hingegen beide mit der Funktion *start()* aufrufen, so starten Sie zwei separate und neue Threads, welche parallel, also gleichzeitig, ablaufen.

Folgendes Beispiel veranschaulicht dieses Prinzip gut:

```
import threading

def funktion1():
   for x in range(100):
      print("EINS")

def funktion2():
   for x in range(100):
      print("ZWEI")
```

```
t1 = threading.Thread(target=funktion1)
t1.start()
t2 = threading.Thread(target=funktion2)
t2.start()
```

Wenn Sie diesen Code ausführen, werden Sie im Ausgabefenster sehen, dass durcheinander gleichzeitig die Texte „Eins“ und „Zwei“ ausgegeben werden. Sollten Sie die Funktion *run()* verwenden, oder einfach beide Funktionen nacheinander aufrufen, werden Sie feststellen, dass zuerst 100 mal der Text „Eins“ und dann 100 mal der Text „Zwei“ ausgegeben wird. Mit Multithreading führen wir hier also beide Funktionen parallel aus.

Wichtig zu wissen ist, dass das Programm selbst natürlich auch ein Thread ist. Das heißt, wenn Sie einen Thread starten, läuft das Hauptprogramm weiter.

```
import threading
import time

def warten():
    print("Warte 5 Sekunden...")
    time.sleep(5)
    print("Fertig!")

t1 = threading.Thread(target=warten)
t1.start()
print("Hauptprogramm läuft weiter!")
```

In diesem Beispiel starten wir einen Thread, welcher eine Funktion aufruft, die durch die Methode *time.sleep()* den Thread für fünf Sekunden anhält. Durch die Ausgabe sehen wir jedoch, dass das Hauptprogramm trotzdem weiterläuft:

```
Warte 5 Sekunden...
Hauptprogramm läuft weiter!
Fertig!
```

THREADS ALS KLASSEN

Sie können bei der Programmierung mit Threads natürlich auch eigene Klassen erstellen, welche von der Hauptklasse *Thread* erben. Das gibt Ihnen dann die Möglichkeit eigene Attribute hinzuzufügen oder die bereits vorhandenen Methoden zu modifizieren.

```
import threading

class MeinThread(threading.Thread):

    def __init__(self, threadId, threadName, nachricht):
        threading.Thread.__init__(self)
        self.threadId = threadId
        self.threadName = threadName
        self.nachricht = nachricht

    def run(self):
        for x in range(100):
            print(self.nachricht)

mt1 = MeinThread(1, "Gruesse", "Hallo Welt!")
mt2 = MeinThread(2, "Abschied", "Bye Welt!")
mt3 = MeinThread(3, "Spam", "dlfjslfhajdsf")

mt1.start()
mt2.start()
mt3.start()
```

In diesem Beispiel erstellen wir eine Klasse *MeinThread*, welche eine ID, einen Namen und eine Nachricht als Attribute hat. Sobald der Thread gestartet bzw. ausgeführt wird, wird diese Nachricht 100-mal ausgegeben. Wichtig ist hierbei, dass wir die Funktion *run()* modifizieren und nicht die Funktion *start()*. Beim Aufruf der Start-Funktion wird nämlich ein neuer Thread gestartet, welcher die Run-Funktion aufruft. Wenn wir also hier die Start-Funktion überschreiben, haben wir kein Multithreading mehr.

SYNCHRONISIEREN VON THREADS

Wenn Sie mehrere Threads laufen lassen, welche beide ein und dieselben Ressourcen benutzen, kann das zu Inkonsistenz und Problemen führen. Um solche Situationen zu vermeiden, gibt es das Konzept des sogenannten *Lockings*. Dabei sperrt ein Thread alle anderen und diese können erst weiterarbeiten, wenn die Sperre aufgehoben wurde.

Folgendes Programm ist zwar sehr trivial, veranschaulicht jedoch sehr gut das Problem. Es würde endlos weiterlaufen, da sich die beiden Threads permanent in die Quere kommen und einmal die Zahl verdoppeln und einmal halbieren. Natürlich könnte man hier die Nutzung von Threads generell in Frage stellen, aber wie gesagt: In diesem trivialen Beispiel geht es nur darum, das Problem zu veranschaulichen.

```
import threading
import time

x = 8192

def halbieren():
    global x
    while(x > 1):
        x /= 2
        print(x)
        time.sleep(1)
    print("ENDE!")

def verdoppeln():
    global x
    while(x < 16384):
        x *= 2
        print(x)
        time.sleep(1)
    print("ENDE!")

t1 = threading.Thread(target=halbieren)
t2 = threading.Thread(target=verdoppeln)

t1.start()
t2.start()
```

Wenn Sie nun wollen, dass der erste Thread den anderen blockiert, so

erstellen Sie zunächst eine Instanz der Klasse *Lock* aus dem Threading-Modul. Wichtig ist hierbei, dass der Lock in beiden Funktionen global gemacht wird. Mit der Funktion *acquire()* können Sie nun dem zweiten Thread sagen, dass er warten soll, solange bis der erste Thread den Lock mit der Funktion *release()* aufhebt. Bei der Reihenfolge gilt das Prinzip: Wer zuerst kommt, mahlt zuerst.

```
import threading
import time

x = 8192

lock = threading.Lock()

def halbieren():
    global x, lock
    lock.acquire()
    while(x > 1):
        x /= 2
        print(x)
        time.sleep(1)
    print("ENDE!")
    lock.release()

def verdoppeln():
    global x, lock
    lock.acquire()
    while(x < 16384):
        x *= 2
        print(x)
        time.sleep(1)
    print("ENDE!")
    lock.release()

t1 = threading.Thread(target=halbieren)
t2 = threading.Thread(target=verdoppeln)

t1.start()
t2.start()
```

Durch das Locking wird nun die Funktion des ersten Threads ausgeführt, während der zweite wartet. Nachdem die Funktion zum Punkt gekommen ist, an dem der Lock aufgehoben wird, läuft der andere Thread weiter und kann seinen Lock dann wieder, für einen eventuellen dritten Thread, aufheben.

AUF THREADS WARTEN

Neben dem Locking kann man auch ganz „normal“ auf Threads warten und zwar mithilfe der Funktion *join()*.

```
import threading

def funktion():
   for x in range(500000):
      print("HELLO WORLD!")

t1 = threading.Thread(target=funktion)
t1.start()

print("ENDE!")
```

In diesem Beispiel würde der Thread t1 gestartet werden, was dazu führt, dass „Hello World“ 500000-mal ausgegeben wird. Da jedoch das Hauptprogramm weiterläuft, wird „ENDE!“ sofort nachher ausgegeben. Es wird also nicht gewartet bis der Thread zu Ende gelaufen ist.

```
import threading

def funktion():
   for x in range(500000):
      print("HELLO WORLD!")

t1 = threading.Thread(target=funktion)
t1.start()

t1.join()

print("ENDE!")
```

Hier wird auf das Beenden des Threads t1 gewartet, indem dessen Funktion *join()* aufgerufen wird. Wichtig ist hierbei, dass nur auf den Thread t1 gewartet wird. Sollten also noch andere Threads, zum Beispiel t2 und t3, laufen, wird auf diese nicht gewartet, es sei denn, man ruft auch deren Join-Funktion auf.

```
import threading

def funktion():
```

```
    for x in range(500000):
        print("HELLO WORLD!")

t1 = threading.Thread(target=funktion)
t1.start()

t1.join(5)

print("ENDE!")
```

In diesem Codebeispiel sehen Sie, dass sie der Join-Funktion auch ein Timeout übergeben können. Hier haben wir fünf Sekunden. Das bedeutet, dass auf t1 gewartet wird, maximal aber fünf Sekunden lang.

SEMAPHOREN

Wenn Sie eine Ressource nicht einfach nur sperren, sondern limitieren wollen, so können Sie zu sogenannten Semaphoren greifen. Diese limitieren den Zugriff auf eine Ressource, sodass zum Beispiel nie mehr als x Zugriffe gleichzeitig geschehen können.

```
import threading
import time

semaphor = threading.BoundedSemaphore(value=5)

def funktion(x):
    semaphor.acquire()
    for i in range(10):
        print(x)
        time.sleep(1)
    semaphor.release()

for x in range(10):
    t = threading.Thread(target=funktion, args=(x,))
    t.start()
```

Das Beispiel ist relativ trivial, veranschaulicht jedoch gut die Funktionsweise einer Zählersemaphore (*BoundedSemaphore).* Wir haben bei dem Semaphor ein Maximum von fünf Zugriffen definiert. Wenn wir uns nun die Schleife unten ansehen, so sehen wir, dass zehn Threads erstellt werden, welche die Funktion *funktion()* aufrufen und ihr die Zahlen 0-9 übergeben. Bei jedem Aufruf der Funktion wird der Semaphor beansprucht und die verbleibenden Plätze werden weniger. Sobald die ersten fünf Threads (Nummer 0-4) ihren Zugriff beansprucht haben, bleibt keinen Platz mehr für die Nummern 5-9. Diese müssen jetzt warten, bis die obere Schleife die Funktion *release()* erreicht hat. Bei jedem Release wird nämlich ein besetzter Platz wieder verfügbar.

Ein pragmatischeres Beispiel wäre es, die Verbindungen zu einem Server zu limitieren:

```
import threading

max = 5
```

```
semaphor = threading.BoundedSemaphore(value=max)

def verbinden():
    semaphor.acquire()
    # Code zum Verbinden auf den Server
    semaphor.release()
```

EVENTS

Wenn Sie einen Thread anhalten wollen und ihn erst dann weiterlaufen lassen möchten, wenn ein bestimmtes Ereignis passiert, so ist das mit der Klasse *Event* möglich.

```
import threading

event = threading.Event()

def funktion():
    print("Warte auf Event!")
    event.wait()
    print("Event erfüllt!")

thread = threading.Thread(target=funktion)
thread.start()

x = input("Event setzen?")
if(x == "ja"):
    event.set()
```

In diesem Beispiel wartet der Thread in der Funktion auf das Event, mittels *wait()*. Der Thread läuft nun erst dann weiter, wenn das Event, mittels *set()*, gesetzt wird.

DAEMON THREADS

Die sogenannten Daemon Threads, sind Threads, welche im Hintergrund laufen und das Programm deshalb auch beendet werden kann, wenn dieser Threads noch weiterläuft. Typische Daemon Threads sind solche, welche Inhalte synchronisieren, laden oder auch Garbage Collection. Einen Thread definieren Sie als Daemon, indem Sie im Konstruktor das Attribut *daemon* auf True setzten.

```
import threading
import time

path = "print.txt"
text = ""

def readFile():
    global path, text
    while True:
        file = open(path)
        text = file.read()
        time.sleep(3)

def printloop():
    global text
    for x in range(30):
        print(text)
        time.sleep(1)

t1 = threading.Thread(target=readFile, daemon=True)
t2 = threading.Thread(target=printloop)

t1.start()
t2.start()
```

In diesem Beispiel haben wir eine Funktion *readFile()* welche in einer Dauerschleife alle drei Sekunden den Inhalt der Datei *print.txt* in den String *text* schreibt. Außerdem haben wir eine Funktion *printloop()*, welche 30-mal jede Sekunde den aktuellen String ausgibt. Wir definieren dann den Thread t1, welcher die Zielfunktion *readFile()* hat, als Daemon. Dieser Thread läuft jetzt im Hintergrund und zwar solange, bis das Programm beendet wird. Der zweite Thread gibt nun 30-mal den aktuellen String aus.

Wenn wir keinen Daemon definieren würden, würde das Programm endlos weiterlaufen, da gewartet werden würde, bis der Thread t1 beendet wird. Da dieser jedoch im Hintergrund läuft, hält er das Programm nicht vom Beenden ab.

MULTIPROCESSING

Neben dem Multithreading, gibt es in Python auch noch das Multiprocessing. Hierbei erstellen wir keine neuen Threads, welche parallel ablaufen, sondern Prozesse.

```
import multiprocessing as mp
import time

def printing():
    for x in range(5):
        print("Hallo!")
        time.sleep(1)

if __name__ == '__main__':
    p1 = mp.Process(target=printing)
    p1.start()
    print("Prozess p1 gestartet!")
```

Wenn Sie dieses Programm ausführen und dabei den Task Manager (unter Windows oder ein ähnliches Tool unter Linux) betrachten, sehen Sie, dass tatsächlich ein neuer Prozess gestartet wird. Noch eindrucksvoller wird das Ganze natürlich, wenn Sie mehrere Prozesse starten.

Wichtig ist, dass Sie hier überprüfen, ob der Name „*__main__*“ ist, um rekursives Erstellen von Subprozessen zu vermeiden.

Ansonsten können Sie mit Prozessen so ziemlich dasselbe machen, wie mit Threads. Sie können Daemon Prozesse im Hintergrund erstellen und auch eigene Prozesse als Klassen definieren.

4 – QUEUES

QUEUES

Queues sind in Python Warteschlangen, welche Objekte in einer bestimmten Reihenfolge aufnehmen und wieder hinauslassen. Die standardmäßige Queue ist die sogenannte FIFO-Queue. FIFO steht für „first in first out“, was so viel heißt wie: Wer zuerst hineinkommt, kommt auch zuerst hinaus.

```
import queue

q = queue.Queue()

for x in range(5):
    q.put(x)

for x in range(5):
    print(q.get(x))
```

In diesem, auch wieder trivialen, Beispiel sieht man was die Queue macht. Durch die Funktion *put()* gelangt ein neues Element in die Warteschlange, in diesem Fall die Zahlen 0 bis 4. Durch die Funktion *get()* werden diese nun wieder rausgeholt.

```
import threading
import queue
import math

def worker():
    while True:
        item = q.get()
        if item is None:
            break
        print("Faktorielle von %s ist %s" % (item,math.factorial(item)))
        q.task_done()

q = queue.Queue()
threads = []

for x in range(5):
    t = threading.Thread(target=worker)
    t.start()
```

```
    threads.append(t)

zahlen = [134000, 14, 5, 300, 98, 88, 11, 23]

for item in zahlen:
    q.put(item)

q.join()

for i in range(5):
    q.put(None)

for t in threads:
    t.join()
```

Dieses Beispiel ist schon etwas praktikabler und komplexer. Das Problem beim Threading ist, dass mehrere Threads sich manchmal dieselbe Ressource teilen und gleichzeitig benutzen müssen. Hierbei ist Locking also keine Alternative. In diesem Beispiel haben wir ein Array voller Zahlen, deren Faktorielle wir wissen möchten. Wir beauftragen fünf Threads damit diese zu berechnen. Das Problem welches wir normalerweise hätten ist, dass es für die Threads schwierig ist, sich abzusprechen, welche die nächste Zahl ist, welche berechnet werden soll. Hier kommt die Queue ins Spiel. Anfangs werden nämlich alle Zahlen nacheinander in die Warteschlange gestellt und von den Threads abgeholt. Jeder Thread hat eine Worker-Funktion, welche nacheinander die Zahlen durchrechnet. Wenn ein Thread bereits da war und eine Zahl abgeholt hat, kann diese Zahl von keinem anderen Thread mehr abgeholt werden.

Die erste Zahl wird also abgeholt. Da diese sehr groß ist, dauert die Berechnung sehr lange. Also rechnen die anderen Threads in der Zwischenzeit mit den anderen Zahlen.

LIFO-QUEUES

LIFO steht für „last in first out”, was bedeutet, dass jene Elemente, welche zuerst hineinkommen, als letztes hinauskommen. Dies kann nützlich sein, wenn man eine Liste umgekehrt einlesen will:

```
import queue

q = queue.LifoQueue()

liste = [1, 2, 3, 4, 5]

for x in liste:
    q.put(x)

while not q.empty():
    print(q.get())
```

PRIORISIERUNG IN QUEUES

Natürlich können Sie in Python auch Queues erstellen, in welchen Sie die Objekte einzeln priorisieren können. Das tun Sie mit einer sogenannten *PriorityQueue*. Bei der Erstellung dieser übergeben Sie dem Konstruktor ein Tupel mit der Priorität und dem Wert.

```
import queue

q = queue.PriorityQueue()

q.put((8, "Dritter"))
q.put((1, "Erster"))
q.put((90, "Letzter"))
q.put((2, "Zweiter"))

while not q.empty():
    print(q.get()[1])
```

In diesem Codebeispiel sehen Sie, dass wir eine priorisierte Queue erstellen und ihr durcheinander drei Elemente mit einer gewissen Priorisierung übergeben (die Zahlen). Wenn wir uns nun alle Elemente der Reihe nach ausgeben lassen, bekommen wir sie der Priorität nach sortiert zurück.

Wir lassen uns in diesem Beispiel das zweite Element des Tupels ausgeben, um nur den Text, ohne Priorität, zu sehen.

5 – SOCKETS

Sobald wir uns mit der Netzwerkprogrammierung befassen, treffen wir relativ schnell auf den Begriff Sockets. Sockets sind die Endpunkte von bidirektionalen Kommunikationskanälen oder einfach ausgedrückt: Zwei Punkte die miteinander reden. Sie können innerhalb eines Prozesses, zwischen Prozessen oder aber auch zwischen zwei Kontinenten über das Internet kommunizieren.

Wichtig ist, dass wir in Python zwei Zugriffslevel bei den Netzwerkdiensten haben. Auf der unteren Ebene haben wir Zugriff auf die einfachen Sockets, mit welchen wie verbindungsorientierte oder verbindungslose Protokolle implementieren können, während andere Python-Module wie FTP oder HTTP auf einer höheren Ebene, dem Application-Layer, arbeiten.

Mittels Sockets können wir über verschiedene Protokolle wie TCP oder UDP kommunizieren, aber auch über die Unix Domain Sockets.

ERSTELLEN VON SOCKETS

Um mit Sockets zu arbeiten, benötigen Sie das Modul *socket*:

```
import socket
```

Um nun einen Socket zu initialisieren, müssen wir einige Dinge im Voraus wissen:

- Internet Socket oder Unix Socket?
- Welche Socket Art?
- Welches Protokoll (Standard = 0)?

Bei dem ersten Punkt wählen wir zwischen den Möglichkeiten AF_UNIX und AF_INET. Es gibt zwar noch andere, welche für uns jedoch derzeit nicht von Bedeutung sind, es sei denn Sie lesen dieses Buch zu einer Zeit in der es nur mehr IPv6 Adressen gibt. Bei der Art des Sockets wählen wir zwischen SOCK_STREAM und SOCK_DGRAM. Ersteres wird von TCP und letzteres von UDP benutzt.

TCP ist ein Protokoll, welches verbindungsorientiert ist, was dazu führt, dass es genauer und verlustfrei ist, dafür jedoch langsamer als UDP.

Das Protokoll ist vorerst nicht weiter wichtig.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Hier haben wir einen Socket definiert, welcher mittels IP und TCP arbeiten soll.

CLIENT-SERVER-ARCHITEKTUR

Um uns mit Sockets in Python vertraut zu machen, werden wir uns die Prinzipien anhand des Client-Server-Modells ansehen. Simpel zusammengefasst ist der Client jener, welches etwas vom Server anfordert und der Server jener, welcher, je nach Berechtigung, dem Client schickt was dieser braucht.

Ein Server eröffnet mit jedem Client eine Session und kann so mehrere Clients bedienen und spezifisch auf diese eingehen.

SERVER SOCKET METHODEN

Es gibt drei Methoden der Klasse *socket*, welche speziell für den Server von hoher Bedeutung sind:

- *bind()* – Bindet die Adresse, bestehend aus Hostname und Port, an den Socket.
- *listen()* – Wartet auf eine Nachricht.
- *accept()* – Akzeptiert die Verbindung mit dem Client.

CLIENT SOCKET METHODEN

Es gibt eine Methode, welcher nur der Client benötigt, um sich zum Server zu verbinden und das ist die Methode *connect()*. Diese Methode schickt dem Server eine Verbindungsanfrage, welche mit *accept()* akzeptiert werden muss.

WEITERE SOCKET METHODEN

Es gibt noch zahlreiche weitere Methoden für Sockets. Im Folgenden schauen wir uns jedoch nur jene an, welche für uns derzeit von Bedeutung sind.

- *recv()* – Erhält den Text, welcher an den Socket gesendet wurde.
- *send()* – Sendet einen Text an einen anderen Socket.
- *recvfrom()* – Erhält den Text, welcher an den Socket durch UDP gesendet wurde.
- *sendto()* – Sendet einen Text über UDP an einen anderen Socket.
- *close()* – Der Socket wird geschlossen.
- *gethostname()* – Liefert der Hostname des Sockets zurück.

EINEN SERVER ERSTELLEN

Als erstes beginnen wir mit der Erstellung eines Servers. Um einen Server zu erstellen, definieren wir zunächst einen Socket, welcher über das Internet geht und TCP benutzt. Dann müssen wir ihm über die Funktion *bind()* noch einen Hostnamen und einen Port übergeben. Wir nehmen im folgenden Beispiel als Hostname *localhost* bzw. *127.0.0.1* und den Port 9999. Dann geben wir ihm den Befehl zum "Hören" bzw. zum Warten auf eine Nachricht, indem wir die Funktion *listen()* aufrufen.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1", 9999))
s.listen()
```

Nachdem wir das gemacht haben, lassen wir eine Endlosschleife laufen, welche Verbindungen von Clients akzeptiert.

server.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1", 9999))
s.listen()

while True:
    client, addresse = s.accept()
    print("Verbunden mit %s" % str(addresse))

    nachricht = "Hallo Client"
    client.send(nachricht.encode('ascii'))
    client.close()
```

So sieht der fertige simple Server aus. In der Schleife wird mit der Methode *accept()* permanent jede Verbindung von Clients akzeptiert. Die Methode liefert zusätzlich noch einen Socket zurück, welcher als Client zur Kommunikation dient und die Adresse des Clients, welcher sich gerade verbunden hat. Mit dem zurückgegebenen Client können wir nun eine Nachricht senden, wobei wir diese mit der Funktion *encode()* zuerst, mittels

ASCII, kodieren müssen, da wir über die Sockets nur Bytes schicken können und keine Strings. Danach schließen wir den Client-Socket.

EINEN CLIENT ERSTELLEN

Um nun unsere Ressourcen (in diesem Fall den String „Hallo Welt“) aus dem Server zu holen, benötigen wir einen Client. Diesen definieren wir genau wie den Server, nur dass dieses Mal nicht die Funktion *bind()*, sondern die Funktion *connect()* benutzt wird. In diese tragen wir den Hostname und den Port des Servers ein.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 9999))
```

Um nun die Nachricht vom Server zu erhalten, müssen wir die *recv()* Methode aufrufen und ihr die maximale Länge übergeben (im folgenden Beispiel 1024 Byte). Danach schließen wir den Socket und geben die Nachricht dekodiert aus.

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 9999))
nachricht = s.recv(1024)
s.close()
print(nachricht.decode('ascii'))
```

Um das Programm nun zu testen, müssen Sie zunächst den Server starten und dann erst den Client. Der Server kann auch von mehreren Clients angefragt werden. Damit es nicht zu viel wird kann man der Methode *listen()* auch eine maximale Anzahl an Verbindungen übergeben.

PORT SCANNER

Mit unserem bisherigen Wissen können wir nun einen simplen Port Scanner programmieren, welche auf einer bestimmten Adresse Ports darauf überprüft, ob sie offen oder zu sind.

WARNUNG: *Port Scanning auf Hosts, für welche man keine Berechtigung hat, ist eine Straftat. Sie sollten nur ihre eigenen Netzwerke oder solche, für welche Sie die Erlaubnis haben, scannen. Ich übernehme keine Haftung für Ihr Handeln!*

```
import socket

ziel = "10.0.0.5"

def portscan(port):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        conn = s.connect((ziel, port))
        return True
    except:
        return False

for x in range(500):
    if(portscan(x)):
        print("Port %s ist offen!" % x)
    else:
        print("Port %s ist geschlossen!" % x)
```

Als erstes definieren wir hier einen String mit unserer Zieladresse. In diesem Fall ist das eine private IP, nämlich 10.0.0.5. Danach definieren wir eine Funktion für den Portscan. In dieser benutzen wir ein *try* um uns mit unserem Socket zu dem Host auf den übergebenen Port zu verbinden. Sollte das gelingen liefert die Funktion *True*, sonst *False*. Dann lassen wir eine Schleife lassen, welche, in diesem Fall, die ersten 500 Ports prüft. Sollte der Port x nun frei sein, wird *True* zurückgegeben und ausgegeben, dass der Port geöffnet ist.

THREADED PORT SCANNER

Das Problem bei diesem simplen Port Scanner ist, dass er viel zu langsam ist, da ein Port nach dem anderen seriell gescannt wird. Um den Port Scanner schneller zu machen, können wir mehrere Threads starten, welche sich die Aufgabe teilen.

```
import socket
from queue import Queue
import threading

ziel = "localhost"

q = Queue()
for x in range(500):
    q.put(x)

def portscan(port):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        conn = s.connect((ziel, port))
        return True
    except:
        return False

def worker():
    while True:
        port = q.get()
        if(portscan(port)):
            print("Port %s ist offen!" % port)
        else:
            print("Port %s ist geschlossen!" % port)

for x in range(30):
    t = threading.Thread(target=worker)
    t.start()
```

Zunächst erstellen wir eine Queue, in welche wir die Portnummern alle der Reihe nach hineingeben. Von dort holen sich die einzelnen Threads dann die zu scannenden Ports raus. Wir benutzen hier eine Queue, damit nicht dieselben Ports doppelt gescannt oder übersprungen werden. Würden wir eine Variable definieren, welche immer um Eins erhöht wird, könnte es passieren, dass gerade zwei Threads denselben Port scannen oder erhöhen. In

diesem Beispiel starten wir dann anschließend 30 Threads, welche ´Ports scannen.

6 – REKURSION

Die Rekursion ist in der Programmierung ein erweitertes Konzept, in welchem es darum geht, dass eine Funktion sich selbst aufruft.

```
def funktion():
    funktion()
```

Sollten Sie nun die Funktion *funktion()* aufrufen, so wird das Programm einen *RecursonError* als Exception werfen. Außerdem hat jedes Programm einen maximalen Stack-Speicher, welches es benutzen darf. Sollte dieser Stack-Speicher auf Grund von Rekursion volllaufen, so bekommt man einen Stack-Overflow.

FAKULTÄTSFUNKTION REKURSIV

Als sinnvolle Anwendung schauen wir uns einmal die Implementierung der Fakultätsfunktion, aus der Mathematik, rekursiv an:

```
def fakt(n):
    if n < 1:
        return 1
    else
        zahl = n * fakt(n-1)
        return zahl
```

Hier sehen wir, dass sich die Funktion im Else-Zweig selbst aufruft und einen Wert niedriger als Parameter übergibt. Diese Funktion ruft sich solange selbst auf, bis sie den Wert Eins erreicht hat. Dann wird die Berechnung durchgeführt und das Ergebnis wird von der einen Instanz der Funktion an die nächste zurückgeliefert.

7 – WITH-STATEMENTS

In Python haben wir es oft mit Objekten zu tun, welche zuerst aufgesetzt und initialisiert werden müssen, dann wird mit ihnen gearbeitet und am Ende müssen sie wieder geschlossen oder gelöscht werden.

Folgende Beispiele zeigen das gut:

Sockets

```
s.connect(("10.0.0.1", 9999))
# Irgendwelche Aktionen
s.close()
```

Dateistreams

```
f = open("dateiname.txt")
# Irgendwelche Aktionen
f.close()
```

Wo immer wir etwas initialisieren, das wir danach schließen müssen, hilft und das With-Statement. Dieses sorgt dafür, dass das „aufräumen“ danach automatisch übernommen wird:

```
with open("datei.txt") as f:
    f.read()
    # Aktionen
```

In diesem Beispiel wird der Filestream, nach der Benutzung, automatisch geschlossen.

WITH IN EIGENEN KLASSEN

Wenn Sie wollen, dass man *with* auch auf Ihre Klassen anwenden kann, so haben Sie dafür zwei Methoden, welche Sie definieren müssen.

```
class Klasse():
    def __init__(self):
        print("Initialisiert!")

    def __enter__(self):
        print("Anfang!")

    def __exit__(self, type, value, traceback):
        print("Ende!")

with Klasse():
    pass
```

In diesem Beispiel wird zunächst die Klasse erstellt, durch das *with* wird die Methode *enter()* aufgerufen und am Ende wird durch die Methode *exit()* alles beendet.

8 – XML PROCESSING

XML ist eine Markup Sprache, genauso wie HTML. Es wird benutzt um hierarchisch strukturierte Daten im Format einer Textdatei darzustellen, welche sowohl von Menschen als auch von Computern gelesen bzw. interpretiert werden kann. XML ist plattform- und applikationsunabhängig. Jeder Programmierer stößt früher oder später auf XML, weswegen es für uns von großer Wichtigkeit ist, dieses Format in Python bearbeiten zu können.

XML PARSER

In der Python Standardbibliothek können wir zwischen zwei Modulen für das XML-Parsing wählen: SAX und DOM.

SAX steht für *Simple API für XML* und eignet sich besser für große XML-Dateien oder dann, wenn unser RAM stark limitiert ist. Bei diesem Modul werden die Dateien nämlich von der Festplatte gelesen und immer nur Teile in den RAM geladen. Es befindet sich nie die gesamte Datei im Arbeitsspeicher, was auch dazu führt, dass man aus der Datei nur lesen, jedoch nichts hineinschreiben kann.

DOM steht für *Dokumenten Objekt Model* und ist von dem World Wide Web Konsortium empfohlen. Bei diesem Modul wird die ganze XML Datei in den Arbeitsspeicher geladen und dort in hierarchischer strukturierter Form gespeichert. Damit kann man dann alle Features von XML ausnutzen. Wir können hierbei auch die Datei ändern.

Logischerweise ist DOM um einiges schneller als SAX, da wir den RAM benutzen und nicht die Festplatte. SAX benutzen wir dann, wenn unser RAM so limitiert ist, dass wir nicht die gesamte Datei problemlos hineinladen können.

Es gibt keinen Grund in einem großen Projekt nicht beide Module zu verwenden. Je nach Anwendungsfall kann man dann zu SAX oder zu DOM greifen.

Für die Beispiele in diesem Kapitel werden wir folgendes XML-File verwenden:

```
<?xml version="1.0"?>
<gruppe>
  <person id="1">
    <name>Max Mustermann</name>
    <alter>20</alter>
    <gewicht>80</gewicht>
    <groesse>188</groesse>
  </person>
  <person id="2">
    <name>Sebastian Tester</name>
```

```
    <alter>45</alter>
    <gewicht>82</gewicht>
    <groesse>185</groesse>
  </person>
  <person id="3">
    <name>Anna Winter</name>
    <alter>33</alter>
    <gewicht>67</gewicht>
    <groesse>167</groesse>
  </person>
  <person id="4">
    <name>Berta Schlau</name>
    <alter>60</alter>
    <gewicht>70</gewicht>
    <groesse>174</groesse>
  </person>
  <person id="5">
    <name>Sarah Holzer</name>
    <alter>12</alter>
    <gewicht>50</gewicht>
    <groesse>152</groesse>
  </person>
</gruppe>
```

Wir haben hier eine Gruppe, welche fünf Personen als Mitglieder hat. Jede Person hat eine ID als Attribut und die vier Elemente Name, Alter, Gewicht und Größe.

XML MIT SAX

Um mit SAX zu arbeiten, benötigen wir einen ContentHandler. Dieser verarbeitet für uns die Attribute und Tags des XML-Files. Dann gibt es noch zwei Methoden: *startDocument()* und *endDocument()*, welche am Start und am Ende des XML-Files aufgerufen werden. Ebenso gibt es die Methode *characters()*, welche dann aufgerufen wird, wenn ein Zeichen gelesen wird.

Um einen XML-Parser zu erstellen, benötigen wir die *make_parser()* Methode. Wollen wir ein Dokument einfach nur parsen, ohne einen Parser zu erstellen, so können wir die Methode *parse()* verwenden. Dieser müssen wir jedoch zwei Parameter übergeben, nämlich den Pfad zur XML-Datei und den ContentHandler. Wir können jedoch auch, mit der Methode *parseString()*, einen einfachen XML-String parsen.

```
import xml.sax

handler = xml.sax.ContentHandler()

parser = xml.sax.make_parser()
xml.sax.parse("gruppen.xml", handler)
```

Um jedoch wirklich effektiv arbeiten zu können, müssen wir eine eigene ContentHandler-Klasse definieren, sodass wir die wichtigen Methoden überschreiben können.

```
import xml.sax

class GruppenHandler(xml.sax.ContentHandler):
    def startElement(self, name, attrs):
        print(name)

handler = GruppenHandler()
parser = xml.sax.make_parser()
parser.setContentHandler(handler)
parser.parse("gruppen.xml")
```

Dieses Beispiel ist sehr simpel. Es gibt einfach nur die Namen der einzelnen Elemente aus, also: Gruppe, Person, Name, Alter, Gewicht und Größe. Wir haben einen eigenen GruppenHandler erstellt und dessen *startElement()* Methode neu definiert, sodass diese den Namen des Elementes ausgibt. Dann

erstellen wir eine Instanz des Handlers, welche wir dann unserem erstellten Parser zuweisen. Dieser parst dann die Datei *gruppen.xml*.

Folgendes Beispiel ist etwas umfangreicher:

```
import xml.sax

class GruppenHandler(xml.sax.ContentHandler):
    def __init__(self):
        self.aktuell = ""
        self.name = ""
        self.alter = ""
        self.gewicht = ""
        self.groesse = ""

    def startElement(self, name, attrs):
        self.aktuell = name
        if self.aktuell == "person":
            print("--- Person ---")
            id = attrs["id"]
            print("ID: %s" % id)

    def endElement(self, name):
        if self.aktuell == "name":
            print("Name: %s" % self.name)
        elif self.aktuell == "alter":
            print("Alter: %s" % self.alter)
        elif self.aktuell == "gewicht":
            print("Gewicht: %s" % self.gewicht)
        elif self.aktuell == "groesse":
            print("Größe: %s" % self.groesse)
        self.aktuell = ""

    def characters(self, content):
        if self.aktuell == "name":
            self.name = content
        elif self.aktuell == "alter":
            self.alter = content
        elif self.aktuell == "gewicht":
            self.gewicht = content
        elif self.aktuell == "groesse":
            self.groesse = content

handler = GruppenHandler()
parser = xml.sax.make_parser()
parser.setContentHandler(handler)
parser.parse("gruppen.xml")
```

Hier haben wir eine eigene Handler-Klasse definiert, welche im Konstruktor die Elemente einer Person definiert und zusätzlich noch ein Attribut mit dem Namen *aktuell*, sodass wir wissen, bei welchem Element wir gerade sind.

In der Methode *startElement()* speichern wir nun in unsere Variable *aktuell* den Namen des aktuellen Elements. Wenn dieses Element eine Person sein sollte, so geben wir einen String aus und die entsprechende ID.

In der Methode *characters()* befassen wir uns mit den Fällen, in welchen wir Elemente haben, welche innerhalb einer Person sind. Wir setzen dann den Wert unseres Handlers gleich dem Wert in der XML-Datei.

Anschließend geben wir diese Werte in der Methode *endElement()* aus.

XML MIT DOM

Das Document Object Model (DOM), ist eine sprachunabhängige API um mit XML zu arbeiten. Mit dieser können wir nun auch Dateien bearbeiten. Hier müssen wir das Modul *minidom* importieren.

```
import xml.dom.minidom
```

Um nun damit zu arbeiten, müssen wir einen DOM-Tree erstellen und die einzelnen Elemente als Collections betrachten.

```
import xml.dom.minidom

xml.dom.minidom

domtree = xml.dom.minidom.parse("gruppen.xml")
gruppe = domtree.documentElement

personen = gruppe.getElementsByTagName("person")

for person in personen:
    print("--- Person ---")
    if person.hasAttribute("id"):
        print("ID: %s" % person.getAttribute("id"))

    name = person.getElementsByTagName("name")[0]
    alter = person.getElementsByTagName("alter")[0]
    gewicht = person.getElementsByTagName("gewicht")[0]
    groesse = person.getElementsByTagName("groesse")[0]

    print("Name: " + name.childNodes[0].data + "\n"
          "Alter: " + alter.childNodes[0].data + "\n"
          "Gewicht: " + gewicht.childNodes[0].data + "\n"
          "Größe: " + groesse.childNodes[0].data)
```

Zunächst lesen wir die Datei in einen DOM-Tree ein, aus welchem wir uns dann das Dokumentenelement holen (gruppe). Dann definieren wir eine Personen-Collection, in welche wir alle Personen-Elemente einlesen. Durch diese lassen wir dann eine Schleife laufen, welche alle Elemente und Attribute ausliest und ausgibt.

Wie bereits gesagt, können Sie mit DOM auch XML-Dateien bearbeiten. Das würde jedoch den Rahmen dieses Buches sprengen. Sollten Sie sich dafür

interessieren, können Sie einen Blick auf die Dokumentation dazu werfen:

https://docs.python.org/3/library/xml.dom.html

9 – LOGGING

Egal was Sie in der Informatik tun, ob Sie einen Server oder ein Programm administrieren, eine Software entwickeln oder irgendetwas anderes, früher oder später kommen Sie ohne Logs nicht mehr weiter. Alles wird heutzutage protokolliert, von Banktransaktionen, über Flüge bis hin zu unseren Betriebssystemen (Windows Event Logs). Log-Files helfen Ihnen dabei Probleme zu finden, Informationen zu erhalten oder Warnungen zu empfangen. Sie sind ein essentielles Tool um Fehler zu vermeiden und zu verstehen.

Bisher haben wir, falls irgendwo ein Fehler aufgetreten ist, stets eine Meldung oder eine Exception auf die Konsole ausgegeben. Bei größeren Anwendungen wird dies jedoch schnell nicht mehr überschaubar und wir müssen unsere Logs kategorisieren und auslagern. Außerdem ist nicht jede Meldung gleichrelevant. Manche Meldungen sind dringend, weil eine essentielle Komponente ausgefallen ist, andere wiederum sind nur nette Informationen.

SECURITY LEVELS

In Python haben wir fünf Stufen der Wichtigkeit und zwar folgende (aufsteigend nach Wichtigkeit sortiert):

1. Debug (Debug)
2. Info (Information)
3. Warning (Warnung)
4. Error (Fehler)
5. Critical (Kritisch)

Debug benutzt man meistens nur um etwas zu testen bzw. zu überprüfen. Typischerweise bei der Fehlersuche auszugeben.

Information wird dann benutzt, wenn ich während dem Betrieb des Programmes, demjenigen Nutzer, welcher die Konsole vor sich hat etwas Informierendes mitteilen möchte. Meistens eine Meldung für ein erfolgreich abgeschlossenes Unterfangen. Beispiel: „User Max Mustermann erfolgreich eingeloggt!“

Eine warnende Meldung loggt man dann, wenn eine Gefahr besteht, welche dem Nutzer mittgeteilt werden soll. Beispiel: „Sie haben bereits 70% des verfügbaren RAM-Speichers alloziert!“

Eine Fehlermeldung wird dann ausgegeben, wenn ein wichtiges Vorhaben im Programm fehlschlägt. Beispiel: Alle Exceptions geben Errors aus.

Kritische Meldungen gibt man dann aus, wenn eine, für den Weiterbetrieb des Programmes, essentielle Komponente ausgefallen ist. Beispiel: „Server abgestürzt – Fehler XYZ!“

LOGGER ERSTELLEN

Um eine Instanz eines Loggers zu erstellen, benötigen wir zunächst die Bibliothek *logging*. In dieser finden wir dann die Funktion *getLogger()*, mit welcher wir einen Logger instanziieren können. Wir können jedoch auch den Konstruktor, der Klasse *Logger*, verwenden, welchem wir dann jedoch einen Namen übergeben müssen.

```
import logging

logger = logging.getLogger()
logger.info("Logger wurde erfolgreich erstellt!")
```

Sie können auch mit der Funktion *log()* loggen, wenn Sie das Security Level angeben:

```
import logging

logger = logging.getLogger()
logger.log(logging.INFO, "Erfolgreich!")
```

IN DATEIEN LOGGEN

Bisher haben wir die Log-Meldungen nur auf die Konsole ausgegeben. Wollen wir die Logs auch in eine Datei schreiben, so benötigen wir einen *FileHandler*.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

handler = logging.FileHandler("datei.log")
handler.setLevel(logging.INFO)

logger.addHandler(handler)
logger.info("Diese Meldung kommt in die Datei!")
```

Wir erstellen einen Logger und einen Handler. Dem Handler übergeben wir als Parameter die Datei, in welche er loggen soll. Dann fügen wir dem Logger den Handler hinzu. Ein Logger kann übrigens mehrere Handler haben und somit in mehrere Dateien gleichzeitig loggen.

LOGS FORMATIEREN

Das Problem, welches wir mit unseren aktuellen Logs haben ist, dass wir einfach nur einen Text haben, welcher in eine Datei geschrieben wird. Wir können Warnungen, kritische Meldungen, Fehler und Informationen nicht von einander unterscheiden. Für diesen Zweck gibt es *Formatter*.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

handler = logging.FileHandler("datei.log")
handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s: %(levelname)s - %(message)s')
handler.setFormatter(formatter)

logger.addHandler(handler)
logger.info("Diese Meldung kommt in die Datei!")
```

Hier definieren wir den Formatter und geben ihm ein bestimmtes Pattern. Er soll bei jedem Eintrag, die aktuelle Zeit, den Security Level und die Meldung loggen. Ein Eintrag sieht dann in etwa so aus:

2018-01-12 13:05:23,059: INFO - Diese Meldung kommt in die Datei!

10 – REGULAR EXPRESSIONS

In der Programmierung haben wir es oft mit langen Texten zu tun, aus welchen wir die gewünschte Information extrahieren wollen. Außerdem gibt es oft bestimmte Muster, welche eine Eingabe erfüllen muss. Denken Sie zum Beispiel einmal an E-Mail-Adressen. Sie müssen zuerst einen Text haben, welcher von einem @ gefolgt wird, dann noch einen Text, gefolgt von einem Punkt und am ende dann noch einen kurzen Text.

Genau um solche Überprüfungen kompakt und effizient zu ermöglichen gibt es sogenannte reguläre Ausdrücke bzw. Regular Expressions.

Dass Kapitel der Regular Expressions ist sehr umfangreich. Es gibt ganze Bücher über dieses Thema, weswegen es den Rahmen sprengen würde zu detailliert darauf einzugehen. Wir werden uns in diesem Buch jedoch trotzdem mit einigen Konzepten auseinandersetzen und uns ansehen, wie wir in Python mit ihnen arbeiten können.

Reguläre Ausdrücke können auf den ersten Blick sehr verwirrend aussehen. Eine Regular Expression für eine gültige E-Mail kann zum Beispiel so aussehen:

^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$.

Wir werden in diesem Buch keine allzu komplizierten RegEx entwickeln, sondern uns eher auf die Grundlagen und die Implementierung in Python beschränken.

IDENTIFIER

Die Identifier in regulären Ausdrücken geben an was dort stehen sollte. Hier sind einige Beispiele:

\d = Irgendeine Zahl

\D = Alles außer eine Zahl

\s = Leerzeichen

\S = Alles außer ein Leerzeichen

\w = Irgendein Buchstabe

\W = Alles außer ein Buchstabe

. = Jedes Zeichen, außer ein Zeilenumbruch

\b = Abstände um ein Wort herum

\. = Ein Punkt

MODIFIER

Modifier erweitern die regulären Ausdrücke und die Identifier:

{x,y} = Eine x- bis y-stellige Zahl wird erwartet

+ = Mindestens eins oder mehr

? = Kein Mal oder ein Mal

* = Kein Mal oder mehr als kein Mal

$ = Am Ende des Strings

^ = Am Anfang des Strings

| = Entweder , Oder – Beispiel: x | y = Entweder x oder y

[] = Wertebereich

{x} = X Mal

{x,y} = X bis Y Mal

WHITE SPACES

Hier noch einige White Space Operatoren:

\n = Neue Zeile

\s = Abstand

\t = Tab

REGULÄRE AUSDRÜCKE IN PYTHON

Um in Python mit regulären Ausdrücken zu arbeiten, benötigen wir das Modul *re*. Dann können wir die Funktion *findall()* benutzten, um mit den Ausdrücken einen Text zu filtern.

```
import re

text = """
Florian ist 19 Jahre alt und Georg ist 28.
"""

alter = re.findall(r'\d{1,3}', text)
print(alter)
```

Der reguläre Ausdruck in diesem Beispiel filtert die ein- bis dreistelligen Zahlen in diesem Text raus. In diesem Fall also 19 und 28.

```
import re

text = """
Florian ist 19 Jahre alt und Georg ist 28.
"""

alter = re.search(r'\d{1,3}', text)
print(alter.group())
```

Mit der Funktion *search()* finden wir nur das erste Vorkommen des Strings, welcher unserem Pattern matcht.

```
import re

text = """
Florian ist 19 Jahre alt und Georg ist 28.
"""

alter = re.match(r'\d{1,3}', text)
print(alter.group())
```

Die Funktion *match()* liefert nur dann ein Match zurück, wenn sie bereits zu Beginn des Strings eins findet. Sonst gibt sie, wie in diesem Fall, ein None zurück.

```
import re
```

```
text = """
Florian ist 19 Jahre alt und Georg ist 28.
"""

text = re.sub(r'\d{1,3}', "100", text)
print(text)
```

Mit der Funktion *sub()* ersetzen wir alle Teile des Textes, welche dem Pattern matchen, mit einem anderen String. In diesem Fall ersetzen wir das Alter, der beiden Personen, mit 100.

11 – DAS OS MODUL

Das Modul *OS* ist dafür gedacht, um mit dem Betriebssystem zu interagieren. Mit diesem Modul können wir Dateien umbenennen, Working Directories einsehen und Kommandozeilenbefehle ausführen.

```
import os

# Aktuelles Verzeichnis ausgeben
print(os.getcwd())

# Neues Verzeichnis erstellen
os.mkdir("NeuerOrdner")

# Verzeichnis umbenennen
os.rename("NeuerOrdner", "AlterOrdner")

# Verzeichnis löschen
os.rmdir("AlterOrdner")

# Kommandozeilenbefehle ausführen
os.system("ping google.com")
```

Weitere Infos hier: https://docs.python.org/3/library/os.html

12 – PYTHON ZU .EXE UMWANDELN

Nun haben Sie einiges über die fortgeschrittene Programmierung mit Python gelernt. Um das Ganze noch abzurunden, befassen wir uns in diesem Kapitel damit, eine .py-Datei in eine .exe-Datei umzuwandeln.

Dafür benötigen wir das Modul *cx_Freeze*. Dieses müssen wir jedoch mit pip installieren (siehe Kapitel 1), da es nicht zur Standardbibliothek gehört:

pip install cx_freeze

Dann können wir mit folgendem Code eine Python Datei in eine ausführbare Windows-Datei umwandeln:

```
from cx_Freeze import setup, Executable

setup(name = "script.py" ,
    version = "0.1" ,
    description = "" ,
    executables = [Executable("script.py")])
```

Bei Executable müssen Sie ihr eigenes Python-Skript angeben. Damit das ganze funktioniert, müssen Sie mit Ihrer Kommandozeile in den Ordner navigieren und folgendes eingeben:

python meinskript.py build

Als Skript geben Sie hierbei, das cx_Freeze-Skript an.

Anschließend finden Sie einen Ordner *build* mit Ihrem Programm.


PYTHON
PROGRAMMIEREN
DATA SCIENCE
FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

In unserer modernen Zeit nimmt die Menge an Daten exponentiell zu. Egal wo man hinhört, überall fallen Begriffe wie Big Data, Data Science, Data Analysis und viele mehr. Mit der Zeit gelingt es uns immer besser aus den zunehmenden Datenmengen hilfreiche Informationen zu gewinnen. Das Feld, welches sich primär damit befasst, nennt sich Data Science bzw. Data Analysis. Egal ob Sie Aktienkurse analysieren oder Ihre eigene künstliche Intelligenz entwickeln wollen – Data Science ist essentiell. Jedes moderne und große System hat eine riesige Menge an Daten, welche intelligent verwaltet und analysiert werden müssen.

Es ist also mehr als sinnvoll sich in diesem Bereich massiv weiterzubilden, sodass Sie ein Teil dieser modernen Entwicklung sein können anstatt von ihr überrannt zu werden.

In diesem Buch wird Ihnen Data Science in Python nähergebracht. Wichtig ist hierbei, dass Sie bereits Python relativ gut beherrschen sollten. Hier wird nicht auf die grundlegende Syntax oder auf simple Konzepte eingegangen. Sollten Sie also das Gefühl haben, dass Sie mit Python noch nicht so gut vertraut sind, empfehle ich Ihnen, sich zunächst meine Bücher für Anfänger und für Fortgeschrittene anzusehen.

Nachdem Sie dieses Buch gelesen haben und das Gelernte auch nebenbei angewandt haben, werden Sie in der Lage sein, grundlegende Konzepte von Data Science in Python zu verstehen und anzuwenden. Sie können dann große Datenmengen sammeln, strukturieren, visualisieren und analysieren. Auch wenn wir in diesem Buch nur sehr oberflächlich auf die tatsächlichen Anwendungsgebiete eingehen werden, können Sie anschließend Aktienkurse, Wetterdaten und ähnliches Analysieren. Außerdem stellt Data Science die Grundlage für Intelligente Systeme, also für Machine Learning, dar.

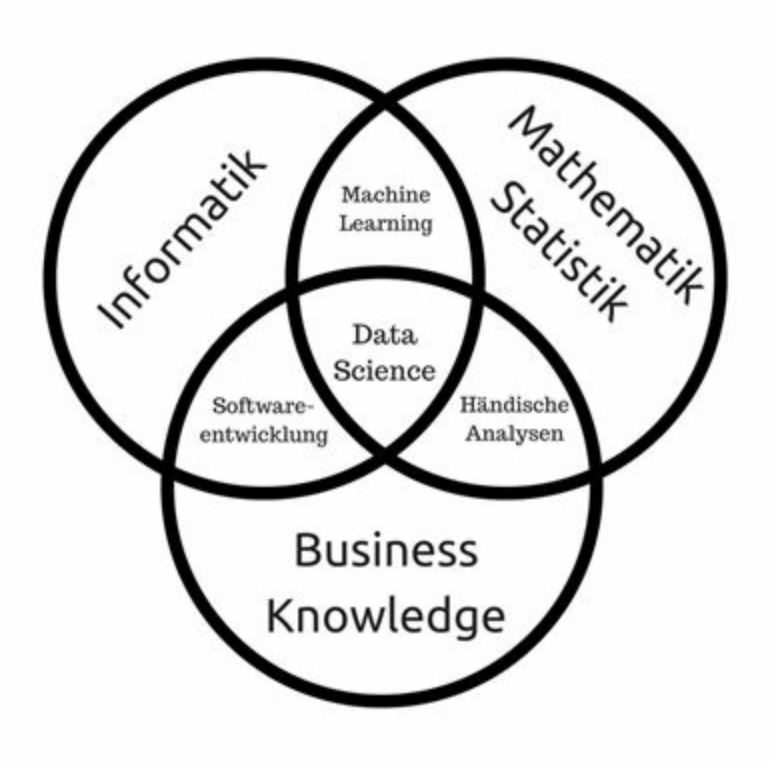
Sie sollten dieses Buch jedoch nicht nur schnell durchlesen und versuchen oberflächlich zu verstehen, worum es geht, sondern auch aktiv nebenbei mitprogrammieren. Dadurch lernen Sie am meisten und das Wissen wird am besten gefestigt. In den folgenden Kapiteln werden Sie zahlreiche

Codebeispiele finden, die Sie nachprogrammieren können. Sie können jedoch auch Ihren eigenen Code schreiben, mithilfe der Dinge die Sie bereits gelernt haben, zu dem Zeitpunkt.

Im Grunde genommen bleibt es Ihnen selbst überlassen, wie Sie diese Lektüre handhaben möchten. In jedem Fall wünsche ich Ihnen maximalen Erfolg beim Programmieren und viel Spaß beim Lernen von Data Science in Python.

1 – WAS IST DATA SCIENCE?

Wenn wir uns mit Data Science bzw. Data Analysis beschäftigen, so versuchen wir stets, Wissen, aus größeren Datenmengen, zu gewinnen. Dabei verwenden wir Modelle, Techniken und Theorien aus den Bereichen Mathematik, Statistik, Informatik, Machine Learning und vielen mehr.



Da wir uns in einer Zeit befinden, in welcher die Menge an Daten exponentiell zunimmt, ist es immer wichtiger, diese großen Datenmengen (Big Data) möglichst effizient und automatisiert zu verarbeiten und zu analysieren. Für diese Aufgabe ist es essentiell, dass wir kompetent in den Bereichen Informatik, Mathematik und Business Knowledge sind. Wenn wir die Wirtschaft außer Acht lassen, befinden wir uns im Bereich des Machine Learnings. Lassen wir die Statistik weg, können wir kaum noch von Analysen sprechen und verzichten wir auf die Technik, so arbeiten wir sehr ineffizient. Im Bereich Data Science befinden wir uns also nur dann, wenn wir von allen drei Kompetenzen Gebrauch machen.

Beispiele für Data Science findet man überall. Wir können

Wahrscheinlichkeiten für Aktienkurse berechnen und darstellen, Normalverteilungen darstellen, statistische Werte berechnen lassen, Prognosen machen und vieles mehr. Egal ob in Wissenschaft, Wirtschaft, Technik oder sonst irgendwo, Data Science wird immer wichtiger.

2 – WARUM PYTHON?

Jetzt stellt sich nur noch die Frage, warum wir zu Python als Programmiersprache für Data Science greifen sollten. Der Grund dafür, dass Python, gerade im Bereich Data Science, so populär ist, ist, dass es zahlreiche ausgezeichnete Bibliotheken gibt, welche eine sehr einfache und effiziente Verarbeitung von Big Data ermöglichen. Es gibt Module für effiziente Listenverarbeitung, für hervorragendes Darstellen von Daten und Zeichnen von Diagrammen, für Scientific Computing und vieles mehr. Da Python Open-Source ist, ist auch die Community entsprechend aktiv und liefert immer wieder neue und effizientere Software-Lösungen.

ALTERNATIVEN ZU PYTHON

Als Alternativen bzw. als Konkurrenz zu Python, im Bereich Data Science, könnte man Sprachen wie R und MATLAB sehen, welche durchaus ihre Daseinsberechtigung haben. MATLAB ist jedoch kostenpflichtig und R spezialisiert sich wirklich nur auf Statistik und Data Analysis. Im Bereich Machine Learning (welcher stark mit Data Science zusammenhängt) und Co. ist R alleine nicht wirklich brauchbar.

WEITERE GRÜNDE

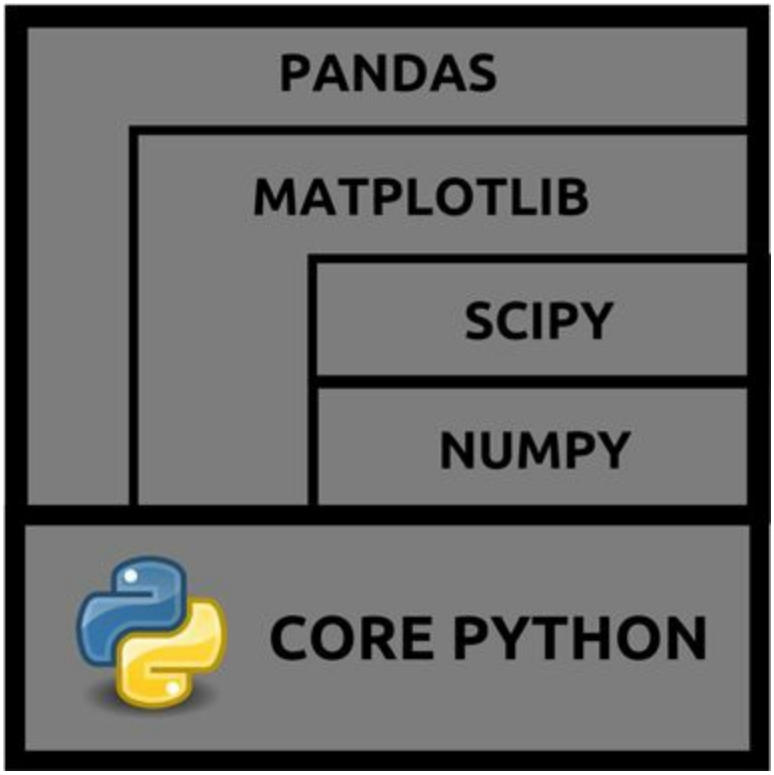
Argumente für die Programmiersprache Python an sich, sind ihre Simplizität gepaart mit der unglaublichen Vielfalt und den zahlreichen Einsatzgebieten. Python ist eine der führenden Programmiersprachen im Bereich Machine Learning, doch auch einige Webapplikationen (wie zum Beispiel Pinterest) sind in Django, welches auf Python basiert, geschrieben.

3 – INSTALLATION DER MODULE

Folgende Module werden wir für Data Science mit Python benötigen:

- Numpy
- Matplotlib
- Pandas

Diese Module basieren alle auf Core-Python. Die folgende Grafik veranschaulicht, die Struktur der Module.



NUMPY

Das Modul Numpy ermöglicht uns das effiziente Arbeiten mit Vektoren, Matrizen und mehrdimensionalen Arrays. Zusätzlich dazu bietet es auch effiziente Methoden für numerische Berechnungen.

Um Numpy zu installieren, geben wir in der Kommandozeile (CMD oder Terminal) ein;

pip install numpy

MATPLOTLIB

Matplotlib ist ein Modul, welches hauptsächlich dafür gemacht ist, Funktionen, Datenmengen und andere Strukturen graphisch darzustellen. Es bietet wahnsinnig viele unterschiedliche Diagramme und Darstellungsmöglichkeiten an. Das Modul selbst benutzt Numpy und baut somit darauf auf.

Um Matplotlib zu installieren, geben wir in der Kommandozeile (CMD oder Terminal) ein;

pip install matplotlib

PANDAS

Pandas ist dann das Top-Level Modul, welches auf Manipulation und Verarbeitung der großen Datenmengen spezialisiert ist. Es liefert auch einige mächtige Datenstrukturen, wie zum Beispiel das DataFrame.

Um Pandas zu installieren, geben wir in der Kommandozeile (CMD oder Terminal) ein;

pip install pandas

Im Laufe dieses Buches werden wir hier und da auf optionale Module stoßen, welche wir dann, je nach Bedarf installieren werden.

4 – ARRAYS MIT NUMPY

Mit dem Modul Numpy alleine, können wir, im Bereich Data Science, nicht wirklich viel anstellen. Es stellt jedoch die Grundlage für eine effiziente Verarbeitung von Listen dar. Es erleichtert das Arbeiten mit Arrays um einiges und liefert viele tolle Funktionalitäten.

Um das Modul benutzen zu können, müssen wir es zunächst importieren:

```
import numpy as np
```

Hier geben wir auch noch einen Alias an, damit wir numpy nicht immer ausschreiben müssen, sondern np verwenden können.

ERSTELLEN VON ARRAYS IN NUMPY

Um in Numpy Arrays zu erstellen, müssen wir einfach nur das Modul ansprechen und die jeweilige Funktion aufrufen:

```
a = np.array([1,2,3,4])
b = np.array([(6,2,3,4), (8, 8, 9, 2)], dtype=float)
```

Wir erzeugen hier in der ersten Reihe ein eindimensionales Array mit vier Werten. In der zweiten Zeile erzeugen wir ein mehrdimensionales Array (2x4), und legen auch noch Float als Datentypen fest, weswegen alle Zahlen in Gleitkommazahlen umgewandelt werden.

Wir können natürlich auch sehr große n-Dimensionale Arrays erstellen.

```
a = np.array([
   [
      [10,20,30,40], [8,8,2,1], [1,1,1,2]
   ],
   [
      [9, 9, 2, 39], [1,2,3,3], [0,0,3,2]
   ],
   [
      [12,33,22,1], [22,1,22,2], [0,2,3,1]
   ]
])
```

In diesem Beispiel haben wir ein 3x3x4 Array. Das bedeutet, dass wir 3 Arrays haben, welche jeweils 3 Arrays beinhalten, mit jeweils 4 Zahlen. Sollten wir nicht in jedem Array vier Zahlen haben, so handelt es sich nur um ein 3x3 Array, da es geometrisch nicht als Würfel angesehen werden kann.

Dimensionen von Arrays kann man sich sehr gut geometrisch vorstellen. Eine Dimension (simples Array) ist eine Linie von Zahlen. Wenn jetzt jede Zahl eine eigene Linie von Zahlen erhält, haben wir ein zweidimensionales Gitter. Wenn nun wiederum jede dieser Zahlen noch eine Reihe von Zahlen erhält, so haben wir ein dreidimensionales Gitter. Das geht natürlich Weiter bis zur Unendlichkeit, doch unser Raumvorstellungsvermögen ist auf drei Dimensionen beschränkt.

PLATZHALTER IN ARRAYS

Numpy gibt uns die Möglichkeit, Arrays mit verschiedensten Werten auszufüllen, in nur einer Zeile. Dafür müssen wir nur die jeweilige Funktion aufrufen und die Struktur des „Gitters“ angeben.

```
a = np.full((3,4,2,2,5), 7)
```

Lassen Sie sich nicht durch die vielen Zahlen erschrecken. Mit der Methode *full()* füllen wir ein Array, einer bestimmten Struktur, mit einer einzigen Zahl auf. In diesem Fall haben wir ein 3x4x2x2x5 Array, welches nur mit der Zahl 7 befüllt wird. Wir hätten hierbei jedoch genauso gut einfach ein 2x2 Array nehmen können.

Es gibt auch noch einige andere Funktionen, um vorstrukturierte Arrays zu befüllen.

```
# 3x4 Array voller Nullen
x1 = np.zeros((3,4))
# 5x3x2x2 Array voller Einsen (4 Dim.)
x2 = np.ones((5,3,2,2))
# Leeres 3x3 Array
x4 = np.empty((3,3))
# 4x5 Array mit Zufallszahlen
x5 = np.random.random((4,4))
```

Wir können jedoch auch eine Reihe von Werten erzeugen, welche gleichmäßig verteilt sind. Dafür benutzen wir die Funktionen *linspace()* und *arange()*.

Die Funktion *linspace()* erzeugt eine Liste mit einer bestimmten Anzahl an Werten, welche alle gleichmäßig verteilt, zwischen zwei Zahlen liegen.

```
# 10 gleichmäßig verteilte Zahlen von 20 bis 35
x2 = np.linspace(20, 35, 10)
```

Hier erstellen wir eine Liste mit 10 Zahlen, welche, gleichmäßig verteilt, zwischen 20 und 35 liegen.

Die Funktion *arange()* erzeugt eine Liste von Werten zwischen zwei Zahlen, in einem bestimmten Schritttempo.

```
# Zahlen von 10 bis 50 in 5er Schritten
x1 = np.arange(10, 50, 5)
```

In diesem Codebeispiel erzeugen wir eine Liste mit Werten zwischen 10 und 50, in 5er-Schritten.

EIGENSCHAFTEN VON ARRAYS

Wir können uns mit Numpy, auch einige interessante Eigenschaften, von unseren neuen Arrays, ansehen. Neben den standardmäßigen Funktionen und Attributen von Sequenzen, liefert das Modul nämlich auch einiges selbst.

Beispielsweise können wir uns, mit der dem Attribut *shape*, die Form bzw. die Struktur eines Arrays ausgeben lassen.

```
a = np.array([
   [
      [10,20,30,40], [8,8,2,1], [1,1,1,2]
   ],
   [
      [9, 9, 2, 39], [1,2,3,3], [0,0,3,2]
   ],
   [
      [12,33,22,1], [22,1,22,2], [0,2,3,1]
   ]
])

print(a.shape)
```

In diesem Fall wäre die Ausgabe *(3,3,4)*, was bedeutet, dass wir 3 Arrays, mit jeweils 3 Arrays, mit jeweils 4 Werten haben. Wir können uns nun auch, durch das Attribut *ndim*, ausgeben lassen, wie viele Dimensionen unser Array hat.

```
print(a.ndim)
```

Hier sind das natürlich drei. Weiters können wir uns, durch das Attribut *size*, anzeigen lassen, wie viele Elemente das Array hat.

```
print(a.size)
```

Bei uns wären das nun 36 Elemente. Zu guter Letzt können wir auch noch, mit dem Attribut *dtype*, einsehen, welchen Datentypen unsere Elemente haben.

```
print(a.dtype)
```

Hier ist der Datentyp int32.

MATHEMATISCHE OPERATIONEN

Wir können auf unsere Arrays in Numpy diverse mathematische Operationen anwenden. Beispielsweise können wir ein Array mit einem Skalar (einfache Zahl) addieren, subtrahieren, multiplizieren und dividieren, oder mit einem anderen Array.

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

print(a + 2)
# = [[3,6], [7,5]]
print(a - 2)
# = [[-1,2], [3,1]]
print(a * 2)
# = [[2,8], [10,6]]
print(a / 2)
# = [[1.5,2.], [2.5,1.5]]
```

Wie wir hier sehen, wird die Rechenoperation immer auf jedes Element individuell angewandt.

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

print(a+b)
# = [[4,6],[9,7]]
print(a-b)
# = [[-2,2],[1,-1]]
print(a*b)
# = [[3,8],[20,12]]
print(a/b)
# = [[0.3333,2.],[1.25,0.75]]
```

Wenn wir zwei Arrays, derselben Struktur, miteinander addieren, multiplizieren usw., so rechnen wir dabei immer mit den zwei Zahlen, welche sich auf dem jeweils selben Index befinden.

Wir können jedoch auch ganze mathematische Funktionen auf Arrays anwenden.

```
# e hoch a
print(np.exp(a))
# Sinus von a
print(np.sin(a))
# Cosinus von a
print(np.cos(a))
# Tangens von a
print(np.tan(a))
# Logarithmus von a
print(np.log(a))
# Wurzel von a
print(np.sqrt(a))
```

Diese Funktionen werden auch wieder auf die einzelnen Elemente im Array angewandt.

AGGREGATFUNKTIONEN

Aggregatfunktionen sind Funktionen, welche Eigenschaften von einer Menge an Daten zusammenfassen. Diese können wir auch auf unsere Arrays bzw. auf unsere Listen anwenden.

Mit der Funktion *sum()* können wir zum Beispiel alle Werte unseres Arrays aufsummieren.

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

print(a.sum())
# = 1 + 4 + 5 + 3 = 13
```

Die Funktionen *min()* und *max()* liefern uns jeweils den niedrigsten und den höchsten Wert des Arrays.

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

print(a.min()) # = 1
print(a.max()) # = 5
```

Durch die Methoden *mean()* und *np.median()* können wir uns auch die statistischen Größen des arithmetischen Mittels und des Medians liefern lassen.

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

print(a.mean()) # = 3.25
print(np.median(a)) # = 3.5
```

Auch die Standardabweichung können wir mit *np.std()* ermitteln.

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

print(np.std(a)) # = 1.479...
```

INDEXIEREN VON ARRAYS

Das Indexieren von Numpy-Arrays erfolgt genauso wie das, von ordinären Python-Listen.

```
a = np.array([
    [[1,2,3], [8,8,2], [0,2,1]],
    [[1,1,1], [2,3,1], [8,2,5]],
    [[0,0,2], [0,2,8], [7,7,7]]
])
```

Wir haben hier ein 3x3x3 Array. Wenn wir also eine bestimmte Stelle ansprechen wollen, so müssen wir drei Indizes angeben. Der erste Index gibt an, welche der drei Reihen ich ansprechen möchte, der zweite welche Spalte und der dritte welche Zahl in der Liste. Die Nummerierung beginnt bei der Zahl Null. Hier ein paar Beispiele:

```
print(a[0][0][0]) # = 1
print(a[1][2][0]) # = 8
print(a[2][2][1]) # = 7
print(a[1][1][2]) # = 1
print(a[2][2][2]) # = 7
```

Wir können jedoch auch weniger Indizes angeben, wenn wir ganze Arrays zurückgeliefert bekommen möchten.

```
print(a[0][0]) # = [1,2,3]
print(a[2]) # = [[0,0,2], [0,2,8], [7,7,7]]
print(a[1][2]) # = [8,2,5]
```

Das ist dann nützlich, wenn wir bestimmte Operationen nur auf Teile des Arrays anwenden möchten. Beispielsweise um den maximalen Wert der dritten Reihe zu finden oder das arithmetische Mittel der zweiten Reihe zu ermitteln.

```
print(a[2].max())
print(a[1].mean())
```

ARRAYS MANIPULIEREN

In Numpy haben wir zahlreiche Möglichkeiten unsere Arrays zu manipulieren. Beispielsweise können wir mit einer einzigen Funktion, die Dimensionen unseres Arrays umtauschen. Dafür benutzen wir die Funktion *np.transpose()*.

```
a = np.array([
    [[1,2,3], [8,8,2], [0,2,1]],
    [[1,1,1], [2,3,1], [8,2,5]],
    [[0,0,2], [0,2,8], [7,7,7]]
])

a = np.transpose(a)

print(a)
```

Aus

```
[[[1 2 3] [8 8 2] [0 2 1]]
 [[1 1 1] [2 3 1] [8 2 5]]
 [[0 0 2] [0 2 8] [7 7 7]]]
```

wird dann

```
[[[1 1 0] [8 2 0] [0 8 7]]
 [[2 1 0] [8 3 2] [2 2 7]]
 [[3 1 2] [2 1 8] [1 5 7]]]
```

Wir können auch jedes beliebige Array in eine eindimensionale Liste umwandeln, mit der Funktion *ravel()*.

```
a = np.array([
    [[1,2,3], [8,8,2], [0,2,1]],
    [[1,1,1], [2,3,1], [8,2,5]],
    [[0,0,2], [0,2,8], [7,7,7]]
])
```

```
a = a.ravel()

print(a)
```

Das Ergebnis sieht dann wie folgt aus:

[1 2 3 8 8 2 0 2 1 1 1 2 3 1 8 2 5 0 0 2 0 2 8 7 7 7]

Wollen wir die Dimensionen eines Arrays manuell ändern, so müssen wir darauf achten, dass unsere Zieldimensionen dieselbe Größe haben, wie unsere aktuellen. Dann können wir sie mit der Funktion *reshape()* ändern. Die Werte werden dadurch nicht beeinträchtigt.

```
a = np.array([
    [[1,2,3], [8,8,2], [0,2,1]],
    [[1,1,1], [2,3,1], [8,2,5]],
    [[0,0,2], [0,2,8], [7,7,7]]
])

a = a.reshape(3,9)
# 3x9 = 3x3x3

print(a)
```

Da 3x9 und 3x3x3 beides 27 ergibt, ist diese Umformung zulässig. Das Array sieht dann so aus:

```
[[1 2 3 8 8 2 0 2 1]
 [1 1 1 2 3 1 8 2 5]
 [0 0 2 0 2 8 7 7 7]]
```

Was wir in Numpy ebenfalls tun können, ist mehrere Arrays aufeinander oder nebeneinander legen und zu einem zusammenfügen. Das nennt man dann Stacken. Dies können wir vertikal oder horizontal tun.

```
a = np.array([1,5,3,59,12,3,8,77])
b = np.array([4,12,2,4,92,13,28,87])
c = np.array([7,3,1,7,10,11,7,12])
```

```
arr = np.vstack((a,b,c))
print(arr)
```

Hier haben wir drei Arrays, derselben Länge, welche wir übereinander stapeln und zu einem Array zusammenfügen. Dieses schaut dann so aus:

```
[[ 1  5  3 59 12  3  8 77]
 [ 4 12  2  4 92 13 28 87]
 [ 7  3  1  7 10 11  7 12]]
```

Diese drei Arrays nebeneinander zu legen, würde darin resultieren, dass wir ein langes eindimensionales Array haben, welches alle Werte hintereinander aufreiht.

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

b = np.array([[4,3],
              [5,7],
              [4,2]])

arr = np.hstack((a,b))
print(arr)
```

Hier macht das Ganze schon deutlich mehr Sinn. Wir haben zwei 3x2 Arrays, welche wir nebeneinander legen. Das Resultat ist folgendes:

```
[[1 2 4 3]
 [2 3 5 7]
 [6 5 4 2]]
```

Anstatt Arrays nur zusammenzufügen, können wir diese auch aufteilen bzw. splitten. Auch das geht wieder vertikal und horizontal. Als Beispiel teilen wir unser soeben zusammengefügtes Array wieder auf.

```
a = np.array([[1,2,4,3],
              [2,3,5,7],
              [6,5,4,2]])
```

```
arr = np.hsplit(a,2)

a1 = arr[0]
a2 = arr[1]

print(a1)
print(a2)
```

Wir splitten das Array hier horizontal an der zweiten Stelle. Das Resultat ist dann folgendes:

```
[[1 2] [2 3] [6 5]]


[[4 3] [5 7] [4 2]]
```

Die Arrays sind hier in einer Reihe aufgeschrieben, damit es kompakter ist. Sie werden aber untereinander aufgelistet.

Dasselbe können wir nun auch vertikal machen.

```
a = np.array([[1,2,3,4,5,6,7,8,9],
              [5,5,5,4,4,4,3,3,3],
              [7,7,7,7,7,7,7,7,7],
              [9,4,3,2,7,4,6,3,1]])

arr = np.vsplit(a,2)

a1, a2 = arr[0], arr[1]

print(a1)
print(a2)
```

Wir splitten hier ein 4x9 Array, an der zweiten Zeile, vertikal auf. Dann erhalten wir zwei 2x9 Arrays, welche so aussehen:

```
[[1 2 3 4 5 6 7 8 9]
 [5 5 5 4 4 4 3 3 3]]
```

```
[[7 7 7 7 7 7 7 7 7]
 [9 4 3 2 7 4 6 3 1]]
```

DATEN SPEICHERN UND LADEN

Zu guter Letzt schauen wir uns an, wie wir in Numpy Daten in Dateien schreiben und aus diesen lesen können. Dabei können wir ein Numpy-Format benutzen oder einfach Text- bzw. CSV-Dateien verwenden.

Numpy-Format

Wenn wir ein Array auf die Platte speichern wollen, so benötigen wir dafür die Funktion *np.save()*.

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

np.save('meinarray',a)
```

Wenn wir das ausführen, wird das Array a in die Datei *meinarray.npy* gespeichert. Um diese Daten wieder zu laden, benötigen wir die Funktion *np.load()*.

```
b = np.load('meinarray.npy')
```

Wir können jedoch auch mehrere Arrays in eine Datei speichern und komprimieren. Dafür benutzen wir die Methode *np.savez()*.

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

np.savez('arrays', a=a, b=b)
```

Das Ganze wird dann in eine .npz Datei gespeichert. Um die Arrays wieder zu laden, müssen wir den jeweiligen Schlüssel angeben.

```
data = np.load('arrays.npz')
arr1 = data['a']
arr2 = data['b']
```

CSV-Dateien

Wollen wir nun Daten in CSV- oder Textdateien speichern, so benötigen wir die Funktion *np.savetxt()*.

```
a = np.array([[10,20,30,40,50],
              [4,3,6,7,8],
              [3,3,3,3,3],
              [1,2,3,4,9]])

np.savetxt('array.csv', a, delimiter=",")
```

Wichtig ist hierbei, dass ein Array, welches in eine CSV-Datei geschrieben wird, nicht mehr als zwei Dimensionen haben darf, da es sonst nicht tabellarisch dargestellt werden kann.

Um Daten aus einer CSV-Datei generieren zu lassen benötigen wir die Funkton *np.genfromtxt()*.

```
a = np.genfromtxt('array.csv', delimiter=",")
```

5 – DIAGRAMME MIT MATPLOTLIB

Bisher haben wir uns nur mit dem Erzeugen, Verarbeiten, Manipulieren, Speichern und Laden der Daten beschäftigt. Mit Matplotlib können wir nun unsere Daten, Sachverhalte, Punkte und Funktionen auch grafisch darstellen.

Um mit Matplotlib arbeiten zu können, müssen wir das Modul zunächst importieren. Um genau zu sein, werden wir hauptsächlich mit PyPlot, einem Untermodul von Matplotlib arbeiten.

```
import matplotlib.pyplot as plt
```

FUNKTIONEN ZEICHNEN

Um in Matplotlib eine Funktion zu zeichnen, müssen wir uns, mit Numpy, zunächst einmal einige Werte definieren.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
```

Hier generieren wir 100 Werte von 0 bis 10. Diese werden die X-Werte unserer Funktion darstellen. Um nun unsere Funktion zu definieren, müssen wir einfach ein neues Array erstellen, mit den dazugehörigen Y-Werten.

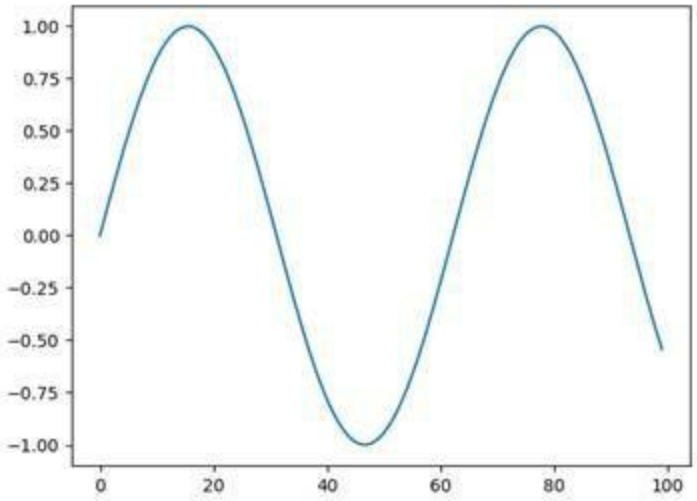
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = np.sin(x)
```

In diesem Fall ist unsere Funktion *y* einfach eine Sinusfunktion. Nun da wir die X- und Y-Werte haben, müssen wir PyPlot sagen, dass es diese Funktion zeichnen soll. Anschließend müssen wir auch noch angeben, dass der fertige Graph angezeigt werden soll.

```
plt.plot(y)
plt.show()
```

Das Resultat von unserem Code, sieht dann wie folgt aus:



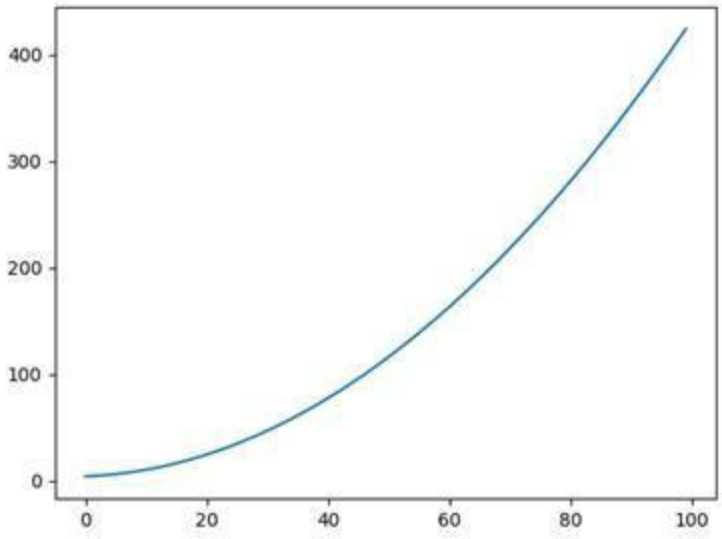
Wir können unsere Funktion beliebig gestalten. Hier ist ein weiteres Beispiel:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = 4 * x ** 2 + 2 * x + 4

plt.plot(y)
plt.show()
```

Hier ist unsere Funktion $4x^2+2x+4$ und wird von PyPlot auch dementsprechend gezeichnet. Das Resultat sieht so aus:


400
300
200
100
0
0
20
40
60
80
100

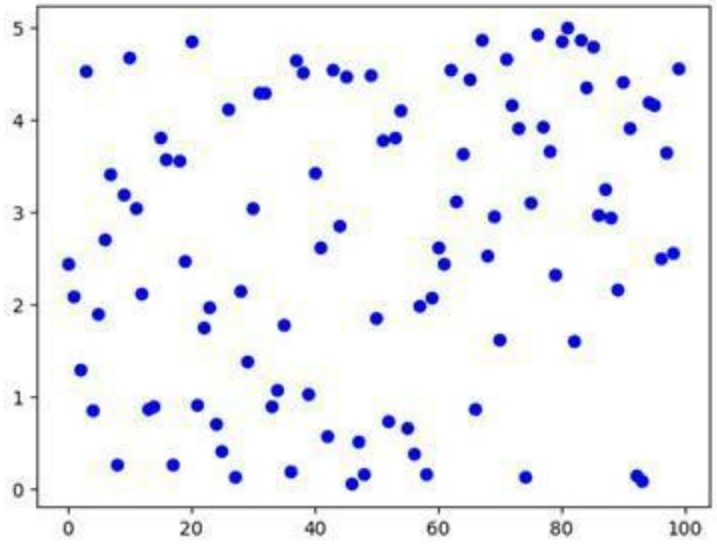
WERTE GRAFISCH DARSTELLEN

Neben Funktionen, können wir mit PyPlot auch Werte und deren Verteilung grafisch darstellen.

```
import numpy as np
import matplotlib.pyplot as plt

x = 5 * np.random.random(100)
plt.plot(x, 'bo')
plt.show()
```

Wir lassen uns hier zunächst 100 Zufallszahlen generieren, welche wir dann mit 5 multiplizieren. Dann zeichnen wir diese und geben bei der Funktion *plot()* noch einen Parameter, für die Farbe und für die Art des Diagramms, an. In unserem Fall steht *bo* für blau und Punktediagramm. Wir könnten auch *ro* hinschreiben und wir hätten ein rotes Punktediagramm. Das Ergebnis sieht so aus:



MEHRERE GRAPHEN IN EINEM FENSTER

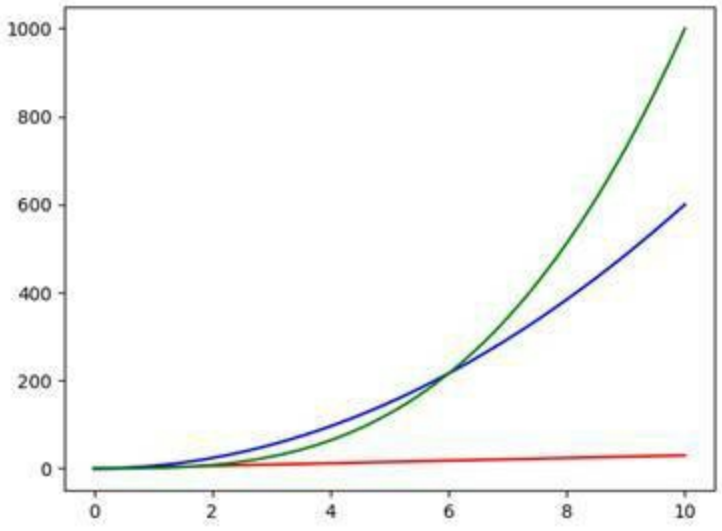
Mit der Funktion *plot()* können wir beliebig viele Graphen auf einmal zeichnen, da die Funktion unendlich viele Parameter übernehmen kann. Die Funktionen bzw. Graphen werden alle in einem Fenster gezeichnet.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,10,100)
lin = 3 * x
quad = 6 * x ** 2
cub = x ** 3

plt.plot(x, lin, 'r', x, quad, 'b', x, cub, 'g')
plt.show()
```

Wir erstellen uns hier wieder 100 X-Werte von 1 bis 10. Diese benutzen wir dann um eine lineare, eine quadratische und eine kubische Funktion zu definieren. Dann übergeben wir der Funktion plot() immer zuerst den X-Wert, gefolgt von dem Y-Wert und der Farbe und Art des Diagramms. Das Resultat sieht folgendermaßen aus:



Dasselbe können wir natürlich auch mit anderen Diagrammarten machen.

MEHRERE DIAGRAMME IN EINEM FENSTER

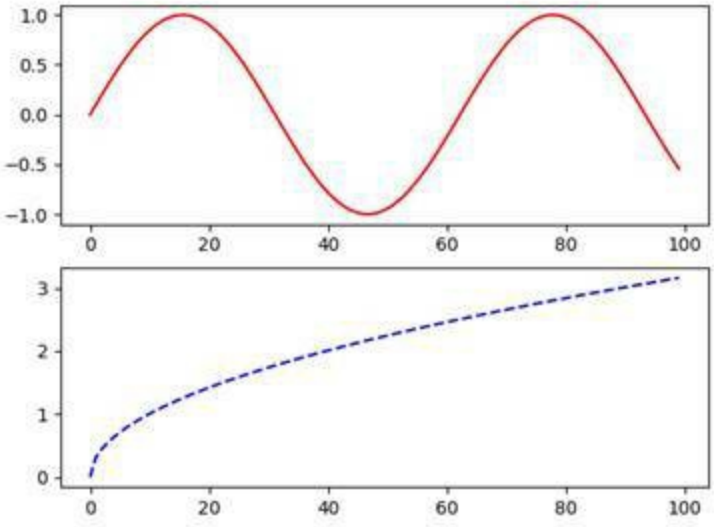
Wir können mit Matplotlib nicht nur mehrere Funktionen bzw. Graphen in einem Diagramm, sondern auch mehrere Diagramme in einem einzigen Fenster haben. Diese nennt man dann Subplots und man nummeriert sie ab 211 aufwärts.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.sqrt(x)

plt.subplot(211)
plt.plot(y1, 'r')
plt.subplot(212)
plt.plot(y2, 'b--')
plt.show()
```

Wir erstellen hier wieder zwei Funktionen, nämlich y1, welche eine Sinusfunktion darstellt und y2, welche eine Wurzelfunktion darstellt. Nun erklären wir dem Interpreter, mit der Funktion subplot(), welchen Subplot wir gerade bearbeiten. Dann zeichnen wir y1 mit einer roten Linie und y2 mit einer blauen strichlierten Linie. Das sieht dann so aus:



Man beachte hierbei, dass die beiden x- und y-Achsen verschiedene Maßstäbe haben.

MEHRERE FENSTER GLEICHZEITIG

Wenn wir in Matplotlib mehrmals hintereinander *plot()* ausführen, so wird immer ein Fenster erst dann geöffnet, wenn das vorherige geschlossen wird. Wollen wir nun mehrere Fenster gleichzeitig starten, so müssen wir *figures* definieren.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y1 = x ** 2
y2 = 5 * x

plt.figure(1)
plt.plot(y1)
plt.figure(2)
plt.plot(y2)
plt.show()
```

Dadurch, dass wir hier zwei *figures* plotten, bevor wir die Funktion *show()* ausführen, werden beide gleichzeitig ausgegeben. Wenn wir hingegen. einfach zweimal hintereinander *plot()* ausführen, überschreibt sich das.

BESCHRIFTUNG DER DIAGRAMME

Um übersichtliche Diagramme zu erstellen, müssen wir diese ordentlich beschriften und gestalten. Dafür stellt Matplotlib uns zahlreiche Funktionen zur Verfügung.

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 172, 4
x = mu + sigma * np.random.randn(10000)

plt.hist(x, 50, normed=1, facecolor='darkblue')

plt.xlabel('Groesse')
plt.ylabel('Wahrscheinlichkeit')
plt.title('Groesse von Schuelern')
plt.text(160, .125, r'$\mu=172,\ \sigma=4$')
plt.axis([155, 190, 0, 0.15])
plt.grid(True)
plt.show()
```

Wir erstellen hier ein Diagramm einer Normalverteilung mit dem Mittelwert 172 und der Standardabweichung 4. Diese beschreibt die Größen der Schüler. Zunächst lassen wir uns einige Zufallswerte erzeugen. Dann erzeugen wir ein Histogramm mit unseren Werten in dunkelblauer Farbe. Mit den label-Funktionen beschriften wir die X- und die Y-Achse. Einen Titel setzen wir mit der Funktion title(). Mit der axis() Funktion bestimmen wir den Wertebereich auf den Achsen. In diesem Fall also 155 bis 190 auf der X-Achse und 0 bis 0.15 auf der Y-Achse. Einen Text können wir mit der Funktion text() erstellen und diesen auch beliebig positionieren. Zu guter Letzt haben wir dann noch die Funktion grid(), welche bestimmt ob wir ein Gitter haben oder nicht. Das Endresultat sieht dann so aus:

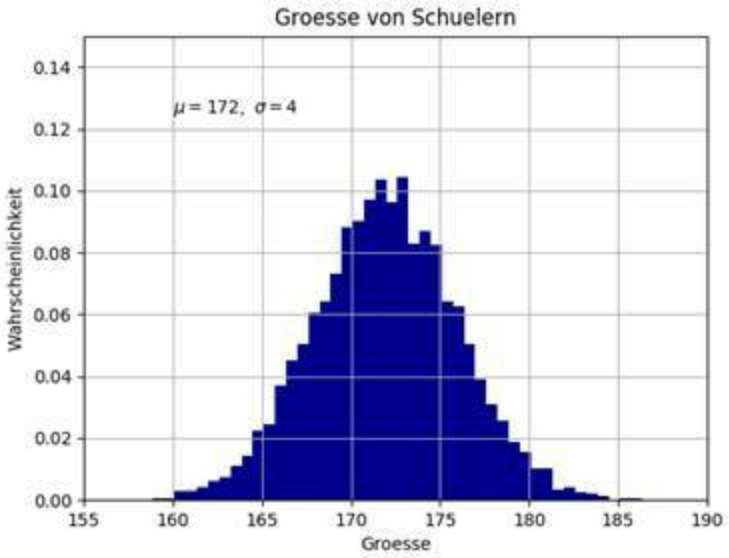

Groesse von Schuelern
$\mu = 172,\ \sigma = 4$
Wahrscheinlichkeit
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
155
160
165
170
175
180
185
190
Groesse

DIAGRAMME SPEICHERN

Wir können unsere erstellten Diagramme nun auch auf die Platte speichern, in Form von Bildern.

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 172, 4
x = mu + sigma * np.random.randn(10000)

plt.hist(x, 50, normed=1, facecolor='darkblue')

plt.xlabel('Groesse')
plt.ylabel('Wahrscheinlichkeit')
plt.title('Groesse von Schuelern')
plt.text(160, .125, r'$\mu=172,\ \sigma=4$')
plt.axis([155, 190, 0, 0.15])
plt.grid(True)
plt.savefig('diagramm.png')
plt.show()
```

Durch die Funktion *savefig()* wird das Diagramm unserer Normalverteilung nun als *diagramm.png* abgespeichert.

6 – STATISTIK MIT MATPLOTLIB

Da wir uns im Bereich Data Science bewegen, spielt Statistik eine essentielle Rolle. Zusammen mit Numpy und Matplotlib, können wir zahlreiche statistische Berechnungen automatisiert ausführen.

BOXPLOT

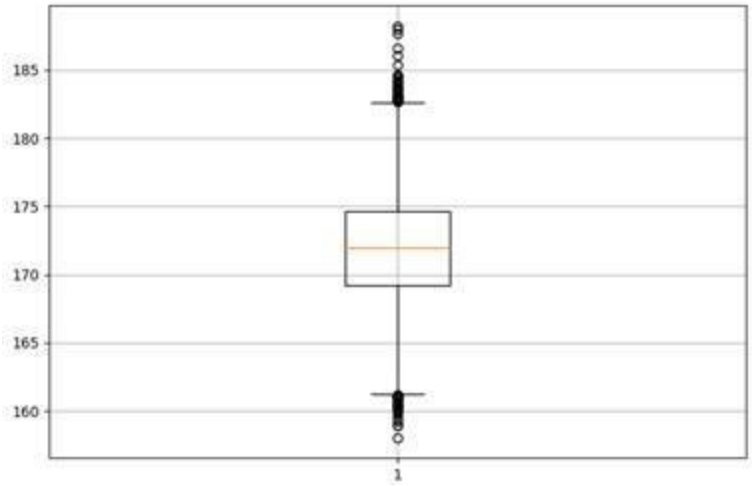
Zum Beispiel können wir, aus einer Menge von Daten ein Boxplot-Diagramm zeichnen lassen. Nehmen wir hier doch einfach die Größen unserer Schüler.

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 172, 4
x = mu + sigma * np.random.randn(10000)

plt.boxplot(x)
plt.grid(True)
plt.show()
```

Hier haben wir dieselben Werte, wie bei dem Histogramm, nur dass wir hier, einen Boxplot zeichnen. Dieser schaut dann so aus:



REGRESSIONSFUNKTION

Was wir ebenfalls tun können, ist, für eine Menge von Punkten eine Regressionsfunktion zu finden.

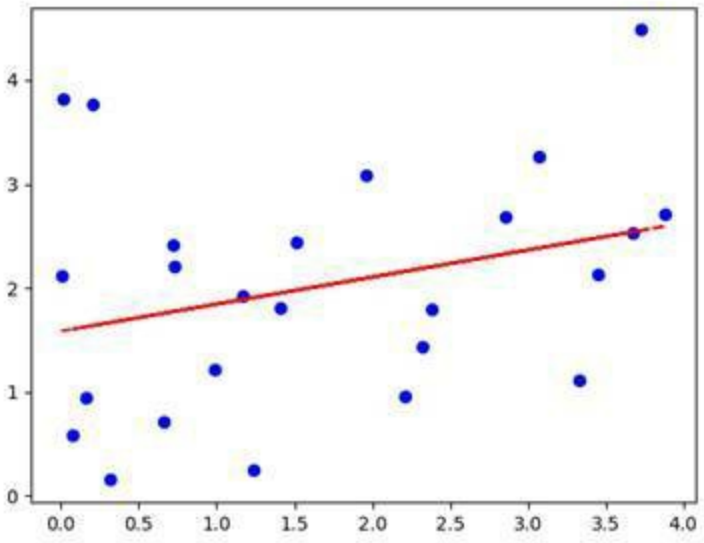
```
import numpy as np
import matplotlib.pyplot as plt

x = 5 * np.random.random(25)
y = 5 * np.random.random(25)

reg = np.polyfit(x,y,1)
regf = np.poly1d(reg)

plt.plot(x, y, 'bo', x, regf(x), 'r--')
plt.show()
```

Wir lassen uns hier zufällige X- und Y-Werte generieren, für welche wir dann eine Annäherungsfunktion suchen. Um die Werte dieser Funktion dann auch tatsächlich in einer Polynomfunktion 1.Grades (linear) umzuwandeln, benötigen wir die Funktion poly1d(). Dann zeichnen wir einmal, als Punkte, unsere Zufallswerte und dann als rot strichlierte Linie unsere Regressionsgerade. Das Ergebnis sieht dann wie folgt aus:



HISTOGRAMME

Wie bereits bei der Beschriftung vorgeführt, können wir in PyPlot auch Histogramme zeichnen, um Wahrscheinlichkeitsverteilungen darzustellen.

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 172, 4
x = mu + sigma * np.random.randn(10000)

plt.hist(x, 50, normed=1, facecolor='darkblue')
plt.show()
```

KREISDIAGRAMME

Wir können mit PyPlot auch die Verteilung unserer Werte in Form eines Kreisdiagramms darstellen.

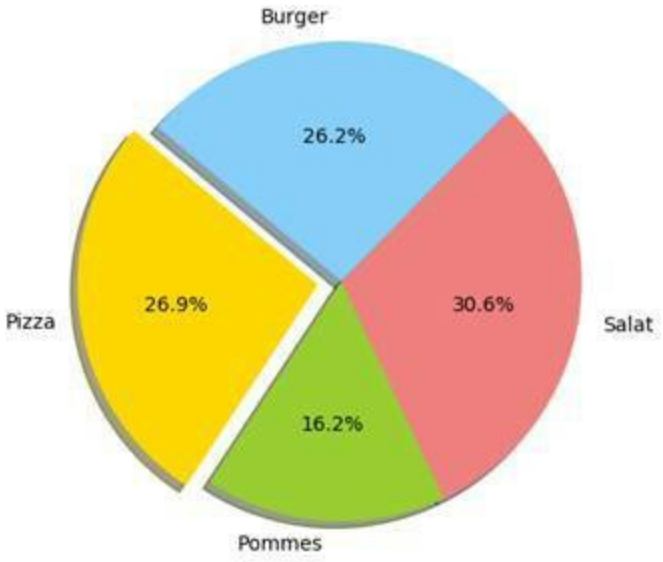
```
import matplotlib.pyplot as plt

label = 'Pizza', 'Pommes', 'Salat', 'Burger'
groessen = [215, 130, 245, 210]
farben = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
explode = (0.1, 0, 0, 0)

plt.pie(groessen, explode=explode, labels=label, colors=farben,
        autopct='%1.1f%%', shadow=True, startangle=140)

plt.axis('equal')
plt.show()
```

In diesem Fall haben wir vier Mahlzeiten, welche man als Lieblingsessen angeben kann. Dann haben wir eine Liste mit den jeweiligen Zahlen und eine mit den Farben für das Diagramm. Mit der Funktion *pie()* können wir nun aus diesen Listen ein Kreisdiagramm erzeugen. Dieses schaut dann so aus:



7 – PANDAS

Mit Pandas werden wir uns hier nur sehr oberflächlich befassen. Es ist ein Modul, welches große Datenmengen übersichtlich formatieren, darstellen und verarbeiten soll. Die Datenstrukturen, auf welchen das Ganze primär aufbaut, sind DataFrames und Series. Bevor wir jedoch beginnen mit Pandas zu arbeiten, müssen wir das Modul erst einmal importieren:

```
import pandas as pd
```

SERIES IN PANDAS

In Pandas gibt es eine Datenstruktur, namens Series, welche eine Reihe darstellt. Es ist im Grunde genommen einfach nur ein eindimensionales Array. Eine Series erzeugen wir folgendermaßen:

```
s = pd.Series([3,2,9,3], index=['a', 'b', 'c', 'd'])
```

Wir haben hier eine Reihe mit vier Zahlen, welche jeweils durch die Buchstaben a, b, c und d indexiert sind. Wenn wir diese ausgeben, bekommen wir folgendes Ergebnis:

```
a  3
b  2
c  9
d  3
```

Um nun unsere Daten aus der Series abzurufen, können wir die Indizes verwenden.

```
s = pd.Series([3,2,9,3], index=['a', 'b', 'c', 'd'])
print(s['a']) # = 3
```

Wir können uns auch gleich auf einmal mehrere Werte liefern lassen.

```
s = pd.Series([3,2,9,3], index=['a', 'b', 'c', 'd'])
print(s[['a', 'c']]) # = 3, 9
```

DATAFRAMES

DataFrames sind zwei-dimensionale Datenstrukturen in Pandas, welche sehr stark einer Tabelle ähneln. Sie hat Spalten und Reihen. Folgendermaßen erzeugen wir ein DataFrame aus bereits vorhandenen Daten:

```
daten = {'Name': ['Alex', "Benni", 'Claudia'],
        'Alter': [19, 18, 21],
        'Groesse': [183, 178, 163]}

df = pd.DataFrame(daten, columns=['Name', 'Alter', 'Groesse'])
```

Wir haben hier prinzipiell einmal ein Dictionary mit Daten. Diese Daten übergeben wir unserem DataFrame, sodass wir dann mit ihnen effizient arbeiten können. Als Spaltennamen (wie in Tabellen bei Datenbanken), legen wir Name, Alter und Größe fest. Unser Output sieht wie folgt aus:

```
      Name  Alter  Groesse
0     Alex     19      183
1    Benni     18      178
2  Claudia     21      163
```

Da wir keine Indizes festgelegt haben, wird standardmäßig mit Zahlen von 0 ab aufwärts indexiert.

Sortieren von DataFrames

Wir können unsere DataFrames auch sortieren lassen und dabei selbst angeben, nach welchen Kriterien das geschehen soll.

```
df = df.sort_values("Alter")
```

In diesem Fall sortieren wir unsere Daten nach Alter. Der Output sieht deshalb nun wie folgt aus:

```
      Name  Alter  Groesse
1    Benni     18      178
```

0 Alex 19 183

2 Claudia 21 163

Speichern und Lesen in CSV und Excel

Unsere DataFrames und Series können wir auch wieder in CSV- und Excel-Dateien speichern und aus diesen laden. Mit der Funktion *to_csv()* schreiben wir den Inhalt unseres DataFrames in eine CSV-Datei.

```
df.to_csv('daten.csv')
```

Wollen wir diese nun wieder ins Programm laden, so benötigen wir die Funktion *pd.read_csv()*.

```
df = pd.read_csv('input.csv')
```

Wenn wir hierbei mit Excel-Dateien arbeiten möchten, so müssen wir zwei Module installieren, nämlich *openpyxl* und *xlrd*. Dies können wir ganz einfach mit pip machen:

```
pip install openpyxl
```

```
pip install xlrd
```

Wenn wir diese Module installiert haben, so können wir mit der Funktion *to_excel()* unsere Daten in ein XLSX-File schreiben.

```
df.to_excel('daten.xlsx', sheet_name='Daten1')
```

Hier speichern wir unser DataFrame in eine Datei namens *daten.xlsx* und zwar auf das Sheet, welches wir *Daten1* nennen. Wollen wir die Daten aus dieser Datei nun wieder laden, so benötigen wir die Funktion *pd.read_excel()*.

```
df = pd.read_excel('input.xlsx')
```

8 – ANWENDUNG: AKTIENHANDEL

Für das was wir in diesem Buch gelernt haben, gibt es zahlreiche Anwendungsbereiche. Wir können mit Pandas, Matplotlib und Numpy Daten von Aktienkursen aus dem Internet laden und diese dann automatisiert analysieren. Um unsere Daten jedoch beziehen zu können, benötigen wir das *pandas-datareader* Modul.

```
import pandas as pd
import pandas_datareader as web
import datetime
import matplotlib.pyplot as plt
```

Dann können wir uns beispielsweise den Aktienkurs von Apple zeichnen lassen, indem wir die Daten aus der Yahoo Finance API beziehen.

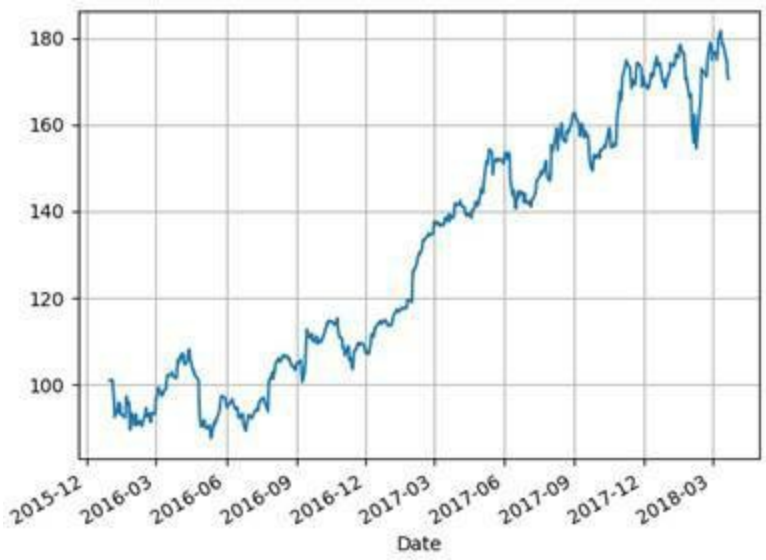
```
start = datetime.datetime(2016,1,1)
end = datetime.date.today()

apple = web.DataReader("AAPL", "yahoo", start, end)

apple["Adj Close"].plot(grid = True)
plt.show()
```

Wir starten unseren Graphen beim Datum 01.01.2016 und ziehen es ihn bis heute durch. Dann laden wir die Daten, mittels DataReader, in unsere Variable. Diese Daten sind nun ein DataFrame, von welchem wir immer den Closing-Preis der Aktie wissen möchten. Daher wählen wir diese Spalte aus und zeichnen einen Graphen. Diesen lassen wir dann anzeigen.

Warnung: Hin und wieder passiert es, dass das Skript beim Versuch die Daten zu laden abstürzt. In diesem Fall einfach ein paar Mal neu starten bis es funktioniert!



Hier könnten wir nun natürlich eine Trendlinie legen um zu sehen, wie sehr der Kurs in der Regel steigt.

```
import pandas as pd
import pandas_datareader as web
import datetime
import matplotlib.pyplot as plt
import numpy as np

start = datetime.datetime(2016,1,1)
end = datetime.date.today()

apple = web.DataReader("AAPL", "yahoo", start, end)

x = list(range(0, len(apple.index.tolist()), 1))
closing = apple["Adj Close"]

date_x = apple.index

reg = np.polyfit(x,closing,1)
regf = np.poly1d(reg)

plt.grid(True)
plt.plot(date_x, closing, 'b', date_x, regf(x), 'r')
plt.show()
```

Der Code kann im ersten Moment etwas verwirrend sein, doch er ist im Grunde genommen ganz simpel. Wir laden hier einfach die Daten von der Apple-Aktie seit dem 01.01.2016 runter. Von diesen speichern wir zunächst die Closing-Werte in ein Array. Zusätzlich speichern wir auch die Werte des

Index, in ein Array für die x-Werte. Außerdem speichern wir diese nochmal in ein Datumsarray für unsere x-Werte. Danach stellen wir eine Regressionsfunktion auf, welche dieselben Werte erhält wie der Graph für den Aktienkurs. Anschließend zeichnen wir beide Graphen, was im Endeffekt so aussieht:

PYTHON
FÜR FINANZEN

FLORIAN DEDOV

PYTHON FÜR FINANZEN

DER SCHNELLE EINSTIEG

Von

FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

Wer auf Dauer Vermögen aufbauen will, muss investieren. Wer sich mit Data Science und Machine Learning auseinandersetzen will, der kommt nur schwer an Python vorbei. Heutzutage werden Finanzanalysen nicht mehr in Excel-Sheets durchgeführt. Man benutzt moderne Technologien und wenn man gut ist, bastelt man sich eigene Tools. Mit Python haben Sie ein mächtiges und simples Werkzeug für Ihre eigenen Analysen.

Es ist also mehr als sinnvoll sich in diesem Bereich massiv weiterzubilden, sodass Sie ein Teil dieser modernen Entwicklung sein können anstatt von ihr überrannt zu werden.

In diesem Buch wird Ihnen die Finanzanalyse in Python nähergebracht. Es ist von Vorteil wenn Sie sich bereits ein wenig mit Python auskennen und auch grundlegendes Finanzwissen wäre nicht schlecht. Hier wird nicht auf die grundlegende Syntax oder auf simple Konzepte eingegangen. Sollten Sie also das Gefühl haben, dass Sie mit Python noch nicht so gut vertraut sind, empfehle ich Ihnen, sich zunächst meine Bücher für Anfänger und für Fortgeschrittene anzusehen. Diese finden Sie auf meiner Amazon-Seite.

Nachdem Sie dieses Buch gelesen haben und das Gelernte auch nebenbei angewandt haben, werden Sie in der Lage sein, grundlegende Konzepte von der Finanzanalyse in Python zu verstehen und anzuwenden. Sie können dann große Mengen an Finanzdaten sammeln, strukturieren, visualisieren und analysieren. Außerdem werden wir auf interessante Bereiche wie Data Science und Machine Learning eingehen.

Sie sollten dieses Buch jedoch nicht nur schnell durchlesen und versuchen oberflächlich zu verstehen, worum es geht, sondern auch aktiv nebenbei mitprogrammieren. Dadurch lernen Sie am meisten und das Wissen wird am besten gefestigt. In den folgenden Kapiteln werden Sie zahlreiche Codebeispiele finden, die Sie nachprogrammieren können. Sie können jedoch auch Ihren eigenen Code schreiben, mithilfe der Dinge die Sie bereits gelernt

haben, zu dem Zeitpunkt.

Im Grunde genommen bleibt es Ihnen selbst überlassen, wie Sie diese Lektüre handhaben möchten. Das Buch ist kompakt und simpel gehalten, sodass Sie es rein theoretisch innerhalb von wenigen Stunden, aber auch innerhalb von mehreren Tagen lesen können. In jedem Fall wünsche ich Ihnen maximalen Erfolg beim Programmieren und viel Spaß beim Lernen von Data Science in Python.

Falls Sie nach dem Lesen dieses Buches, der Meinung sind, dass es positiv zu Ihrer Programmierkarriere beigetragen hat, würde es mich sehr freuen, wenn Sie eine Rezension auf Amazon hinterlassen. Danke!

1 – WARUM PYTHON?

Python ist in den letzten Jahren zu einer der führenden Programmiersprachen geworden. Wenn wir mal einen Blick auf den TIOBE-Index (https://www.tiobe.com/tiobe-index/) werfen, welcher die Popularität von Programmiersprachen aufreiht, sehen wir, dass Python die drittpopulärste Sprache nach Java und C ist (Stand: Februar 2019). Diese haben jedoch verschiedene Anwendungsbereiche. Java fokussiert sich in erster Linie auf Enterprise-Applikationen und Webapps, während C primär zur hardwarenahen Programmierung eingesetzt wird, bei welcher man eine sehr hohe Komplexität und viel Spielraum hat.

Für die Bereiche Machine Learning, Data Science, Künstliche Intelligenz und natürlich auch Finanzen, ist Python die führende Sprache. Konkurrierende Sprachen sind R und teilweise MATLAB, welche jedoch zum einen noch lange nicht so weit verbreitet sind wie Python und zum anderen, auch eher ihre Anwendung hauptsächlich nur im mathematisch, statistischen Bereich finden. Python hingegen ist unglaublich vielseitig und mächtig. Zudem ist die Sprache wahnsinnig simpel und einfach zu erlernen.

Durch das vielseitige Angebot an Bibliotheken wie Numpy, Matplotlib und Scikit-Learn, haben wir ein mächtiges Arsenal an Modulen, mit welchen wir in kürzester Zeit hochkomplexe und mächtige Analysen durchführen können.

2 – INSTALLATION DER MODULE

In diesem Buch, werden wir Daten laden, verarbeiten, darstellen und auch mit Machine Learning neue Dinge aus den Daten lernen. Dafür benötigen wir mindestens folgende Bibliotheken:

- Pandas
- Pandas-Datareader
- Matplotlib
- Numpy
- Scikit-Learn
- Beautifulsoup4

Gehen wir einmal die einzelnen Module durch, betrachten ihre Funktionalitäten und schauen uns an, wie wir diese installieren.

PANDAS

Pandas ist eine der meistbenutzten Python-Bibliotheken. Sie bietet uns eine vielseitige und mächtige Datenstruktur, nämlich das *DataFrame*. Dieses kann man sich vorstellen, wie eine Excel-Tabelle, aus welcher wir Daten gezielt auslesen und einlesen können. Falls Sie mit SQL vertraut sind, so könnten Sie es sich als Tabelle in einer Datenbank vorstellen.

Um Pandas zu installieren geben wir in unserer Kommandozeile (CMD oder Terminal) folgendes ein:

```
pip install pandas
```

PANDAS-DATAREADER

Um unsere Finanzdaten aus dem Internet zu laden, benutzen wir den Pandas-Datareader. Dieser kann gezielt Informationen aus der Yahoo Finance API in ein DataFrame laden.

Installation: pip install pandas-datareader

MATPLOTLIB

Diese Bibliothek liefert und zahlreiche Werkzeuge, mit welchen wir die Finanzdaten graphisch darstellen können. Liniendiagramme, Kreisdiagramme, 3D-Darstellungen und vieles mehr, sind in diesem Modul enthalten.

Installation: pip install matplotlib

NUMPY

Numpy werden wir als solches meistens gar nicht manuell verwenden. Es wird jedoch von Pandas und Matplotlib benutzt, da es uns ermöglicht, effizient mit Matrizen und größeren Datenmengen zu agieren.

Installation: pip install numpy

SCIKIT-LEARN

Dieses Modul liefert uns eine Fülle an Machine-Learning Algorithmen, mit welchen wir aus unseren Daten lernen und neue Daten vorhersagen können.

Installation: pip install scikit-learn

BEAUTIFULSOUP4

Zugegeben, der Name der Bibliothek ist etwas verwirrend. Es handelt sich hierbei jedoch um ein Modul, welches uns ermöglicht, Daten aus Webseiten zu *scrapen*. Wir können also hierbei gezielt Informationen aus HTML-Code herausfiltern.

Installation: pip install beautifulsoup4

3 – FINANZDATEN LADEN

So nun da wir alle Bibliotheken installiert haben, werden wir für dieses Kapitel zunächst jene importieren, welche wir benötigen, um unsere ersten Finanzdaten zu laden.

```
from pandas_datareader import data as web
import datetime as dt
```

Hier importieren wir zunächst aus dem Modul *pandas_datareader* das Untermodul *data*. Wir geben diesem auch einen *Alias*, also einen zweiten Namen, nämlich *web*, damit wir es leichter ansprechen können. Außerdem ist es die Konvention. Wichtig ist hierbei, dass wir bei dem Modul einen Unterstrich, und nicht wie bei der Installation einen Bindestrich, verwenden.

Des Weiteren importieren wir das Modul *datetime*, welches in dem Standard-Python enthalten ist. Auch diesem geben wir ein Alias und zwar *dt*.

Im Folgenden werden wir die Daten von dem Unternehmen *Apple* von der Yahoo Finance API laden.

```
from pandas_datareader import data as web
import datetime as dt

# Zeitraum festlegen
start = dt.datetime(2017,1,1)
ende = dt.datetime(2019,1,1)

# AAPL (Apple) Daten von Yahoo Finance laden
df = web.DataReader('AAPL', 'yahoo', start, ende)

# Daten ausgeben
print(df.head())
```

Im ersten Schritt legen wir zwei Daten fest. Einmal das Start- und einmal das Enddatum. Wir wollen also die Daten der Aktie, in diesem Zeitraum laden.

Dann erstellen wir eine Variable *df*, welche das *DataFrame* beinhalten soll, das durch den *DataReader* geladen wird. Als Parameter bekommt dieser

zunächst das Ticker-Symbol übergeben. Das zweite ist dann die API, die in unserem Fall jene von Yahoo ist. Und zu guter Letzt kommen die beiden Daten, welche den Zeitraum eingrenzen.

Anschließend geben wir, mit der Funktion *head*(), die ersten fünf Zeilen aus.

Warnung: Es kann hin und wieder passieren, dass die Yahoo Finance API nicht antwortet und daher eine Fehlermeldung kommt. In diesem Fall liegt es nicht am Code und kann meistens durch erneutes Versuchen oder Warten behoben werden.

Wir bekommen hier ein DataFrame mit sieben Spalten, wobei die erste, also das Datum, den Index darstellt. Der Wert von dem Index ist immer einzigartig und identifiziert die jeweilige Zeile. Die anderen sechs Werte, sind die für uns relevanten Finanzwerte: *Open*, *High*, *Low*, *Close*, *Adj Close* und *Volume*.

Open: Welchen Wert die Aktie an dem Tag hatte, als die Börse geöffnet hat.

Close: Welchen Wert die Aktie an dem Tag hatte, als die Börse geschlossen hat.

High: Der höchste Wert an dem jeweiligen Tag.

Low: Der niedrigste Wert an dem jeweiligen Tag.

Adj Close: Steht für *Adjusted Close* und ist dasselbe wie der Close, nur mit Rücksicht auf Dinge wie Stock-Splits usw.

Volume: Anzahl an Aktien, welche an diesem Tag den Besitzer gewechselt haben, also verkauft und gekauft wurden.

EINZELNE WERTE AUSLESEN

Wenn wir uns nur für den *Close* Wert interessieren, so können wir auch nur diesen, zusammen mit dem Datum ausgeben.

```
print(df['Close'])
```

Wir arbeiten hierbei, wie mit einem Dictionary. Wenn wir nun den reinen Zahlenwert, an einem bestimmten Datum haben möchten, können wir das wie folgt tun:

```
print(df['Close']['2017-02-14'])
```

Was jedoch auch geht ist, dass wir zum Beispiel das fünfte Element aus der Tabelle ausgeben:

```
print(df['Close'][5])
```

4 – DATEN SPEICHERN UND LESEN

Wir können diese Daten nicht nur aus dem Internet herunterladen, sondern auch in Dateien schreiben und aus Dateien auslesen.

CSV

Um unser DataFrame, mit allen Informationen, in eine CSV-Datei zu speichern, benutzen wir die Funktion *to_csv()*.

```
# DataFrame in eine CSV speichern
df.to_csv("apple.csv")
```

Den Inhalt der Datei können wir uns anschließend mit einem Editor, oder mit Excel ansehen. Standardmäßig werden die Einträge durch Kommas getrennt. Wenn wir das ändern wollen, können wir einen *separator* mit dem Schlüsselwort *sep* angeben.

```
df.to_csv("apple.csv", sep=';')
```

EXCEL

Wenn wir unsere Daten direkt in ein Excel-File schreiben möchten, tun wir das mit der Funktion *to_excel()*.

```
df.to_excel('apple.xlsx')
```

Das Ganze sieht dann so aus:

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2017-01-03 0:00:00	115.8	116.33	114.76	116.15	112.14	28781900
3	2017-01-04 0:00:00	115.85	116.51	115.75	116.02	112.0145	21118100
4	2017-01-05 0:00:00	115.92	116.86	115.81	116.61	112.5841	22193600
5	2017-01-06 0:00:00	116.78	118.16	116.47	117.91	113.8392	31751900
6	2017-01-09 0:00:00	117.95	119.43	117.94	118.99	114.882	33561900
7	2017-01-10 0:00:00	118.77	119.38	118.3	119.11	114.9978	24462100
8	2017-01-11 0:00:00	118.74	119.93	118.6	119.75	115.6157	27588600
9	2017-01-12 0:00:00	118.9	119.3	118.21	119.25	115.133	27086200
10	2017-01-13 0:00:00	119.11	119.62	118.81	119.04	114.9302	26111900
11	2017-01-17 0:00:00	118.34	120.24	118.22	120	115.8571	34439800

Abb. 1: DataFrame in Excel

HTML

Auch in HTML können wir unsere Daten abbilden. Dies geschieht mit der Funktion *to_html()*.

```
df.to_html('apple.html')
```

Das Resultat ist dann einfach eine Tabelle auf einer HTML-Seite.

JSON

Zu guter Letzt schauen wir uns auch noch das JSON-Format an, da dieses oftmals in JavaScript, aber auch in anderen Sprachen benutzt wird. Hier benutzen wir die Funktion *to_json*.

```
df.to_json('apple.json')
```

DATEN AUS DATEIEN LESEN

Sollten wir in die Situation kommen, dass wir die Daten nicht von der Yahoo Finance API zur Verfügung gestellt bekommen, sondern aus Dateien lesen müssen, so können wir auch auf diese Weise mit Pandas arbeiten.

Hierzu müssen wir jedoch das Modul *pandas* explizit importieren:

```
import pandas as pd
```

Nun können wir aus allen zuvor besprochenen Datentypen auch Daten auslesen. Egal ob CSV, Excel, HTML oder JSON. Hierzu gibt es immer eine dementsprechende *read* Funktion, welche ein DataFrame für *pandas* zurückgibt.

```
df = pd.read_csv("apple.csv", sep=";")
df = pd.read_excel("apple.xlsx")
df = pd.read_html("apple.html")
df = pd.read_json("apple.json"
```

Auch beim Einlesen können wir einen Separator angeben, bei der CSV-Datei, damit keine Fehler passieren.

5 – GRAPHISCHE DARSTELLUNG

So eine Tabelle ist ja schön und gut, doch um ein Gesamtbild, über die Kursentwicklung zu erlangen, müssen wir das Ganze graphisch darstellen.

Dazu benötigen wir nun die Bibliothek *matplotlib* bzw. um genau zu sein, die Klasse *pyplot*. Diese importieren wir wie folgt:

```
import matplotlib.pyplot as plt
```

Auch hier verwenden wir wieder ein Alias, um uns die Arbeit zu erleichtern.

DIAGRAMM ZEICHNEN

Um nun ein einfaches Diagramm der Werte zu zeichnen, benutzen wir die Funktion *plt()*.

```
# Daten zeichnen und Diagramm anzeigen
df['Adj Close'].plot()
plt.show()
```

Wir wählen hier also den gewünschten Wert, aus dem Dataframe und zeichnen diesen. Diesmal entscheiden wir uns für den *Adjusted Close* Wert, da dieser aussagekräftiger ist, weil er mehrere Faktoren berücksichtigt.

Die Funktion *show()* ist wichtig, um das gezeichnete Diagramm dann letzten Endes auch anzuzeigen. Das Resultat schaut dann so aus:

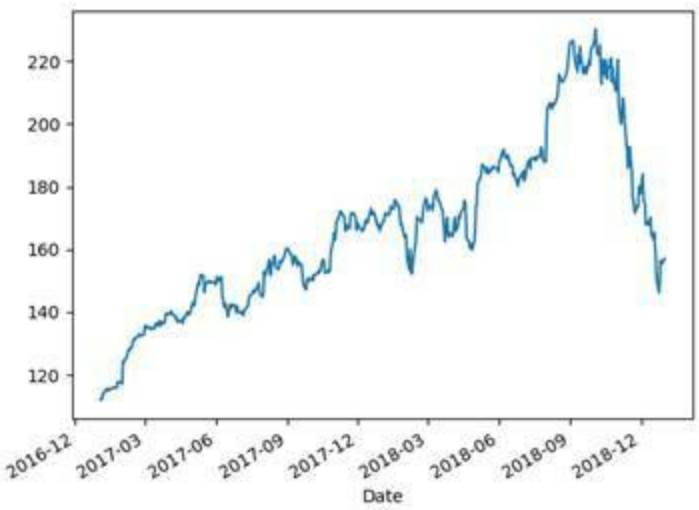


Abb. 2: Aktienkurs visualisiert

Wie wir sehen, erkennt Matplotlib sofort, dass die Daten auf die X-Achse gehören und die Adjusted Close Werte auf die Y-Achse. Wir sparen uns Unmengen an Arbeit.

DIAGRAMM GESTALTEN

Wir haben mit Matplotlib auch die Möglichkeit unsere Diagramme zu gestalten. Wollen wir zum Beispiel ein Gitter aktivieren, können wir das wie folgt tun:

```
plt.grid(True)
```

Wichtig ist, dass wir diese Spezifikation noch vor dem *show()* Befehl angeben.

Was uns ebenso möglich ist, ist, einen Stil für unsere Diagramme anzugeben. Dafür müssen wir jedoch ein Untermodul aus Matplotlib importieren, nämlich *style*.

```
from matplotlib import style
```

Dann können wir mit der *use()* Funktion, unseren gewünschten Stil anwenden. Einer der beliebtesten ist zum Beispiel *ggplot*.

```
style.use('ggplot')
```

Das Ganze sieht dann so aus:

Abb. 3: Diagramm mit ggplot-Stil

Es gibt in Matplotlib eine Vielzahl an verschiedenen Stilen. Einige interessante sind folgende:

- dark_background
- classic
- fivethirtyeight
- seaborn

Experimentieren Sie einfach ein wenig mit den Stilen herum und finden Sie jenen, welcher Ihnen am besten gefällt.

Um die Namen aller verfügbaren Stile anzuzeigen, führen Sie folgenden Code aus:

```
print(plt.style.available)
```

6 – ANALYSE UND KENNZAHLEN

Nun haben wir gelernt, wie wir Finanzdaten herunterladen, auslesen, speichern und graphisch darstellen können. Daher werden wir uns jetzt mit der Analyse und Manipulation der Daten befassen.

100 DAY MOVING AVERAGE

Wir werden direkt mit einem Beispiel einsteigen, nämlich mit der Kennzahl des *100 Day Moving Averages*. Diese Zahl ist relativ simpel. Wir nehmen die Werte der letzten 100 Tage her und berechnen das arithmetische Mittel bzw. den Durchschnitt.

Diesen Wert werden wir dann in unser DataFrame einbinden und ihn mit dem jeweiligen Tageskurs vergleichen.

Dafür müssen wir zunächst eine neue Spalte erstellen. Pandas macht dies automatisch, sobald wir dem Spaltennamen Werte zuweisen. Sprich, wir müssen sie nicht wirklich erstellen, sondern können sie gleich mit Werten befüllen.

```
df['100ma'] = df['Adj Close'].rolling(window = 100, min_periods = 0).mean()
```

Diese Zeile mag auf den ersten Blick verwirrend wirken, da einige neue Funktionen vorhanden sind. Doch diese sind nicht sonderlich komplex.

Wir erstellen hier eine neue Spalte, welche, für jede Zeile in der Tabelle, die vorherigen 100 Werte nimmt und aus diesen den Durchschnitt bildet.

Die Funktion *rolling()* sammelt eine bestimmte Anzahl an vorherigen Werten, um eine gewisse statistische Berechnung zu ermöglichen. Wie viele das sind, wird durch den Parameter *window* angegeben. In unserem Fall 100.

Der Parameter *min_periods* gibt an, wie viele Werte mindestens vorhanden sein müssen, um eine Berechnung durchzuführen. Das ist relevant, weil die

ersten Datensätze unseres DataFrames nicht hundert andere vor sich haben, aus welchen ein Mittel berechnet werden kann. Mit der Zahl Null sagen wir, dass die Berechnung von Anfang an erfolgt, selbst wenn keine vorherigen Werte vorhanden sind. Das führt dazu, dass der erste Eintrag, sich selbst als Mittelwert nimmt, der zweite das Mittel aus den ersten beiden berechnet, der dritte aus den ersten dreien und so weiter, bis zu einem Maximum von 100 Werten.

Zu guter Letzt wenden wir auf unsere gesammelten Werte, eine statistische Funktion an. Wir haben uns hier für *mean()*, also den Mittelwert, entschieden. Es gibt jedoch auch andere, wie *max()*, *min()* oder *median()*.

NAN WERTE

Falls wir uns für eine Mindestanzahl entscheiden, bei dem Parameter *min_periods*, wird es zwangsläufig zu Datensätzen mit einem NaN-Wert kommen. Das steht für *Not a Number*, also *keine Zahl*. Um diese Einträge dann auch unserem DataFrame zu löschen, gibt es die Funktion *dropna()*.

```
df.dropna(inplace=True)
```

Diese entfernt einfach alle Zeilen, welche in irgendeiner Spalte einen NaN-Wert haben. Der Parameter *inplace*, sorgt dafür, dass das aktuelle DataFrame durch diese Funktion ersetzt wird. Wir sparen uns damit die erneute Zuweisung.

```
df.dropna(inplace=True)
df = df.dropna()
```

Diese beiden Zeilen haben die exakt gleiche Funktion. Generell wird aber die Schreibweise mit *inplace* als sauberer und professioneller angesehen.

Wir können uns nun unsere Resultate ansehen.

```
print(df.head())
```

```
                High         Low  ...   Adj Close      100ma
```

```
Date                                    ...
2017-03-15  140.750000  139.029999  ...  136.198685  125.108436
2017-03-16  141.020004  140.259995  ...  136.421692  125.330265
2017-03-17  141.000000  139.889999  ...  135.742950  125.530509
2017-03-20  141.500000  140.229996  ...  137.168365  125.750091
2017-03-21  142.800003  139.729996  ...  135.597504  125.932450
```

Offensichtlich hat alles gut funktioniert, doch wirklich übersichtlich ist das Ganze nicht. Um das zu ändern, werden wir uns eine graphische Übersicht verschaffen.

GRAPHISCHE DARSTELLUNG DES 100MA

Um eine schöne und informative Übersicht, über unsere ganzen Daten und Kennzahlen zu erlangen, werden wir nun unsere *Adjusted Close*, unser *100ma* und unsere *Volumes* alle gleichzeitig graphisch darstellen.

Wir haben dann also unseren Kurs im Vergleich mit dem 100-Tages-Mittel und gleichzeitig noch die Information, wie viele Aktien an einem Tag den Besitzer gewechselt haben.

Hierzu werden wir sogenannte *Subplots* verwenden. Das bedeutet, wir werden in einem Fenster zwei Diagramme darstellen. Diese Subplots werden in Python in der Regel als *Axes* (also Achsen) bezeichnet. Warum auch immer. Wir werden diese Konvention hier so beibehalten.

```
ax1 = plt.subplot2grid((6,1),(0,0),rowspan=5, colspan=1)
ax2 = plt.subplot2grid((6,1),(5,0),rowspan=1, colspan=1, sharex=ax1)
```

Wir definieren hier also zwei Subplots, mit der Funktion *subplot2grid()*. Dieser übergeben wir einige Parameter. Das erste Zahlenpaar gibt an, wie viele Zeilen und Spalten das Fenster haben soll. In diesem Fall sechs Zeilen und eine Spalte. Das zweite Zahlenpaar gibt die Startposition des jeweiligen Subplots an.

(0, 0) ist die Position ganz oben links. (5, 0) ist die fünfte Zeile ganz links. Mit den Parametern *rowspan* und *colspan* geben wir an, wie viele Zeilen

(rows) und Spalten (columns) unser Suplot besetzen wird, also wie groß es sein wird.

Der letzte Parameter *sharex* gibt an, dass sich unser Subplot *ax2*, die X-Achse mit dem Subplot *ax1* teilt. Sprich, wenn wir das eine Bewegen oder reinzoomen, tun wir das, auf der X-Achse auch bei dem anderen.

```
ax1.plot(df.index, df['Adj Close'])
ax1.plot(df.index, df['100ma'])
ax2.bar(df.index, df['Volume'])
```

Jetzt zeichnen wir auf unser Suplot *ax1* unseren Kurs, zusammen mit unserem *100ma*. Auf *ax2* stellen wir unser tägliches Volumen dar.

Wichtig ist zu bemerken, dass wir bei dem Kurs und bei unserer Kennzahl, ein normales Liniendiagramm, mit dem Befehl *plot()*, zeichnen. Bei unserem Volumen jedoch, benutzen wir die Funktion *bar()*, für ein Balkendiagramm, weil das hier einfach mehr Sinn macht.

Beide Funktionen übernehmen als Parameter zunächst die X- und dann die Y-Werte. Unser X-Wert ist immer der Index, also unser Datum. Und die jeweiligen Y-Werte sind der Kurs, die Kennzahl und das Volumen.

Mit einem *show()* sieht unser Diagramm nun so aus:

Abb. 4: Subplots mit Kurs, 100ma und Volumen

WEITERE KENNZAHLEN

Natürlich können wir auf diese Weise noch zahlreiche andere Kennzahlen berechnen, darstellen und vergleichen.

Wir könnten zum Beispiel eine Spalte machen, welche uns immer zeigt, um wie viel Prozent ein Kurs, an dem jeweiligen Tag, gefallen oder gestiegen ist.

```
df['change'] = (df['Close'] - df['Open']) / df['Open']
```

Hier ziehen wir den Close-Wert, vom Open-Wert ab und dividieren das Ergebnis durch den Open-Wert. Das Resultat ist die Änderung pro Tag. Wenn wir diese noch mal 100 multiplizieren, bekommen wir das Ergebnis in Prozent.

Eine andere Idee wäre es, die Differenz zwischen High und Low im Vergleich mit dem Close-Wert zu betrachten. Dadurch bekommen wir ein Gefühl für die tägliche Volatilität.

```
df['hl_pct'] = (df['High'] - df['Low']) / df['Close']
```

Je nachdem was Sie hier interessiert und was für Sie nützlich ist, können Sie sich allerhand statistische Berechnungen einfallen lassen. Diese können Sie dann, genauso wie vorhin gezeigt, auch graphisch darstellen.

7 – CANDLESTICK CHARTS

In diesem Kapitel werden wir noch tiefer in die Statistik und die graphische Darstellung eintauchen. Wir werden aus unseren *Adjusted Close* Werten, für zehn Tage immer den Anfangswert (Open), den höchsten Wert (High), den niedrigsten Wert (Low) und den Endwert (Close) ermitteln. Diese werden wir dann alle vier gemeinsam in einem Candlestick-Diagramm abbilden.

Hierfür werden wir jedoch noch ein neues Modul installieren und zwei importieren müssen.

pip install mpl-finance

Das Modul *mpl_finance* werden wir benötigen, um das Candlestick Chart zu zeichnen. Wir müssen also zusätzlich folgende Bibliotheken importieren:

```
import matplotlib.dates as mdates
from mpl_finance import candlestick_ohlc
```

Die *dates* von Matplotlib, welche wir als *mdates* importieren, liefern uns das nötige Format für unsere Candlesticks. Die Funktion *candlestick_ohlc* zeichnet für uns dann die Werte in der Reihenfolge OHLC (Open, High, Low Close). Das ist wichtig, da das Modul auch eine OCHL Funktion hat. Diese wollen wir nicht wählen.

RESAMPLING

Bevor wir irgendetwas graphisch darstellen können, müssen wir zunächst unsere *Adjusted Close* Werte so aufbereiten, dass wir immer die vier Werte für die jeweiligen zehn Tage haben. Das nennt man Resampling.

```
df_ohlc = df['Adj Close'].resample('10D').ohlc()
```

Wir erstellen hier ein neues DataFrame namens *df_ohlc*. Dieses wird dann unsere vier Werte beinhalten. Zunächst überarbeiten wir unsere Daten jedoch mit der Funktion *resample()*. Als Parameter übergeben wir dieser, den

gewünschten Zeitraum. In diesem Fall *10D* für zehn Tage.

Damit haben wir unsere *Adjusted Close* Werte für immer jeweils zehn Tage zusammengefügt. Daraus bilden wir jetzt mit der Funktion *ohlc()* unsere gewünschten vier Werte.

Um das Volumen nicht zu vernachlässigen, werden wir wieder zwei Subplots zeichnen lassen und das Volumen für die zehn Tage aufsummieren.

```
df_volume = df['Volume'].resample('10D').sum()
```

Auch hier benutzen wir auf dieselbe Weise die *resample()* Funktion. Hier arbeiten wir jedoch mit *sum()*, da wir nur an der Summe der Volumen interessiert sind.

DATUMSFORMAT

Da das *mpl_finance* Modul leider mit dem Datumsformat von Matplotlib arbeitet und nicht mit *datetime*, müssen wir einige Änderungen an unserem DataFrame vornehmen.

```
df_ohlc.reset_index(inplace=True)
df_ohlc['Date'] = df_ohlc['Date'].map(mdates.date2num)
```

Zunächst erzeugen wir einen künstlichen Index (Nummerierung) und machen *Date* zu einer gewöhnlichen Spalte. Dies tun wir mit der Funktion *reset_index()*.

Nun müssen wir mit der Funktion *map()* unsere Daten im *datetime* Format austauschen. Dazu benutzen wir wiederum die Funktion *date2num* aus Matplotlib. Diese wandelt unsere Daten in Zahlen um, welche *mpl_finance* versteht.

CANDLESTICK CHART

Nun haben wir unsere Daten so aufbereitet, dass wir sie graphisch darstellen können. Wir erstellen uns also wieder zwei Subplots.

```
ax1 = plt.subplot2grid((6,1),(0,0),rowspan=5, colspan=1)
ax2 = plt.subplot2grid((6,1),(5,0),rowspan=1, colspan=1, sharex=ax1)
ax1.xaxis_date()
```

Hier sehen wir diesmal die Funktion *xaxis_date()*, welche einfach nur dafür sorgt, dass auf der X-Achse immer das jeweilige Datum abgebildet ist.

```
candlestick_ohlc(ax1, df_ohlc.values, width=5, colorup='g', colordown='r')

ax2.fill_between(df_volume.index.map(mdates.date2num), df_volume.values)
```

Bei unserer Candlestick-Funktion, geben wir hier fünf Parameter an. Die ersten beiden sind unsere X- und Y-Werte (ax1 und Values). Dann geben wir die Breite in Pixel an, welche unsere Candlesticks haben sollen. Zu guter Letzt legen wir dann noch die Farben fest, welche für das Diagramm benutzt werden sollen. Der Wert *colorup* ist die Farbe, wenn der Kurs steigt und der Wert *colordown*, jene Farbe, wenn der Kurs fällt.

Für unser Volumen benutzen wir die Funktion *fill_between()*. Diese hat einfach den Effekt, dass wir die Zwischenräume, zwischen den Balken ausfüllen. Dieser übergeben wir ebenso die X- und die Y-Werte. Da wir für dieses DataFrame das Datum noch nicht konvertiert haben, müssen wir das hier noch tun (genau wie bei dem ersten DataFrame).

Nach einem Aufruf von *plt.show()* können wir dann das Endergebnis begutachten:

Abb. 5: Candlestick Chart

Jeder diese Candlesticks enthält nun die Informationen, über *Open*, *High*, *Low* und *Close*. Es empfiehlt sich das Gitter einzuschalten, um die Daten

besser ablesen zu können.

```
ax1.grid(True, color='black')
```

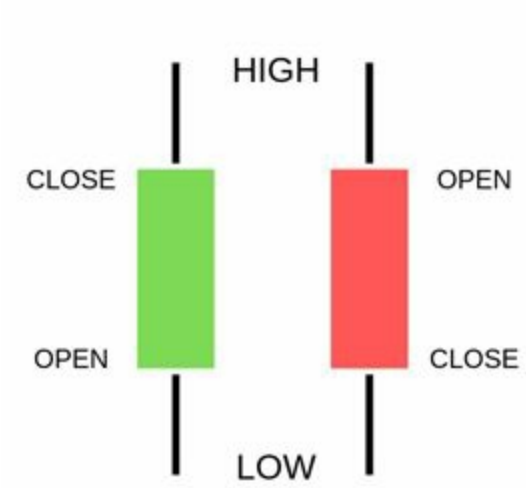


Abb. 6: Candlesticks im Detail

Anhand dieser Grafik sehen Sie, wie Candlesticks aufgebaut sind. Das oberste Ende ist der High-Wert. Sein Gegenstück, der Low-Wert, ist am untersten Ende zu finden. Je nachdem ob der Kurs an dem Tag gestiegen (grün) oder gefallen (rot) ist, finden wir den Close-Wert oben und den Open-Wert unten, oder umgekehrt.

8 – S&P 500 INDEX

Nun wissen wir also, wie wir von einem Unternehmen Daten laden, speichern, analysieren und gut darstellen können. Wenn wir nun jedoch größere Berechnungen durchführen und sehr oft, sehr viele Daten brauchen, ist es nicht vorteilhaft, jedes Mal die Yahoo Finance API dafür zu benutzen.

Was wir daher machen können ist, uns die Daten, der 500 Unternehmen, welche im S&P 500 vertreten sind, für einen bestimmten Zeitraum, einmalig herunterzuladen und unsere Daten ab dann immer von dort zu beziehen. Der S&P 500 ist ein Index, welcher die 500 größten börsennotierten Unternehmen der USA zusammenfasst.

WEBSCARPING

Die Yahoo Finance API bietet keine Funktion, welche es uns ermöglicht auszulesen, welche Unternehmen in diesem Index enthalten sind. Deswegen werden wir nun auf eine neue Technologie zurückgreifen, nämlich *Webscraping*.

Dabei lesen wir gezielt jene Daten, aus dem HTML-Code einer Webseite, aus, welche wir benötigen. Für dieses Beispiel werden wir die S&P 500 Unternehmen Seite von Wikipedia verwenden, und zwar die englische, da dieser wahrscheinlich öfter aktualisiert wird.

Link: https://en.wikipedia.org/wiki/List_of_S%26P_500_companies

Auf dieser Seite sehen wir direkt zu Beginn eine Tabelle mit den gewünschten Daten. Dort haben wir einmal den Namen des Unternehmens in der ersten Spalte, das Ticker-Symbol (welches wir für die Finance API benötigen) in der zweiten Spalte und noch viele weitere Informationen.

Um nun zu verstehen wie wir Daten mit einem Webscraper auslesen können, müssen wir uns zunächst den HTML Code der Tabelle ansehen. Wir machen einen Rechtsklick und lassen uns den *Seitenquelltext anzeigen*.

Dort sehen wir, dass wir dort ein *table* Element haben. In dieser Tabelle haben wir Reihen (*tr*) und in diesen Reihen haben wir wieder Daten (*td*). Die Inhalte dieser können wir also filtern.

DATEN AUSLESEN

Für das Webscraping mit Python werden wir das Modul *beautifulsoup4* verwenden. Dieses ermöglicht es uns, gezielt Daten aus HTML-Seiten auszulesen. Installiert haben wir es bereits, folglich müssen wir es nur noch importieren. Ebenso werden wir die Bibliothek *requests* von Python benötigen, um erst einmal an den HTML-Code zu gelangen.

```
Import bs4 as bs
import requests
```

Zunächst werden wir mit einem Request den Wikipedia Quelltext in ein Objekt laden.

```
Response = requests.get('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
```

Wir erstellen einfach ein Objekt namens *response* und laden dann mit *requests.get()* den Inhalt der Seite hinein.

Dann müssen wir ein sogenanntes Soup-Objekt erstellen, um dann den Inhalt *parsen* (also zergliedern) zu können.

```
Soup = bs.BeautifulSoup(response.text, 'lxml')
```

Wir übergeben hier das *text* Attribut von unserem Response Objekt, als ersten Parameter. Als zweiten übergeben wir dann den gewünschten Parser. Da müssen wir nicht sonderlich viel verstehen. Hier wählen wir *lxml*. Es kann jedoch sein, dass Sie diesen vielleicht nicht auf Ihrem Computer haben. In diesem Fall können Sie ihn wie gewohnt mit *pip* installieren.

Wenn wir das haben, definieren wir nun ein Objekt *tabelle*, welches dann nur den gewünschten Teil enthält.

```
Tabelle = soup.find('table', {'class':'wikitable sortable'})
```

Wir benutzen die Funktion *find* um in unserer *Soup* nach allen *tables* zu suchen, welche die Klassen *wikitable* und *sortable* haben. Die Klassenangabe

dient nur zur Absicherung, falls es noch andere Tabellen auf der Seite geben sollte.

Als nächstes erstellen wir eine Liste für unsere Ticker-Symbole, lesen diese aus und laden Sie hinein.

```
Tickers = []

for reihe in tabelle.findAll('tr')[1:]:
    ticker = reihe.findAll('td')[1].text
    tickers.append(ticker)
```

Mit der *findAll()* Funktion bekommen wir alle Elemente, welche ein *tr* sind, also alle Reihen. Für jede dieser Reihen (außer der ersten, wegen dem Header) lesen wir jetzt den Text aus der zweiten Spalte (also dem zweiten *td*, wobei der Index 1 ist, da wir von 0 aufwärts zählen) aus. Dieser Text ist das Ticker-Symbol. Dieses speichern wir und laden es dann in unsere Liste. Dann geben wir noch die Werte zum Überprüfen aus und zurück. So schaut das Gesamte dann aus:

```
import bs4 as bs
import requests

# Funktion für das Auslesen der Ticker-Symbole
def lade_sp500_ticker():

    # Wikipedia Quelltext laden
    response = requests.get('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies')

    # Soup Object erstellen um mit lxml Objecte parsen zu können
    # lxml muss eventuell installiert werden
    soup = bs.BeautifulSoup(response.text, 'lxml')

    # Finde Tabellen mit Klassen wikitable und sortable
    tabelle = soup.find('table')


    # Leere Liste für Ticker-Symbole
    tickers = []

    # Für jede Tabellenreihe wird der Text der zweiten Spalte gespeichert
    for reihe in tabelle.findAll('tr')[1:]:
        ticker = reihe.findAll('td')[1].text
        tickers.append(ticker)

    # Zum Testen
    print(tickers)

    return tickers

lade_sp500_ticker()
```

Wie wir sehen, haben wir unseren Code hier als Funktion definiert, damit wir diese aufrufen können, wenn wir sie brauchen. Unser Programm wird nämlich noch ein wenig erweitert.

SERIALISIERUNG

Damit wir unsere Ticker-Liste nicht immer wieder aufs Neue scrapen müssen, werden wir sie lokal abspeichern und dann immer auslesen, wenn wir sie brauchen. Das tun wir indem wir sie serialisieren. Bei der Serialisierung speichern wir ein Objekt, samt aktuellem Zustand, in eine Datei. Dann können wir es in genau jenem Zustand wieder laden, wann wir wollen.

In Python benutzen wir hierzu die Bibliothek *pickle*. Diese importieren wir wie gewohnt.

```
import pickle
```

Wir erweitern nun unsere Funktion um zwei Zeilen. Das tun wir nach der For-Schleife, aber vor dem *return*.

```
with open("sp500tickers.pickle", 'wb') as f:
    pickle.dump(tickers, f)
```

Hier öffnen wir die Datei *sp500tickers.pickle* im Schreibmodus. Die Funktion *dump()* serialisiert das Objekt *tickers* und schreibt es in die Datei. Das Schlüsselwort *with* sorgt hierbei einfach dafür, dass unser Filestream ordnungsgemäß und sauber geschlossen wird.

PREISE LADEN

Nun werden wir für unsere ganzen Ticker alle Preise laden und lokal abspeichern. Das wird ein paar Hundert Megabyte benötigen (je nach Zeitraum). Zunächst brauchen wir jedoch wieder drei Imports für diese Funktion, welche wir jedoch schon kennen.

```
import os
import datetime as dt
import pandas_datareader as web
```

Das Modul *os* stellt uns einfach Optionen von unserem Betriebssystem zur Verfügung.

Wir werden hier eine zweite Funktion erstellen, welche für jeden Ticker die Daten herunterlädt. Folglich wird diese Funktion, zunächst die Ticker laden müssen.

```
def lade_preise_von_yahoo(ticker_neuladen=False):

    if ticker_neuladen:
        tickers = lade_sp500_ticker()
    else:
        with open("sp500tickers.pickle", 'rb') as f:
            tickers = pickle.load(f)
```

Unsere Funktion hat einen Parameter *ticker_neuladen*, welcher standardmäßig *False* ist. Sollten wir ihn auf *True* setzen, so wird er unsere bereits vorhandene Funktion aufrufen und damit unser Objekt *tickers* befüllen. Andernfalls benutzen wir die Funktion *load()* von *pickle*, um unsere Datei im Lesemodus zu öffnen und das serialisierte Objekt in unsere Variable zu laden.

```
if not os.path.exists('kursdaten'):
    os.makedirs('kursdaten')
```

Als nächstes prüfen wir, mit *path.exists()*, ob das Verzeichnis *kursdaten* vorhanden ist. Falls nicht, erstellen wir es mit *makedirs()*.

Dann kommen wir zum Hauptteil unserer Funktion, nämlich dem eigentlichen Laden der Kursdaten.

```
start = dt.datetime(2010,1,1)
end = dt.datetime(2019,12,31)

for ticker in tickers:
    if not os.path.exists('kursdaten/{}.csv'.format(ticker)):
        print("{} wird geladen...".format(ticker))
        df = web.DataReader(ticker, 'yahoo', start, end)
        df.to_csv('kursdaten/{}.csv'.format(ticker))
    else:
        print("{} bereits vorhanden!".format(ticker))
```

Wir legen wie immer als erstes unseren gewünschten Zeitraum fest, mit Start-

und Enddatum. Dann lassen wir für jeden Ticker eine CSV-Datei anlegen, falls diese noch nicht existiert. Die Funktionen und Abläufe, sollten schon aus den vorherigen Kapiteln bekannt sein. Das Ausführen der Funktion kann ein paar Minuten in Anspruch nehmen.

Damit haben wir nun unsere zweite fertige Funktion:

```
def lade_preise_von_yahoo(ticker_neuladen=False):

    if ticker_neuladen:
        tickers = lade_sp500_ticker()
    else:
        with open("sp500tickers.pickle", 'rb') as f:
            tickers = pickle.load(f)

    if not os.path.exists('kursdaten'):
        os.makedirs('kursdaten')

    start = dt.datetime(2010,1,1)
    end = dt.datetime(2019,12,31)

    for ticker in tickers:
        if not os.path.exists('kursdaten/{}.csv'.format(ticker)):
            print("{} wird geladen...".format(ticker))
            df = web.DataReader(ticker, 'yahoo', start, end)
            df.to_csv('kursdaten/{}.csv'.format(ticker))
        else:
            print("{} bereits vorhanden!".format(ticker))
```

DATEN KOMPILIEREN

Alle guten Dinge sind drei. Daher werden wir nun eine abschließende, dritte Funktion schreiben, welche unsere gesammelten Daten kompiliert. Das bedeutet, dass wir aus den 500 verschiedenen CSV-Dateien, eine einzige machen, welche die *Adjusted Close* Werte aller 500 Unternehmen, für den angegebenen Zeitraum, enthält.

Wir fangen auch diese Funktion damit an, dass wir zunächst, die Tickers auslesen.

```
def daten_kompilieren():
    with open("sp500tickers.pickle", "rb") as f:
        tickers = pickle.load(f)
```

```
main_df = pd.DataFrame()
```

Wir haben hier auch direkt schon ein leeres DataFrame erstellt, welches am Ende dann, alle Daten kombiniert beinhalten soll.

Um unser DataFrame zu befüllen, lesen wir aus jedem einzelnen Ticker-DataFrame den *Adjusted Close* Wert aus und fügen diesen zu unserem *main_df* hinzu.

```
print("Daten werden kompiliert...")
for ticker in tickers:
    df = pd.read_csv("kursdaten/{}.csv".format(ticker))
    df.set_index("Date", inplace=True)

    df.rename(columns = {"Adj Close": ticker}, inplace=True)
    df.drop(["Open","High","Low","Close", "Volume"], 1, inplace=True)

    if main_df.empty:
        main_df = df
    else:
        main_df = main_df.join(df, how='outer')
```

Wir beginnen mit der Meldung, dass wir kompilieren. Dann lassen wir eine For-Schleife für alle Ticker laufen. Für jeden Ticker erstellen wir ein DataFrame und lesen alle Werte aus der jeweiligen CSV-Datei aus. Wir definieren die Spalte *Date* als Index, mit der Funktion *set_index()*.

Dann benennen wir die Spalte *Adjusted Close* um und zwar in das jeweilige Ticker-Symbol. Das hat den Sinn, dass dieses DataFrame am Ende dann nur eine einzige Spalte in unserem *main_df* sein wird. Mit der Funktion *drop()* löschen wir nun die anderen vier Spalten.

Sollte das nun der erste Eintrag in unserem *main_df* sein, so übernehmen wir gleich den Inhalt, um eine Grundstruktur zu definieren. Ab dann wird jedes weitere DataFrame nur angehängt. Das funktioniert mit der Methode *join()*, welche Sie, falls Sie sich schon mal mit Datenbanken befasst haben, vielleicht aus SQL kennen. Als *how* wählen wir hier *outer*, wir führen also einen Outer-Join durch. Das führt hier dazu, dass wir keine Reihen verlieren, wenn ein Symbol dort keine Werte hat, andere jedoch schon.

Nach der Schleife ist unser Haupt-DataFrame fertig und muss nur noch in eine CSV-Datei geschrieben werden.

```
main_df.to_csv('sp500_daten.csv')
print("Daten kompiliert!")
```

Wie die Meldung zeigt, sind wir dann fertig mit dem Kompilieren unserer Daten. Nun führen wir einmal unsere Funktionen aus. Das kann wieder einige Minuten in Anspruch nehmen, da sehr viele Daten geladen werden müssen.

```
lade_preise_von_yahoo(ticker_neuladen=True)
daten_kompilieren()
```

DATEN VISUALISIEREN

Jetzt haben wir lokal eine CSV-Datei mit allen Tickern des S&P 500 Index. Wir brauchen also, für vergangene Daten, die Yahoo Finance API, eine Weile nicht mehr anzufragen.

```
sp500 = pd.read_csv('sp500_daten.csv')
sp500['MSFT'].plot()
plt.show()
```

Wir laden unsere CSV-Datei in ein DataFrame und können dann, einfach in die eckigen Klammern unser gewünschtes Ticker-Symbol angeben. In diesem Fall zeichnen wir den Kurs von Microsoft. Nicht vergessen: Für das *show()* müssen wir wieder *matplotlib.pyplot* importieren.

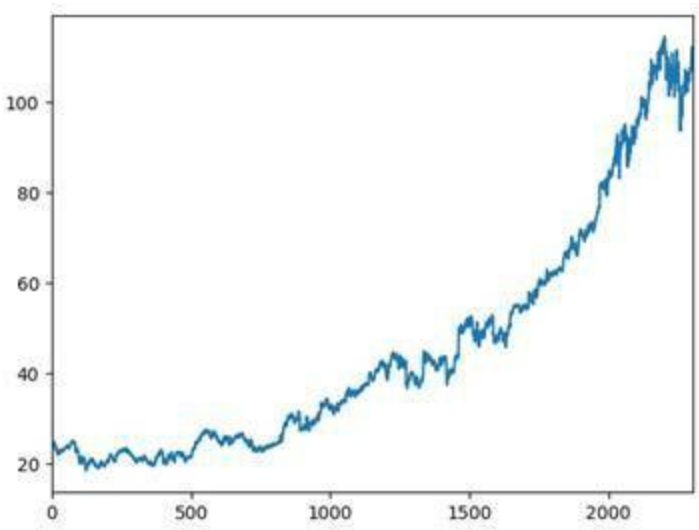


Abb. 7: Kurs von Microsoft (MSFT)

KORRELATIONEN

Abschließend für dieses Kapitel schauen wir uns noch eine sehr interessante Funktion von Pandas DataFrames an. Diese Funktion heißt *corr()* und steht für die Korrelation.

```
korrelation = sp500.corr()
```

Wir erstellen hier nur ein neues DataFrame *korrelation*, welches, wie der Name schon sagt, die Werte der Korrelationen, zwischen den einzelnen Unternehmen, auswertet. Im Klartext heißt das, dass wir uns ansehen inwiefern einzelne Aktien sich gegenseitig beeinflussen.

```
print(korrelation)
```

	MMM	ABT	ABBV
MMM	1.000000	0.919446	0.928664
ABT	0.919446	1.000000	0.907837
ABBV	0.928664	0.907837	1.000000
ABMD	0.812892	0.888733	0.888560
ACN	0.963421	0.952070	0.936506

Abb. 8: Ein Teil der Korrelationstabelle

Die Zahlen, welche wir hier sehen zeigen uns an wie „ähnlich“ die Veränderung der Kurse zwischen den einzelnen Aktien ist. Die Aktie **MMM** und **MMM** haben eine Korrelation von 100% weil es dieselbe Aktie ist. **ABBV** und **MMM** haben wiederum nur noch ungefähr 93% Korrelation, was immer noch sehr viel ist.

Wenn Sie sich die gesamte Tabelle ansehen, werden Sie feststellen, dass es teilweise Korrelationen gibt die nicht einmal 1% sind und sogar welche die negativ sind. Das bedeutet, dass wenn Aktie A fällt, Aktie B steigt und umgekehrt.

Diese Tabelle kann bei der Analyse und Vorhersage von Kursen sehr hilfreich sein. Bei Bedarf können Sie sich auch etwas Kreatives einfallen lassen, um diese Daten graphisch darzustellen.

9 – TRENDLINIEN

Jetzt da wir unsere Daten lokal haben und gut mit ihnen arbeiten können, befassen wir uns mit den sogenannten Trendlinien. Diese zeigen uns visuell, in welche Richtung sich der Kurs entwickelt. Mathematisch gesehen ist es eine simple Lineare Regression.

```
import numpy as np
```

Wir werden für dieses Kapitel das Modul *numpy* verwenden, welches mit *pandas* bereits mitinstalliert worden sein sollte. Andernfalls können Sie es mit *pip* installieren.

Zunächst wählen wir aus unseren Daten den gewünschten Ticker aus.

```
sp500 = pd.read_csv('sp500_daten.csv')
daten = sp500['AAPL']
```

Wir laden hier aus unserer CSV-Datei die Daten für das Unternehmen Apple. Als nächstes müssen wir uns darum kümmern, unser Datum zu quantifizieren, um es als X-Wert benutzen zu können.

```
x = list(range(0, len(daten.index.tolist()), 1))
```

Dafür erstellen wir hier eine Liste, die so lange ist wie die Anzahl an Daten in unserem DataFrame. Hierzu benutzen wir die Funktion *tolist()* beim Attribut *index*, um unsere Indizes als Liste bereitzustellen. Dann benutzen wir die Funktion *len*, um die Länge zu ermitteln und erstellen anschließend mit der Funktion *range()* unsere Zahlenreihe.

Nun werden wir auf unsere Daten ein lineares Regressionsmodell anwenden.

```
fit = np.polyfit(x, daten.values, 1)
fit1d = np.poly1d(fit)
```

Mit der Funktion *polyfit()* bilden wir ein Regressionsmodell, zwischen x und unseren Daten. Die 1 steht hierbei für *ersten Grades*, also linear. Dann definieren wir noch eine 1-Dimensionale Polynomfunktion, um das Format

anzugleichen.

Zu guter Letzt zeichnen wir das Ganze noch.

```
plt.grid(True)
plt.plot(daten.index, daten.values, 'b', daten.index, fit1d(x), 'r')
plt.show()
```

Wir zeichnen einmal in blau unseren Kurs und einmal in rot unsere Regressionswerte. Das Resultat sieht so aus:

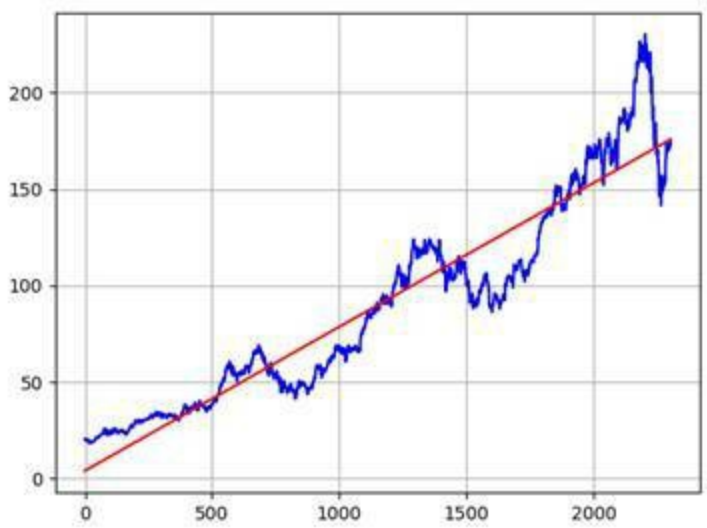


Abb. 8: Kurs mit Regressionslinie

10 – KURSE VORHERSAGEN

Zum Abschluss dieses Buches werden wir uns ein wenig in den Bereich des Machine Learnings hineinwagen. Auch hier werden wir wieder eine simple Lineare Regression benutzen, um unseren Kurs für die Zukunft vorherzusagen.

Hierzu werden wir einige neue Bibliotheken importieren müssen:

```
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

Anfangen werden wir wieder damit, dass wir unsere S&P 500 Daten laden und den gewünschten Kurs auslesen.

```
sp500 = pd.read_csv('sp500_daten.csv')

df = sp500[['Date']]
df['AAPL'] = sp500['AAPL']
df.set_index('Date', inplace=True)
```

Wir gehen diesmal jedoch ein bisschen anders vor. Wir erstellen ein neues DataFrame, in welches wir die Datums-Spalte laden und als Index definieren. Ebenso fügen wir dort die Daten unseres gewünschten Tickers ein. Das hat den Grund, dass wir ein DataFrame benötigen, weil wir noch eine Spalte hinzufügen werden.

```
tage = 50
df['Shifted'] = df['AAPL'].shift(-tage)
df.dropna(inplace=True)
```

Wir definieren hier als erstes wie viele Tage wir vorhersagen möchten. In diesem Fall sind es 50. Dann erstellen wir eine neue Spalte, in welche wir unsere Werte, um 50 Tage verschoben, laden. Das machen wir mit der Funktion *shift()*. Anschließend löschen wir alle Datensätze, welche NaNs enthalten.

Nun müssen wir unsere Daten so aufbereiten, dass unser Modell daraus lernen kann.

```
X = np.array(df.drop(['Shifted'],1))
y = np.array(df['Shifted'])
X = preprocessing.scale(X)
```

Hierzu benutzen wir Numpy, um aus den einzelnen Spalten des DataFrames, Arrays zu machen. Als X-Werte nehmen wir das DataFrame ohne die verschobenen Werte und als Y-Werte nur die verschobenen Werte.

Mit der Funktion *preprocessing.scale()* formatieren wir unsere X-Werte nun so, wie sie benötigt werden für das Lineare Regressionsmodell.

TRAINIEREN UND TESTEN

Wenn wir im Machine Learning ein Modell darauf trainieren, für unbekannte Daten eine Vorhersage zu treffen, benutzen wir einen Teil der uns bekannten Daten dazu, es zu trainieren und einen anderen Teil, um zu überprüfen, wie

genau die Vorhersagen sind. Das Verhältnis ist in der Regel 80/20.

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2)
```

Wir definieren hier vier Variablen (Reihenfolge ist wichtig), welche wir mit der Funktion *train_test_split* befüllen. Diese Funktion nimmt unsere X- und Y-Werte und teilt sie in dem angegebenen Verhältnis in Trainings- und Testdaten auf. In diesem Fall ist unsere *test_size* 0.2, was bedeutet, dass wir 80% als Trainingsdaten und 20% als Testdaten benutzen.

```
modell = LinearRegression()
modell.fit(X_train, y_train)
genauigkeit = modell.score(X_test, y_test)
print(genauigkeit)
```

Mit *LinearRegression()* definieren wir nun unser Modell. Die Funktion *fit()* trainiert es, indem es die Trainings-Daten benutzt.

Wir können nun mit der Funktion *score()* und mit Hilfe unserer Test-Daten überprüfen, wie genau das Modell ist. Sie sollte ungefähr bei 80-95% liegen in der Regel. Je mehr Tage man benutzt, desto geringer wird sie logischerweise.

DATEN VORHERSAGEN

Wir können nun unser trainiertes und getestetes Modell benutzen, um Vorhersagen zu machen. Es sei jedoch gewarnt, dass diese nicht zu ernst zu nehmen sind, da das lineare Regressionsmodell nicht sehr aussagekräftig ist bei Aktienkursen.

```
X = X[:-tage]
X_neu = X[-tage:]
vorhersagen = modell.predict(X_neu)
print(vorhersagen)
```

Wir versuchen nun für die nächsten 50 Tage die Y-Werte, also die Kurse, zu ermitteln. Das machen wir mit der Funktion *predict()*. Dieser übergeben wir die Werte, der nächsten Tage und sie gibt uns ein DataFrame mit Vorhersagen zurück.

Bei diesem Beispiel sah die Ausgabe folgendermaßen aus:

[193.34283124 192.75455257 189.63952856 188.60764114 188.97410055 199.78498044
205.46527982 206.0439297 207.08547567 205.1952571 205.33026846 206.90223108 206.30221036
207.59904567 208.45067983 208.92488639 211.90565804 216.02838588 213.97671028
213.57023262 213.57990605 214.00573057 214.65413335 216.37679318 218.08007638
221.25437621 223.23833779 225.75456141 226.46103455 225.01903821 221.37053181
219.62852512 216.75422077 222.0963667 219.40594687 224.57388171 222.0866635 216.31872281
216.66711522 216.79292938 218.39944849 216.105818 219.13494192 220.48984759 218.77687608
223.16090567 223.92546405 225.39648069 227.3513922 230.05150035]

PYTHON
MACHINE LEARNING

FLORIAN DEDOV

PYTHON

FÜR MACHINE LEARNING

DER SCHNELLE EINSTIEG

Von

FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

Machine Learning ist mittlerweile überall und es wächst stetig weiter. Ob in Robotern, Videospielen, Finanzmärkten, Haushaltsgeräten oder Autos. Die Entwicklung von künstlichen Intelligenzen ist nicht mehr zu stoppen und sie liefert immenses Potenzial. Wer sich jedoch nicht mit diesem Thema auseinandersetzt, wird schneller als er glaubt davon überrollt.

Die Sprache, welche den Markt dominiert ist eindeutig Python. Man findet zwar auch einige KIs, welche in JavaScript, C++, R oder Java geschrieben sind, doch das sind eher Nischen. Wer sich mit Machine Learning befasst muss Python beherrschen!

In diesem Buch lernen Sie, wie Sie mit Python komplexe Machine Learning Modelle trainieren und anwenden können, Sie werden am Ende in der Lage sein, eigene KIs zu schreiben, welche Ihre gewünschten Daten analysieren und Vorhersagen machen können.

Was Sie jedoch beachten müssen ist, dass dieses Buch grundlegende Kenntnisse von Python voraussetzt. Ebenso ist es sinnvoll sich halbwegs gut mit Mathematik auszukennen. Falls Sie noch ein absoluter Anfänger in Python sind, empfehle ich Ihnen zunächst meine Bücher für Anfänger und Fortgeschrittene zu lesen. Diese finden Sie auf meiner Amazon-Seite.

Wenn Sie dieses Buch lesen, ist es sehr wichtig, dass Sie sich nicht einfach von der Theorie und den Beispielen berieseln lassen, sondern aktiv mitprogrammieren. Nur können Sie das, was Sie lernen auch festigen und anwenden. Sie sollten bereits eine funktionierende Entwicklungsumgebung und Python 3 installiert haben. Wenn Sie das noch nicht haben, verweise ich Sie wieder auf mein Buch für Anfänger.

Bevor wir nun in das Buch starten, noch eine Sache:

Falls Sie nach dem Lesen dieses Buches, der Meinung sind, dass es positiv zu Ihrer Programmierkarriere beigetragen hat, würde es mich sehr freuen,

wenn Sie eine Rezension auf Amazon hinterlassen. Danke!

1 – WAS IST MACHINE LEARNING

Künstliche Intelligenz, Machine Learning und Neuronale Netze sind Begriffe, welche man heutzutage überall hört. Doch was steckt eigentlich hinter diesen? Was ist Machine Learning?

Im Grunde genommen ist es ein Teilbereich der Künstlichen Intelligenz. John McCarthy definierte diese wie folgt: *„Die Wissenschaft und die Technik, welche sich damit befassen, intelligente Maschinen zu erschaffen, speziell intelligente Computerprogramme.“*

Wir versuchen in diesem Gebiet also, einen Computer, oder eine computergesteuerte Einheit, dazu zu bringen, intelligent zu denken. Und zwar in einer ähnlichen Weise, wie wir Menschen es tun.

Dabei gehen wir so vor, dass wir zunächst beobachten, wie das menschliche Hirn denkt, lernt und Entscheidungen trifft, während es versucht ein Problem zu lösen. Was wir daraus lernen benutzen wir dann, als Basis dafür, intelligente Software und Systeme zu entwickeln.

Machine Learning ist nur einer von mehreren Wegen, dies zu bewerkstelligen. Hierbei geben wir unserem System, keine expliziten Anweisungen. Wir konstruieren ein Machine Learning Modell und dieses lernt und entscheidet dann selbstständig. Doch auch hier gibt es wieder Unterbereiche.

SUPERVISED LEARNING

Der erste Bereich des Machine Learnings, mit welchem wir uns in diesem Buch auseinandersetzen werden, ist das *Supervised Learning*, also das geführte Lernen.

Hierbei geben wir unserem Modell eine Menge an Inputs bzw. Eingaben und dazugehörige Outputs bzw. Ergebnisse. Unser Modell soll so lernen, wie es Entscheidungen zu treffen hat und dann das Gelernte nutzen, um

Vorhersagen für unbekannte Inputs zu machen.

Die klassischen Verfahren für Supervised Learning sind Lineare Regression, K-Neighbors Klassifikation und Support Vector Machines.

UNSUPERVISED LEARNING

Der Gegensatz zum Supervised Learning ist das *Unsupervised Learning*, also das nicht geführte Lernen. Hierbei geben wir unserem Modell ausschließlich eine Menge an Inputs und dieses muss selber Muster darin erkennen und Entscheidungen treffen.

Hier hat unser Modell keine Information über Richtig und Falsch. Es muss daher Gemeinsamkeiten und Muster in den Daten erkennen und selbstständig einordnen.

Das klassische Verfahren ist hierbei K-Means Clustering, aber auch Neuronale Netze verwenden oftmals Unsupervised Learning Algorithmen.

REINFORCEMENT LEARNING

Die letzte Art von Machine Learning ist *Reinforcement Learning*, also das bestärkende Lernen. Hierbei erzeugen wir anfangs zufällige Modelle und bestärken diese in Verhaltensweisen, welche wir als positiv erachten. Je mehr ein Modell unseren Anforderungen und Kriterien entspricht, desto mehr „Belohnungen“ bekommt diese, in Form von einer Art Note.

Man kann sich das Ganze wie natürliche Selektion und Evolution vorstellen. Wir erzeugen 100 Modelle und nur die 50 besten dürfen bleiben und leicht abgeänderte Versionen ihrer selbst erzeugen.

Die Verfahren, welche hier wichtig sind, sind Genetische Algorithmen. In diesem Buch werden wir jedoch nicht auf diese eingehen

DEEP LEARNING

Oftmals hört man auch den Begriff *Deep Learning* und verwechselt diesen oft mit Machine Learning. Deep Learning ist nur ein Teilgebiet des Machine Learnings, innerhalb welches man mit Neuronalen Netzen arbeitet. Neuronale

Netze sind jedoch ein sehr umfangreiches und komplexes Thema, weswegen ich vor habe in naher Zukunft ein eigenes Buch nur über dieses Thema zu verfassen. In diesem Buch werden wir nur sehr oberflächlich auf Deep Learning eingehen.

ANWENDUNGSBEREICHE

Es wäre einfacher aufzuzählen, wo Machine Learning heutzutage keine Anwendung findet, als die Bereiche, in denen es benutzt wird. Dennoch schauen wir uns im Folgenden kurz an, wo Machine Learning heutzutage in erster Linie angewandt wird.

- Forschung
- Selbstfahrende Autos
- Raumfahrt
- Wirtschaft und Finanzwesen
- Medizin und Gesundheitswesen
- Physik, Biologie, Chemie
- Maschinenbau
- Mathematik
- Robotik
- Bildung
- Forensik
- Polizei und Militär
- Marketing
- Suchmaschinen
- GPS und Routenfindung
- …

Wir könnten ewig Beispiele aufzählen. Ich glaube das verdeutlicht, warum Sie sich in diesem Bereich definitiv weiterbilden sollten. Mit diesem Buch gehen Sie in die richtige Richtung.

2 – WARUM PYTHON

Jetzt da wir nun wissen, was Machine Learning ist und wieso es wichtig ist, etwas darüber zu lernen, stellt sich die Frage, welche Programmiersprache wir lernen sollten. Immerhin müssen wir unsere künstlichen Intelligenzen auch in irgendeiner Form implementieren.

Meine klare Antwort ist hierbei **Python**! Und dafür gibt es zahlreiche gute Gründe. Python hat sich über die letzten Jahre zu DER Sprache für Data Science, Machine Learning und Scientific Computing entwickelt. Bei fast allem, das mit KIs zu tun hat ist Python in irgendeiner Form involviert. Gleichzeitig ist die Sprache aber eine *General Purpose Language*, also eine Sprache, die zahlreiche Anwendungsbereiche hat.

Wenn Sie einen Blick auf den TIOBE-Index werfen, welcher die Popularität von Programmiersprachen auflistet, sehen Sie, dass Python den dritten Platz belegt, mit steigender Tendenz (Stand: März 2019). Das gleiche gilt auch für das Github Ranking und auch im Stackoverflow Ranking sehen wir die Sprache vorne mit dabei.

TIOBE-Index: https://www.tiobe.com/tiobe-index/

GitHub Octoverse: https://bit.ly/2OcY45U

StackOverflow: https://insights.stackoverflow.com/survey/2018/

Durch die massive und immer weiter steigende Popularität von Python, hat sich eine riesige Community gebildet, gerade im Machine Learning Bereich. Für jedes Problem finden Sie daher online bereits eine Lösung.

Hinzu kommt das Python ein riesiges Arsenal an mächtigen Bibliotheken für Machine Learning und Data Science hat. Mit Modulen wie *Numpy*, *Pandas*, *Tensorflow* und *SkLearn* haben wir die Möglichkeit, sehr schnell effiziente

Programme zu schreiben.

Python ist im Allgemeinen eine sehr simple und einfach zu lernende Sprache. Es ist also auf jeden Fall eine gute Wahl. Alternativen sind R, MATLAB, Java oder C++, welche jedoch bei weitem kein so gutes Gesamtpaket liefern.

3 – BIBLIOTHEKEN INSTALLIEREN

Bevor es nun in das erste praktische Kapitel geht, müssen wir einige essentielle Bibliotheken installieren. Hierbei ist es wichtig, dass Sie schon einen Paketmanager, wie *pip*, installiert haben. Installieren Sie folgende Pakete:

```
pip install numpy
pip install pandas
pip install matplotlib
pip install sklearn
pip install tensorflow
```

NUMPY

Das Modul *numpy* werden wir benutzen, um mit Arrays zu arbeiten und um effizient mit Matrizen zu arbeiten. Diese Bibliothek bietet uns die Möglichkeit, sehr simpel und effektiv mit linearer Algebra zu arbeiten.

PANDAS

Mit *pandas* haben wir die Möglichkeit, mit sogenannten DataFrames zu arbeiten, welche uns eine Excel-ähnliche Verarbeitungsmethode von Datensätzen zur Verfügung stellt. Die meisten unserer Daten werden wir in DataFrames laden.

MATPLOTLIB

Die Bibliothek *matplotlib* stellt uns ein sehr umfangreiches Arsenal an Visualisierungsmöglichkeiten zur Verfügung. Wir werden diese Bibliothek benutzen, um Datensätze und Modelle graphisch darzustellen.

SKLEARN

Unser Hauptmodul für Machine Learning wird *sklearn* sein. Dieses beinhaltet

zahlreiche Modelle, Werkzeuge, Algorithmen und Datensätze, mit welchen wir arbeiten können.

TENSORFLOW

Tensorflow benutzen wir, um Neuronale Netze aufzubauen, ohne diese bis ins letzte Detail verstehen zu müssen.

4 – LINEARE REGRESSION

Jetzt, da das Grundwissen da ist, befassen wir uns mit dem ersten und simpelsten Verfahren, des Machine Learnings – Lineare Regression. Es gehört zum Supervised Learning, was bedeutet, dass wir zum Trainieren Inputs und Outputs benötigen.

MATHEMATISCHE ERKLÄRUNG

Bevor wir unser Modell in Python trainieren und testen, werden wir uns zunächst ansehen, wie Lineare Regression mathematisch funktioniert. Vielleicht haben Sie schon einmal etwas davon in Statistik gehört.

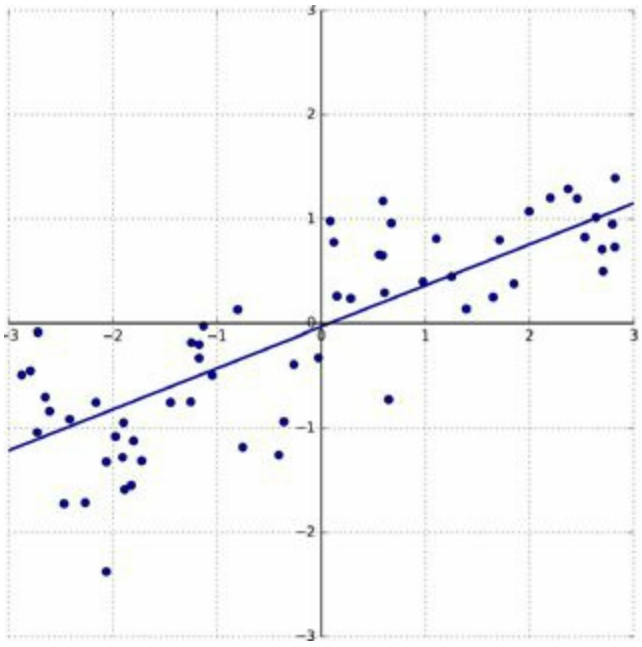


Abb. 1: Lineare Regressionsgerade

In diesem Bild sehen sie eine Menge von Punkten, welche einen X- und einen Y-Wert haben. Den X-Wert nennen wir hierbei *Feature* und den Y-Wert *Label*. Unser lineares Regressionsmodell ist hierbei die Linie. Diese ist so platziert, dass sie möglichst nahe zu allen Punkten gleichzeitig liegt. Wir haben also die Linie auf die vorhandenen Punkte „trainiert“.

Die Idee ist nun, dass wir einen neuen X-Wert hernehmen, ohne den dazugehörigen Y-Wert zu kennen. Wir orientieren uns dann an der Linie und finden dort den Y-Wert, welchen uns das Modell vorhersagt. Da diese Linie jedoch sehr generalisiert ist, werden wir ein relativ ungenaues Ergebnis erhalten.

Man muss jedoch auch erwähnen, dass lineare Modelle ihre Effektivität erst dann wirklich entfalten, wenn wir es mit zahlreichen Features (also Dimensionen) zu tun haben. Wenn wir versuchen einen Zusammenhang zwischen Fehlstunden, Lernzeit und Endnote zu ermitteln, werden wir ein ungenaueres Ergebnis erhalten, als wenn wir 30 Parameter mit einbeziehen. Logischerweise haben wir dann jedoch keine Gerade, oder ebene Fläche mehr, sondern eine *Hyperebene*. Also mehr oder weniger das Äquivalent zu einer Geraden, in höheren Dimensionen.

DATEN LADEN

Um nun mit unserem Code anzufangen, benötigen wir zunächst Daten, mit welchen wir arbeiten möchten. Hierbei bedienen wir uns einem Dataset von UCI.

Link: https://archive.ics.uci.edu/ml/datasets/student+performance

Wir laden, von der *Data Folder* die ZIP-Datei herunter und entpacken von dort die Datei *student-mat.csv* in den Ordner, in welchem wir programmieren.

Nun fangen wir mit dem Code an. Wir importieren zunächst die wichtigen Bibliotheken.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

Von *sklearn* importieren wir hier zum Einen, die Lineare Regression und zum Anderen, eine Methode, welche uns ermöglicht, unsere Daten in Trainings- und Testdaten aufzuteilen.

Unsere erste Aktion ist es, die Daten aus der CSV-Datei in ein Pandas-

DataFrame zu laden. Das machen wir mit der Funktion *read_csv()*.

```
daten = pd.read_csv('student-mat.csv', sep=';')
```

Wichtig ist hierbei, dass wir unseren *Separator (sep)* auf Semikolon umstellen, da standardmäßig von einem Komma ausgegangen wird, bei CSV-Dateien.

Im nächsten Schritt überlegen wir uns, welche Features (also Spalten) für uns relevant sind, und was genau wir vorhersagen wollen. Eine Beschreibung aller Features finden Sie auf der zuvor angeführten Webseite. Wir werden uns in diesem Beispiel auf die folgenden Spalten beschränken.

```
'''
Age - Alter
Sex - Geschlecht
Studytime - Lernzeit
Absences - Abwesenheiten
G1 - Erste Note
G2 - Zweite Note
G3 - Dritte Note (Label)
'''
daten = daten[['age', 'sex', 'studytime', 'absences', 'G1', 'G2', 'G3']]
```

Im Klartext bedeutet das, dass wir von unserem DataFrame nur diese Spalten auswählen, von den 33 möglichen. G3 ist unser Label und der Rest sind unsere Features. Jedes Feature ist eine Achse im Koordinatensystem und jeder Punkt ist ein Datensatz, also eine Zeile in der Tabelle.

Wir haben hier jedoch ein kleines Problem. Das Feature *sex*, also das Geschlecht, ist nicht numerisch, sondern als *F* oder *M* abgespeichert. Damit wir jedoch damit arbeiten, und es im Koordinatensystem eintragen können, müssen wir es in Zahlen umwandeln.

```
daten['sex'] = daten['sex'].map({'F': 0, 'M': 1})
```

Dies tun wir mit der *map()* Funktion. Wir mappen hier ein Dictionary auf unser Feature. Jedes *F* wir dadurch zu 0 und jedes *M* zu 1. Nun können wir damit arbeiten.

Zu guter Letzt definieren wir noch die Spalte des gewünschten Labels als Variable, um leichter damit arbeiten zu können.

```
vorhersage = 'G3'
```

DATEN VORBEREITEN

Unsere Daten sind jetzt vollständig geladen und ausgewählt. Damit wir sie jedoch unserem Modell zum Trainieren geben können, müssen wir sie noch umformatieren. Die Modelle von *sklearn* nehmen nämlich keine Pandas-DataFrames an, sondern Numpy-Arrays. Deshalb wandeln wir unsere Features in ein X-Array und unser Label in ein Y-Array um.

```
X = np.array(daten.drop([vorhersage], 1))
y = np.array(daten[vorhersage])
```

Mit *np.array()* konvertieren wir die ausgewählten Spalten in ein Array. Die Funktion *drop* liefert und das DataFrame, ohne die angegebene Spalte zurück. Unsere X-Werte sind jetzt alle unsere Spalten, bis auf die Endnote. Diese ist alleine im Y-Array.

Um unser Modell nun trainieren und testen zu können, müssen wir unsere verfügbaren Daten aufteilen. Der erste Teil wird benutzt, um die Hyperebene so hinzukriegen, dass sie möglichst gut zu unseren Daten passt. Der zweite Teil überprüft dann die Genauigkeit der Vorhersage, mit bisher unbekannten Daten.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
```

Mit der Funktion *train_test_split()* teilen wir unser X- und unser Y-Array in vier Arrays. Die Reihenfolge muss genau so sein, wie hier gezeigt. Der Parameter *test_size* gibt an, wie viel Prozent der Datensätze, für das Testen benutzt werden sollen. In diesem Fall sind es 10%. Das ist auch ein guter und zu empfehlender Wert. Wir machen dies, um mit Daten, welche unser Modell noch nie zuvor gesehen hat, zu testen, wie akkurat es ist.

TRAINIEREN UND TESTEN

Nun können wir damit beginnen, unser Modell zu trainieren und zu testen. Dafür definieren wir zunächst unser Modell.

```
linear = LinearRegression()
linear.fit(X_train, y_train)
```

Mit dem Konstruktor *LinearRegression()* erstellen wir unser Modell. Die Funktion *fit()* ist unsere Trainingsfunktion. Ihr übergeben wir die Testdaten für X und Y. So schnell ist unser Modell schon trainiert.

Testen können wir das Ganze jetzt mit der Methode *score()*.

```
genauigkeit = linear.score(X_test, y_test)
print(genauigkeit)
```

Dieser übergeben wir die Testdaten und speichern das Ergebnis in eine Variable ab, welche wir dann ausgeben. Da das Aufteilen von Trainings- und Testdaten immer zufällig erfolgt, werden wir bei jedem Durchlauf leicht abgeänderte Ergebnisse haben. Ein durchschnittliches Ergebnis könnte so aussehen:

```
0.9130676521162756
```

Wir können nun auch ganz einfach Vorhersagen machen.

```
X_neu = np.array([[18, 1, 3, 40, 15, 16]])
y_neu = linear.predict(X_neu)
print(y_neu)
```

Wir definieren ein Numpy-Array mit den Features in der richtigen Reihenfolge und benutzen dann die Funktion *predict()*, um unser dazugehöriges Y-Array zu bekommen. Der Output sieht zum Beispiel so aus:

```
[17.12142363]
```

Hier hätten wir also wahrscheinlich die Note 17.

VISUALISIEREN VON ZUSAMMENHÄNGEN

Da wir es hier mit hohen Dimensionen zu tun haben, können wir keinen Graphen von unserem Modell zeichnen. Das geht nur in zwei oder drei Dimensionen. Was wir jedoch visualisieren können, sind Zusammenhänge zwischen einzelnen Features.

```
plt.scatter(daten['studytime'], daten['G3'])
plt.show()
```

Hier zeichnen wir mit der Funktion *scatter()* ein Punktdiagramm, welches

den Zusammenhang, zwischen der Lernzeit und der Endnote darstellt.

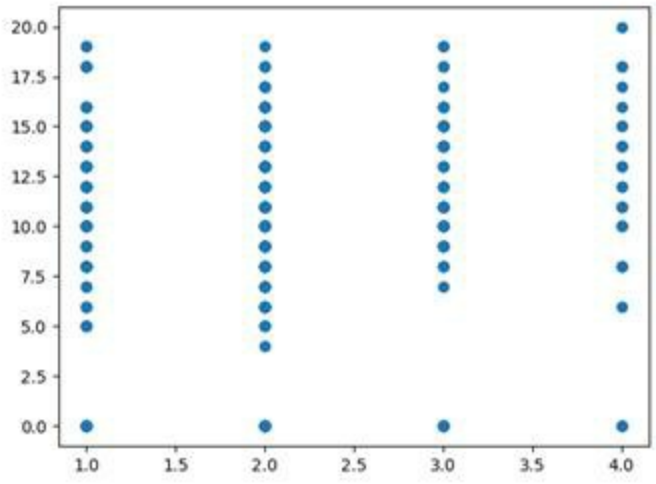


Abb. 2: Scatterplot von Zusammenhang

Da wir hier nur ein einziges Feature benutzen, um unser Label vorherzusagen, ist das Diagramm logischerweise nicht sonderlich aussagekräftig. Sie können jedoch mit beliebig vielen Features experimentieren.

5 – KLASSIFIKATION (KNN)

Mit Linearer Regression haben wir nun für Input-Werte, Output-Werte als Vorhersagen bekommen. Manchmal wollen wir jedoch keine genauen Werte wissen, sondern wir wollen Datensätze bzw. Objekte einfach gruppieren bzw. klassifizieren. Die Methode welche wir uns hierfür ansehen werden, nennt sich *K-Nearest-Neighbors.*

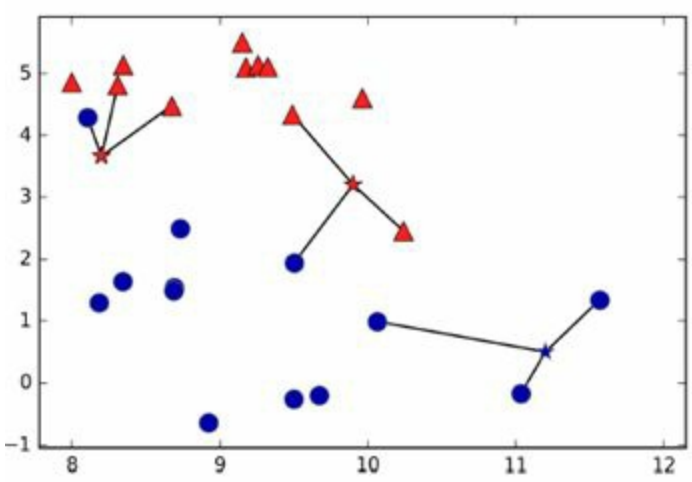


Abb. 3: K-Nearest-Neighbors Klassifikation (Drei Nachbarn)

Dieses Verfahren funktioniert etwas anders, als die Lineare Regression. Wir haben hier einen Punkt, welcher sich aus diversen Features zusammensetzt. Unser Label ist dann die Gruppe bzw. Klasse (in diesem Fall die Farbe und Form des Punktes). Wenn wir nun einen neuen Punkt haben, dessen Klasse wir nicht wissen, ermitteln wir diese, indem wir uns die nächstgelegenen Punkte ansehen. Hierbei ist es wichtig zu definieren, wie viele Nachbarpunkte man überhaupt berücksichtigt.

Eine Möglichkeit ist es, so vorzugehen, dass man die Klasse, des nächstgelegenen Punktes, einfach übernimmt. Bei mehreren Punkten, ist es etwas anders.

Das K in K-Nearest-Neighbors steht für die Anzahl der Nachbarn, welche man berücksichtigt. Wenn wir beispielsweise drei Nachbarpunkte betrachten,

übernehmen wir jene Klasse, von welcher mehr Punkte in der Nähe sind. Bei zwei roten und einem blauen, wird unser neuer Punkt rot. Interessant wird es, wenn wir zum Beispiel zwei oder vier Nachbarn wählen und dann ein Unentschieden haben. Dann müssen wir nämlich auch die Distanz berücksichtigen. Unser Verfahren kalkuliert dann die Klasse, welche gesamt am wenigsten weit entfernt ist.

DATEN LADEN

In diesem Kapitel werden wir unsere Daten direkt aus dem *sklearn* Modul beziehen. Für das Programm benötigen wir folgende Imports:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
```

Beim letzten Import, laden wir ein Dataset, welches Daten über Brustkrebs beinhält.

```
daten = load_breast_cancer()

print(daten.feature_names)
print(daten.target_names)
```

Wir laden diese mit der Funktion *load_breast_cancer()* und lassen uns die Namen der Features und der Targets ausgeben. Unsere Features sind alle Parameter, welche bei der Bestimmung der Klasse helfen sollen. Bei den Targets haben wir in diesem Dataset zwei Optionen: *malignant* und *benign*, beziehungsweise bösartig und gutartig.

DATEN VORBEREITEN

Auch hier konvertieren wir unsere Daten wieder in Numpy-Arrays und teilen sie in Trainings- und Testdaten auf.

```
X = np.array(daten.data)
y = np.array(daten.target)

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.1)
```

Data sind unsere Features und *Target* sind unsere Klassen bzw. unsere Labels.

TRAINIEREN UND TESTEN

Wir definieren zunächst einmal unseren K-Neighbors Klassifizierer und trainieren das Modell.

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

Der Parameter *n_neighbors* gibt an, wie viele Nachbarpunkte wir berücksichtigen wollen. In diesem Fall nehmen wir fünf.

Dann testen wir unser Modell wieder auf seine Genauigkeit.

```
Accuracy = knn.score(X_test, y_test)
print(accuracy)
```

Wie wir sehen ist die Genauigkeit ziemlich hoch:

0.9649122807017544

Auch hier können wir wieder Vorhersagen für neue, unbekannte Daten treffen. Die Erfolgschance bei der Klassifizierung ist sogar sehr hoch. In diesem Fall über 96%.

```
X_neu = np.array([[...]])
ergebnis = knn.predict(X_neu)
```

Die Daten zu visualisieren, ist hier leider nicht möglich, da wir 30 Features haben und kein 30-dimensionales Koordinatensystem zeichnen können.

6 – SUPPORT VECTOR MACHINES

Jetzt wird es mathematisch schon etwas anspruchsvoller. Wir kommen zu den *Support Vector Machines*. Diese sind sehr mächtig, sehr effizient und erzielen in manchen Bereichen sogar weitaus bessere Resultate als Neuronale Netze. Auch hier handelt es sich um eine Art der Klassifikation. Die Methodik ist jedoch eine andere.

Hier achten wir nicht auf Nachbarpunkte, sondern wir suchen eine lineare Funktion, welche unsere Punkte möglichst genau in zwei Gruppen aufteilt. Und zwar so generalisiert wie möglich.

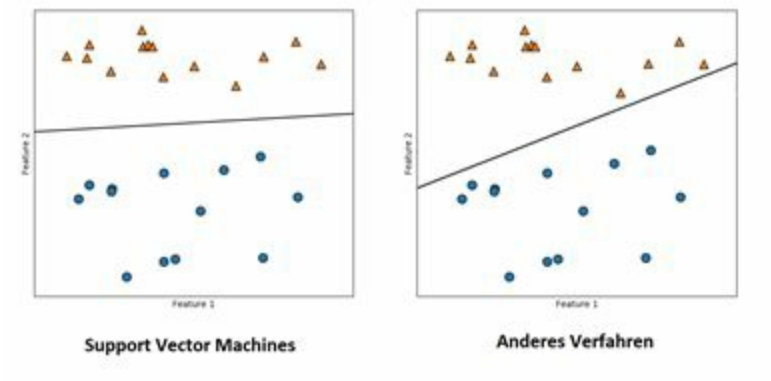


Abb. 4: Support Vector Machine im Vergleich

Generalisiert bedeutet, so allgemein wie es geht. Wie wir sehen, teilen beide Geraden, in dem oberen Bild, die Daten ordnungsgemäß auf. Die linke Funktion (SVM) ist jedoch viel generalisierter und wirkt auch „richtiger“. Die rechte Gerade ist sehr spezialisiert auf die bereits vorhandenen Daten. So etwas nennt man im Machine Learning Bereich *Overfitting*. Das Modell ist sehr stark auf die Trainingsdaten abgestimmt und hat somit schlechtere Chancen, bei neuen Daten, eine korrekte Vorhersage zu treffen.

Mit Support Vector Machines versuchen wir eine Gerade zu finden, welche von den nächsten Punkten, der beiden Klassen, am gleichmäßig und am weitesten entfernt ist. Das geschieht mit sogeannten *Support Vectors*, also

parallelen Linien.

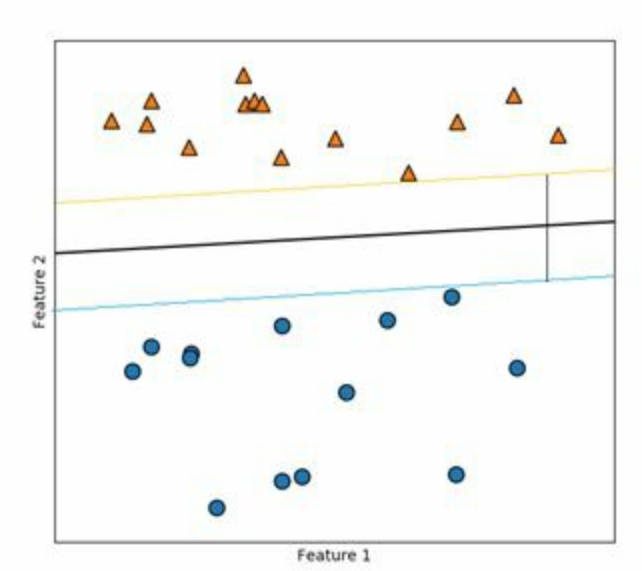


Abb. 5: Support Vectors

KERNELS

Die Daten, welche wir uns bisher angesehen haben, sind relativ leicht zu klassifizieren, da sie klar auseinander liegen. Solche Daten finden wir in der realen Welt so gut wie nie. Ebenso arbeiten wir auch hier in höheren Dimensionen mit vielen Features.

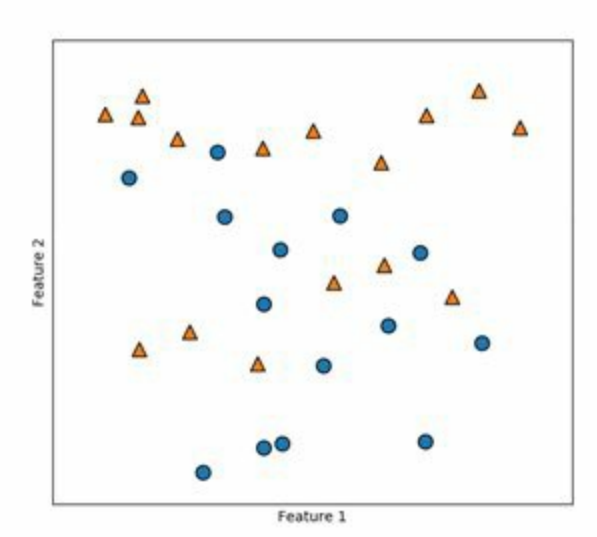


Abb. 6: Beispiel Datensatz

Daten, welche wir der realen Welt entnehmen, sehen oftmals so aus, wie

diese in Abbildung 6. Hier ist es unmöglich eine Gerade zu zeichnen und selbst eine quadratische oder kubische Funktion hilft uns hier nicht. Wenn das der Fall ist, können wir uns mit sogenannten *Kernels* helfen. Diese erweitern unsere Daten um eine neue Dimension. Wir hoffen dadurch die Komplexität der Daten zu erhöhen und eventuell eine Hyperebene als Abgrenzung benutzen zu können.

Wichtig ist hierbei, dass der Kernel, also in diesem Fall die dritte Achse bzw. Dimension, sich aus den bereits vorhandenen Features zusammensetzt. Es ist also kein willkürliches neues Feature sondern eine Kombination, aus den anderen. Wir könnten zum Beispiel sagen, dass unser Kernel *Feature 1 / Feature 2* ist. Es macht jedoch natürlich nur wenig Sinn, so etwas zu tun. Daher gibt es bereits vordefinierte und bewährte Kernels, welche wir nutzen können.

SOFT-MARGIN

Es gibt bei Support Vector Machines ein Konzept, welches sich *Soft-Margin* nennt. Dabei handelt es sich, um bewusst falsch klassifizierte Punkte, um das Modell generalisierter zu halten. Wir wirken damit Ausreißern entgegen.

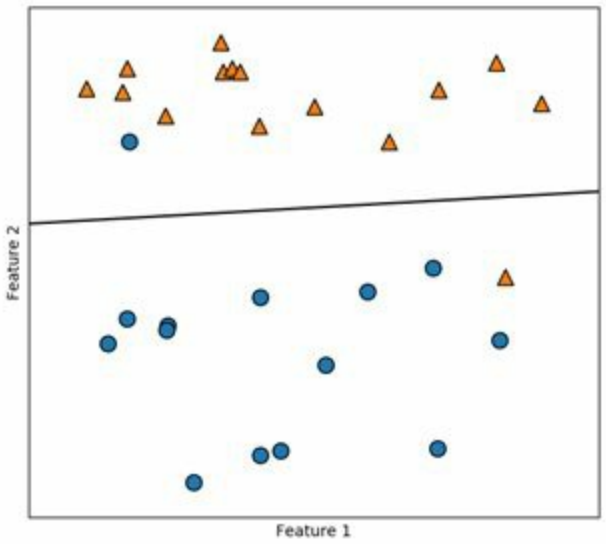


Abb. 7: Soft-Margin

Auch wenn wir hier offensichtlich zwei Punkte falsch klassifiziert haben, wirkt diese Gerade nicht unbedingt falsch. Wir haben hier zwei Fehler

zugelassen, da unsere Funktion sonst sehr ungenau und overfitted wäre.

DATEN LADEN UND VORBEREITEN

Für dieses Verfahren, werden wir wieder das Brustkrebs-Dataset verwenden. Folgende Bibliotheken werden wir hier benötigen:

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
```

Wie Sie sehen, importieren wir hier auch wieder den *KNeighborsClassifier*, da wir am Ende die Genauigkeiten vergleichen möchten.

```
daten = load_breast_cancer()

X = daten.data
y = daten.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=30)
```

Wir haben hier bei unserem *Split* einen neuen Paramter namens *random_state* benutzt. Dieser sorgt nur dafür, dass unser Split immer der exakt selbe ist. Normalerweise werden die Daten nämlich immer zufällig aufgeteilt. Sie können hier jede beliebige Zahl verwenden, die Sie möchten. Bei jeder Zahl haben Sie einen bestimmten Split, welcher bei jedem Durchlauf gleich bleibt.

TRAINIEREN UND TESTEN

Zunächst definieren wir unseren Support Vector Classifier, also unser Modell.

```
modell = SVC(kernel='linear', C=3)
modell.fit(X_train, y_train)
```

Wie Sie sehen, haben wir hier zwei Parameter angegeben. Der erste ist unser Kernel. Hier geben wir an, welche Art von Kernel wir benutzen wollen. Hier haben wir vier verschiedene zur Auswahl: *linear, poly, rbf* und *sigmoid*. Den *rbf* Kernel benutzen wir am besten, wenn unsere Daten ein kreisförmiges Muster haben und wir keine Linie zeichnen können. Der Grund warum wir

hier *linear* und nicht *poly* nehmen ist, die Laufzeit. Poly ist zwar in der Regel genauer, braucht jedoch irrsinnig lange, um das Modell zu trainieren.

Der zweite Parameter *C* ist unsere Soft-Margin, also die Anzahl, der tolerierten Fehler. In diesem Fall drei.

```
genauigkeit = modell.score(X_test, y_test)
print(genauigkeit)
```

Wenn wir unser Modell nun testen, bekommen wir eine sehr gute Genauigkeit.

```
0.9649122807017544
```

Im Vergleich dazu können wir, mit demselben *random_state*, den KNeighborsClassifier beobachten.

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

knn_genauigkeit = knn.score(X_test, y_test)
print(knn_genauigkeit)
```

Das Ergebnis ist nur minimal schlechter. Bei komplexeren Daten kann es jedoch durchaus einen größeren Unterschied geben.

```
0.9473684210526315
```

7 – CLUSTERING (K-MEANS)

Bisher waren alle Verfahren, welche wir uns angesehen haben, aus dem Bereich des Supervised Learnings. Mit Clustering begeben wir uns erstmals in das Unsupervised Learning.

Das bedeutet, dass wir in unseren Daten keine klaren Antworten bzw. Lösungen finden. Wir haben keine vorgegebenen Klassen oder Endwerte, auf welche wir hintrainieren oder mit welchen wir testen können. Wir haben einfach rohe Daten und unser Modell muss diese in sogenannte Cluster einteilen. Die Anzahl der Cluster wird hier (genau wie bei den K-Nearest-Neighbors) durch das *K* definiert.

Das Clustern an sich funktioniert mit sogenannten *Zentroiden* bzw. *Centroids*. So nennt man die Punkte, welche in dem Zentrum des jeweiligen Clusters stehen.

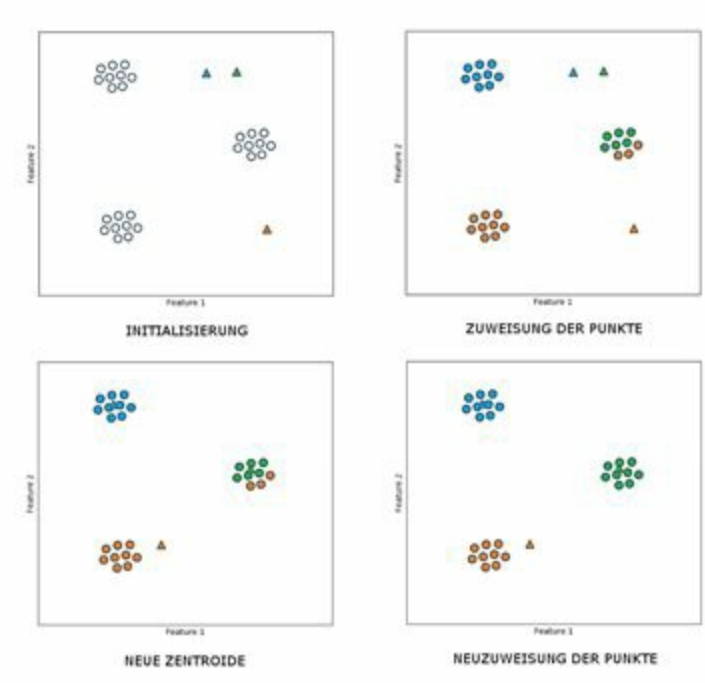


Abb. 8: K-Means-Clustering

Die Abbildung 8 beschreibt den Ablauf des Clusterings ziemlich gut. Zunächst setzen wir in unseren Daten unsere Zentroide absolut willkürlich und zufällig. In diesem Fall haben wir drei Cluster, also auch drei Zentren. Den ersten Schritt nennen wir Initialisierung.

Dann betrachten wir jeden Punkt einzeln und weisen ihm den Cluster, des nächsten Zentroiden zu. Im dritten Schritt richten wir die Zentroide neu aus, sodass diese in der Mitte zwischen allen Ihren zugehörigen Punkten liegen. Dann weisen wir die Punkte wieder neu zu. Das Ganze wiederholen wir nun so lange, bis sich nahezu nichts mehr tut. Dann haben wir ein optimales Modell für das Clustering gefunden. Dieses sollte dann in etwa so aussehen:

Abb. 9: Optimales KMC-Modell

Natürlich sind solche schönen Daten in der Realität so gut wie nie anzutreffen. Ebenso arbeiten wir auch hier nicht nur mit zwei Features und haben daher viel mehr Dimensionen.

DATEN LADEN

Für das K-Means Clustering werden wir ein Dataset laden, welches handgeschriebene Ziffern von 0 bis 9 enthält. Folgende Bibliotheken werden wir benötigen:

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import scale
```

```
from sklearn.datasets import load_digits
```

Nun laden wir die Daten und bereiten sie auch vor.

```
ziffern = load_digits()
daten = scale(ziffern.data)^
```

TRAINIEREN

Um nun zu trainieren, definieren wir, wie gewohnt, unser Modell und fitten es auf die Daten.

```
kmc = KMeans(n_clusters=10, init="random", n_init=10)
kmc.fit(daten)
```

Was hier wichtig ist, sind die Parameter. Der Parameter *n_clusters* gibt an wie viele Cluster wir haben möchten, also wie viele Gruppen. Da wir die Ziffern von Null bis Neun haben, ist 10 der naheliegende Wert.

Mit *init* definieren wir die Art der Initialisierung, also wo unsere Zentroide anfangs platziert werden. Wir haben *random* gewählt. Dadurch werden unsere initialen Centroids zufällig gesetzt.

Zu guter Letzt haben wir noch *n_init*. Dieser Wert gibt an, wie viele Durchläufe wir machen, um den effizientesten Weg zu finden.

Jetzt können wir natürlich, falls vorhanden, neue Daten vorhersagen, wie auch bei anderen Modellen.

```
kmc.predict([...])
```

Wir bekommen hier logischerweise nicht die Antwort, ob unsere Zahl eine 4, 5 oder 9 ist. Aber wir bekommen die Information, zu welchem Cluster, also zu welcher Gruppe sie gehört.

8 – NEURONALE NETZE

Wie bereits anfangs erwähnt, sind *Neuronale Netze* ein wirklich außerordentlich komplexes Thema und viel zu umfangreich, um diese in einem einzelnen Kapitel abzudecken. Aus diesem Grund habe ich vor, in naher Zukunft ein Buch ausschließlich über Neuronale Netze zu schreiben. In diesem Kapitel werden wir nur sehr oberflächlich, ein Modell trainieren, welches handgeschriebene Zahlen erkennt.

AUFBAU EINES NEURONALEN NETZES

Mit Neuronalen Netzen versuchen wir, unsere Programme, nach dem Vorbild des menschlichen Gehirns aufzubauen, nämlich mit Neuronen.

Das menschliche Gehirn ist aus mehreren Milliarden Neuronen zusammengesetzt, welche alle miteinander verbunden sind.

Neuronale Netze sind Gebilde, welche sich ein ähnliches Prinzip zunutze machen.

Der Aufbau eines Neuronalen Netzes sieht zunächst ziemlich simpel aus.

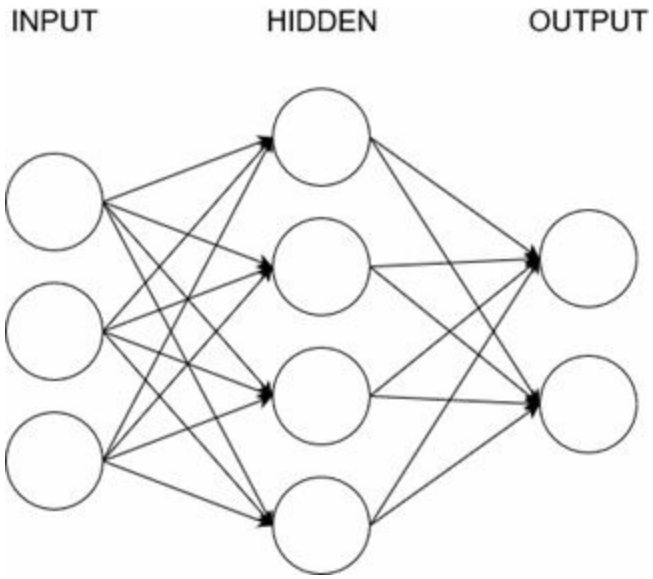


Abb. 10: Aufbau eines Neuronalen Netzes

Wir sehen in dieser Abbildung drei Schichten. Einmal die Eingabeschicht, einmal die Ausgabeschicht und dazwischen, einen sogenannten Hidden Layer.

Die Eingabeschicht ist für unsere Inputs. Dort kommen die Dinge hinein, welche wahrgenommen oder eingegeben werden. Im Grunde genommen also unsere Features.

Wir können nämlich auch neuronale Netze benutzen, um Daten zu klassifizieren, oder Aussagen über diese zu treffen.

Die Ausgabeschicht liefert uns dann die Ergebnisse. Dort sehen wir dann den Output, welcher durch die Inputs generiert wurde.

Alles dazwischen sind Abstraktionsschichten, welche die Komplexität des Systems erhöhen und die interne Logik erweitern. In der Regel kann man sagen, dass das System umfangreicher und komplexer ist, je mehr Hidden Layer und je mehr Neuronen wir haben.

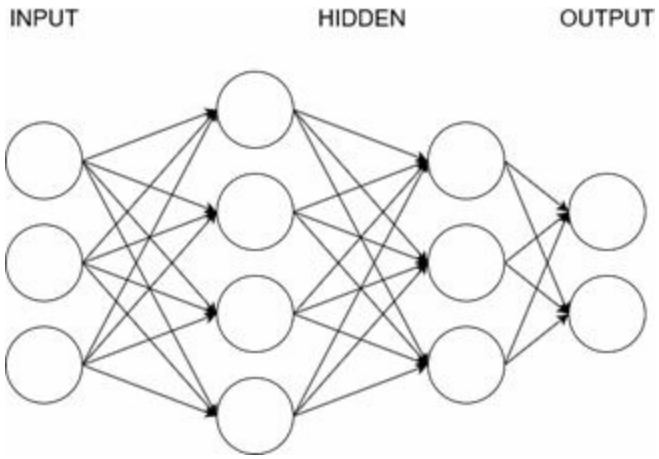


Abb. 11: Neuronales Netz mit zwei Hidden Layers

AUFBAU EINES NEURONS

Um zu verstehen, wie ein Neuronales Netz im Allgemeinen funktioniert, müssen wir uns zunächst, die einzelnen Neuronen ansehen.

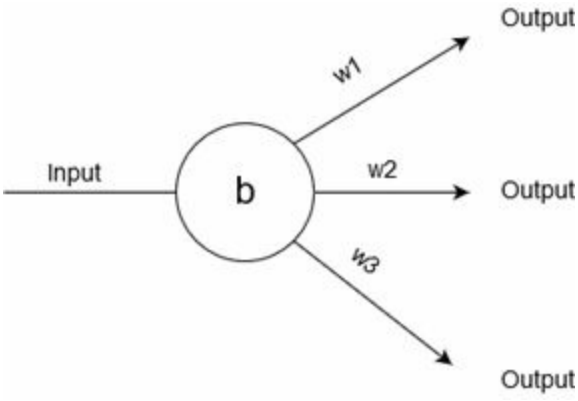


Abb. 12: Aufbau eines Künstlichen Neurons

Wie Sie sehen, hat ein Neuron einen gewissen Input, welcher, entweder der Output eines anderen Neurons ist, oder der Input in der ersten Schicht.

Dieser Input ist eine Zahl. Nun wird diese Zahl mit jedem einzelnen *w* (steht für *weight* – zu Deutsch: Gewicht) multipliziert. Dann wird noch der sogenannte *bias* b abgezogen. Das Ergebnis ist der jeweilige Output.

Was ich gerade erklärt habe und was Sie auf dem Bild sehen, ist jedoch eine veraltete Version eines Neurons und zwar das Perzeptron. Heutzutage benutzen wir weitaus komplexere Sigmoid-Neuronen, welche umfangreichere mathematische Funktionen nutzen, um den Wert der Outputs zu ermitteln.

Dann können Sie sich in etwa vorstellen, wie komplex ein System wird, wenn alle diese Neuronen, sich gegenseitig beeinflussen.

Zwischen unseren Inputs und den Outputs, liegen oftmals zahlreiche Hidden Layer mit Hunderten und Tausenden Neuronen. Diese Abstraktionsebenen führen dann zu dem Ergebnis.

FUNKTIONSWEISE VON KNN

Doch was hat das Ganze nun mit Künstlicher Intelligenz zu tun? Neuronale Netze sind Strukturen, mit vielen Parametern und Schrauben an denen man drehen kann. Wenn man diese richtig konfiguriert, so kommt man von beliebigen Inputs, zu gewünschten Outputs. Da es bei tausenden von Neuronen jedoch ziemlich unübersichtlich für den Menschen wird, benutzen

wir Machine Learning Algorithmen, um diese für uns zu konfigurieren.

Wir tun dies indem wir, zum Beispiel beim Supervised Learning, Trainingsdaten bereitstellen. Das sind Inputs und dazugehörige Outputs, welche wir in das Neuronale Netz einlesen, sodass dieses, seine Gewichte und seine *Biases* dementsprechend konfiguriert.

Mit genug Trainingsdaten, können wir unser Netz also trainieren, sodass es lernt, die gewünschten Ergebnisse zu erzeugen.

Diese Technik ist so unglaublich mächtig, dass Computer damit lernen Autos zu lenken, Schach zu spielen und Weltmeister in Videospielen wie Dota 2 zu schlagen.

Dieses Lernen geschieht mit dem sogenannten *Gradient Descent* Algorithmus, welchen wir uns hier jedoch nicht näher ansehen werden, da dieser hochmathematisch und komplex ist.

ZIFFERN ERKENNEN

Da wir jetzt nicht auf die Funktionsweise aller Algorithmen und Arten von Neuronen eingehen können, werden wir nur grob besprechen, was wir in den einzelnen Codezeilen tun. Wir werden folgende Imports benötigen:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

DATEN LADEN

Die Daten, welche wir hier verwenden werden, laden wir direkt aus den Datasets von Tensorflow.

```
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Das Dataset mit den handgeschriebenen Zahlen nennt sich *MNIST*. Die Zahlen sind in Form von Numpy-Arrays, als 28x28 Pixel große Scans vorhanden.

Hier laden wir unser Dataset und die Funktion *load_data()* liefert,

automatisch, bereits aufgeteilte Trainings- und Testdaten zurück. Nun müssen wir diese noch Normalisieren.

```
X_train = tf.keras.utils.normalize(X_train, axis=1)
X_test = tf.keras.utils.normalize(X_test, axis=1)
```

Wir machen Sie einfach zu multidimensionalen Vektoren mit der Länge Eins. Wenn Sie das mathematisch nicht verstehen, ist es halb so wild.

NEURONALES NETZ AUFBAUEN

Nun beginnen wir unser Neuronales Netz aufzubauen und zu definieren.

```
modell = tf.keras.models.Sequential()
modell.add(tf.keras.layers.Flatten())
modell.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
modell.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
modell.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
```

Anfangs definieren wir das Modell als *Sequential*. Das ist einfach eine Art von Neuronalem Netz. Dann fügen wir einen Eingabelayer und zwei Hidden Layer hinzu. Ersterer hat den Typ Flatten und letztere haben den Typ *Dense*. Mit dem Parameter *units* geben wir an, wie viele Neuronen ein Layer haben soll. *Activation* definiert die Aktivierungsfunktion, welche die Neuronen haben sollen.

Als letztes erstellen wir unseren Ausgabelayer, welcher ebenso den Typen *Dense* hat. Dieser hat zehn Neuronen, um die Zahlen von 0 bis 9 klassifizieren zu können.

Das Modell, welches wir nun definiert haben, müssen wir ebenso kompilieren.

```
modell.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Hier definieren wir den *Optimizer*, die *Loss-Function* und die Metriken. Darauf gehen wir hier auch nicht näher ein.

TRAINIEREN UND TESTEN

Nun können wir unser Modell, wie immer, trainieren und testen.

```
modell.fit(X_train, y_train, epochs=3)
loss, genauigkeit = modell.evaluate(X_test, y_test)
print(loss)
print(genauigkeit)
```

Wir haben hier bei der *fit()* Funktion einen Parameter mit dem Namen *epochs*. Dieser gibt an, wie viele Trainingsdurchläufe wir haben möchten. Mit jedem steigt natürlich die Genauigkeit.

Hier testen wir auch mit einer anderen Methode, nämlich mit *evaluate()*. Dieser Funktion liefert uns zwei Werte zurück, weswegen wir auch zwei Variablen anlegen. Die Genauigkeit ist hierbei der zweite Wert.

Wir können unser Neuronales Netz nun trainieren und testen. Die Genauigkeit ist ziemlich hoch.

0.9684

EIGENE ZIFFERN EINLESEN

Mit etwas Aufwand, können wir nun auch eigene Ziffern, zum Beispiel in Paint, zeichnen und diese von unserem Netzwerk klassifizieren lassen. Hierzu benötigen wir die Bibliothek *cv2* bzw. *opencv-python* (Installations- und Importname sind unterschiedlich).

pip install opencv-python

```
import cv2
```

Sie brauchen hierfür nur ein Bild mit einer Ziffer, welches die Maße 28x28 Pixel hat. Dieses laden wir dann einfach in Python.

```
bild = cv2.imread('ziffer.png')[:,:,0]
bild = np.invert(np.array([bild]))
```

Mit der Funktion *imread()* laden wir das Bild in unser Programm. Hier entfernen wir mit der Notation, die letzte Spalte. Das tun wir, weil sonst das Format nicht passt. Dann müssen wir noch die *invert()* Funktion von Numpy benutzen und unser Bild in ein invertiertes Array konvertieren, damit es unser Neuronales Netz nicht verwirrt.

Nun können wir bereits eine Vorhersage machen, diese aufgeben und, mit Matplotlib, sogar noch unser Bild anzeigen.

```
vorhersage = model.predict(bild)
print("Vorhersage: {}".format(np.argmax(vorhersage)))
plt.imshow(bild[0])
plt.show()
```

Unsere Vorhersage machen wir, wie gewohnt, mit der Funktion *predict()*. Der Grund warum wir beider Ausgabe die Funktion *argmax()* benutzen müssen, ist, dass die Vorehrsage ein Array mit zehn Werten ist, von welchen einer 1 und alle anderen 0 sind. Die Funktion gibt uns die Stelle aus und erleichtert uns das Lesen.

Mit der Funktion *imshow()* von Matplotlib, zeigen wir unser Bild an. Die letztendliche Ausgabe sieht so aus:

Vorhersage: 7

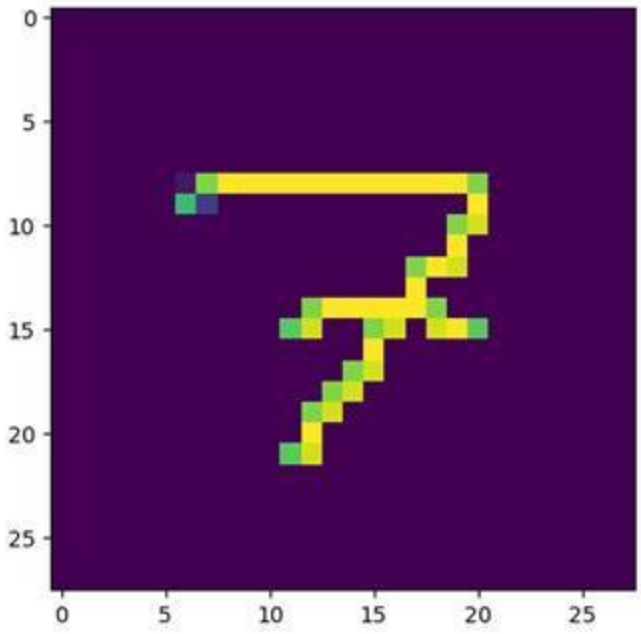


Abb. 13: Bild in Matplotlib

Natürlich wir sie nicht immer zu 100% stimmen. Sie können jedoch lange herumprobieren, die Grenzen testen und viel Spaß haben.

9 – MODELLE SPEICHERN UND OPTIMIEREN

In diesem letzten Kapitel befassen wir uns noch mit dem Speichern und dem Optimieren von Modellen. Bisher haben wir immer unsere Daten geladen, unser Modell trainiert, getestet und dann benutzt. Wollen wir jedoch ein Modell dauerhaft abspeichern, um es später wieder benutzen zu können, ohne es neu zu trainieren, müssen wir anders vorgehen.

SERIALISIERUNG

Hierfür gibt es das Konzept der *Serialisierung*. Bei dieser speichern wir Objekte zur Laufzeit in Dateien ab. Dabei speichern wir nicht nur die Attribute, sondern auch den aktuellen Zustand. Später können wir diese dann wieder einlesen und weiterbenutzen. In Python benutzen wir dafür die Bibliothek *pickle*.

```
import pickle
```

MODELLE SPEICHERN

Als Beispiel für das Speichern von Modellen werden wir hier wieder auf unsere Klassifikation der Brustkrebsdaten zurückgreifen und zwar mittels Support Vector Machine.

```
import pickle
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

daten = load_breast_cancer()

X = daten.data
y = daten.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)

modell = SVC(kernel='linear', C=3)
```

```
modell.fit(X_train, y_train)
```

Dieser Code dürfte für Sie nicht neu sein. Wir trainieren hier einfach einen Support Vector Classifier. Diesen werden wir jedoch nicht direkt benutzen, sondern zunächst abspeichern und ihn dann in einem anderen Skript wieder verwenden.

```
with open('svm_modell.pickle', 'wb') as datei:
    pickle.dump(modell, datei)
```

Wir öffnen hier einen FileStream für die Datei *svm_modell.pickle* und zwar im Modus für das Schreiben von Bytes. Dann benutzen wie die Funktion *dump()* von Pickle, um unser Modell in die geöffnete Datei zu speichern.

Sollten Sie diese Syntax nicht wirklich verstehen, empfehle ich Ihnen meine Bücher für Anfänger und für Fortgeschrittene im Python Programmieren.

MODELLE LADEN

Unser Modell nun zu laden und in einem anderen Skript zu benutzen ist sogar noch simpler.

```
import pickle

with open('svm_modell.pickle', 'rb') as datei:
    modell = pickle.load(datei)

modell.predict([...])
```

Wir öffnen unsere Datei und benutzen die Funktion *load()* von Pickle, um unser Modell zu laden. Dann können wir es genauso benutzen, wie wir es gewohnt sind.

MODELLE OPTIMIEREN

Diese Serialisierung von Modellen können wir benutzen, um ein möglichst genaues und gutes Modell zu trainieren, es also zu optimieren, und dann abzuspeichern.

```
bestes = 0

for x in range(2500):
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)

modell = SVC(kernel='linear', C=3)
modell.fit(X_train, y_train)
genauigkeit = modell.score(X_test, y_test)
if genauigkeit > bestes:
    bestes = genauigkeit
    print("Höchste Genauigkeit: ", genauigkeit)
    with open('svm_modell.pickle', 'wb') as datei:
        pickle.dump(modell, datei)
```

Das Konzept ist ziemlich simpel. Wir haben eine Variable, welche immer die bisher höchste Genauigkeit abspeichert. Anfangs ist diese natürlich Null. Dann lassen wir eine Schleife laufen (in diesem Fall 2500 Mal) und trainieren immer wieder unser Modell, indem wir die Daten verschieden aufteilen.

Beim Testen des Modells, prüfen wir dann, ob die Genauigkeit höher ist, als die bisher höchste. Sollte das der Fall sein, überschreiben wir unser serialisiertes und abgespeichertes Modell. Somit finden wir, jenes mit der höchsten Genauigkeit.

Hierbei muss natürlich erwähnt werden, dass die Gefahr der Overfittings, vor allem bei simplen Datasets, sehr hoch ist. Es ist nicht unmöglich eine Genauigkeit von 100% zu erreichen. Ob diese dann aber auch für neue, unbekannte Daten so akkurat ist, wissen wir nicht.

PYTHON

NEURONALE NETZE

FLORIAN DEDOV

PYTHON FÜR NEURONALE NETZE

DER SCHNELLE EINSTIEG

VON

FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

Kaum ein Thema ist in der Informatik derzeit populärer als Machine Learning. Es ist faszinierend, wie Computer nur anhand von Daten und Algorithmen lernen, ohne direkte Instruktionen, komplexe Probleme zu lösen. Mithilfe von Machine Learning haben wir es geschafft einen Computer Stimmen erkennen und reproduzieren, Objekte erkennen und Autos selbstständig fahren zu lassen. Interessant ist hierbei, dass man bei großen Errungenschaften immer wieder von *Deep Learning* hört. Selten hört man, dass mit einer Linearen Regression oder mit einem klassischen Klassifikationsalgorithmus ein Durchbruch erzielt wurde. Fast jedes Mal handelt es sich um Neuronale Netze.

Deep Learning ist das Teilgebiet des Machine Learnings, welches sich mit Neuronalen Netzen befasst. Diese Neuronalen Netze sind dem menschlichen Gehirn nachempfunden und erzielen eindrucksvolle Resultate.

Sie besiegen Profi-Schachspieler, fahren selbstständig Autos und sogar in komplexen Computerspielen, wie Dota 2, sind diese künstlichen Intelligenzen den Menschen schon voraus.

Hinzu kommt, dass wir in diesen Gebieten unglaublich schnell Fortschritte machen und es nahezu unmöglich ist, sich auszumalen, in welche Richtung das Ganze über die nächsten 10-20 Jahre geht.

Eines ist jedoch sicher: Wer künstliche Intelligenzen, Machine Learning und speziell Neuronale Netze versteht, hat einen immensen Vorteil jenen Menschen gegenüber, die von der Entwicklung überrollt werden. Es ist also sehr wichtig, dass Sie sich mit diesem Gebiet auseinandersetzen.

DIESES BUCH

Dieses Buch wird Ihnen von Grund auf erklären, was Neuronale Netze sind,

wie sie funktionieren und wie Sie diese in der Programmiersprache Python erstellen und anwenden können. Wir werden mit eindrucksvollen Beispielen arbeiten und Sie werden von manchen Ergebnissen erstaunt sein.

Was in diesem Buch jedoch vorausgesetzt wird, sind fortgeschrittene Python-Kenntnisse und zumindest ein grundlegendes Verständnis von Machine Learning. Auch grundlegende Mathematik-Kenntnisse sind von Vorteil. Falls Ihnen diese Skills fehlen, können Sie sich gerne auf meiner Amazon Autorenseite, meine einleitenden Werke ansehen. Diese beginnen bei Büchern für Anfänger und Fortgeschrittene und ziehen sich dann weiter durch die Gebiete der Data Science, des Machine Learnings und der Finanzanalyse mit Python. Natürlich können Sie sich das notwendige Wissen auch anders aneignen. In diesem Buch wird es jedoch keine Erklärungen zur grundlegenden Python-Syntax geben.

Autorenseite: https://amzn.to/395BNB5

LESETIPPS FÜR DIESES BUCH

Im Grunde genommen steht es Ihnen frei, sich selbst auszusuchen, wie Sie dieses Buch lesen möchten. Wenn Sie der Meinung sind, dass die einleitenden Kapitel für Sie uninteressant sind oder, dass Sie bereits alles wissen, können Sie diese auch überspringen. Ebenfalls können Sie das Buch von vorne bis hinten durchlesen, ohne jemals eine Zeile Code zu schreiben. Doch das ist alles nicht empfehlenswert.

Ich persönlich empfehle Ihnen jedes Kapitel in der richtigen Reihenfolge zu lesen, da diese aufeinander aufbauen. Der Code funktioniert zwar auch ohne die ersten Kapitel, doch dann fehlt es Ihnen an Verständnis und Sie haben keine Ahnung, was Sie implementieren und warum es funktioniert oder nicht.

Außerdem ist es unglaublich wichtig, dass sie nebenher aktiv mitprogrammieren. Nur so verstehen Sie wirklich die Inhalte in diesem Buch. Es wird in den späteren Kapitel viel Code geben. Lesen Sie ihn sich durch, verstehen Sie ihn, aber implementieren Sie ihn auch auf Ihrer eigenen Maschine. Experimentieren Sie herum. Was passiert wenn Sie einzelne Parameter ändern? Was passiert, wenn sie noch etwas dazugeben? Probieren Sie alles aus.

Mehr gibt es glaube ich hier nicht zu sagen. Ich wünsche Ihnen viel Erfolg und Spaß beim Lernen über Neuronale Netze mit Python. Ich hoffe, dass dieses Buch Ihnen bei Ihrer weiteren Laufbahn behilflich sein kann!

Eine kleine Sache noch! Falls Sie nach dem Lesen dieses Buches, der Meinung sind, dass es positiv zu Ihrer Programmierkarriere beigetragen hat, würde es mich sehr freuen, wenn Sie eine Rezension auf Amazon hinterlassen. Danke!

1 - GRUNDLAGEN NEURONALER NETZE

WAS SIND NEURONALE NETZE?

Bevor wir nun lernen, wie Neuronale Netze funktionieren, welche Arten es gibt und wie man in Python mit ihnen arbeitet, sollten wir zunächst einmal klären, was Neuronale Netze überhaupt sind.

Künstliche Neuronale Netze sind mathematische Strukturen, welche dem menschlichen Gehirn nachempfunden sind. Sie setzen sich aus sogenannten *Neuronen* zusammen, welche miteinander verbunden sind.

Das menschliche Gehirn besteht aus mehreren Milliarden solcher Neuronen. Künstliche Neuronale Netze sind Gebilde, welche sich ein ähnliches Prinzip zunutze machen.

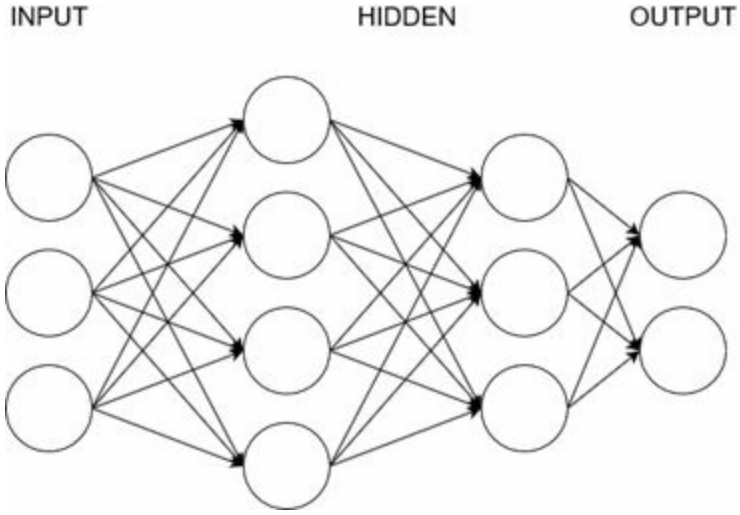


Abb. A.1: Aufbau eines Neuronalen Netzes

Der Aufbau eines Neuronalen Netzes sieht zunächst ziemlich simpel aus. Wir sehen in dieser Abbildung drei Schichten. Einmal die Eingabeschicht, einmal die Ausgabeschicht und dazwischen, zwei sogenannte *Hidden Layer*.

Die Eingabeschicht ist für unsere Inputs. Dort kommen die Dinge hinein,

welche wahrgenommen oder eingegeben werden. Wenn wir zum Beispiel wissen wollen, ob ein Bild eine Katze oder einen Hund darstellt, würden hier die Pixel eingegeben werden. Wenn wir herausfinden wollen, ob eine Person übergewichtig ist oder nicht, würden hier Parameter wie Größe, Gewicht etc. eingegeben werden.

Die Ausgabeschicht liefert uns dann die Ergebnisse. Dort sehen wir dann den Output, welcher durch die Inputs generiert wurde. Also eine Klassifikation, Vorhersagewerte oder ähnliches.

Alles dazwischen sind Abstraktionsschichten, welche die Komplexität des Systems erhöhen und die interne Logik erweitern. Das sind unsere Hidden Layer. In der Regel kann man sagen, dass das System umso umfangreicher und komplexer ist, umso mehr Hidden Layer und umso mehr Neuronen es hat.

AUFBAU VON NEURONEN

Um zu verstehen, wie ein Neuronales Netz im Allgemeinen funktioniert, müssen wir uns zunächst die einzelnen Neuronen ansehen.

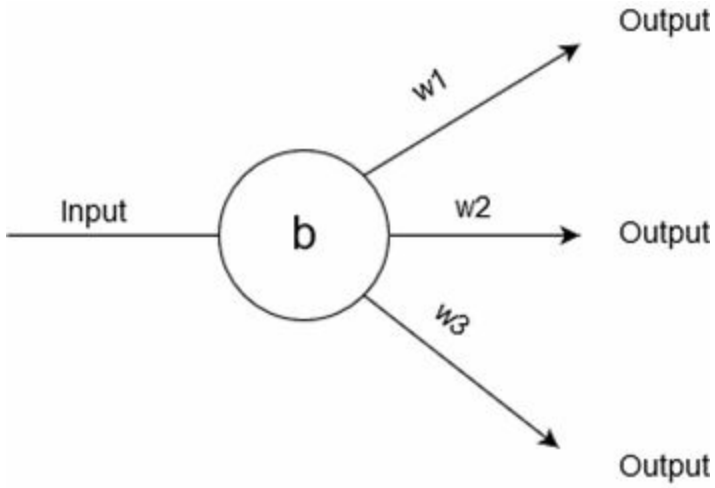


Abb. A.2: Aufbau eines künstlichen Neurons (Perzeptron)

Wie Sie sehen, hat ein Neuron einen gewissen Input, welcher, entweder der Output eines anderen Neurons ist, oder der Input in der ersten Schicht.

Dieser Input ist ein numerischer Wert und er wird nun mit jedem einzelnen *w* (steht für *weight* – zu Deutsch: Gewicht) multipliziert. Dann wird noch der

sogenannte *bias b* abgezogen. Das Ergebnis ist der jeweilige Output. Diese Outputs gehen dann weiter zu anderen Neuronen.

Was ich gerade erklärt habe und was Sie auf dem Bild sehen, ist jedoch eine veraltete Version eines Neurons und zwar das Perzeptron. Heutzutage benutzen wir weitaus komplexere Neuronen, wie das Sigmoid-Neuron, welche umfangreichere mathematische Funktionen nutzen, um den Wert der Outputs zu ermitteln.

Dann können Sie sich in etwa vorstellen, wie komplex ein System wird, wenn alle diese Neuronen, sich gegenseitig beeinflussen.

Zwischen unseren Inputs und den Outputs, liegen oftmals zahlreiche Hidden Layer mit Hunderten und Tausenden Neuronen. Diese Abstraktionsebenen führen dann zu dem Ergebnis.

Nun da wir verstehen, was Neuronale Netze sind, können wir uns etwas detaillierter mit deren Funktionsweise befassen.

AKTIVIERUNGSFUNKTIONEN

Zu diesen Perzeptronen kommen nun jedoch zahlreiche *Aktivierungsfunktionen* hinzu, welche das Ganze komplexer machen. Das sind jene Funktionen, welche die Outputs eines Neurons ermitteln. Wir nehmen also unsere Inputs, Gewichte und den Bias und übergeben das Resultat an eine Aktivierungsfunktion. Diese bestimmt dann das finale Ergebnis.

SIGMOID-AKTIVIERUNGSFUNKTION

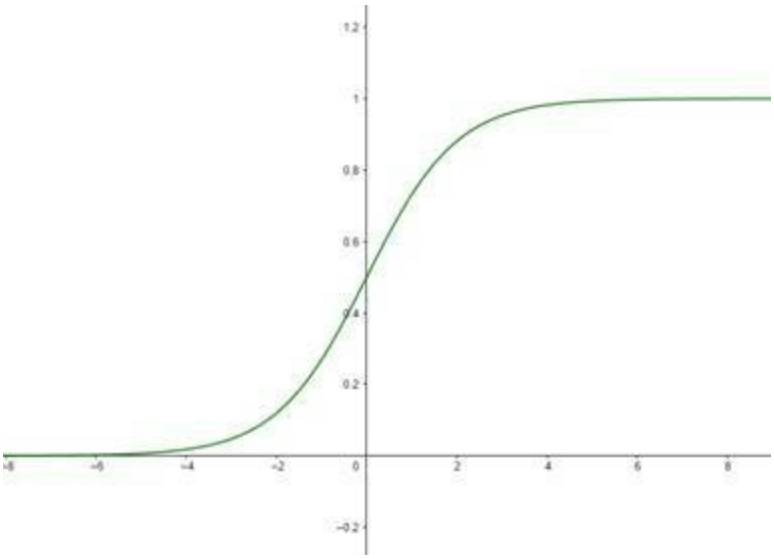


Abb. 1.1: Sigmoid-Aktivierungsfunktion

Eine verbreitete und gerne benutzte Aktivierungsfunktion ist beispielsweise die Sigmoid-Aktivierungsfunktion. Diese liefert immer einen Wert zwischen Null und Eins. Je kleiner der Input, desto näher ist der Output bei Null und je größer, desto näher ist der Output bei Eins.

Die mathematische Funktion sieht folgendermaßen aus:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Sie müssen diese Funktion nicht genau verstehen, wenn Sie Probleme damit haben. Was jedoch durch den Einser im Zähler ersichtlich wird, ist, dass die Funktion, immer einen Wert zwischen Null und Eins zurückliefert, da der Nenner immer positiv ist.

RELU-AKTIVIERUNGSFUNKTION

Die wahrscheinlich derzeit am häufigsten benutzte Aktivierungsfunktion ist die sogenannte ReLU-Aktivierungsfunktion. ReLU steht hierbei für *Rectified Linear Unit*, was so viel bedeutet, wie *Gleichgerichtete lineare Einheit.*

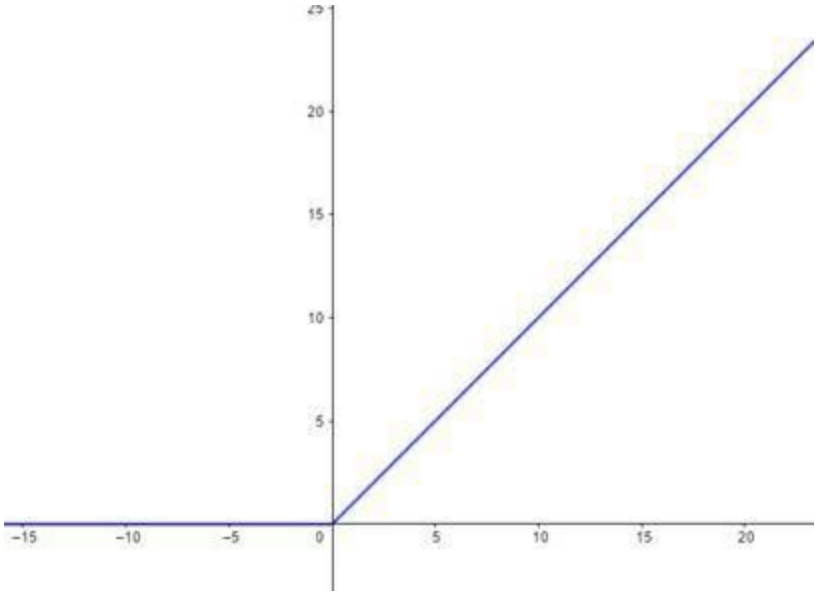


Abb. 1.2: ReLU-Aktivierungsfunktion

Diese Funktion ist außerordentlich simpel und dennoch sehr nützlich. Falls der Eingangswert negativ ist, wird immer Null zurückgeliefert. Sollte der Eingangswert positiv sein, so wird dieser immer direkt zurückgeliefert.

Mathematisch aufgeschrieben, wäre das Ganze:

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

Obwohl diese Funktion so simpel ist, erfüllt sie ihren Zweck und wird regelmäßig als Aktivierungsfunktion für die meisten Hidden-Layer bei Klassifikationen verwendet.

Es gibt auch noch jede Menge anderer Aktivierungsfunktionen. Das Ziel dieses Abschnitts ist es nicht Ihnen alle vorzustellen, sondern zu verstehen, was eine Aktivierungsfunktion macht. Wir werden in den späteren Kapiteln dann jeweils die nötigen Funktionen anwenden.

ARTEN VON NEURONALEN NETZEN

Neuronale Netze unterscheiden sich jedoch nicht nur anhand der Aktivierungsfunktionen der einzelnen Neuronen. Es gibt mehrere Arten von Neuronalen Netzen. In diesem Buch, werden wir uns einige ansehen und hier

in diesem Kapitel, erhalten Sie einen ersten Überblick.

FEED FORWARD NEURAL NETWORK

Die sogenannten *Feed Forward Neural Networks*, könnte man als die *klassischen* Neuronalen Netze bezeichnen, von welchen wir auch primär bis dato gesprochen haben. Hier wandert die Information immer nur in einer Richtung und zwar vom Input in Richtung Output. Es gibt keine Kreise oder Zyklen.

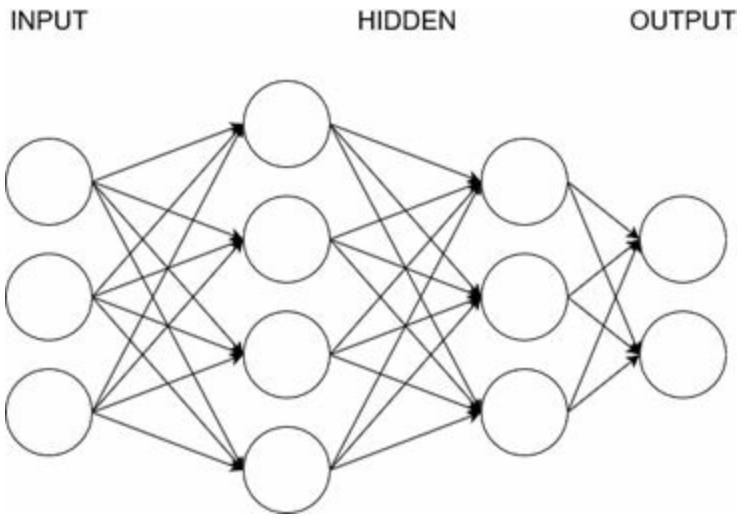


Abb. 1.3: Feed Forward Neural Network

Wie Sie sehen ist das die Grafik, von vorhin und wenn Sie genau hinsehen, merken Sie, dass die Verbindungen alle in eine Richtung zeigen. Die Information wandert nur nach vorne.

RECURRENT NEURAL NETWORK

Anders sieht es bei den sogenannten *Recurrent Neural Networks* aus (zu Deutsch: *Rekurrentes Neuronales Netz*). In diesen haben die Neuronen einer Schicht (Layer), nicht nur Verbindungen zu der nächsten Schicht, sondern auch zu Neuronen derselben Schicht oder zu vorherigen Schichten. Das wird auch *Rückkopplung* genannt.

Wenn wir den Ausgang eines Neurons als Eingang desselben Neurons benutzen, so sprechen wir von *direkter Rückkopplung*. Verbinden wir den Ausgang mit anderen Neuronen derselben Schicht, so sprechen wir von *seitlicher Rückkopplung*. Und sollten wir unseren Ausgang mit Neuronen

vorangegangener Schichten verbinden, so ist das eine *indirekte Rückkopplung*.

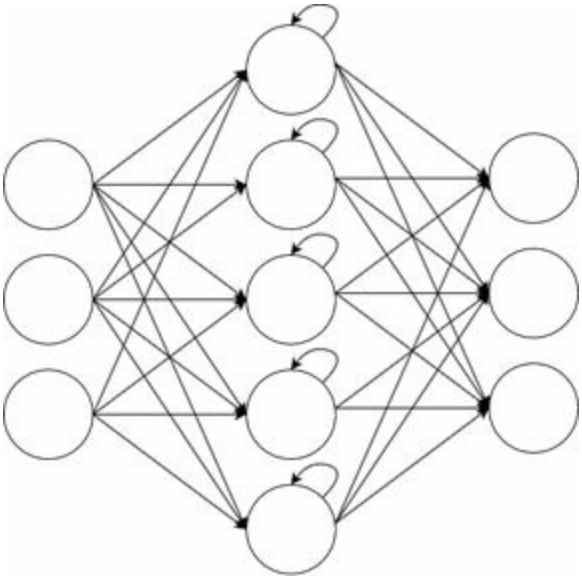


Abb. 1.4: Recurrent Neural Network (direkte Rückkopplung)

Der Vorteil eines solchen Rekurrenten Neuronalen Netz ist, dass es als Eingabe nicht nur die aktuellen Werte nimmt, sondern auch noch die Erinnerung, an die letzten Werte. Man könnte sagen, dass es mehrere Iterationen „zurückblickt“.

Diese Art von Neuronalen Netzen wird gerne bei Sprach- und Texterkennung benutzt bzw. allgemein bei sequentiellen Daten, da die Rückkopplung sich hier oftmals als sehr nützlich erweist. Für Bildverarbeitung bzw. –erkennung ist sie jedoch ungeeignet.

CONVOLUTIONAL NEURAL NETWORK

Dafür gibt es nämlich die sogenannten *Convolutional Neural Networks*. Diese werden primär für Verarbeitung von Bild- und Audiodateien verwendet. Prinzipiell ist diese Art von Neuronalem Netz besonders effektiv wenn es darum geht, Muster in Daten zu erkennen. Dabei kann es sich um allerlei Daten handeln. Wir werden in dem passenden Kapitel etwas genauer auf die Funktionsweise und Struktur von diesen Neuronalen Netzen eingehen, doch für dieses Kaptiel, wollen wir uns zumindest oberflächlich ansehen, was hier passiert.

Abb. 1.5: Xs und Os für eine Klassifikation

Nehmen wir ein ganz einfaches Beispiel. Hier haben wir mehrere Xs und Os als Beispiele gegeben, im 16x16 Pixel Format. Jedes Pixel ist ein Input-Neuron und soll verarbeitet werden. Am Ende soll unser Neuronales Netz bestimmen, ob es sich hierbei um ein X oder ein O handelt.

Dieses Beispiel ist natürlich trivial, da es auch mit einer simplen K-Nearest-Neighbors Klassifikation, oder einer Support Vector Machine gelöst werden kann. Aber wir benutzen es nur um ein Prinzip zu veranschaulichen.

Für uns Menschen sind diese Bilder eindeutig zu unterscheiden, da alle Xs und Os sehr ähnlich sind. Für einen Computer jedoch sind diese Bilder gänzlich verschiedene Inputs. Wie gesagt, ist das hier kein Problem, doch wenn es sich um die Klassifikation von Hunden oder Katzen bei einer höheren Auflösung handelt, wird alles schon um einiges komplexer.

Was Convolutional Neural Networks nun machen ist, anstatt sich nur die einzelnen Pixel anzusehen, suchen sie nach sogenannten Features.

Die meisten Xs haben zum Beispiel ein ähnliches Zentrum, mit vier Pixeln, von welchem vier Stränge weggehen. Außerdem haben sie lange diagonale Linien. Os wiederum haben ein leeres Zentrum und kürzere Linien als Abschnitte. Bei Katzen (im Vergleich zu Hunden) könnten Schnurrhaare zum Beispiel so ein Feature sein.

Das ist alles wie gesagt sehr oberflächlich erklärt und wir werden uns das Ganze etwas genauer im dazugehörigen Kapitel ansehen. Doch im Prinzip

suchen sich Convolutional Neural Networks einfach nur die herausstechenden Features heraus und gewichten diese.

TRAINIEREN UND TESTEN

Damit ein Machine Learning Modell die richtigen Vorhersagen trifft, müssen wir es zunächst trainieren und testen. Hierzu nehmen wir die gesamten, bereits ausgewerteten (z.B. klassifizierten) Daten und teilen diese in Trainings- und Testdaten auf. Meistens nimmt man etwa 20% der Daten als Testdaten und 80% als Trainingsdaten. Mit letzteren „zeigen“ wir unserem Modell, was wir uns erwarten und wie menschliche Experten es bereits ordnungsgemäß gemacht haben. Im Falle von Katzen- und Hundebildern könnten wir zum Beispiel 8000 Fotos in unser Neuronales Netz einlesen und diesem mitteilen, um welche Art von Tier es sich nun handelt (mehr zum technischen Ablauf folgt in Kürze).

Wenn unser Netz sich den Trainingsdaten angepasst hat, prüfen wir dessen Genauigkeit mit den Testdaten. Das sind Daten bzw. Bilder, welche unser Modell noch nie gesehen hat. Doch auch hier haben wir bereits die Lösungen parat. Wir schauen uns nun also an, wie unser Modell diese unbekannten Bilder klassifiziert. Anhand der Ergebnisse sehen wir, wie gut unser Neuronales Netz ist.

ERROR UND LOSS

Bei der Ermittlung der Genauigkeit bzw. der Performance unseres Neuronalen Netzes, gibt es zwei entscheidende Metriken, nämlich *Error* und *Loss*.

Ich möchte hier nicht zu genau auf die Berechnung dieser beiden Werte eingehen, da vor allem Loss oftmals zu Verwirrungen führt. Im Grunde genommen kann man sagen, dass der Error angibt, wie weit unsere Vorhersagen, von den gewünschten Ergebnissen abweichen. Das ist ein relativer Wert und dieser wird in Prozenten ausgedrückt. Ein Error von 0.21 würde zum Beispiel bedeuten, dass 79% der Bilder richtig klassifiziert wurden und 21% falsch. Diese Metrik ist für Menschen verständlich.

Beim Loss wird das Ganze schon etwas komplizierter. Hier gibt es meistens

eine sogenannte *Loss Function*, welche den Wert berechnet. Dieser Wert sagt uns dann, wie schlecht unser Modell *performt*. Je nach gewählter Loss Function, sieht dieser Wert anders aus. Das ist jedenfalls jener Wert, welchen wir minimieren wollen, um unser Modell zu optimieren.

GRADIENT DESCENT

Das Minimieren dieses Wertes erfolgt durch den sogenannten *Gradient Descent* Algorithmus (*Gradientenverfahren)*. Die Mathematik hinter diesem ist für viele Leute etwas verwirrend, aber ich werde mein bestes versuchen, um Ihnen alles so einfach wie möglich zu erklären.

Nehmen wir an, wir hätten eine Loss Function gegeben, welche in etwa so aussieht (triviales Beispiel):

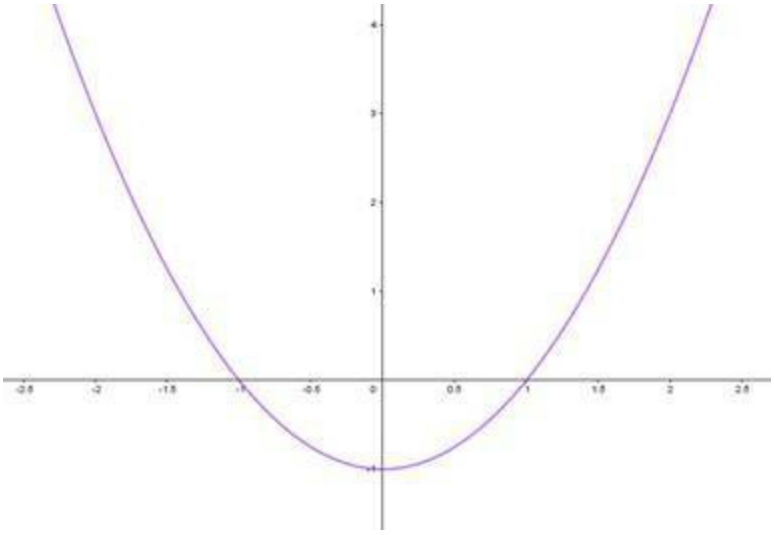


Abb. 1.6: Triviale Muster-Loss Function

Natürlich würde eine tatsächliche Loss Function nicht so aussehen, da sie wenig Sinn machen würde, doch wir schauen uns dieses simple Beispiel zunächst einmal an.

Wie bereits gesagt, ist es unser Ziel, den Output dieser Funktion zu minimieren. Wir suchen also den x-Wert, für welchen wir den kleinsten y-Wert herausbekommen. In diesem Fall ist es, wie unschwer zu erkennen ist, der x-Wert 0, welcher -1 als Ergebnis liefert. Doch wie findet unser Algorithmus das heraus?

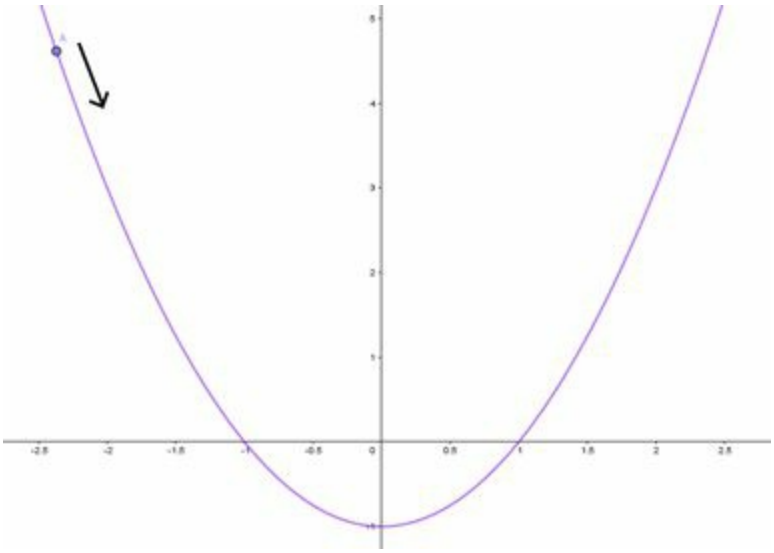


Abb. 1.7: Gradient Descent Visualisierung

Da der Computer, im Gegensatz zu uns, keinen Graphen vor Augen hat, aus welchem klar hervorgeht, wo sich das Minimum befindet, beginnen wir, indem wir einen zufälligen Startpunkt (A) auswählen. Dann errechnen wir die Steigung in diesem Punkt und erfahren, in welche Richtung wir uns bewegen müssen, damit der y-Wert steigt. Dann nehmen wir die entgegengesetzte Richtung (also jene, in welche der y-Wert abfällt) und machen einen sehr kleinen Schritt in diese. Bei dem neuen Punkt angekommen, wiederholen wir diesen Prozess. Das tun wir so lange, bis wir in einem Tal ankommen, wo die Steigung Null ist (also beim lokalen Minimum).

Sie können sich das etwa so vorstellen, als würden Sie einen Ball hinunterrollen lassen. Er würde unweigerlich zum tiefsten lokalen Punkt rollen.

Die Betonung liegt hierbei auf *lokal*. Betrachten wir zum Beispiel einmal die folgende Funktion.

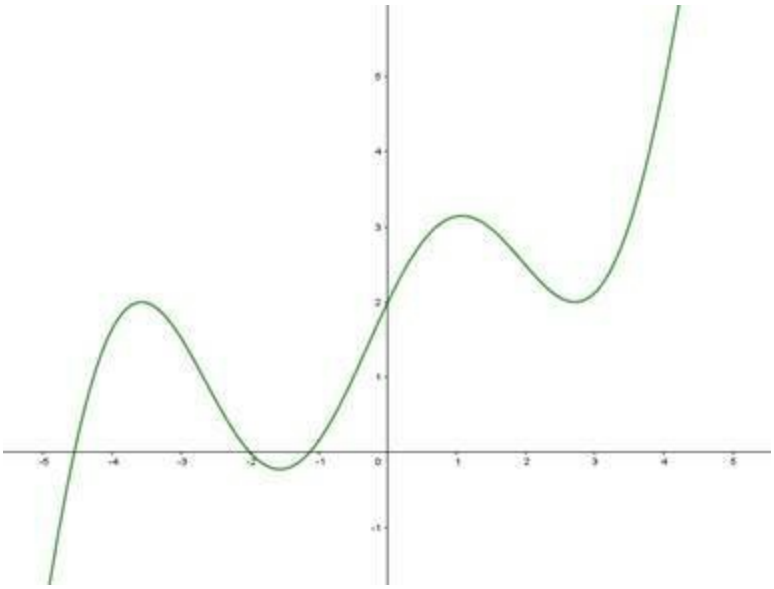


Abb. 1.8: Komplexere Beispielsfunktion

Das Problem bei dieser Funktion ist, dass es mehr als ein Minimum gibt. Es gibt quasi mehrere „Täler“. In welchem Tal wir landen, hängt davon ab, welche Startposition wir wählen. Hier mag das leicht sein, doch wir werden gleich sehen, warum dieser Zufallsfaktor nicht so leicht zu umgehen ist.

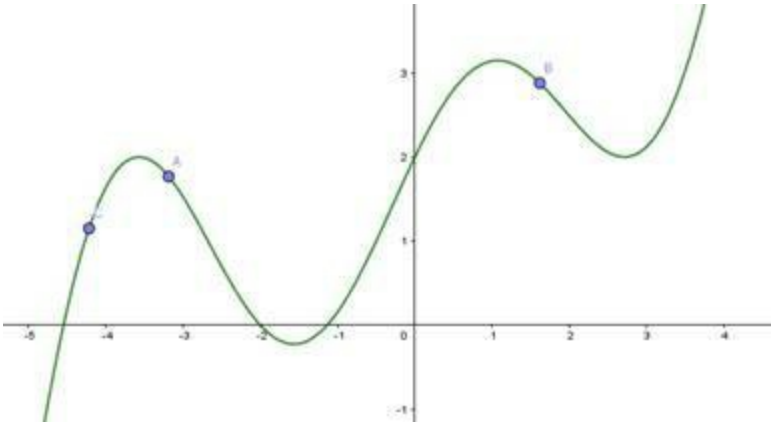


Abb. 1.9: Drei verschiedene Startpunkte

Hier sehen wir drei verschiedene Startpunkte in derselben Funktion. Wenn wir den Startpunkt B erwischen, kommen wir zwar in ein Tal, doch dieses Tal ist nicht einmal ansatzweise so niedrig oder optimal wie das Tal, in welches uns der Startpunkt A befördern würde. Ebenso gibt es den Startpunkt C, welcher in ein noch tieferes Tal führt (vorausgesetzt, dass dort überhaupt eins vorzufinden ist).

Zahlreiche Features

Das ist bis dato alles schön und gut, doch die Beispielsfunktionen, welche wir uns bisher angesehen haben, hatten alle einen Eingangswert und einen Ausgangswert.

Ein Neuronales Netz wiederum hat tausende, wenn nicht sogar mehr, Gewichte, Biases und andere Parameter, welche man anpassen kann, um die Ausgänge zu verändern. Unserer tatsächlichen Loss Function übergeben wir alle unsere Weights und Biases und diese Funktion gibt uns dann, anhand von den erwarteten und den tatsächlichen Ergebnissen, den Loss zurück.

Um ein wenig mehr Gefühl für das Ganze zu bekommen, betrachten wir zunächst eine drei-dimensionale Funktion.

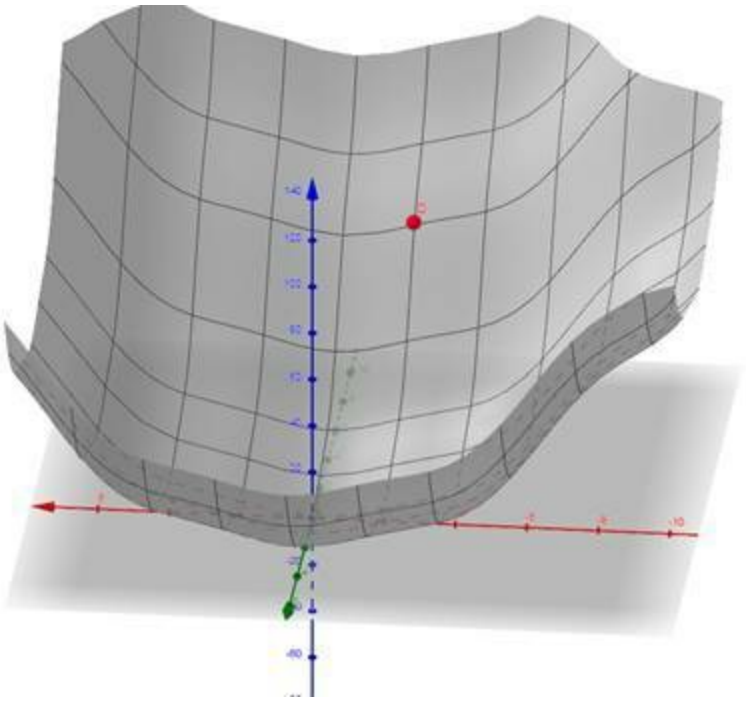


Abb. 1.10: Drei-Dimensionale Beispielsfunktion

Hierbei ist das Prinzip dasselbe. Wir haben einen Punkt (diesmal in einem drei-dimensionalen Raum) und dieser Punkt, soll in das Minimum „rollen“.

Der Unterschied hier ist jedoch, dass wir nicht nur nach links oder rechts können, sondern in jede mögliche Richtung der Ebene. Wir müssen nun also die Steigung relativ zur jeweiligen Achse betrachten. Mathematisch

gesprochen, müssen wir uns also mit partieller Ableitung bzw. partiellem Differenzieren befassen.

Wir betrachten also in dem Punkt zunächst, wie sich der Wert auf der Ausgangsachse (vertikal nach oben) verändert, wenn wir den Wert auf der ersten Eingangsachse verändern. Welche Richtung ist jene, mit der größten Steigung im Bezug zu diesem Eingangswert. Diese Richtung negieren wir wieder. Das Ganze wiederholen wir dann für die zweite Eingangsachse. Am Ende gehen wir wieder einen minimalen Schritt in die resultierende Richtung. Also in jene, mit dem stärksten Abfall. So rollen wir auch hier unweigerlich in das lokale Minimum.

Und diesen Prozess können Sie sich nur in mehreren tausend Dimensionen darstellen. Jedes Gewicht und jeder Bias ist eine Achse bzw. eine zusätzliche Dimension. Unser Algorithmus muss also alle diese Parameter berücksichtigen und so verändern, dass das Resultat minimal wird. Natürlich können wir das Ganze ab der vierten Dimension bereits nicht mehr wirklich visuell darstellen, da es unsere Vorstellungskraft übersteigt.

Die Mathematik

Wir werden uns nicht allzu viel mit der detaillierten Mathematik hinter diesem Verfahren auseinandersetzen, doch es ist durchaus hilfreich, sich zumindest einmal die mathematische Notation anzusehen.

Nehmen wir an, wir hätten eine Loss Function, welche wie folgt aussieht:

$$C(w,b) \equiv \frac{1}{2n} * \sum_{x} ||f(x) - y||^2$$

Was diese Funktion im Klartext macht ist folgendes: Wir übergeben als Parameter die Weights und die Biases. Dann berechnen wir die Summe aller Unterschiede, zwischen den Vorhersagen des Modells und den gewünschten Ergebnissen. Hierbei ist *f(x)* die Vorhersage des Neuronalen Netzes und *y* das gewünschte Ergebnis. Wir berechnen den Betrag des Unterschiedes, damit wir jedenfalls einen positiven Wert haben. Anschließend quadrieren wir jeden Unterschied. Zu guter Letzt dividieren wir das Ganze durch zwei Mal die Anzahl der Elemente.

Noch vereinfachter gesagt: Wir nehmen alle Unterschiede zwischen den Ergebnissen, quadrieren sie, summieren sie auf und dividieren sie durch die doppelte Anzahl der Elemente, um einen durchschnittlichen Loss zu ermitteln.

Das ist die Loss Function und nun wollen wir diese minimieren, mittels Gradient Descent.

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \ldots, \frac{\partial C}{\partial v_n}\right)$$

Wir wollen die Steigung von dieser Funktion berechnen. Diese setzt sich zusammen aus allen partiellen Ableitungen von C. Die Vektoren *v1, v2* usw. sind hierbei die Vektoren von den einzelnen Weights und Biases.

Das einzige was wir nun tun müssen, ist das inverse dieser Steigung herzunehmen und dann einen kleinen Schritt in diese Richtung zu machen.

$$-\nabla C * \varepsilon$$

Wir nehmen also die negative Steigung und multiplizieren diese mit einem minimalen Wert.

Machen Sie sich keine Sorgen, sollten Sie die Mathematik hier nicht komplett durchblicken. Der Fokus dieses Buches liegt auf der Anwendung. Sie können also auch weiterlesen, sollten Sie die Mathematik nicht ganz verstehen.

BACKPROPAGATION

Wir wissen also nun was nötig ist, um unser Neuronales Netz zu optimieren. Die Frage ist jedoch, wie wir das Ganze umsetzen. Wie berechnen wir die Steigung? Wie passen wir die einzelnen Weights und Biases an? Dafür gibt es den *Backpropagation* Algorithmus. Auch hier werde ich mein bestes versuchen, um Ihnen das Ganze so simpel wie möglich zu erklären.

Im Grunde genommen ist Backpropagation einfach jener Algorithmus, welcher die Steigung für den Gradient Descent berechnet. Wir ermitteln damit also, in welche Richtung, wir welche Parameter, wie stark verändern

müssen, um ein besseres Ergebnis zu erhalten.

Dabei geht dieser folgendermaßen vor: Zunächst wird die Vorhersage des Modells, mit dem gewünschten Ergebnis verglichen.

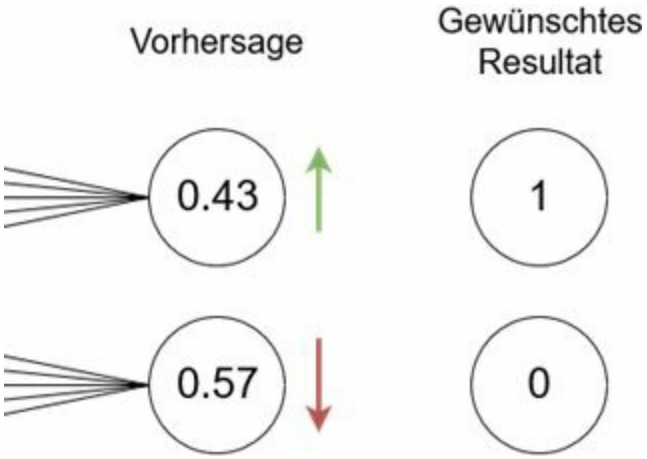


Abb. 1.11: Vergleich des Output-Layers

Was wir hier sehen ist der Output-Layer (mit zwei Neuronen) im Vergleich zu den gewünschten Resultaten. Wenn das erste Neuron beispielsweise jenes wäre, welches anzeigt, dass das Bild ein Hund ist und das zweite jenes, welches anzeigt, dass es eine Katze ist, so würde hier ein Hundefoto falsch als Katzenfoto klassifiziert werden.

Wir schauen uns also nun an, wie wir die Resultate ändern müssten, um auf die gewünschten Ergebnisse zu kommen. Beachten Sie hierbei jedoch, dass wir keinen direkten Einfluss auf die einzelnen Neuronen haben, sondern nur auf die Gewichte und die Biases.

Wir möchten den Wert des ersten Neurons erhöhen und den Wert des zweiten Neurons verringern. Dafür müssen wir einen Layer zurückblicken.

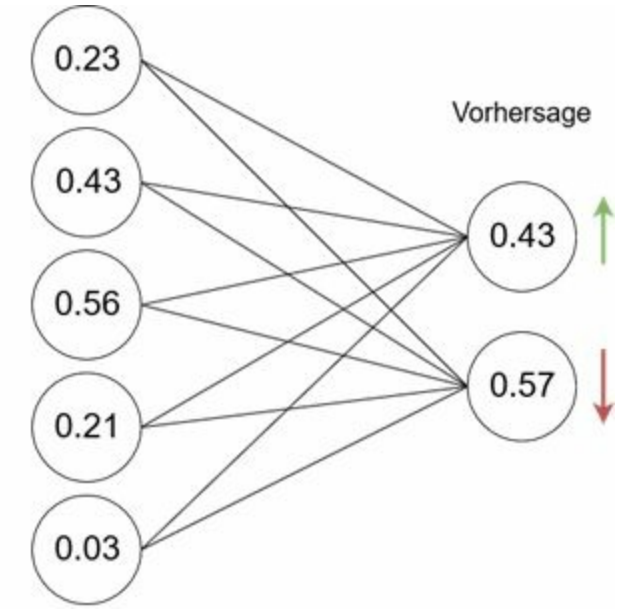


Abb. 1.12: Zurückführen auf vorherige Layer

Um den Wert eines Neurons zu verändern können wir entweder die Weights manipulieren, den Bias manipulieren oder aber die Inputs, und somit die Outputs der vorherigen Neuronen, verändern.

Wenn wir nun betrachten wie sich der finale Wert des ersten Output-Neurons zusammensetzt, werden wir sehen, dass manche Verbindungen den Wert erhöhen, während andere diesen verringern. Anders formuliert: Es gibt bestimmte Neuronen im vorangehenden Layer, welche bei einer Veränderung in eine bestimmte Richtung, dafür sorgen, dass unser Output-Neuron näher an den gewünschten Wert kommt.

Wir überlegen uns nun also, für jedes einzelne Neuron dieser Schicht, wie wir den Wert verändert haben möchten, sodass unsere Vorhersage akkurater wird. Behalten Sie hierbei wieder im Hinterkopf, dass wir die Aktivierung dieser Neuronen nicht direkt beeinflussen können. Wir haben nur Kontrolle über die Weights und Biases. Dennoch macht es Sinn, sich zu nächst einmal darüber klar zu werden, wie die einzelnen Neuronen sich letzten Endes im Idealfall verändern sollen.

Hierbei ist es wichtig, dass wir nicht vergessen, dass all dies bis dato nur für ein einziges Trainingsbeispiel durchgeführt wurde. Wir betrachten ein einziges Beispiel und wie dieses Beispiel die Neuronen geändert haben

möchte. Dieser Prozess muss jedoch für alle zigtausend Beispiele durchgeführt werden. Wir schauen uns also im Grunde genommen an, wie die Neuronen sich im Durchschnitt verändern sollen. Wenn 7000 Beispiele „verlangen“, dass ein bestimmtes Neuron hochgeschraubt gehört, während 3000 verlangen, dass dieses niedriger werden soll, wird es zwar hochgeschraubt, doch nicht so sehr wie wenn 10.000 Beispiele dies verlangen würden.

Das machen wir nun für jedes einzelne Neuron. Wenn wir also nun alle 10.000 Beispiele durchgespielt haben, wissen wir, wie sich die Neuronen im vorherigen Layer ungefähr verändern sollen. Und denselben Prozess führen wir hier für jedes einzelne Neuron erneut durch. Wie müssen sich die Neuronen im Layer vor diesem verändern, damit die einzelnen Neuronen sich so verändern, wie dies erwünscht ist?

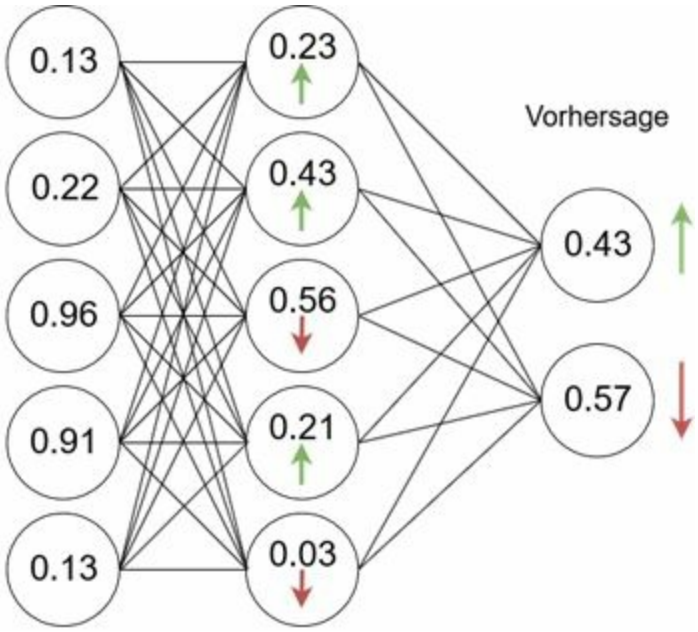


Abb. 1.13: Auf weitere Schicht zurückführen

Wenn wir diesen Prozess nun bis zum Input-Layer durchführen, erhalten wir die **durchschnittlich** geforderte Veränderung für alle Weights und Biases. Das entspricht dann genau der negativen Steigung des Gradient Descent Algorithmus. Wir nehmen eine kleine Veränderung vor und wiederholen den gesamten Prozess. Je nach Zeit, Rechenleistung und anderen Parametern, macht man das so lange, bis man mit den Resultaten (also der Genauigkeit) zufrieden ist.

ZUSAMMENFASSUNG

Da wir in diesem Kapitel eine Menge besprochen haben, hier noch einmal eine sehr grobe Zusammenfassung der Inhalte:

- Für Neuronen in Neuronalen Netzen gibt es sogenannte *Aktivierungsfunktionen*, welche den Ausgang bestimmen.
- Die „klassischen“ Neuronalen Netze sind *Feed Forward Neural Networks*. Dort wandert die Information nur in eine Richtung.
- In *Recurrant Neural Networks* arbeitet man mit Rückkopplung und es ist möglich die Ausgänge von zukünftigen Layern oder sogar von sich selbst als Eingang zu benutzen. Damit schafft man eine Art Gedächtnis.
- *Convolutional Neural Networks* werden primär für Bild- und Audioverarbeitung eingesetzt. Sie teilen die Daten in Features auf.
- In der Regel benutzen wir etwa 80% unserer bereits ausgewerteten Daten als Trainingsdaten und 20% als Testdaten.
- Der *Error* gibt an, wie viel Prozent der Testdaten falsch klassifiziert wurden. Der *Loss* ist eine Zahl, welche anhand einer *Loss Function* errechnet wird.
- Diese Loss Function gilt es zu Minimieren. Das tun wir mit dem sogenannten *Gradient Descent* Algorithmus.
- Dieser findet für uns das lokale Minimum der Funktion. Man kann sich das graphisch in etwa so vorstellen, dass ein Ball in das lokale Tal rollt.
- *Backpropagation* ist der Algorithmus, welcher die Steigung für den Gradient Descent Algorithmus errechnet.
- Das tut dieser indem er vom Output-Layer nach hinten in das Neuronale Netz die Werte anpasst.

2 - BIBLIOTHEKEN INSTALLIEREN

ENTWICKLUNGSUMGEBUNG

Nun da wir die einleitende Theorie hinter uns haben, kümmern wir uns um die Einrichtung unserer Entwicklungsumgebung. Grundsätzlich steht es Ihnen frei Ihre Entwicklungsumgebung selbst zu wählen. Ich empfehle jedoch eine professionelle Entwicklungsumgebung wie PyCharm oder das Jupyter Notebook von Anaconda.

PyCharm: https://www.jetbrains.com/pycharm/download/

Anaconda: https://www.anaconda.com/distribution/

BIBLIOTHEKEN

Ebenso werden wir für unsere Projekte ein paar externe Bibliotheken verwenden. Es ist zwar wichtig, dass wir die theoretischen Abläufe in Neuronalen Netzen verstehen, doch wir müssen uns das Leben nicht unnötig schwer machen und das Rad neu erfinden.

Aus diesem Grund werden wir für dieses Buch die Bibliothek *Tensorflow* benötigen. Dies ist die beliebteste Bibliothek für die Arbeit mit Neuronalen Netzen in Python, aber auch in anderen Programmiersprachen. Für die Installation verwenden wir *pip*:

```
pip install tensorflow
```

Im Laufe des Buches wird es öfters Stellen geben, an welchen andere externe Module benötigt werden. An diesen wird auch noch einmal darauf hingewiesen werden, dass die Module zu installieren sind. Mit den Standardbibliotheken für Data Science kann man jedoch nicht viel falsch machen:

```
pip install numpy
```

pip install matplotlib

pip install pandas

All diese Bibliotheken werden uns einen Großteil der Arbeit abnehmen. Wir müssen uns so gut wie um keine Mathematik kümmern, da wir mit diesen Bibliotheken auf der Anwenderebene arbeiten.

3 - HANDSCHRIFTERKENNUNG

Kommen wir nun langsam zu der Anwendung der ganzen Theorie. In diesem Kapitel wird es unser Ziel sein, ein Neuronales Netz zu erstellen und zu trainieren, welches dann, mit einer erstaunlichen Genauigkeit, handgeschriebene Ziffern erkennt. Unser Modell wird Ziffern von 0 bis 9 erkennen und klassifizieren können.

NOTWENDIGE BIBLIOTHEKEN

Für dieses Kapitel werden Sie folgende Imports benötigen:

```
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

Tensorflow ist hierbei die essentielle Bibliothek. Diese erlaubt uns die notwendigen Datensätze zu laden, die Neuronalen Netze aufzubauen etc. Die drei anderen Bibliotheken sind für die Funktionalität gar nicht notwendig. Wir importieren diese nur, um am Ende eigene Zahlen einzulesen.

Numpy werden wir für das umformatieren unserer eigenen Bilder benutzen und *Matplotlib* wird uns dabei behilflich sein, die numerischen Daten als Bilder darzustellen.

CV2 ist das *OpenCV* Modul und wird uns erlauben, unsere eigenen Bilder in das Programm einzulesen. Dieses Modul werden Sie noch installieren müssen:

pip install opencv-python

DATEN LADEN UND VORBEREITEN

Bevor wir nun mit dem tatsächlichen Aufbau und mit der Benutzung von Neuronalen Netzen beginnen, befassen wir uns zunächst damit, woher wir

unsere Daten bekommen und wie wir diese vorbereiten.

Für den Einstieg werden wir uns das klassische *MNIST Dataset* herunterladen. Dieses enthält 60.000 Trainingsbeispiele und 10.000 Testbeispiele von händisch geschriebenen Zahlen in der Auflösung von 28x28 Pixel. Wir werden das Untermodul *Keras* benutzen, um an diesen Datensatz zu gelangen.

```
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Um das MNIST Dataset auszuwählen greifen wir auf die *datasets* von *keras* zu und dann auf *mnist*. Dann rufen wir die Funktion *load_data* auf, um die Trainings- und Testdaten zu laden. Beachten Sie hierbei, dass diese Funktion zwei Tupel zurückliefert. Das erste enthält dabei die 60.000 Trainingsbeispiele, aufgeteilt in x- und y-Werte und das zweite enthält die 10.000 Testbeispiele, aufgeteilt in x- und y-Werte.

Um das Ganze jetzt noch etwas einheitlicher und kompakter zu machen, werden wir unsere Datensätze *normalisieren*. Das bedeutet, dass jeder Wert, auf einen Wert zwischen 0 und 1 herunterskaliert wird.

```
X_train = tf.keras.utils.normalize(X_train, axis=1)
X_test = tf.keras.utils.normalize(X_test, axis=1)
```

Das tun wir mit der *normalize* Funktion des *utils* Moduls von *keras*. Als *axis* geben wir hier 1 an. Nun haben wir unsere Trainings- und Testdaten optimal strukturiert und normalisiert. Wir sind bereit und können mit dem Aufbau des Neuronalen Netzes starten.

AUFBAU DES NEURONALEN NETZES

Jetzt geht es darum sich zu überlegen, welche Struktur für unsere Aufgabe Sinn macht. Da es sich um Bilder bzw. Scans handelt, wäre es naheliegend ein Convolutional Neural Network zu benutzen. Wenn wir uns die offizielle Seite des MNIST-Datensatzes ansehen, werden wir eine Tabelle vorfinden, welche verschiedene Arten und Strukturen von Neuronalen Netzen für diesen Datensatz vergleicht.

MNIST Webseite: http://yann.lecun.com/exdb/mnist/

Dort sehen wir auch tatsächlich, dass Convolutional Neural Networks außerordentlich gut abschneiden. Dennoch werden wir hier ein normales Feed Forward Neural Network verwenden, da dies zum einen völlig ausreicht, aber zum anderen auch weil wir die Convolutional Neural Networks in einem eigenen Kapitel besprechen werden. Es macht also Sinn mit den grundlegenden Strukturen zu beginnen.

```
model = tf.keras.models.Sequential()
```

Wir benutzen das Modul *models* von *keras*, um ein neues Neuronales Netz zu erstellen. Der *Sequential* Konstruktor erstellt dieses für uns. Nun haben wir also unser Modell, welches jedoch noch keine Layer hat. Diese müssen wir erst hinzufügen.

```
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
```

Um einen Layer hinzuzufügen, benutzen wir die Funktion *add* unseres Modells. Dann können wir von dem Modul *layers* einen Layer-Typen auswählen. Für den Input-Layer werden wir einen *Flatten* Layer, mit der Eingangsform 28x28 auswählen. Im Klartext bedeutet das, dass wir 784 Neuronen in unserem Input-Layer haben (Produkt aus 28 mal 28). Flatten gibt hierbei nur an, dass wir 784 Neuronen in einer Reihe haben anstatt in einem 28x28 Gitter. Diese Eingangsneuronen stellen die Pixel unserer Beispiele da. Unser Ziel ist es, dass wir durch diese Pixel am Ende auf eine Lösung kommen (also eine Klassifikation).

```
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
```

Als nächstes fügen wir zwei *Dense* Layer hinzu. Diese stellen unsere Hidden-Layer da und sollen die Komplexität unseres Modells erhöhen. Beide Layer haben jeweils 128 Neuronen. Als Aktivierungsfunktion haben wir hier, bei dem Parameter *activation*, die ReLU-Funktion gewählt (siehe Kapitel 1). Dense bedeutet hierbei nur, dass jedes Neuron dieser Schicht, mit jedem Neuron der letzten bzw. nächsten Schicht verbunden ist. Also ein ganz „normaler“ Layer.

```
model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
```

Zu guter Letzt definieren wir nun noch unseren Output-Layer. Auch dieser ist

hierbei ein Dense-Layer, hat jedoch nur zehn Neuronen und eine andere Aktivierungsfunktion. Die zehn Neuronen geben am Ende an, wie sehr unser Modell glaubt, dass es sich bei den eingegebenen Pixeln, um die jeweilige Zahl handelt. Das erste Neuron steht für die Null, das zweite für die Eins etc.

Die Aktivierungsfunktion, welche wir hier gewählt haben ist eine ganz besondere – die *Softmax* Funktion. Diese sorgt dafür, dass die Ergebnisse des Output-Layers eine Summe von 1 ergeben. Sie wandelt also die absoluten Werte in relative Ergebnisse um, sodass wir sehen zu wie viel Prozent eine Ziffer vorhergesagt wird.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
```

Zusammengefasst haben wir also einen Flatten Input-Layer mit 784 Neuronen für die Eingangspixel, zwei Hidden-Layer für die Komplexität und einen Output-Layer mit den Wahrscheinlichkeiten für die zehn möglichen Ziffern.

KOMPILIEREN DES MODELLS

Bevor wir nun mit dem Trainieren und Testen anfangen, müssen wir dieses Modell kompilieren. Dabei optimieren wir es und definieren ebenso die Loss Function.

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Wir werden hierbei nicht genau auf den *Optimizer* und auf die Loss Function eingehen. Die Parameter die hier übergeben werden sind jedoch eine gute Wahl für diese Aufgabe. Ebenso definieren wir hier die Metriken, welche uns interessieren. In diesem Fall fokussieren wir uns auf die Genauigkeit, also auf die *Accuracy*.

TRAINIEREN UND TESTEN

Nun geht es an den Hauptteil des Ganzen – das Trainieren und das Testen.

Hierzu benötigen wir jeweils nur eine Zeile.

```
model.fit(X_train, y_train, epochs=3)
```

Mit der *fit* Funktion trainieren wir unser Modell. Hierzu übergeben wir die x- und y-Trainingswerte. Als zusätzlichen Parameter geben wir die Anzahl der Epochen an. Diese Zahl gibt an, wie oft unser Modell dieselben Beispiele sehen wird.

```
val_loss, val_acc = model.evaluate(X_test, y_test)
print(val_loss)
print(val_acc)
```

Anschließend benutzen wir die *evaluate* Methode, um unser Modell zu testen. Dieser übergeben wir die Testdaten. Hier bekommen wir dann zwei Werte, nämlich den Loss und die Genauigkeit.

Meistens erhalten wir hier eine Genauigkeit von etwa 95%. Das ist außerordentlich gut, wenn man bedenkt, dass wir zehn mögliche Antworten haben. Ein zufälliges Raten würde uns also etwa 10% bringen. Unser Modell leistet also sehr gute Arbeit.

```
model.save('ziffern.model')
```

Mit der *save* Methode können wir unser Model nun speichern, damit wir es nicht jedes Mal erneut trainieren müssen. Wenn wir es dann laden möchten tun wir das mit der *load_model* Funktion.

```
model = tf.keras.models.load_model('ziffern.model')
```

EIGENE ZIFFERN KLASSIFIZIEREN

Nun wissen wir, dass unser Modell funktioniert, doch wir wollen es auch anwenden. Hierzu werden wir unsere eigenen handgeschriebenen Ziffern klassifizieren.

Dazu können Sie entweder, in Paint, die Zeichnung auf 28x28 Pixel stellen und mit der Maus oder einem Grafiktablett zeichnen oder Sie können tatsächliche handgeschriebene Zahlen einscannen und auf das benötigte Format herunterskalieren.

```
bild = cv2.imread('ziffer.png')[:,:,0]
```

```
bild = np.invert(np.array([bild]))
```

Um das Ganze dann in unser Skript zu bekommen, benutzen wir die *imread* Methode der OpenCV Bibliothek. Wir geben den Dateinamen an und benutzen dann das Slicing am Ende, um nur eine Spalte auszuwählen, da sonst das Format nicht passt. Damit unser Neuronales Netzt nicht verwirrt ist, müssen wir dann das eingelesene Bild invertieren.

```
vorhersage = model.predict(bild)
print("Vorhersage: {}".format(np.argmax(vorhersage)))
plt.imshow(bild[0])
plt.show()
```

Nun machen wir eine Vorhersage mit der *predict* Funktion. Diese liefert uns ein Array mit den zehn Werten des Output-Layers. Um daraus ein Resultat zu generieren, benutzen wir die *argmax* Funktion. Diese gibt uns die Position zurück, welche den höchsten Wert hat. Mit der *imshow* Funktion von Matplotlib können wir uns zu guter Letzt die Zahl auch anzeigen lassen.

Vorhersage: 7

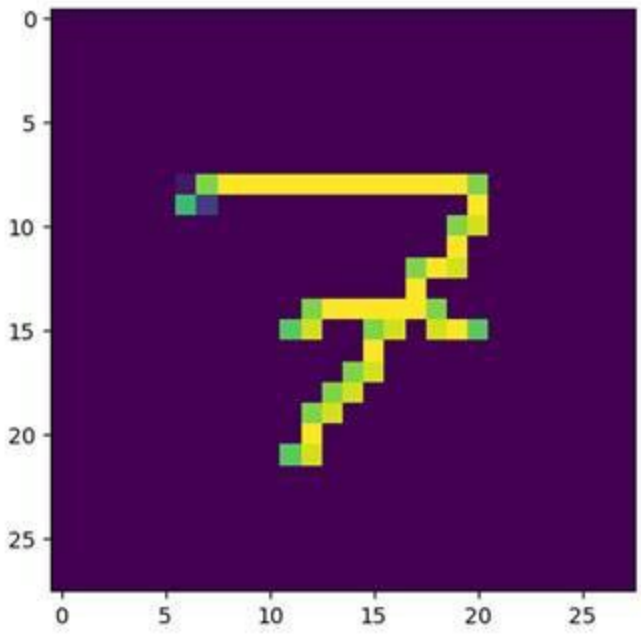


Abb. 3.1: Klassifizierte Zahl 7

Das Modell ist zwar sehr akkurat, doch auch hier können Fehler unterlaufen. Vor allem dann, wenn jemand eine unübliche Art hat bestimmte Zahlen zu

schreiben. Doch dieses Beispiel sollte Ihnen dabei geholfen haben, Neuronale Netze etwas besser zu verstehen.

4 – TEXTE GENERIEREN LASSEN

Oftmals haben wir es im Machine Learning mit sequentiellen Daten zu tun, also mit Daten, wo nicht jeder Input für sich betrachtet werden soll, sondern im Kontext zu den vorangehenden und nachfolgenden Daten.

RECURRENT NEURAL NETWORKS

Im ersten Kapitel haben wir bereits erwähnt, dass in solchen Fällen sogenannte *Recurrent Neural Networks* am sinnvollsten sind. Zur Wiederholung: Rekurrente Layer geben ihre Outputs nicht nur an den nächsten Layer weiter, sondern auch an den vorangehenden oder an den eigenen.

Besonders sinnvoll sind diese Neuronalen Netze also dort, wo wir mit zeitabhängigen Daten zu tun haben. Beispiele hierfür sind Wetterdaten und Aktienkurse, wo wir immer Rücksicht auf die Zeitverläufe nehmen müssen.

In diesem Kapitel werden wir uns jedoch mit der Generierung von Texten befassen. Texte können auch als sequentielle Daten angesehen werden, da nach jedem Buchstaben und nach jeder Kombination aus Buchstaben, bestimmte andere Buchstaben folgen. Hierbei ist es also wichtig, sich nicht nur den vorherigen Buchstaben anzusehen, sondern zum Beispiel die letzten 20 oder 30.

Was wir also hier vorhaben, ist unserem Modell Texte von Shakespeare oder Goethe zu zeigen, aus welchen es dann lernen soll, selbst ähnliche, neue Texte zu generieren.

LONG-SHORT-TERM MEMORY (LSTM)

Als wir im ersten Kapitel über Rekurrente Neuronale Netze gesprochen haben, haben wir uns die Ur-Version dieser Neuronen angesehen – einfache Neuronen, welche mit sich selbst, oder mit Neuronen vorangehender Layer,

verbunden sind.

Diese sind durchaus nützlich, doch es gibt mittlerweile Rekurrente Neuronen, welche um einiges mehr können und effektiver sind. Für dieses Beispiel werden wir die sogenannten *LSTM-Neuronen* betrachten. Das steht für *Long-Short-Term-Memory* und deutet darauf hin, dass diese Neuronen eine Art Gedächtnis haben.

Das Problem mit den handelsüblichen Rekurrenten Neuronen ist, dass diese unter Umständen wichtige Informationen vergessen können, da sie keine Mechanismen eingebaut haben, die differenzieren, welche Informationen relevant sind und welche nicht. Hierzu ein kleines Beispiel. Betrachten Sie die folgende Bewertung eines Produktes:

Wunderbar! Dieses Getränk schmeckt unglaublich toll und erinnert an eine Mischung aus Erdbeere und Kiwi. Ich habe nur die Hälfte getrunken, aber ich werde es mir definitiv wieder kaufen!

Wenn Sie diese Bewertung lesen und in zwei Tagen einem Freund davon erzählen möchten, so werden Sie sich definitiv nicht an jedes einzelne Wort erinnern. Viele Wörter wie „und“, „ich“, „werde“ und „dieses“ werden von Ihnen komplett vergessen werden, es sei denn Sie lesen sich den Text mehrmals durch und versuchen ihn sich zu merken.

In erster Linie würden sie sich an Ausdrücke wie „Wunderbar!“, „Mischung aus Erdbeere und Kiwi“ und „definitiv wieder kaufen!“ erinnern und diese auch wiedergeben. Und genau das tun auch LSTM-Netze. Sie filtern jene Information heraus, welche relevant sind und vergessen jene, welche irrelevant sind. Sie fokussieren sich aufs Essentielle. Durch diese Wörter wird klar, dass es sich hierbei um eine positive Bewertung handelt. Die anderen Wörter können genauso gut in einer negativen Rezension vorkommen.

Wir werden hier nicht auf die exakte Funktionsweise von LSTMs eingehen. Diese ist für dieses Buch zu umfangreich und zu komplex. Für uns ist hierbei in erster Linie wichtig, dass wir verstehen, dass ins LSTMs Mechanismen enthalten sind, um das Herausfiltern von relevanten Daten zu optimieren. Daher benutzen wir diese anstatt die standardmäßigen Rekurrenten Neuronen.

SHAKESPEARES TEXTE LADEN

Fangen wir zunächst einmal damit an, den Text für das Trainieren unseres Neuronalen Netzes in unser Skript zu laden. Hierfür werde ich Ihnen nun einige Alternativen anbieten.

Zum einen können Sie jenen Datensatz verwenden, welcher in den offiziellen Tensorflow Tutorials verwendet wird – den Datensatz von Shakespeare Texten. Dieser ist jedoch auf Englisch.

Alternativ werden wir uns auch ansehen, wie wir ähnliche Texte von Goethe, aber auch von Shakespeare auf Deutsch einlesen können.

Zu guter Letzt steht es Ihnen jedoch auch frei alle anderen möglichen Textdateien zu verwenden. Vielleicht finden Sie irgendwo eine Sammlung von Reden eines US-Präsidenten oder eines Geistlichen. Oder vielleicht exportieren Sie einfach WhatsApp-Chatverläufe und benutzen diese als Trainingsdaten. Das bleibt alles Ihnen überlassen. Achten Sie jedoch darauf, dass die Daten mehr oder weniger „sauber“ sind und ausreichend viel Text vorhanden ist.

```
import tensorflow as tf

path_to_file = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')
text = open(path_to_file, 'rb')\
   .read().decode(encoding='utf-8').lower()
```

Auf Grund der Seitenbreite, schaut der Code hier vielleicht etwas unschön formatiert aus. Lassen Sie sich dadurch jedoch nicht ablenken. Was wir hier tun ist nichts Kompliziertes. Zunächst benutzen wir die *get_file* Methode aus *keras.utils*, welche die Datei aus der URL lokal abspeichert (alternative URLs folgen gleich). Diese Funktion liefert unter anderem dem Dateipfad zurück. Dann benutzen wir einen ganz normalen File-Stream und lesen den Text aus der Datei ein. Am Ende wenden wir noch die *lower* Funktion auf unseren Text an, damit wir keine Unterschiede zwischen Klein- und Großbuchstaben erkennen. Das macht vieles leichter und ist für den Inhalt irrelevant.

Hier sind die Links für die jeweiligen Texte:

Shakespeare Englisch

https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt

Shakespeare Deutsch

https://archive.org/stream/shakespeareundd00kunsgoog/shakespeareundd00kunsgoog_djvu.txt

Goethe Deutsch

https://archive.org/stream/bub_gb_z9tJAAAAIAAJ/bub_gb_z9tJAAAAIAAJ_djvu.txt

Was hierbei sehr wichtig ist, ist, dass der Goethe Text sehr oft den String „digitized by google“ in sich hat. Wenn Sie mit diesem Text sinnvoll arbeiten möchten, lohnt es sich alle Vorkommnisse zu entfernen.

```
text = text.replace('digitized by google', '')
```

Das machen Sie am besten mit der *replace* Funktion.

DATEN VORBEREITEN

In diesem Buch werden wir als Beispiel mit dem deutschen Goethe Text arbeiten. Hier also nochmal die ersten Zeilen für das Laden des Textes.

```
dateipfad = tf.keras.utils.get_file('goethe.txt',
'https://archive.org/stream/bub_gb_z9tJAAAAIAAJ/bub_gb_z9tJAAAAIAAJ_djvu.txt')

text = open(dateipfad, 'rb')\
   .read().decode(encoding='utf-8').lower()

text = text.replace('digitized by google', '')
```

TEXT UMWANDELN

Das einzige was wir jetzt haben, ist jedoch nur eine große Textdatei mit sehr viel Text von Goethe. Diesen Datensatz müssen wir nun so bearbeiten, dass dieser von einem Neuronalen Netz gelesen werden kann. Dafür müssen wir Textdaten in numerische Daten umwandeln.

Bevor wir das jedoch tun, ist es sinnvoll, je nach Leistungsfähigkeit Ihres Computers, nur einen Teil des Textes auszuwählen.

Wenn Sie einen leistungsstarken Computer oder viel Zeit haben, können Sie natürlich das Modell auf den ganzen Text trainieren. Das macht das Modell natürlich auch leistungsstärker und genauer. Was Sie jedoch auch tun können ist einfach einen Teil davon auszuwählen. Der Goethe Text hat in etwa 1,6 Millionen Zeichen. Für ein halbwegs ansehnliches Resultat reicht es auch 200.000 Zeichen für das Trainieren zu benutzen.

```
text = text[500000:700000]
```

Durch Index-Slicing können wir hier die Zeichen von Nummer 500.000 bis 700.000 benutzen. Das sind dann 200.000 Zeichen aus der Mitte des Textes. Wie bereits gesagt, können Sie selbst entscheiden, wie viel vom Text sie benutzen möchten. Je mehr Sie benutzen, desto genauer wird das Neuronale Netz, aber desto länger wird es auch dauern dieses zu trainieren.

```
zeichen = sorted(set(text))

zeichen_indizes = dict((z, i) for i, z in enumerate(zeichen))
indizes_zeichen = dict((i, z) for i, z in enumerate(zeichen))
```

Als nächsten Schritt wollen wir nun jedem Zeichen, welches in dem ausgewählten Text vorkommt, eine Zahl zuweisen. Hierfür kombinieren wir die *set* Funktion mit der *sorted* Funktion. Erstere bildet eine Collection aus Zeichen, wobei jedes nur ein einziges Mal vorkommt. Mit der zweiten Funktion sortieren wir diese Collection anschließend, aufsteigend nach ASCII-Wert.

Nun erstellen wir zwei Dictionaries, welche jeweils für jedes Zeichen ein Key-Value Pair, aus Zeichen und einer Zahl, bilden. In einem Dictionary haben wir hierbei alle Zeichen als Keys und deren dazugehörigen Zahlenwert als Value, während es im anderen Dictionary umgekehrt ist.

Beachten Sie hierbei, dass die *enumerate* Funktion, welche hier benutzt wird, nicht den ASCII-Wert berechnet, sondern einfach jedes Zeichen durchnummeriert.

Diese beiden Dictionaries werden wir nun benutzen, um Zeichen in Zahlen und dann diese Zahlen später wieder in Zeichen umzuwandeln. Das tun wir, da unser Neuronales Netz nur mit numerischen Werten arbeiten kann.

TEXT UNTERTEILEN

Als nächstes müssen wir nun unseren Text in Sub-Texte, also Sequenzen aufteilen. Hierfür definieren wir zunächst eine Sequenzlänge und eine Step-Size.

```
SEQUENZ_LAENGE = 50
STEP_SIZE = 3

saetze = []
naechstes_zeichen = []
```

Unser Ziel hier ist es, Features und Targets zu erzeugen. Als Features wollen wir unserem Neuronalen Netz am Ende gewisse Sätze als Inputs geben und dieses soll am Ende dann ein „nächstes Zeichen“ ausgeben. So soll dann ein Text generiert werden.

Die Sequenzlänge gibt hierbei an, wie lange diese Trainingstexteinheiten sein sollen. Wenn wir eine Länge von 50 auswählen, so werden wir dem Neuronalen Netz 50 Zeichen zeigen und dann das Zeichen, welches darauf folgen soll. Die Step-Size gibt nur an, wie viele Zeichen wir nach einem Trainingssatz überspringen bis zum nächsten Trainingssatz. Eine zu kleine Step-Size sorgt dafür, dass wir viel zu viele Beispiele haben, während eine zu große für mangelnde Performance sorgen kann. Bei der Sequenzlänge ist es ebenso wichtig eine Zahl zu wählen, welche groß genug ist, um dem Modell ganze Sätze zeigen zu können, jedoch nicht zu groß, damit es sich nicht auf zu viel vorangehende Daten verlässt.

Wir haben nun die zwei leeren Listen definiert, welche am Ende unsere Features und Targets für das Training beinhalten sollen. Die Sätze sind das was wir als Input wählen und die nächsten Zeichen sind der erwartete Output. Mit diesen beiden Listen trainieren wir dann unser Neuronales Netz. Wir müssen sie jedoch zunächst befüllen.

```
for i in range(0, len(text) - SEQUENZ_LAENGE, STEP_SIZE):
    saetze.append(text[i: i + SEQUENZ_LAENGE])
    naechstes_zeichen.append(text[i + SEQUENZ_LAENGE])
```

Hierfür lassen wir einfach eine For-Schleife über unseren Text, abzüglich der gewählten Sequenzlänge, laufen. Mit jeder Iteration, wird die Laufvariable *i*

um *STEP_SIZE* erhöht.

Zusätzlich wird in jeder Iteration der Abschnitt, von *i* bis *i* plus der Sequenzlänge, zu der Satzliste hinzugefügt. In unserem Fall werden also Sätze der Länge 50 hinzugefügt und dann verschieben wir das Anfangszeichen um drei Stellen. Ebenso wird dann jener Buchstabe, welcher auf diesen Text folgen würde, zu der Zeichenliste hinzugefügt.

UMWANDELN IN NUMPY-FORMAT

Nun müssen wir diese beiden Listen in ein brauchbares Numpy-Format für unser Modell transformieren.

```
x = np.zeros((len(saetze),SEQUENZ_LAENGE,
        len(zeichen)), dtype=np.bool)
y = np.zeros((len(saetze),
        len(zeichen)), dtype=np.bool)
```

Hierfür definieren wir zunächst zwei Numpy-Arrays, welche anfangs mit Nullen befüllt werden. Das Array *x* für die Features ist dreidimensional und die Form setzt sich aus der Anzahl der Sätze, der Sequenzlänge und der Anzahl der möglichen Zeichen zusammen. Beachten Sie, dass der Datentyp hierbei *bool* ist, was für *boolean* steht. Es handelt sich hierbei also um Wahrheitswerte und nicht um numerische Werte. In unserem x-Array wird später drinnen stehen, welche Zeichen in welchen Sätzen an welcher Stelle vorkommen.

Das y-Array für die Targets ist zweidimensional und die Form setzt sich aus der Anzahl der Sätze und der Anzahl der möglichen Zeichen zusammen. Auch hier arbeiten wir mit booleschen Werten. Nun müssen wir die beiden Arrays mit den richtigen Werten befüllen.

```
for i, satz in enumerate(saetze):
   for t, char in enumerate(satz):
      x[i, t, zeichen_indizes[char]] = 1
   y[i, zeichen_indizes[naechstes_zeichen[i]]] = 1
```

Der Code mag zunächst etwas verwirrend aussehen, doch er ist eigentlich ganz simpel. Alles was wir hier tun ist jene Zeichen, welche in einem Satz vorkommen, in unserem Array mit einer Eins zu markieren. Wir benutzen

hierbei die *enumerate* Funktion nur, damit wir wissen, welche Felder wir indexieren müssen. In unserem y-Array setzen wir das richtige Nachfolge-Zeichen ebenso auf Eins.

Da wir nur mit numerischen Werten arbeiten können, benutzen wir unser *zeichen_indizes* Dictionary, um für die jeweiligen Zeichen, die entsprechenden Indizes herauszufinden.

Um das alles ein wenig verständlicher zu machen, schauen wir uns ein Beispiel an. Nehmen wir an, dass das Zeichen *‘g‘* den Index 17 erhalten hat. Wenn dieses Zeichen nun im dritten Satz (also der Satz mit dem Index 2), an der vierten Stelle (also die Stelle mit dem Index 3), vorkommt, so würden wir das Feld *x[2,3,17]* auf Eins setzen.

REKURRENTES NETZ AUFBAUEN

Nun haben wir endlich brauchbare Daten, mit welchen auch unser Neuronales Netz arbeiten kann. Dieses müssen wir jedoch erst aufbauen. Hierfür benötigen wir folgende Bibliotheken:

```
import random
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.layers import Activation, Dense, LSTM
```

Die Bibliothek *random* werden wir für eine spätere Hilfsfunktion benötigen. *NumPy* haben wir bereits benutzt. Die grundlegende *Tensorflow* Bibliothek haben wir benötigt, um unsere Daten aus dem Internet zu laden.

Für unser gesamtes Neuronales Netz benötigen wir wieder das *Sequential* Model von Keras und als Optimizer werden wir diesmal *RMSprop* verwenden. Als ersten Layer werden wir einen *LSTM-Layer* benötigen, welcher von einem *Dense*- und einem *Activation-Layer* gefolgt werden wird. Dementsprechend müssen wir auch diese Strukturen aus Keras importieren.

```
model = Sequential()
model.add(LSTM(128,
        input_shape=(SEQUENZ_LAENGE,
                len(zeichen))))
```

```
model.add(Dense(len(zeichen)))
model.add(Activation('softmax'))
```

Unser Modell ist relativ simpel. Auch hier gilt wieder: Wenn wir mehr Rechenleistung und Zeit zur Verfügung haben, lohnt es sich definitiv ein komplexeres Netz aufzubauen, sodass unsere Ergebnisse besser werden.

In unserem Modell gelangen die Inputs zunächst in einen LSTM-Layer mit 128 Neuronen und einer Input-Shape, welche sich aus der Sequenzlänge und der Anzahl an Zeichen zusammensetzt. Wie dieser Layer grundlegend funktioniert haben wir bereits am Anfang dieses Kapitels besprochen. In dieser Schicht befindet sich quasi das Gedächtnis unseres Modells. Dieser Layer wird gefolgt von einem Dense-Layer, mit so vielen Neuronen, wie wir verschiedene Zeichen haben. Das ist quasi unser Hidden-Layer. Zu guter Letzt haben wir einen Output-Layer vom Typ *Activation*, welcher die Softmax-Funktion als Aktivierungsfunktion nutzt. Diese kennen wir bereits aus dem letzten Kapitel.

```
model.compile(loss='categorical_crossentropy',
        optimizer=RMSprop(lr=0.01))

model.fit(x, y, batch_size=256, epochs=4)
```

Nun optimieren wir unser Modell noch und kompilieren es. Hierbei nutzen wir diesmal einen anderen Optimizer, nämlich den *RMSprop* mit einer Lernrate von 0.01.

Dann trainieren wir unser Modell noch auf unsere Trainingsdaten mit einer *batch_size* von 256 und mit 4 Epochen. Die Batch-Size gibt an wie viele Datensätze wir dem Neuronalen Netz immer auf einmal zeigen.

HILFSFUNKTION

Unser Modell ist nun trainiert und bereit Text zu generieren. Um das jedoch zu tun, müssen wir den Output, welchen unser Neuronales Netz liefert auch irgendwie sinnvoll verarbeiten und in Text umwandeln. Hierzu benötigen wir eine kleine Hilfsmethode.

```
def sample(preds, temperature=1.0):
  preds = np.asarray(preds).astype('float64')
```

```
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Diese Methode habe ich aus dem offiziellen Keras-Tutorial kopiert.

Keras Tutorial: https://keras.io/examples/lstm_text_generation/

Diese Funktion erhält als Parameter die Vorhersagen unseres Modells (durch *model.predict(…)*) und ermittelt wählt dann ein Zeichen als nächstes Zeichen aus. Je höher hierbei der zweite Parameter *temperature* ist, desto „gewagter“ wird der Versuch. Ein niedriger Wert sorgt also für eine konservative Vorhersage, während ein hoher Wert experimenteller ist. Diese Hilfsfunktion werden wir in der finalen Funktion benötigen.

TEXTE GENERIEREN LASSEN

Jetzt fehlt uns nur noch die finale Funktion, mithilfe welcher wir unseren Text generieren lassen können.

```
def text_generieren(laenge, vielfalt):
  start_index = random.randint(0, len(text) - SEQUENZ_LAENGE - 1)
  generierter_text = ''
  satz = text[start_index: start_index + SEQUENZ_LAENGE]
  generierter_text += satz
  for i in range(laenge):
      x = np.zeros((1, SEQUENZ_LAENGE, len(zeichen)))
      for t, char in enumerate(satz):
          x[0, t, zeichen_indizes[char]] = 1

      vorhersagen = model.predict(x, verbose=0)[0]
      naechster_index = sample(vorhersagen,
                               vielfalt)
      naechstes_zeichen = indizes_zeichen[naechster_index]

      generierter_text += naechstes_zeichen
      satz = satz[1:] + naechstes_zeichen
  return generierter_text
```

Diese Methode ist etwas umfangreicher, doch wir werden Sie auch wieder Schritt für Schritt betrachten. Zunächst lassen wir mittels *random* eine zufällige Einstiegsstelle in den Text generieren. Von dieser Stelle aus nehmen

wir eine Sequenzlänge an Text als Anfangstext. Das ist notwendig, da unser Modell nicht einfach aus dem Nichts Texte generieren kann. Der erste Teil unseres Textes ist also aus dem Originaltext kopiert.

Diesen ersten Teil wandeln wir nun wieder, wie wir es bereits getan haben, in ein NumPy-Array um. Dann nehmen wir dieses her und lesen es in unser Modell ein, welches uns nun die Wahrscheinlichkeiten für die verschiedenen nächsten Zeichen vorhersagen soll.

Diese Wahrscheinlichkeiten nehmen wir dann und übergeben sie der Hilfsfunktion *sample*. Wie Sie vielleicht bemerkt haben, hat unsere Funktion zwei Parameter. Die Länge des zu generierenden Textes und die Vielfalt. Die Vielfalt ist gleichbedeutend mit der *temperature* der *sample* Funktion und wird dieser auch so übergeben.

Am Ende bekommen wir also von unserer Hilfsfunktion eine Vorhersage, in Form von einer Zahl, heraus. Diese Zahl müssen wir nun, mit unserem zweiten Dictionary, in ein Zeichen umwandeln, welches wir dann unserem Gesamttext hinzufügen. Wir wiederholen diesen Prozess so lange, bis wir unsere gewünschte Länge erreicht haben.

RESULTATE

Die Texte, welche generiert werden sind weit entfernt von Perfektion, doch es ist interessant sich einige Beispiele anzusehen.

Einstellung: *Länge: 500, Vielfalt: 0.4*

„die eitelkeit hat mich verführt der stinn

seiner franz schied ihren seiner sturen der lenzen hat den gefreungen zu den steht wieder den

gefühlte gegen den freuden verschieder in der den haupter sie ihrer seiner

den stur zu sein

Einstellung: *Länge: 500, Vielfalt: 0.6*

werther, daß jeder allgemeine satte absterder beiden

der schönen zurachen so den stutzen das in dem -

müchsten einer beiffenden der geffand den tiesen einer als sien in diemes wieden danals

sanz seine gegenden. der alberenbeiten

Einstellung: *Länge: 500, Vielfalt: 0.8*

gegenwart des allmächtigen, der

uns nach zu ausgewiedenus die ein beisternelen die ihren von dem unterstellen und wirder reine frindischen

das stahn dabs erwaser

erde gegen strehtten rede das

rückken. wie darauf gebinmes, von denen herz

Wie Sie sehen machen die Texte zwar nicht viel Sinn und es kommen vereinzelt Wörter vor, welche eigentlich gar keine sind. Dennoch ist es ziemlich beeindruckend, dass unser Modell gelernt hat diese Sätze selbst zu formulieren. Das sind keine kopierten Sätze aus dem Text, sondern selbstständig, Zeichen für Zeichen, formulierte Texte. Und das von einem Computer der nicht einmal weiß was ein Wort, geschweige denn ein Satz, ist.

5 – BILDER UND OBJEKTE ERKENNEN

In dem ersten Kapitel haben wir bereits erwähnt, dass sogenannte *Convolutional Neural Networks* besonders gut darin sind, Bilddaten zu verarbeiten. Sie erkennen Muster und Teilsegmente dieser Daten und liefern somit genauere Ergebnisse als handelsübliche Neuronale Netze.

FUNKTIONSWEISE VON CNNS

Im Grunde genommen erhalten die Convolutional Neural Networks ihren Namen dadurch, dass sie aus *Convolutional Layers* bestehen. Diese werden meist von *Pooling Layers* gefolgt, welche die Informationen dann noch einmal filtern. Ein Convolutional Neural Network kann aus beliebig vielen solcher Einheiten bestehen. Beispielsweise könnten Sie einen Input-Layer haben, dann einen Convolutional Layer, gefolgt von einem Pooling Layer, und diese Kombination drei Mal, bis Sie dann am Ende ein paar Dense Layer anhängen, welche zum Output-Layer führen.

CONVOLUTIONAL LAYER

Wie ein ganz gewöhnlicher Layer, erhält auch ein Convolutional Layer Inputs von der vorangehenden Schicht, verarbeitet diesen und leitet ihn weiter an den nächsten Layer. Bei Convolutional Layern ist die Verarbeitung eine sogenannte „Faltung“. Dazu gleich mehr.

Meistens ist ein Convolutional Layer zwei- oder dreidimensional. Wenn wir schwarz-weiß Bilder einlesen und klassifizieren, arbeiten wir mit zwei Dimensionen, während wir bei farbigen Bildern mit drei Dimensionen arbeiten. Die einzelnen Werte stellen hier jeweils die Pixel dar, sollte es sich um Bilder handeln.

Betrachten wir mal oberflächlich wie so eine Verarbeitung aussehen könnte.

Abb. 5.1: Bild eines Autos

Wenn Sie sich dieses Bild ansehen, merken Sie sofort, dass es sich hierbei um ein Auto handelt. Das liegt daran, dass sie schon zahlreiche Autos in Ihrem Leben gesehen und das Label „Auto“ gelernt haben. Ein Computer tut sich hier schwerer. Wenn wir einem Computer zehn mögliche Kategorien geben (Auto, Flugzeug, Katze, Hund, Tisch etc.) wird es für ihn von Natur aus unmöglich sein, bei Bildern verschiedenster Art die Objekte richtig zu klassifizieren.

Mit gewöhnlichen Neuronalen Netzen würden wir Pixel für Pixel einlesen und versuchen die Bilder zu klassifizieren. Das mag bei handgeschriebenen Zahlen leicht funktionieren, da wir es hier nur mit schwarzen Pixeln auf weißem Hintergrund zu tun haben, doch bei komplexeren Bildern versagen wir. Ein Auto kann von verschiedenen Perspektiven fotografiert werden. Ebenso kann der Hintergrund mal Hell, mal dunkel sein. Hier ist es also wichtig, dem Computer beizubringen die einzelnen Attribute und Muster herauszufiltern. Beispielsweise die Reifen, die Form des Autos, die Seitenspiegel etc. Und genau das tun Convolutional Layer. Sie filtern die relevanten Features aus den Bildern.

Abb. 5.2: Beispielhafte Analyse der Features

Wenn Sie darüber nachdenken ist das genau der Prozess den auch wir Menschen verwenden. Wir betrachten nicht jeden einzelnen „Pixel“ unseres Sichtfeldes, um dann zu überprüfen, ob es sich um ein Auto handelt. Wir schauen uns das große Gesamte an. Wir sehen Reifen, das Kennzeichen, die Form, die Windschutzscheibe und bevor wir überhaupt Zeit haben nachzudenken, wissen wir, dass das was wir sehen ein Auto ist.

Gehen wir nun etwas tiefer auf die technischen Details von Convolutional Layern ein. Im Grunde genommen ist ein Convolutional Layer einfach nur eine Matrix, welche auf unsere Daten angewandt wird.

0.762	0.561	0.022
0.675	0.132	0.982
0.111	0.671	0.231

Nehmen wir diese 3x3 Matrix einmal als Beispiel. Diese Matrix ist jetzt unser Filter bzw. unser Convolutional Layer. Zusätzlich zu dieser Matrix können wir uns nun auch ein Bild vorstellen, welches auf dieselbe Art und Weise kodiert ist. Das bedeutet, dass jedes Pixel einen Wert von 0 bis 1 zugewiesen bekommt. Je höher der Wert, desto dunkler ist das Pixel, also desto näher ist es bei Schwarz, und je niedriger der Wert, desto heller bzw. desto näher ist das Pixel bei Weiß. Was wir nun also tun ist, unsere Matrix auf jedes einzelne 3x3 Feld unseres Bildes anzuwenden. Anwenden bedeutet hierbei, dass wir das Skalarprodukt der Matrizen bilden. Wir nehmen also die ersten 3x3 Pixel unseres Bildes als Matrix und bilden das Skalarprodukt mit unserem Filter. Dann verschieben wir unsere Auswahl um eine Spalte und

wenden den Filter auf die nächsten 3x3 Pixel an. Das Skalarprodukt wird dann zum Ergebnis des neuen Pixels.

Jetzt stellen Sie sich wahrscheinlich zwei Fragen. Zum einen: Was bringt das Ganze? Und zum anderen: Woher kommen die Werte in unserer Matrix. Die Antwort auf beide Fragen ist dieselbe.

Zunächst sind diese Werte zufällig gewählt und irrelevant. Daher bringt auch anfangs das „Filtern“ nicht wirklich viel. Wir befinden uns hier jedoch in der Welt des Machine Learnings. Was wir also zunächst tun, ist unsere Bilder mit zufälligen Startwerten zu filtern. Dann schauen wir uns die Ergebnisse und die Genauigkeit unseres Modells an. Logischerweise werden diese sehr mangelhaft sein. Also passen wir mittels Backpropagation diese Werte immer weiter an, sodass sich unsere Ergebnisse stetig verbessern. Das funktioniert deswegen, weil sich unsere Filter, durch das ständige Trainieren und Anpassen an verschiedene Beispiele, zu Muster-Detektoren entwickeln. Wenn gewisse Muster immer wieder auftauchen, so werden die entsprechenden Werte unserer Matrix sehr hoch sein.

Nun müssen Sie sich vorstellen, dass solche Filter meistens keine 3x3 Matrizen sondern eher 64x64 Matrizen oder größer sind und ein Convolutional Layer aus mehreren solchen Filtern besteht. Außerdem haben wir oftmals mehrere Convolutional Layer hintereinander. Dementsprechend wird das Ganze ziemlich komplex, aber auch sehr effektiv.

POOLING LAYER

Pooling Layer sind jene Schichten, welche meist auf die Convolutional Layer folgen. Deren primäre Aufgabe ist es den Output der Convolutional Layer zu vereinfachen. Grob gesagt sorgen diese Layer dafür, dass sich auf das Wesentliche fokussiert wird, bevor die Daten weitergegeben werden.

Die populärste Art von Pooling ist das sogenannte *Max-Pooling*. Bei diesem wird aus jeder 2x2 Matrix des Outputs nur der höchste Wert für die weitere Verarbeitung übernommen.

Das Pooling verringert den Platzbedarf, erhöht die Berechnungsgeschwindigkeit, hilft gegen Overfitting und spart im

Allgemeinen Ressourcen, ohne dass die Performance darunter leiden muss.

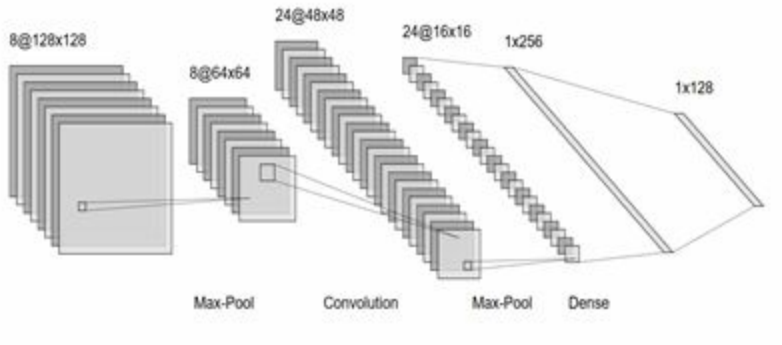


Abb. 5.3: Convolutional Neural Network

Hier sehen Sie wie ein Convolutional Neural Network aussehen könnte. Als erstes wird ein Bild in einen Convolutional Layer eingelesen. Das Ergebnis dieses wird dann, mittel Max-Pooling, vereinfacht und wiederum an einen weiteren Convolutional Layer weitergegeben. Dieser liefert den Output erneut an einen Pooling-Layer. Das Resultat von diesem wird dann durch zwei Dense-Layer geleitet, welche dann das finale Resultat (z.B. eine Klassifikation) liefern.

BILDDATEN LADEN UND VORBEREITEN

Kommen wir nun zum praktischen Teil dieses Kapitels. In diesem Kapitel werden wir einen Datensatz von Keras benutzen, welcher Bilder in zehn verschiedene Kategorien einteilt. Und zwar in folgende:

['**Flugzeug**', '**Auto**', '**Vogel**', '**Katze**', '**Reh**',
'**Hund**', '**Frosch**', '**Pferd**', '**Schiff**', '**LKW**']

Dieser Datensatz enthält zigtausende Bilder von verschiedenen Objekten, zusammen mit deren richtiger Klassifizierung. Unser Ziel ist es hierbei ein Convolutional Neural Network auf diese Daten zu trainieren, um dann im Endeffekt neue, unbekannte Bilder akkurat klassifizieren zu lassen.

Dafür müssen wir jedoch zunächst den Datensatz laden und vorbereiten. Für dieses Projekt benötigen wir folgende Bibliotheken:

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
```

```
from tensorflow.keras import datasets, layers, models
```

Beachten Sie hierbei, dass sie OpenCV noch installieren müssen, sollten Sie dies nicht bereits getan haben. Hierzu benutzen Sie einfach folgende Befehle:

```
pip install opencv-python
```

Wenn wir unsere Daten von Keras laden, bekommen wir, praktischer Weise, bereits zwei Tupel mit den Trainings- und Testdaten.

```
(trainings_bilder, trainings_label), (test_bilder, test_label)
= datasets.cifar10.load_data()
trainings_bilder, test_bilder =
trainings_bilder / 255.0, test_bilder / 255.0
```

Wir benutzen hier die Methode *load_data* des Datensatzes *cifar10* von den Keras-Datasets. Ebenso normalisieren wir unsere Daten direkt, indem wir alle Werte durch 255 dividieren. Da es sich hierbei um RGB-Werte handelt, und da die Werte zwischen 0 und 255 liegen, sind anschließend alle Werte zwischen 0 und 1.

Als nächstes definieren wir unsere Klassennamen in einer Liste, da unsere Ergebnisse alle numerisch sein werden, wir aber wissen möchten, um welche Art von Objekt es sich nun handelt.

```
klassen_namen = ['Flugzeug', 'Auto', 'Vogel', 'Katze',
'Reh', 'Hund', 'Frosch', 'Pferd', 'Schiff', 'LKW']
```

Nun können wir einen Teil dieser Daten visualisieren, um uns ein Bild über diesen Datensatz machen zu können.

```
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(trainings_bilder[i], cmap=plt.cm.binary)
    plt.xlabel(klassen_namen[trainings_label[i][0]])

plt.show()
```

Wir lassen hierfür eine For-Schleife 16 mal laufen, und erstellen Subplots in einem 4x4 Gitter. Die x-Ticks und die y-Ticks setzen wir auf eine leere Liste, damit uns keine Koordinaten stören. Dann benutzen wir die *imshow* Funktion

von Matplotlib, um das jeweilige Bild anzuzeigen. Als Label für dieses Bild benutzen wir dann den jeweiligen Klassennamen für den Index des Beispiels.

Abb. 5.4: Bilder aus dem Cifar10 Datensatz mit Labels

Dieser Datensatz enthält sehr viele Datensätze. Sollten Sie also nur über einen leistungsschwachen Computer oder wenig Zeit verfügen, macht es Sinn nur einen Teil der Daten zu verwenden.

```
trainings_bilder = trainings_bilder[:20000]
trainings_label = trainings_label[:20000]
test_bilder = test_bilder[:4000]
test_label = test_label[:4000]
```

Hier nehmen wir beispielsweise nur die ersten 20.000 Bilder als Trainingsdaten und nur die ersten 4.000 Testbilder als Testdaten. Natürlich wird unser Modell immer akkurater, je mehr Beispiele wir benutzen. Doch bei leistungsschwächeren Rechnern kann die Berechnung ewig dauern.

NEURONALES NETZ AUFBAUEN

Nun haben wir unsere Daten geladen und vorbereitet. Als nächsten Schritt bauen wir unser Neuronales Netz auf.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(32, 32, 3)))
```

```
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Wir definieren hier ein *Sequential Neural Network* wie gewohnt. Dann kommen unsere Inputs direkt in einen Convolutional Layer (Conv2D). Dieser hat 32 Filter bzw. Channel in der Form von 3x3 Matrizen. Die Aktivierungsfunktion ist ReLU, welche wir bereits kennen und die Input Shape ist 32x32x3. Das kommt daher, dass unsere Bilder die Maße 32x32 Pixel haben und drei Ebenen für die Farben aus RGB. Das Resultat kommt dann in einen *MaxPooling2D* Layer, welcher den Output vereinfacht. Dann führen wir diesen Output wieder in einen Convolutional Layer, daraufhin wieder in einen Max-Pooling Layer und dann erneut in einen Convolutional Layer. Das Resultat von diesem führen wir direkt in einen *Flatten Layer*, welcher dafür sorgt, dass unsere Daten von der Matrizenform in eine flache, eindimensionale Vektorform umgewandelt werden. Diese Daten kommen dann in einen Hidden-Dense-Layer mit 64 Neuronen und zu guter Letzt in den Output-Layer, welcher auch ein Dense-Layer ist mit 10 Neuronen und der Aktivierungsfunktion Softmax. Zur Erinnerung: Softmax ist jene Aktivierungsfunktion, welche die Resultate relativiert, sodass die Summe dieser 1 ist und wir somit die Prozente als Output bekommen.

TRAINIEREN UND TESTEN

Nun müssen wir unser Modell nur noch kompilieren, trainieren und testen bevor wir es benutzen können.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Wir benutzen hier wieder den Adam Optimizer und als Loss Function die *Sparse Categorical Crossentropy*.

```
model.fit(trainings_bilder,
          trainings_label,
```

```
        epochs=10,
        validation_data=(test_bilder, test_label))
```

Wir trainieren unser Modell nun mit den Trainingsdaten in zehn Epochen. Das bedeutet, dass unser Modell dieselben Daten zehn Mal wieder sehen wird.

```
test_loss, test_acc = model.evaluate(test_bilder,
                                     test_label,
                                     verbose=2)
```

Nun testen wir unser Modell nur noch mit der *evaluate* Funktion und unseren Testdaten. Dabei setzen wir den Parameter *verbose* auf 2, um möglichst viele Informationen zu erhalten.

```
- 1s - loss: 0.8139 - acc: 0.7090
```

Wir haben eine Genauigkeit von 70% was definitiv sehr eindrucksvoll ist, wenn man bedenkt, dass wir es mit zehn verschiedenen Klassen zu tun haben. Wenn wir raten würden, hätten wir eine 10% Chance richtig zu liegen. Unser Modell ist also etwa sieben Mal so akkurat.

EIGENE BILDER KLASSIFIZIEREN

Richtig interessant wird das alles jedoch erst dann, wenn wir neue eigene Bilder klassifizieren lassen, welche das Modell noch nie gesehen hat. Hierzu suchen Sie sich am besten auf Google oder irgendeiner anderen Seite Bilder der jeweiligen Klassen raus.

Abb. 5.5: Auto und Pferd

Ich werde dafür diese beiden Bilder benutzen, welche ich auf Pixabay gefunden habe (da diese lizenzfrei sind).

Wichtig ist hierbei, dass wir die Bilder zunächst auf das Format von 32x32

Pixel bringen. Dafür können wir diese entweder zuschneiden oder stauchen und skalieren. Ich verwende hierfür das Programm Gimp.

Abb. 5.6: Auto und Pferd skaliert auf 32x32 Pixel

Nun müssen wir das Ganze noch in unser Programm laden. Hierfür benutzen wir OpenCV.

```
img1 = cv.imread('pferd.jpg')
img1 = cv.cvtColor(img1, cv.COLOR_BGR2RGB)
img2 = cv.imread('auto.jpg')
img2 = cv.cvtColor(img2, cv.COLOR_BGR2RGB)
```

Die Funktion *imread* lädt das Bild in unser Skript. Wir benutzen dann noch die Funktion *cvtColor* um unser Farbschema von BRG (Blau, Rot, Grün) auf RGB (Rot, Grün, Blau) zu ändern.

```
plt.imshow(img1, cmap=plt.cm.binary)
plt.show()
```

Mit der *imshow* Funktion von Matplotlib können wir diese Bilder auch im Skript anzeigen lassen.

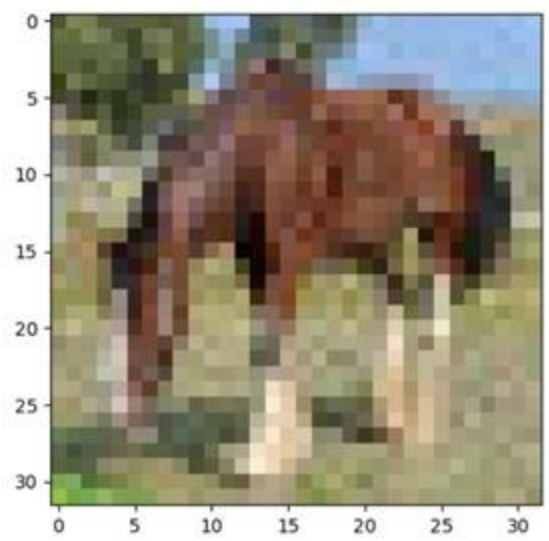


Abb. 5.7: Pferd in Matplotlib

Jetzt können wir diese Bilder als Input in unser Modell einlesen und uns die Klassifizierung ansehen.

```
vorhersage = model.predict(np.array([img1]) / 255)
index = np.argmax(vorhersage)
print(klassen_namen[index])
```

Wir benutzen als erstes die *predict* Funktion, um ein Ergebnis zu erhalten. Hierzu müssen wir jedoch unser Bild als NumPy-Array übergeben und durch 255 dividieren (auf Grund der Normalisierung). Das Resultat ist dann der Output unserer Softmax Aktivierungsfunktion. Aus dieser filtern wir mit *np.argmax* den Index des höchsten Werts heraus. Diesen benutzen wir dann, um das Resultat aus der Liste unserer Klassennamen zu holen.

Pferd Auto

Das Resultat spricht für sich. Diese beiden Bilder wurden absolut korrekt klassifiziert. Natürlich wird das nicht immer der Fall sein. Es kann durchaus sein, dass ein Reh, welches in einem atypischen Winkel fotografiert wurde, als Pferd klassifiziert wird. Aber man kann schon sagen, dass die Leistung unseres Modells eindrucksvoll ist.

6 – ABSCHLUSS UND RESSOURCEN

Wir haben in den letzten fünf Kapiteln einiges gelernt und vieles davon war sehr komplex. In diesem finalen Kapitel möchte ich Sie daher noch einmal im Schnelldurchlauf durch die Kapitel führen und Ihnen ein paar ergänzende Informationen geben.

REVIEW: GRUNDLAGEN

Das erste Kapitel war vermutlich das anspruchsvollste, da wir sehr tief in die Theorie gegangen sind und das natürlich viel Mathematik bedeutet hat. Hierbei liegt es an Ihnen zu entscheiden, wie tief Sie sich mit der Materie auseinandersetzen möchten.

Wenn Sie wirklich ein Machine-Learning Experte werden möchten und eigene innovative Entwicklungen in diesem Bereich erzielen möchten, so macht es sehr viel Sinn, diese Mathematik bis ins letzte Detail zu verstehen und zu meistern.

Sollten Sie jedoch kein tieferes Interesse an Machine Learning an sich haben, so reicht es definitiv auf der Use-Case Ebene zu bleiben. Wenn es also beispielsweise Ihr Ziel ist eine App im Fitnessbereich zu programmieren, welche sich Machine Learning zu Nutze macht, so reicht es wenn Sie wissen was Neuronale Netze sind, welche Arten es gibt, wie diese grundlegend funktionieren und wie man sie benutzt. Mehr benötigen Sie nicht.

Sobald es in den Bereich der Forschung, Entwicklung und Innovation des Machine Learnings geht, so kommen Sie nicht um komplexe Mathematik herum.

Fragen Sie sich an dieser Stelle, ob Sie folgende Konzepte wirklich gut verstanden haben:

- Neuronale Netze und deren Aufbau

- Aufbau von Perzeptronen
- Aktivierungsfunktionen
- Trainieren und Testen von Modellen
- Error und Loss
- Gradient Descent
- Backpropagation

Sollten Sie merken, dass Sie hier noch Wissens- oder Verständnislücken haben, blättern Sie noch einmal durch das erste Kapitel und festigen Sie das Wissen. Ebenso macht es Sinn bei Unklarheiten im Internet zu recherchieren. Beim Programmieren kommt es immer wieder zu Fehlern und Unklarheiten, welche man nicht alle in einem Buch abdecken kann. Scheuen Sie sich daher nicht Google, StackOverflow und Dokumentationen zu benutzen.

REVIEW: NEURONALE NETZE

Im zweiten Kapitel ging es um die Grundlagen von Tensorflow und wir haben ein erstes simples Neuronales Netz entworfen, welches handgeschriebene Zahlen klassifiziert. Wenn Sie hier etwas nicht verstanden haben, hatten Sie höchstwahrscheinlich auch Probleme mit den nachfolgenden Kapiteln, da diese aufeinander aufbauen. Stellen Sie also sicher, dass Sie die Grundlagen dieses Kapitels gemeistert haben.

Fragen Sie sich an dieser Stelle, ob Sie folgende Konzepte wirklich gut verstanden haben:

- Datensätze von Keras laden
- Trainings- und Testdaten aufteilen
- Modelle aufbauen mit Tensorflow
- Kompilieren von Modellen
- Trainieren und Testen von Modellen
- Eigene Bilder einlesen und vorbereiten

Überlegen Sie sich hierbei ebenso, was Sie noch mit diesem Wissen anfangen könnten. Wir haben ein einziges Beispiel gemacht. Informieren Sie sich über weitere Datensätze und Ideen, welche Sie mit simplen Feed Forward Neural Networks umsetzen können.

Keras Datensätze: https://keras.io/datasets/

Scikit-Learn Datensätze:
https://scikit-learn.org/stable/datasets/index.html

REVIEW: RECURRENT NEURAL NETWORKS

Spannend wurde es dann im dritten Kapitel, als endlich Rekurrente Neuronale Netze zum Einsatz kamen. Wir haben hier Texte von Goethe und Shakespeare benutzt, um unserem Modell das Generieren von Texten beizubringen.

Ein großer Teil der Arbeit hier war es, die Textdaten umzuwandeln und vor allem den Output des Netzwerks sinnvoll zu verwenden, um Texte zu generieren.

Die Herausforderung lag hier nicht wirklich auf dem Typen von Rekurrenten Netzen. Diese haben wir mittels LSTM-Layern schnell erstellt. Der Hauptfokus dieses Kapitels war das ganze Drum-Herum. Oftmals ist das Modell selbst, im Machine Learning gar nicht das Problem. Wenn wir vorgefertigte Datensätze von Keras benutzen, ist alles natürlich sehr leicht. Doch wenn Sie Daten aus der „echten Welt“ benutzen, werde diese weder strukturiert, noch komplett, noch in irgendeiner anderen Weise für Sie vorteilhaft aufbereitet sein. Daher ist es wichtig, dass Sie auch diesen Aspekt des Machine Learnings meistern – das Vorbereiten von Daten.

Fragen Sie sich an dieser Stelle, ob Sie folgende Konzepte wirklich gut verstanden haben:

- Daten aus dem Internet in Python laden
- Daten umformatieren und so aufbereiten, dass Sie sinnvoll Neuronalen Netzen übergeben werden können
- Grundlegende Vorteile von LSTM-Layern
- Funktionsweise von Rekurrenten Neuronalen Netzen

Auch hier können Sie wieder mit anderen Ideen experimentieren. Rekurrente Neuronale Netze sind sehr performant wenn es um sequentielle und zeitabhängige Daten geht. Das können Aktienkurse, Wetterdaten, Texte,

Musik und vieles mehr sein. Sein Sie kreativ und recherchieren Sie ein wenig. Starten Sie dann ihr eigenes Mini-Projekt mithilfe von Rekurrenten Neuronalen Netzen.

REVIEW: CONVOLUTIONAL NEURAL NETWORKS

Zu guter Letzt haben wir uns im letzten Kapitel mit Convolutional Neural Networks befasst. Diese sind besonders gut wenn es um Mustererkennung geht und daher sehr nützlich bei der Verarbeitung von Bild- und Audiodaten.

Wir haben ein Keras-Dataset verwendet, welches und zigtausende klassifizierte Bilder bietet. Mithilfe dieses Datensatzes haben wir unser Modell darauf trainiert, zehn verschiedene Objekte (wie Pferde, Flugzeuge, Autos etc.) zu erkennen und mit etwa 70% Genauigkeit zu klassifizieren.

Fragen Sie sich an dieser Stelle, ob Sie folgende Konzepte wirklich gut verstanden haben:

- Convolutional Layer
- Pooling-Layer und Pooling–Funktionen
- Filter bzw. Channels und Matrizen

Auch hier gilt wieder: Experimentieren Sie selbst herum. Gestalten Sie vielleicht Ihr eigenes Dataset und schauen Sie wie gut Ihr Neuronales Netz abschneidet. Wenden Sie Ihr Wissen auf andere vorgefertigte Datensätze an. Recherchieren Sie vielleicht, wie man Audiodaten in Python einliest und verarbeitet und erstellen Sie dann ein Convolutional Neural Network, welches Stimmen oder gar Wörter erkennt.

NEURALNINE

Zu guter Letzt möchte ich Sie noch auf eine Ressource hinweisen, welche sehr viel mit diesem Buch zu tun hat. Neben meinen deutschsprachigen Büchern, produziere ich auch sehr viel englischsprachigen Content. Sollte die Sprache also für Sie kein Problem sein, so können Sie sich gerne die Inhalte von NeuralNine ansehen.

YouTube: https://bit.ly/3a5KD2i

Webseite: https://www.neuralnine.com/

Instagram: https://www.instagram.com/neuralnine/

Auf meinem YouTube Kanal finden Sie kostenlose Tutorials über das Thema Programmieren und Machine Learning. Auf meiner Webseite erwarten Sie viele Blog Posts, welche auch etwas tiefer in die Mathematik mancher Algorithmen gehen. Und auf meiner Instagram Seite veröffentliche ich täglich Infographics zum Thema Programmieren und Informatik.

PYTHON
COMPUTER VISION

FLORIAN DEDOV

PYTHON FÜR COMPUTER VISION

DER SCHNELLE EINSTIEG

VON

FLORIAN DEDOV

INHALTSVERZEICHNIS

EINLEITUNG

Die Computer Vision ist einer der spannendsten Bereiche der Informatik. Dieses Gebiet befasst sich damit, wie Computer Bild- und Videodaten wahrnehmen und verarbeiten. Die Technologien in diesem Bereich sind maßgeblich für unsere Zukunft. Mittels Computer Vision können wir unlesbare, unscharfe und schlecht beleuchtete Texte lesbar machen. Ebenso können wir in Echtzeit Objekte und Gesichter erkennen lassen. Wir können Filter, Transformationen und zahlreiche Effekte anwenden.

In der Programmiersprache Python können wir, mithilfe der Bibliothek OpenCV (welche auch für andere Programmiersprachen verfügbar ist), bereits mit wenigen Zeilen Code, eindrucksvolle Resultate erzielen.

Diese Skills sind für viele Bereiche essentiell. Überwachungs- und Sicherheitssysteme sind auf die Computer Vision angewiesen. Die gesamte Robotik basiert zu einem großen Teil auf diesem Gebiet. Doch auch in der Medizin, der Bildverarbeitung, der Filmtechnik, der Industrie und der Automatisierung, ist die Computer Vision von hohem Nutzen.

DIESES BUCH

In diesem Buch lernen Sie, wie Sie mit OpenCV und Python interessante Projekte im Bereich der Computer Vision verstehen und umsetzen können. Sie lernen in jedem Kapitel ein wenig Theorie, gefolgt von interessanten praxisorientierten Beispielen.

Was in diesem Buch jedoch vorausgesetzt wird, sind fortgeschrittene Python-Kenntnisse und ein grundlegendes Verständnis von Data Science. Sollten Ihnen Bibliotheken wie NumPy, Matplotlib und Pandas kein Begriff sein, so empfehle ich Ihnen zunächst mein Python Buch für Data Science zu lesen oder sich das Wissen anderweitig anzueignen. Wir werden einige dieser Bibliotheken zwar verwenden, doch ihre grundlegende Funktionsweise wird

hier nicht im Detail erläutert.

Falls Ihnen einige Skills im Python-Bereich fehlen, können Sie sich gerne auf meiner Amazon Autorenseite, meine einleitenden Werke ansehen. Diese beginnen bei Büchern für Anfänger und Fortgeschrittene und ziehen sich dann weiter durch die Gebiete der Data Science, des Machine Learnings und der Finanzanalyse mit Python. Natürlich können Sie sich das notwendige Wissen auch anders aneignen. In diesem Buch wird es jedoch keine Erklärungen zur grundlegenden Python-Syntax oder zu den oben genannten Bibliotheken geben.

Meine Autorenseite: https://amzn.to/395BNB5

LESETIPPS FÜR DIESES BUCH

Im Grunde genommen steht es Ihnen frei, sich selbst auszusuchen, wie Sie dieses Buch lesen möchten. Wenn Sie der Meinung sind, dass die einleitenden Kapitel für Sie uninteressant sind oder, dass Sie bereits alles wissen, können Sie diese auch überspringen. Ebenfalls können Sie das Buch von vorne bis hinten durchlesen, ohne jemals eine Zeile Code zu schreiben. Doch das ist alles nicht empfehlenswert.

Ich persönlich empfehle Ihnen jedes Kapitel in der richtigen Reihenfolge zu lesen, da diese aufeinander aufbauen. Der Code funktioniert zwar auch ohne die vorangehenden Kapitel, doch dann mangelt es Ihnen an Verständnis und Sie haben keine Ahnung, was Sie implementieren und warum es funktioniert oder nicht.

Außerdem ist es unglaublich wichtig, dass sie nebenher aktiv mitprogrammieren. Nur so verstehen Sie wirklich die Inhalte in diesem Buch. Es wird in den Kapiteln viel Code geben. Lesen Sie ihn sich durch, verstehen Sie ihn, aber implementieren Sie ihn auch auf Ihrer eigenen Maschine. Experimentieren Sie herum. Was passiert wenn Sie einzelne Parameter ändern? Was passiert, wenn sie noch etwas dazugeben? Probieren Sie alles aus.

Mehr gibt es glaube ich hier nicht zu sagen. Ich wünsche Ihnen viel Erfolg und Spaß beim Lernen über Computer Vision mit Python. Ich hoffe, dass

dieses Buch Ihnen bei Ihrer weiteren Laufbahn behilflich sein kann!

Eine kleine Sache noch! Falls Sie nach dem Lesen dieses Buches, der Meinung sind, dass es positiv zu Ihrer Programmierkarriere beigetragen hat, würde es mich sehr freuen, wenn Sie eine Rezension auf Amazon hinterlassen. Danke!

BIBLIOTHEKEN INSTALLIEREN

Für dieses Buch werden Sie einige Bibliotheken benötigen, welche nicht im Standard-Stack von Core-Python enthalten sind. Das bedeutet, dass Sie diese separat installieren müssen. Für die Installation verwenden wir *pip*.

Zunächst installieren wir die üblichen Data Science Bibliotheken:

pip install numpy

pip install matplotlib

pip install pandas

Diese sollten Ihnen mehr oder weniger bekannt sein. Sie sind nicht der Hauptfokus in diesem Buch, doch sie werden uns an der einen oder anderen Stelle behilflich sein. Die Haupt-Bibliothek ist OpenCV. Diese müssen wir ebenfalls installieren:

pip install opencv-python

All diese Bibliotheken werden uns einen Großteil der Arbeit abnehmen, welche wir ohne diese manuell erledigen müssten.

1 – BILDER UND VIDEOS LADEN

Bevor wir mit der Verarbeitung von Bildern und Videos beginnen, schauen wir uns in diesem Kapitel zunächst an, wie wir Bild- und Videodaten in unser Skript laden.

Für dieses Kapitel, werden Sie folgende Imports benötigen:

```
import cv2 as cv
import matplotlib.pyplot as plt
```

Hier merken Sie, dass OpenCV bei der Installation zwar *opencv-python* hieß, jedoch mit *cv2* importiert wird. Wir benutzen hier jedoch einen Alias, nämlich *cv*, damit unser Code auch für zukünftige Versionen leicht zu ändern ist. Ebenso importieren wir Matplotlib, da wir auch mit dieser Bibliothek ein wenig mit Bildern arbeiten können.

BILDER LADEN

Um ein Bild zu laden, müssen wir uns natürlich zunächst ein Bild vorbereiten. Hierbei können Sie ein Bild aus dem Internet herunterladen oder aber Ihre eigenen Bilder benutzen. Ich benutze für dieses Buch lizenzfreie Bilder von Pixabay oder solche, welche ich selber produziert habe.

```
img = cv.imread('auto.jpg', cv.IMREAD_COLOR)
```

Um nun ein Bild in unser Skript einzulesen, benutzen wir die *imread* Funktion von OpenCV. Dieser übergeben wir zunächst den Pfad unseres Bildes und dann zusätzlich mit welchen Farben wir das Bild einlesen möchten.

In diesem Fall haben wir uns für *IMREAD_COLOR* entschieden, weil wir das Bild mit Farbe einlesen möchten. Dieses Bild können wir nun auch anzeigen.

```
cv.imshow('Auto', img)
cv.waitKey(0)
```

```
cv.destroyAllWindows()
```

Dafür benutzen wir die *imshow* Funktion, welcher wir zum einen den Titel bzw. den Identifier des Bildes mitgeben und zum anderen das Bildobjekt selbst. Danach folgen zwei Befehle, welche wir noch sehr oft benutzen werden. Der erste ist die *waitKey* Funktion, welche darauf wartet, dass eine Taste gedrückt wird, bevor das Skript weiterläuft. Der Parameter ist hierbei das Delay, also die Verzögerung. In diesem Fall setzen wir diese auf den Wert 0. Danach folgt die Funktion *destroyAllWindows*, welche am Ende alle Fenster schließt und beendet.

Abb. 1.1: Bild eines Autos in OpenCV

Wenn wir anstatt von *IMREAD_COLOR* beispielsweise *IMREAD_GRAYSCALE* gewählt hätten, so würde das Bild folgendermaßen aussehen.

Abb. 1.2: Bild in Grayscale

BILDER ANZEIGEN MIT MATPLOTLIB

Eine weitere Möglichkeit Bilder in Python anzuzeigen ist mit Matplotlib. Auch diese Bibliothek hat eine *imshow* Methode, welche uns das Bild darstellt.

```
plt.imshow(img)
plt.show()
```

Das Problem ist hierbei jedoch, dass Matplotlib ein anderes Farbschema benutzt als OpenCV. Das endet dann in folgendem Resultat.

Abb. 1.3: Bild eines Autos in Matplotlib

FARBSCHEMATA KONVERTIEREN

Während OpenCV mit dem RGB-Farbschema (Rot, Grün, Blau) arbeitet, arbeitet Matplotlib mit dem BGR-Farbschema (Blau, Grün, Rot). Das bedeutet im Endeffekt, dass die Werte für Blau und Rot in unserem Bild vertauscht sind. Um das zu ändern, können wir das Farbschema jedoch ändern bzw. konvertieren.

```
img = cv.cvtColor(img, cv.COLOR_RGB2BGR)
```

Mit der Funktion *cvtColor* können wir unser Bild konvertieren. Hierzu übergeben wir zunächst das Bildobjekt und dann noch die gewünschte Konvertierung. In diesem Fall wählen wir *COLOR_RGB2BRG*, da wir unser Bild von RGB in BGR umwandeln. Danach haben wir auch in Matplotlib die ursprünglichen Farben.

VIDEOS LADEN

Neben Bildern können wir auch ganze Videos mit OpenCV in unser Skript laden.

```
import cv2 as cv

video = cv.VideoCapture('stadt.mp4')

while True:
    ret, frame = video.read()

    cv.imshow('Video Datei', frame)

    if cv.waitKey(30) == ord('x'):
        break

video.release()
cv.destroyAllWindows()
```

Hierzu benutzen wir das Objekt *VideoCapture* und übergeben diesem die Videodatei. Dann starten wir eine Endlosschleife, welche mithilfe der Funktion *read* immer ein Frame weiter „liest“. Dieses Frame zeigen wir dann direkt mit der *imshow* Methode an. Am Ende der Schleife haben wir dann noch ein *waitKey*, welches überprüft ob die Taste ‘x‘ gedrückt wurde

(beliebig austauschbar). Als Parameter übergeben wir hier 30, was bedeutet, dass wir 30 Millisekunden warten, bevor wir das nächste Frame anzeigen. Eine Sekunde hat 1000 Millisekunden. Wenn wir also alle 30 Millisekunden ein Frame zeigen, haben wir eine FPS-Rate (Frames pro Sekunde) von 33. Das ist natürlich beliebig anpassbar. Zu guter Letzt benutzen wir die *release* Funktion, um die Videoquelle freizugeben.

Abb. 1.4: Screenshot vom Abspielen des Videos

Wenn das Video jedoch ausläuft, ohne dass wir es manuell terminiert haben, so stürzt unser Skript ab und wir erhalten eine Exception. Das können wir ganz einfach beheben, indem wir ein kleines If-Statement einbauen, welches überprüft ob ein Return-Wert vorhanden ist.

```
while True:
    ret, frame = video.read()

    if ret:
        cv.imshow('Video Datei', frame)

        if cv.waitKey(30) == ord('x'):
            break
    else:
        break
```

Alternativ könnten wir unser Video jedoch auch endlos laufen lassen, indem wir statt dem *break* die Videoquelle neu setzen.

```
while True:
    ret, frame = video.read()
```

```
    if ret:
        cv.imshow('Video Datei', frame)

        if cv.waitKey(30) == ord('x'):
            break
    else:
        video = cv.VideoCapture('stadt.mp4')
```

Jedes Mal wenn das Video in diesem Beispiel ausläuft, starten wir vom Anfang an.

KAMERA LADEN

Neben Bildern und Videos können wir auch Kameradaten in OpenCV laden. Wenn wir als VideoCapture keine Datei, sondern eine Nummer (Indexierung der Kamera) angeben, so können wir die Bilder von unserer Kamera in Echtzeit ansehen.

```
import cv2 as cv

video = cv.VideoCapture(0)

while True:
    ret, frame = video.read()

    if ret:
        cv.imshow('Video Datei', frame)

        if cv.waitKey(1) == ord('x'):
            break
    else:
        video = cv.VideoCapture('stadt.mp4')

video.release()
cv.destroyAllWindows()
```

Hier geben wir 0 als Videoquelle an, was unsere erste angeschlossene Kamera ist. Sollten Sie zwei, drei oder mehr Kameras angeschlossen haben, so können Sie diesen Index verändern, um diese auszuwählen.

Hier können wir ebenfalls den Delay der *waitKey* Funktion anpassen. Sollten Sie zum Beispiel Überwachungskameras installieren, wären eine geringere FPS-Zahl, und damit ein höherer Delay, angebracht, da die Kameras

durchgehend laufen und Sie so viel Speicherplatz sparen würden. Falls Sie jedoch einfach Effekte und Filter ausprobieren möchten, so ist ein Delay von einer Millisekunde optimal.

2 – GRUNDLEGENDE VERARBEITUNG

Nun da wir wissen, wie wir sowohl Bilder als auch Videos und Kameradaten in unser Skript laden können, befassen wir uns mal etwas näher mit der grundlegenden Verarbeitung dieser Daten.

ZEICHNEN

Zunächst sehen wir uns hierbei an, wie wir auf Bildern zeichnen können. Mit OpenCV können wir einfache Formen, wie Linien, Rechtecke und Kreise, auf unsere Bilder zeichnen.

```
cv.line(img, (50,50), (250,250), (255,255,0), 15)
cv.rectangle(img, (350,450), (500,350), (0,255,0), 5)
cv.circle(img, (500, 200), 100, (255,0,0), 7)
```

Hierfür benutzen wir die Funktionen *line*, *rectangle* und *circle*. Es gibt jedoch natürlich auch weitere wie *fillPoly*, um beispielsweise ein Polygon auszufüllen.

Die Parameter variieren natürlich bei den einzelnen Funktionen. Bei *line* und *rectangle* beispielsweise, geben wir nach dem Bild immer zunächst zwei Punkte, in Form von Tupeln, an. Das sind der Startpunkt und der Endpunkt, angegeben als x- und y-Koordinaten. Dann übergeben wir noch ein Tupel mit den drei RGB-Werten für die Farbe. Zu guter Letzt haben wir dann noch die Dicke der Linien.

Beim Kreis wiederum haben wir nur einen Punkt gegeben und das ist der Mittelpunkt des Kreises. Darauf folgt der Radius und dann kommen wieder die Farbe und die Liniendicke.

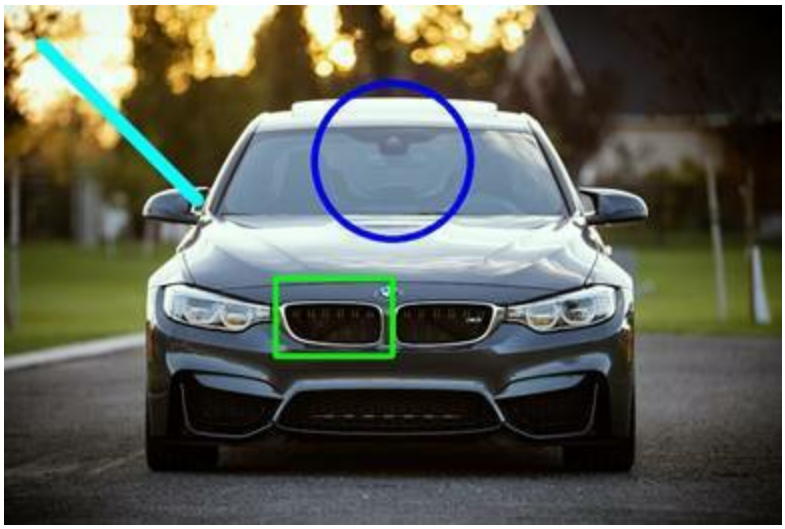


Abb. 2.1: Gezeichnete Formen mittels OpenCV

In der oberen Abbildung sehen Sie, wie so etwas aussehen kann. Primär ist dieses primitive Zeichnen sinnvoll, wenn wir etwas Bestimmtes hervorheben wollen. Beispielsweise wenn wir nach einer Objekterkennung ein Objekt einrahmen wollen, um zu signalisieren, dass es erkannt wurde.

ZEICHNEN MIT MATPLOTLIB

Es ist zudem auch möglich mit Matplotlib auf unsere Bilder zu zeichnen. Wie man mit Matplotlib genau zeichnet werde ich hier nicht näher erläutern, da ich das schon in meinem Data Science Buch ausführlich erklärt habe. Wenn Sie jedoch mathematische Graphen auf das Bild zeichnen möchten, so können Sie das mit Matplotlib gut machen.

```
x_values = np.linspace(100,900,50)
y_values = np.sin(x_values) * 100 + 300

plt.imshow(img, cmap='gray')
plt.plot(x_values,y_values, 'c', linewidth=5)
plt.show()
```

Hier zeichnen wir beispielsweise eine modifizierte Sinus-Funktion mitten über unser Bild. Das mag in diesem Fall wenig Sinn machen, kann aber durchaus sinnvolle Anwendungsfälle haben.

Abb. 2.2: Mathematische Funktion über Bild

ELEMENTE KOPIEREN

Was wir mit OpenCV ebenfalls machen können, das sich als sehr nützlich erweist, ist Abschnitte zu kopieren bzw. auszuschneiden. Das tun wir mittels Index-Slicing.

```
img[0:200, 0:300] = [0, 0, 0]
```

In diesem Beispiel ersetzen wir alle Pixel von 0 bis 200 auf der Y-Achse und von 0 bis 300 auf der X-Achse mit schwarzen Pixeln. Wir weisen diesem Bereich also eine Liste mit drei Nullen als RGB-Codes zu.

Abb. 2.3: Ausschnitt des Bildes ersetzen

Durch das Index-Slicing, wie wir es gerade benutzt haben, können wir auch bestimmte Bereiche aus dem Bild auswählen und verschieben bzw. kopieren.

```
kopieren = img[300:500, 300:700]
```

```
img[100:300, 100:500] = kopieren
```

Mit diesem Code speichern wir alle Pixel im Bereich von 300 bis 500 auf der Y-Achse und vom Bereich 300 bis 700 auf der X-Achse in eine Zwischenvariable. Den Inhalt können wir dann einem anderen Bereich im Bild zuweisen, welcher dieselben Maße hat.

```
img[300:500, 300:700] = [0,0,0]
```

Wenn wir den Bereich nicht kopieren, sondern verschieben möchten, können wir die ursprünglichen Pixel ersetzen. Bei Überschneidungen sollte man hier jedoch vorsichtig sein, da durch diese Zeile alles geschwärzt wird.

Abb. 2.4: Verschieben und Kopieren von Bereichen

Den Unterschied sehen Sie hier eindeutig. Bei dem einen Bild wird ein Bereich einfach nur kopiert und eingefügt, während bei dem anderen der Bereich verschoben bzw. ausgeschnitten wird.

BILD UND VIDEOS SPEICHERN

Wenn wir mit der Verarbeitung von unseren Bildern und Videos fertig sind, so können wir diese auch abspeichern. Bei Bildern ist das Ganze sehr einfach.

```
cv.imwrite('auto_neu.jpg', img)
```

Wir benutzen hierbei einfach nur die *imwrite* Funktion von OpenCV. Je nach Dateiendung, welche wir angeben, wird unser Bild entsprechend kodiert.

VIDEOS SPEICHERN

Bei Videos ist das alles auch nicht sonderlich komplex. Hier müssen wir

jedoch noch ein paar zusätzliche Dinge definieren.

```
capture = cv.VideoCapture(0)
fourcc = cv.VideoWriter_fourcc(*'XVID')
writer = cv.VideoWriter('video.avi', fourcc, 60.0, (640,480))
```

Neben der normalen *VideoCapture* müssen wir hier auch noch den sogenannten FourCC angeben. Das ist sozusagen der Codec, welchen wir benutzen, um das Video letztendlich zu kodieren. Dieser gibt das Format an. In diesem Fall wählen wir hier XVID. Das ist eine Open-Source Variante des MPEG-4 Codecs.

Zusätzlich dazu müssen wir unseren *VideoWriter* definieren. Diesem übergeben wir zum einen den Dateinamen und den Codec, aber auch die FPS (Frames pro Sekunde) und die gewünschte Auflösung. In diesem Fall speichern wir in die Datei *video.avi* die Kameradaten mit 60 FPS (30 reichen auch) und einer Auflösung von 640x480 Pixeln.

```
while True:
    ret, frame = capture.read()

    writer.write(frame)

    cv.imshow('Cam', frame)

    if cv.waitKey(1) == ord('x'):
        break
```

In unserer Endlosschleife benutzen wir dann in jeder Iteration die Funktion *write*, um unsere Frames in die Datei zu schreiben.

```
capture.release()
writer.release()
cv.destroyAllWindows()
```

Am Ende ist es noch wichtig, dass wir unsere Capture, als auch unseren Writer releasen.

3 – THRESHOLDING

Kommen wir nun zu einem der spannendsten Bereiche der Computer Vision, dem Thresholding. Zu Deutsch nennt sich das Ganze *Schwellenwertverfahren*. Hierbei geht es darum, Bilder zu segmentieren. Das ist zum einen wichtig für Technologien wie Objekt- und Gesichtserkennung, aber auch für das Herausfiltern von Informationen und für das Optimieren von schlecht belichteten Bildern.

LOGO EINFÜGEN

Im folgenden Abschnitt werden wir Thresholding benutzen, um ein Logo teilweise transparent zu machen und es dann in ein Bild einzufügen.

Abb. 3.1: Bild von einem Arbeitsplatz

Dieses lizenzfreie Bild soll unser Hauptbild sein und oben links möchten wir nun ein Logo einfügen.

Abb. 3.2: Musterlogo

Dieses Bild werden wir hierzu als Musterlogo verwenden. Es ist ein weißes M auf blauem Hintergrund. Was wir nun möchten ist, dass das M transparent wird, sodass man das Hauptbild durchsehen kann.

Hier kommt Thresholding ins Spiel. Wir werden mit OpenCV den weißen Bereich erkennen und diesen dann transparent machen.

```
img1 = cv.imread('laptop.jpg')
img2 = cv.imread('logo.png')
```

Hierfür laden wir zunächst die beiden Bilder in unser Skript, wie wir es bereits gelernt haben. Im nächsten Schritt kommen wir zum Thresholding.

```
logo_grau = cv.cvtColor(img2, cv.COLOR_RGB2GRAY)
ret, maske = cv.threshold(logo_grau, 180, 255, cv.THRESH_BINARY_INV)
```

Nun konvertieren wir unser Logo in Graustufen, da es uns nur um die weiße Farbe geht. Dafür benutzen wir den Modus *COLOR_RGB2GRAY*. Dann kommt die *threshold* Funktion. Dieser übergeben wir zunächst das Graustufen-Logo und geben dann an, ab welchem Wert, zu welchem Wert geändert werden soll. In diesem Fall ändern wir jeden Wert, welcher höher als 180 (also hellgrau) ist, zu 255 (ganz weiß). Hierzu benutzen wir das Verfahren *THRESH_BINARY_INV*.

Als Rückgabewerte erhalten wir hier unter anderem die Maske, welche wir anzeigen lassen können.

```
cv.imshow('Maske',maske)
```

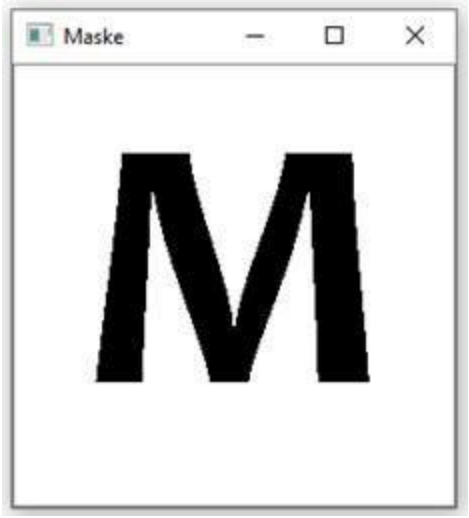


Abb. 3.3: Maske nach Thresholding

Um den nächsten Schritt zu verstehen, muss uns zunächst klar werden, was wir hier erzielen möchten. Wir möchten den blauen Hintergrund so erhalten, wie er ist und den weißen Text komplett entfernen. In OpenCV werden wir uns hierzu dem bitweisen UND bedienen. Das ist eine logische Operation, welche ein True liefert, wenn beide Operanden True sind. In unserem Kontext ist Schwarz gleich False und Weiß gleich True. Wenn wir also mit der weißen Farbe ein logisches UND durchführen, erhalten wir als Resultat das was der andere Operand liefert. Sollten wir also weiße Farbe mit unserem Hintergrund aus Bild eins logisch „verunden“ so erhalten wir genau diesen Hintergrund als Resultat.

Bei der schwarzen Farbe ist es genau anders herum. Dadurch, dass diese False darstellt, wird das Ergebnis zu null Prozent übernommen und es ändert sich nichts.

Wenn Sie das Prinzip verstanden haben, wird Ihnen wahrscheinlich auffallen, dass unser M die falsche Farbe hat. Wenn es transparent sein soll, so muss es weiß sein. Daher müssen wir unsere Maske invertieren.

```
maske_inv = cv.bitwise_not(maske)
maske_inv = np.invert(maske) # Alternativ
```

Hierzu können wir entweder die Funktion *bitwise_not* von OpenCV für das bitweise Negieren verwenden oder aber die *invert* Funktion von NumPy. Nur nicht beides, da Sie sonst doppelt invertieren.

Nun kommen wir zum eigentlichen Einfügen des Logos. Hierzu müssen wir zunächst bestimmen, wie groß dieses ist und den entsprechenden Bereich auswählen für das UND.

```
zeilen, spalten, channels = img2.shape
bereich = img1[0:zeilen, 0:spalten]
```

Wir benutzen das *shape* Attribut unseres Logos, um die Maße und die Channels zu erhalten. Dann speichern wir den entsprechenden Beriech unseres Hintergrundbildes in eine Variable.

```
img1_bg = cv.bitwise_and(bereich, bereich, mask=maske_inv)
img2_fg = cv.bitwise_and(img2, img2, mask=maske)
```

Nun wenden wir die beiden bitweisen Operationen an. Wir definieren hier zwei Teile des oberen linken Ecks, welche wir anschließend kombinieren werden. Zum einen definieren wir den Hintergrund von dem ersten Bild, indem wir auf den ausgewählten Bereich die inverse Maske anwenden. Hier sorgen wir dafür, dass das M transparent wird. Durch die zweite Zeile fügen wir dann mit der ersten Maske den blauen Hintergrund hinzu.

```
resultat = cv.add(img1_bg, img2_fg)
img1[0:zeilen, 0:spalten] = resultat
```

Zu guter Letzt benutzen wir die *add* Funktion von OpenCV, um die beiden Schichten zu kombinieren. Dann weisen übernehmen wir das Resultat in die obere linke Ecke des Bildes.

```
cv.imshow('Resultat', img1)
```

Abb. 3.4: Resultat des Thresholdings

Das Resultat ist genau wie wir es uns gewünscht haben.

SCHLECHTE BILDER LESBAR MACHEN

Kommen wir nun zu einem Beispiel, welches um einiges eindrucksvoller ist.

Abb. 3.5: Schlecht lesbare Buchseite

Können Sie klar lesen, was auf dieser Buchseite steht? Es ist nicht so, dass es unmöglich ist, doch es ist sehr anstrengend. Die Lichtverhältnisse sind sehr schlecht. Mittels Thresholding können wir diesen Text jedoch viel lesbarer machen.

Eine erste Idee wäre es das Bild in Graustufen umzuwandeln und das binäre Thresholding anzuwenden.

```
img = cv.imread('buchseite.jpg')
img_grau = cv.cvtColor(img, cv.COLOR_RGB2GRAY)
ret, threshold = cv.threshold(img_grau, 32, 255, cv.THRESH_BINARY)
```

Jedes Pixel welches weißer als der Wert 32 (dunkelgrau) ist, wird zu 255 (komplett weiß) umgewandelt und jeder Wert darunter zu 0 (komplett

schwarz).

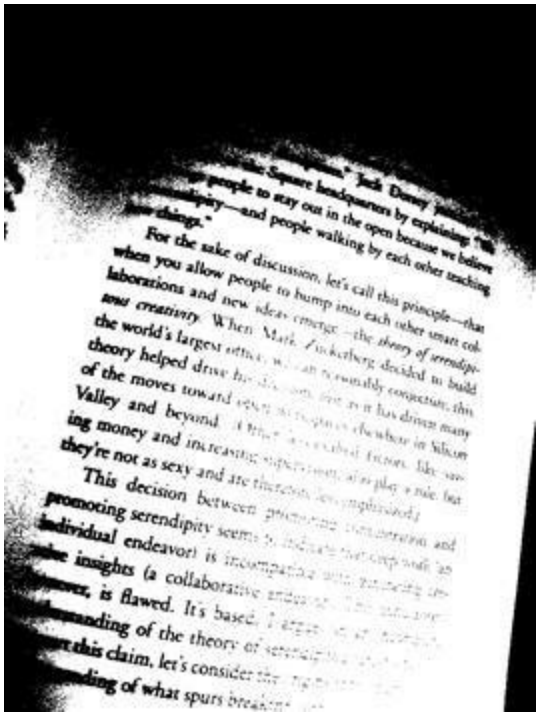


Abb. 3.6: Resultat mittels binärem Thresholding

Wie Sie sehen ist das Resultat nicht wirklich gut. Teilweise sind Stellen, welche schwarz sein sollten so hell, dass sie ausgebleicht werden und teilweise werden Stellen, welche erhellt werden sollten geschwärzt. Wir benötigen eine dynamischere Variante des Thresholdings.

Hier kommt das adaptive Gauß'sche Thresholding ins Spiel. Dieses Verfahren hilft uns unser binäres Thresholding bedingter anzuwenden.

```
gaus = cv.adaptiveThreshold(img_grau, 255,
                            cv.ADAPTIVE_THRESH_GAUSSIAN_C,
                            cv.THRESH_BINARY, 81, 4)
```

Wir übergeben hier zunächst unser Bild, gefolgt von dem maximalen Wert (hier 255, da das weiß ist). Dann wählen wir das adaptive Verfahren aus, welches in diesem Fall das Gauß'sche ist (*ADAPTIVE_THRESH_GAUSSIAN_C*). Als nächstes wählen wir den Threshold-Typen, welcher nach wie vor binär sein soll. Die letzten beiden Parameter sind nun essentiell. Der erste ist die *blockSize* und gibt an wie groß die Blöcke in Pixel sein wollen, welche sich das Verfahren ansieht, um das

Thresholding durchzuführen. Je größer dieser Wert ist, desto mehr wird berücksichtigt. Kleinere Feinheiten werden dann jedoch auch weniger beachtet. Dieser Wert muss ungerade sein und 81 passt für dieses Beispiel gut. Der letzte Parameter nennt sich *C* und gibt an, wie viel vom Median abgezogen wird. Mit diesem Parameter muss man oft ein wenig herumexperimentieren, da dieser das Bild oftmals schärfer und glatter macht.

```
cv.imshow('Gaus', gaus)
```

Schauen wir uns unser Resultat einmal an.

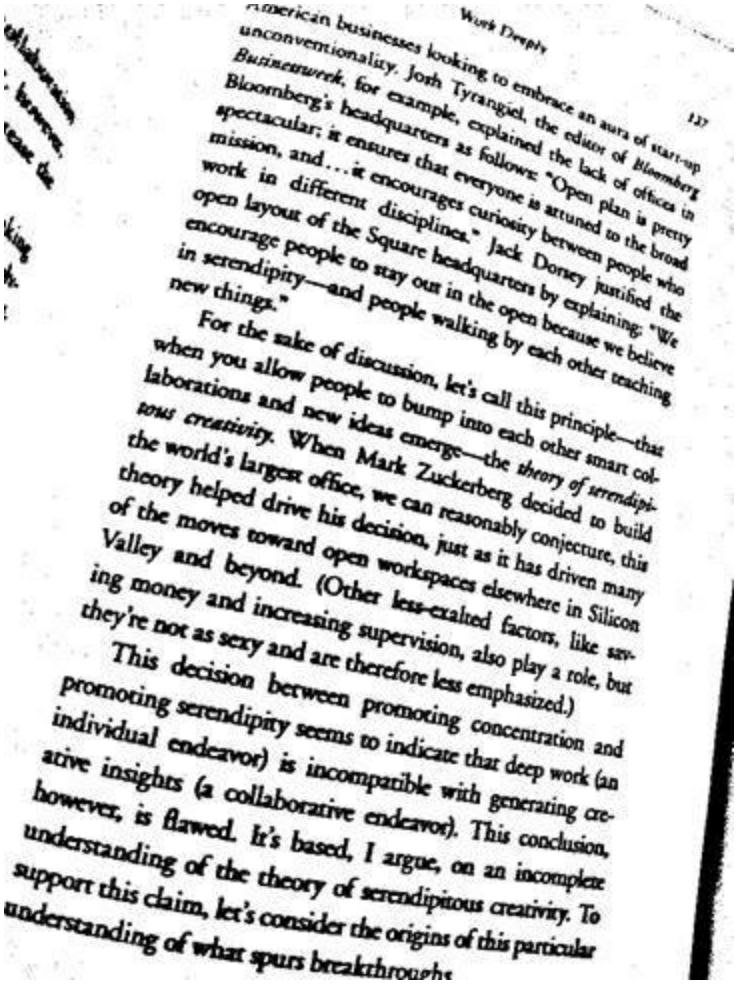


Abb. 3.6: Resultat mittels adaptivem Thresholding

Dieses Resultat ist schon viel eindrucksvoller. Es ist nicht perfekt, aber man kann den Text vollständig entziffern und das Bild ist definitiv angenehmer zu lesen als das Original.

Der Ausschnitt, welchen Sie hier sehen ist übrigens aus dem Buch *Deep Work* von *Cal Newport*.

Wie Sie sehen ist mit Thresholding einiges möglich. Bevor Sie mit dem nächsten Kapitel weitermachen, versuchen Sie doch auch selbst einige schlechte Bilder von Zeitungen oder Büchern zu machen und diese dann mit OpenCV und Thresholding zu verbessern. Experimentiere Sie ein wenig mit Ihrem dazugewonnenen Wissen herum.

4 – FILTERING

In diesem Kapitel wird es um das sogenannte *Filtering* gehen. Hierbei ist jedoch nicht die Rede von den Effekten, welche Sie in sozialen Netzwerken auf Ihre Fotos anwenden können. Vielmehr handelt es sich hier um das Herausfiltern bestimmter Teile aus Bildern und Videos.

Beispielsweise möchten wir vielleicht alle roten bzw. rötlichen Objekte herausfiltern aus einem Video. Oder aber wir interessieren uns nur für jene Segmente, welche heller sind als ein bestimmter Grenzwert.

Abb. 4.1: Foto eines Papageis

Nehmen wir dieses simple Bild als triviales Beispiel. Wir haben hier einen roten Papagei vor einem grün-blauen Hintergrund. Mittels Filtering könnten wir hier zum Beispiel das Gefieder herausfiltern und hervorheben.

Das kann den einfachen Sinn und Zweck haben, dass wir diesen Teil des Bildes einfach herausheben möchten. Oder aber wir benutzen das Filtern zur Erkennung von bestimmten Dingen. Beispielsweise möchten wir vielleicht alle roten Objekte in einem Video erkennen und zählen.

FILTERMASKE ERSTELLEN

Zunächst müssen wir unser Bild oder Video in unser Skript laden. Am Ende dieses Kapitels gehen wir noch einmal kurz durch, wie wir auch Video- und Kameradaten filtern können.

```
img = cv.imread('papagei.jpg')
hsv = cv.cvtColor(img, cv.COLOR_RGB2HSV)
```

Wichtig ist hierbei, dass wir das Farbschema, welches wir benutzen ändern. Wir wandeln unser Bild von RGB auf HSV um. RGB steht für *Red*, *Green* und *Blue*, während HSV für *Hue*, *Saturation* und *Value* steht. Zu Deutsch bedeutet das *Farbwert*, *Sättigung* und *Hellwert*.

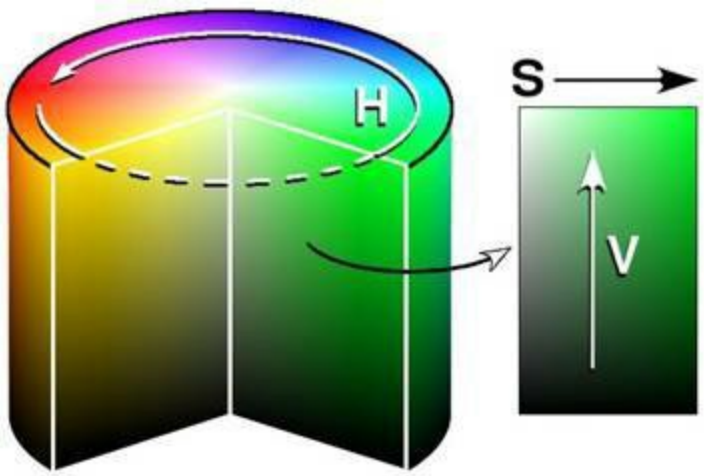


Abb. 4.2: HSV-Farbraum (Quelle: Wikipedia)

Der H-Wert bestimmt also welche Farbe des Spektrums wir wählen. Der S-Wert gibt an, wie hoch unsere Sättigung ist bzw. wie kräftig unsere Farbe ist. Und der V-Wert gibt an, wie hell die gewählte Farbe ist.

Wir wählen dieses Farbschema, da es um einiges leichter ist mit diesen drei Parametern zu filtern und Grenzen zu definieren, als mit den drei RGB-Farben.

```
minimum = np.array([100, 60, 0])
maximum = np.array([255, 255, 255])
```

Das bringt uns gleich zum nächsten Schritt. Nun definieren wir zwei Grenzen, zwischen welchen sich unsere gewünschten Farben befinden. In diesem Fall soll unser Farbwert zwischen 100 und 255 liegen. Dadurch erhalten wir alle Werte, welche rötlich und orange sind. Ebenso verlangen

wir eine Sättigung von mindestens 60, damit gräulichere Töne nicht mitgefiltert werden. Die Helligkeit ignorieren wir hier, daher erlauben wir das volle Spektrum von 0 bis 255.

```
maske = cv.inRange(hsv, minimum, maximum)
resultat = cv.bitwise_and(img, img, mask = maske)
```

Im nächsten Schritt definieren wir eine Maske, mithilfe der *inRange* Funktion. Diese setzt alle Werte, welche in unserem Bild *hsv* zwischen den beiden Grenzwerten liegen auf Weiß und alle anderen auf Schwarz. Dann benutzen wir die Funktion *bitwise_and*, um das ursprüngliche Bild mit sich selbst zu verunden, und hierbei die entstandene Maske anzuwenden.

```
cv.imshow('Maske', maske)
```

Schauen wir uns die Maske einmal an.

Abb. 4.3: Resultierte Filtermaske

Die weißen Pixel, sind also jene, welche wir in unser Resultat übernehmen werden. Alle schwarzen Pixel werden schwarz bleiben. Hierzu sehen wir uns das Resultat an.

```
cv.imshow('Resultat', resultat)
```

Abb. 4.4: Resultierendes Bild

Im Grunde genommen schon ein Ganz nettes Resultat, doch wir sehen hier und da einige Pixel, welche nicht dort sein sollten wo sie sind. Vor allem wenn Sie Ihre Kamera in Echtzeit filtern, werden Sie ein starkes Bildrauschen feststellen.

BLURRING UND SMOOTHING

Um das Resultat nun zu optimieren verwenden wir die Techniken des *Blurrings* und des *Smoothings*. Wir machen unser Endresultat also etwas unschärfer, haben dadurch allerdings auch weniger Bildrauschen und weniger ungewollte Pixel.

Was wir hier als erstes machen, ist ein Durschnitt-Array zu erstellen.

```
durchschnitt = np.ones((15, 15), np.float32) / 225
```

Wir erstellen hier zunächst ein Array mit lauter Einsern, welches die Form 15x15 hat. Dieses dividieren wir dann durch 225 (das Produkt aus 15 und 15). Damit haben wir nun für jedes Pixel einen Faktor der den Durchschnitt berechnet.

Das Ziel hier ist es, mithilfe des Durschnitts der Pixel in einem Bereich von 15x15, einzelne Fehlpixel auszugleichen.

```
smoothed = cv.filter2D(resultat, -1, durchschnitt)
```

Mithilfe der Funktion *filter2D* wenden wir nun unseren Durschnitt-Kernel auf

das Bild an. Der zweite Parameter, welchen wir auf -1 setzen, gibt die Tiefe des Bildes an. Durch die negative Zahl, übernehmen wie die Tiefe des Ursprungsbildes.

```
cv.imshow('Smoothed', smoothed)
```

Abb. 4.5: Resultierendes Bild geglättet (smoothed)

Wie Sie sehen sind die meisten falsch platzierten Pixel weg, doch das Bild ist jetzt etwas unscharf.

Wichtig ist, dass wir hierbei die Reihenfolge der Funktionen beachten. Wir haben hier zunächst die Maske auf unser Bild angewandt und dann dieses Resultat geglättet. Wir können aber auch direkt die Maske glätten und sie dann erst anwenden. Dann erhalten wir ein etwas anderes Resultat.

```
smoothed2 = cv.filter2D(maske, -1, durchschnitt)
smoothed2 = cv.bitwise_and(img, img, mask=smoothed2)

cv.imshow('Smoothed2', smoothed2)
```

Abb. 4.6: Resultierendes Bild nach geglätteter Maske

In diesem spezifischen Beispiel ist diese zweite Methode unbrauchbar. Wir sehen noch viel mehr störende Pixel als vor dem Glätten (Smoothing). Doch bei anderen Verfahren kann das Wechseln der Reihenfolge durchaus sinnvoll sein.

GAUSSSCHER WEICHZEICHNER (BLUR)

Eine weitere Methode, welche wir hier anwenden können ist der gauß‘sche Weichzeichner.

```
blur = cv.GaussianBlur(resultat, (15, 15), 0)
```

Hier übergeben wir nach dem Bild ebenso die Größe der Blöcke, welche weichgezeichnet werden sollen. Das Resultat dieser Methode ist etwas weniger verpixelt als das des normalen Smoothings.

MEDIAN BLUR

Der wahrscheinlich effektivste Weichzeichner ist der *Median Blur*. Dieser bearbeitet jeden Channel eines Bildes einzeln und wendet dabei den Median-Filter an.

```
median = cv.medianBlur(resultat, 15)

cv.imshow('Median', median)
```

Hier übergeben wir nur das Bild und die Größe der Blöcke ergibt sich aus der Seitenlänge, welche wir übergeben. Hier hätten wir also wieder 15x15 Blöcke.

Abb. 4.7: Resultierendes Bild nach Median Blur

Das Resultat hier kann sich definitiv sehen lassen, da es so ziemlich jegliches Bildrauschen eliminiert. Dennoch ist unser Bild jetzt viel zu weich gezeichnet. Hier können wir also mit der Reihenfolge experimentieren und zuerst unsere Maske weichzeichnen.

```
median2 = cv.medianBlur(maske, 15)
median2 = cv.bitwise_and(img, img, mask=median2)

cv.imshow('Median2', median2)
```

Abb. 4.8: Resultat nach Änderung der Reihenfolge

Damit wird unser Resultat schon sehr gut. Das Bildrauschen ist quasi entfernt, doch die Schärfe des ursprünglichen Bildes bleibt erhalten.

KAMERADATEN FILTERN

Diese ganzen Filter können wir wie bereits erwähnt nicht nur bei Bildern anwenden, sondern auch live bei der eigenen Kamera.

```
import cv2 as cv
import numpy as np

kamera = cv.VideoCapture(0)

while True:
    _, img = kamera.read()
    hsv = cv.cvtColor(img, cv.COLOR_RGB2HSV)

    minimum = np.array([100, 60, 0])
    maximum = np.array([255, 255, 255])

    maske = cv.inRange(hsv, minimum, maximum)

    median = cv.medianBlur(maske, 15)
    median = cv.bitwise_and(img, img, mask=median)

    cv.imshow('Median', median)

    if cv.waitKey(5) == ord('x'):
        break

cv.destroyAllWindows()
kamera.release()
```

Wie wir es bereits gelernt haben, haben wir hier ein Kameraobjekt erstellt und in einer Dauerschleife die Frames abgefangen. Diese haben wir dann jedes Mal durch unseren Filter laufen lassen und das Ergebnis haben wir angezeigt. Wenn Sie das zu Hause bei Ihrer eigenen Webcam versuchen, werden Sie merken, dass alles was nicht rötlich oder orange ist, ausgeschwärzt wird.

5 – OBJEKTERKENNUNG

Nun kommen wir zu einem außerordentlich interessanten Themengebiet der Computervision – der Objekterkennung. Zunächst werden wir uns dabei ein paar vorangehende Themen wie Edge Detection, Template- und Feature Matching und Background Subtraction ansehen. Am Ende werden wir dann mittels Kaskadierung tatsächliche Objekte in Bildern und Videos erkennen können.

EDGE DETECTION

Fangen wir zunächst mit der *Edge Detection* bzw. der Kantenerkennung an. Oftmals ist es hilfreich, wenn wir unsere Bilder und Videodaten auf das Wesentliche reduzieren. Wir werden hier zwar keine eigenen Algorithmen für die Objekterkennung schreiben, doch wenn wir das tun würden, wäre unter Umständen auch die Kantenerkennung ein nützliches Element.

Abb. 5.1: Bild eines Raumes

Für dieses Beispiel nehmen wir das oben gezeigte Bild eines gewöhnlichen Raumes. Ein Computer interessiert sich nur bedingt für Details wie Schatten und Konturen. Wir benutzen hier also einen Algorithmus, um die wesentlichen Kanten herauszufiltern.

```
import cv2 as cv

img = cv.imread('raum.jpg')
kanten = cv.Canny(img, 100, 100)
cv.imshow('Kanten', kanten)

cv.waitKey(0)
cv.destroyAllWindows()
```

Die Funktion *Canny* von OpenCV tut dies für uns. Wir übergeben dieser das Bild und zwei Thresholding Parameter, also zwei Toleranzwerte. Das Resultat sieht interessant aus.

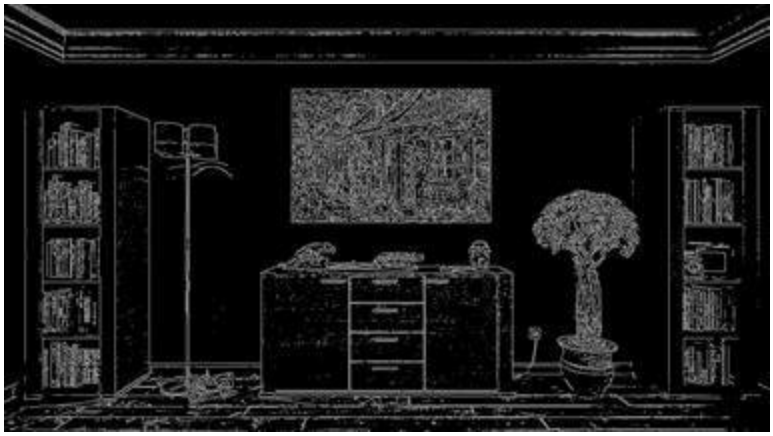


Abb. 5.2: Bild nach Edge Detection

Wie Sie sehen, sind alle wesentlichen Inhalte noch vorhanden, doch das Bild wurde auf die Kanten reduziert.

TEMPLATE MATCHING

Eine weitere interessante Technik ist die des *Template Matchings*. Hierbei geben wir unserem Skript ein paar Templates bzw. Vorlagen bestimmter Objekte, welche es dann in dem Bild finden soll. Anders als bei der Objekterkennung, muss es sich hierbei jedoch um sehr genaue Matchings handeln. Für die allgemeine Erkennung von Gesichtern oder Uhren eignet sich diese Technik also eher weniger.

Abb. 5.3: Bild eines Arbeitsplatzes

Nehmen wir hier als Beispiel dieses Bild eines Arbeitsplatzes. Prinzipiell können wir uns hier jedes beliebige Objekt aussuchen, doch das einzige, welches hier öfters vorkommt sind die Tasten.

Abb 5.4: F-Taste als Template

Wir schneiden also, mit einem Programm wie beispielsweise Gimp, eine beliebige Taste aus, welche wir als Template benutzen werden.

```
img_bgr = cv.imread('arbeitsplatz.jpg')
img_grau = cv.cvtColor(img_bgr, cv.COLOR_BGR2GRAY)

template = cv.imread('taste.jpg', 0)
breite, hoehe = template.shape[::-1]
```

Wir laden zunächst das Hauptbild und das Template in unser Programm. Dann machen wir eine Kopie des Bildes und konvertieren diese in Graustufen. Ebenso speichern wir die Breite und Höhe des Templates in Pixel.

```
resultat = cv.matchTemplate(img_grau, template,
```

```
                cv.TM_CCOEFF_NORMED)
```

Die Hauptarbeit übernimmt nun die Funktion *matchTemplate*. Dieser übergeben wir unser Bild, unser Template und wir geben die Methode an, welche wir verwenden wollen für das Matching. In diesem Fall *TM_CCOEFF_NORMED*.

Was wir hier erhalten ist ein Array mit den jeweiligen Aktivierungen der Bildbereiche, wo dieses Template vorkommt.

```
threshold = 0.8
bereich = np.where(resultat >= threshold)
```

Als nächsten Schritt definieren wir einen bestimmten Threshold bzw. Schwellenwert. Dieser gibt an, wie weit ein Bereich von dem genauen Template abweichen darf. Ein Threshold von 0.8 bedeutet, dass ein Bereich mindestens 80% Ähnlichkeit haben muss mit dem Template, um erkannt zu werden. Diesen Wert können Sie anpassen, wie Sie möchten und ein wenig herumexperimentieren. Die Funktion *where* von NumPy liefert uns jeweils die Indizes der Pixel, bei welchen der Wert hoch genug ist.

```
for punkt in zip(*bereich[::-1]):
    cv.rectangle(img_bgr, punkt,
                 (punkt[0] + breite, punkt[1] + hoehe),
                 (0, 0, 255), 2)
```

Dann lassen wir eine For-Schleife über die gezippte Version der Bereiche laufen. Für jeden Punkt in diesen Bereichen, welcher aktiv genug war, zeichnen wir nun ein Rechteck mit der Breite und Höhe des Templates. Dieses zeigt uns dann an, dass unser Algorithmus dort etwas gefunden hat.

```
cv.imshow('Resultat', img_bgr)
cv.waitKey(0)
cv.destroyAllWindows()
```

Zu guter Letzt können wir uns nun unser Resultat mit den jeweiligen Markierungen ansehen.

Abb. 5.4: Arbeitsplatz nach Template Matching

Falls Sie eine Schwarz-Weiß Version dieses Buches lesen, achten Sie auf die Umrandungen bei den Tasten. Die roten Linien sind dann schwarz in diesem Bild.

Wie Sie sehen erkennt unser Algorithmus ziemlich viele Tasten. Wenn Sie noch mehr Tasten erkennen möchten, müssten Sie die den Threshold niedriger setzen. Dann laufen Sie jedoch Gefahr, dass auch falsche Klassifikationen stattfinden.

FEATURE MATCHING

Stellen Sie sich vor Sie hätten zwei Bilder, welche dieselben Objekte aus verschiedenen Perspektiven zeigen.

Abb. 5.5: Arbeitsplatz aus anderer Perspektive

Hier haben wir beispielsweise den zuvor gezeigten Arbeitsplatz aus einer anderen Perspektive vor uns. Für uns ist es nicht schwer zu erkennen, dass es sich um dieselben Inhalte aus einem anderen Blickwinkel handelt, doch für unseren Computer sind das komplett verschiedene Pixel.

Was uns hier helfen wird ist *Feature Matching*. Hierbei extrahieren Algorithmen für uns die essentiellsten Punkte und Deskriptoren unserer Bilder. Dann sucht es diese Punkte auch im anderen Bild und verbindet sie.

```
img1 = cv.imread('arbeitsplatz1.jpg', 0)
img2 = cv.imread('arbeitsplatz2.jpg', 0)

orb = cv.ORB_create()

keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
keypoints2, descriptors2 = orb.detectAndCompute(img2, None)
```

Nachdem wir unsere beiden Bilder eingelesen haben erstellen wir ein *Orientational BRIEF* Objekt (kurz ORB). Dieses wird uns helfen die essentiellen Punkt zu finden. Wir führen die Funktion *detectAndCompute* dieses Objektes aus und wenden sie auf beide Bilder an. Als Resultat erhalten wir dann die Keypoints und Deskriptoren für beide Bilder.

In diesem Beispiel importieren wir die Bilder in schwarz-weiß (daher die null). Das tun wir, weil man die farbigen Linien am Ende so besser sehen kann. Sie können jedoch auch die originalen Bilder einlesen.

```
matcher = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
matches = matcher.match(descriptors1, descriptors2)
matches = sorted(matches, key = lambda x: x.distance)
```

Nun erstellen wir eine Instanz der Klasse *BFMatcher* und wählen das *NORM_HAMMING* Verfahen aus. Dieser Matcher erlaubt uns die Keypoints zu vereinen und herauszufinden, wo welche Keypoints von dem einen Bild im anderen Bild sind. Das Resultat sortieren wir dann in der letzten Zeile, nach Distanz, sodass wir die kürzesten Distanzen oben haben. Das ist wichtig, da wir im nächsten Schritt nur ein paar der besten Resultate herausfiltern möchten.

```
resultat = cv.drawMatches(img1,keypoints1,
```

```
    img2,keypoints2,
    matches[:10], None, flags=2)
```

Zu guter Letzt visualisieren wir diese Matches noch mit der Funktion *drawMatches*. Hierzu übergeben wir beide Bilder mit den Keypoints und geben an, welche Matches wir darstellen möchten. In diesem Fall die ersten zehn, welche jene mit der kürzesten Distanz sind.

```
resultat = cv.resize(resultat, (1600,900))
cv.imshow('Resultat',resultat)
cv.waitKey(0)
```

Jetzt skalieren wir unser finales Resultat noch, sodass wir es problemlos anzeigen können.

Abb. 5.6: Feature Matching der beiden Bilder

Auf diesem Bild in diesem Buch werden Sie schwer etwas erkennen können, da OpenCV die Linien sehr fein zeichnet. Wenn Sie den Code jedoch auf Ihrem Computer ausführen, so können Sie die Linien besser sehen.

Im Endeffekt werden die wichtigsten Punkte links mit denselben Punkten rechts verbunden. Meistens funktioniert das auch sehr gut. Einige Fehler unterlaufen leider trotzdem.

BEWEGUNGSERKENNUNG

Bevor wir nun zur Objekterkennung übergehen befassen wir uns noch mit einer sehr interessanten Technik, wenn es darum geht Bewegung in Videos zu erkennen – der *Background Subtraction*. Hierbei benutzen wir ein

Verfahren, welches durch die Veränderung von Pixeln erkennt, was der Hintergrund und was der Vordergrund ist und sich dann nur auf den Vordergrund fokussiert.

Abb. 5.7: Ausschnitt eines Videos

Hierfür werde ich ein Video benutzen, in welchem mehrere Menschen auf einer Art Platz herumgehen. Sie können hierbei natürlich Ihre eigenen Videos verwenden oder alternativ die Webcam benutzen.

```
# Alternativ: video = cv.VideoCapture(0)
video = cv.VideoCapture('personen.mp4')
subtractor = cv.createBackgroundSubtractorMOG2(20, 50)
```

Wir laden wie gewohnt unser Video und erstellen dann mittels der Funktion *createBackgroundSubtractorMOG2* einen neuen Subtractor. Diesem übergeben wir (optional) zwei Parameter. Der erste ist die Länge der *history*, also wie weit unser Subtractor zurückblicken soll, und der zweite ist der Threshold. Auch hier können Sie wieder ein wenig mit diesen Werten herumexperimentieren, bis Sie ein optimales Resultat erhalten.

```
while True:
    _, frame = video.read()
    maske = subtractor.apply(frame)

    cv.imshow('Maske', maske)

    if cv.waitKey(5) == ord('x'):
        break

cv.destroyAllWindows()
video.release()
```

Nun können wir wieder in einer Endlosschleife unsere Resultate anzeigen. Das einzige was wir hier tun ist, mit der *apply* Funktion unseren Subtractor auf das aktuelle Frame anzuwenden. Das Resultat ist unsere Maske.

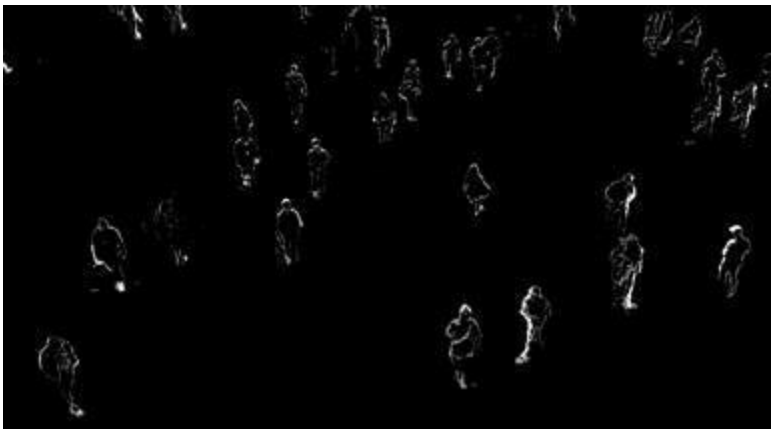


Abb. 5.8: Resultat als Maske

Auch wenn das Resultat nicht perfekt ist und man die Werte noch besser anpassen könnte, erfüllt der Subtractor seinen Zweck. Wir sehen nur jene Teile des Videos, welche sich verändern und damit den vermeidlichen Vordergrund.

OBJEKTERKENNUNG

Nun kommen wir letztendlich zur Objekterkennung. Wir möchten hierbei, anders als beim Template Matching, so generalisiert wie möglich arbeiten. Wir möchten also nicht unbedingt immer denselben Computer, dieselbe Taste oder dasselbe Gesicht erkennen, sondern prinzipiell Gesichter, Computer, Tasten etc. Hierfür benötigen wir jedoch ausgereifte Modelle und Trainingsdaten. Da es viel zu aufwendig ist sich selber welche anzufertigen, werden wir in diesem Buch einfach vorgefertigte Ressourcen aus dem Internet verwenden.

RESSOURCEN LADEN

Wir benötigen sogenannte *HaarCascades*, welche in Form von XML-Dateien bereitgestellt werden. In diesem Buch werden wir eine solche Datei für die Erkennung von Gesichtern und eine solche Datei für die Erkennung von Wanduhren benutzen. Hierfür folgende Links:

Gesichtsdatei: https://bit.ly/3bkHNHs

Uhrendatei: https://bit.ly/3bdLQF8

Beachten Sie hierbei die Lizenzen der Dateien. Die erste Datei ist von der Firma Intel und unterliegt dem Copyright. Solange Sie diese Dateien nur für Ihren Privatgebrauch und für das Lernen benutzen, sollten jedoch keine Probleme entstehen.

```
gesichter_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')
uhren_cascade = cv.CascadeClassifier('clock.xml')
```

Wir erstellen nun zwei *CascadeClassifier* für die beiden Objekte und übergeben die beiden XML-Dateien dabei als Parameter.

OBJEKTE ERKENNEN

Abb. 5.9: Gruppe von Menschen

Zunächst werden wir dieses Bild von einer Gruppe von Menschen benutzen, um ein paar Gesichter zu erkennen.

```
img = cv.imread('personen.jpg')
img = cv.resize(img, (1400, 900))

grau = cv.cvtColor(img, cv.COLOR_RGB2GRAY)
gesichter = gesichter_cascade.detectMultiScale(grau, 1.3, 5)
```

Wir skalieren das Bild und konvertieren es in Graustufen. Dann benutzen wir die Funktion *detectMultiScale* von unserem Classifier, um anhand der XML-

Datei die Gesichter zu erkennen. Wir übergeben hier zwei optionale Parameter. Zum einen den Skalierungsfaktor des Bildes, welcher umso höher gewählt werden kann, je höher die Bildqualität ist und zum anderen die Mindestanzahl an Nachbarklassifizierungen, um sicher zu gehen.

Im Endeffekt war es das auch schon und wir müssen das Ganze nur noch visualisieren.

```
for (x,y,w,h) in gesichter:
    cv.rectangle(img, (x, y),
                 (x + w, y + h),
                 (255, 0, 0), 2)
    cv.putText(img, 'GESICHT',
               (x,y+h+30),
               cv.FONT_HERSHEY_SIMPLEX, 0.8,
               (255,255,255), 2)
```

Wir iterieren hier einmal über jedes erkannte Gesicht und holen uns die zwei Koordinaten und die Breite und Höhe. Mit diesen Daten zeichnen wir dann ein Rechteck um das Gesicht und setzen einen Text darunter.

Abb. 5.9: Klassifizierte Gesichter

Wie Sie sehen ist das Resultat recht eindrucksvoll. Dasselbe machen wir nun auch mit Wanduhren.

Abb. 5.10: Bild eines Zimmers mit Wanduhr

In diesem Zimmer befindet sich eine Wanduhr, welche wir von unserem Skript erkennen lassen möchten. Wir wiederholen also die Vorgehensweise und fügen Teile zu unserem Skript hinzu.

```
gesichter = gesichter_cascade.detectMultiScale(grau, 1.3, 5)
uhren = uhren_cascade.detectMultiScale(grau, 1.3, 10)
```

Hier benutzen wir als Nachbar-Mindestwert die Zahl Zehn, da sonst zu viele Fehler auftreten.

```
for (x,y,w,h) in uhren:
   cv.rectangle(img, (x, y),
                (x + w, y + h),
                (0, 0, 255), 2)
   cv.putText(img, 'UHR',
              (x, y + h + 30),
              cv.FONT_HERSHEY_SIMPLEX, 0.8,
              (255, 255, 255), 2)
```

Auch hier zeichnen und beschriften wir die gefundenen Uhren. Natürlich müssen wir das Bild, welches am Anfang eingelesen wird auch noch ändern.

Abb. 5.11: Klassifizierte Wanduhr

Auch das funktioniert wie Sie sehen sehr gut. Ich konnte leider kein lizenzfreies Bild finden, auf welchem sich Personen und Wanduhren befinden. Doch natürlich würde unser Skript auch mehrere verschiedene Objekte gleichzeitig erkennen. Ebenso funktioniert das alles auch mit Videos und Kameradaten.

Experimentieren Sie ein wenig mit verschiedensten Bildern herum. Suchern Sie vielleicht nach anderen HaarCascades und arbeiten Sie mit der Kamera. Seien Sie kreativ und probieren Sie neue Sachen, denn nur so lernen Sie wirklich viel dazu.

WIE GEHT ES JETZT WEITER?

Wenn Sie die Konzepte in diesem Buch verstanden haben und wissen wie Sie diese anwenden können, so sind Sie auf Ihrer Programmierkarriere einen großen Schritt weitergekommen. Diese Fähigkeiten sind heutzutage und noch viel mehr in der Zukunft von unschätzbarem Wert.

Sie sind in der Lage Bild- und Videodaten auf sehr komplexe Weise zu verarbeiten und wichtige Informationen herauszufiltern. In Kombination mit Machine Learning und Data Science sind das mächtige Werkzeuge.

Je nach Anwendungsfall, werden Sie sich zusätzliche Skills aneignen müssen, da kein Buch der Welt, Ihnen alles auf den Weg geben kann, was Sie wissen müssen. Wenn Sie Ihre Fähigkeiten in der Wissenschaft unter Beweis stellen möchten, benötigen Sie dort das jeweilige Fachwissen. Genauso in der Medizin, im Sport und in jedem anderen Bereich. Informatik und Mathematik alleine bringen Sie nicht weit. Sie sollten nun jedoch eine solide Basis für die Weiterbildung haben. Ob Sie damit nun Überwachungssysteme aufsetzen, Objekterkennung implementieren oder etwas anderes, bleibt Ihnen überlassen.

Sollten Sie sich für mehr Machine Learning interessieren, statten Sie meiner Amazon-Autorenseite einen Besuch ab. Dort habe ich mehrere Bücher welche sich mit Python für Machine Learning auseinandersetzen.

NEURALNINE

Hier möchte ich Sie noch auf eine Ressource hinweisen, welche sehr viel mit diesem Buch zu tun hat. Neben meinen deutschsprachigen Büchern, produziere ich auch sehr viel englischsprachigen Content. Sollte die Sprache also für Sie kein Problem sein, so können Sie sich gerne die Inhalte von NeuralNine ansehen.

YouTube: https://bit.ly/3a5KD2i

Webseite: https://www.neuralnine.com/

Instagram: https://www.instagram.com/neuralnine/

Auf meinem YouTube Kanal finden Sie kostenlose Tutorials über das Thema Programmieren und Machine Learning. Auf meiner Webseite erwarten Sie viele Blog Posts, welche auch etwas tiefer in die Mathematik mancher Algorithmen gehen. Und auf meiner Instagram Seite veröffentliche ich täglich Infographics zum Thema Programmieren und Informatik.

ABSCHLIESSENDE WORTE

Zu guter Letzt hätte ich noch eine kleine Bitte an Sie. Es dauert nur wenige Minuten und kostet Sie keinen einzigen Cent. Falls Ihnen dieses Buch gefallen hat und Sie der Meinung sind, dass Sie etwas gelernt haben, würde ich mich sehr freuen, wenn Sie eine kurze Rezension auf Amazon verfassen könnten. Damit helfen Sie mir und zukünftigen Lesern dabei hochqualitativere Bücher zu schaffen. Ebenso können Sie gerne Kritikpunkte anbringen, sodass diese verbessert werden können.

Ich bedanke mich herzlichst, bei Ihnen fürs Lesen und ich hoffe zu tiefst, dass Sie dieses Buch einen Schritt weiter auf Ihrer Programmierkarriere gebracht hat.

Florian Dedov

Falls Ihnen diese Sammlung gefallen hat, würde Ich sie gerne darum bitten, auf Amazon eine kleine Rezension zu schreiben. Es kostet sie wenige Minuten und keinen Cent, mir hilft es jedoch enorm!

Danke!