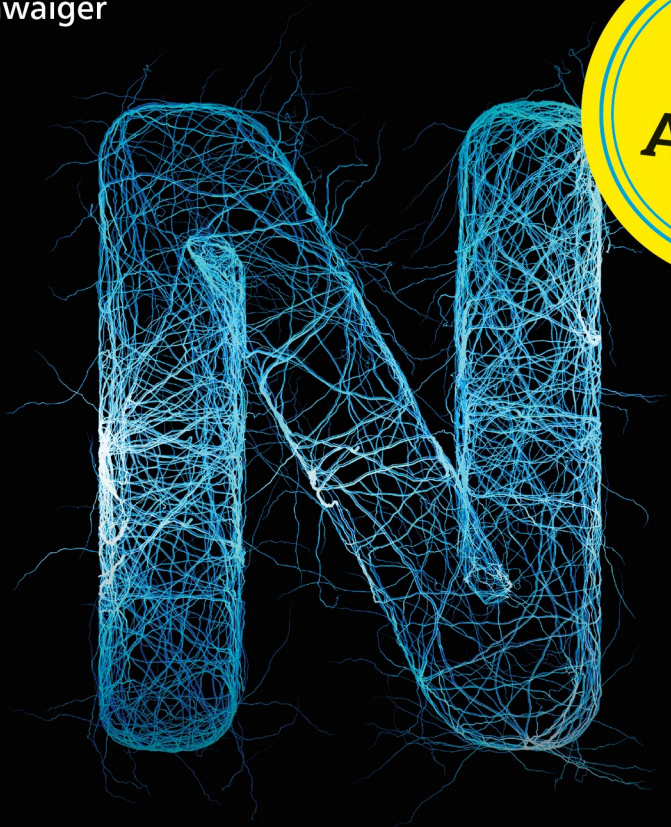
Joachim Steinwendner
Roland Schwaiger

2. Auflage



Neuronale Netze programmieren

mit Python

- Schneller Einstieg mit allen Python- und Mathegrundlagen
- Lernalgorithmen, Aktivierungsfunktionen, Backpropagation u. v. m.
- Mit fertiger Lernumgebung, Einstieg in TensorFlow 2 und Keras

Alle Beispielprojekte zum Download

Rheinwerk Computing

Kapitel 2

Das minimale Starterkit für die Entwicklung von neuronalen Netzen mit Python

Am Ende sieht's ein Tor, ein Klügrer in der Mitte,
und nur der Weise sieht das Ziel beim ersten Schritte.
Friedrich Rückert (deutscher Schriftsteller, 1788–1866)

Damit der erste Schritt leichter fällt, werden wir uns in diesem Kapitel zuerst ein Arbeitsumfeld schaffen. Aus technischer Sicht ist das eine Entwicklungsumgebung für die Programmierung von neuronalen Netzen in der Programmiersprache Python.

2.1 Die technische Entwicklungsumgebung

Wir richten uns also ein und installieren eine sogenannte *integrierte Entwicklungsumgebung*, die Werkzeuge zum Editieren des Quellcodes, zum Testen, zum Debuggen (d. h. zur Fehlersuche) und natürlich zum Kompilieren (d. h. zum Übersetzen in computerverständlichen Code) enthält. In der englischen Sprache ist dafür auch die Abkürzung *IDE* gebräuchlich, die für *Integrated Development Environment* steht.

Es gibt unzählige Arten, Python auf Rechnern mit unterschiedlichen Betriebssystemen zu installieren. Auf der Website der Python Software Foundation (*https://docs.python.org/3/using/index.html*) finden Sie Installationsanweisungen für die drei wichtigsten Betriebssysteme macOS, Windows und Linux.

2.1.1 Die Anaconda-Distribution

Die Installationsanweisungen auf der Website der Python Software Foundation sind sehr lehrreich, dennoch empfehlen wir die Verwendung der Anaconda-Distribution (*https://www.anaconda.com*). Anaconda ist eine der beliebtesten Data-Science-Plattformen und kommt mit einigen sehr hilfreichen Eigenschaften:

- einer große Anzahl an wichtigen Data-Science-Modulen, mit den für uns wichtigen Modulen NumPy, scikit-learn, TensorFlow, Keras
- dem Virtual Environment Manager für Windows, Linux und macOS; dieser erlaubt das Arbeiten in Entwicklungsumgebungen mit ganz spezifischen Versionen von Modulen
- dem Conda-Paket zur Installation und zum Upgraden von Modulen

Wichtig ist zu erwähnen, dass wir in diesem Buch durchgängig Python Version 3 verwenden, also sollten Sie auch die Anaconda-Version für Python 3.x herunterladen (*x* steht hier für eine beliebige Zahl größer gleich 0). Die detaillierten Installationsanleitungen finden Sie unter folgender URL: *https://docs.anaconda.com/anaconda/*.

Diese Webseite enthält noch zusätzliche Informationen und Tausende von Paketen, die bei Bedarf sehr einfach installiert werden können.

Aufgabe

Bevor Sie nun im Buch fortfahren, installieren Sie zuerst die Anaconda-Distribution auf Ihrem Rechner. Die Distribution ist so gestaltet, dass die Installation mit wenig Aufwand und selbstständig durchläuft. Laden Sie auch gleich das *Cheat Sheet* für Anaconda (*https://docs.anaconda.com/anaconda/user-guide/cheatsheet*) herunter, das eine kompakte Übersicht zur Anaconda-Distribution bietet. Danach starten Sie den Anaconda Navigator.

Anaconda Navigator

Für die Verwaltung und das Starten von Entwicklungsumgebungen bietet die Anaconda-Distribution den sogenannten *Anaconda Navigator* an. Dieses Programm erlaubt die Administration der verschiedenen Python-Pakete, und wir können daraus unter anderem unser Hauptwerkzeug für die Python-Programmierung, das Jupyter Notebook, starten. Sobald der Anaconda Navigator gestartet ist, ist es möglich, unterschiedliche Applikationen unter der aktuellen Umgebung (bzw. Environment) zu starten – am Beispiel von Abbildung 2.1 wären das qtconsole, spyder, glueviz, orange, rstudio.

Deutsch – englisch?

Für viele Moduldistributionen, die wir hier verwenden, gibt es oft nur englische Dokumentation und Begriffe. Wir werden versuchen, diese Begriffe zu erklären, aber im Text den englischen Begriff verwenden. Wir opfern also die Schönheit der deutschen Sprache gezwungenermaßen der Eindeutigkeit. Das sei an folgendem Beispiel illustriert: Im Anaconda Navigator kann man unterschiedliche *Umgebungen* einrichten, im Navigator wird das als *Environment* bezeichnet.

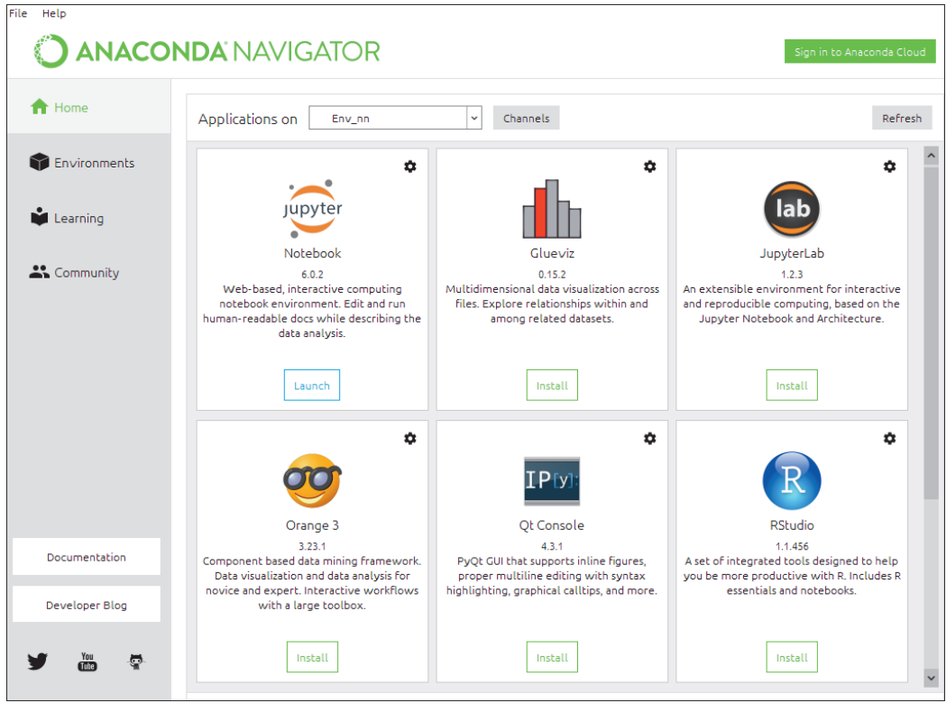


Abbildung 2.1 Der Anaconda Navigator

Das aktuelle Environment ist bei ausgewähltem HOME-Menü in der obersten Zeile angegeben, in unserem Beispiel in Abbildung 2.2 ist das die Umgebung mit der Bezeichnung ENV_NN.

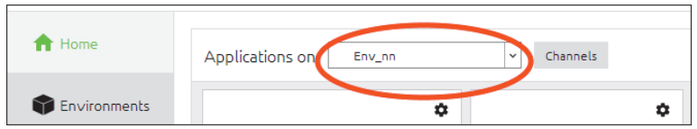


Abbildung 2.2 Die aktuelle Umgebung (Environment)

Erstellen von neuen Environments

Welchen Sinn eigene *Environments* haben, versuchen wir an folgenden Beispielen zu erklären:

- Für manche Projekte braucht es nur eine kleine Anzahl an Modulen. Da wäre es sinnlos, alle vorhandenen Pakete in einem Environment zu installieren.

- Die Open Source Community ist sehr schnell, das bedeutet auch das Vorhandensein von unterschiedlichen Versionen des gleichen Moduls. Manchmal brauchen Module, die aufeinander aufbauen, ganz bestimmte Versionen.
- Manche Module haben geräteabhängige Versionen (z. B. TensorFlow und TensorFlow-gpu – für PCs mit einer oder mehreren GPUs). Auch dafür lassen sich eigene Environments einrichten.

Im Anaconda Navigator befindet sich im linken Bereich der Menüpunkt ENVIRONMENTS, der uns eine Übersicht der Environments bietet (mittlerer Bereich) und der pro Environment installierten Pakete (rechter Bereich), wie in Abbildung 2.3 ersichtlich.

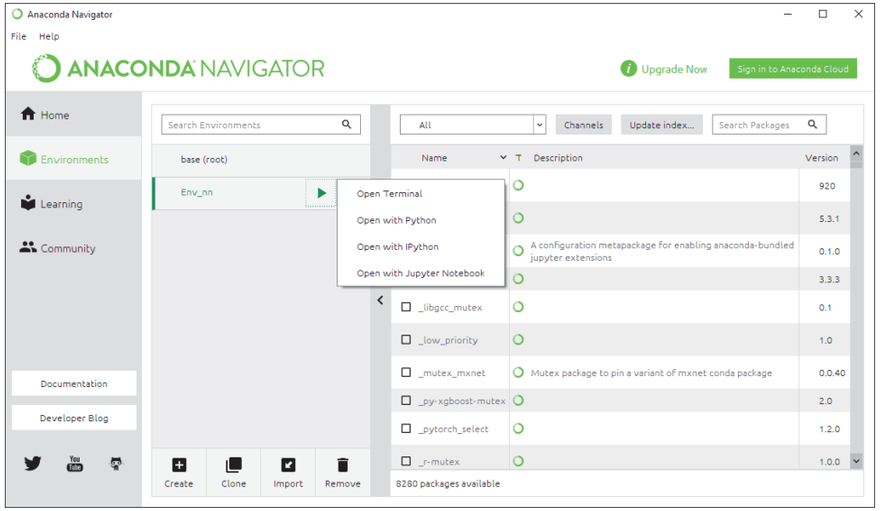


Abbildung 2.3 Das »Environment«-Fenster

Mit Druck auf den CREATE-Knopf (mittlerer Bereich unten) können Sie ein neues Environment erstellen und einen passenden Namen vergeben.

Installation von neuen Paketen

Für die Installation von neuen Pakten in einem Environment gibt es mehrere Methoden – hier werden wir zwei Möglichkeiten zeigen. Eine richtet sich mehr an die Fans von grafischen Benutzeroberflächen (Graphical User Interface, GUI), während die andere für Verfechter der Kommandozeile (Command-Line Interface, CLI) gedacht ist.

- **Installation über die grafische Benutzeroberfläche (GUI)**
  Zuerst müssen Sie ein Environment auswählen oder neu erstellen durch Auswahl des Menüpunktes ENVIRONMENT im Explorer-Teil, der sich links im Anaconda Navigator

befindet (siehe Abbildung 2.3). Dann klicken Sie im nun erscheinenden mittleren Teil auf ein Environment (oder erstellen ein neues). Anschließend wählen oder suchen Sie im rechten Teil ein Paket, das installiert werden soll. Das Anklicken der Schaltfläche APPLY startet dann die Installation.

- **Installation über die Kommandozeile (CLI)**
  Eine weitere Installationsmöglichkeit ergibt sich über die Eingabe des folgenden Kommandos in ein *Terminal*. Das Terminal öffnen Sie, indem Sie in der ENVIRONMENT-Ansicht mit der linken Maustaste auf das Dreiecksymbol neben einem Environment-Namen klicken: Es öffnet sich ein Untermenü, wie Sie es in Abbildung 2.3 sehen. Der erste Menüpunkt, OPEN TERMINAL, öffnet ein Kommandozeilenfenster. In der Eingabeaufforderung (auch als *Prompt* bezeichnet) ist auch das aktuelle Environment ersichtlich.

  Für die Installation eines Pakets (in unserem Beispiel `scipy`) ist folgender Befehl im Terminal einzugeben:

  `conda install scipy` (für die neueste `scipy`-version) oder

  `conda install scipy=0.15.0` (für die `scipy`-version 0.15.0)

  Dieser Installationsvorgang kann durchaus ein paar Minuten dauern, da zum einen Pakete aus dem Internet heruntergeladen werden und zum anderen Abhängigkeiten zu anderen Paketen bestehen können, die es erfordern, zusätzliche Pakete zu installieren. Aber keine Angst, die Abhängigkeiten sind vorgegeben, und jedes Paket weiß sozusagen selbst, was es noch braucht. Abbildung 2.4 zeigt die Kommandozeilen-Variante der Installation von `scipy`, und im unteren Drittel des Terminals sind die zusätzlich notwendigen Pakete bzw. Updates angezeigt.

Terminal = Kommandozeilenfenster

Das Fenster für die Eingabe von Kommandozeilen wird oft auch als *Terminal* bezeichnet, so auch hier im Anaconda Navigator. Die Eingabezeile im Terminal beginnt mit einer sogenannten *Eingabeaufforderung* oder *Prompt*. Diese Prompts sind abhängig vom Betriebssystem sehr unterschiedlich (Windows: `C:\>`, Unix: `benutzer@maschine:~$`. In der Kommandozeile ist ein Befehl einzugeben, der mit der Eingabetaste abgeschlossen und ausgeführt wird.

Aufgabe

Erstellen Sie ein neues Environment mit dem Namen »Env_nn«. Installieren Sie das Paket `numpy` über die grafische Benutzeroberfläche (GUI) und das Paket `pandas` über das Kommandozeilenfenster (CLI).

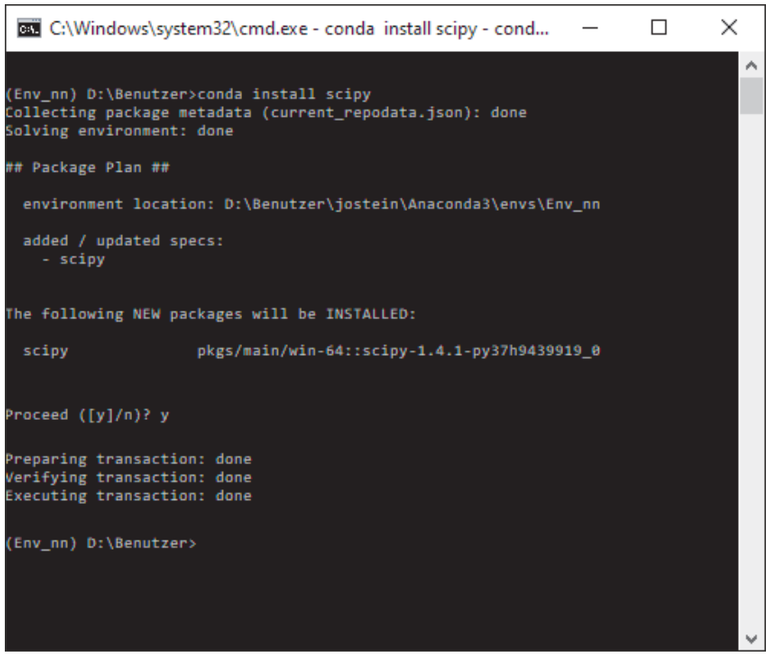


Abbildung 2.4 Installation via Terminal im Environment »Env_nn«

2.1.2 Unser Cockpit: Jupyter Notebook

Ein *Jupyter Notebook* ist eine Webapplikation, die es erlaubt, Dokumente zu kreieren und mit einer Community zu teilen, und die Python-Code, Gleichungen, Visualisierungen sowie erklärenden Text enthält. Jupyter Notebooks sind also eine Art digitaler Notizblock.

Neben dem Vorteil der Kombination von Dokumentation und Code können diese Notizblöcke auch leicht ausgetauscht werden. Die Notebooks werden im sogenannten *JSON*-Format gespeichert.

Es ist somit das ideale Werkzeug bzw. die ideale Entwicklungsumgebung, um mit neuronalen Netzen und Python-Code zu experimentieren, aber auch ernsthafte Applikationen zu entwickeln.

Der Begriff Jupyter

Das Jupyter Notebook ist ein Ergebnis des Projekts Jupyter (*http://jupyter.org*). Der Projektname leitet sich von den drei Programmiersprachen **Ju**lia, **Pyt**hon und **R** ab.

Starten von Jupyter Notebook

Jupyter Notebook lässt sich im Anaconda Navigator unter Auswahl des gewünschten Environments entweder im HOME-Bereich starten (siehe Abbildung 2.1 – bildhafte Darstellung von Jupyter Notebook links oben) oder im ENVIRONMENT-Bereich durch einen Klick auf das Dreieckssymbol des entsprechenden Environments und Auswahl von Jupyter Notebook (siehe Abbildung 2.3).

Jupyter Notebook in der Cloud

Mit Anaconda haben Sie Jupyter Notebook lokal auf Ihrem Rechner installiert und sind damit mehr oder weniger unabhängig von einer Anbindung an das Internet. Da Jupyter Notebook eine Webapplikation ist, bietet es sich an, dieses Werkzeug in der Cloud zur Verfügung zu stellen. Das hat den Vorteil, dass man mehr Rechenpower einsetzen kann, was vor allem beim Arbeiten mit tiefen neuronalen Netzen mit vielen Daten von Vorteil ist. In der Tat gibt es eine Unmenge an Anbietern, die Ressourcen via Jupyter Notebook zur Verfügung stellen – allerdings auch ab einer gewissen Nutzungsintensität kostenpflichtig sind. Zudem muss man sich entsprechend registrieren und oftmals auch eine Kreditkarte angeben. Am Ende des Kapitels geben wir eine (unvollständige) Liste an Anbietern an.

Die Zellen eines Jupyter Notebooks

Ein Dokument, das mit Jupyter Notebook erstellt wird, besteht aus einer Liste von aufeinanderfolgenden Zellen. Diese Zellen können verschiedene Ausprägungen annehmen. Wir interessieren uns nur für die zwei wichtigsten:

1. *Markdown*-Zelle: Dieser Zelltyp erlaubt es, einen erklärenden Text einzugeben. Dieser Zelltyp ist allerdings viel mächtiger, denn wir können den Text auch mit Schriftart, Fettdruck, Formeln, Weblinks, Bildern etc. formatieren. Markdown ist eine Auszeichnungssprache, die mit dem Ziel entwickelt wurde, dass schon die Ausgangsform leicht lesbar ist. Diese besteht aus reinem Text inklusive Formatierungszeichen; z. B. erzeugt die Eingabe `**Fettdruck**` in einer Markdown-Zelle **Fettdruck**.
2. *Code*-Zelle: Mit Zellen dieser Art können wir programmieren. Hier können wir Python-Code eingeben. Natürlich können wir auch hier Python-Kommentare eingeben, allerdings ohne Formatierungsmöglichkeit.

Beiden Zelltypen ist gemein, dass ein Text eingegeben wird (entweder Text mit Formatierungszeichen oder Python-Code). Um den Inhalt dann als formatierten Text auszugeben oder den Code ablaufen zu lassen, muss die Zelle erst »ausgeführt« werden, das heißt, der Prozess zur Übersetzung in formatierten Text bzw. der Durchführung der Codeanweisung muss erst angestoßen werden, entweder durch eine Menüauswahl (CELL • RUN CELLS) oder durch eine Tastenkombination (⇧+↵).

Das Jupyter-Notebook-Dokument wird im sogenannten JSON-Format abgespeichert und hat die Dateierweiterung *.ipynb*. Das Dokument kann so einfach weitergegeben werden. Informationen zum JSON-Format finden sich auf der Website *http://www.json.org*. Wir gehen nicht näher darauf ein, gesagt sei nur so viel, dass es sich um ein schlankes Datenaustauschformat handelt, das für Menschen einfach zu lesen und zu schreiben und für Computer leicht zu übersetzen und zu erzeugen ist.

Aufgabe

Und los geht's. Starten sie ein Jupyter Notebook, und versuchen Sie, die Inhalte von Abbildung 2.5 nachzubauen. Experimentieren Sie mit dieser Entwicklungsumgebung. Wer des Programmierens noch nicht so mächtig ist bzw. noch keine Erfahrung mit Python hat, kann sich jetzt den Anhang A, »Python kompakt«, zu Gemüte führen und schon einige Codebeispiele im Jupyter Notebook ausprobieren.

Interaktives Python

Jupyter ist so konzipiert, dass man es mit verschiedenen Programmiersprachen verwenden kann, daher auch die Herkunft des Namens (siehe Kasten »Der Begriff Jupyter«). Um andere Sprachen verwenden zu können, müsste der entsprechende Kernel installiert werden (jetzt wird der Begriff *Kernel* in der Menüleiste von Abbildung 2.5 klarer). Wir verwenden den Python-Kernel oder genauer gesagt den IPython-Kernel, wobei das I in IPython für Interactive steht. Dieser IPython-Kernel bietet uns noch erweiterte Funktionalitäten, die wir in den Notebooks zur Verfügung haben. Zumeist sind das Kommandos, die mit speziellen Zeichen, wie %, ?, ! oder auch der ⇥-Taste aufgerufen bzw. kombiniert werden. Die Kommandos, die mit dem Präfix % aufgerufen werden, nennt man auch *Magic Functions*. Die Magic Function `%quickref`, die einfach in eine Code-Zelle eines Jupyter Notebooks eingegeben wird, liefert eine sehr gute Übersicht über die Interaktivitätsfunktionalitäten von Jupyter. Wir beschreiben in der Folge einige wenige und wichtige Beispiele zum Zugriff auf die Dokumentation von Python-Funktionen bzw. zum Debuggen von Python-Code in Jupyter Notebooks.

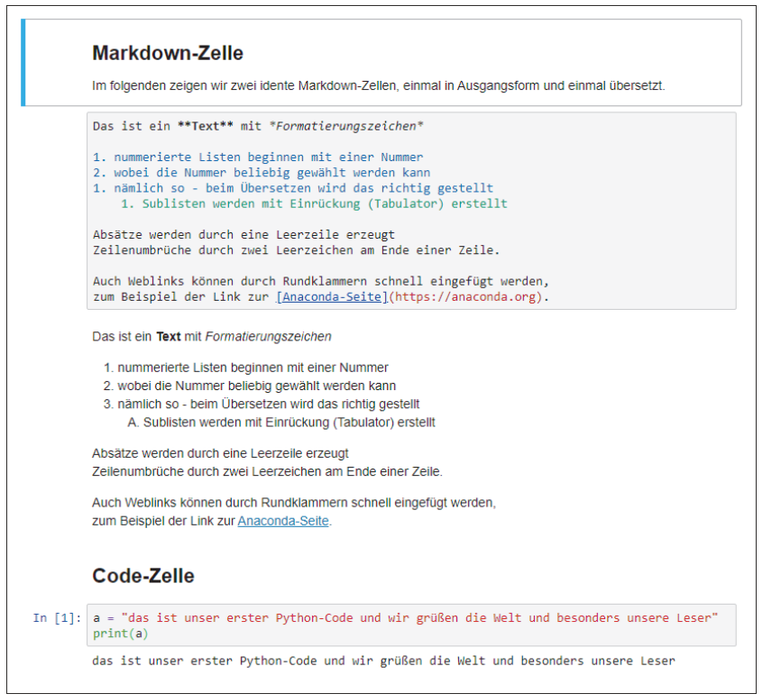


Abbildung 2.5 Markdown- und Code-Zellen in einem Jupyter-Notebook-Dokument

Zugriff auf Hilfe und Dokumentation

In Bereich des Data Science ist eine unglaubliche Fülle an Informationen vorhanden, die es auch einem erfahrenen Data-Science-Experten kaum möglich macht, alles im Kopf zu behalten. Viel wichtiger ist das effektive und schnelle Auffinden von Informationen, sei es in der Internetwolke, in Bedienungsanleitungen oder innerhalb der Entwicklungsumgebung.

Jupyter Notebooks bietet für den Benutzer einige einfache und schnelle Methoden, Fragen zu beantworten, wie:

- Welche Argumente braucht dieser Funktionsaufruf?
- Wie wurde eine Funktion oder Klasse implementiert?
- Was enthält das importierte Modul an Funktionen, Methoden etc.?

Eine schnelle und nützliche Informationsquelle ist der *Docstring*, der eine knappe und präzise Zusammenfassung eines Objekts enthält. Die folgenden Codebeispiele zeigen, wie man schnell zu dieser Information kommt:

```
# Der Docstring kann mit der Funktion help ausgegeben werden
help(print)
# Ausgabe:
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Listing 2.1 Docstring-Ausgabe

Statt der Funktion `help()` kann aber auch ganz einfach ein Fragezeichen am Ende der Funktion gesetzt werden. Abhängig von der Entwicklungsumgebung wird das in einer Ausgabezelle angezeigt, in einem Popup-Fenster oder in Jupyter Notebook in einem eigenen Ausgabebereich (siehe Abbildung 2.6).

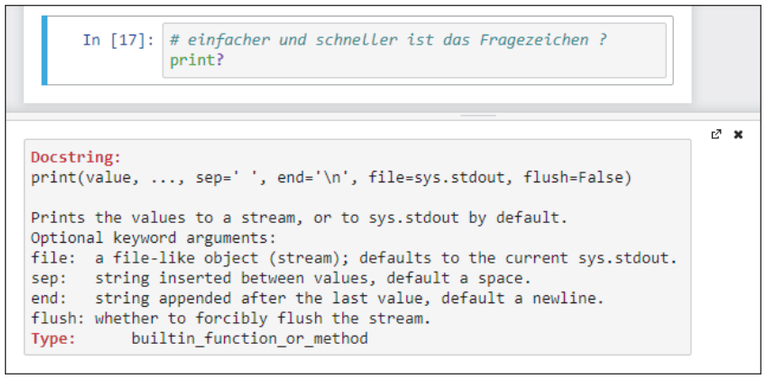


Abbildung 2.6 Ausgabebereich bzw. Pager im Jupyter Notebook

Das Fragezeichen kann sehr vielfältig eingesetzt werden, es kann an Funktionen angehängt werden oder aber auch an Objekte, wie in Listing 2.2 zu sehen ist. Dabei gibt es im ersten Fall Information zur Funktion `myList.append` im Docstring-Format aus und im zweiten Fall Informationen über das Objekt `myList`, wie Typ, Form oder Länge.


```
# das Fragezeichen ? kann vielfältig eingesetzt werden
myList = ['NumPy', 'scikit', 'Keras', 'TensorFlow']
myList.append?
# Ausgabe: (im Pager-Fenster - ganz unten im Webbrowser)
Signature: myList.append(object, /)
Docstring: Append object to end of the list.
Type:      builtin_function_or_method

# oder einfach Information über ein Objekt
myList?
# Ausgabe: (im Pager-Fenster - ganz unten im Webbrowser)
Type:        list
String form: ['NumPy', 'scikit', 'Keras', 'TensorFlow']
Length:      4
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

Listing 2.2 Das Fragezeichen

> **Hinweis**
>
> Da sich die Open-Source-Welt sehr schnell ändert, können die hier dargestellten Ausgaben auf Ihrem Rechner abhängig von der installierten Python-Version etwas anders aussehen. Die Ausgaben in diesem Listing zeigen Ausgaben von Python Version 3.7.

Jupyter Notebook bietet natürlich auch über die grafische Benutzeroberfläche eine Möglichkeit, schnell an Informationen über Funktionen und Objekte zu kommen. Setzen Sie den Cursor auf den Funktions- oder Objektnamen und drücken Sie die Tastenkombination ⇧+⇆, erscheint eine Ballonhilfe, wie in Abbildung 2.7 gezeigt.

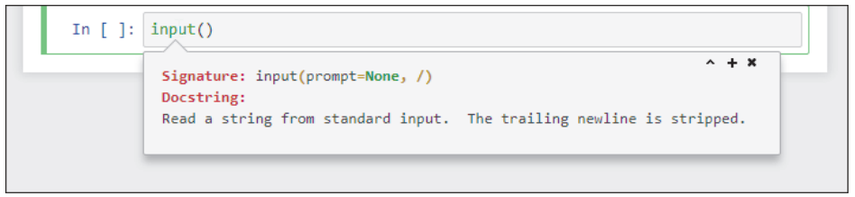


Abbildung 2.7 Kontexthilfe zur Builtin-Funktion »input()«

Aber auch die Tabulator-Taste ⇆ lässt uns Inhalte von Modulen, Klassen oder Objekten erforschen. In Abbildung 2.8 sehen Sie eine Auflistung der Funktionen des Moduls `math` bei Eingabe von `m.` ⇆ in eine Code-Zelle von Jupyter.

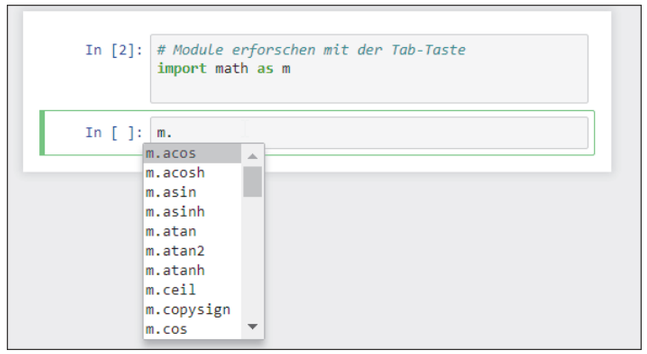


Abbildung 2.8 Die Tabulator-Taste listet die Attribute und Funktionen von Modulen auf ...

Das gilt natürlich auch für unser Listenobjekt `myList` aus Listing 2.2. mit der Eingabe von `myList.` ⇆.

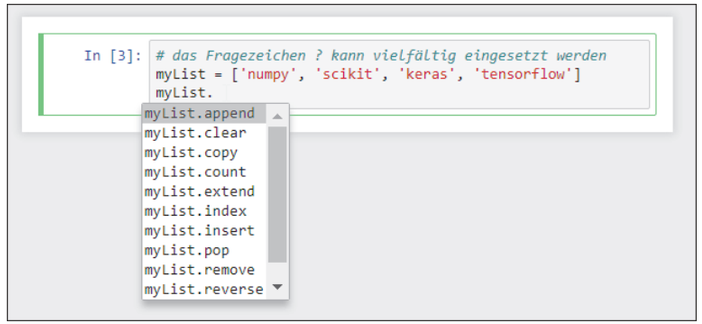


Abbildung 2.9 ... und auch von Objekten.

Fehler und Debugging

Für eine einfache Fehleranalyse gibt es die Magic Function `%xmode`, die es erlaubt, die Menge an ausgegebener Fehlerinformation zu steuern. Diese Funktion verlangt eines der Argumente `Plain`, `Context`, `Verbose`, die den Detaillierungsgrad an Debug-Information im Falle eines Fehlers steuern.

Debugging – Fun Fact

Unter diesem Begriff versteht man in der Welt der Programmierer das Nachverfolgen des Anwendungsablaufs. Das wird vor allem dann gemacht, wenn die Anwendung nicht das tut, was sie sollte, beziehungsweise einen Fehler (also einen *Bug*) generiert. Der Ursprung des Wortes wird oft auf den 9. September 1945 gesetzt.

Man fand Ungeziefer (Bug), genauer gesagt eine Motte, in einem Relais eines Computers. Diese Motte war verantwortlich für einen Fehler, der diesen Computer lahmlegte. Abbildung 2.10 zeigt das damalige Logbuch mit der eingelegten Motte. Allerdings wurde der Begriff schon vorher in diesem Sinne verwendet, die damalige Mitarbeiterin fand es wie wir witzig, dass ein echter Bug zu einem Computerausfall führte.

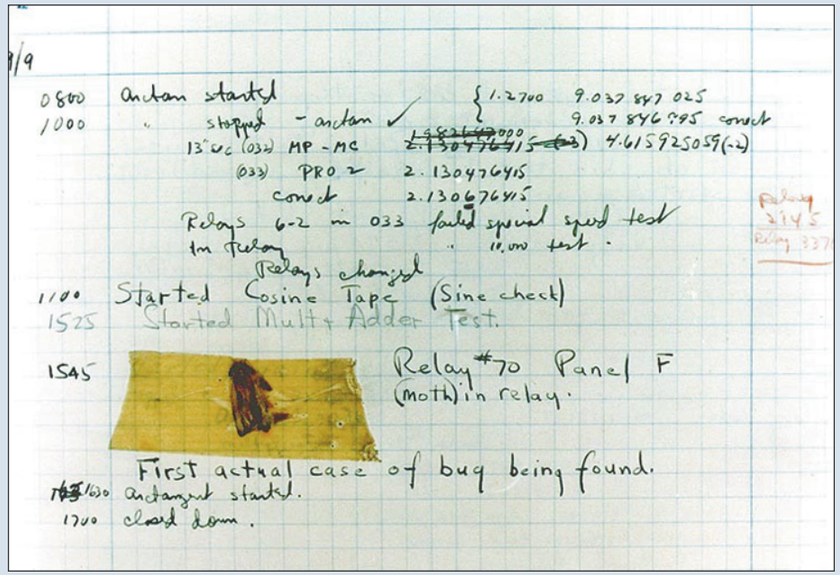


Abbildung 2.10 Der erste Bug in der Computergeschichte
(Quelle: U.S. Naval Historical Center Online Library)

In den folgenden Beispielen sehen Sie die Auswirkungen. Wir greifen hier schon ein bisschen auf die Python-Programmierung vor. Wenn Sie sich da noch unwohl fühlen, verweise ich auf Anhang A. Wir definieren uns zuerst eine einfache Funktion `avg_beelength()`, die die Aufgabe hat, die Mittelwerte der Längen unserer Bienen zu berechnen. Innerhalb dieser Funktion rufen wir dann die Funktion `fehlerfunc()` auf, wohl wissend, dass der Aufruf einen Fehler generieren könnte.

```
def fehlerfunc(x,y):
    return x/y

def avg_beelength(valuelist):
    total = sum(valuelist)
    count = len(valuelist)

    return fehlerfunc(total,count)

bees = (3, 3.5, 4.1, 5.2, 5, 2)
nobees = ()
avg_beelength(bees)
# Ausgabe:
3.8000000000000003
```

Listing 2.3 Die Funktion »avg_beelength« ohne Fehler

So weit, so gut, Listing 2.3 definiert unsere beiden Funktionen. Mit der Liste der Bienenlängen, `bees`, erhalten wir ein passables Ergebnis und vor allem keinen Fehler.

Sehen wir uns in Listing 2.4 an, welche Fehlermeldung angezeigt wird, abhängig vom eingestellten `%xmode`, wenn wir die gleiche Funktion mit der `nobees`, also einer Liste ohne Werte, aufrufen.

```
# Das ist die Standardeinstellung
%xmode Context
avg_beelength(nobees)
# Ausgabe:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-11-be3dced12b16> in <module>
      1 # Das ist die Standardeinstellung
      2 get_ipython().run_line_magic('xmode', 'Context')
----> 3 avg_beelength(nobees)

<ipython-input-10-185b8ad2cde6> in avg_beelength(valuelist)
      6     count = len(valuelist)
      7
----> 8     return fehlerfunc(total,count)
      9
     10 bees = (3, 3.5, 4.1, 5.2, 5, 2)
```



```
<ipython-input-10-185b8ad2cde6> in fehlerfunc(x, y)
      1 def fehlerfunc(x,y):
----> 2     return x/y
      3
      4 def avg_beelength(valuelist):
      5     total = sum(valuelist)

ZeroDivisionError: division by zero
# etwas kompaktere Fehlermeldung
%xmode Plain
avg_beelength(nobees)
# Ausgabe:
Exception reporting mode: Plain
Traceback (most recent call last):

  File "<ipython-input-12-fe49876a1c72>", line 3, in <module>
    avg_beelength(nobees)

  File "<ipython-input-10-185b8ad2cde6>", line 8, in avg_beelength
    return fehlerfunc(total,count)

  File "<ipython-input-10-185b8ad2cde6>", line 2, in fehlerfunc
    return x/y

ZeroDivisionError: division by zero

# ausführliche Fehlermeldung
%xmode Verbose
avg_beelength(nobees)
# Ausgabe:
Exception reporting mode: Verbose
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-13-3d86e356f330> in <module>
      1 # ausführliche Fehlermeldung
      2 get_ipython().run_line_magic('xmode', 'Verbose')
----> 3 avg_beelength(nobees)
        global avg_beelength = <function avg_beelength at 0x0000000004FE1B88>
        global nobees = ()
```

```
<ipython-input-10-185b8ad2cde6> in avg_beelength(valuelist=())
      6     count = len(valuelist)
      7
----> 8     return fehlerfunc(total,count)
        global fehlerfunc = <function fehlerfunc at 0x000000000500BDC8>
        total = 0
        count = 0
      9
     10 bees = (3, 3.5, 4.1, 5.2, 5, 2)

<ipython-input-10-185b8ad2cde6> in fehlerfunc(x=0, y=0)
      1 def fehlerfunc(x,y):
----> 2     return x/y
        x = 0
        y = 0
      3
      4 def avg_beelength(valuelist):
      5     total = sum(valuelist)

ZeroDivisionError: division by zero
```

Listing 2.4 Fehlermeldungen mit unterschiedlichem Detailgrad

In allen Modi wird ein sogenannter *Stacktrace* bzw. *Traceback* angezeigt, der den Fehlerverlauf anzeigt. Er beginnt mit der Funktion auf der obersten Ebene und geht bis zum letzten Element, das den Fehler ausgelöst hat. In unserem Fall ist das `return x/y` in der Funktion `fehlerfunc()`.

Der Standardmodus `Context` liefert aber nicht nur die Funktionenfolge wie im Modus `Plain`, sondern auch die unmittelbare Umgebung im Programmcode. Damit findet man zum einen die Programmstellen leichter, zum anderen liegt die Fehlerursache oft in der unmittelbaren Umgebung.

Der Modus `Verbose` zeigt uns noch mehr, nämlich den Wert der lokalen Variablen. Es zeigt in den letzten Zeilen, dass die Werte für `x` und `y` gleich 0 sind. Damit ist uns die Ursache schon klar – wir haben durch die Zahl 0 dividiert (Division by Zero), was natürlich nicht geht.

Wird der Code allerdings komplexer, kann die Analyse der Fehlermeldung sehr schwierig werden, da der Traceback extrem lange werden kann.

Traceback (bzw. Stacktrace)

Der Traceback ist ein Werkzeug zur Rückverfolgung eines Programmfehlers. Ausgehend vom Auftreten des Fehlers werden alle daran beteiligten Funktionen angezeigt, was die Rückverfolgung bis zur Fehlerursache ermöglicht. Je komplexer und verschachtelter der Programmcode, desto länger werden solche Stacktraces.

Manchmal liefert der Code keinen Fehler, tut aber trotzdem nicht ganz das, was man eigentlich wollte. Um dann genauere Informationen zu erhalten, was alles bei der Ausführung des Programmcodes passiert, verwendet man einen *Debugger*. Im Jupyter Notebook wird der interaktive Debugger mit dem Befehl `%debug` gestartet.

Rufen wir nun diesen interaktiven Debugger auf, dann wird ein Kommandozeilenfenster geöffnet mit dem Prompt `ipdb>`. In dieser Kommandozeile können wir Befehle eingeben, die uns unter anderem die Inhalte unserer Variablen anzeigen. Somit können wir hoffentlich erkennen, wo der Fehler liegt. Listing 2.5 zeigt, wie der interne Debugger zu verwenden ist.

In unserem Traceback erkennen wir, dass wir zumindest zwei Stufen haben. Die letzte zeigt in der Funktion `fehlerfunc(x,y)`, wo genau der Fehler passiert ist. Hier haben wir Zugriff auf alle Variablen innerhalb der Funktion, also `x`, `y` und natürlich eventuelle globale Variablen. Wollen wir eine Stufe höher gehen, in diesem Fall in die Funktion `avg_beelength()`, so müssen wir innerhalb des Debuggers den Befehl `up` eingeben. Mit der Ausgabe der Variablen `count` und `total` auf dieser Stufe ergibt dies beide Male 0 und führt uns zum eigentlichen Fehler, nämlich der Eingabe einer leeren Liste.

```
%debug

# Ausgabe:
> <ipython-input-1-78d6430f450b>(2)fehlerfunc()
      1 def fehlerfunc(x,y):
----> 2     return x/y

ipdb> print(x, y)
0
ipdb> print(count)
*** NameError: name 'count' is not defined

ipdb> up
> <ipython-input-10-185b8ad2cde6>(8)avg_beelength()
      6     count = len(valuelist)
      7
```

```
----> 8     return fehlerfunc(total,count)
      9
     10 bees = (3, 3.5, 4.1, 5.2, 5, 2)

ipdb> print(total, count)
0 0
ipdb> quit
```

Listing 2.5 Die Verwendung des interaktiven Jupyter Debuggers

Natürlich könnte man wie immer Bücher über das Thema Fehleranalyse und Debugging schreiben. Für unsere Zwecke reichen die hier vorgestellten Werkzeuge.

2.1.3 Wichtige Python-Module

Im Folgenden werden wir in aller Kürze die für dieses Buch wichtigen Python-Module beschreiben. Für jedes dieser Module gibt es sogenannte *Cheat Sheets*, die auf einer DIN-A4-Seite die wichtigsten Funktionen und Klassen zu einem Modul beispielhaft zeigen. Sie finden sie unter *https://www.datacamp.com/community/data-science-cheatsheets*. Dennoch sei an dieser Stelle bereits gewarnt, dass wir im Verlauf des Buches Elemente aus Modulen verwenden, die wir nicht extra vorstellen oder genauer beschreiben. Diese Arbeitsweise ist durchaus typisch in Python-Projekten, da es fast zu jeder Aufgabe eine Bibliothek gibt, die einfach zu verwenden ist und uns eine Menge Arbeit, genauer gesagt Eigenprogrammierung, erspart.

NumPy

NumPy steht für *Numerical Python* und hat für das wissenschaftliche Rechnen eine große Bedeutung. Es enthält eine optimierte Datenstruktur für mehrdimensionale Matrizen und die dazugehörigen Funktionen der linearen Algebra.

Das Herzelement dieses Moduls ist die Klasse `ndarray` für mehrdimensionale Matrizen.

Sie finden NumPy unter *http://www.NumPy.org*.

matplotlib

Dieses Modul erlaubt die Erstellung von wissenschaftlichen Diagrammen und Visualisierungen. Es stellt Funktionen für Liniendiagramme, Histogramme, Streudiagramme zur Verfügung. Wird matplotlib mit Jupyter Notebook verwendet, so können die Visualisierungen direkt im Notebook dargestellt werden.

Sie finden matplotlib unter *http://www.matplotlib.org*.

scikit-learn

scikit-learn ist ein sehr beliebtes Werkzeug und enthält eine große Anzahl an modernen Machine-Learning-Algorithmen. Es ist gleichermaßen in der Industrie und im Universitätsbereich in Verwendung.

Sie finden scikit-learn unter *https://scikit-learn.org*.

pandas

Pandas erlaubt die Datenvorverarbeitung und Zusammenführung von Daten aus unterschiedlichen Quellen. Im Besonderen können Daten durch eine SQL-ähnliche Sprache mittels Abfragen und Zusammenführen von Tabellen zu den Strukturen `Series` (eindimensional), `DataFrames` (zweidimensional) und `Panels` (dreidimensional) eingelesen werden. Vor allem die `DataFrames` sind für uns von Bedeutung.

Sie finden pandas unter *https://pandas.pydata.org*.

TensorFlow

TensorFlow ist eigentlich ein ganzes Paket an Bibliotheken, Online-Ressourcen etc. das wir genauer im Anhang C beschreiben werden. Google ist es zu verdanken, dass wir damit für das Deep Learning entwickelte Machine-Learning-Bibliotheken zur Verfügung haben. Die Berechnungen erfolgen mit sogenannten *Datenflussgraphen* (*data flow graphs*). Ein Graph wiederum besteht aus Knoten, die durch Kanten verbunden sind. Jeder mit TensorFlow berechnete Algorithmus besteht aus mathematischen Operationen (repräsentiert als Knoten) und Daten (repräsentiert als Kanten). Damit lassen sich alle Berechnungen von einer einfachen Summe bis zu hochkomplexen Matrixoperationen durchführen und darstellen.

Im Buch verwenden wir die Version TensorFlow 2, die eine von Francois Chollet entwickelte Keras-Bibliothek zur einfacheren Handhabung von Deep Neural Nets integriert hat.

Sie finden das Framework unter *http://www.tensorflow.org*.

Keras

Keras ist eine eigenständige Bibliothek, die mit verschiedenen Neuronale-Netze-Modulen arbeiten kann, wie zum Beispiel TensorFlow, MXNet oder Theano. Diese Module werden als *Backends* bezeichnet. Mit dem Release TensorFlow 2 hat Google Keras vollständig integriert. Keras wird aber auch als eigenständige Bibliothek weitergeführt, obwohl auch Francois Chollet den Benutzern vorschlägt, in Zukunft nur mit TensorFlow 2 zu arbeiten.

Sie finden die Bibliothek und ihre Dokumentation hier:

- *http://www.keras.io*
- *https://www.tensorflow.org/guide/keras*

2.1.4 Jupyter-Notebook-Cloud-Ressourcen

Die folgende Liste ist ein guter Startpunkt, aber aus mehreren Gründen unvollständig. Zum einen gibt es eine fast unüberschaubare Anzahl an Anbietern, zum anderen bieten die angegebenen Webseiten viel mehr als nur Jupyter Notebooks an. Wir haben einige ausgewählt, die bis zu einem gewissen Grad die Ressourcen gratis zur Verfügung stellen.

Jupyter Community

Zuerst muss natürlich die Webseite der Jupyter Community erwähnt werden, die Jupyter-Ressourcen zur Verfügung stellt – allerdings ohne GPU-Unterstützung und eigentlich nur zum Ausprobieren.

Links:

- *https://jupyter.org/try*

Google Colaboratory (Colab)

Hier bietet Google Jupyter Notebooks mit GPU/TPU-Ressourcen an, die auch mit dem eigenen Google Drive verbunden werden können.

Links:

- *https://colab.research.google.com/*

Kaggle

Kaggle ist eine Data-Science-Plattform, die sich ursprünglich auf Machine Learning Competitions spezialisiert hat und ist eigentlich für jeden angehenden Data-Science-Experten ein Muss.

Links:

- *https://www.kaggle.com/notebooks*

Microsoft Azure

Auch Microsoft ist mit Azure sehr stark in der Machine-Learning-Szene vertreten.

Links:

- *https://notebooks.azure.com*

Amazon SageMaker

Wie viele Cloudanbieter hat auch Amazon mit Amazon Web Services (AWS) Jupyter Notebooks im Portfolio.

- *https://aws.amazon.com/de/sagemaker*

2.2 Zusammenfassung

Dieses Kapitel führt in den Anaconda Navigator und Jupyter Notebook ein, die zusammen eine komfortable technische Entwicklungsumgebung für die Programmiersprache Python darstellen. Um eine kleine Einführung in Python zu erhalten, können Sie jetzt zu Anhang A, »Python kompakt«, vorblättern.

Python bietet eine Unmenge an Bibliotheken. Die wichtigsten für die Programmierung von neuronalen Netzen haben wir hier kurz beschrieben, mit Verweisen auf sehr gute Tutorials. Eine Liste mit Anbietern von Jupyter Notebooks in der Cloud vervollständigt dieses Kapitel.

Kapitel 3

Ein einfaches neuronales Netz

Ein erster Schritt: ein einfaches Netz, das die Einordnung von »Dingen« in zwei Kategorien ermöglicht.

Dieses Kapitel beschreibt die einfachste Form neuronaler Netze, die sogenannten *Perceptrons*, auch als *Lineare Klassifizierer* bezeichnet. Wir werden Schritt für Schritt mit Bildern und Codes die Funktionsweise der Perceptrons erarbeiten. Einfache Beispiele erläutern die Materie, und am Ende können Sie einem fahrenden Roboter beibringen, Löcher zu erkennen oder zu vermeiden, in die Wand zu fahren.

3.1 Vorgeschichte

Stellen Sie sich vor, Sie müssen keine Planung mehr machen, weil diese ein neuronales Netz für Sie übernimmt. Lassen Sie uns diese Art von Planung als »automatische Planung« bezeichnen. Das Szenario sieht folgendermaßen aus: Personen sollen nach individuellen Terminwünschen und unternehmerischen Bedarfen so geplant werden, dass der unternehmerische Zweck mithilfe ebendieser Personen umgesetzt werden kann. Beispiele dafür sind die *Personaleinsatzplanung* eines Tante-Emma-Ladens oder der Stundenplan in einer Schule.

3.2 Her mit dem neuronalen Netz!

Eine Frage vorweg: Wie würden Sie die zwei Gruppen in der folgenden Zeichnung voneinander trennen, wenn Sie einen Stift und ein Lineal verwenden dürften? Wenn Sie das geschafft haben – finden Sie noch weitere Möglichkeiten?

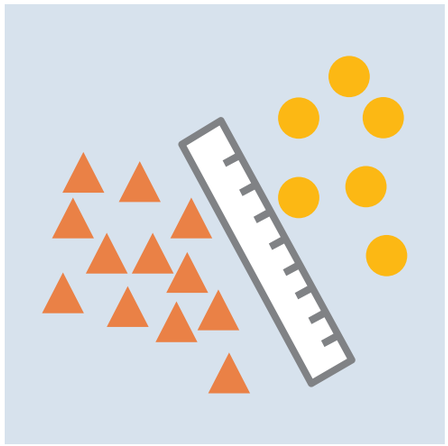


Abbildung 3.1 Linea(l/r)e Trennung

Für uns Menschen ist es denkbar einfach, diese Aufgabe zu lösen. Wir sehen sofort, dass zwei Gruppen vorhanden sind, die wir einfach durch eine *Gerade* trennen können. Doch wie kann dieser Vorgang maschinell durchgeführt werden, welche Ansätze stehen dafür zur Verfügung? Die sperrige Antwort lautet je nach Fachgruppe:

- Computerwissenschaftler: durch einen *Klassifikator*
- Statistiker: durch einen *linearen Diskriminator*
- Data Scientist: durch ein *Perceptron*, ein spezielles KNN
- [Wählen Sie bitte Ihre Fachgruppe]: [Wählen Sie bitte Ihren Ansatz]

Wir werden natürlich die Linie des Data Scientist weiterverfolgen. Aber was ist mit Perceptron gemeint? Diese Frage wird uns bis zum Ende dieses Kapitels beschäftigen, und zwar mit folgendem Beispiel:

Die Filialleiterin – Frau Äppel – eines kleinen Ladens hat zwei Mitarbeiter zu planen, Herrn Lauch und Frau Karotte. Beide haben von Montag bis Freitag prinzipiell Zeit zum Arbeiten. Samstag und natürlich Sonntag ist der Laden zu. Frau Äppel hätte gerne jeden Tag **mindestens einen** Mitarbeiter zur Seite, also zum Beispiel am Montag

- Frau Karotte allein
- oder Herrn Lauch allein
- oder beide gemeinsam, aber sicher mindestens einen von beiden

Das klingt jetzt ein wenig kompliziert, wenn man das so hört. Einfacher ist es zu verstehen, wenn wir das in Form von Tabelle 3.1 aufschreiben:

Frau Karotte	Herr Lauch	→	Montag
nicht anwesend	nicht anwesend	→	**nicht ok**
anwesend	nicht anwesend	→	**ok**
nicht anwesend	anwesend	→	**ok**
anwesend	anwesend	→	**ok**

Tabelle 3.1 Erster Personalplan

Wenn Frau Karotte und Herr Lauch nicht anwesend sind, dann ist der Wunsch von Frau Äppel nicht erfüllt. Es ist keiner da zum Helfen, und nach so einem Tag ist Frau Äppel »nicht ok«, da sie alles allein machen muss. Besser ist es, wenn zum Beispiel Frau Karotte anwesend ist, denn das ist auf alle Fälle »ok« für Frau Äppel.

Mit dieser Tabellendarstellung sind die möglichen Kombinationen schon viel einfacher zu verstehen. Aber optimal ist das immer noch nicht. Ein KNN kann mit den Begriffen »nicht anwesend«, »anwesend«, »nicht ok« und »ok« nicht rechnen, und daher müssen wir uns eine andere Darstellung mit Zahlen überlegen.

Anmerkung am Rande: Datenaufbereitung

Dieser Schritt der Datenaufbereitung ist einer der schwierigsten Schritte im Gesamtprozess des Arbeitens mit KNN. Woher sollte man bitte um Himmels willen wissen, in welches Format die Daten für das KNN zu transformieren sind? Woher sollte man wissen, was aus anwesend/nicht anwesend werden sollte? Im zweiten Teil werden wir uns dieses Thema noch genauer vornehmen.

Ohne jetzt großartig eine Theorie zum Datenvorverarbeiten für das Arbeiten mit KNN zu entwickeln, stellen wir die Frage aber trotzdem, wie man Text so wie »ok« oder »nicht ok« als KNN-Input passend aufbereitet: Haben Sie dazu eine Idee? Wie könnte die Vorverarbeitung aus Ihrer Sicht aussehen? Bitte kurz nachdenken.

Danke für das Nachdenken, wir haben uns dazu folgenden Vorschlag überlegt: Wenn ein Mitarbeiter *anwesend* ist, dann schreiben wir »1«, und wenn der Mitarbeiter *nicht anwesend* ist, »0«. Diese Festlegung scheint möglicherweise willkürlich, und in der Tat ist sie das auch. Jedoch ist der Wert 1, wenn etwas »an« oder »gut« ist, passend für das positive Ereignis, und 0 für »aus« oder »schlecht« auch intuitiv nachvollziehbar. Daher passt diese Zuordnung in diesem Kontext aus unserer Sicht ganz gut. Andererseits ersetzen wir »nicht ok« durch »0« und »ok« durch »1«, um darzustellen, wie das gewünschte Ergebnis der Planung aussehen sollte.

Wenn wir unsere Überlegungen wieder im Tabellenformat darstellen, bekommen wir die Inhalte in Tabelle 3.2:

Frau Karotte	Herr Lauch	→	Montag
0	0	→	0
1	0	→	1
0	1	→	1
1	1	→	1

Tabelle 3.2 Zweiter Personalplan mit KNN-geeigneter Codierung

Damit können wir nun rechnen. Moment einmal, wieso rechnen? Wir wollen einem KNN beibringen, dass es **nicht okay** ist, wenn **kein Mitarbeiter** da ist, und es in allen anderen Anwesenheitskonstellationen immer **okay** ist. Das heißt, dass wir uns eine Berechnung überlegen müssen, um das gewünschte Ergebnis zu bekommen. Für das Erste ohne ein KNN, also so richtig mit Nachdenken und Selberrechnen. Das wird superwichtig für das Verständnis der eigentlich sehr einfachen Berechnungen, die in einem KNN durchgeführt werden.

Der erste Versuch könnte sein, dass Frau Karotte und Herr Lauch zusammengezählt werden – es werden nicht wirklich die Personen summiert, sondern die Anwesenheit der beiden. Das Ergebnis wird dann von dieser Summe mit einer einfachen Regel abgeleitet, die da lautet: »Wenn die Summe größer gleich 1 ist, dann ist das Ergebnis 1, ansonsten ist das Ergebnis 0.« Das Ganze nun wieder in Form einer Tabelle:

Frau Karotte	Berechnung	Herr Lauch	→	Summe	Ergebnis
0	+	0	=	0	0
1	+	0	=	1	1
0	+	1	=	1	1
1	+	1	=	2	1

Tabelle 3.3 Errechneter Personalplan aus den Anwesenheiten

Diese Überlegungen könnten wir natürlich in Form eines Netzes darstellen, so wie in Abbildung 3.2 zu sehen, um eine andere Sicht als die der Tabellendarstellung zu erhalten. Wir wollen uns ja schrittweise der KNN-Visualisierung nähern.

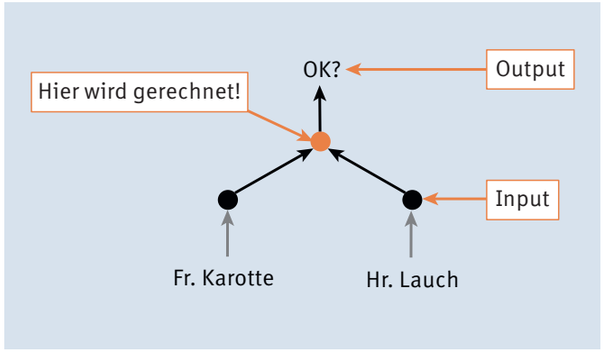


Abbildung 3.2 Die Entscheidung für Frau Karotte und Herrn Lauch berechnen

Diese Art der Darstellung ist nicht mehr weit weg von der Darstellung eines KNN. Wir müssen noch ein paar Details ergänzen, aber im Prinzip sind wir schon fertig – fast.

3.3 Neuron-Zoom-in

Eines der Details, das noch zu ergänzen ist, betrifft die Art der Berechnung. Sehen wir uns das Neuron genauer an.

Aus Tabelle 3.3 wissen wir, dass wir eine Berechnung und dann eine Entscheidung benötigen. Diese Erkenntnis ergänzen wir in der Darstellung des Berechnungsknotens in Abbildung 3.3.

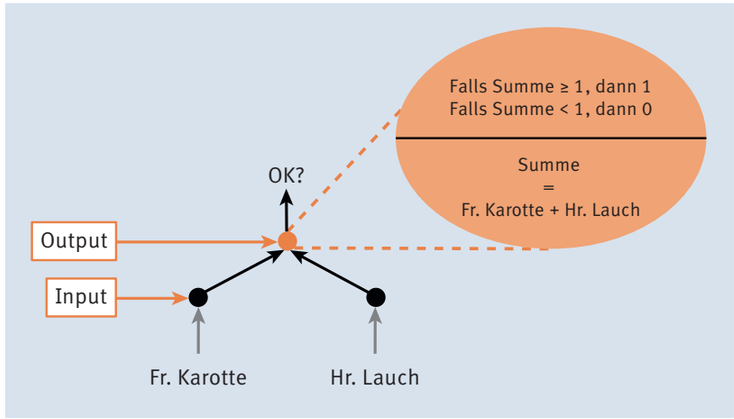


Abbildung 3.3 Ergebnisberechnung für Frau Karotte und Herrn Lauch im Detail

Um die Details zum Berechnungsknoten in Form zu bringen, sodass ein Python-Programm damit umgehen kann, benötigen wir eine Formel bzw. eine Rechenvorschrift, und die könnte so aussehen:

$$Summe = Fr.Karotte + Hr.Lauch$$

und

$$Ergebnis = Entscheidung(Summe)$$

wobei

$$Entscheidung = \begin{cases} 1, & \text{falls } Summe \geq 1 \\ 0, & \text{falls } Summe < 1 \end{cases}$$

Noch ein Wort zu der Zeile *Ergebnis* = *Entscheidung* (*Summe*): Wir haben die sogenannte *Funktionsschreibweise* für die Berechnung der Entscheidung verwendet. Diese besagt nur, dass das Ergebnis der Funktion *Entscheidung* von der *Summe* abhängt. Oder anders gesagt, dass das Ergebnis der Funktion *Entscheidung* in Abhängigkeit von der *Summe* ermittelt wird. Sehr oft werden die Funktionen mit $f(x)$ bezeichnet, wie zum Beispiel $f(x) = x^2$. Das besagt nur, dass die Funktion f in Abhängigkeit von x definiert ist.

Die Funktionsschreibweise kann direkt in die Python-Programmierung übernommen werden. Wir machen das nun und definieren uns eine Funktion mit der eben beschriebenen Funktionalität. Falls Sie Ihr Jupyter Notebook noch nicht gestartet haben, dann würden wir Sie bitten, dass Sie das nun erledigen. Sie können auch gerne nochmals zu Kapitel 2, »Das minimale Starterkit für die Entwicklung von neuronalen Netzen mit Python«, blättern und die Details nachlesen. Wir werden die Programmbeispiele in unterschiedlichen Notebooks organisieren, wobei wir pro Kapitel ein Notebook vorgesehen haben. Ihr nächster Schritt besteht somit darin, dass Sie sich ein neues Notebook anlegen (Abbildung 3.4).

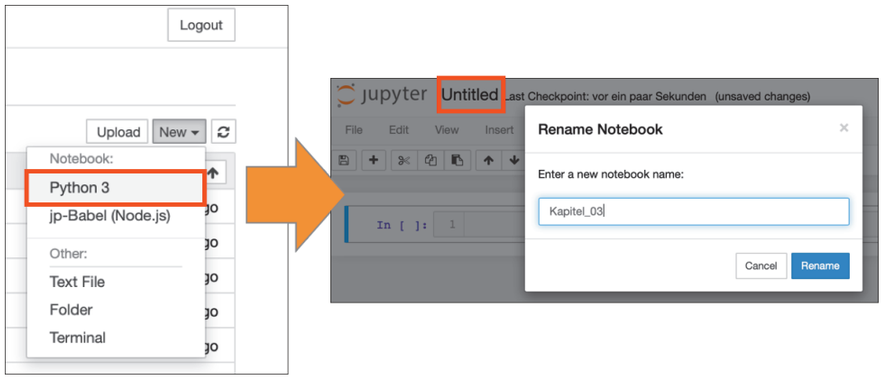


Abbildung 3.4 Das Notebook »Kapitel_03« anlegen

Nutzen Sie dazu die NEW-Drucktaste, und im nun angezeigten Menü wählen Sie den Eintrag PYTHON 3. Daraufhin erzeugt Jupyter das neue Notebook, und Sie können mittels Mausklick auf den Text UNTITLED einen Namen für das Notebook vergeben, zum Beispiel »Kapitel_03«.

Wenn Sie die Drucktaste RENAME anklicken, dann wird das Notebook in »Kapitel_03« umbenannt, und Sie können mit dem Programmieren beginnen. Wie werden zwei Zellen verwenden; die erste Zelle enthält die Überschrift und die zweite Zelle den Code.

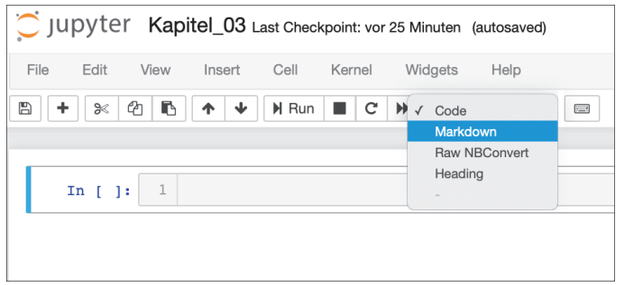


Abbildung 3.5 Eine Zelle mit »Markdown« markieren

Wir werden die Zelle 1 als MARKDOWN-Zelle attribuieren (siehe Abbildung 3.5), um dem Code in der folgenden Zelle eine Überschrift zu spendieren. Klicken Sie dazu in Zelle 1, und wählen Sie im Menü CELL die Menüpunkte CELL TYPE • MARKDOWN oder im CODE-Dropdown den Wert MARKDOWN.

Nachdem dies geschehen ist, können Sie die Auszeichnungselemente für eine Überschrift in der Zelle eingeben, wie zum Beispiel ein »#« für eine Überschrift der Ebene 1 und, durch ein Leerzeichen getrennt, den Text für die Überschrift (Abbildung 3.6).

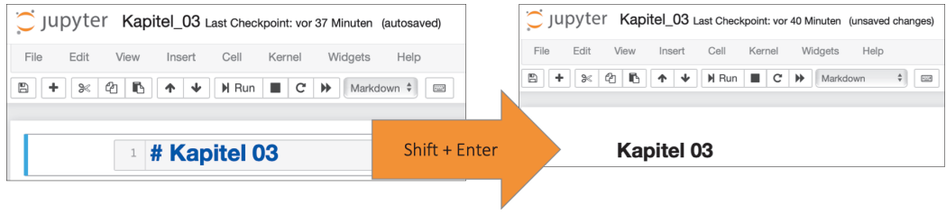


Abbildung 3.6 Notebook mit Überschrift

Wenn Sie Ihre Eingabe mit der Tastenkombination ⇧ + ↵ bestätigen, wird der Text in der Markdown-Zelle formatiert dargestellt.

Immerwiederkehrender Header

Wir haben in den Notebooks zum Herunterladen noch zwei Aspekte eingefügt, die in allen Notebooks vorkommen.

- Einen Hinweis, dass Sie beim erstmaligen Start den Menüpunkt CELL • RUN ALL ausführen sollten. Dies empfehlen wir, da damit gewährleistet ist, dass die Scripts in allen Code-Zellen der Reihe nach ausgeführt und die besprochenen Outputs erzeugt werden.
- Eine MARKDOWN-Zelle und eine CODE-Zelle, die eine etwaige Warnung zum Beispiel hinsichtlich zukünftig deprecated (dt. veralteter) Anweisungen unterdrücken. Wenn Sie die Warnung sehen wollen, dann kommentieren Sie einfach den Code in der Zelle.

Durch die Bestätigung mit der Tastenkombination ⇧ + ↵ wird automatisch eine neue Code-Zelle eingefügt, in der Sie nun weitere Markdowns bzw. Ihren Code platzieren können, wie zum Beispiel den in Listing 3.1.

```
def entscheidung( summe ):
    """ Berechnung der Entscheidung zum Wert summe
    Input:  summe
    Output: 1, falls summe >= 1,
            0 sonst
    """
    if summe >= 1:
        return 1
    else:
        return 0
#-----------------------------
# Berechnung der entscheidung
ergebnis = entscheidung(1)
# Ausgabe in Zelle als String
print('Input: ', 1)
print('Output:', ergebnis)
# Ausgabe:
Input:  1
Output: 1
```

Listing 3.1 Die Entscheidung als Stufenfunktion

Die vorher besprochene Berechnung der Entscheidung bilden wir als Funktion `entscheidung` mit dem Parameter `summe` ab. Danach folgt im Code ein Funktions-*Docstring*, der die Funktion dokumentiert, so wie im Kapitel 2 beschrieben. Sehen Sie sich dazu bitte die Abbildung 3.7 an.

Kapitel 03

ACHTUNG: Bitte zum Starten im Menü `Cell • Run All` ausführen.

Deaktivieren der Warnungen

In [0]:
```
import warnings
warnings.filterwarnings('ignore')
```

Die Entscheidung

Listing 3.1, Abbildung 3.7

In [2]:
```
def entscheidung( summe ):
    """ Berechnung der Entscheidung zum Wert summe
    Input:  summe
    Output: 1, falls summe >= 1,
            0 sonst
    """
    if summe >= 1:
        return 1
    else:
        return 0
#-----------------------------
# Berechnung der entscheidung
ergebnis = entscheidung(1)
# Ausgabe in Zelle
print('Input:  ' + str(1))
print('Output: ' + str(ergebnis))
```

```
Input:  1
Output: 1
```

Dokumentationsstring

In [3]:
```
# Ausgabe des Docstring mittels help-Funktion
print(entscheidung.__doc__)
```

```
 Berechnung der Entscheidung zum Wert summe
    Input:  summe
    Output: 1, falls summe >= 1,
            0 sonst
```

Abbildung 3.7 Programm und Ausgaben für die Funktion »entscheidung«

Nach dem Docstring folgt der Code für die Berechnung der Entscheidung. Falls der übergebene Wert größer als oder gleich 1 ist, dann wird als Ergebnis der Berechnung 1 zurückgegeben, ansonsten 0.

Wir haben direkt nach der Definition den Aufruf der Funktion eingefügt und mit dem Wert 1 getestet. Der Rückgabewert wird in der Variable `ergebnis` gespeichert und am

Ende mit dem `print`-Befehl nach der Code-Zelle ausgegeben. Die Ausgaben der Programme werden wir hinter dem Code platzieren und fett markieren. Das Ergebnis in Ihrem Jupyter Notebook sollte dann so ähnlich aussehen wie in Abbildung 3.7.

Wir hoffen, dass Sie mit diesen ausführlich beschriebenen Schritten gut gewappnet in die nächste Programmierung gehen, in der wir kurz die Anwendung des Moduls `matplotlib` besprechen werden. Dieses Modul kann, wie bereits erwähnt, für die Erstellung von Diagrammen verwendet werden. Naheliegenderweise werden wir den Output der Funktion `entscheidung` damit visualisieren.

3.4 Stufenfunktion

In der Funktion `entscheidung` haben wir unterschieden, ob die Summe kleiner als 1 ist oder nicht (dann ist sie größer oder gleich 1). Der Wert 1 wird als *Schwellenwert* bezeichnet, da wie bei einer Türschwelle ein Höhensprung – oder Tiefensprung, falls ich von oben komme – eingebaut ist. Schlagartig ändert sich der berechnete Wert der Funktion von 0 in 1.

Die Mathematiker haben sich für solch eine Funktion die Bezeichnung *Stufenfunktion* überlegt, im Englischen findet man die Bezeichnungen *step function* dafür. Dabei kann der Schwellenwert einen beliebigen Wert annehmen. Falls der Schwellenwert 0 ist, wird die Stufenfunktion als *Heaviside-Funktion* bezeichnet, benannt nach dem Mathematiker *Oliver Heaviside*.

Wir verwenden für das einfache KNN, das wir gleich bauen werden, genau diese Stufenfunktion. Die Python-Funktion `entscheidung` dafür haben wir ja bereits programmiert. Bevor wir uns dem KNN zuwenden, werden wir aber noch ein weiteres Python-Script programmieren, um das Erstellen von Grafiken zu üben. Das Python-Modul `matplotlib` eignet sich hervorragend dafür, da es Funktionen für Liniendiagramme, Histogramme, Streudiagramme usw. zur Verfügung stellt. Außerdem haben wir als Jupyter-Notebook-Verwender den Vorteil, dass die Visualisierungen der Diagramme direkt im Notebook dargestellt werden.

Legen Sie für das neue Script wieder eine Markdown-Zelle mit der Überschrift »Stufenfunktion« an, so wie Sie es bereits vorher gemacht haben, und zusätzlich eine Code-Zelle für das folgende Script in Listing 3.2.

Noch eine Anmerkung, bevor es losgeht: Wir werden die Funktion `entscheidung` in diesem Script wiederverwenden. Damit dies gelingen kann, ist es nötig, dass Sie die Zelle mit der Funktion `entscheidung` ausführen. Erst dann kann Ihr Script in einer Code-Zelle eine Funktion in einer anderen Code-Zelle aufrufen.


```
# Import der Module für Plots
import matplotlib.pyplot as plt
# Ganz wichtig, sonst wird der Plot nicht angezeigt
%matplotlib inline

def entscheidung( summe ):
    """ Berechnung der Entscheidung zum Wert summe
    Input:  summe
    Output: 1, falls summe >= 1,
            0 sonst
    """
    if summe >= 1:
        return 1
    else:
        return 0
#------------------------------
# x-Werte des Graphen
x = [-1,0,0.999,1,2]
# y-Werte mit der Funktion entscheidung berechnen und mithilfe
# einer List Comprehension eine neue Liste erzeugen (siehe Anhang A)
y = [ entscheidung(i) for i in x ]
# Erzeugen des Graphen mit einer orangefarbigen Stufe und der Bezeichnung step
plt.step(x, y, color='Orange', label='step')

# Die Achsen setzen
plt.grid(True)
# Die horizontale und vertikale 0-Achse etwas dicker in Schwarz zeichnen
plt.axhline(0, color='black', lw=1)
plt.axvline(0, color='black', lw=1)

# Achsenbeschriftung und Titel
plt.xlabel('Summe')
plt.ylabel('Ergebnis')
plt.title('Stufenfunktion')

# Legendenplatzierung festlegen
plt.legend(loc='center right')

# Den Graphen anzeigen
plt.show()
```

Listing 3.2 Verwendung von »pyplot«

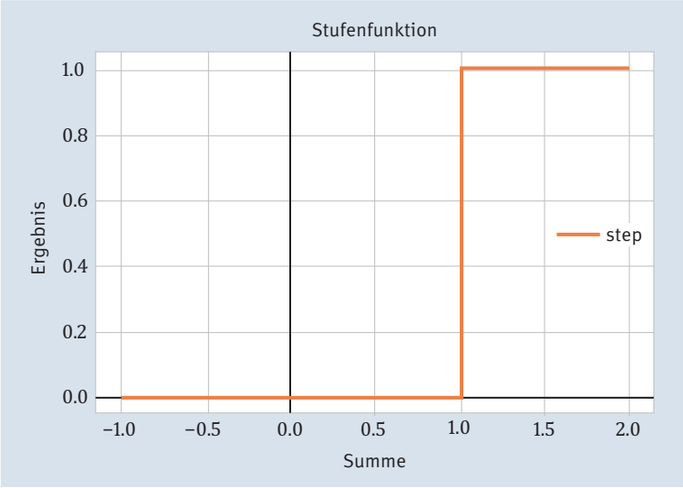


Abbildung 3.8 Stufenfunktion

Zuerst ist es natürlich nötig, `matplotlib.pyplot` zu importieren und einen passenden Namen im lokalen Namensraum zu definieren, `plt` als Name hat sich dafür eingebürgert. Die Anweisung `%matplotlib inline` ist eine sogenannte *Magic Function* und mit ihr wird festgelegt, dass der Output der `plot`-Anweisung inline direkt unter der produzierenden Zelle im Notebook durchgeführt wird. Mehr zu den Magic Functions finden Sie im Kapitel 2. Danach werden die Daten in Listen zusammengestellt, wir verwenden eine Liste mit `x`- und eine mit `y`-Werten. Die Werte der `y`-Liste werden durch die Funktion `entscheidung` berechnet, indem für alle Elemente der `x`-Liste die Funktion aufgerufen wird. Die Funktion `plt.step` erzeugt die Stufe, wobei zum Beispiel die Farbe zum Zeichnen angegeben werden kann. Dazwischen folgen einige Detailbearbeitung der `figure`, und schlussendlich wird die `figure` mit `plt.show()` angezeigt.

3.5 Perceptron

Durch unsere Vorüberlegungen wird es jetzt sehr einfach, auf das sogenannte *Perceptron* zuzusteuern. Das Perceptron ist eines der ältesten und einfachsten KNN-Modelle. Es entstand im Jahre 1957 und wurde von Frank Rosenblatt erfunden. Die Knoten im Perceptron werden als *Linear Threshold Unit* (LTU) bezeichnet, weil es als Ausgabefunktion die Stufenfunktion verwendet und damit eine lineare Trennung der Eingabedaten durchführt. Zur besseren Veranschaulichung haben wir eine Schautafel in Abbildung 3.9 mit allen Bestandteilen des Perceptrons und der Berechnungsbausteine zusammen-

gestellt – wie ein kleiner Experimentierkasten, aus dem wir nun nach und nach Teile herausholen und besprechen werden.

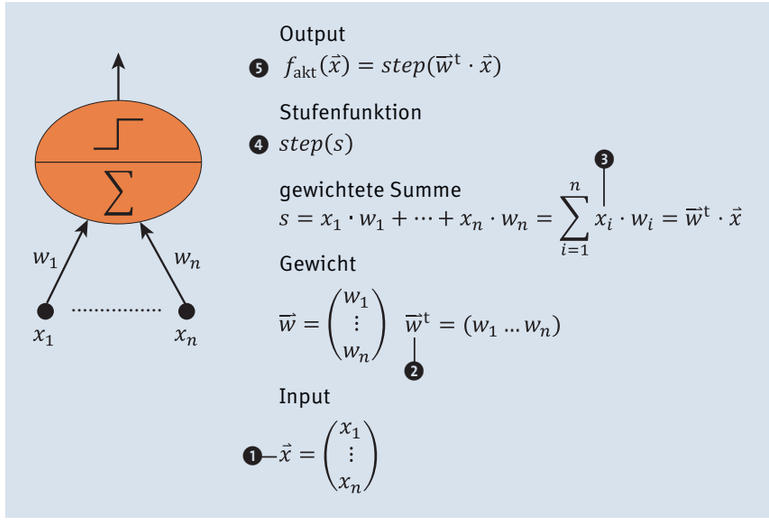


Abbildung 3.9 Perceptron – Bestandteile und Berechnungsbausteine

Da steht einem möglicherweise der Schweiß auf der Stirne, wenn man diese ganzen mathematischen Symbole sieht. Aus unserer Sicht verschleiert die kompakte und elegante mathematische Notation die Einfachheit des Modells. Wir haben damit nämlich aufgeschrieben, was wir bis jetzt bei der Planung von Frau Karotte und Herrn Lauch gemacht haben – nur allgemeiner. Um Ihnen das zu beweisen, gehen wir die Bausteine aus der Schautafel in den nächsten Abschnitten Punkt für Punkt durch:

❶ Baustein, Input x_1, in Abschnitt 3.6, »Punkte im Raum – Vektorrepräsentation«
❷ Baustein, $\vec{w}^t$, in Abschnitt 3.7, »Horizontal und vertikal – Spalten- und Zeilenschreibweise«
❸ Baustein, $\sum_{i=1}^{n} x_i \cdot w_i$, in Abschnitt 3.8, »Die gewichtete Summe«
❹ Baustein, $step(s)$, in Abschnitt 3.9, »Schritt für Schritt – Stufenfunktionen«
❺ Baustein, den Output, in Abschnitt 3.10, »Die gewichtete Summe reloaded«

3.6 Punkte im Raum – Vektorrepräsentation

Los geht's: Der Input x_1 steht für die **Anwesenheit** von Frau Karotte; er kann die Werte 0 – da ist sie leider nicht anwesend – oder 1 – sie kann glücklicherweise mithelfen – an-

nehmen. Der Input x_2 steht für die Anwesenheit von Herrn Lauch und kann natürlich mit denselben Werten belegt werden. Wenn zum Beispiel Frau Karotte anwesend ist und Herr Lauch noch Urlaub hat, könnten wir das so schreiben:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Die Schreibweise für die kombinierten Werte nennt man übrigens *Vektorschreibweise*. Die Anzahl der Werte nennt man die *Dimension*; in unserem Fall wäre die *Dimension* = 2, dies wird auch *zweidimensional* genannt. Der Punkt liegt also in einer zweidimensionalen Ebene, wenn man von oben aus der dritten Dimension hinuntersieht[1]. Punkte lassen sich auch in Diagrammen darstellen, wie das vorherige Beispiel in Abbildung 3.10.

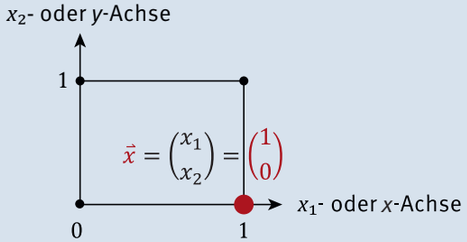


Abbildung 3.10 Punkt im kartesischen Koordinatensystem

Wir zeichnen die Werte in einem sogenannten *Koordinatensystem* ein. (Super wäre jetzt kariertes Papier, da sind schon so schöne Kästchen für das Einzeichnen der Punkte vorgedruckt.) Eine Dimension ist die x_1- oder einfach x-Dimension, wobei die möglichen Werte dieser Dimension im Koordinatensystem auf der x-Achse dargestellt werden. Und die zweite Achse, die x_2- oder auch y-Achse, wird darauf im rechten Winkel gezeichnet. Mit diesen zwei Angaben, x_1 und x_2, ist die Lage eines Punktes eindeutig festgelegt.

3.6.1 Aufgabe: Werte vervollständigen

Nun sind Sie wieder an der Reihe: Zeichnen Sie die anderen möglichen Punkte für unser Planungsbeispiel mit Herrn Lauch und Frau Karotte im Koordinatensystem ein, und beschriften Sie die Punkte mithilfe der Vektorenschreibweise, so wie vorher besprochen.

1 Eine schöne Geschichte zu den Dimensionen ist »Flatland. A Romance of Many Dimensions«, 1884, von Edwin Abbott unter dem Pseudonym A. Square.

Lösung:

Es ergibt sich folgende Abbildung

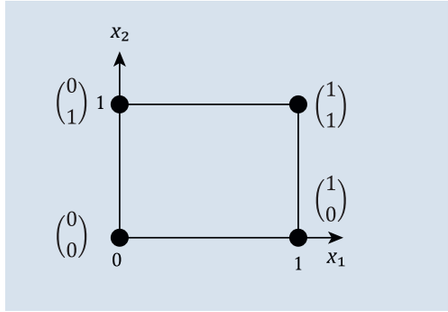


Abbildung 3.11 Alle Punkte zum Planungsbeispiel in Vektorenschreibweise

Als Belohnung für den erfolgreichen Abschluss der Übung schreiben Sie ein kleines Python-Script, das für Sie das Zeichnen der Punkte übernimmt. Dort sehen Sie, wie in Python ein Vektor deklariert wird, und zusätzlich setzen Sie zur Visualisierung einen *Scatter-Plot* (deutsch: *Streudiagramm*) ein. Ein *Scatter-Plot* stellt Wertepärchen als Punkte in einem Diagramm im kartesischen Koordinatensystem dar.

```
# Mathematik
import numpy as np
# Import der pyplot-Funktionen
import matplotlib.pyplot as plt
# Ganz wichtig, sonst wird der Plot nicht angezeigt
%matplotlib inline

# Die Funktion array wandelt eine Python-Liste in ein numpy-Array um
# Die x1 = x-Koordinaten
x1    = np.array([0, 0, 1, 1])
# Die x2 = y-Koordinaten
x2    = np.array([0, 1, 0, 1])
# Die Farben für die Punkte
color = np.array(['black', 'black', 'red', 'black'])
# Die Punktgröße für jeden Punkt
size  = np.array([100, 100, 500, 100])

# Den Vektor x1 mit allen x1-Koordinaten ausgeben
print('Der Vektor x1: ', x1)
```

```
# Die Achsen setzen
plt.grid(True)

# Den plot zeichnen für die x1- und x2-Koordinaten, Farbe und Größe
plt.scatter(x1, x2, c=color, s=size)

# Die Achsen setzen
plt.xlabel('x1')
plt.ylabel('x2')

# Die Achseneinteilungen setzen
plt.xticks([0.0,1.0])
plt.yticks([0.0,1.0])

# Endlich die figure ausgeben
plt.show()
# Ausgabe:
Der Vektor x1:  [0 0 1 1]
```

Listing 3.3 Scatter-Plot

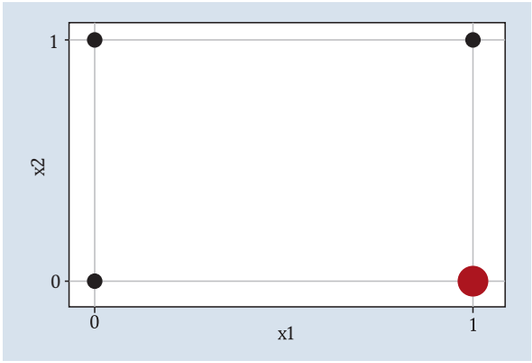


Abbildung 3.12 Scatter-Plot für die Planungspunkte

Wir starten im Script wieder mit dem Import des `matplotlib.pyplot`-Moduls und vergeben dafür den Namen `plt`. Danach importieren wir das Modul `numpy` für mathematische Operationen, speziell das Rechnen mit Vektoren und anderen mathematischen Konstrukten, die wir später noch besprechen werden. Danach folgt ein erklärungsbedürftiger Abschnitt im Programm. In den vorherigen Erläuterungen haben wir Punkte mit einem Vektor dargestellt, der aus den zwei Dimensionen x_1 und x_2 besteht. Der Einsatz

der `scatter`-Funktion erfordert, dass die Koordinaten der einzelnen Vektoren in zwei `np.array`-Objekte aufgeteilt werden müssen, also pro Dimension ein Array und pro Punkt ein Eintrag im Array. Zur Kontrolle geben wir das Array für die x_1-Dimension mit der `print`-Funktion aus.

Im weiteren Verlauf setzen wir für den Plot die gewünschten Eigenschaften, wie zum Beispiel die Achsenbeschriftungen mit `plt.xlabel('x1')`, `plt.ylabel('x2')` und Achseneinteilungen `plt.xticks()` und `plt.yticks()`. Am Ende des Scripts wird der Plot mittels `plt.show()` ausgegeben und gezeichnet.

3.6.2 Aufgabe: Iris-Datensatz als Scatter-Plot ausgeben

Die folgende Aufgabe dient dazu, Ihnen zu zeigen, wie Sie Daten aus einer Datei in Python importieren, zeilenweise verarbeiten und für die Visualisierung verwenden können. Die ersten Schritte, die das Einlesen betreffen, müssen wir natürlich detailliert erläutern, da wir dieses Thema noch nicht besprochen haben.

Als Datenmaterial für den Plot haben wir den berühmten *Iris-Datensatz* auserkoren, der einen typischen Testfall für Klassifikationstechniken darstellt. In diesem Datensatz befindet sich eine Menge von 150 Beobachtungen von je vier Eigenschaften von Schwertlilien. Die vier Eigenschaften sind die Breite und die Länge des Kelchblatts (Sepalum) sowie die des Kronblatts (Petalum), gemessen in Zentimeter. Weiterhin ist zu jeder vorhandenen Eigenschaftskombination die Art von Schwertlilie angegeben, das sind Iris-setosa, Iris-virginica oder Iris-versicolor.

Sepal Length	Sepal Width	Petal Length	Petal Width	Class
5.1	3.5	1.4	0.2	Iris-setosa
7.0	3.2	3.5	1.0	Iris-versicolor
6.3	3.3	6.0	2.5	Iris-virginica
...	...	...	...	...

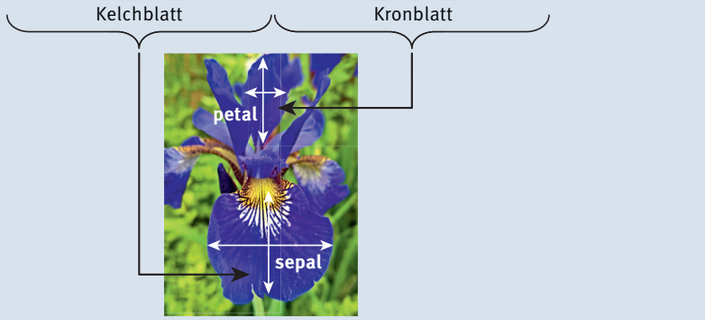


Abbildung 3.13 Blattmaße für Schwertlilien (© Kaggle)

Ihre Aufgabe besteht nun darin, einen *Scatter-Plot* mit den Python-Mitteln, die wir bisher besprochen haben, zu zeichnen. Natürlich müssen wir Ihnen noch verraten, wo Sie den Datensatz für die Schwertlilien beziehen können. Besuchen Sie die Webseite zum Buch, und laden Sie von dort den Datensatz *iris.csv* herunter. Als Ablageort der Datei wählen Sie das Verzeichnis, von dem aus das Jupyter Notebook gestartet wurde, damit die Datei dann ohne Pfadangabe eingelesen werden kann.

1. Lesen Sie das `iris.csv`-File ein. Verwenden Sie dazu die Anweisung `with open("iris.csv", "r") as fobj:`, die Ihnen eine File-Handle zum Auslesen (Parameter `"r"`) zurückgibt. Mit der Anweisung `for line in fobj:` können Sie den Datensatz Zeile für Zeile abarbeiten, und nachdem Sie alle Zeilen verarbeitet haben, wird die File-Handle `fobj` automatisch geschlossen.

   Erzeugen Sie drei Python-Listen, `x1`, `x2` und `colors`, um die Daten für Sepal Length, Sepal Width und die Farben für die Datenpunkte aufzunehmen. Um diese Daten ermitteln zu können, benötigen Sie noch den Spaltenaufbau des Datensatzes pro Zeile, den Sie in Abbildung 3.13 sehen können.

   Dabei sind die Längenangaben in den Spalten in Zentimeter, und die Spalte `Class` kann die folgenden Werte annehmen:

   - Iris-setosa
   - Iris-versicolor
   - Iris-virginica

   Die einzelnen Spalten sind durch ein Komma getrennt.
2. Nutzen Sie `matplotlib.pyplot` für die Erzeugung des Scatter-Plots.

Lösung:

```
# Mathematik
import numpy as np
# Import der pyplot-Funktionen
import matplotlib.pyplot as plt
# Ganz wichtig, sonst wird der Plot nicht angezeigt
%matplotlib inline

# x1 sind die Koordinaten der x-Achse, x2 die der y-Achse
x1    = []
x2    = []
# Farben für die Datenpunkte
colors = []
# Mapping der Schwertlilien zu Farben mit
# einem python dictionary
```



```
iris_colors = { 'Iris-setosa' : 'red',
                'Iris-versicolor' : 'green',
                'Iris-virginica' : 'blue'
              }
# File-Inhalt einlesen
# File Handle fobj wird automatisch geschlossen
with open("iris.csv", "r") as fobj:
    # Den Datensatz zeilenweise verarbeiten
    for line in fobj:
        # Split in einzelne Worte
        words = line.rstrip().split(",")
        # Leerzeilen auslassen
        if len(words) != 5:
            continue
        # SepalLength
        x1.append(float(words[0]))
        # SepalWidth
        x2.append(float(words[1]))
        # Farbe
        colors.append(iris_colors[words[4]])
# Gitter im Scatter-Plot zeichnen
plt.style.use('seaborn-whitegrid')
# Achsenbeschriftung und Titel
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Scatter Plot')
# Den Plot ausgeben
plt.scatter(np.array(x1), np.array(x2), color=colors )
# Plot anzeigen
plt.show()
```

Listing 3.4 Sepal Length und Sepal Width als Scatter-Plot

Anfangs importieren wir wieder die benötigten Module für Plot und Arrays. Danach wird der Datensatz zeilenweise gelesen und verarbeitet. Mit der Function `line.rstrip()` wird eine Kopie des Strings `line` erzeugt, aus dem die Leerzeichen am Ende entfernt wurden. Mittels `split(",")` wird dann der String in seine Einzelwerte zerlegt, immer am Komma getrennt, und die Einzelwerte werden zum Array `words` zusammengefügt. Die ersten zwei Werte im `words`-Array, `words[0]` und `words[1]`, definieren die Koordinaten für den Plot. Der letzte Wert im Array an der Position 4 ist der Name der Lilie, die für die Ermittlung der Farbe verwendet wird. Mit dem Python-Dictionary `iris_colors` findet die

Zuordnung zwischen den Namen und den Farben statt. Danach setzen wir noch die Ausgabe eines Rasters im Plot, übergeben die Daten an den Scatter-Plot und bekommen damit die Ausgabe aus Abbildung 3.14.

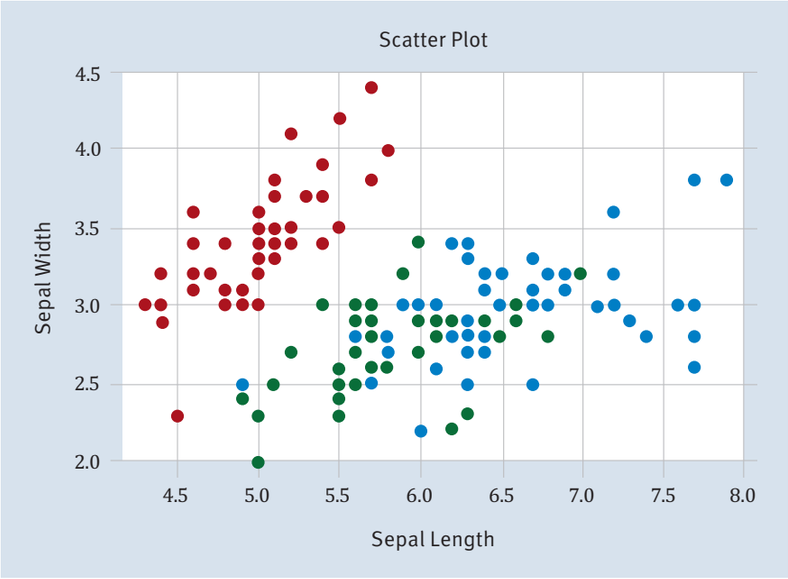


Abbildung 3.14 Die Lilien als Scatter-Plot mit den Koordinaten »Sepal Length« und »Sepal Width«

3.7 Horizontal und vertikal – Spalten- und Zeilenschreibweise

$\vec{w}$ wird als Gewichtsvektor bezeichnet. Wenn wir vom Lernen in neuronalen Netzen sprechen, dann ist damit die Anpassung der Gewichte gemeint. Um zu verstehen, was ein Gewicht ist, stellen Sie sich eine Verbindung zwischen zwei Neuronen vor. Das eine Neuron, nennen wir es A, sendet ein Signal an das zweite Neuron, B. Wie stark das Signal in B ankommt, steuert das Gewicht der Verbindung. Wenn das Gewicht einen Wert zwischen 0 und 1 hat, dann wird die Signalstärke verkleinert; wenn der Wert größer als 1 ist, dann wird das Signal verstärkt, und wenn der Wert kleiner als 0 ist, dann wird das Signal negativ. Somit regelt das Gewicht die Signalstärke, die beim Neuron B ankommt. Details dazu sehen Sie in Kapitel 4, wenn wir das Perceptron-Lernen besprechen.

Zusätzlich zum Inputvektor, in dem wir die Werte übereinandergeschrieben haben (die sogenannte *Spaltenschreibweise*), haben wir den Gewichtsvektor auch in *Zeilenschreibweise* angeführt. Dabei werden die Werte in einer Zeile nebeneinandergeschrieben.

Damit man den Unterschied zu einem Spaltenvektor sehr einfach erkennt, wird dem $\vec{w}$ einfach ein t verpasst, und damit wissen wir, dass es in Zeilenschreibweise dargestellt wird. »Verpassen« ist jetzt ein wenig unmathematisch, nennen wir es einfach *transponieren*. Der Grund, warum wir den ganzen Hokuspokus aufführen, ist, dass wir damit einfach die Multiplikation zwischen den Input-Werten und den Gewichten mathematisch beschreiben und damit die Berechnung in Python optimal implementieren können, nämlich mit $\vec{w}^t \cdot \vec{x}$ anstatt $w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n$, wobei die Anzahl der Komponenten in $\vec{w}$ und $\vec{x}$ gleich sein muss. Das ist doch viel netter, oder? Jedenfalls kurz und bündig.

So führen Sie also die Multiplikation zwischen den Vektoren durch: w_1 mal x_1 plus w_2 mal x_2 plus ... bitte selbstständig vervollständigen.

$$\vec{w}^t \cdot \vec{x} = (w_1 w_2 \ldots w_n) \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n$$

3.7.1 Aufgabe: Ermittlung des Skalarprodukts mithilfe von NumPy

Python sieht die Multiplikation von Vektoren mithilfe von Funktionen des Moduls `numpy` vor, genauer mit der Funktion `dot`. Die Multiplikation wird als *Dot-Product*, *Skalarprodukt* oder *inneres Produkt* bezeichnet. Sie macht aus zwei Sequenzen von Zahlen eine Zahl, wie Sie zuvor gesehen haben.

Fun Facts

1. Die Multiplikation zwischen den Vektoren heißt *Skalarprodukt*, weil das Ergebnis ein Skalar, also ein Wert, ist und kein Vektor.
2. Der englische Name *dot product* kommt daher, dass bei der Multiplikation zwischen den Vektoren ein Punkt steht.

Nun wird es Zeit, das Skalarprodukt mithilfe von Python zu berechnen. Dafür benötigen Sie wieder eine Code-Zelle in Ihrem Jupyter Notebook und, wenn Sie wollen, auch eine Markup-Zelle für eine Überschrift.

Bevor Sie mit der Implementierung beginnen, lesen Sie das folgende Python-Script Zeile für Zeile genau durch, und analysieren Sie die Berechnungen. Nach Ihrer Analyse werden wir Sie nach **zwei** Möglichkeiten fragen, mithilfe von NumPy das innere Produkt zu berechnen.

```
# Die Multiplikation von Vektoren mit numpy
#  Dot Product, Skalarprodukt, inneres Produkt
#   macht aus zwei Sequenzen von Zahlen eine Zahl (algebraisch)
```

```
#  Es ist der Cosinus des Winkels zwischen zwei Vektoren (geometrisch),
#   multipliziert mit deren Längen
# Mathematik
import numpy as np

# numpy array erzeugen
x = np.array([0,1])
# numpy array erzeugen
w = np.array([0.5,0.7])

# Vektor x ausgeben
print("x =", x)
# Vektor w ausgeben
print("w =", w)

# numpy arrays sind keine Matrizen, und
# die Operatoren *, +, -, / funktionieren Element für Element
print("w*x =",w*x)

# Inneres Produkt. Man muss den Vektor nicht transponieren,
# das erledigt numpy für uns
print("np.dot(w,x) =", np.dot(w,x))

# Alternative Syntax zum Dot Product
print("w.dot(x) =", w.dot(x))
# Ausgabe:
x = [0 1]
w = [0.5 0.7]
w*x = [0.  0.7]
np.dot(w,x) = 0.7
w.dot(x) = 0.7
```

Listing 3.5 Skalarprodukt mit Python

Welche zwei Möglichkeiten würden Sie für die Ermittlung des inneren Produkts mit NumPy in Betracht ziehen? Natürlich haben Sie die NumPy-`dot()`-Varianten als die passenden Kandidaten gefunden!

Wir machen weiter mit der Erläuterung der Elemente im Baukasten. Das dritte Element aus dem Baukasten ist die Darstellung der *gewichteten Summe*.

3.8 Die gewichtete Summe

Die Mathematiker leihen sich für die Summe der $w_i \cdot x_i$ ein Symbol aus dem griechischen Alphabet – das große Sigma: $\sum$. Außerdem werden einige zusätzliche »Mascherl« am Sigma befestigt:

$$\sum_{i=1}^{n}$$

Die Mascherl bedeuten, dass die Laufvariable i von einem Anfangswert bis zu einem Endwert nacheinander alle Werte annimmt (in ganzen Schritten *durchläuft* sie den Bereich), in unserem Beispiel vom Anfangswert 1 bis zum Endwert n. Nehmen wir an, dass $n = 3$, dann sind das die Schritte 1, 2 und 3. Mit dem Wissen können wir die Summe aller $w_i \cdot x_i$ – nennen wir sie s – jetzt sehr kompakt schreiben:

$$s = \sum_{i=1}^{n} w_i \cdot x_i$$

Damit ergibt sich der dritte Baustein:

$$s = \sum_{i=1}^{n} w_i \cdot x_i = \vec{w}^{\mathrm{t}} \cdot \vec{x} = w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n$$

Weiter geht es zum vierten Baustein im Baukasten, der Stufenfunktion. Diese hatten wir zwar bereits mehrfach besprochen, jedoch wollten wir sie in einer mathematischen Darstellung präsentieren.

3.9 Schritt für Schritt – Stufenfunktionen

Für die Stufenfunktion $step(s)$ definieren wir die folgende Funktion:

$$step(s) = \begin{cases} 0, & \text{falls } s < \theta \\ 1, & \text{falls } s \geq \theta \end{cases}$$

Egal, mit welchem Wert Sie die Stufenfunktion aufrufen, das Ergebnis ist entweder 0 oder 1, jedoch ist es vom Schwellenwert *Theta*, dem griechischen Buchstaben θ, abhängig, wann der Sprung von 0 auf 1 stattfindet. Nachdem der Wert der Schwelle nicht mehr fixiert ist, kann er auch durch Lernen verändert werden.

Nachdem wir die Stufenfunktion verallgemeinert haben, können wir uns das fünfte Element aus dem Baukasten ansehen, die Berechnung der Funktion $f_{\text{akt}}()$.

3.10 Die gewichtete Summe reloaded

Also noch eine Überlegung zu der gewichteten Summe und der Stufenfunktion. Das geht doch einfacher zu schreiben als diese komplizierte Unterscheidung zum Beispiel bei der Heaviside-Funktion: falls größer θ, dann das, sonst das ...

$$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n \geq \theta$$

Wenn wir die Ungleichung nun umformen, also einfach das θ auf die andere Seite bewegen, dann kommen wir in Richtung erweiterter Gewichtsvektor, der um die eine Dimension für den Schwellenwert erweitert wird. Diese neue Dimension sollte am Index 0 eingefügt werden, damit der ursprüngliche Vektor mit seinen Indizes erhalten bleibt:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n - \theta \geq 0$$

Noch ein kleiner Trick aus der Zauberkiste: Damit wir den Schwellenwert an der Stelle 0 einfügen können, nennen wir $-\theta$ nun einfach w_0. Ist das nicht super? Damit können wir Folgendes schreiben:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n + w_0 \geq 0$$

Damit haben wir eine schöne einheitliche Form gefunden, um wieder $\vec{w}^t \cdot \vec{x}$ zu schreiben. Natürlich muss der Vektor $\vec{x}$ ebenfalls um ein Element verlängert werden. Das hat aber einen sehr einfachen Wert. Können Sie sich vorstellen, welchen? Kurz nachdenken ...

Das können wir mit folgender Abbildung 3.15 beantworten, wo Sie sehen, dass das Neuron für den Input x_0 immer den Wert 1 hat. Es wird als *Bias-Neuron* bezeichnet.

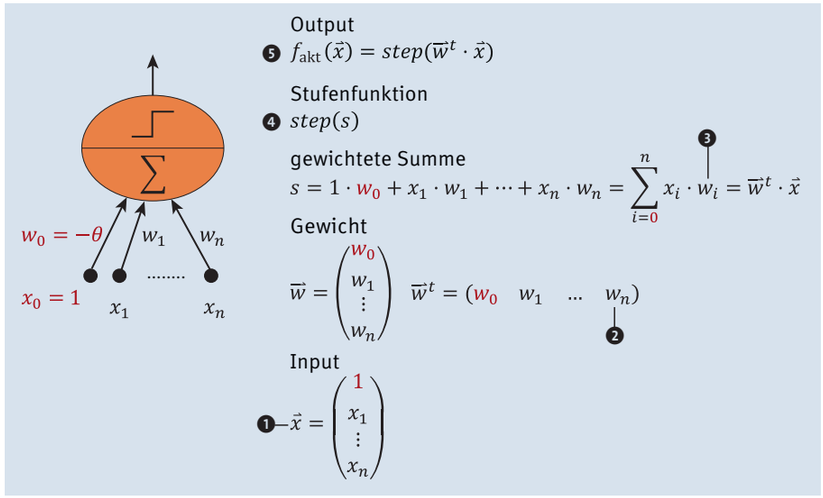


Abbildung 3.15 Bausteine des Perceptrons

Falls Sie das Nachdenken beendet haben, finden Sie die Erweiterungen zu den Vektoren und der gewichteten Summe rot markiert in der Abbildung 3.15.

3.11 Alles zusammen

Es wird Zeit, ein kleines Python-Programm zu schreiben, um unsere Erkenntnisse zum Perceptron in Programmcode umzuwandeln. Sehen Sie sich dazu zuerst die folgenden Erläuterungen und Codezeilen an, und übertragen Sie die in eine neue Code-Zelle in Ihrem Jupyter Notebook, um den gezeigten Output zu produzieren.

Als Input verwenden wir die An-/Abwesenheiten von Herrn Lauch und Frau Karotte. Jede Zeile in der Tabelle 3.4 stellt einen Input-Vektor mit drei Werten dar: 1 für das Bias-Neuron, Anwesenheit Frau Karotte und Anwesenheit Herr Lauch.

Bias-Neuron	Frau Karotte	Herr Lauch	→	Montag	Input-Vektor
1	0	0	→	0	1
1	0	1	→	1	2
1	1	0	→	1	3
1	1	1	→	1	4

Tabelle 3.4 Der erweiterte Input-Vektor für das Planungsproblem

Im Beispiel (1, 0, 1) steht 1 für das Bias-Neuron, 0 für Frau Karotte bedeutet nicht anwesend, 1 für Herr Lauch bedeutet anwesend, und das ergibt insgesamt eine 1 für eine akzeptable Planungssituation. Insgesamt haben wir vier Input-Vektoren, die am einfachsten mithilfe eines zweidimensionalen Arrays verwaltet werden. Nachdem die Input-Vektoren mit `x` bezeichnet werden, verwenden wir ein großes `X` als Bezeichnung für das Array. Die gewünschten Outputs, in der Tabelle in der Spalte »Montag« zu finden, fassen wir als Vektor `y` zusammen. In der folgenden Implementierung haben wir die Heaviside-Funktion für die Stufenfunktion implementiert, also mit Schwellenwert 0, da wir den Schwellenwert zuvor in die gewichtete Summe integriert haben.

Des Weiteren haben wir den Aspekt der **Fehlerberechnung** ergänzt. Für jeden Input wird der berechnete Output mit dem gewünschten Output verglichen. Nachdem das Perceptron nur 0 oder 1 aufgrund der Heaviside-Funktion ausgeben kann und der gewünschte Output nur 0 oder 1 ist, kann die Differenz aus dem gewünschten und dem errechneten Wert nur die Werte –1, 0, 1 annehmen. Um die Einzelfehler summieren und damit die Gesamtaussage über die Ermittlungsgenauigkeit des Perceptrons für die

Daten liefern zu können, wenden wir noch den Betrag auf den Einzelfehler an. Ansonsten würde ein Fehler von –1 den Gesamtfehler verringern.

```
# Mathematik
import numpy as np
# Plot
import matplotlib.pyplot as plt

# 3-dimensionaler Input = Bias-Neuron, Fr. Karotte, Hr. Lauch
# 4 Inputvektoren
X = np.array([
    [1,0,0],
    [1,0,1],
    [1,1,0],
    [1,1,1]])
# Die 4 gewünschten Ergebniswerte
y = np.array([0,1,1,1])

# Heaviside-Funktion
def heaviside( summe ):
    """ Berechnung der Entscheidung zum Wert summe
    Input:  summe
    Output: 1, falls summe >= 0,
            0 sonst
    """
    if summe >= 0:
        return 1
    else:
        return 0

# Perceptron-Berechnung (Forward Path)
def perceptron_eval(X,y):
    """ Perceptron-Berechnung
    Input:  X, Inputvektor
            y, der gewünschte Output
    Output: Der Gesamtfehler, d. h. Summe aus dem Betrag der Differenz
            von errechnetem und gewünschtem Output
    """
    # Der Gesamtfehler
    gesamtfehler = 0;
```



```
    # Die Gewichte so wählen, dass das OR-Problem gelöst werden kann
    w = np.array([-1,1,1])
    # Index i und Element x Ermittlung vom Array X
    for i, x in enumerate(X):
        # x = Zeile für Zeile verwenden
        # Inneres Produkt zwischen x und w
        summe = np.dot(w,x)
        ergebnis = heaviside(summe)
        # Fehler
        fehler = np.abs(ergebnis - y[i])
        # Gesamtfehler
        gesamtfehler += fehler
        # Ausgabe
        print("Fr. Karotte = {}, Hr. Lauch = {},
          gewünschtes Ergebnis = {}, errechnetes Ergebnis = {}, Fehler = {}".
          format(x[1], x[2], y[i], ergebnis, fehler))
    # Gesamtfehler pro Epoche über ganzen Trainingsdatensatz
    return gesamtfehler
#-----------------------------
# Core Function zum Auswerten des Inputs
gesamtfehler = perceptron_eval(X,y)
print("Gesamtfehler = %1d" % (gesamtfehler))
# Ausgabe:
Fr. Karotte = 0, Hr. Lauch = 0, gewünschtes Ergebnis = 0, errechnetes Ergebnis = 0,
Fehler = 0
Fr. Karotte = 0, Hr. Lauch = 1, gewünschtes Ergebnis = 1, errechnetes Ergebnis = 1,
Fehler = 0
Fr. Karotte = 1, Hr. Lauch = 0, gewünschtes Ergebnis = 1, errechnetes Ergebnis = 1,
Fehler = 0
Fr. Karotte = 1, Hr. Lauch = 1, gewünschtes Ergebnis = 1, errechnetes Ergebnis = 1,
Fehler = 0
Gesamtfehler = 0
```

Listing 3.6 Lösung für das sehr einfache Personalplanungsproblem

Der Fehler ist in unserem Beispiel natürlich 0, da wir den Gewichtsvektor optimal gewählt haben. Experimentieren Sie mit den Gewichten im Gewichtsvektor `w`, sehen Sie sich die möglichen Fehler an, und versuchen Sie, diese anhand der vorher besprochenen und implementierten Berechnungen nachzuvollziehen.

3.12 Aufgabe: Roboterschutz

Wir haben noch eine Aufgabe für Sie vorbereitet, die darin besteht, einen Roboter vor dem Absturz zu retten. Dieser kleine Roboter fährt vor sich hin und führt nichts Böses im Schilde. Doch die Umgebung ist ihm nicht wohlgesonnen, und große gefährliche Löcher meinen es nicht gut mit ihm. Glücklicherweise besitzt der Roboter zwei Hell-Dunkel-Sensoren, links und rechts vorn montiert, die die Bodenhelligkeit vor dem Roboter bestimmen können.

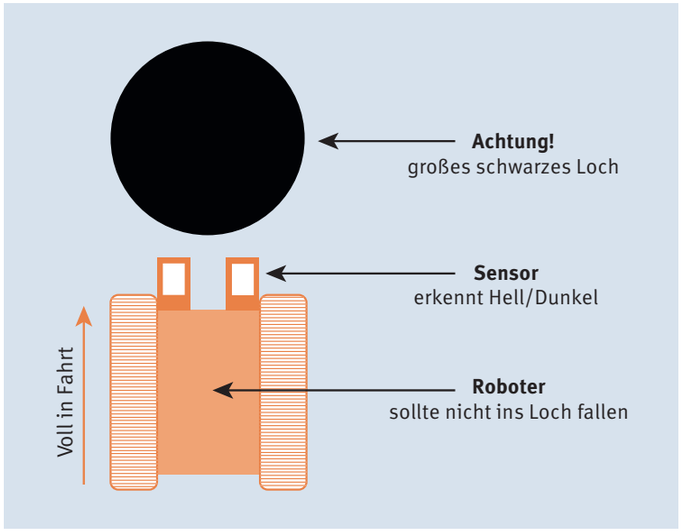


Abbildung 3.16 Einfache Robotersensorik

Wenn ein Sensor **Hell** erkennt, dann liefert dieser den Wert 0, wenn er **Dunkel** erkennt, dann liefert der Sensor 1. Um das Leben des Roboters zu schützen, wäre es schön, wenn Sie ein einfaches Perceptron entwickeln, das erkennt, ob sich ein Loch direkt vor dem Roboter befindet oder ob eine ungefährliche Fahrt möglich ist.

Um das Script zu implementieren, legen Sie bitte eine neue Code-Zelle in Ihrem Jupyter Notebook an, und entwickeln Sie dort Ihre Lösung. Orientieren Sie sich bei dem Beispiel an Listing 3.6.

Lösung:

Wenn wir uns die Helligkeitsaussagen der Sensoren ansehen, dann kann mal der linke Sensor anschlagen und mal der rechte oder beide oder gar keiner. Die Kombinationen der Helligkeitsaussagen veranschaulichen wir uns mithilfe von Tabelle 3.5.

Sensor links	Sensor rechts	Loch
0	0	0
1	0	0
0	1	0
1	1	1

Tabelle 3.5 Locherkennung mithilfe der Sensorwerte

Wir verwenden, so wie von uns empfohlen, die Implementierung aus Listing 3.6 und passen den gewünschten Output `y = np.array([0,0,0,1])` und den Gewichtsvektor auf `w = np.array([-2,1,1])` an, sodass das gewünschte Ergebnis aus Tabelle 3.5 berechnet wird. Zusätzlich haben wir den Text der Ausgabe an die Aufgabenstellung angepasst.

```
# Mathematik
import numpy as np
# Plot
import matplotlib.pyplot as plt
# 3-dimensionaler Input = Bias-Neuron, Sensor links, Sensor rechts
# 4 Inputvektoren
X = np.array([
    [1,0,0],
    [1,0,1],
    [1,1,0],
    [1,1,1]])
# Die 4 gewünschten Ergebniswerte
y = np.array([0,0,0,1])
# Heaviside-Funktion
def heaviside( summe ):
    """Berechnung der Entscheidung zum Wert summe
        Input:  summe
        Output: 1, falls summe >= 0,
                0 sonst
    """
    if summe >= 0:
        return 1
    else:
        return 0
```

```
# Perceptron-Berechnung (Forward Path)
def perceptron_eval(X,y):
    """ Perceptron-Berechnung
    Input:  X, Inputvektor
            y, der gewünschte Output
    Output: Der Gesamtfehler, d. h. Summe aus dem Betrag der Differenz
            von errechnetem und gewünschtem Output
    """
    # Der Gesamtfehler
    gesamtfehler = 0;
    # Die Gewichte so wählen, dass das Roboter-Problem gelöst werden kann
    w = np.array([-2,1,1])
    # Index i und Element x Ermittlung vom Array X
    for i, x in enumerate(X):
        # x = Zeile für Zeile verwenden
        # Inneres Produkt zwischen x und w
        summe = np.dot(w,x)
        ergebnis = heaviside(summe)
        # Fehler
        fehler = np.abs(ergebnis - y[i])
        # Gesamtfehler
        gesamtfehler += fehler
        # Ausgabe
        print("Sensor L = {}, Sensor R = {}, gewünschtes Ergebnis =
 {}, errechnetes Ergebnis = {}, Fehler = {}".
              format(x[1], x[2], y[i], ergebnis, fehler))
    # Gesamtfehler pro Epoche über ganzen Trainingsdatensatz
    return gesamtfehler
#-----------------------------
# Core Function zum Auswerten des Inputs
gesamtfehler = perceptron_eval(X,y)
print("Gesamtfehler = %1d" % (gesamtfehler))
# Ausgabe:
Sensor L = 0, Sensor R = 0, gewünschtes Ergebnis = 0, errechnetes Ergebnis = 0,
Fehler = 0
Sensor L = 0, Sensor R = 1, gewünschtes Ergebnis = 0, errechnetes Ergebnis = 0,
Fehler = 0
Sensor L = 1, Sensor R = 0, gewünschtes Ergebnis = 0, errechnetes Ergebnis = 0,
Fehler = 0
Sensor L = 1, Sensor R = 1, gewünschtes Ergebnis = 1, errechnetes Ergebnis = 1,
Fehler = 0
Gesamtfehler = 0
```

Listing 3.7 Die Perceptron-Steuerung für den Roboter

3.13 Zusammenfassung

Das Perceptron gehört nun Ihnen, Sie haben den Aufbau und die Berechnungen im Perceptron theoretisch und praktisch durchgeführt. Dabei sind so wichtige Dinge wie gewichtete Summe, Schwellenwertberechnungen, Grafiken und grafische Analysen vorgekommen. Zugegebenermaßen haben wir bis jetzt ausschließlich Auswertungen umgesetzt und das Lernen des Perceptrons noch nicht berücksichtigt. Das werden wir im folgenden Kapitel sofort nachholen.

3.14 Referenzen

- Rosenblatt, F.: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. 1958. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.588.3775*
- *https://archive.ics.uci.edu/ml/datasets/iris/iris.data*

Auf einen Blick

Inhalt