Jupyter Book

Contents

This is a small sample book to give you a feel for how book content is structured.

> **Note**
>
> Here is a note!

And here is a code block:

```
e = mc^2
```

Check out the content pages bundled with this sample book to see more.

What is possible with the Jupyter Book?

Die Idee ist es, ein Video der jeweiligen Schraube aufzunehmen und aus diesem Video die Bilder mit Python zu extrahieren. Um es am Anfang etwas einfacher zu gestalten nehmen wir zunächst nur die Schraubenköpfe. Wir beginnen mit 100 Bildern pro Schraube. Diese ersten 100 Bilder sollen alle unter gleichen Bedingungen aufgenommen werden, sie werden zum trainieren des Modells genutzt. Die Testbilder werden werden separat aufgenommen, sie können Schatten enthalten und in unterschiedlicher Beleuchtung aufgenommen werden um die Leistungsfähigkeit des Modells einschätzen zu können.

Markdown Files

Whether you write your book's content in Jupyter Notebooks (`.ipynb`) or in regular markdown files (`.md`), you'll write in the same flavor of markdown called **MyST Markdown**.

What is MyST?

MyST stands for "Markedly Structured Text". It is a slight variation on a flavor of markdown called "CommonMark" markdown, with small syntax extensions to allow you to write **roles** and **directives** in the Sphinx ecosystem.

What are roles and directives?

Roles and directives are two of the most powerful tools in Jupyter Book. They are kind of like functions, but written in a markup language. They both serve a similar purpose, but **roles are written in one line**, whereas **directives span many lines**. They both accept different kinds of inputs, and what they do with those inputs depends on the specific role or directive that is being called.

Using a directive

At its simplest, you can insert a directive into your book's content like so:

```
```{mydirectivename}
My directive content
```
```

This will only work if a directive with name `mydirectivename` already exists (which it doesn't). There are many pre-defined directives associated with Jupyter Book. For example, to insert a note box into your content, you can use the following directive:

```
```{note}
Here is a note
```
```

This results in:

> **Note**
>
> Here is a note

In your built book.

For more information on writing directives, see the MyST documentation.

Using a role

Roles are very similar to directives, but they are less-complex and written entirely on one line. You can insert a role into your book's content with this pattern:

```
Some content {rolename}`and here is my role's content!`
```

Again, roles will only work if `rolename` is a valid role's name. For example, the `doc` role can be used to refer to another page in your book. You can refer directly to another page by its relative path. For example, the role syntax `{doc}`intro`` will result in: Jupyter Book.

For more information on writing roles, see the MyST documentation.

Adding a citation

You can also cite references that are stored in a `bibtex` file. For example, the following syntax: `{cite}holdgraf_evidence_2014` will render like this: [HdHPK14].

Moreover, you can insert a bibliography into your page with this syntax: The `{bibliography}` directive must be used for all the `{cite}` roles to render properly. For example, if the references for your book are stored in `references.bib`, then the bibliography is inserted with:

```
```{bibliography}
```
```

Resulting in a rendered bibliography that looks like:

[HdHPK14] Christopher Ramsay Holdgraf, Wendy de Heer, Brian N. Pasley, and Robert T. Knight. Evidence for Predictive Coding in Human Auditory Cortex. In *International Conference on Cognitive Neuroscience*. Brisbane, Australia, Australia, 2014. Frontiers in Neuroscience.

Executing code in your markdown files

If you'd like to include computational content inside these markdown files, you can use MyST Markdown to define cells that will be executed when your book is built. Jupyter Book uses *jupytext* to do this.

First, add Jupytext metadata to the file. For example, to add Jupytext metadata to this markdown page, run this command:

```
jupyter-book myst init markdown.md
```

Once a markdown file has Jupytext metadata in it, you can add the following directive to run the code at build time:

```
```{code-cell}
print("Here is some code to execute")
```
```

When your book is built, the contents of any `{code-cell}` blocks will be executed with your default Jupyter kernel, and their outputs will be displayed in-line with the rest of your content.

For more information about executing computational content with Jupyter Book, see The MyST-NB documentation.

Herzlich Willkommen

Diese Masterarbeit behandelt die Erkennung von Schrauben auf Bildern und soll den Einstieg in das Thema Neuronale Netze erleichtern.

Content with notebooks

You can also create content with Jupyter Notebooks. This means that you can include code blocks and their outputs in your book.

Markdown + notebooks

As it is markdown, you can embed images, HTML, etc into your posts!

Markedly Structured Text

You can also \(add_{math}\) and
\[math^{blocks} \]
or
\[\begin{split} \begin{aligned} \mbox{mean} la_{tex} \\ \\ math blocks \end{aligned} \end{split}\]
But make sure you $Escape $your $dollar signs $you want to keep!

MyST markdown

MyST markdown works in Jupyter Notebooks as well. For more information about MyST markdown, check out the MyST guide in Jupyter Book, or see the MyST markdown documentation.

Code blocks and outputs

Warning

Jupyter Book will also embed your code blocks and output in your book. For example, here's some sample Matplotlib code:

```
from matplotlib import rcParams, cycler
import matplotlib.pyplot as plt
import numpy as np
plt.ion()
```

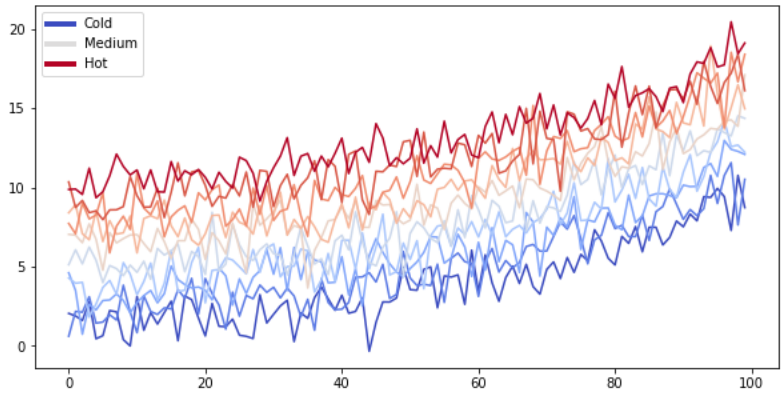
```
<matplotlib.pyplot._IonContext at 0x2588535c688>
```

```
# Fixing random state for reproducibility
np.random.seed(19680801)

N = 10
data = [np.logspace(0, 1, 100) + np.random.randn(100) + ii for ii in range(N)]
data = np.array(data).T
cmap = plt.cm.coolwarm
rcParams['axes.prop_cycle'] = cycler(color=cmap(np.linspace(0, 1, N)))


from matplotlib.lines import Line2D
custom_lines = [Line2D([0], [0], color=cmap(0.), lw=4),
                Line2D([0], [0], color=cmap(.5), lw=4),
                Line2D([0], [0], color=cmap(1.), lw=4)]

fig, ax = plt.subplots(figsize=(10, 5))
lines = ax.plot(data)
ax.legend(custom_lines, ['Cold', 'Medium', 'Hot']);
```



There is a lot more that you can do with outputs (such as including interactive outputs) with your book. For more information about this, see the Jupyter Book documentation

Bilder aus Video

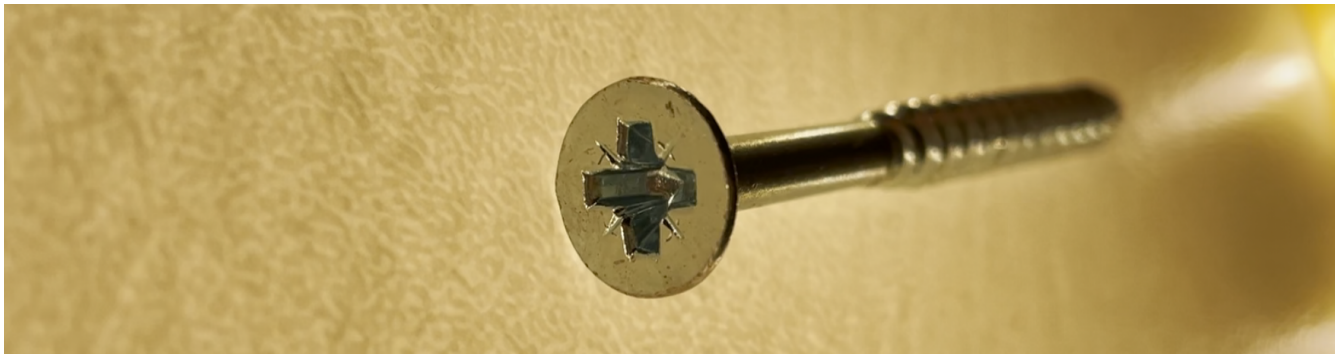


Fig. 1 This is a Pozidriv **Schraube**!

Hier ist ein **Link**.

Das Bild der Schraube stammt aus einem Video, welches mit einem Iphone 12 pro aufgenommen wurde. Die Aufnahme ist sehr detailreich, das Pozidriv Profil sowie auch Beschädigungen, Rost und Ablagerungen auf der Oberfläche, sind gut zu erkennen.

Nachdem wir die Videos aufgenommen haben und die Softwareumgebung installiert ist, beginnen wir mit dem Python Programm zum Extrahieren von Bildern aus Videos:

> **Note**
>
> Es stellt sich die Frage wie detailreich die Schraubenbilder sein müssen um eine gute Erkennung zu ermöglichen?

Extracting and Saving Video Frames using OpenCV-Python

```
# OpenCV importieren:

import cv2



# path = 'relativer Speicherpfad / Dateiname':

path = 'pozi/pozi'



# Video laden:

cap = cv2.VideoCapture('pozi.MOV')
i = 0


# Prüfen ob ein Video geladen wurde:

if cap.isOpened() == False:
    print('ERROR: Datei nicht gefunden')



# Die Frames des Videos lesen:

while(cap.isOpened()):
    ret, frame = cap.read()

    # sobald keine Frames mehr gelesen werden können (ret==False) wird abgebrochen:
    if ret == False:
        break

    # Die Frames speichern
    cv2.imwrite(path+str(i)+'.jpg', frame)
    i += 1


cap.release()
cv.destroyAllWindows()
```

Resize Image

```
import cv2

img = cv2.imread('pozi/pozi800.jpg', cv2.IMREAD_UNCHANGED)

print('Original Dimensions : ',img.shape)

scale_percent = 50 # percent of original size
width = int(img.shape[1] * scale_percent / 100)
height = int(img.shape[0] * scale_percent / 100)
dim = (width, height)

#resize image
resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)

print('Resized Dimensions : ',resized.shape)

cv2.imshow("Resized image", resized)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_23276/1098803240.py in <module>
----> 1 import cv2
      2
      3 img = cv2.imread('pozi/pozi800.jpg', cv2.IMREAD_UNCHANGED)
      4
      5 print('Original Dimensions : ',img.shape)

ModuleNotFoundError: No module named 'cv2'
```

Image Data Generator

Preprocessing

Mit dem Image Data Generator werden wir unsere Bilder weiterverarbeiten. Wir werden den Datensatz künstlich erweitern um so das Modell robuster zu machen.

Bilder in Numpy Array speichern

Um sehr viele Bilder abzuspeichern in einer Form die für das CNN passend ist, werden die Bilder in ein Numpy Array gespeichert. Dieses Numpy Array ist zunächst 3 Dimensional, wird später aber noch auf 4D erweitert.

Wie man nun die Bilder oder besser gesagt die Pixelwerte in ein Numpy Array speichert zeigt der folgende Programmcode:

```
# lob importieren
import glob

# import numpy
import numpy as np

# import Image from PIL
from PIL import Image

# Dateinamen der Bilder in filelist speichern
filelist = glob.glob('../pozi/*.jpg')

# Alle Bilder nacheinander öffnen und hintereinander in ein Numpy Array speichern
x = np.array([np.array(Image.open(fname)) for fname in filelist])
```

Labels erstellen

Wie erstellt man die Labels?

Datensatz in Numpy Array exportieren

(X_Train, y_Train, X_Test, y_Test) in ein Numpy Array gemeinsam speichern:

Ein einzelnes Neuron

Um den Einstieg zu erleichtern, schauen wir uns zunächst ein einzelnes Neuron genauer an.

Lernziele:

- verstehen was ein einzelnes Neuron macht
- Ein einzelnes Neuron in Python modellieren
- die Konzepte hinter Neuronalen Netzen besser verstehen

Ein einzelnes Neuron besitzt mehrere, gewichtete Eingänge und einen Ausgang siehe folgende Abbildung.

Weitere Informationen im Artikel, Neuronale Netze: Ein Blick in die Blackbox.

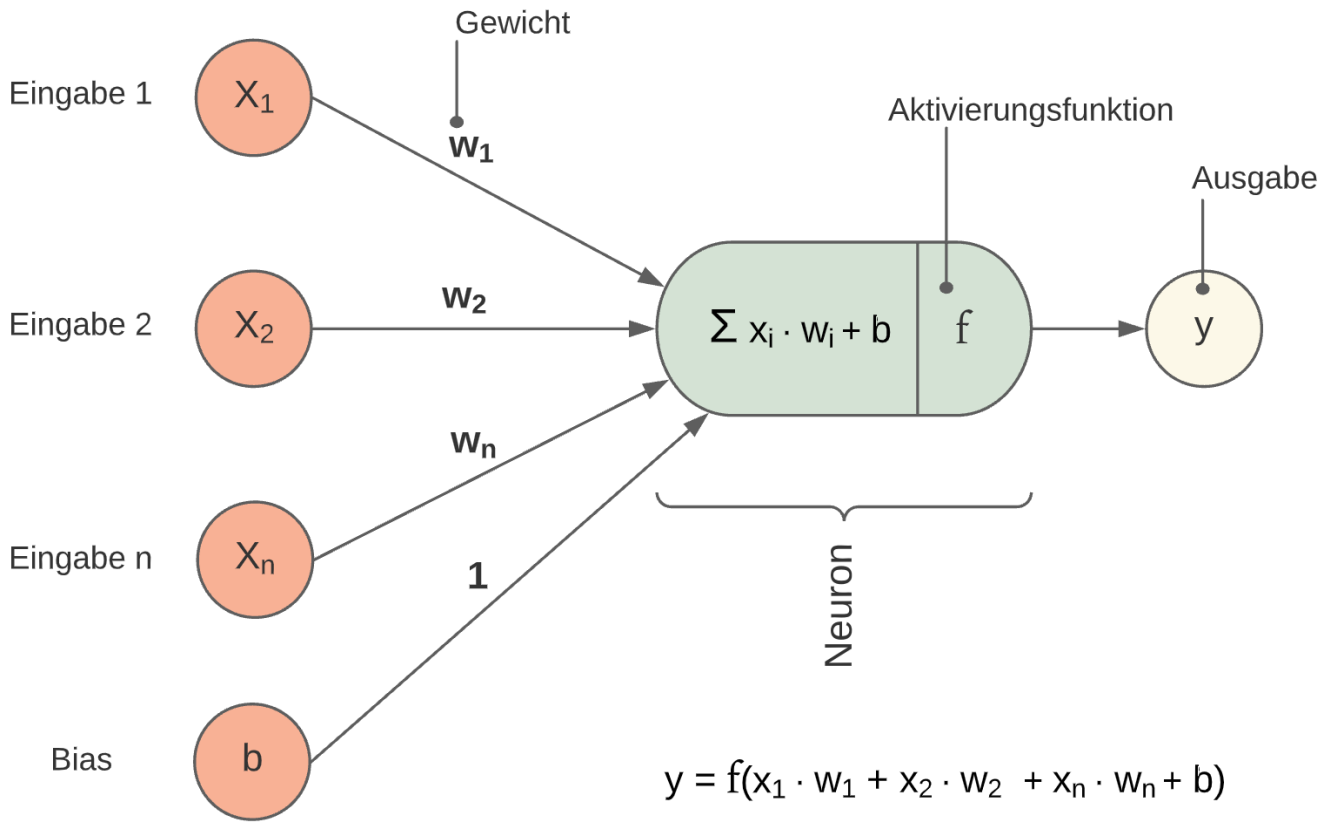


Fig. 2 Modell eines einzelnen Neurons.

Die gewichteten Eingabewerte werden mit einem Bias verrechnet, vom Neuron aufsummiert und durch eine Aktivierungsfunktion wird ein Ausgabewert berechnet.

Wir beginnen zunächst mit dem einfachsten Fall:

Ein vereinfachtes Neuron

In der Abbildung ist ein vereinfachtes künstliches Neuron zu sehen. Dieses Neuron besitzt keine Aktivierungsfunktion und keinen Bias-Term. Es besitzt lediglich eine Summierfunktion.

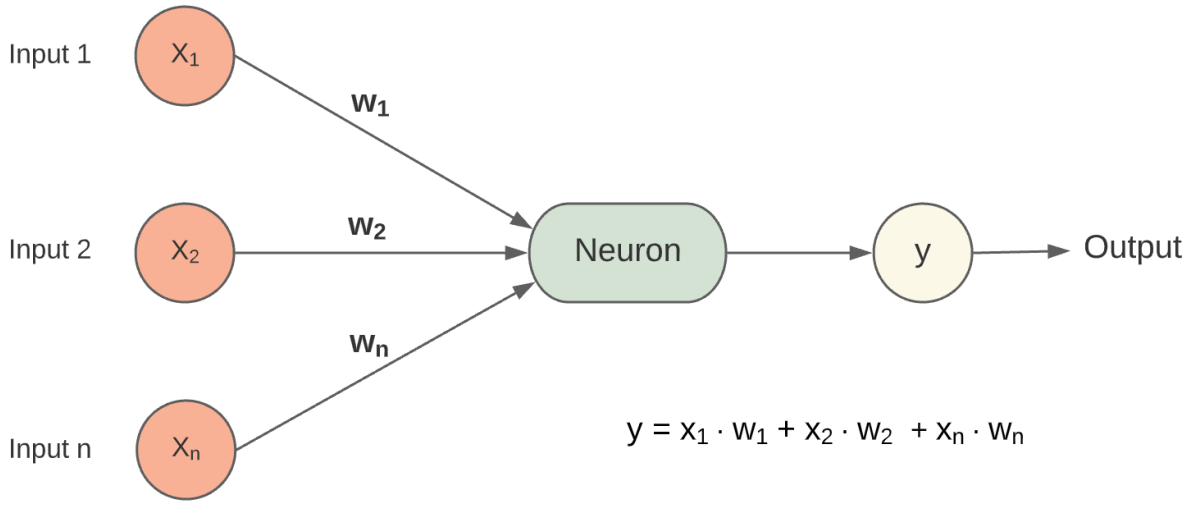


Fig. 3 Modell eines einzelnen Neurons ohne Bias und Aktivierungsfunktion.

Die **Eingabedaten X** werden nun mit den **Gewichten w** verrechnet. Das Neuron bildet anschließend die Summe und gibt diese als **Ausgabewert y** aus. Wie die Werte für die Gewichte bestimmt werden kann man im **Kapitel: Lernvorgang** nachlesen. Für das weitere Vorgehen, reicht es aus, zu wissen, dass es diese Gewichte gibt.

Anhand dieses einfachen Neurons kann man den konzeptionellen Aufbau Neuronaler Netze besser verstehen. Was dieses Neuron kann, schauen wir uns anhand eines Beispiels in Python an.

Lineare Regression ohne Bias

evt weglassen und nur mit Bias erklären

Linear Regression and Bias

Beispiel: Millimeter in Zoll umrechnen

```
X = [
    [1],
    [15],
    [60]
]

y = [
    0.0393701,
    0.590551,
    2.3622
]
```

```
from sklearn import linear_model
model = linear_model.LinearRegression(fit_intercept = False)
model.fit(X, y)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_15240/2661756792.py in <module>
----> 1 from sklearn import linear_model
      2 model = linear_model.LinearRegression(fit_intercept = False)
      3 model.fit(X, y)

ModuleNotFoundError: No module named 'sklearn'
```

Das Modell hat nun den Zusammenhang der Trainingsdaten gelernt. Das gelernte Gewicht entspricht dabei der Steigung der Regressionsgeraden.

```
# Steigung bzw Koeffizient oder Gewicht
print(model.coef_)
```

```
[1.8]
```

Diesen gelernten Zusammenhang können wir nun auch auf neue Daten anwenden:

```
print(100 * 0.03937)
```

```
3.9370000000000003
```

Eine "Vorhersage" für eine Reihe an Werten bekommt man mit der predict-Methode:

```
model.predict([
    [120],
    [130]
])
```

```
array([4.72440047, 5.11810051])
```

Das einzelne Neuron ist bereits in der Lage den Zusammenhang zwischen Millimeter und Zoll aus den Trainingsdaten zu lernen.

Lineare Regression mit Bias

Beispiel: Grad Celsius in Fahrenheit umrechnen

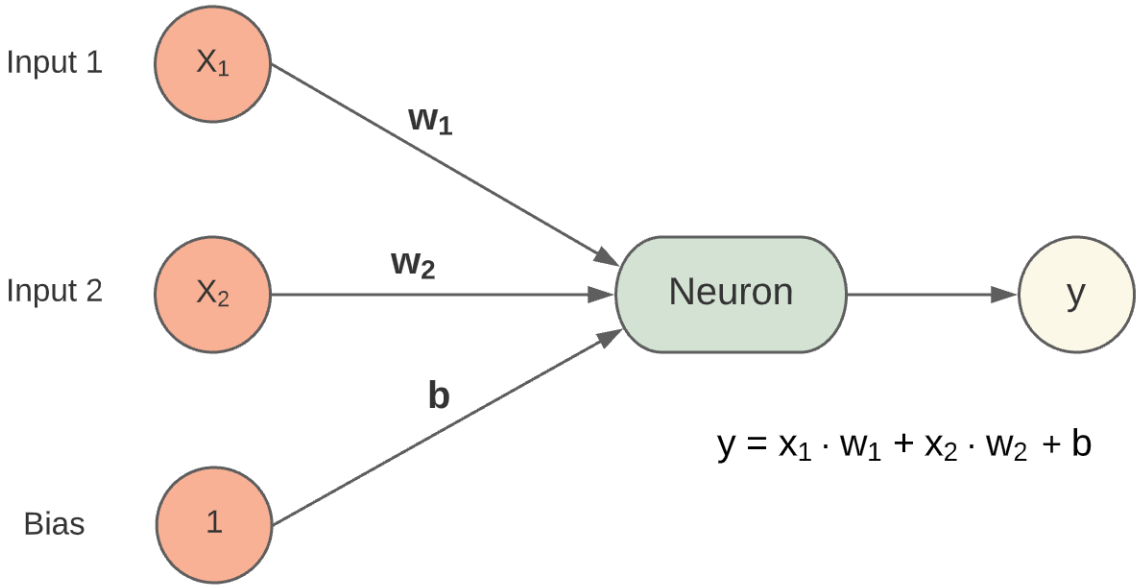


Fig. 4 Modell eines einzelnen Neurons mit Bias.

```
X = [
    [-10],
    [0],
    [20]
]

y = [
    14,
    32,
    68
]
```

```
from sklearn import linear_model
model = linear_model.LinearRegression(fit_intercept = True)
model.fit(X, y)
```

```
LinearRegression()
```

Dadurch, dass wir dem Modell mit **fit_intercept = True** einen weiteren Freiheitsgrad zur Verfügung stellen, ist das Modell in der Lage den Zusammenhang zwischen Grad Celsius und Fahrenheit zu lernen.

```
print(model.coef_)
print(model.intercept_)
```

```
[1.8]
32.0
```

Das Neuron hat den Zusammenhang korrekt gelernt.

Die Berechnungsformel lautet: **°F = °C * 1,8 + 32 (von Celsius nach Fahrenheit)**

Beispiel: Verbrauch von Autos:

Ein weiteres berühmtes Machinelearningbeispiel ist der MPG Datensatz.

Verbrauch von Autos vorhersagen

Aufgabe:

Eine Firma hat ein neues Auto angekündigt, aber noch keine Verbrauchsdaten angegeben. Kannst du den Verbrauch (in l/100km) des Autos schätzen, indem du ein Modell trainierst?

Das Auto hat:

- 8 Zylinder
- 200PS
- 2500kg

Lese dazu die Datei `mpg-dataset.csv` ein. Trainiere anschließend ein Modell, und sage den Verbrauch (in l/100km) dieses Autos vorher!

```
def mpg_to_l_per_100km(mpg):
    LITERS_PER_GALLON = 3.785411784
    KILOMETERS_PER_MILES = 1.609344

    return (100 * LITERS_PER_GALLON) / (KILOMETERS_PER_MILES * mpg)

print(mpg_to_l_per_100km(100))
```

```
2.352145833333333
```

```
import pandas as pd

df = pd.read_csv("mpg-dataset.csv")
```

```
X = df[["cylinders", "horsepower", "weight"]]
```

```
y = mpg_to_l_per_100km(df["mpg"])
```

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X, y)
```

```
LinearRegression()
```

```
print(model.coef_)
print(model.intercept_)
```

```
[0.2757425  0.03284566 0.00229432]
-0.5232093219825558
```

```
print(model.predict([
    [8, 200, 2500]
]))
```

```
[13.98766161]
```

```
/home/martin/anaconda3/envs/scikit/lib/python3.9/site-packages/sklearn/base.py:445: UserWarning:
X does not have valid feature names, but LinearRegression was fitted with feature names
  warnings.warn(
```

Aktivierungsfunktion

Wozu ist die Aktivierungsfunktion nötig?

Die vereinfachten, linearen Neuronen lassen sich nicht hintereinander schalten/verknüpfen bzw macht es keinen Sinn dies zu tun. Eine Beispielrechnung verdeutlicht das:

Beispiel: Am Eingang X liegt eine 5 an und am Ausgang soll eine 20 ausgegeben werden. Die beiden Gewichte bekommen 2 als Faktor. Das gleiche Ergebnis würde heraus kommen wenn man ein Neuron mit einem Gewicht und dem Faktor 4 verwendet.

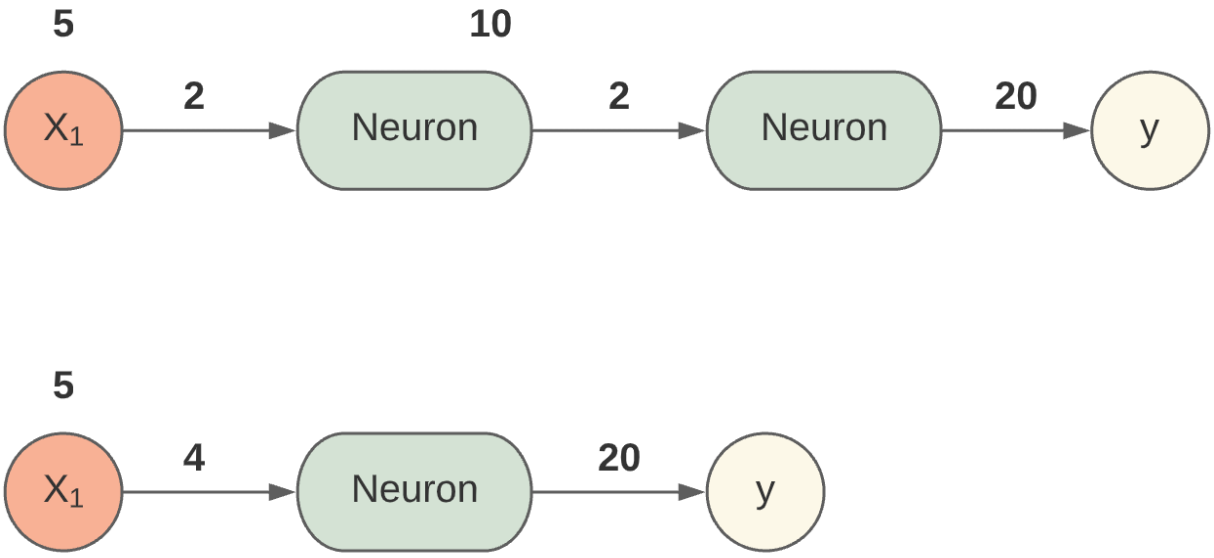


Fig. 5 Lineare Neuronen in Reihe geschalten.

- "hintereinanderschalten" von vereinfachten, linearen Neuronen ohne Aktivierungsfunktion nicht möglich
- Bisher keine Ja/Nein Antworten oder Ausgabewerte zwischen 0 und 1 möglich

Die **Sigmoidfunktion** bildet die Ergebnisse auf den Zahlenbereich zwischen 0 und 1 ab. Die Ergebnisse können in der Form als eine Wahrscheinlichkeit aufgefasst werden:

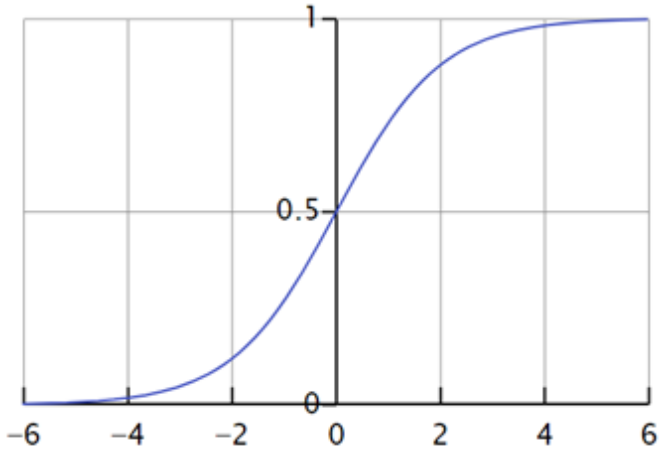


Fig. 6 Modell eines einzelnen Neurons mit Bias und Aktivierungsfunktion.

Logistische Regression

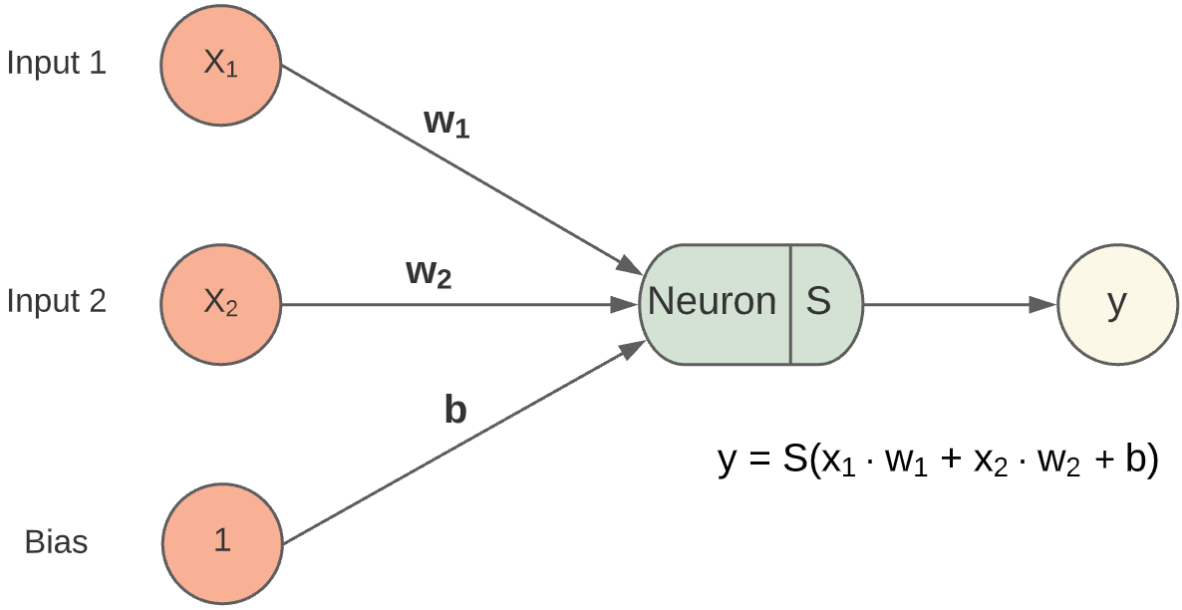


Fig. 7 Modell eines einzelnen Neurons mit Bias und Aktivierungsfunktion.

Beispiel: Wird ein Studierender die Prüfung bestehen?

```
# X = Wie viele Stunden wurde gelernt?

X = [
    [50],
    [60],
    [70],
    [20],
    [10],
    [30],
]

y = [
    1,
    1,
    1,
    0,
    0,
    0,
]
```

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(C = 100000)
model.fit(X, y)
```

```
LogisticRegression(C=100000)
```

```
model.predict([
    [44]
])
```

```
array([1])
```

```
model.predict_proba([
    [35]
])
```

```
array([[0.99873289, 0.00126711]])
```

Wie lernt ein neuronales Netz?

In diesem Abschnitt soll ein Einblick in den Aufbau und den Lernvorgang eines Neuronalen Netzes geschaffen werden.

Hidden Layer

Bisher hatten wir nur ein Neuron. Da ein neuronales Netz aus mehreren solcher Neuronen aufgebaut ist, wollen wir uns in diesem Abschnitt damit befassen wie die einzelnen Neuronen zu einem Netz zusammen geschalten werden, wie diese einzelnen Neuronen arbeiten und wie es das Neuronale Netz schafft etwas zu lernen.

Ein Netz aus mehreren Neuronen: Wir beginnen wieder mit einem einfachen Beispiel und verbinden ein paar Neuronen zu einem einfach NN:

- **Input Layer:** X1, X2, X3, b
- **Hidden Layer** Neuron 1, Neuron 2, Neuron 3
- **Output Layer** Neuron 4

Beispiel:

- X1: Anzahl Zylinder
- X2: Leistung kw
- X3: Gewicht kg

Die Neuronen verteilen sich selbst auf verschiedene Features auf. Jedes Neuron spezialisiert sich auf eine bestimmte Eigenschaft z.B.:

- Neuron 1: Kleinwagen oder SUV (relevant: X1, X2, X3)
- Neuron 2: Preis
- Neuron 3: Beschleunigung (relevant: X2,X3)

Relevante Verbindungen werden vom Algorithmus verstärkt und nicht benötigte Verbindungen werden ignoriert (Gewicht wird sehr klein oder Null).

Der Output-Layer soll z.B. den Verbrauch vorhersagen und wird entsprechend jene Verbindungen verstärken, die besonders großen Einfluss auf den Verbrauch haben.

Die Aktualisierung der Gewichte hat einen großen Einfluss auf diesen Vorgang.

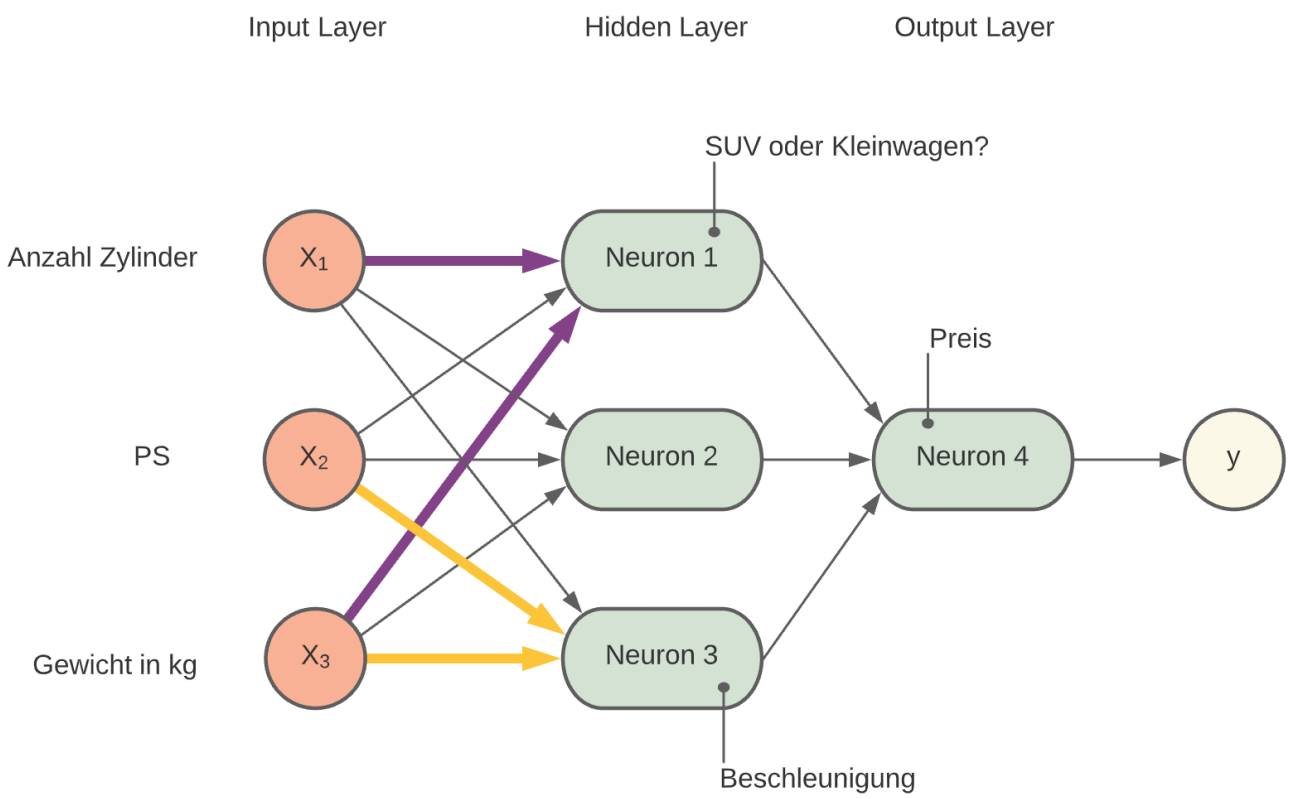


Fig. 8 Two-Layer-Neural Net.

Weitere Informationen: Neural network architecture.

> **Note**
>
> Die Neuronen im Hidden Layer übernehmen jeweils verschiedene Hilfsaufgaben.. Der Output Layer kombiniert der Ergebnisse aus dem Hidden Layer und gibt eine Vorhersage aus.

```
- Neuronale Netze mit einem beliebig großen Hidden-Layer können jede beliebige Funktion approximieren.
- je mehr Knoten das Netz besitzt, desto genauer kann es die math. Funktionen annähern.
```

Gewichte aktualisieren

Nach der Initialisierung der Gewichte. Wird ein Ausgangswert berechnet. Passt dieser "Vorhersagewert" nicht zum richtigen Ergebnis, dann müssen die Gewichte dementsprechend angepasst werden, so dass das Ergebnis stimmt.

In der Abb. werden w1 und w2 so lange erhöht bis der Vorhersagewert zum richtigen Wert passt.

Das Neuron gibt 0.5 aus, obwohl der richtige Wert 0.75 ist. Das bedeutet, das Modell ist noch nicht so gut an die Daten angepasst. Um das Modell den Daten besser anzupassen, stehen nur die Gewichte als Stellschrauben zur Verfügung und diese können nun Stückweise erhöht werden bis das Modell die Daten ausreichend approximiert hat siehe Abbildung unten.

Kostenfunktion

Die Aktualisierung der Gewichte wird mit Hilfe einer Kostenfunktion erreicht. Es gibt verschiedene Kostenfunktionen, eine davon ist die **"quadratische Fehlerfunktion"**.

Weitere Kostenfunktionen und Informationen hier.

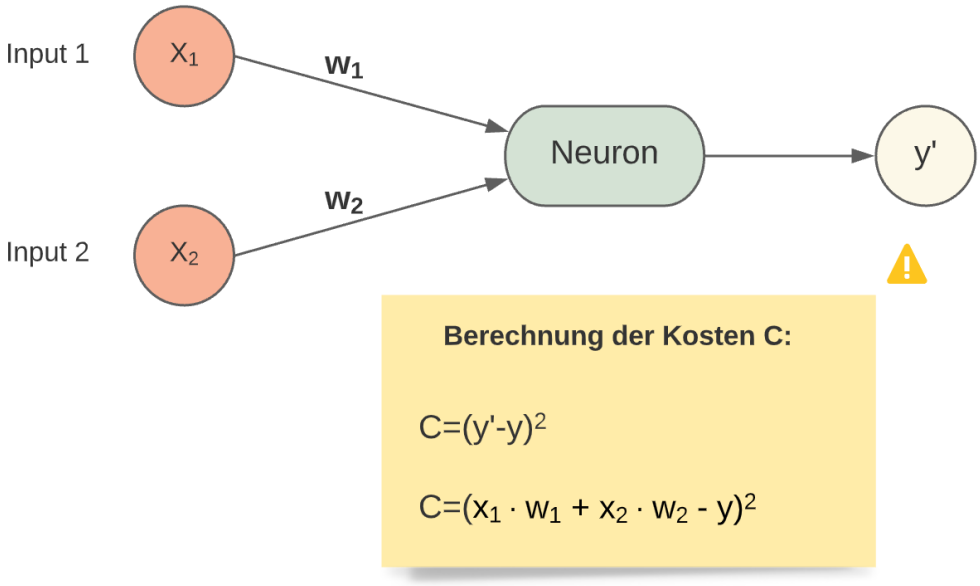


Fig. 9 Kostenfunktion

Die Quadrierung des Fehlers in der Kostenfunktion, bewirkt eine viel größere Bestrafung für größere Fehler.

Werden nun wie üblich mehrere Datensätze trainiert, müssen die Gewichte nach jedem Datensatz angepasst werden. Je nachdem wie die Vorhersage vom Ergebnis y abweicht.

Die Abbildung dient nur zur Veranschaulichung. Im Abschnitt Gradientenabstieg wird gezeigt wie die Kostenfunktion in Python minimiert wird.

Eine Kostenfunktion dient zum Minimieren des Fehlers. Man stellt eine Funktion auf, die den Fehler zwischen Schätzung und richtigem Wert berechnet und sucht dann die zugehörigen Gewichte, die den Fehler minimal werden lassen. Die Kosten C werden als Funktion der Gewichte formuliert. Da man es bei NN's meistens mit komplexeren Funktionen und sehr vielen Gewichten zu tun hat, kann man das nicht mehr analytisch lösen. Für so einen Fall eignet sich das Gradientenabstiegsverfahren.

Gradientenabstieg

Wichtig zum Verständnis des Trainings von neuronalen Netzen.

Wie aktualisiert der Computer mehrere tausend Gewichte?

Mit dem Gradientenabstiegsverfahren wird Schritt für Schritt das Minimum einer Funktion gesucht. Bei einfachen Funktionen kann man das noch analytisch lösen aber bei komplexeren Funtionen benötigt man das Gradientenabstiegsverfahren.

Wie findet man das Minimum bei komplexen Funktionen wie im Bild rechts? Die Antwort lautet Gradientenabstiegsverfahren. Um dieses Verfahren näher zu erläutern beginnen wir wieder mit einem einfachen Fall,
Für den gradient descent gab es bzgl Lernrate usw eine gute Geogebra erklärung in einem anderen Kurs von Jannis, diese Erklärung hier einfügen.

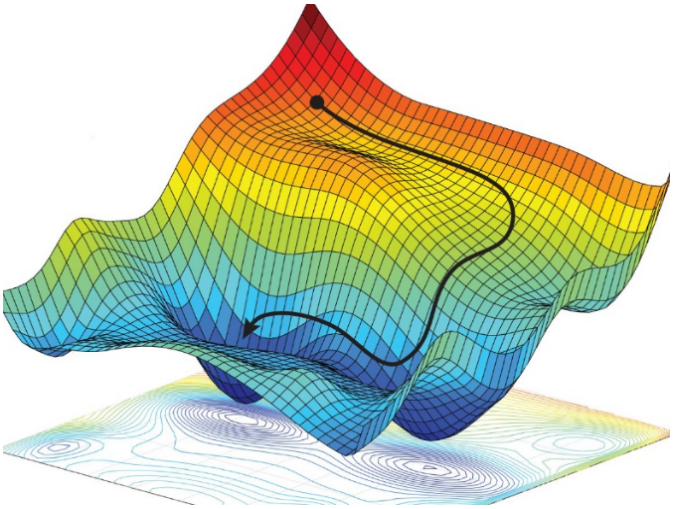


Fig. 10 Die Gewichte der ersten Batch werden trainiert.

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

def f(x):
    return x ** 2 - 4 * x + 5


def f_ableitung(x):
    return 2 * x - 4


x = 5

#Schrittweite bzw Lernrate (lr):
lr = 0.05

plt.scatter(x, f(x), c="r")
for i in range(0, 25):
    steigung_x = f_ableitung(x)
    x = x - lr * steigung_x
    plt.scatter(x, f(x), c="r")
    print(x)

xs = np.arange(-2, 6, 0.1)
ys = f(xs)
plt.plot(xs, ys)
plt.show()
```

Important

> **Note**
>
> Man kann nun mit der Lernrate experimentieren und z.B. einen großen Wert wählen. Es kann passieren, dass bei einer zu großen Schrittweite das Minimum übersprungen wird und somit nicht gefunden werden kann.

Gut zu wissen

In hochdimensionalen Räumen spielen lokale Minimas keine Rolle mehr. Erklärung folgt.

In der Praxis hat man es eher mit komplexeren Funktionen mit mehreren Minimas zu tun. Die Gefahr, in einem lokalen Minimum stecken zu bleiben, ist in höher Dimensionalen Räumen zu vernachlässigen. Erklärung kommt später noch.

Geogebra dateien zeigen

Stochastic Gradient Descent

Lernziele:

- Was ist eine Batch?
- Wozu braucht man Batches?
- Welche Größe sollten die Batches haben?

Die Kosten für die gesamten Trainingsdaten zu berechnen und anschließend die Gewichte zu aktualisieren, würde bei sehr vielen Gewichten einen sehr hohen Rechenaufwand bedeuten, da die Kostenfunktion dann sehr viele variable Gewichte enthält. Deswegen geht man bei NN's so vor, dass man nicht die Kosten für die gesamten Daten sondern nur für einzelne Batches berechnet und somit die Kosten approximiert. Das macht man dann für alle Batches und aktualisiert nach jedem Batch die Gewichte. So werden die Gewichte pro kompletten Durchgang mehrmals aktualisiert und nicht nur einmal am Ende eines kompletten Durchgangs. Das bringt allerdings mit sich, dass die Gewichte hin und her springen, im „ZickZack zum Minimum laufen" Das führt insgesamt zu einem schnelleren Lernvorgang.

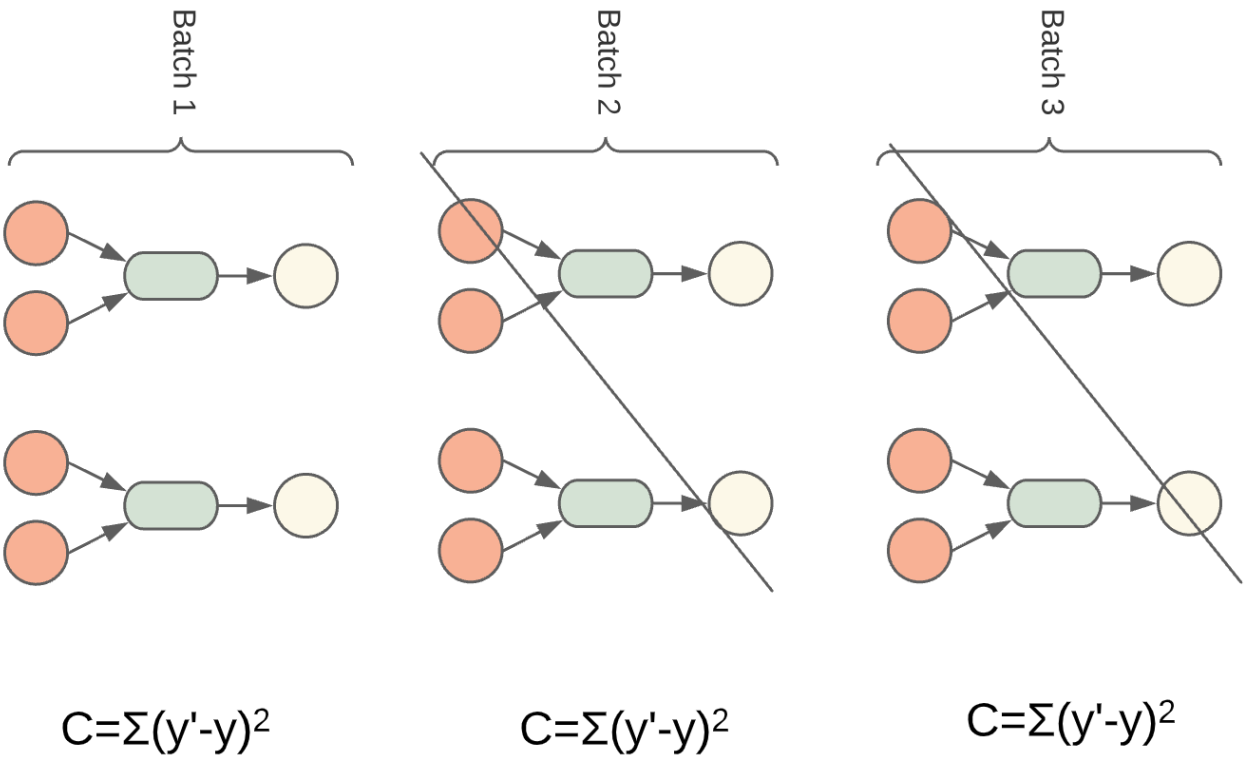


Fig. 11 erste Batch

Anstatt alle Gewichte mit einmal zu bestimmen ist es vorteilhafter den Trainingssatz in einzelne Batches aufzuteilen, somit wird die Berechnung schneller und die Gewichte werden nach jedem Durchlauf angepasst.

Ablauf der Gewichtsanpassung für eine Batch:

- Vorhersage machen
- Kosten berechnen
- Gewichte anpassen
- dann nächste Batch

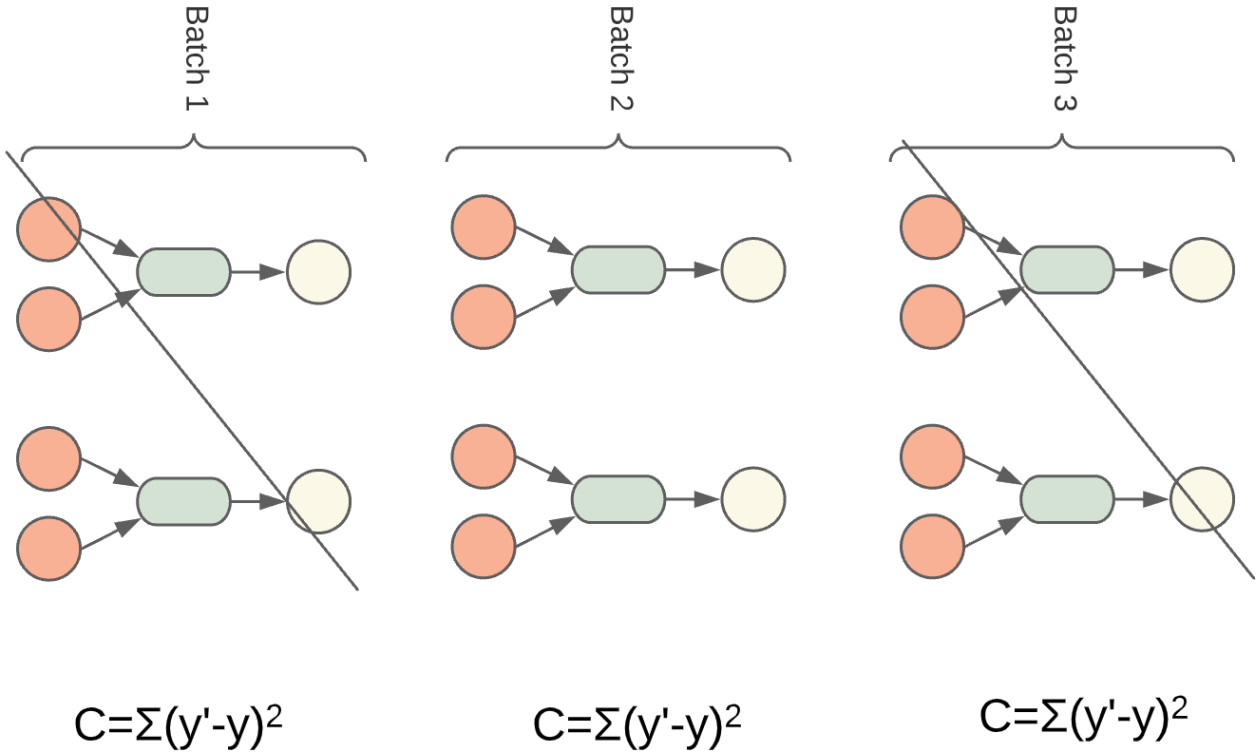


Fig. 12 Zweite Batch

Begriffsdefinition:

- Batch: Eine Gruppe von Trainingsdaten innerhalb des Datensatzes
- Epoche: Alle Batches wurden einmal durchlaufen
- Lernrate: Schrittgröße

Backpropagation

NN's wurden erst durch Backpropagation Leistungsstark. Dadurch erst war es möglich, das gesamte NN zu trainieren. Hier noch etwas Erklärung rund um Backpropagation und NN's allg. einfügen. Wie werden die Gewichte der vorherigen Schicht aktualisiert? Durch Backpropagation!

- verleiht den NN's ihre Leistungsfähigkeit
- verhalf zum Durchbruch von NN's
- Idee aus den 70ern
- vorher konnte man nur Teile eines Netzes trainieren

Problematik beim Lernen von mehrschichtigen NN's:

- Wie werden die Gewichte einer vorherigen Schicht aktualisiert?

Lösung: Backpropagation…

Die Gewichte des gesamten NN werden immer wieder aktualisiert, solange bis der Vorhersagewert so nah wie möglich am gewünschten Wert liegt. Wie das genau gemacht wird, soll in diesem Abschnitt gezeigt werden.

Ein grobes, einfaches Beispiel zur Backward-Propagation:

Mit einem einfachen, groben Beispiel soll der Einstieg in das Verständis der Backpropagation erleichtert werden. Die Mathematik hinter der "Backpropagation" ist für Nicht-Informatiker/-mathematiker teilweise nicht so einfach zu verstehen. Für den Einstieg wird daher auf ein grobes Rechenbeispiel zurückgegriffen. Es soll an dieser Stelle zunächst nur der grobe Vorgang der BP veranschaulicht werden.

- Es wird eine Vorhersage mit dem NN gemacht, diese Vorhersage \(\hat{y}\), weicht vom gewünschten / wahren Wert y ab
- Die Abweichung e (Error) ist ein Maß dafür, wie stark die Vorhersage vom wahren Wert abweicht
- Um die Abweichung zu minimieren müssen nun die Gewichte aktualisiert werden

Der Fehler e wird nun an die Ausgänge des Hidden-Layer transformiert:

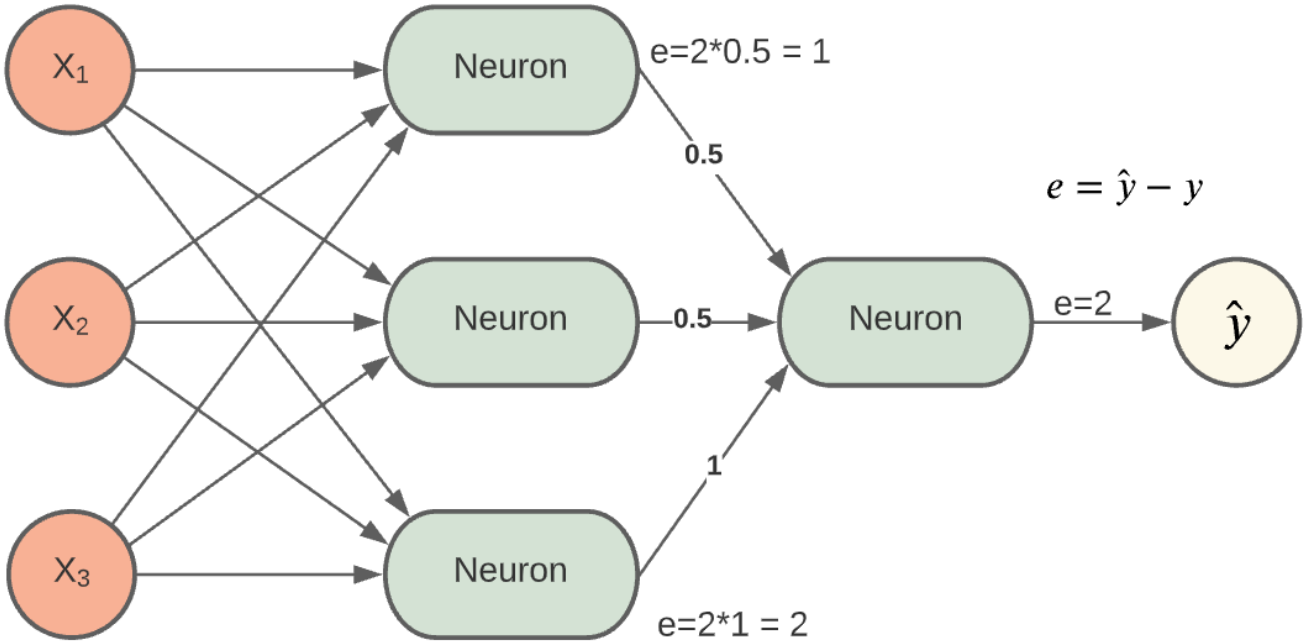


Fig. 13 Backpropagation

Initialisierung der Gewichte

Die Initialisierung kann darüber entscheiden, ob das NN trainieren kann oder nicht. Wie initialisieren wir die Gewichte? Mit Null wie im rechten Bild? Die Neuronen würden nur Nullen ausgeben. Das macht also keinen Sinn. Doch welche Werte soll man da am besten wählen? Weiterhin dürfen die Gewichte nicht alle mit den gleichen Werten initialisiert werden. Das ist auch aktiver Forschungsgegenstand, denn bei mehrschichtigen Netzen wird es umso wichtiger die Gewichte „richtig" bzw. nicht komplett falsch zu wählen. Besser ist es, den Gewichten unterschiedliche Werte zu geben. Das können zufällige, eher kleine Werte sein. So ist sichergestellt, dass jedes Neuron eine andere Funktion berechnet. Somit kann dann auch die Backpropagation richtig arbeiten und die Gewichte anpassen.

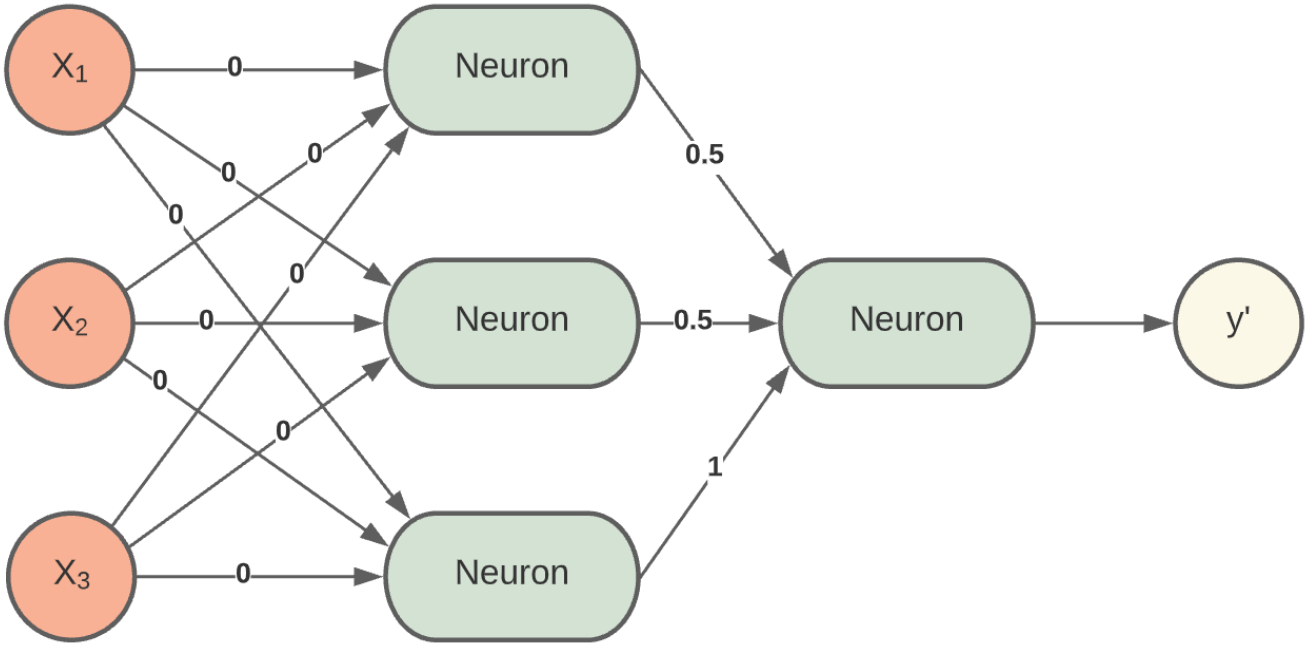


Fig. 14 Backpropagation2

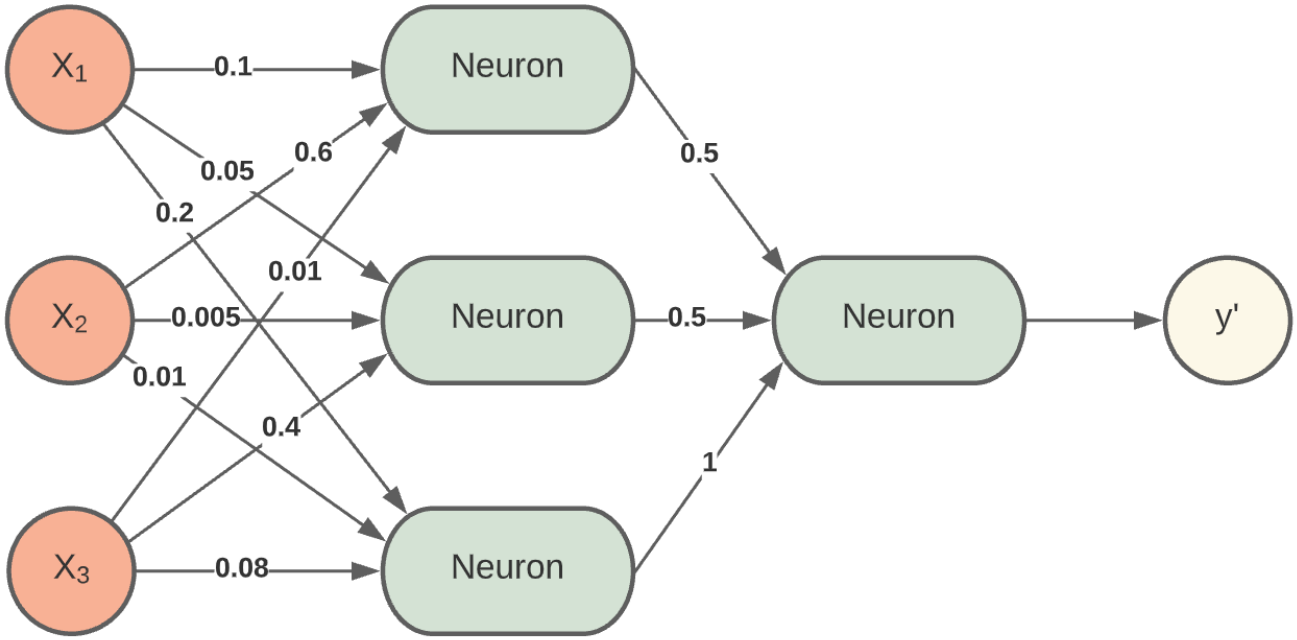


Fig. 15 Backpropagation3

Einfaches neuronales “from Scratch”

In diesem Kapitel wird ein Neuronales Netz erstellt, welches entscheiden kann ob ein bestimmtes Objekt auf dem Bild zu sehen ist oder etwas anderes. Dieses Netz kann also erstmal nur eine Gruppe z.B. Sechskantschraube von allen anderen unterscheiden.

Wir werden 28x28 Pixel große Bilder von Schraubenköpfen verwenden.

```
import numpy as np
from numpy import load
from scipy.special import expit
from sklearn.preprocessing import OneHotEncoder
import pickle
import matplotlib.pyplot as plt
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20476/1027057003.py in <module>
      1 import numpy as np
      2 from numpy import load
----> 3 from scipy.special import expit
      4 from sklearn.preprocessing import OneHotEncoder
      5 import pickle

ModuleNotFoundError: No module named 'scipy'
```

```
X_train = load('Dataset/X_train.npy').astype(np.float32).reshape(-1, 784)*1.0/255.0
y_train = load('Dataset/y_train.npy')


oh = OneHotEncoder()
y_train_oh = oh.fit_transform(y_train.reshape(-1, 1)).toarray()

X_test=load('Dataset/X_test.npy').astype(np.float32).reshape(-1, 784)*1.0/255.0
y_test=load('Dataset/y_test.npy')
```

```
y_train_oh.shape
```

```
(10500, 5)
```

```
print(X_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(10500, 784)
(4500, 784)
(4500,)
```

label check:

```
i=8
print(y_train[i])
print(y_train_oh[i])
plt.imshow(X_train[i].reshape(28,28)*255,cmap='gray',vmin=0,vmax=255)
plt.show
# 0: innensechskant
# 1: philips
# 2: pozidriv
# 3: sechskant
# 4: torx
```

```
print(X_train.shape)
print(X_test.shape)
```

```
(10500, 784)
(4500, 784)
```

einfaches NN in Python

Ein NN ohne Deep Learning Bibliotheken:

Genauigkeit interpretieren

Was sagt eine genauigkeit von 90% aus? Ist das gut oder schlecht?

> **Warning**
>
> **Beispiel:**
>
> - ja / nein
> - 10% / 90%
>
> Wenn das Modell immer “Nein” sagt wird es eine Genauigkeit von 90% erreichen obwohl es alle “ja-Beispiele” nicht erkannt hat.

NN1 “Einer gegen Alle”

```
import numpy as np
from numpy import load
#from scipy.special import expit
from sklearn.preprocessing import OneHotEncoder
import pickle
import matplotlib.pyplot as plt
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_18516/1473354027.py in <module>
      2 from numpy import load
      3 #from scipy.special import expit
----> 4 from sklearn.preprocessing import OneHotEncoder
      5 import pickle
      6 import matplotlib.pyplot as plt

ModuleNotFoundError: No module named 'sklearn'
```

In diesem Kapitel wird ein Neuronales Netz erstellt, welches entscheiden kann ob ein bestimmtes Objekt auf dem Bild zu sehen ist oder etwas anderes. Dieses Netz kann also erstmal nur eine Gruppe z.B. Sechskantschraube von allen anderen unterscheiden.

Wir werden 28x28 Pixel große Bilder von Schraubenköpfen verwenden.

```
#import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
X_train = load('Dataset/X_train.npy').astype(np.float32).reshape(-1, 784)*1.0/255.0
y_train = load('Dataset/y_train.npy').astype(np.int32)

X_test=load('Dataset/X_test.npy').astype(np.float32).reshape(-1, 784)*1.0/255.0
y_test=load('Dataset/y_test.npy').astype(np.int32)

y_train_A = y_train == 3 #sechskant soll erkannt werden
y_train_B = y_train == 2 #pozidriv soll erkannt werden
```

```
X_train.shape
```

```
(10500, 784)
```

```
i=0
print(y_train_A[i])
#print(X_train[i].reshape(28,28))
plt.imshow(X_train[i].reshape(28,28)*255.0,cmap='gray',vmin=0,vmax=255)
plt.show
# 0: innensechskant
# 1: philips
# 2: pozidriv
# 3: sechskant
# 4: torx
```

Neuronales Netz A (Sechskant)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(16, activation="sigmoid", input_shape=(784,)))
model.add(Dense(1, activation="sigmoid"))

#sgd = stochastic gradient descent
model.compile(optimizer="sgd", loss="binary_crossentropy", metrics=["accuracy"])


####################
model.fit(
    X_train,
    y_train_A,
    epochs=20,
    batch_size=500)
```

Neuronales Netz B (Pozidriv)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(16, activation="sigmoid", input_shape=(784,)))
model.add(Dense(1, activation="sigmoid"))

model.compile(optimizer="sgd", loss="binary_crossentropy", metrics=["accuracy"])

#################
model.fit(
    X_train,
    y_train_B,
    epochs=10,
    batch_size=500)
```

```
model.evaluate(X_train.reshape(10500, 784), y_train)
```

```
model.evaluate?
```

```
print(model.metrics_names)
```

NN 2: Mehrere unterscheiden

import os os.environ[“CUDA_VISIBLE_DEVICES”] = “-1”

```
# Vorstellung: MNIST-Daten!
# http://yann.lecun.com/exdb/mnist/
# FashionMNIST: https://github.com/zalandoresearch/fashion-mnist

import gzip
import numpy as np
from tensorflow.keras.utils import to_categorical

from numpy import load
import matplotlib.pyplot as plt

X_train = load('Dataset/X_train.npy').astype(np.float32).reshape(-1, 784)*1.0/255.0
y_train = load('Dataset/y_train.npy').astype(np.int32)

X_test=load('Dataset/X_test.npy').astype(np.float32).reshape(-1, 784)*1.0/255.0
y_test=load('Dataset/y_test.npy').astype(np.int32)

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_18584/1306288492.py in <module>
      5 import gzip
      6 import numpy as np
----> 7 from tensorflow.keras.utils import to_categorical
      8
      9 from numpy import load

ModuleNotFoundError: No module named 'tensorflow'
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
model = Sequential()

model.add(Dense(50, activation="sigmoid", input_shape=(784,)))
model.add(Dense(5, activation="sigmoid"))

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

```
X_train.reshape(10500, 784)
```

```
model.fit(
    X_train.reshape(10500, 784),
    y_train,
    epochs=80,
    batch_size=500)
```

```
model.evaluate(X_test.reshape(-1, 784), y_test)
```

```
model.predict(X_test.reshape(-1, 784))
```

Model testen

```
%matplotlib inline

import matplotlib.pyplot as plt

print(y_test[0])

plt.imshow(X_test[0].reshape(28,28), cmap="gray")
plt.show()
```

```
pred = model.predict(X_test.reshape(-1, 784))
```

```
import numpy as np
# Klasse mit höchster Wahrscheinlichkeit ausgeben:
np.argmax(pred[0])
```

```
4
```

```
np.argmax(pred, axis=1)
```

```
array([4, 2, 3, ..., 2, 1, 0])
```

Confusion Matrix

```
import pandas as pd
ytrue = pd.Series(np.argmax(y_test, axis= 1), name = 'ytrue')
ypred = pd.Series(np.argmax(pred, axis= 1), name = 'pred')
pd.crosstab(ytrue, ypred)
```

Convolutional Neural Networks (CNN)

Ein Theorem aus dem Jahr 1988, das “Universal Approximation Theorem”, sagt, dass jede beliebige, glatte Funktion, durch ein NN mit nur einem Hidden Layer approximiert werden kann. Nach diesem Theorem nach, würde dieses einfache NN bereits in der Lage sein jedes beliebige Bild bzw. die Funktion der Pixelwerte zu erlernen. Die Fehler und die lange Rechenzeit zeigen die Probleme in der Praxis. Denn um dieses Theorem zu erfüllen sind für sehr einfache Netze unendlich viel Rechenleistung, Zeit und Trainingsbeispiele nötig. Diese stehen i.d.R. nicht zur Verfügung. Für die Bilderkennung haben sich CNN’s als sehr wirksam erwiesen. Die Arbeitsweise soll in diesem Abschnitt erläutert werden. Der Grundgedanke bei der Nutzung der Convolutional Layer ist, dem NN zusätzliches “Spezialwissen” über die Daten zu geben. Das NN ist durch den zusätzlichen Convolutional Layer in der Lage, spezielle Bildelemente und Strukturen besser zu erkennen.

Es werden meist mehrere Convolutional Layer hintereinander geschalten. Das NN kann auf der ersten Ebene lernen, Kanten zu erkennen. Auf weiteren Ebenen lernt es dann weitere “Bild-Features” wie z.B. Übergänge, Rundungen o.ä. zu erkennen. Diese werden auf höheren Ebenen weiterverarbeitet.

Beispiel einer einfachen 1D-Faltung:

Die beiden einfachen Beispiele sollen die Berechnung verdeutlichen. Die Filterfunktion wird auf die Pixel gelegt und Elementweise multipliziert. Im folgenden Beispiel werden 3 Pixel eines Bildes verwendet. Die Ergebnisse sagen etwas über den Bildinhalt aus:

- positives Ergebnis: Übergang von hell zu dunkel

- negatives Ergebnis: Übergang von dunkel nach hell
- neutrales Ergebnis: Übergang wechselnd, hell-dunkel-hell oder dunkel-hell-dunkel

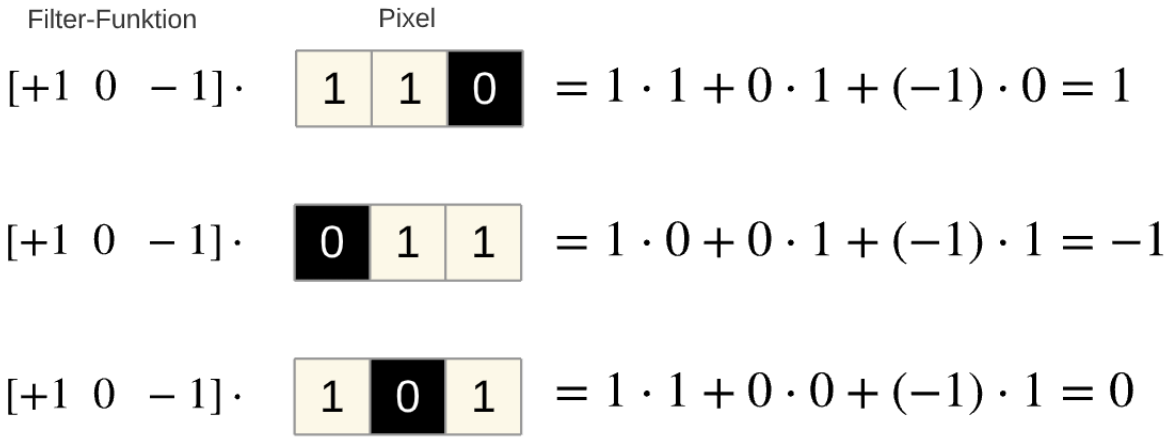


Fig. 16 Eindimensionale Faltung

Da ein Bild aus mehr als 3 Pixel besteht, muss die Filterfunktion über das gesamte Bild "geschoben" werden. Das folgende Beispiel demonstriert den Vorgang der Convolution im Fall eines eindimensionalen Filter. Der Filter besteht in diesem Fall wieder aus einem Zeilenvektor mit 3 Elementen. Der Filter wird nun Pixelweise über die Bildzeile geschoben, die Ergebnisse werden gespeichert und geben wiederum Aufschluss über die Bildstruktur. Die Ergebnisse zeigen wieder die enthaltene Bildstruktur:

- 1: hell-dunkel
- 0: hell-dunkel-hell
- 0: dunkel-hell-dunkel
- 1: hell-dunkel –1: dunkel-hell

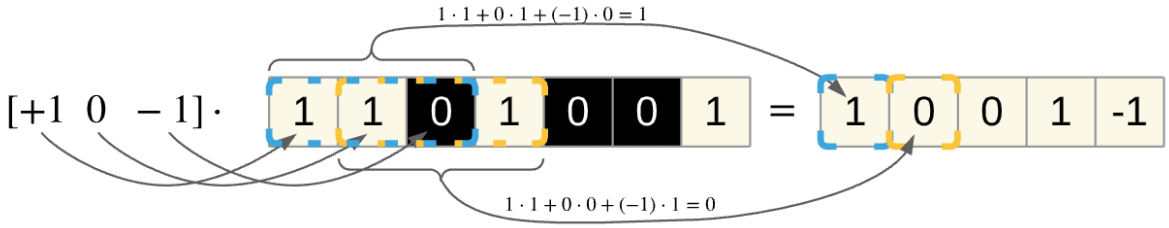


Fig. 17 Eindimensionale Faltung mit mehreren Übergängen

2-Dimensionale Faltung

In der Praxis werden in der Bilderkennung 2-dimensionale Filter verwendet, ein häufig verwendetes Format ist ein 3x3 Filter. Der Vorgang ist analog zum eindimensionalen Fall, der Filter wird über das gesamte Bild geschoben. Das folgende Beispiel zeigt einen Filter, der in der Lage ist, senkrechte Kanten zu erkennen.

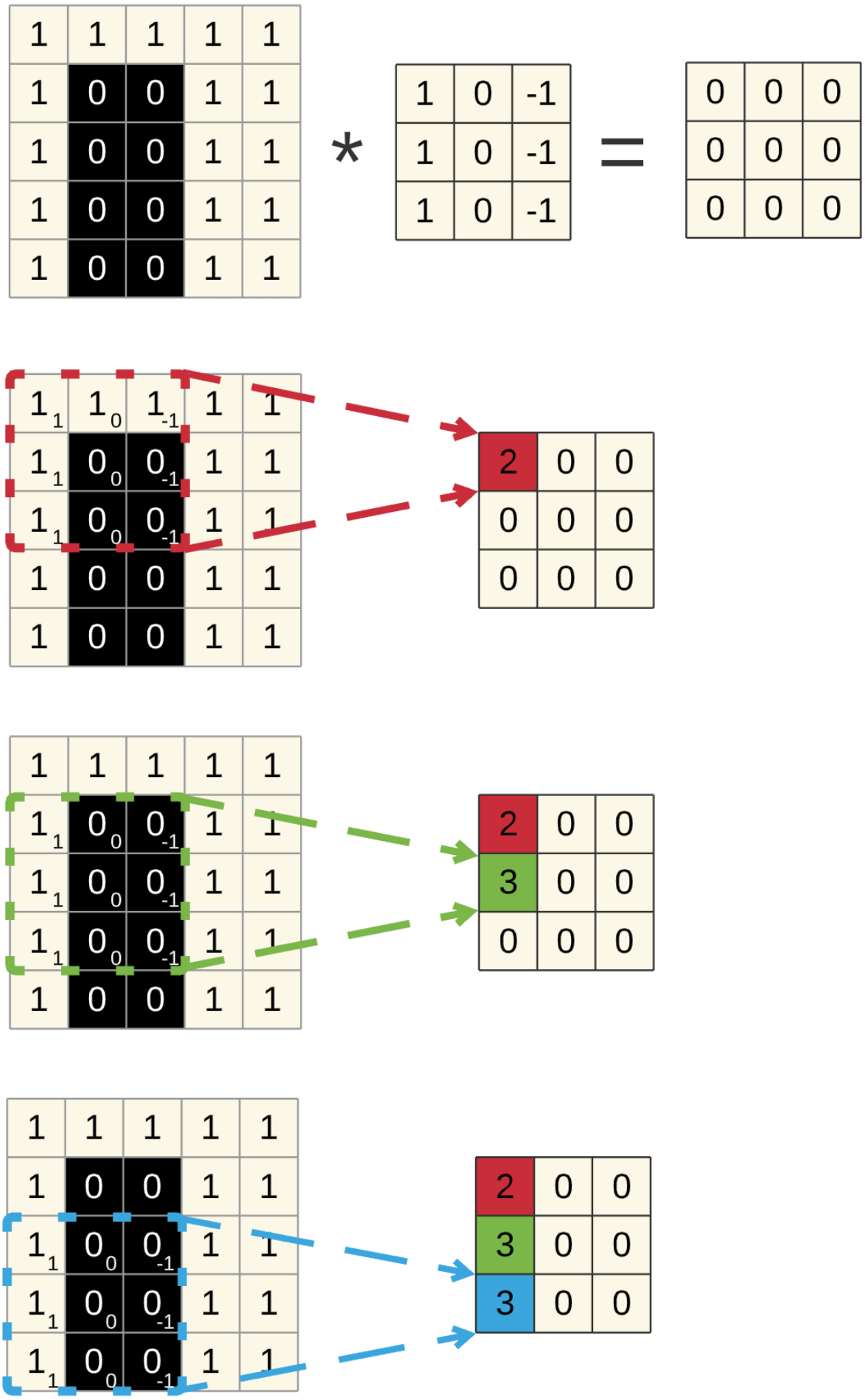


Fig. 18 Eindimensionale Faltung mit mehreren Übergängen

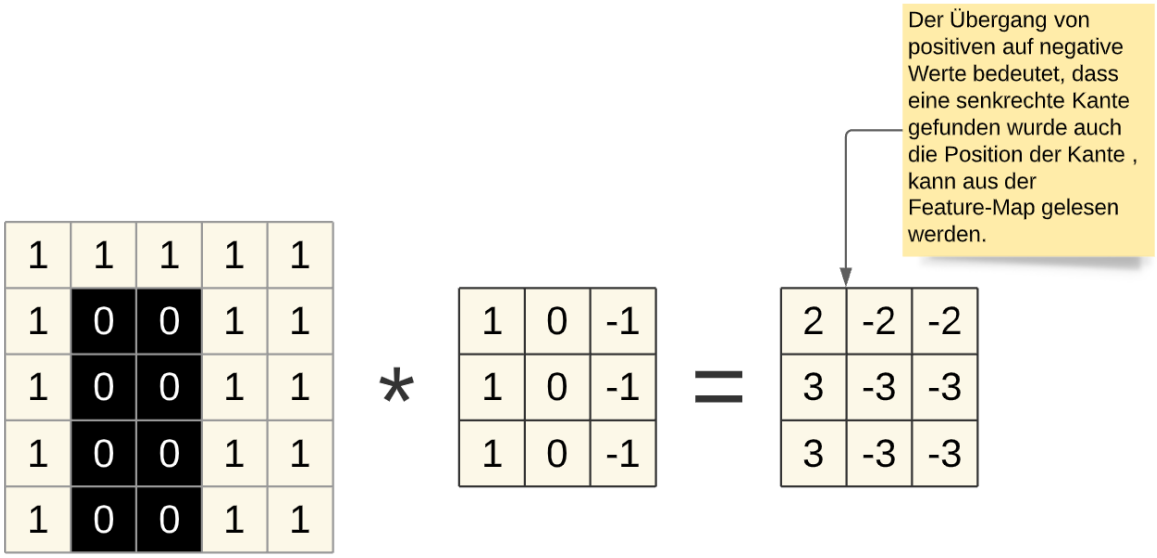


Fig. 19 Eindimensionale Faltung mit mehreren Übergängen

Die Werte der Filter bilden die Gewichte des Convolutional Layer. Diese Gewichte werden durch das Training selbst bestimmt und somit ist das CNN in der Lage, sich selbstständig auf relevante Features zu fokussieren.

Im folgenden noch weitere Ergebnisse für bestimmte Bildstrukturen:

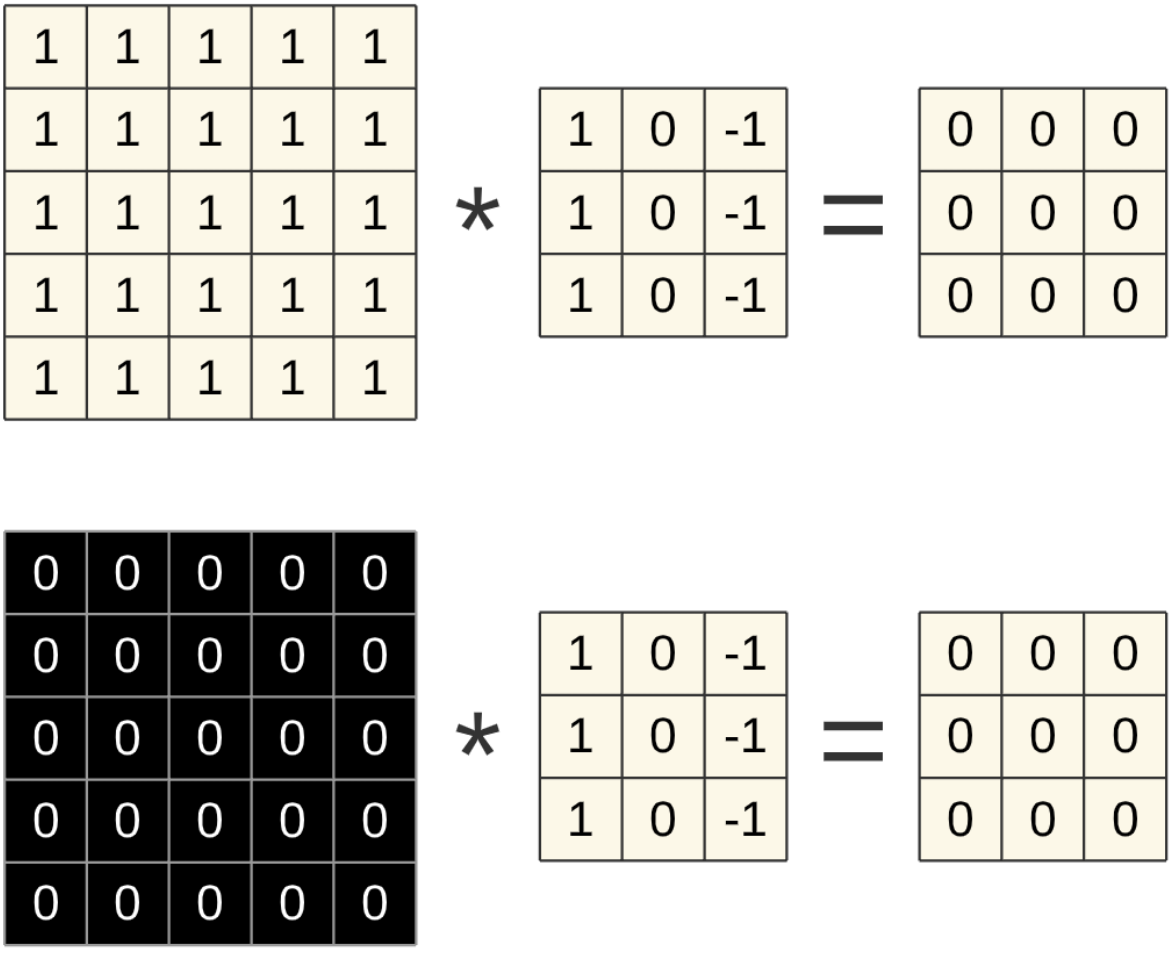


Fig. 20 Eindimensionale Faltung mit mehreren Übergängen

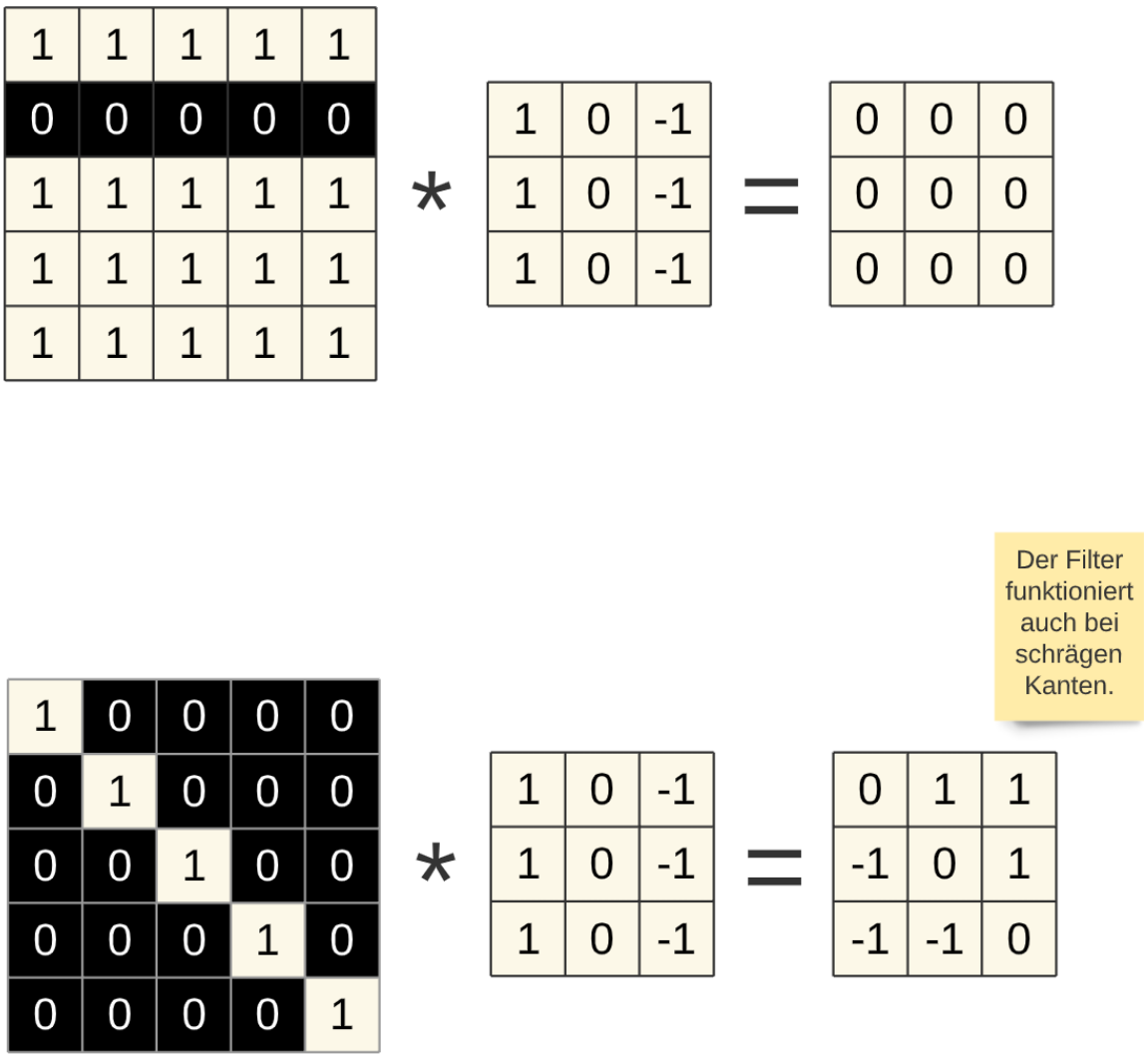


Fig. 21 Eindimensionale Faltung mit mehreren Übergängen

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

Mit Hilfe eines CNN-Layer bekommt das neuronale Netz ein “Verständnis” für Bilder “eingebaut”. Das NN ist somit auf die Erkennung von Bildern spezialisiert und demensprechend Leistungsfähiger als ein NN ohne dieses Bildverständnis.

- Kantenerkennung

Das CNN besitzt gegenüber dem neuronalem Netz eine Intuition darüber was ein Bild ist.

Das Neuronale Netz kann auf der ersten Ebene lernen, Kanten zu erkennen. Diese Ebene ist dann für die Kantenerkennung zuständig. Kante ist Kante egal wo auf dem Bild. Diese “Features” werden in den nachfolge Schichten weiterverarbeitet.

Beispiel einer einfachen Convolution:

https://medium.com/swlh/image-processing-with-python-convolutional-filters-and-kernels-b9884d91a8fd

Die Filter oder Kernels gibt man nicht vor sondern lässt die Werte vom Convolutional Layer ermitteln. Die Kernels werden dabei so bestimmt dass sie für das Problem am meisten Sinn machen.

Wir möchten nicht nur vertikale Kanten finden, sondern auch schräge und waagerechte. Da jeder Filter für ein bestimmtes Feature zuständig ist, benötigt das CNN mehrere solcher Filter um alle relevanten Zusammenhänge extrahieren zu können. Die Anzahl an Filter die wir bereitstellen hängt von den Daten ab und ist ein Hyperparameter den man tunen muss.

CNN mit Keras

Wir wollen nun ein CNN mit Keras entwickeln.

```
# Vorstellung: MNIST-Daten!
# http://yann.lecun.com/exdb/mnist/
# FashionMNIST: https://github.com/zalandoresearch/fashion-mnist

import gzip
import numpy as np
import numpy as np
from numpy import load
from tensorflow.keras.utils import to_categorical


X_train = load('../02_NN/Dataset/X_train.npy').astype(np.float32)#.reshape(-1, 784)
y_train = load('../02_NN/Dataset/y_train.npy')


#oh = OneHotEncoder()
#y_train_oh = oh.fit_transform(y_train.reshape(-1, 1)).toarray()

X_test=load('../02_NN/Dataset/X_test.npy').astype(np.float32)#.reshape(-1, 784)
y_test=load('../02_NN/Dataset/y_test.npy')

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
print(y_train)
```

```
print(X_train.shape)
```

```
(10500, 28, 28)
```

Das Format der Daten passt noch nicht zum geforderten Eingangsformat. Das CNN verlangt

Bei einem Wert am Ausgang zwischen 0 und 1 verwendet man “binary crossentropy”. Hat man mehrere Werte / Kategorien am Ausgang, dann verwendet man categorical crossentropy.

stochastic gradient descent

```
# CNN!

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten

model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation="relu", input_shape=(28, 28, 1)))
#model.add(Conv2D(32, kernel_size=(3, 3), activation="relu"))
model.add(Flatten())
model.add(Dense(5, activation="softmax"))

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

model.fit(
    X_train.reshape(10500,28,28,1),
    y_train,
    epochs=20,
    batch_size=500)
```

rmsprop

```
# CNN!

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten

model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation="relu", input_shape=(28, 28, 1)))
#model.add(Conv2D(32, kernel_size=(3, 3), activation="relu"))
model.add(Flatten())
model.add(Dense(5, activation="softmax"))

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

model.fit(
    X_train.reshape(10500,28,28,1),
    y_train,
    epochs=20,
    batch_size=500)
```

Two Conv2D Layer

```
# CNN!

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten

model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation="relu", input_shape=(28, 28, 1)))
model.add(Conv2D(32, kernel_size=(3, 3), activation="relu"))
model.add(Flatten())
model.add(Dense(5, activation="softmax"))

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

model.fit(
    X_train.reshape(10500,28,28,1),
    y_train,
    epochs=20,
    batch_size=500)
```

Pooling

Ein Pooling Layer, hat die Aufgabe, das CNN “toleranter” gegen geringe Abweichungen zu machen. Im folgenden Beispiel wurde die Kante stückweise um insgesamt 2 Pixel nach rechts verschoben. Auf einem Bild sollte es für das Erkennen keine so große Rolle spielen aber für den Convolutional-Layer spielt es eine Rolle in den Ergebnissen. Es ist nicht tolerant gegenüber Verschiebungen.

Jede Verschiebung der senkrechten Kante, erzeugt eine andere Featur-Map:

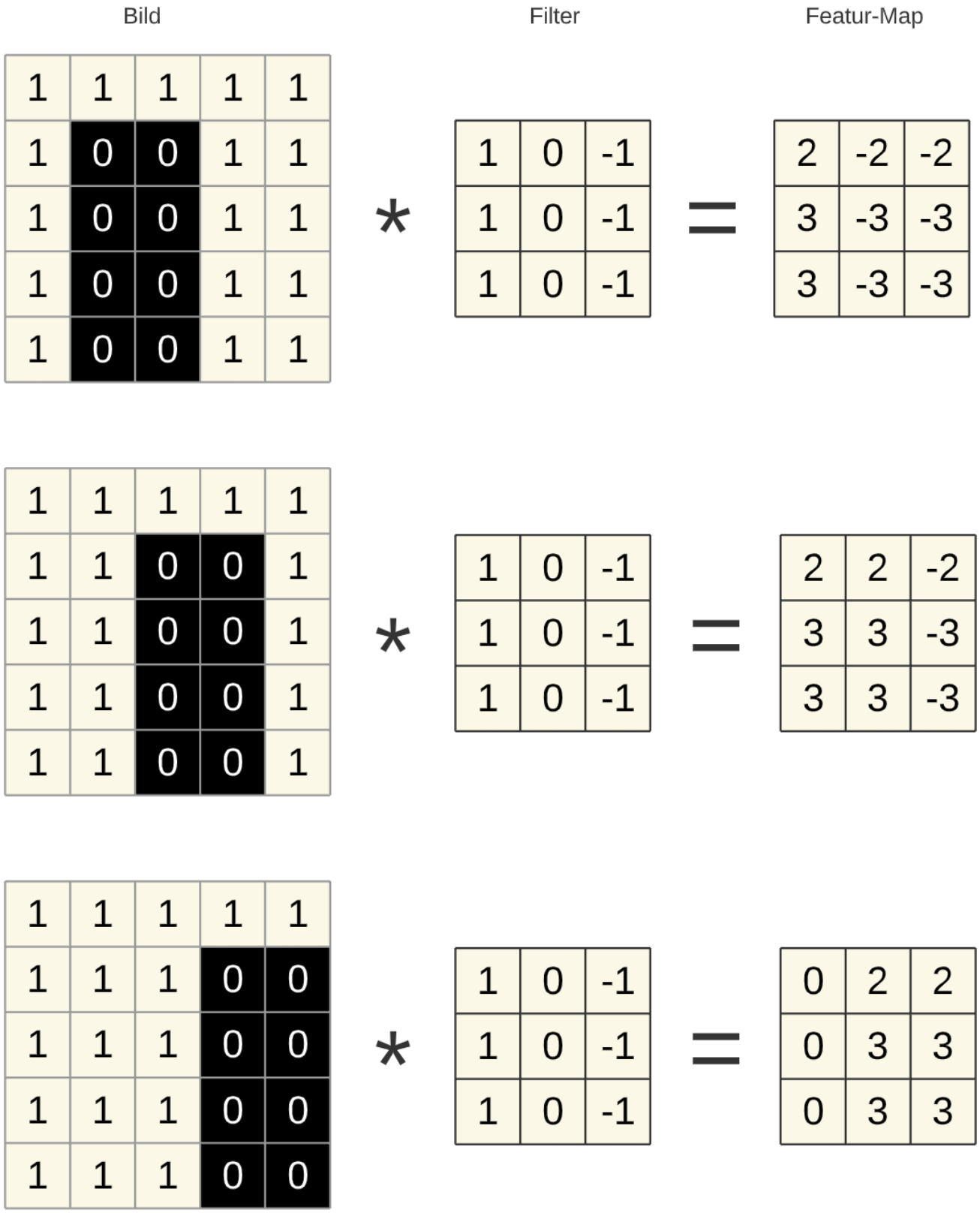


Fig. 22 Kantenerkennung ohne Pooling.

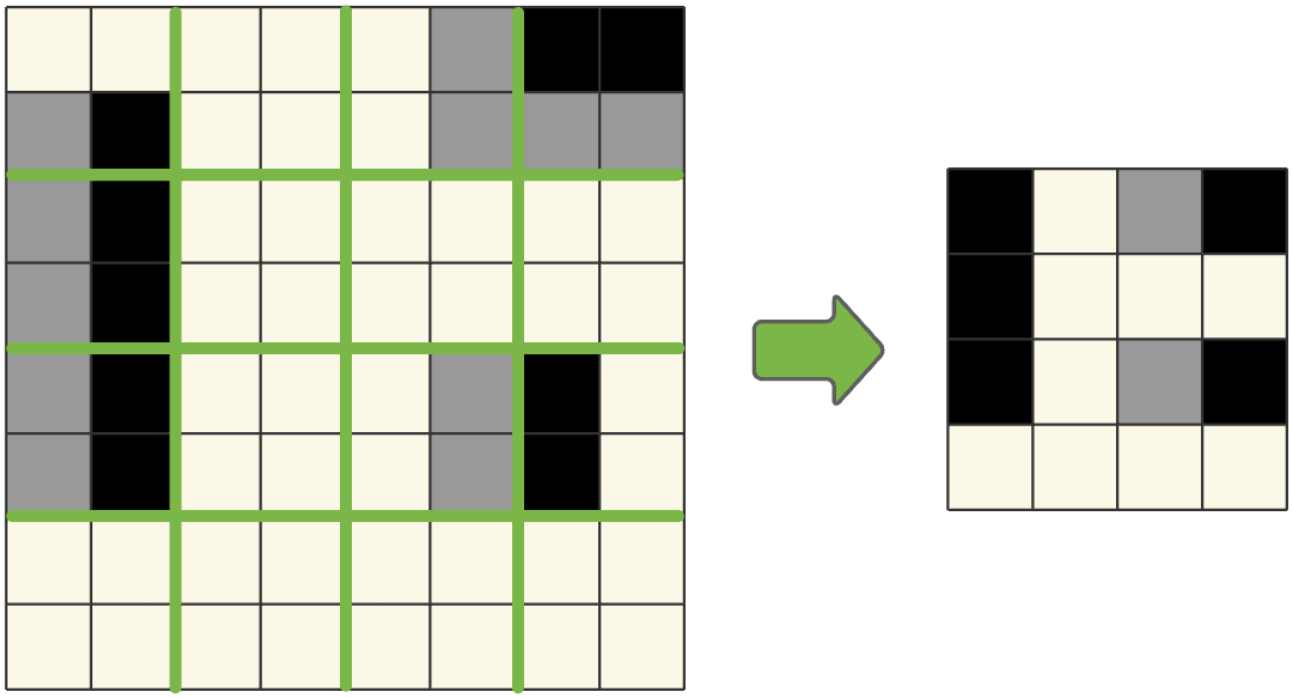


Fig. 23 Kantenerkennung mit Pooling.

Vorteile vom Pooling:

- bessere Generalisierung
- geringere Anzahl an Ausgabeknoten durch Dimensionsreduktion
- lernen geht schneller

CNN1

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

```
# Vorstellung: MNIST-Daten!
# http://yann.lecun.com/exdb/mnist/
# FashionMNIST: https://github.com/zalandoresearch/fashion-mnist

import gzip
import numpy as np
from numpy import load
from sklearn.preprocessing import OneHotEncoder
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
```

```
X_train = load('Dataset/X_train.npy').astype(np.float32).reshape(-1, 28,28,1)
y_train = load('Dataset/y_train.npy')

X_test=load('Dataset/X_test.npy').astype(np.float32).reshape(-1,28,28,1)
y_test=load('Dataset/y_test.npy').astype(np.int32)

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
print(X_train.shape)
y_train.shape
```

```
(10500, 28, 28, 1)
```

```
(10500, 5)
```

```
y_train[0]
```

```
array([0., 0., 0., 0., 1.], dtype=float32)
```

```
model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation="relu", input_shape=(28,28,1,)))
model.add(Conv2D(32,(3,3),activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(5, activation="softmax"))

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

model.fit(
    X_train,
    y_train,
    epochs=15,
    batch_size=500)
```
