(Künstliche) Neuronale Netze
(Artificial) Neural Networks

Christian Wilms

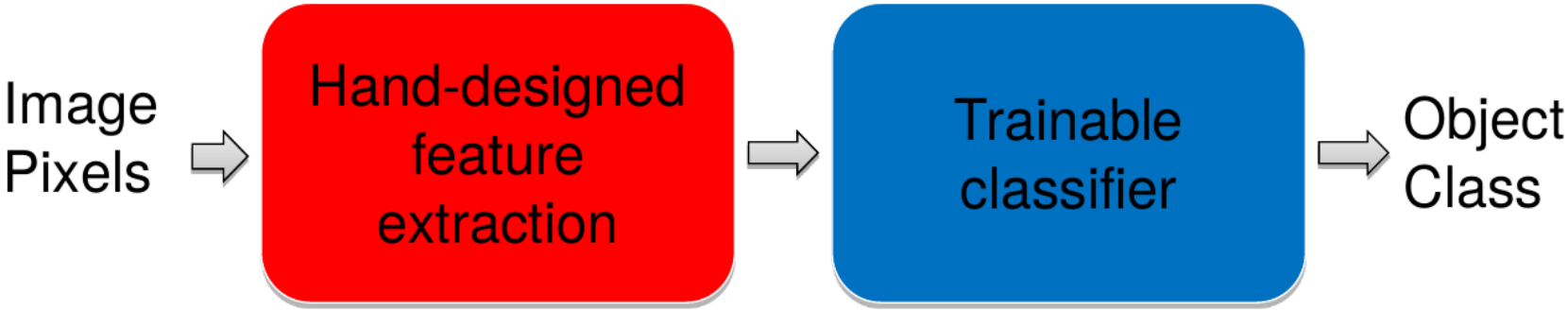
Computer Vision Group
Universität Hamburg

Wintersemester 2023/24

29. November 2023

Übersicht

Pipeline der Klassifikation

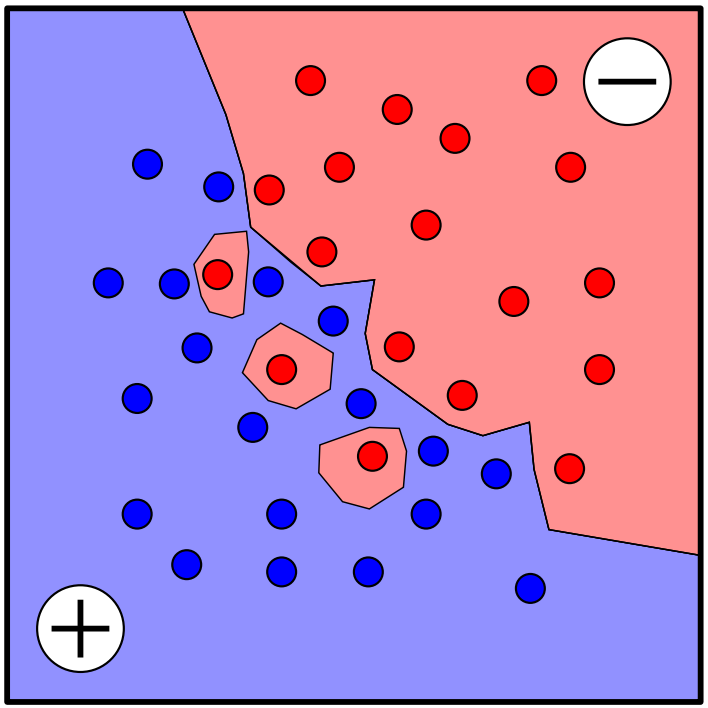


- Mittelwert
- *Standardabweichung*
- Histogramme
- Exzentrizität
- *HOG*
- ...

- Nächster-Nachbar-Klassifikator
- *k-Nächster-Nachbar-Klassifikator*

Bisher: Nächster-Nachbar-Klassifikator

Prinzip: Klassifikation auf Basis ähnlicher Nachbarn



Trainingsdaten mit zwei Merkmalen

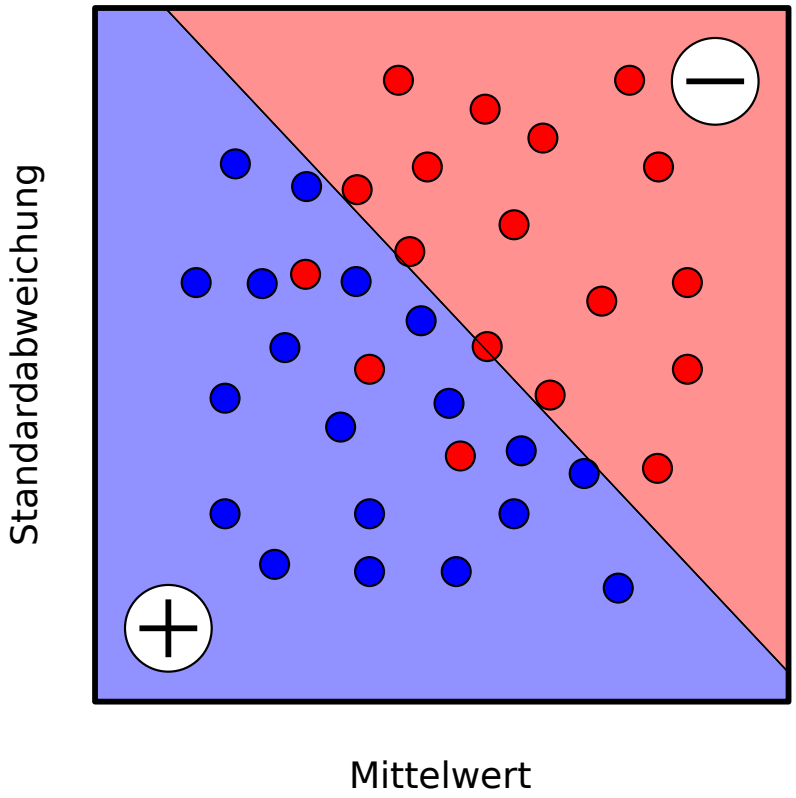
Entscheidungsgrenze:
Beliebige Form, implizit definiert

Nachteile

- viel Speicher
- hohe Laufzeit
- mangelnde Generalisierung
- Einfluss von Ausreißer

Wie kann es effizienter und robuster gehen?

Linearer Klassifikator

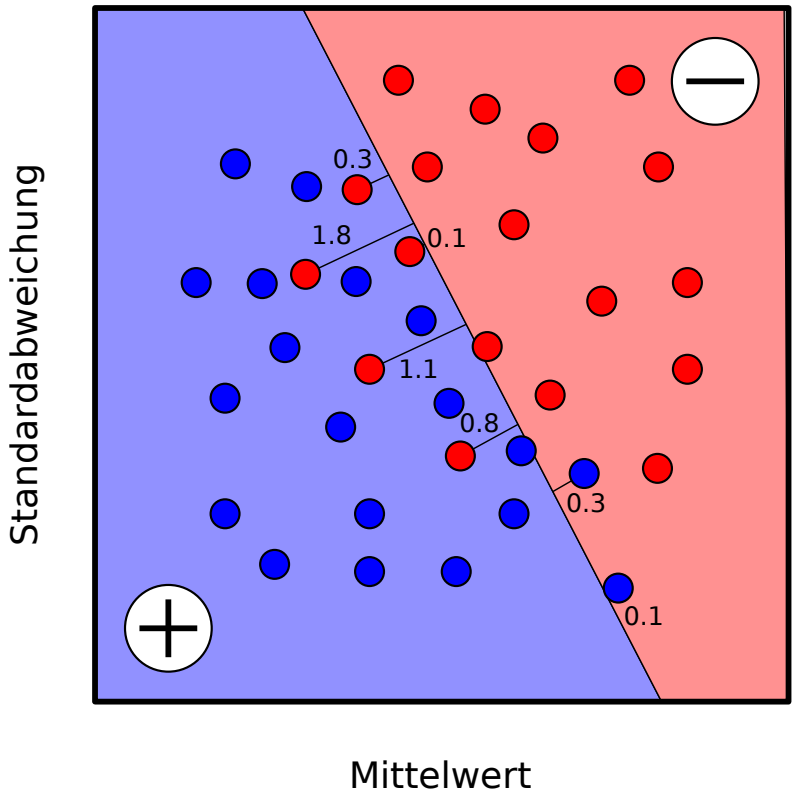


Lineare Funktion als Klassifikator:

$$y = f(\vec{x}) = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

- x_1 und x_2 sind Merkmale
- w_1, w_2 und b sind Gewichten
- Vorzeichen von y als Vorhersage
- Entscheidungsgrenze liegt bei $y = 0$

Welche Entscheidungsgrenze ist am besten?



Idee: Fehler bei der Klassifikation auf Trainingsdaten bestimmen

Loss-Funktion

- bewertet die Entscheidungsgrenze
- Fehler erhöhen den Loss
- kleiner Loss $\rightarrow$ gute Entscheidungsgrenze

Wie kommt man auf eine gute Entscheidungsgrenze?

Prinzip: Optimierung der Entscheidungsgrenze von Initialzustand

Woran wir drehen können?

Gewichte w_1, w_2 und b

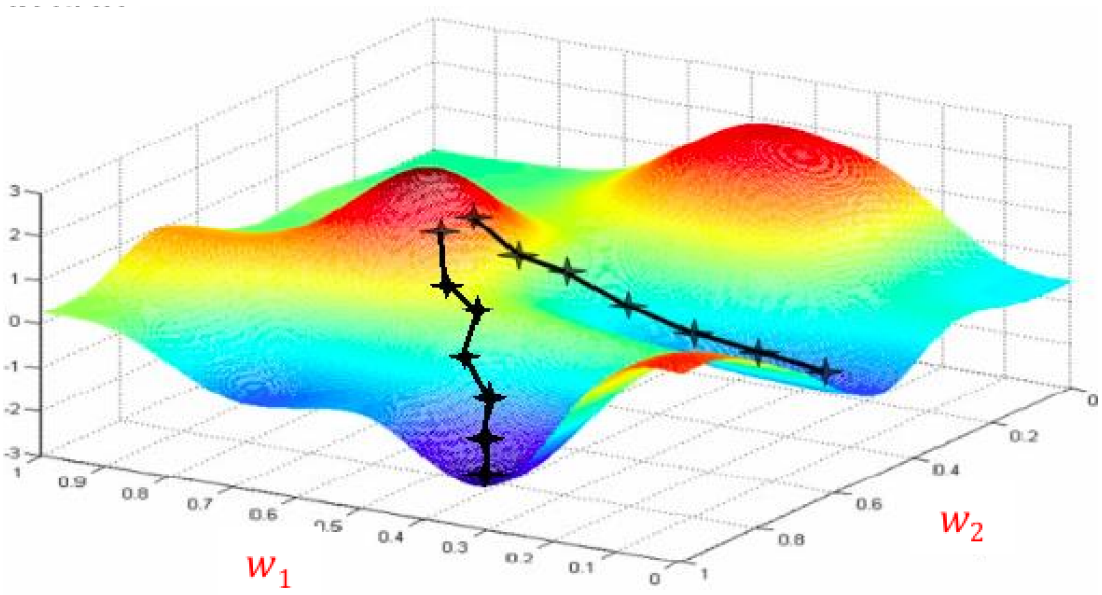
Wie können wir die Gewichte optimieren?

1. w_1, w_2 und b zufällig initialisieren
2. für ein Trainings-Bild Klassifikation prüfen
3. bei falscher Klassifikation Loss ermitteln
4. w_1, w_2 und b anpassen mit Gradientenabstieg
5. zurück zu 2

Wie funktioniert der Gradientenabstieg?

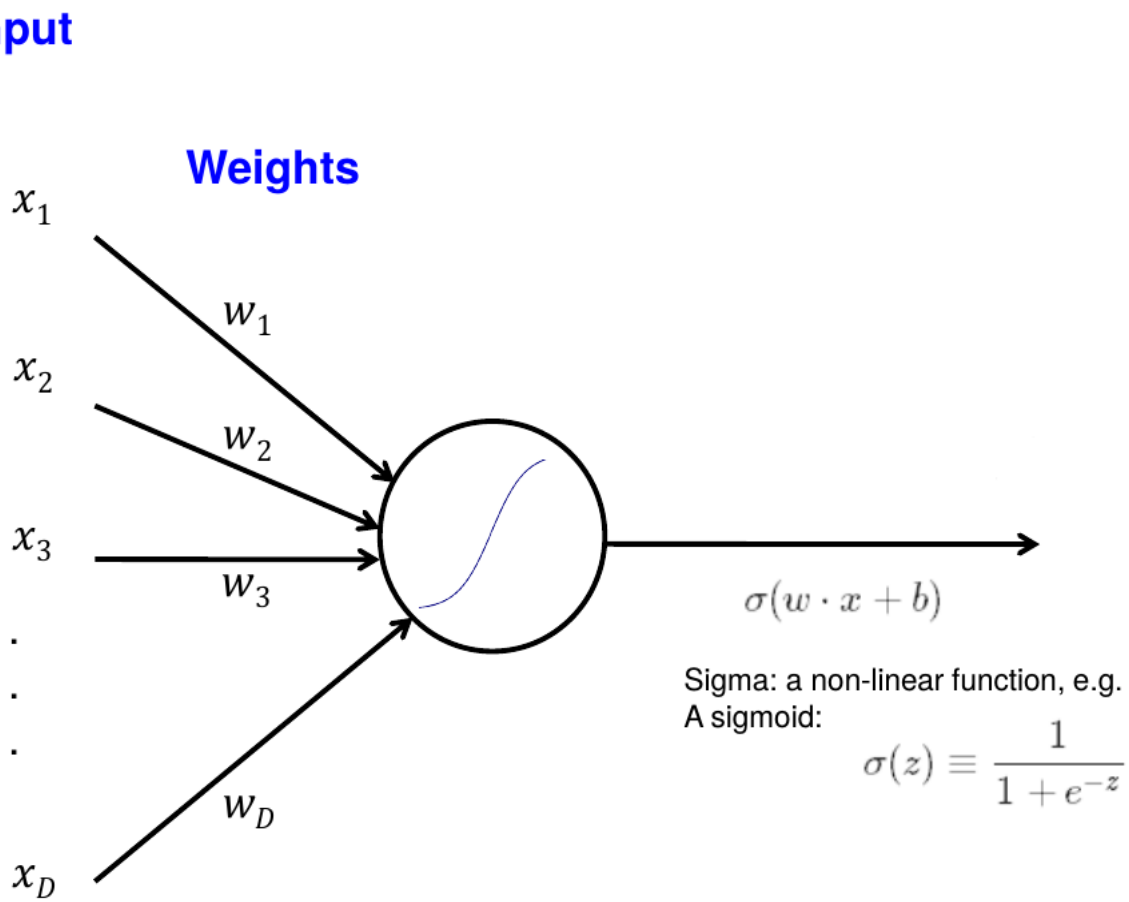
Gradientenabstiegsverfahren

- Loss-Funktion ist abh. von Gewichten
- partielle Ableitungen der Loss-Funktion bilden $\rightarrow$ Gradient
- den Gradient ein Stück entlang gehen
- dort w_1, w_2 und b neu bestimmen, die besser passen



Geht das auch mit mehr als zwei Merkmalen?

Ja, die Anzahl der w_i steigt dabei entsprechend. $\Rightarrow$ **Neuron**



Aktivierungsfunktion (σ) für nicht-lineare Abhängigkeiten

Übersicht

Wie kann man damit komplexere Entscheidungsgrenzen bauen?

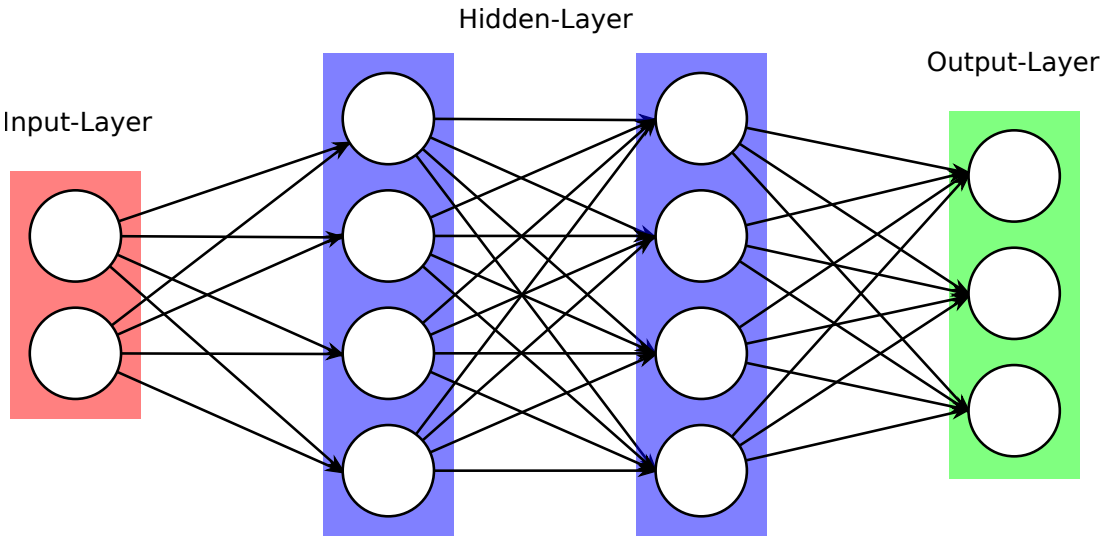
Basisprinzip Neuronales Netz

- Neuronen in Schichten anordnen
- mehrere Schichten hintereinander
- Neuronen zwischen Schichten verbinden
- Netze mit tausenden Neuronen/Millionen von Gewichten möglich

$\Rightarrow$ Komplexität der Entscheidungsgrenze ist abh. von der Anzahl der Neuronen im Netz

Eine interaktive Visualisierung findet ihr hier

Struktur eines Neuronalen Netzes



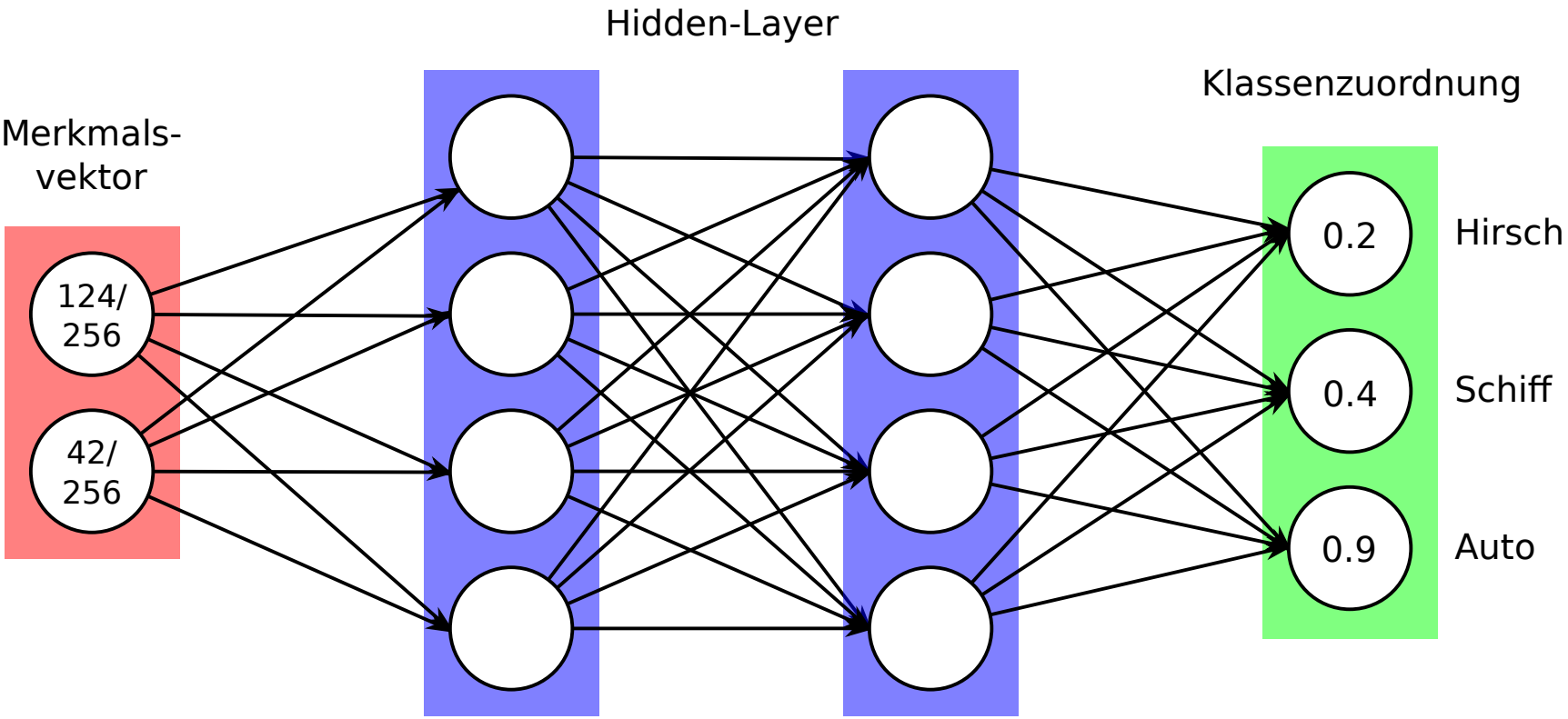
Input Layer repräsentieren die Merkmale (Eingabe)

Hidden Layer kombinieren die Eingaben zu neuen Merkmalen

Output Layer fassen die obersten Merkmale zu einer Wahrscheinlichkeit pro Klasse zusammen

Alle Layer sind hier fully-connected (`Dense`)!

Beispiel


Hidden-Layer
Klassenzuordnung
Merkmals-
vektor
124/
256
42/
256
0.2
Hirsch
0.4
Schiff
0.9
Auto

Wie können Neuronale Netze trainiert werden?

Ähnlich zu einem Neuron, aber die Gradienten/der Fehler wird durch das Netz zurück propagiert.
$\Rightarrow$ **Backpropagation**

1. Initialisierung aller Gewichte mit zufälligen Werten.
2. Trainingsbild/**Batch** von Bildern durch das Netzwerk schicken
3. Loss berechnen
4. partielle Ableitungen der Loss-Funktion bestimmen
5. alle Gewichte entspr. der partiellen Ableitungen updaten

Üblicherweise werden die Trainingsdaten mehrfach durch das Netz geschickt, ein Durchgang wird dabei **Epoche** genannt.

Übersicht

Muss ich das alles selber implementieren?

NEIN!

Bibliotheken

Tensorflow Bibliothek für Neuronale Netze in Python

Keras Paket in Tensorflow, das die Nutzung vereinfacht

Technische Voraussetzungen

- je tiefer das Netz, umso länger dauert alles
- kleine Netze kann man auf der CPU rechnen, große auf der GPU

Wie baut man mit Keras ein Netz?

- leeres Model erzeugen
- Layer wie bei einer Liste hinzufügen
- Model kompilieren

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(128, activation='relu', input_shape
    =(2,)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='Adam', metrics=['accuracy'])
```

Dense-Layer in Keras (fully-connected Layer)

`Dense(units, activation, input_shape)`

`units`	Anzahl der Neuronen im Layer
`activation`	Aktivierungsfunktion für alle Neuronen des Layers, meist ReLU oder Softmax
`input_shape`	Form der Daten, die in den Layer kommen (nur erster Layer)

Beispiel

```
from tensorflow.keras.layers import Dense
model.add(Dense(128, activation='relu', input_shape
    =(2,)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax')) #Softmax
    im letzten Layer
```

Model kompilieren

Das Model gebaut und die Layer auf Kompatibilität geprüft.

```
model.compile(loss, optimizer, metrics)
```

`loss` Loss-Funktion, meist Categorical Crossentropy

`optimizer` Verfahren zum Gradientenabstieg (hier Adam)

`metrics` Model direkt evaluieren (Accuracy = Trefferquote)

Beispiel

```
>>> model.compile(loss=
                  'sparse_categorical_crossentropy',
                  optimizer='Adam',
                  metrics=['accuracy'])
```

Wie müssen die Daten für das Netz aussehen?

Merkmalsvektoren

- ein Array mit allen Merkmalsvektoren
- je Zeile ein Merkmalsvektor
- Shape: (Anz. Bilder $\times$ Anz. Merkmale im Vektor)

Labels

- ein 1D-**Array** für alles Labels
- Labels müssen IDs der Klassen sein $(0, 1, 2, \dots)$

Wie trainiert man ein Netz in Keras?

`model.fit(x, y, batch_size, epochs)`

- `x` Merkmalsvektoren der Trainingsbilder als Array
- `y` Label der Trainingsbilder als one-hot-encoding
- `batch_size` Größe eines Batches (Trainingsbilder für die zusammen der Loss berechnet wird)
- `epochs` Anzahl der Epochen

Beispiel

```
>>> model.fit(X_train, Y_train, batch_size=1, epochs
    =10)
Epoch 1/10
60/60 [==================] - 0s 4ms/sample - 7.1515 -
     acc: 0.4167
```

Keras kann auch direkt evaluieren!

`model.evaluate(x, y)`

- `x` Merkmalsvektoren der Testbilder als Array
- `y` Label der Testbilder als one-hot-encoding

Die Rückgabe ist ein Tupel aus Loss und Trefferquote über die Testdaten.

Beispiel

```
>>> score = model.evaluate(X_test, Y_test)
30/30 [==============================] - 0s 1ms/
    sample - loss: 2.3285 - acc: 0.4667
```

Übersicht

Klassifikation: Nächster-Nachbar-Klassifikator vs. Neuron/Perceptron

- Erklärung: A Complete Guide to K-Nearest-Neighbors with Applications in Python and R
- Erklärung/Beispiel: scikit-learn - 1.6.2. Nearest Neighbors Classification
- [GW]: Kapitel 12.5 Neural Networks and Deep Learning
  - The Perceptron (Anm.: Der preceptron algorithm to learn a decision boundary ist eine Vereinfachung der in den Folien behandelten Backpropagation, indem als Loss-Funktion nur $y - t$ genutzt wird. Dann muss allerdings das Voreziechen des Updates selbst bestimmt werden, s. Gl. 12-40 und 12-41 in [GW].
- Erklärung: What the Hell is Perceptron?
  Anmerkung: Wir haben die Step Function weggelassen und erhalten so ein Neuron.

Neuronale Netze

- [GW]: Kapitel 12.5 Neural Networks and Deep Learning
  - Multilayer Feedforward Neural Networks
  - Forward Pass Through a Feedforward Neural Network
- Visualisierung: Tinker With a Neural Network
- Video: SIMPLE EXPLANATION Of Multilayer Perceptron - Beginners Guide to Neural Networks - Oscar Alsing @ YouTube
  Anmerkung: Nutzt und erklärt sehr gut die vorher genannte Visualisierung.
- Video: Neural Networks Demystified [Part 1: Data and Architecture] - Welch Labs @ YouTube
- Video: But what *is* a Neural Network? — Chapter 1, deep learning - 3Blue1Brown @ YouTube

Gradientenabstieg

- Erklärung/Beispiel: Machine Learning Crash Course - Reducing Loss
- Video: Neural Networks Demystified [Part 3: Gradient Descent] - Welch Labs @ YouTube
- Video: Gradient descent, how neural networks learn — Chapter 2, deep learning - 3Blue1Brown @ YouTube

Training Neuronaler Netze/Backpropagation

- [GW]: Kapitel 12.5 Neural Networks and Deep Learning
  - Using Backpropagation to Train Deep Neural Networks
- Visualisierung: Tinker With a Neural Network
- Vertiefung: A Friendly Introduction to Cross-Entropy Loss
- Video: Neural Networks Demystified [Part 4: Backpropagation] - Welch Labs @ YouTube
- Video: What is backpropagation really doing? — Chapter 3, deep learning - 3Blue1Brown @ YouTube
- Video: Backpropagation calculus — Appendix to deep learning chapter 3 - 3Blue1Brown @ YouTube
- Beispiel: A Step by Step Backpropagation Example

Keras

- Erklärung/Beispiel: Keras - Getting started with the Keras Sequential model
  Anmerkung: Bei den Beispielen nur bis MLP for binary classification, als Optimierer wurde dort `rmsprop` statt `Adam` benutzt.
- Erklärung/Beispiel: Keras: Deep Learning for humans mit Getting started: 30 seconds to Keras
- Erklärung/Beispiel: Develop Your First Neural Network in Python With Keras Step-By-Step
  Anmerkung: Dort wurde als Aktivierungsfunktion im letzten Layer `binary_crossentropy`, da es nur zwei Klassen gibt.

Referenzen I

[GW], R. Gonzalez und R. Woods
Digital Image Processing
4th ed., Global Edition, Pearson, 2018.
siehe Moodle