

Praktikum Machine Learning

Versuch 6: Neuronale Netze und Deep Learning

Inhaltsverzeichnis

1	Einführung	2
2	Kurzer Einblick in Python	2
2.1	Installation von Python und TensorFlow	2
2.2	Interaktive Nutzung von Python	3
2.3	Python Programme	5
2.4	Beispielprogramm	6
3	Machine Learning mit Python	8
3.1	Lineare Regression mit NumPy	8
3.2	Lineare Regression mit Scikit-Learn	10
4	Einstieg in TensorFlow	11
4.1	Variablen und Platzhalter	13
4.2	Lineare Regression mit TensorFlow	15
5	Feedforward-Netzwerke in TensorFlow	17
5.1	Softmax-Regression	18
5.2	Vollständiges Feedforward-Netzwerk	21

Lernziele

Nachdem Sie diesen Versuch durchgearbeitet haben,

- haben Sie einen ersten Einblick in die Programmiersprache Python.
- können Sie einfache Machine-Learning-Problemstellungen mit Hilfe von Python und der Bibliothek NumPy lösen.
- können Sie graphische Darstellung mit der Bibliothek Matplotlib erstellen.
- können Sie einfache neuronale Netze mit dem Framework TensorFlow trainieren und anwenden.

1 Einführung

In diesem Versuch implementieren wir Neuronale Netze mit Hilfe der Programmiersprache Python und dem Framework TensorFlow. Wir beginnen mit einer kurzen Einführung der Programmiersprache Python und werden dabei sehen, dass sich Aufgabenstellungen im Bereich Machine Learning sehr gut mit Python lösen lassen. Python-Bibliotheken, wie z.B. NumPy, SciPy oder Scikit-Learn, vereinfachen die Implementierung von Machine Learning Algorithmen weiter. Im Gegensatz zu Matlab hat Python und alle hier besprochenen Erweiterungen den Vorteil, dass sie frei verfügbar sind. Nach einem Einstieg in TensorFlow implementieren wir einfache Feed-forward Netzwerke für die Bilderkennung. Die Hinweise zur Installation von Python und zur Ausführung von Python-Programmen beschränken sich hier auf Windows. Natürlich kann Python und TensorFlow auch mit Linux und Mac OS verwendet werden. Die meisten Code-Beispiele, die im Folgenden besprochen werden, finden Sie auch im zum Versuch gehörenden zip-File. Somit können Sie diese auf einfache Weise selbst testen, modifizieren oder erweitern.

2 Kurzer Einblick in Python

Python ist eine einfach zu erlernende und sehr übersichtliche Programmiersprache, die sich derzeit einer sehr großen Beliebtheit erfreut. Als universelle Programmiersprache hat Python sehr viele Anwendungsgebiete. Im Bereich Machine Learning ist Python derzeit wohl die am weitesten verbreitete Programmiersprache.

Im Gegensatz zu C und C++ ist Python eine interpretierte Sprache. Der Quellcode wird also nicht vorab von einem Compiler und einem Linker in eine ausführbare Datei übersetzt, sondern die Befehle werden nach und nach während der Programmlaufzeit von einem Interpreter abgearbeitet. Damit ist Python sehr gut für die interaktive Arbeit geeignet. Allerdings wird dieser Vorteil der Flexibilität durch eine etwas geringere Effizienz im Vergleich zu compilierten Sprachen erkauft. Leistungsfähige Bibliotheken machen diesen Nachteil aber weitgehend wett.

Ein weiterer wichtiger Unterschied zu C/C++ ist, dass in Python keine Variablendeklarationen notwendig sind. In C/C++ hat jede Variable einen bestimmten Datentyp, der während der Lebenszeit der Variable nicht verändert wird. In Python wird der Typ einer Variable aufgrund der Zuweisung ermittelt und kann sich während der Programmlaufzeit auch verändern.

Für erfahrene C/C++ Programmierer ist Python relativ einfach zu lernen. Dieser Versuch kann keine umfassende Python-Einführung geben, sondern soll nur einen ersten Einblick vermitteln. Ausführliche Python-Kurse bieten u.a. die folgenden Lehrbücher [1, 2, 3] und die folgenden online-Kurse [4, 5]. Gute Nachschlagewerke sind [6, 7].

2.1 Installation von Python und TensorFlow

Auf den Rechnern in den CIP-Pools (Seybothstraße) ist Python und TensorFlow bereits installiert. Da insbesondere TensorFlow sehr häufig aktualisiert wird, ist die in den CIP-Pools installierte Version in der Regel nicht die aktuelle.

Wenn Sie auf Ihrem eigenen Rechner mit Python und TensorFlow arbeiten wollen, müssen Sie diese zunächst installieren. Die folgenden Abschnitte erklären, wie Sie dazu in Windows vorgehen können. Das beschriebene Vorgehen ist für Windows 8.1 verifiziert, müsste in Windows 10 aber auch funktionieren.

■ Installation von Python 3.6.x

Laden Sie von der Webseite <https://www.python.org/> der Python Software Foundation eine Version 3.6.x von Python herunter und installieren Sie diese. Es gibt zwar schon die Version 3.7.x, diese wird jedoch aktuell (7. September 2018) von TensorFlow noch nicht unterstützt. Sie können die Installationsdatei

auch direkt hier herunterladen: <https://www.python.org/ftp/python/3.6.6/python-3.6.6-amd64.exe>.

Hinweis: Setzen Sie bei der Installation den Haken für die Einbindung in die Windows PATH-Variable.

■ Aktualisieren von pip3

Mit Hilfe des Paketverwaltungsprogramms pip3 wird im Folgenden TensorFlow installiert. Vorher müssen wir die Paketverwaltung aber noch aktualisieren. Dazu öffnen wir die Eingabeaufforderung in Windows. In Windows 10 muss man dazu die folgenden Schritte ausführen:

- Eingeben von cmd in der Cortana-Suchleiste links unten am Bildschirmrand.
- Auswahl des Eintrags Eingabeaufforderung mit der rechten Maustaste
- Auswahl der Option Ausführen im Kontextmenü.

In der Eingabeaufforderung kann man dann, wie in Abbildung 1 gezeigt, mit dem Befehl

```
pip3 install --upgrade pip
```

das Paketverwaltungsprogramm aktualisieren

■ Installation von TensorFlow

Mit dem Befehl

```
pip3 install --upgrade tensorflow
```

installieren wir jetzt in der Windows-Eingabeaufforderung TensorFlow. Falls Ihr Rechner über eine externe Grafikkarte verfügt (derzeit werden von TensorFlow nur NVidia Grafikkarten unterstützt), sollten Sie TensorFlow mit GPU-Support installieren, weil dadurch das Training neuronaler Netze um ein Vielfaches beschleunigt wird. Eine ausführliche Beschreibung dazu finden Sie hier: https://www.tensorflow.org/install/install_windows.

■ Installation weiterer Bibliotheken

Mit dem Befehl

```
python -m pip install --user scipy matplotlib sklearn pandas jupyter
```

installieren wir jetzt in der Windows-Eingabeaufforderung die Bibliotheken SciPy, Matplotlib und Scikit-Learn, Pandas und Jupyter.

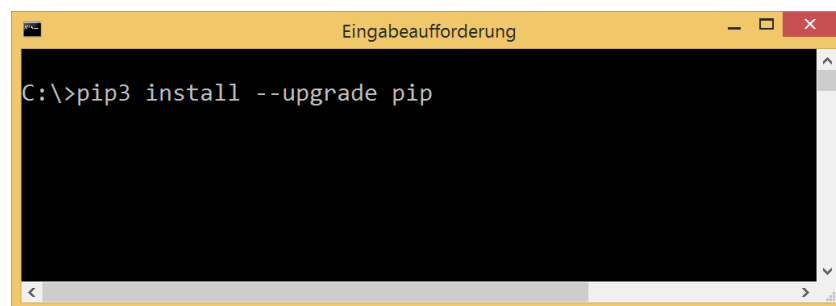


Abbildung 1: Aktualisieren von pip3 in der Windows-Eingabeaufforderung.

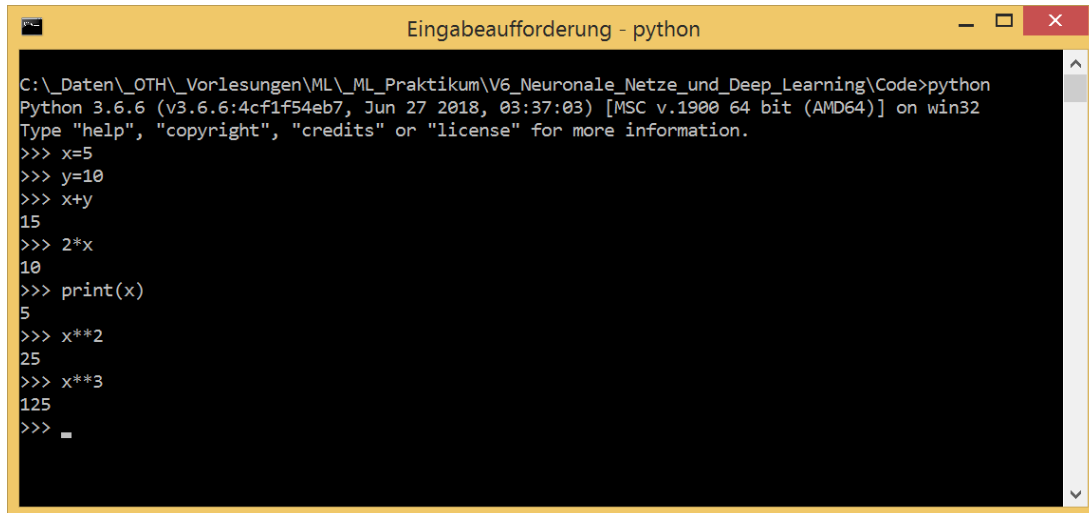
2.2 Interaktive Nutzung von Python

Die einfachste Art, Python zu nutzen, ist der sogenannte interaktive Modus. Er wird gestartet, indem wir in der Eingabeaufforderung den Python-Interpreter ohne Argumente aufrufen:

```
python
```

Alternativ können wir auch Python 3.6 (64 Bit) im Windows-Startmenü aufrufen. In diesem Modus können wir einfach Python-Anweisungen eingeben. Diese werden dann direkt an den Interpreter geschickt und ausgeführt. Auf diese Weise kann man sehr gut experimentieren und schnell etwas testen. Der interaktive Modus

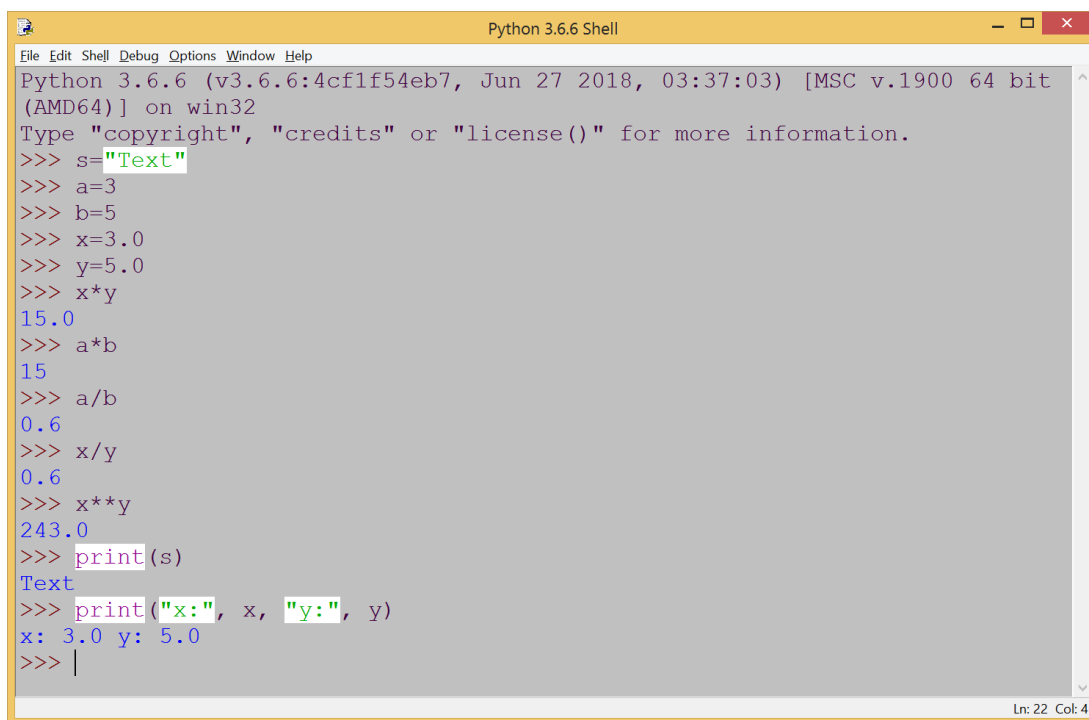
ist an der Python Eingabeaufforderung >>> zu erkennen. Zeilen, ohne >>> stellen Ausgaben des Interpreters dar. Abbildung 2 zeigt, dass man Python im interaktiven Modus z.B. wie einen Taschenrechner benutzen kann.



```
C:\_Daten\_OTH\_Vorlesungen\ML\_ML_Praktikum\V6_Neuronale_Netze_und_Deep_Learning\Code>python
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> x=5
>>> y=10
>>> x+y
15
>>> 2*x
10
>>> print(x)
5
>>> x**2
25
>>> x**3
125
>>> _
```

Abbildung 2: Verwendung von Python im interaktiven Modus.

Eine weitere Möglichkeit der interaktiven Arbeit mit Python bietet die zum Python-Paket gehörende integrierte Entwicklungsumgebung IDLE, wobei das Akronym IDLE für Integrated Development (and Learning) Environment steht. IDLE kann über das Windows-Startmenü aufgerufen werden. Abbildung 3 zeigt wie mit IDLE interaktiv Berechnungen ausgeführt werden können. Wir sprechen hier auch von der Python Shell.



```
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> s="Text"
>>> a=3
>>> b=5
>>> x=3.0
>>> y=5.0
>>> x*y
15.0
>>> a*b
15
>>> a/b
0.6
>>> x/y
0.6
>>> x**y
243.0
>>> print(s)
Text
>>> print("x:", x, "y:", y)
x: 3.0 y: 5.0
>>> |
```

Abbildung 3: Interaktive Nutzung von IDLE.

Die dritte Möglichkeit Python interaktiv zu nutzen sind Jupyter-Notebooks. Wenn Sie auf der Windows-Eingabeaufforderung

```
jupyter notebook
```

eingeben, wird ein Jupyter-Server gestartet. Gleichzeitig öffnet sich ein Webbrowser mit dem dieser Server betrachtet werden kann. Mit `New, Python 3` (siehe Abbildung 4) öffnen Sie ein neues Notebook, in dem Sie interaktiv arbeiten können. Abbildung 5 zeigt, wie das Notebook für einfache Berechnungen eingesetzt werden kann.

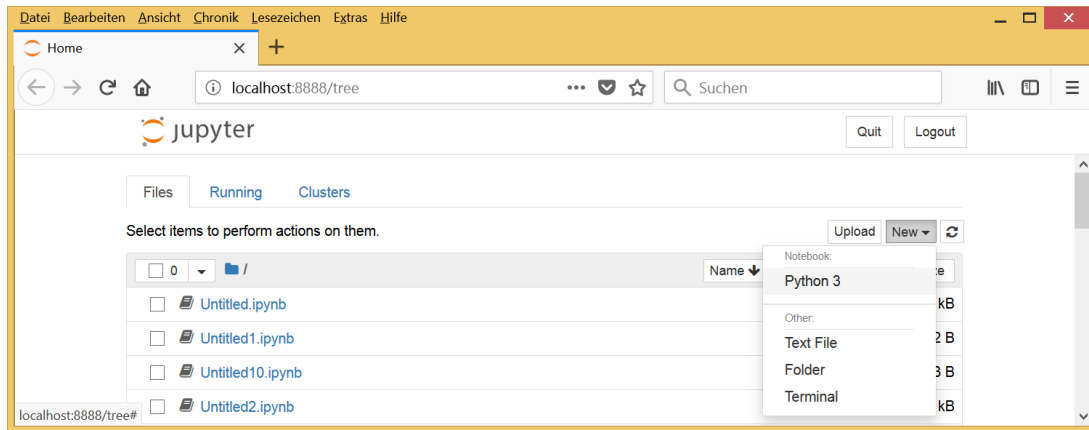


Abbildung 4: Öffnen eines neuen Jupyter-Notebooks.

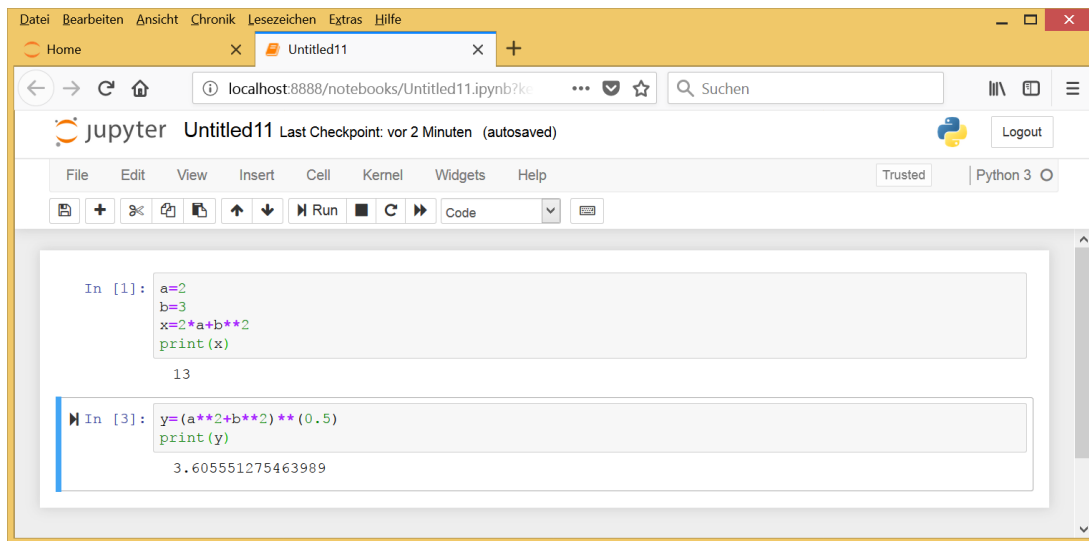


Abbildung 5: Ausführung einfacher Berechnungen in einem Jupyter-Notebook.

Durchführung

Aufgabe D1:

Führen Sie die folgenden Variablendefinitionen und Berechnungen jeweils in der Windows-Eingabeaufforderung, mit Hilfe von IDLE und in einem Jupyter-Notebook durch.

- Definieren Sie die drei Variablen a , b und c , indem Sie diesen die Werte 4, 5 und 6 zuweisen.
- $x = a + 2 * b + 3 * c^2$
- $y = a/b$
- $z = a^3 + b^c$
- $v = \sqrt{a} + b^{-c}$

2.3 Python Programme

Python kann natürlich mehr, als nur einzelne Befehle auf der Kommandozeile auszuführen. Mit Python lassen sich auch sehr komplexe Projekte realisieren. Dazu werden Python Programme bzw. Python Skripte erstellt. Zur Erstellung der Programme können wir einen beliebigen Texteditor verwenden. Die Programme selbst sind einfache Textdateien, deren Name mit der Erweiterung `.py` endet. Der folgende Code zeigt als Beispiel das Python Programm `HelloWorld.py`, welches "Hello World!" mit etwas Dekoration am Bildschirm ausgibt.

```
1 x = "-----"
2 print(x)
```

```
3 print("Hello World!")
4 print(x)
```

Um Python Programme auszuführen, gibt es verschiedene Möglichkeiten:

■ Mit der Windows-Eingabeaufforderung

Um das Programm `HelloWorld.py` auszuführen, geben wir in der Windows Eingabeaufforderung ein

```
python HelloWorld.py
```

Wenn wir in der `PATH`-Variable den Pfad zu unserer Python-Installation eingetragen haben, kann der Aufruf verkürzt werden, indem wir `python` einfach weglassen und nur noch schreiben:

```
HelloWorld.py
```

Abbildung 6 zeigt die beiden beschriebenen Varianten mit der zugehörigen Ausgabe auf dem Bildschirm.

Hinweis: Eine komfortablere Alternative zur Windows-Eingabeaufforderung ist eine Git Bash. Wenn Sie auf Ihrem Rechner git installiert haben, können Sie mit der Git Bash auch Python Skripte ausführen.

■ Durch Doppelklick im Filemanager

Im Windows-Filemanager kann ein Python Skript durch Doppelklick gestartet werden. Bei Programmen ohne Nutzerinteraktion hat das allerdings den Nachteil, das das Ausführungsfenster sofort nach Beendigung des Programms wieder geschlossen wird und man somit die Ausgaben nicht richtig sehen kann.

■ Mit IDLE

Dazu starten wir IDLE und öffnen mit `File, Open` die Datei `HelloWorld.py`. Es öffnet sich ein neues Fenster, in dem wir mit `Run, Run Module` das Programm ausführen können. Die Ausgabe erscheint dann im ursprünglichen IDLE-Fenster.

■ Mit einer integrierten Entwicklungsumgebung

In den meisten integrierten Entwicklungsumgebungen lässt sich das aktuelle Projekt direkt ausführen, meist durch Klicken eines kleinen grünen Pfeils nach rechts.

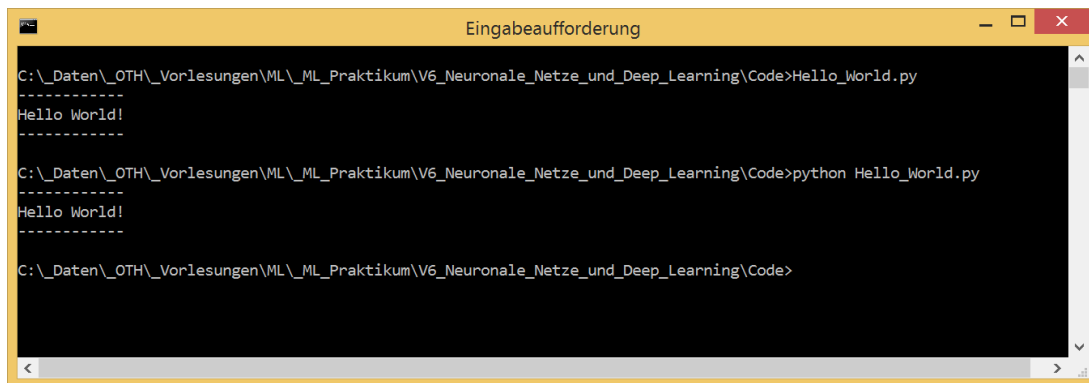


Abbildung 6: Zwei Varianten zur Ausführung des Python Programms `HelloWorld.py`.

Während bei kleinen Projekten das Arbeiten mit IDLE oder einem einfachen Texteditor, wie z.B. Notepad++, zweckmäßig ist, bieten bei größeren Projekten integrierte Entwicklungsumgebungen (IDE: integrated development environment) viele Vorteile. Drei Beispiele für solche IDEs sind PyDev (<http://www.pydev.org/>), PyCharm (<https://www.jetbrains.com/pycharm/>) und Sublime Text (<https://www.sublimetext.com/>). Eine umfangreiche Liste verschiedener Python-IDEs gibt es zum Beispiel hier: <https://wiki.python.org/moin/PythonEditors>.

2.4 Beispielprogramm

Im Folgenden schauen wir uns das Beispielprogramm `Beispiel_Statistik.py` an, welches eine ganze Reihe von Python Kontrollstrukturen enthält. Das Programm erzeugt eine Anzahl von normalverteilten Zu-

fallszahlen. Aus diesen Zufallszahlen wird dann der Mittelwert und die Standardabweichung geschätzt und ausgegeben.

```
1 import numpy as np
2 import math
3
4 # Funktion: Einlesen
5 def einlesen():
6     print("Geben Sie die folgenden Werte ein:")
7     print("Mittelwert:")
8     mx = float(input())
9     print("Standardabweichung:")
10    sx = float(input())
11    print("Anzahl der Werte:")
12    nx = int(input())
13    return mx, sx, nx
14
15 # Funktion zur Berechnung des Mittelwerts
16 def mittelwert(x,n):
17     summe = 0.0
18     for i in range(1, n, 1):
19         summe += x[i]
20     mx = summe/n
21     return mx
22
23 # Funktion zur Berechnung der Standardabweichung
24 def standardabweichung(x,n):
25     mx = mittelwert(x,n)
26     summe = 0.0
27     for i in range(1, n, 1):
28         summe += (x[i]-mx)**2
29     vx = summe/(n-1)
30     sx = math.sqrt(vx)
31     return sx
32
33 # Hauptprogramm
34 nochmal = "ja"
35 while nochmal == "ja":
36     m, s, N = einlesen()
37     x = np.random.normal(m,s,N)
38     print("Sollen die Zufallszahlen ausgegeben werden?")
39     Entscheidung = input()
40     if Entscheidung == "ja":
41         print(x)
42     mu = mittelwert(x, N)
43     sigma = standardabweichung(x,N)
44     print(f"Mittelwert: {mu:.2f}")
45     print(f"Standardabweichung: {sigma:.2f}")
46     print("Weiteren Satz an Zufallszahlen erzeugen und analysieren?")
47     nochmal = input()
```

In den Zeilen 1 und 2 werden die Bibliotheken NumPy und Math eingebunden. Für NumPy wird zusätzlich noch das Kürzel np vereinbart.

In den Zeilen 4 bis 13 wird die Funktion `einlesen()` definiert. Der Funktionskopf in Zeile 5 wird durch die Anweisung `def` eingeleitet. Danach folgt der Name der Funktion, die runden Klammern und ein Doppelpunkt. Die Funktion `einlesen()` hat keine Parameter, deshalb ist die runde Klammer leer. Sie gibt mit der `return`-Anweisung drei (!) Werte zurück. Die Rückgabe von mehr als einem Wert ist in Python kein Problem. Beim Aufruf der Funktion in Zeile 36 werden die drei Rückgabewerte in drei Variablen übergeben. Diese drei Variablen bilden ein sogenanntes Tupel. Der Rumpf der Funktion ist lediglich durch eine Einrückung gekennzeichnet. Python verwendet zur Strukturierung des Codes keine Klammern, sondern ausschließlich Einrückun-

gen. Die Funktion `input()` wird für das Einlesen von Nutzereingaben verwendet. `input` liefert zunächst eine Zeichenkette, die mit `float` in eine Gleitkommazahl umgewandelt wird und z.B. in der Variable `mx` gespeichert wird. Zur Ausgabe von Text am Bildschirm wird der Befehl `print()` verwendet. In den Zeilen 15 bis 31 werden zwei weitere Funktionen definiert, die jeweils 2 Übergabeparameter und einen Rückgabewert haben.

Im Hauptprogramm, welches in Zeile 33 beginnt, wird eine `while`-Schleife durchlaufen, solange die Nutzereingabe für die Variable `nochmal` gleich `ja` ist. Die Variable `nochmal` ist von Typ `str` (String, Zeichenkette). Für C-Programmierer ist es bemerkenswert, dass der Vergleichsoperator `==` in Zeile 35 auch mit Zeichenketten funktioniert. Die Funktion `numpy.random.normal()` stammt aus der Bibliothek NumPy und liefert hier in einem eindimensionalen Array normalverteilte Zufallszahlen. Beispiele für die Syntax einer `while`-Schleife, einer `for`-Schleife und einer `if`-Abfrage sind in den Zeilen 35, 18 bzw. 40 zu sehen.

Durchführung

Aufgabe D2:

Schreiben Sie ein Python Programm `Rechteck.py`, welches den Flächeninhalt, den Umfang und die Länge der Diagonale eines Rechtecks berechnet. Im Hauptprogramm soll der Nutzer die Länge und die Breite des Rechtecks eingeben können. Diese beiden Werte werden dann an die Funktion `rechteck()` übergeben, welche den Flächeninhalt, den Umfang und die Diagonallänge zurückgibt. Diese drei Werte sollen dann im Hauptprogramm mit zwei Stellen nach dem Komma ausgegeben werden. Schauen Sie sich für die formatierte Ausgabe die Zeilen 44 und 45 im Skript `statistik.py` an.

3 Machine Learning mit Python

Dank der leistungsfähigen eingebauten Datentypen und einiger leistungsfähiger Bibliotheken ist Python hervorragend für das Lösen technisch-wissenschaftlicher Probleme [8] und insbesondere für Machine Learning Aufgaben [9, 10, 11, 12, 13, 14] geeignet. Um zu zeigen, dass wir mit Python bereits ohne den Einsatz von TensorFlow sehr gut Machine Learning Algorithmen implementieren können, realisieren wir im Folgenden die lineare Regression auf unterschiedliche Weisen.

3.1 Lineare Regression mit NumPy

Das folgende Python Skript zeigt, wie sich die lineare Regression mit Hilfe von Matrixmultiplikationen mit der Bibliothek NumPy lösen lässt und wie die Ergebnisse unter Verwendung des Modules Pyplot aus der Bibliothek Matplotlib grafisch dargestellt werden können. Viele Befehle aus NumPy bzw. Pyplot ähneln den entsprechenden Matlab-Befehlen. Allerdings gibt es auch viele Unterschiede im Detail, so dass eine gewisse Einarbeitung notwendig ist. Einen Vergleich zwischen NumPy und Matlab finden Sie hier <https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html>. Einführungen in NumPy bieten z.B. der folgende User Guide [15], die SciPy Lecture Notes [16] oder das folgende Buch [17].

Der folgende Code `Beispiel_Linreg.py` führt eine lineare Regression für die Auto-Daten aus Versuch 2 durch. Die Berechnung erfolgt mit Hilfe der Normalengleichung.

```
1 #Bibliotheken importieren
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pandas as pd
5
```



```

6 # Daten importieren und Variablen extrahieren
7 df = pd.read_excel('Autos_DE.xlsx')
8 DATA = df.values;
9 y = np.array(DATA[2:,0], dtype='float') #Verbrauch
10 x = np.array(DATA[2:,3], dtype='float') #Hubraum
11 m = y.shape[0]
12
13 # Scatterplot der Daten
14 plt.figure(1)
15 plt.scatter(x,y,s=3,c='red')
16 plt.grid(True)
17 plt.title('Scatterplot')
18 plt.xlabel('Leistung x in PS')
19 plt.ylabel('Verbrauch y in Liter/100km')
20 plt.show(block=False) #block=False: Skript laeuft weiter
21
22 #Berechnung der Regressionsparameter
23 x0 = np.array(np.ones(m), dtype='float') #Hilfsvariable bestehend aus Einsen
24 X = np.stack((x0,x)).T #Datenmatrix X
25 Rxx = np.dot(X.T,X) #Korrelationsmatrix
26 beta = np.dot(np.dot(np.linalg.inv(Rxx),X.T),y)
27 print("Parametervektor beta= ", beta)
28
29 #Graphische Darstellung der Regressionsgerade
30 x_=np.linspace(np.min(x),np.max(x),200) #Hilfsvariable x_
31 y_ = beta[0]+beta[1]*x_; #Hilfsvariable y_
32 plt.plot(x_,y_,lw=2)
33 plt.grid(True)
34 plt.title('Datenpunkte mit Regressionsgerade')
35 plt.xlabel('Leistung x in PS')
36 plt.ylabel('Verbrauch y in Liter/100km')
37 plt.show(block=False)
38
39 plt.show() #Verhindert, dass Grafikfenster sofort geschlossen wird

```

In den Zeilen 2 bis 4 werden die Bibliotheken NumPy, Pyplot und Pandas importiert. Die Daten werden mit Hilfe des Pandas-Befehls `pandas.read_excel()` aus der Excel-Datei `Autos_DE.xlsx` eingelesen und dann in die Variablen `x` und `y` extrahiert. Pandas ist eine leistungsfähige Bibliothek für Python, die den Zugriff auf Daten vereinfacht. Ein Tutorial zu Pandas finden Sie z.B. hier [18]. Die Variablen `x` und `y` sind eindimensionale NumPy `ndarrays`, also Vektoren.

Mit dem Pyplot-Befehl `scatter` wird in Zeile 15 ein Streudiagramm gezeichnet. Der Parameter `s=3` setzt die Größe (size) der Punkte auf 3 und `c='red'` setzt deren Farbe auf rot. Der abschließende `show`-Befehl in Zeile 20 löst den Zeichenvorgang aus: Ein Grafikfenster mit dem Streudiagramm öffnet sich. Dass das Python-Skript weiterläuft, auch wenn das Grafikfenster geöffnet bleibt, wird durch `block=False` erreicht. Ansonsten würde das Skript nämlich pausieren und warten bis der Nutzer das Grafikfenster schließt.

In den Zeilen 22 bis 26 werden die Regressionsparameter $\hat{\beta}$ gemäß der Gleichung

$$\hat{\beta} = (X^T X)^{-1} X^T y \quad (1)$$

berechnet (siehe Versuch 2). Dazu wird zunächst die Hilfsvariable x_0 (Zeile 23) zu den eigentlichen Daten hinzugefügt, so dass die Datenmatrix X entsteht (Zeile 24). Die Variable `x0` ist ein eindimensionales NumPy `ndarray`, X ist ein zweidimensionales NumPy `ndarray`, also eine Matrix. Zur Matrixmultiplikation und Matrixinversion werden die NumPy-Befehl `dot` und `inv` eingesetzt. `X.T` repräsentiert die transponierte Datenmatrix.

In den Zeilen 30 und 31 wird die Regressionsgerade berechnet und dann in den Zeilen 33 bis 39 grafisch dargestellt. Der Parameter `lw=2` beim `plot`-Befehl stellt die Linienbreite (line width) auf 2 ein. Der abschließende

`show`-Befehl verhindert, dass das Grafikfenster am Ende des Skript sofort geschlossen wird. Wenn wir den `show`-Befehl aus Zeile 37 durch den `show`-Befehl aus Zeile 39 ersetzen würden, könnte wir auf die Zeile 39 verzichten. Die hier dargestellte Version erlaubt es jedoch, zwischen den beiden `show`-Befehlen das Skript zu erweitern. Abbildung 7 zeigt die grafische Darstellung der Datenpunkte sowie der Regressionsgeraden.

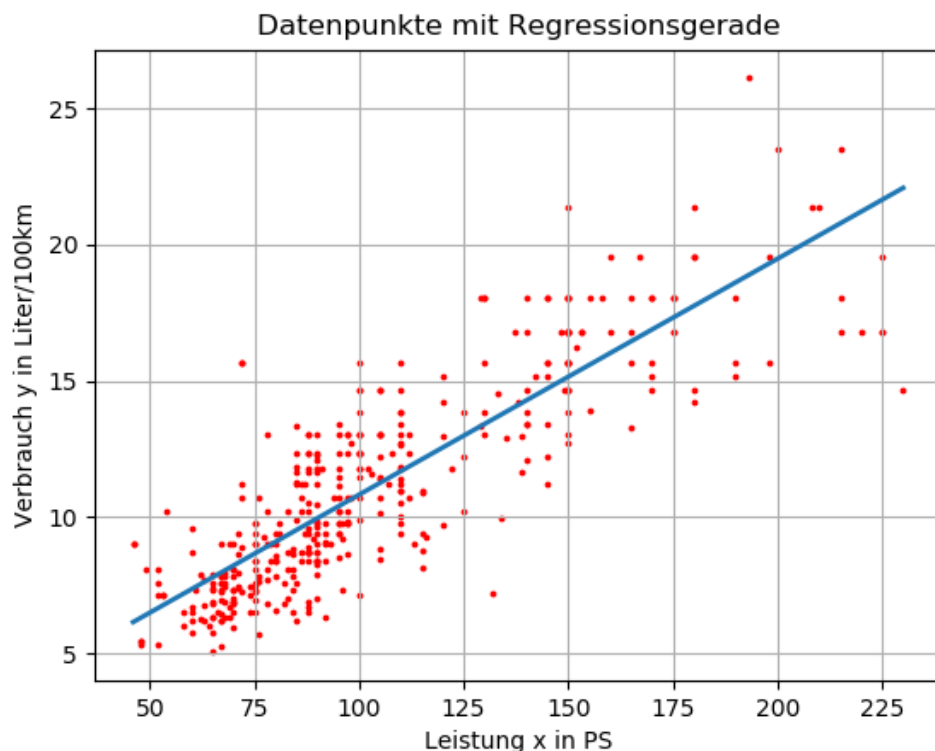


Abbildung 7: Streudiagramm und Regressionsgerade für die Autodaten.

Durchführung

Aufgabe D3:

Erweitern Sie das im File `Beispiel_Linreg.py` gegebene und oben besprochene Python-Skript, so dass es eine lineare Regression mit allen 6 Eingangsvariablen durchführt. Stellen Sie in einem Grafikfenster nebeneinander die Streudiagramme und die Regressionsgeraden für die Eingangsvariablen x_2 (Hubraum) und x_4 (Gewicht) grafisch dar.

3.2 Lineare Regression mit Scikit-Learn

Die Bibliothek Scikit-Learn bietet leistungsfähige Implementierungen vieler Machine-Learning Algorithmen und arbeitet mit NumPy und SciPy zusammen. Gute Einführungen finden Sie in der online-Dokumentation [19] sowie in den Büchern [9, 11].

Für die lineare Regression verfügt Scikit-Learn über die Klasse

```
sklearn.linear_model.LinearRegression
```

Damit lassen sich einfach die Regressionsparameter bestimmen. Das folgende Listing `Beispiel_Linreg_sklearn.py` zeigt Ausschnitte aus einem Python-Skript, welches die Regressionsgerade für die Autodaten bestimmt. Die mit `...` gekennzeichneten Teile sind identisch mit dem Skript `Beispiel_Linreg.py` aus dem vorherigen Abschnitt.

```

1 #Bibliotheken importieren
2 ...
3 from sklearn.linear_model import LinearRegression
4
5 # Daten importieren und Variablen extrahieren
6 ...
7 # Scatterplot der Daten
8 ...
9
10 #Berechnung der Regressionsparameter
11 x0 = np.array(np.ones(n), dtype='float') #Hilfsvariable bestehend aus Einsen
12 X = np.stack((x0,x)).T #Datenmatrix X
13 lm = LinearRegression()
14 lm.fit(X,y)
15 beta1=lm.coef_[1]
16 beta0=lm.intercept_
17 print("beta0=", beta0, "beta1=", beta1)
18
19 #Graphische Darstellung der Regressionsgerade
20 x_=np.linspace(np.min(x),np.max(x),200) #Hilfsvariable x_
21 y_ = beta0+beta1*x_; #Hilfsvariable y_
22 ...

```

In Zeile 13 wird ein Object `lm` für die lineare Regression angelegt. Dessen Parameter werden in der folgenden Zeile mit der Methode `fit` bestimmt. Nach dem Aufruf von `fit` liegen die aus den Daten geschätzten Parameter in den Attributen `intercept_` und `coef_` vor.

4 Einstieg in TensorFlow

TensorFlow ist ein leistungsfähiges Software-Paket für das wissenschaftliche Rechnen. Es ist darauf optimiert, große Datenmengen möglichst effizient zu verarbeiten. Die Verarbeitung kann dabei auch auf mehreren CPUs (Central Processing Units) bzw. GPUs (Graphics Processing Units) verteilt werden. Das Haupteinsatzgebiet sind Deep-Learning-Anwendungen. TensorFlow kann aber auch in ganz anderen Gebieten verwendet werden, z.B. zur Simulation von Differentialgleichungen. Ursprünglich wurde TensorFlow vom Google BrainTeam für die interne Verwendung entwickelt und wurde im November 2015 unter einer Open-Source-Lizenz veröffentlicht. Seitdem hat es sich zu einem der meistverwendeten Bibliotheken zur Realisierung neuronaler Netze entwickelt.

Gute Einführungen zu TensorFlow bieten die “Getting Started Pages” von TensorFlow [20] und das Buch von A. Geron [9], welches auch in einer deutschen Übersetzung erhältlich ist [10]. Mir persönlich haben beim Einstieg in TensorFlow v.a. die “Getting Started Pages” einer älteren Version von TensorFlow geholfen [21, 22], an denen sich z.T. die folgende Einführung orientiert.

Wie NumPy ist TensorFlow für umfangreiche Berechnungen wie z.B. Matrixmultiplikationen optimiert. Genauso wie bei NumPy führt TensorFlow die Berechnungen jedoch nicht direkt in Python durch, sondern nutzt dazu hocheffizienten Code, der in einer anderen Programmiersprache (bei TensorFlow ist das i.d.R. C++) implementiert wurde. In NumPy entsteht beim Übergang von Python auf den effizienten Code ein Overhead, der die Berechnungen verlangsamt. Um diesen Overhead zu reduzieren und insbesondere die Berechnungen so effizient wie möglich auf GPUs oder verteilten Systemen durchführen zu können, geht TensorFlow noch einen Schritt weiter. Um Berechnungen auszuführen, sind in TensorFlow immer zwei Schritte notwendig:

- **Erzeugung des Berechnungsgraphen:**

Zunächst muss ein sogenannter *Berechnungsgraph* (Englisch: *Computational Graph*) erzeugt werden, der genau festlegt, was berechnet werden soll. Der Berechnungsgraph wird mit Hilfe von Befehlen erstellt, kann

aber auch graphisch visualisiert werden. Der Berechnungsgraph ist eine Folge von Operationen, die mit Hilfe von verbundenen *Knoten* (Englisch *nodes*) dargestellt werden.

■ Ausführen des Berechnungsgraphen:

Im zweiten Schritt wird der Berechnungsgraph durch den effizienten Code ausgeführt. Weil dabei nicht mehr zwischen Python und dem effizienten Code hin- und hergewechselt werden muss, wird der Overhead deutlich minimiert.

Als ein einfaches Beispiel für dieses Vorgehen schauen wir uns die Berechnung der Funktion

$$f(x, y) = x^2 + xy + y^2 + 5$$

an. Der zugehörige Berechnungsgraph ist in Abbildung 8 dargestellt und der zugehörige Python-Code befindet sich im folgenden Listing `Beispiel_TF_fxy.py`, welches für die gegebenen Werte der Variablen $x = 5$ und $y = 6$ den Funktionswert $f(x, y) = 96$ liefert.

```
1 #Bibliotheken importieren
2 import tensorflow as tf
3 import os
4 os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
5
6 #Erzeugen des Berechnungsgraphen
7 x = tf.Variable(5.0, dtype=tf.float32, name="x") #Variable x mit Wert 5
8 y = tf.Variable(6.0, dtype=tf.float32, name="y") #Variable y mit Wert 6
9 c = tf.constant(5.0, dtype=tf.float32)
10 f = x*x+x*y+y*y+c #Definition des Berechnungsgraphen
11
12 #Ausführen des Berechnungsgraphen
13 sess = tf.Session() #Öffnen einer TensorFlow Session
14 sess.run(x.initializer) #Initialisierung der Variablen x
15 sess.run(y.initializer) #Initialisierung der Variablen y
16 result = sess.run(f) #Ausführen des Berechnungsgraphen
17 print(result) #Ausgabe des Ergebnisses
18 sess.close() #Schließen der Session
```

In den Zeilen 2 bis 4 werden die benötigten Bibliotheken importiert. Zeile 4 dient lediglich zur Unterdrückung von Warnungen. In den Zeilen 7 bis 9 wird der Berechnungsgraph erzeugt, indem zunächst die Variablen x und y einschließlich ihrer Typen und ihrer Anfangswerte festgelegt werden. Letztendlich stellen diese beiden Variablen Knoten des Berechnungsgraphen dar. In Zeile 9 werden diese Knoten durch Operationen miteinander zur Funktion $f(x, y)$ verknüpft. Jede Operation (im Berechnungsgraph gibt es drei Multiplikationen und drei Additionen) stellt selbst einen Knoten des Berechnungsgraphen dar. Ganz allgemein ist ein Knoten eines Berechnungsgraphen eine Einheit, die keinen, einen oder mehrere Tensoren als Eingangswert verarbeitet und einen Tensor als Ausgangswert liefert. Tensoren sind die grundlegenden Dateneinheiten in TensorFlow. Ein Tensor ist ein n -dimensionales Array, wie in den folgenden Beispielen aus [21] veranschaulicht wird:

```
3 # a rank 0 tensor; this is a scalar with shape []
[1., 2., 3.] # a rank 1 tensor; this is a vector with shape [3]
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```

Wichtig für das Verständnis von TensorFlow ist, dass durch die Zeilen 7 bis 10 noch keinerlei Berechnungen ausgeführt werden. Die Variablen x und y werden noch nicht einmal initialisiert. Es wird lediglich der in Abbildung 8 dargestellte Berechnungsgraph beschrieben.

Zur Ausführung des Berechnungsgraphen dienen die Zeilen 13 bis 18. Hier wird zunächst eine TensorFlow-Session geöffnet, dann werden die beteiligten Variablen initialisiert und schließlich (Zeile 16) der Berechnungsgraph ausgeführt. Anschließend wird das Ergebnis auf dem Bildschirm dargestellt und die TensorFlow-Session wird geschlossen.

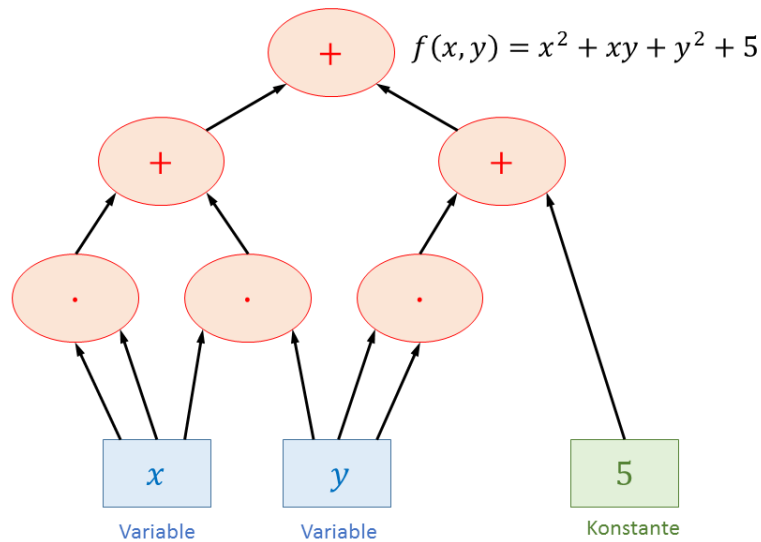


Abbildung 8: Berechnungsgraph für die Funktion $f(x, y)$.

4.1 Variablen und Platzhalter

Neben Variablen, Konstanten und Operationen kommen in TensorFlow auch sogenannte *Platzhalter* (*placeholder*) als Knoten in Frage. Genauso wie Variablen können auch die Platzhalter unterschiedliche Werte annehmen. Was ist dann der Unterschied zwischen Variablen und Platzhaltern? In einem Machine Learning Code werden die Platzhalter für die Trainingsdaten verwendet und die Variablen für die Parameter, die aus den Trainingsdaten gelernt werden sollen. Der Unterschied wird anhand späterer Beispiele noch deutlich. Zunächst setzen wir einfach das obige Beispiel des Berechnungsgraphen aus Abbildung 8 mit Platzhaltern um. Dazu ändern wir den obigen Code geringfügig ab und erhalten das folgende Listing `Beispiel_TF_fxy_placeholder.py`:

```

1 #Bibliotheken importieren
2 import tensorflow as tf
3 import os
4 os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
5
6 #Erzeugen des Berechnungsgraphen
7 x = tf.placeholder(dtype=tf.float32) #Platzhalter x
8 y = tf.placeholder(dtype=tf.float32) #Platzhalter y
9 c = tf.constant(5.0, dtype=tf.float32)
10 f = x*x+x*y+y*y+c #Definition des Berechnungsgraphen
11
12 #Ausführen des Berechnungsgraphen
13 sess = tf.Session() #Öffnen einer TensorFlow Session
14 result = sess.run(f, {x: 5.0, y: 6.0}) #Ausführen des Berechnungsgraphen
15 print(result) #Ausgabe des Ergebnisses
16 sess.close() #Schließen der Session

```

In den Zeilen 7 und 8 werden die Knoten `x` und `y` jetzt als Platzhalter definiert. Platzhalter haben keinen Anfangswert, sondern lediglich einen Typ. Beim Ausführen des Berechnungsgraphen in Zeile 14 wird den beiden Platzhaltern in der `run`-Methode des Objekts `sess` jeweils ein Wert zugewiesen.

Im folgenden Skript `Beispiel_TF_x2.py` nutzen wir Platzhalter, um die Funktion $y = x^2$ grafisch darzustellen.

```

1 #Bibliotheken importieren
2 import tensorflow as tf
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import os
6 os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

```

```

7
8 #Erzeugen des Berechnungsgraphen
9 x = tf.placeholder(dtype=tf.float32) #Platzhalter x
10 y = x*x #Definition des Berechnungsgraphen
11
12 #Ausführen des Berechnungsgraphen
13 x_np = np.arange(-3.0,3.0,0.01)
14 sess = tf.Session() #Oeffnen einer TensorFlow Session
15 y_np = sess.run(y, {x: x_np}) #Ausführen des Berechnungsgraphen
16 print(y_np) #Ausgabe des Ergebnisses
17 print(y_np.shape) #Ausgabe des Ergebnisses
18 sess.close() #Schließen der Session
19
20 #Grafische Darstellung des Ergebnisses
21 plt.figure(1)
22 plt.plot(x_np,y_np,lw=2)
23 plt.grid(True)
24 plt.title('y=x^2')
25 plt.xlabel('x')
26 plt.ylabel('y')
27 plt.show()

```

Dazu definieren wir in Zeile 9 den Platzhalter x . Der Berechnungsgraph für $y = x^2$ wird in Zeile 10 festgelegt. Zur Ausführung des Berechnungsgraphen erstellen wir zunächst ein NumPy-Array x_np , welches Werte von -3.0 bis 3.0 in Schritten von 0.01 enthält. In der Methode `run` des Objekts `sess` wird dieses Array an den Platzhalter x übergeben. Damit wird der Berechnungsgraph $y = x*x$ für jeden Wert aus x_np durchgeführt und die Methode `run` liefert ein NumPy-Array y_np zurück, welche die zu x_np gehörigen Funktionswerte enthält. Der Vorteil, die Berechnung in TensorFlow statt direkt in NumPy durchzuführen, besteht darin, dass auf diese Weise die Berechnung aller Werte ohne Unterbrechung und ohne Overhead durchgeführt werden kann. Natürlich lohnt sich das für ein so einfaches Beispiel mit so wenig Daten nicht. Für das Training großer neuronaler Netze mit großen Datensätzen bringt diese Vorgehensweise einen entscheidenden Geschwindigkeitsvorteil. In den Zeilen 21 bis 27 wird das Berechnungsergebnis mit Hilfe der Matplotlib grafisch dargestellt, wie in Abbildung 9 gezeigt.

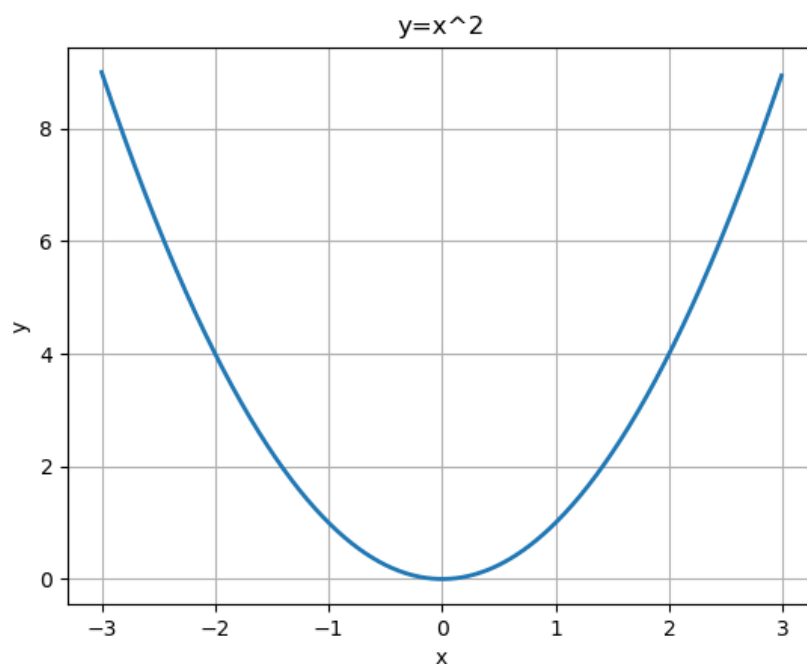


Abbildung 9: Grafische Darstellung der Funktion $y = x^2$.

Durchführung

Aufgabe D4:

Zeichnen Sie mit Hilfe von Python und Matplotlib die Geradenschar $y = mx + t$ im Intervall $x = [-3; 3]$, wobei m eine Konstante mit dem Wert $m = 0.5$ ist und t eine Variable mit den Werten $t \in \{-2, -1, 0, 1, 2\}$ ist. Führen Sie die Berechnung der Funktionswerte in TensorFlow aus. Benutzen Sie für x einen TensorFlow-Platzhalter, für m eine TensorFlow-Konstante und für t eine TensorFlow-Variable. Ändern Sie den Wert von t mit dem Befehl `tf.assign`.

4.2 Lineare Regression mit TensorFlow

Ähnlich wie in Abschnitt 3.1 lässt sich die lineare Regression auch mit Hilfe von TensorFlow berechnen, indem man direkt die Normalengleichung löst. Die Implementierung des folgenden Codes `Beispiel_TF_Linreg.py` setzt genau diese Idee für die Auto-Daten aus Versuch 2 um. Dabei werden alle 6 Eingangsvariablen für die lineare Regression verwendet.

```
1 import tensorflow as tf
2 import numpy as np
3 import os
4 os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
5 import matplotlib.pyplot as plt
6 import pandas as pd
7
8 # Daten importieren und Variablen extrahieren
9 df = pd.read_excel('Autos_DE.xlsx')
10 DATA = df.values;
11 ynp = np.array(DATA[2:399,0], dtype='float') #Ausgangsvariable
12 X0 = np.array(DATA[2:399,1:7], dtype='float') #Eingangsvars ohne x0
13 X0 = X0.T
14
15 # Hilfsvariable mit Einsen der Datenmatrix voranstellen
16 m, n = X0.shape
17 x0 = np.array(np.ones((1,n)), dtype='float') #Hilfsvariable bestehend aus Einsen
18 Xnp = np.concatenate((x0,X0)).T #Eingangsvars mit x0
19
20 # Berechnungsgraph für die Parameter beta erstellen
21 X = tf.constant(Xnp, dtype=tf.float32, name="X")
22 y = tf.constant(ynp.reshape(-1, 1), dtype=tf.float32, name="y")
23 XT = tf.transpose(X)
24 beta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)
25
26 #Berechnungsgraph ausführen
27 with tf.Session() as sess:
28     beta_value = beta.eval()
29
30 print(beta_value)
31 sess.close()
```

Zunächst werden mit Pandas die Daten aus dem Excel-File `Autos_DE.xlsx` in die entsprechenden NumPy-Arrays extrahiert (Zeilen 9 bis 13) und anschließend die Hilfsvariable `x0` bestehend aus lauter Einsen vorangestellt, so dass die NumPy-Datenmatrix `Xnp` entsteht (Zeilen 16 bis 18). In den Zeilen 21 bis 24 wird der Berechnungsgraph für die Ermittlung des Parametervektors `beta` nach der Normalengleichung erstellt und in den Zeilen 27 bis 31 ausgeführt.

Eine geschlossene Lösung wie die Normalengleichung gibt es für die wenigsten Machine Learning Probleme. Deshalb ist es interessant, bereits für einfache Aufgabenstellungen, wie die lineare Regression, Optimierungs-

verfahren einzusetzen, die sich auch auf schwierigere Probleme verallgemeinern lassen. Der folgende Code `Beispiel_TF_Linreg_gradient.py` zeigt eine Implementierung der linearen Regression mit Hilfe des Gradientenverfahrens.

```

1 import numpy as np
2 import pandas as pd
3 from sklearn import preprocessing
4 import tensorflow as tf
5 import os
6 os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
7
8 # Modelparameter
9 beta0 = tf.Variable([0.3], dtype=tf.float32) #Achsenabschnitt
10 beta1 = tf.Variable([-0.3], dtype=tf.float32) #Steigung
11
12 # Model input and output
13 x = tf.placeholder(tf.float32) #Eingangsvariable
14 h = beta1 * x + beta0          #Hypothese
15 y = tf.placeholder(tf.float32) #Ausgangsvariable
16
17 # Kostenfunktion
18 C = tf.reduce_sum(tf.square(h - y)) # Summe der Fehlerquadrate
19
20 # Berechnungsgraph zur Minimierung der Kostenfunktion
21 optimizer = tf.train.GradientDescentOptimizer(0.001)
22 train = optimizer.minimize(C)
23
24 # Bereitstellung der Trainingsdaten
25 df = pd.read_excel('Autos_DE.xlsx')
26 DATA = df.values;
27 y_train = np.array(DATA[2:,0], dtype='float') #Verbrauch
28 x_train = np.array(DATA[2:,3], dtype='float') #Leistung
29 mx = np.mean(x_train) #Mittelwert
30 sx = np.std(x_train)  #Standardabweichung
31 x_scaled = preprocessing.scale(x_train) #skalierte Version von x
32
33 # Ausführung des Berechnungsgraphen für das Training
34 init = tf.global_variables_initializer()
35 sess = tf.Session()
36 sess.run(init) #Parameter initialisieren
37 for i in range(100): #Ausfuehrung von 1000 Gradienteniterationen
38     sess.run(train, {x:x_scaled, y:y_train})
39
40 # evaluate training accuracy
41 betalhat, beta0hat, Chat = sess.run([beta1, beta0, C], {x:x_train, y:y_train})
42 print("beta1: ", betalhat/sx, "beta0: ", beta0hat-betalhat*mx/sx, "C: ", Chat)

```

Die beiden Parameter der Regressionsgerade `beta0` und `beta1` werden als TensorFlow-Variablen in den Zeilen 9 und 10 definiert, während für die Eingangsvariable `x` und die Ausgangsvariable `y` in den Zeilen 13 und 15 je ein TensorFlow-Placeholder verwendet wird. Die Hypothese und die Kostenfunktion werden in den Zeilen 14 und 18 definiert. Zur Ermittlung der optimalen Parameter setzen wir aus dem TensorFlow-Modul `tf.train` die Klasse `GradientDescentOptimizer` ein, um die Kostenfunktion zu minimieren. In den Zeilen 21 und 22 wird der zugehörige Berechnungsgraph definiert. In den Zeilen 34 bis 38 wird dieser Graph ausgeführt. Dazu werden an die beiden Platzhalter `x` und `y` die Trainingsdaten `x_scaled` und `y_train` übergeben. Um eine schnelle Konvergenz des Gradientenabstiegsverfahrens zu erreichen, ist eine Skalierung der Eingangsdaten notwendig. Diese wird mit der Methode `preprocessing.scale` aus Scikit-Learn durchgeführt. Wie in Versuch 2 ausgeführt, konvergiert das Gradientenverfahren bei der linearen Regression, wenn die Schrittweite α wie folgt gewählt wird:

$$0 \leq \alpha \leq \frac{2}{\lambda_{\max}}$$

wobei λ_{\max} der größte Eigenwert der Korrelationsmatrix $R = \frac{1}{m} \mathbf{X}^T \mathbf{X}$ ist. Da die Klasse `GradientDescentOptimizer` bei der Berechnung des Gradientenvektors nicht durch die Anzahl m der Trainingsdaten dividiert, muss die zulässige Schrittweite wie folgt modifiziert werden:

$$0 \leq \alpha \leq \frac{2}{m \cdot \lambda_{\max}} \quad (2)$$

Deshalb wählen wir die relativ kleine Schrittweite von $\alpha = 0.001$ (Zeile 21).

Um die Werte der Parameter und der Kostenfunktion zu erhalten, führen wir in Zeile 41 die Berechnungsgraphen von `beta1`, `beta0` und `C` aus. Beim anschließenden `print`-Befehl erfolgt eine manuelle Zurückskalierung der Parameter.

5 Feedforward-Netzwerke in TensorFlow

In diesem Abschnitt schauen wir uns näher an, wie man einfache neuronale Netze, sogenannte Feedforward-Netzwerke in TensorFlow implementiert. Als Beispiel dazu verwenden wir die Erkennung handgeschriebener Ziffern und greifen auf den sehr weit verbreiteten MNIST Datensatz [23] zurück. Der folgende Code und die zugehörige Erklärung sind an [22] angelehnt. Wir sehen uns zunächst den MNIST-Datensatz und die zugehörige Erkennungsaufgabe an, lösen sie dann zunächst mit der Softmax-Regression (Abschnitt 5.1) und anschließend mit einem vollständigen Feedforward-Netzwerk (Abschnitt 5.2).

Beim MNIST-Datensatz geht es darum, handgeschriebene Ziffern, wie in Abbildung 10 dargestellt, richtig zu erkennen. Dabei handelt es sich um ein Klassifikationsproblem mit 10 Klassen, nämlich die Ziffern 0 bis 9.



Abbildung 10: Zufällig ausgewählte Beispiele für Ziffern aus dem MNIST Datensatz [22].

Jede Ziffer wird dabei als ein Bild mit 28×28 Pixeln gespeichert. Jedes Pixel (Picture Element) kann Grauwerte zwischen 0 (weiß) und 1 (schwarz) annehmen. Wie in Abbildung 11 dargestellt, kann jede Ziffer als eine 28×28 Matrix von Grauwerten interpretiert werden. Wir können diese Matrix in einen Vektor mit $28 \cdot 28 = 784$ Elementen umformen. Dabei spricht man auch von “matrix flattening” oder “parameter unrolling”. Das heißt, wir können letztendlich jedes Bild einer Ziffer als einen Punkt in einem 784-dimensionalen Vektorraum betrachten.

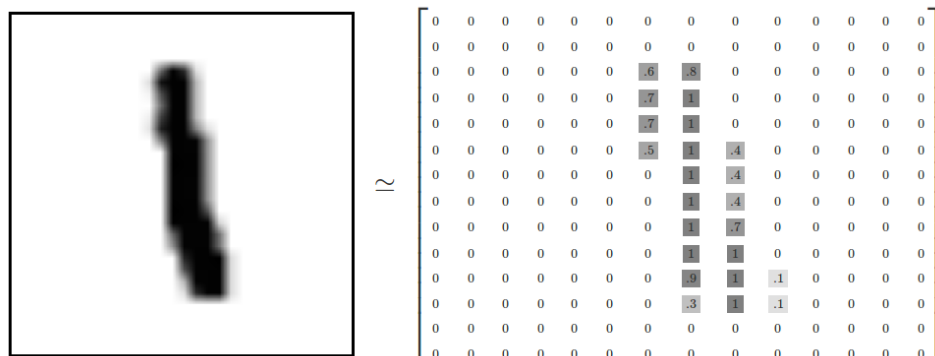


Abbildung 11: Bild einer Ziffer als Matrix von Grauwerten [22].

Der Trainingsdatensatz von MNIST besteht aus 55000 Bildern handgeschriebener Ziffern. Wenn wir jedes Bild in einen 784-dimensionalen Vektor umwandeln, dann ist der Trainingsdatensatz `mnist.train.xs` der Eingangsvariablen ein Tensor der Dimension $[55000, 784]$, wie in Abbildung 12 dargestellt. Da wir zwischen 10 Klassen unterscheiden wollen, ist die Ausgangsvariable y ein Vektor der Länge 10, dessen Einträge einer 1-aus-10-Codierung entnommen sind (One-Hot-Codierung). Das Element der richtigen Klasse hat dabei jeweils den Wert 1, alle anderen Elemente haben den Wert 0. Der zugehörige Tensor der Trainingsdaten ist in Abbildung 13 dargestellt.

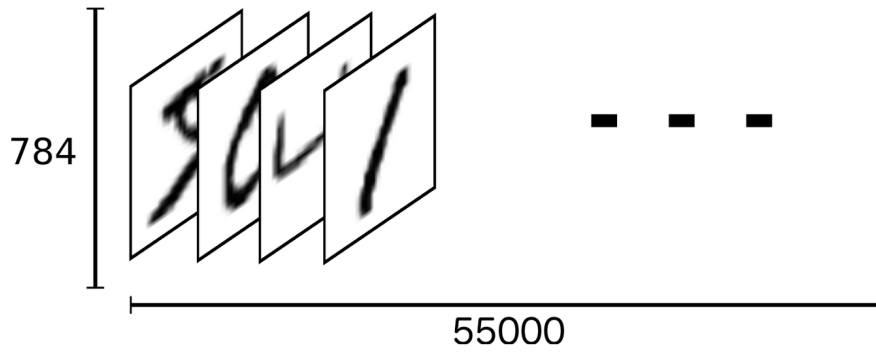


Abbildung 12: Tensor der Trainingsdaten für die Eingangsvariablen [22].

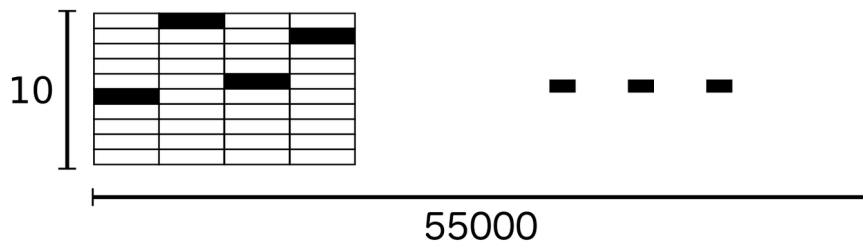


Abbildung 13: Tensor der Trainingsdaten für die Ausgangsvariablen (One-Hot-Codierung) [22].

Bei der Ziffernerkennung, die wir in den folgenden beiden Abschnitten auf zwei unterschiedliche Arten lösen werden, haben wir also 784 Eingangsvariablen zur Verfügung, um 10 Klassen zu unterscheiden.

5.1 Softmax-Regression

Eine der einfachsten Möglichkeiten, einen Klassifikator für $K = 10$ Klassen zu implementieren ist die Softmax-Regression (treffender aber weniger üblich ist der Name Softmax-Klassifikation). Dabei handelt es sich um eine Erweiterung der logistischen Regression auf den Fall von mehr als zwei Klassen. Man kann die Softmax-Regression auch als ein neuronales Netz mit einer Eingabeschicht und einer Ausgabeschicht betrachten, welches keine verdeckten Schichten besitzt. Der Softmax-Klassifikator für das MNIST-Problem ist in Abbildung 14 dargestellt. Bei den meisten neuronalen Netzen, die für die Klassifikation verwendet werden, ist die Ausgabeschicht als Softmax-Klassifikator ausgeführt. Deshalb hat der Softmax-Klassifikator eine äußerst große Bedeutung. Man bezeichnet die Ausgabeschicht dann auch als ein “fully connected layer”.

Die Hypothese eines Softmax-Klassifikators berechnet sich wie folgt

$$z_i = \sum_j W_{i,j} x_j + b_i$$

$$h_i = \text{softmax}(z_i)$$

wobei für die Softmax-Operation gilt

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

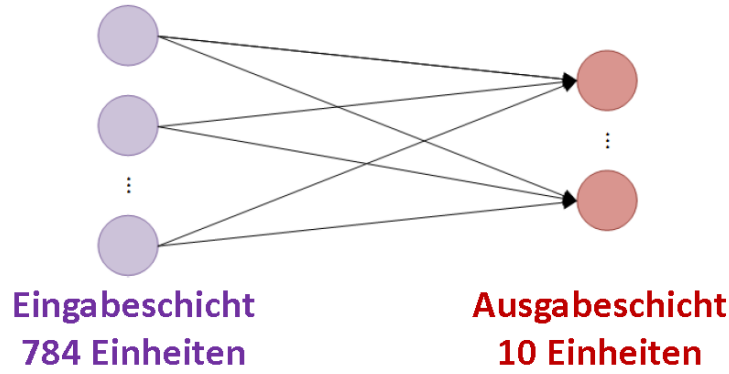


Abbildung 14: Veranschaulichung des Softmax-Klassifikators für MNIST [22].

In Vektor-Matrix-Schreibweise lässt sich die Hypothese folgendermaßen ausdrücken

$$\mathbf{h}(\mathbf{x}) = \begin{pmatrix} P(y = 1 | \mathbf{x}; \mathbf{W}; \mathbf{b}) \\ P(y = 2 | \mathbf{x}; \mathbf{W}; \mathbf{b}) \\ \vdots \\ P(y = K | \mathbf{x}; \mathbf{W}; \mathbf{b}) \end{pmatrix} = \frac{1}{\sum_{j=1}^K \exp(\mathbf{W}_j^T \mathbf{x} + b_j)} \begin{pmatrix} \exp(\mathbf{W}_1^T \mathbf{x} + b_1) \\ \exp(\mathbf{W}_2^T \mathbf{x} + b_2) \\ \vdots \\ \exp(\mathbf{W}_K^T \mathbf{x} + b_K) \end{pmatrix}$$

Die Abbildungen 15 und 16 veranschaulichen die obige Gleichung.

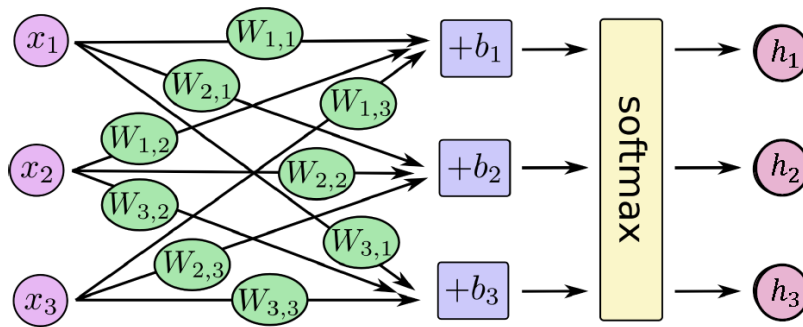


Abbildung 15: Berechnung der Hypothese für den Softmax-Klassifikator [22].

Darstellung als Gleichung

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \text{softmax} \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix}$$

Umschreiben in Vektor-Matrix-Notation

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \text{softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Abbildung 16: Veranschaulichung der Gleichungen zur Berechnung der Hypothese für den Softmax-Klassifikator [22].

Der folgende Code `Beispiel_MNIST_Softmax.py` (modifiziert nach [22]) implementiert einen Softmax-Klassifikator für das MNIST-Beispiel.

```
1 # Copyright 2015 The TensorFlow Authors. All Rights Reserved.
2 #
```

```

3 # Licensed under the Apache License, Version 2.0 (the "License");
4 # you may not use this file except in compliance with the License.
5 # You may obtain a copy of the License at
6 #
7 #     http://www.apache.org/licenses/LICENSE-2.0
8 #
9 # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 # =====
15
16 from __future__ import absolute_import
17 from __future__ import division
18 from __future__ import print_function
19
20 import argparse
21 import sys
22
23 #Initialisierungen von Tensorflow, Numpy und Matplotlib
24 import os
25 os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
26 import tensorflow as tf
27 from matplotlib import pyplot as plt
28 import numpy as np
29 tf.logging.set_verbosity(tf.logging.ERROR) #Deaktivierung von Warnungen
30
31 #Import der MNIST Daten
32 from tensorflow.examples.tutorials.mnist import input_data
33 mnist = input_data.read_data_sets('MNIST_data', one_hot = True)
34
35 #Die Funktion gen_image stellt MNIST Bilder auf dem Bildschirm dar
36 def gen_image(arr):
37     two_d = (np.reshape(arr, (28, 28)) * 255).astype(np.uint8)
38     plt.imshow(two_d, interpolation='nearest')
39     return plt
40
41 # Auswahl zweier zufaelliger Bilder und Darstellung auf dem Bildschirm
42 batch_xs, batch_ys = mnist.test.next_batch(2)
43 gen_image(batch_xs[0]).show()
44 gen_image(batch_xs[1]).show()
45
46 FLAGS = None
47
48 # Hauptprogramm
49 def main(_):
50     # Importieren der Daten
51     mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
52
53     # Create the model
54     x = tf.placeholder(tf.float32, [None, 784]) #Eingangsvariable, Bild in Vektor umgewandelt
55     W = tf.Variable(tf.zeros([784, 10])) #Parametermatrix der Gewichte
56     b = tf.Variable(tf.zeros([10])) #Parametervektor der Bias-Terme
57     h = tf.matmul(x, W) + b #Hypothese, 1x10 Vektor mit Wahrscheinlichkeiten
58
59     # Kostenfunktion und Optimierungsverfahren
60     y = tf.placeholder(tf.float32, [None, 10]) #Ausgangsvariable, 1x10 Vektor
61     # Die folgende Definition der Kostenfunktion ist eine numerisch stabile Version die
62     # Kreuzentropie zu definieren. Eine direkte Implementierung der Formel ist numerisch
63     # problematisch und deshalb nicht zu empfehlen
64     C = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels=y, logits=h))
65     # Auswahl des Gradientenverfahrens als Optimierungsmethode
66     train_step = tf.train.GradientDescentOptimizer(0.5).minimize(C)
67
68     sess = tf.InteractiveSession() #Oeffnen einer interaktiven Session
69     tf.global_variables_initializer().run() #Initialisierung der Variablen: W und b
70
71     # Durchfuehrung des Trainings
72     # Zur Minimierung der Kostenfunktion werden 1000 Iterationen des stochastischen
73     # Gradientenverfahrens durchgefuehrt.
74     # Für einen Update-Schritt werden nur 100 (mini batch size = 100) Trainingsbilder verwendet

```

```

75 # Dadurch wird die Optimierung deutlich schneller, als wenn man in jedem Update-Schritt alle
76 # 55000 Trainingsbilder verwenden wuerde
77 for _ in range(1000):
78     batch_xs, batch_ys = mnist.train.next_batch(100) #zufaellige Auswahl der 100 Bilder
79     sess.run(train_step, feed_dict={x: batch_xs, y: batch_ys})
80
81 # Test des gerade trainierten Modells
82 result = tf.argmax(h, 1) #Berechnungsgraph zur Auswahl der wahrscheinlichsten Ziffer
83 print(sess.run(result, feed_dict={x: mnist.test.images, #Ausfuehren des Berechnungsgraphen
84                                y: mnist.test.labels}))
85 correct_labels = tf.argmax(y, 1) #Berechnungsgraph zur Ermittlung der tatsaechlichen Ziffer
86 print(sess.run(correct_labels, feed_dict={x: mnist.test.images,
87                                y: mnist.test.labels}))
88
89 #Berechnung der Erkennungsrate
90 correct_prediction = tf.equal(tf.argmax(h, 1), tf.argmax(y, 1))
91 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
92 print(sess.run(accuracy, feed_dict={x: mnist.test.images,
93                                y: mnist.test.labels}))
94
95 if __name__ == '__main__':
96     parser = argparse.ArgumentParser()
97     parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data',
98                         help='Directory for storing input data')
99     FLAGS, unparsed = parser.parse_known_args()
100    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

Bis einschließlich Zeile 29 werden lediglich Bibliotheken importiert und Einstellungen vorgenommen. Die Trainingsdaten werden in Zeile 31 und 32 importiert. Zur Veranschaulichung des Datensatzes werden in Zeile 43 und 44 zwei zufällig ausgewählte Ziffern graphisch dargestellt.

Das Training und die Auswertung des Softmax-Klassifikators finden im Hauptprogramm statt, welches in Zeile 49 beginnt. Zur Berechnung der Hypothese h werden zunächst der Platzhalter x für die Eingangsdaten und die TensorFlow-Variablen W und b für die Gewichte und Biases definiert. W ist eine 784×10 Matrix und b ist ein 10×1 Vektor. Der Berechnungsgraph zur Ermittlung der Hypothese wird in Zeile 57 definiert. Der Berechnungsgraph für die Kostenfunktion (Zeile 64) greift auf den Platzhalter y für die Labels und die Funktion `softmax_cross_entropy_with_logits_v2` zurück. Bei letzterer handelt es sich um eine numerisch stabile Implementierung der Kostenfunktion für den Softmax-Klassifikator. Da eine direkte Implementierung der zugehörigen Gleichung numerisch problematisch ist, wird in [22] davon abgeraten.

Nach dem Öffnen einer interaktiven Session und der Initialisierung der Variablen, wird in den Zeilen 77-79 das Training durchgeführt. Dabei kommt das sogenannte statistische Gradientenverfahren [24] zum Einsatz. Im Gegensatz zum herkömmlichen Gradientenverfahren wird dabei in jedem Iterationsschritt der Gradient nur mit Hilfe von relativ wenigen Trainingsvektoren geschätzt (hier mit 100 Vektoren). Dadurch wird die Optimierung deutlich schneller als wenn man in jedem Schritt alle 55000 Trainingsvektoren verwenden würde, ohne dass dabei entscheidende Nachteile bezüglich der Qualität des Optimums in Kauf genommen werden müssen.

Zum Test des trainierten Modells, wird in Zeile 82 zunächst aus dem Hypothesenvektor der Eintrag mit der größten Wahrscheinlichkeit ausgewählt. Der zugehörige Index wird in den Zeilen 89 und 90 mit dem Index der korrekten Ziffer verglichen und gemittelt. Dadurch wird die Erkennungsrate ermittelt, welche für unser Beispiel bei ca 92% liegt. Das klingt zwar ganz passabel, geht aber für die MNIST-Aufgabenstellung noch deutlich besser, wie wir noch sehen werden.

5.2 Vollständiges Feedforward-Netzwerk

Wenn wir wie im vorherigen Abschnitt den Softmax-Klassifikator direkt auf die Pixel der Bilder anwenden, erhalten wir lineare Entscheidungsschwellen in einem hochdimensionalen Merkmalsraum. Daraus ergeben sich einige interessante Fragen:

- Sind lineare Entscheidungsschwellen hier schon ausreichend?

- Sind die rohen Pixel hier gute Merkmale oder lassen sich noch bessere Merkmale für die Klassifikation finden?
- Wie kann man bei der Suche nach besseren Merkmalen systematisch vorgehen?

Eine Antwort auf die letzte Frage liefern neuronale Netze. Neben der bereits bekannten Eingabeschicht und der Ausgabeschicht kommen hier noch verdeckte Schichten hinzu, wie in Abbildung 17 gezeigt. Neuronale Netze mit zwei oder mehr verdeckten Schichten werden auch als “Deep Neural Networks” (DNNs) bezeichnet. Wenn wir Aufgabenstellungen mit DNNs lösen, sprechen wir von “Deep Learning” (DL).

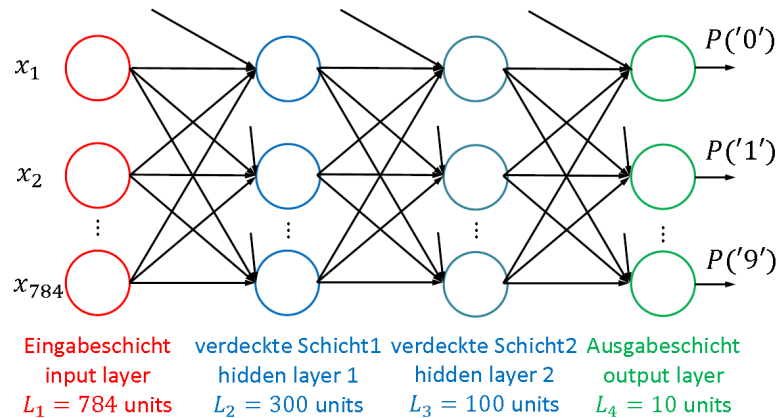


Abbildung 17: Neuronales Netz für die Erkennung handgeschriebener Ziffern (MNIST).

Die verdeckten Schichten eines neuronalen Netzwerks können wir als Verarbeitungseinheiten betrachten, welche aus den Eingangsmerkmalen schrittweise mit jeder verdeckten Schicht bessere Merkmale bestimmen. Die Ausgabeschicht ist ein gewöhnlicher Softmax-Klassifikator, der die von der letzten verdeckten Schicht berechneten Merkmale für die Klassifikation verwendet. Bei einem *Feedforward Neural Network* (FFNN) ist jede Einheit einer Schicht nur mit allen Einheiten der nachfolgenden Schicht über die Gewichte (*Weights*: W) verbunden. Es gibt keine Verbindungen innerhalb einer Schicht und auch keine Verbindungen zu Vorgängerschichten. Abbildung 17 zeigt das FFNN für das MNIST-Problem, welches im folgenden Code `Beispiel_MNIST_FFNN.py` realisiert wird [9].

```
1 # Dieser Code basiert auf
2 # https://github.com/ageron/handson-ml/blob/master/10_introduction_to_artificial_neural_networks.ipynb
3
4 # Importieren von Bibliotheken
5 import numpy as np
6 import os
7 os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
8 import tensorflow as tf
9 tf.logging.set_verbosity(tf.logging.INFO)
10
11 # ***** Konstruktion des Berechnungsgraphen *****
12 # Konstanten, welche die Struktur des Netzwerks festlegen
13 n_inputs = 28*28 #Anzahl der Eingangsvariablen = Anzahl der Pixel pro Bild
14 n_hidden1 = 300 #Anzahl der Einheiten in der ersten verdeckten Schicht
15 n_hidden2 = 100 #Anzahl der Einheiten in der zweiten verdeckten Schicht
16 n_outputs = 10 #Anzahl der Einheiten in der Ausgabeschicht = Anzahl der Klassen
17
18 # Platzhalter für die Eingangsdaten X und die Labels y
19 # Anzahl der Trainings-Datensätze bleibt hier offen (shape=(None))
20 X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
21 y = tf.placeholder(tf.int32, shape=(None), name="y")
22
23 # Laden der Trainings- und Testdaten
24 (X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
25
26 # Anzahl der Trainings-/Test-Datensätze bleibt hier offen (-1)
27 X_train = X_train.astype(np.float32).reshape(-1, 28*28) / 255.0
28 X_test = X_test.astype(np.float32).reshape(-1, 28*28) / 255.0
29 y_train = y_train.astype(np.int32)
30 y_test = y_test.astype(np.int32)
```

```

31
32 # Aufteilen der Trainingsdaten in Trainings- und Validierungsdaten
33 X_valid, X_train = X_train[:5000], X_train[5000:]
34 y_valid, y_train = y_train[:5000], y_train[5000:]
35
36 # Berechnungsgraph fuer das neuronale Netzwerk
37 # mit tf.layers.dense wird jeweils eine Schicht des Netzwerks aufgebaut
38 # hidden1 hat X als Eingang, hidden2 hat hidden1 als Eingang usw.
39 # hidden1 und hidden2 haben ReLu als Aktivierungsfunktion
40 # logits ist die Ausgabeschicht. Hier wird keine Aktivierungsfunktion ausgewaehlt
41 # Deshalb muss nachtraeglich die softmax Operation angewendet werden
42 with tf.name_scope("dnn"):
43     hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
44                             activation=tf.nn.relu)
45     hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
46                             activation=tf.nn.relu)
47     logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
48     y_proba = tf.nn.softmax(logits)
49
50 # Berechnungsgraph fuer die Kostenfunktion
51 with tf.name_scope("loss"):
52     xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
53                                                             logits=logits)
54     loss = tf.reduce_mean(xentropy, name="loss")
55
56 # Berechnungsgraph fuer die Optimierung der Kostenfunktion (eigentliches Training)
57 # gewaehlt wird Gradientenabstieg mit Schrittweite 0.01
58 learning_rate = 0.01
59 with tf.name_scope("train"):
60     optimizer = tf.train.GradientDescentOptimizer(learning_rate)
61     training_op = optimizer.minimize(loss)
62
63 # Berechnungsgraph zur Ermittlung der Erkennungsrate (accuracy)
64 with tf.name_scope("eval"):
65     correct = tf.nn.in_top_k(logits, y, 1)
66     accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
67
68 # ***** Ausfuehrung des Berechnungsgraphen *****
69 # Initialisierung der Variablen
70 init = tf.global_variables_initializer()
71 saver = tf.train.Saver()
72
73 # Parameter fuer das stochastische Gradientenverfahren
74 # In einer Epoche werden alle Trainingsdaten einmal benutzt
75 # batch_size ist die Anzahl der Daten die fuer eine Iteration benutzt werden
76 # Wir haben 55000 Trainingsdaten und batch_size = 50:-> 55000/50=1100 Iterationen pro Epoche
77 # Da wir n_epochs = 40 setzen, benutzen wir alle Trainingsdaten 40mal
78 n_epochs = 40
79 batch_size = 50
80
81 # shuffle_batch mischt die Trainingsdaten durch,
82 # so dass in jeder Epoche die Aufteilung auf die batches
83 # unterschiedlich ist. Dadurch wird die Gefahr, in einem
84 # lokalen Minimum stecken zu bleiben reduziert.
85 # Dadurch verringert sich die Gefahr des Overfittings
86 def shuffle_batch(X, y, batch_size):
87     rnd_idx = np.random.permutation(len(X))
88     n_batches = len(X) // batch_size
89     for batch_idx in np.array_split(rnd_idx, n_batches):
90         X_batch, y_batch = X[batch_idx], y[batch_idx]
91         yield X_batch, y_batch
92
93 # Durchfuehrung des Trainings
94 # Aeussere Schleife ueber die Epochen
95 # Innere Schleife ueber alle Batches innerhalb einer Epoche
96 # sess.run fuehrt eine Iteration des Gradientenabstiegs durch
97 # Fuer jede Epoche wird die Erkennungsrate des letzten Batches
98 # (= Erkennungsrate mit einem Teil der Trainingsdaten)
99 # und die Erkennungsrate fuer die Validierungsdaten ausgegeben
100 with tf.Session() as sess:
101     init.run()
102     for epoch in range(n_epochs):

```



```

103         for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
104             sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
105             acc_batch = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
106             acc_val = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
107             print(epoch, "Batch accuracy:", acc_batch, "Val accuracy:", acc_val)
108
109         save_path = saver.save(sess, "./my_model_final.ckpt")
110
111 # Klassifikation der ersten 20 Testdaten und Vergleich der Ergebnisse mit den Labels
112 with tf.Session() as sess:
113     saver.restore(sess, "./my_model_final.ckpt") # or better, use save_path
114     X_new_scaled = X_test[:20]
115     Z = logits.eval(feed_dict={X: X_new_scaled})
116     y_pred = np.argmax(Z, axis=1)
117
118 print("Predicted classes:", y_pred)
119 print("Actual classes: ", y_test[:20])

```

Die TensorFlow-Implementierung teilt sich wie üblich in die beiden Abschnitte

- Konstruktion des Berechnungsgraphen: Zeile 11 bis 66
- Ausführung des Berechnungsgraphen: Zeile 68 bis 119

auf. In den Zeilen 13 bis 16 werden Python-Konstanten definiert, welche die Struktur des Netzwerkes bestimmen. Nach der Definition der Platzhalter für die Eingangsdaten und die Labels (Zeile 20 und 21) werden die Trainings- und Testdaten geladen. Erstere werden anschließend noch in Trainings- und Validierungsdaten aufgeteilt.

Der Berechnungsgraph für das neuronale Netz wird in den Zeilen 43 bis 48 festgelegt. Es folgen die Berechnungsgraphen für die Kostenfunktion, die Optimierung der Parameter und die Berechnung der Erkennungsrate.

In der Ausführungsphase werden wichtige Hyperparameter für das Training festgelegt. Mit `n_epochs=40` wird festgelegt, dass das Training in 40 Epochen abläuft. In einer Epoche werden alle Trainingsdaten genau einmal verwendet. Das heißt, dass hier alle Trainingsdaten 40mal verwendet werden. Die `batch_size` legt fest, wieviele Trainingsdatenpunkte für die Ermittlung des Gradienten verwendet werden. Nachdem alle Datenpunkte eines "Batches" für die Ermittlung des Gradienten abgearbeitet worden sind, erfolgt ein Update-Schritt des Gradientenverfahrens.

Die Funktion `shuffle_batch` dient dazu, die Trainingsdaten in jeder Epoche neu durchzumischen. Dadurch wird die Gefahr in einem lokalen Minimum der Kostenfunktion stecken zu bleiben sowie die Gefahr des overfittings reduziert [25].

Das eigentliche Training des neuronalen Netzes wird in den Zeilen 100 bis 109 durchgeführt. Die Zeilen 112 bis 119 dienen dazu, die ersten 20 Bilder der Testdaten zu klassifizieren und mit den tatsächlichen Labels zu vergleichen.

Diese Lösung der MNIST-Aufgabenstellung erreicht eine Erkennungsrate von ca 97.6%. Das ist eine erhebliche Verbesserung gegenüber dem Softmax-Klassifikator aber auch noch ein ganzes Stück vom Stand der Technik entfernt, der bei ca. 99.77% liegt [23].

Durchführung

Aufgabe D5:

Erweitern Sie den Code `Beispiel_MNIST_FFNN.py` um eine dritte verdeckte Schicht mit 200 Einheiten und ReLU-Aktivierungsfunktion. Verbessert sich dadurch die Erkennungsrate für die Validierungsdaten?

Aufgabe D6:

Trainieren und testen Sie nach der Vorlage von `Beispiel_MNIST_FFNN.py` ein Feedforward Neural Network zur Klassifikation von Schwertlilien (Blumen, Englischer Name: Iris) anhand der Längen und Breiten der Kelchblätter (Englisch: sepal) und der Kronblätter (Englisch: petal) ihrer Blüten. Bei dieser Machine-Learning-Aufgabe handelt es sich um ein sehr einfaches Klassifikationsproblem, welches in Einführungen sehr weit verbreitet ist und unter dem Namen “iris monitor” bekannt geworden ist. Der zugehörige historische Datensatz [26] ist hier näher beschrieben [27].

- Machen Sie sich zunächst etwas genauer mit der Problemstellung bekannt. Eine schöne Erklärung finden Sie z.B. hier: [28].
- Um den Datensatz aus den csv-Dateien `iris_training.csv` und `iris_test.csv` [29] zu laden, können Sie den folgenden Code `Beispiel_Load_Iris_Data.py` verwenden.

```
1 # Laden der Trainings- und Testdaten
2 IRIS_TRAINING = os.path.join(os.path.dirname(__file__), "iris_training.csv")
3 IRIS_TEST = os.path.join(os.path.dirname(__file__), "iris_test.csv")
4 training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
5     filename=IRIS_TRAINING, target_dtype=np.int32, features_dtype=np.float32)
6 test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
7     filename=IRIS_TEST, target_dtype=np.int32, features_dtype=np.float32)
8
9 X_train = training_set.data
10 y_train = training_set.target
11 X_valid = test_set.data
12 y_valid = test_set.target
```

- Legen Sie eine sinnvolle Struktur des Netzwerks fest. Überlegen Sie, wieviele verdeckte Schichten sinnvoll sind, und wieviele Einheiten diese verdeckte Schichten haben.
- Trainieren Sie ein FFNN mit der oben ermittelten Struktur.
- Ermitteln Sie die Erkennungsrate dieses Netzwerks für die Testdaten.

Weiterführende Literatur

- [1] T. Theiss. *Einstieg in Python*. Rheinwerk Verlag, 2017.
- [2] A.B. Downey. *Programmieren lernen mit Python*. O'Reilly Verlag, 2014.
- [3] B. Klein. *Einführung in Python 3. Für Ein- und Umsteiger*. Carl-Hanser-Verlag, 2017.
- [4] B. Klein. Pythonkurs. <https://www.python-kurs.eu/index.php>. Zuletzt abgerufen: 07.09.2018.
- [5] Python Software Foundation. The python tutorial. <https://docs.python.org/3/tutorial/>. Zuletzt abgerufen: 07.09.2018.
- [6] Michael Weigend. *Python ge-packt*. mitp, 2017.
- [7] J. Ernesti and P. Kaiser. *Python 3: Das umfassende Handbuch: Sprachgrundlagen, Objektorientierte Programmierung, Modularisierung*. Rheinwerk Verlag, 2017.
- [8] H.-B. Woyand. *Python für Ingenieure und Naturwissenschaftler*. Carl-Hanser-Verlag, 2018.
- [9] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017.
- [10] A. Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow*. O'Reilly, 2017. Deutsche Übersetzung eines englischen Buches.

- [11] S. Raschka. *Machine Learning mit Python*. MITP, 2017. Deutsche Übersetzung eines englischen Buches.
- [12] F. Chollet. *Deep Learning with Python*. Manning, 2018.
- [13] J. Frochte. *Machinelles Lernen – Grundlagen und Algorithmen in Python*. Carl-Hanser-Verlag, 2018.
- [14] T. Rashid. *Neuronale Netze selbst programmieren*. O'Reilly, 2017.
- [15] Numpy Community. Numpy user guide. <https://docs.scipy.org/doc/numpy-1.15.1/user/>. Zuletzt abgerufen: 10.09.2018.
- [16] G. Varoquaux et al. Scipy lecture notes. <http://www.scipy-lectures.org/>. Zuletzt abgerufen: 10.09.2018.
- [17] M. Cuhadaroglu U.M. Cakmak. *Mastering Numerical Computing with NumPy*. Packt, 2018.
- [18] J. Evans. Pandas tutorials. <http://pandas.pydata.org/pandas-docs/stable/tutorials.html>. Zuletzt abgerufen: 10.09.2018.
- [19] Scikit-Learn Developers. Scikit-learn documentation. <http://scikit-learn.org/stable/documentation.html>. Zuletzt abgerufen: 10.09.2018.
- [20] TensorFlow Contributors. Getting started with TensorFlow. <https://www.tensorflow.org/tutorials/>. Zuletzt abgerufen: 23.09.2018.
- [21] TensorFlow Contributors. Getting started with TensorFlow. https://github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/docs_src/get_started/get_started.md. Zuletzt abgerufen: 23.09.2018.
- [22] TensorFlow Contributors. MNIST for ML beginners. https://github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/docs_src/get_started/mnist/beginners.md. Zuletzt abgerufen: 23.09.2018.
- [23] Y. LeCun et al. The MNIST database. <http://yann.lecun.com/exdb/mnist/>. Zuletzt abgerufen: 26.09.2018.
- [24] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [25] V. Calomme et al. Data science stack exchange. <https://datascience.stackexchange.com/questions/24511/why-should-the-data-be-shuffled-for-machine-learning-tasks>. Zuletzt abgerufen: 26.09.2018.
- [26] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.
- [27] Wikipedia Contributors. Iris flower data set — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Iris_flower_data_set&oldid=859431235, 2018. [zuletzt abgerufen 27.09.2018].
- [28] TensorFlow Contributors. Getting started with TensorFlow. https://github.com/tensorflow/tensorflow/blob/r1.6/tensorflow/docs_src/get_started/get_started_for_beginners.md. Zuletzt abgerufen: 27.09.2018.
- [29] TensorFlow Contributors. Monitors. <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/tutorials/monitors>. Zuletzt abgerufen: 27.09.2018.