TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Mathematik

Masterarbeit

Analyse der Konvergenzgeschwindigkeit eines einfach berechenbaren Neuronale-Netze-Regressionsschätzers

Adrian Gabel

XX.03.2020

Betreuer: Prof. Dr. Michael Kohler

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Adrian Gabel, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.
Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Darmstadt, 22.02.2020

Adrian Gabel

Inhaltsverzeichnis

Einleitung **6**

1 Grundlagen und Hilfsresultate **7**

1.1 Definitionen 7

1.2 Hilfsresultate 7

2 Konstruktion des Neuronale Netze Regresssionsschätzers **9**

3 Resultat zur Konvergenzgeschwindigkeit **10**

4 Anwendungsbeispiel auf Simulierte Daten **11**

Literaturverzeichnis **13**

Appendix **14**

Einleitung

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer. Diese Arbeit orientiert sich an [EKT+07] und [Koh10].

Kapitel 1

Grundlagen und Hilfsresultate

Der Zweck dieses Kapitels ist es, grundlegende Definitionen zu sammeln, die in den folgenden Kapiteln verwendet werden. Weiterhin werden wir Hilfsresultate darstellen und beweisen welche wir vor allem für das Resultat der Konvergenzgeschwindigkeit des einfach berechenbaren Neuronale Netze Regresssionsschätzer benötigt werden.

1.1 Definitionen

Definition 1.1.1 (Stoppzeit, vgl. [Kle13] 9.15 und 9.16)**.**

(i) Eine *Stoppzeit* τ mit Werten in $[0,T]$ und $T \geq 0$ ist eine bezüglich $(X_t)_{0 \leq t \leq T}$ messbare Funktion. Zusätzlich muss für jedes $r \in [0,T]$ das Ereignis $\{\tau \leq r\}$ in der durch $(X_s)_{0 \leq s \leq r}$ erzeugten σ-Algebra $\mathscr{F}_r = \sigma((X_s)_{0 \leq s \leq r})$ enthalten sein. Die Klasse aller Stoppzeiten mit Werten im Intervall $[0,T]$ bezeichnen wir mit $\mathscr{T}([0,T])$.

(ii) Ist $T \in \mathbb{N}_0$, so nennen wir eine bezüglich $X_0,\ldots,X_T$ messbare Funktion τ mit Werten in $\{0,1,\ldots,T\}$, *Stoppzeit*, wenn zusätzlich für jedes $r \in \{0,\ldots,T\}$ das Ereignis $\{\tau = r\}$ in der durch $X_0,\ldots,X_r$ erzeugten σ-Algebra $\sigma(X_0,\ldots,X_r)$ enthalten ist. Die Klasse aller Stoppzeiten mit Werten in $\{0,\ldots,T\}$ bezeichnen wir mit $\mathscr{T}(0,\ldots,T)$.

1.2 Hilfsresultate

Lemma 1.2.1. *Sei* $\sigma\colon \mathbb{R} \to \mathbb{R}$ *eine Funktion und R,a* > 0.

a) Angenommen σ *ist zwei mal stetigdifferenzierbar und* $t_{\sigma,id} \in \mathbb{R}$ *so, dass sigma*$'(t_{\sigma,id}) \neq 0$ *ist. Dann gilt mit*

$$f_{id}(x) = \frac{R}{\sigma'(t_{\sigma,id})} \cdot \left(\sigma\left(\frac{x}{R} + t_{\sigma,id}\right) - \sigma(t_{\sigma,id})\right)$$

für beliebige $x \in [-a,a]$:

$$|f_{id}(x) - x| \leq \frac{\|\sigma''\|_\infty \cdot a^2}{2 \cdot |\sigma'(t_{\sigma,id})|} \cdot \frac{1}{R}.$$

b) *Angenommen* σ *ist drei mal stetigdifferenzierbar und* $t_{\sigma,sq} \in \mathbb{R}$ *so, dass sigma*$''(t_{\sigma,sq}) \neq 0$ *ist. Dann gilt mit*

$$f_{sq}(x) = \frac{R^2}{\sigma''(t_{\sigma,sq})} \cdot \left(\sigma\left(\frac{2 \cdot x}{R} + t_{\sigma,sq}\right) - 2 \cdot \sigma(\frac{x}{R} + t_{\sigma,sq}) + \sigma(t_{\sigma,sq}) \right)$$

für beliebige $x \in [-a,a]$:

$$|f_{sq}(x) - x^2| \leq \frac{5 \cdot \|\sigma'''\|_\infty \cdot a^3}{3 \cdot |\sigma'(t_{\sigma,sq})|} \cdot \frac{1}{R}.$$

Beweis. a) Sei $u = \frac{c}{R} + t_{\sigma,id}$, $\xi = 0$ und $x \in [-a,a]$beliebig. Wir wissen, dass f_{id} 1-mal differenzierbar ist, da nach Vorraussetzung σ 2-mal stetigdifferenzierbar ist, existiert nach der Restgliedformel von Lagrange (REFERENZ) ein $c \in [\xi, x]$, sodass mit Ausklammern von $\frac{R}{\sigma'(t_{\sigma,id})}$ folgt:

$$\begin{aligned}
|f_{id}(x) - x| \\
&\leq \left| \frac{R}{\sigma'(t_{\sigma,id})} \cdot \left(\sigma\left(\frac{\xi}{R} + t_{\sigma,id}\right) - \sigma(t_{\sigma,id}) + \frac{1}{R}\sigma'\left(\frac{\xi}{R} + t_{\sigma,id}\right)(x - \xi) \right.\right. \\
&\quad \left.\left. + \frac{1}{2R^2}\sigma''(\frac{c}{R} + t_{\sigma,id})(x-\xi)^2) \right) - x \right| \\
&= \left| \frac{R}{\sigma'(t_{\sigma,id})} \cdot \left(\sigma(t_{\sigma,id}) - \sigma(t_{\sigma,id}) + \frac{x}{R}\sigma'(t_{\sigma,id}) + \frac{x^2}{2R^2}\sigma''(\frac{c}{R} + t_{\sigma,id}) \right) - x \right| \\
&= \left| \frac{R}{\sigma'(t_{\sigma,id})} \cdot \left(\frac{x}{R}\sigma'(t_{\sigma,id}) + \frac{x^2}{2R^2}\sigma''(u) \right) - x \right| \\
&= \left| \frac{\sigma''(u) \cdot x^2}{2R \cdot \sigma'(t_{\sigma,id})} + x - x \right| \\
&\leq \frac{\|\sigma''\|_\infty \cdot a^2}{2 \cdot |\sigma'(t_{\sigma,id})|} \cdot \frac{1}{R},
\end{aligned} \tag{1.1}$$

Wobei sich die letzte Ungleichung aus den Eigenschaften der Supremumsnorm ergibt und zudem aus $x \in [-a,a] \Leftrightarrow -a \leq x \leq a$ durch Quadrieren der Ungleichung folgt, dass $x^2 \leq a^2$ ist.

□

Kapitel 2

Konstruktion des Neuronale Netze Regresssionsschätzers

Kapitel 3

Resultat zur Konvergenzgeschwindigkeit

In diesem Kapitel stellen wir das Hauptresultat dieser Arbeit vor, ein Resultat zur Konvergenzgeschwindigkeit des in dieser Arbeit vorgestellten Neuronale Netze Regresssionsschätzers.

Satz 3.0.1. *Unter obigen Voraussetzungen gilt für alle* $t \in \{0, \dots, T\}$ *und* $\mathbb{P}_{X_t}$*-f.s. für alle* $x \in \mathbb{R}^d$

$$V_t(x) = \mathbb{E}\left[f_{\tau^*_{t-1}}(X_{\tau^*_{t-1}}) \mid X_t = x\right] = \max\{f_t(x), q_t(x)\}. \tag{3.1}$$

Weiterhin gilt

$$\mathbf{V_0} = \mathbb{E}\left[f_{\tau^*}(X_{\tau^*})\right]. \tag{3.2}$$

Beweis. □

Kapitel 4

Anwendungsbeispiel auf Simulierte Daten

Literaturverzeichnis

[AHM09] ALBRECHER H., BINDER, A. und P. MAYER: *Einführung in die Finanzmathematik*. Birkhäuser, Basel, 2009.

[Car96] CARRIERE, J.: *Valuation of the early-exercise price for options using simulations and nonparametric regression*. Insurance: mathematics and Economics, 19(1):19–30, 1996.

[EKT+07] EGLOFF, D., M. KOHLER, N. TODOROVIC et al.: *A dynamic look-ahead Monte Carlo algorithm for pricing Bermudan options*. The Annals of Applied Probability, 17(4):1138–1171, 2007.

[Gla13] GLASSERMAN, P.: *Monte Carlo methods in financial engineering*, Band 53 der Reihe *Stochastic Modelling and Applied Probability*. Springer, New York, 2013.

[Kle13] KLENKE, A.: *Wahrscheinlichkeitstheorie*. Springer, Berlin, 2013.

[Koh10] KOHLER, M.: *A review on regression-based Monte Carlo methods for pricing American options*. In: DEVROYE L., KARASÖZEN, B. KOHLER M. und R. KORN (Herausgeber): *Recent Developments in Applied Probability and Statistics*, Seiten 37–58. Physica, Heidelberg, 2010.

[KS98] KARATZAS, I. und S. SHREVE: *Methods of mathematical finance*, Band 39 der Reihe *Stochastic Modelling and Applied Probability*. Springer, New York, 1998.

[LS01] LONGSTAFF, F. und E. SCHWARTZ: *Valuing American options by simulation: a simple least-squares approach*. The review of financial studies, 14(1):113–147, 2001.

[Mal00] MALKIEL, B.: *Börsenerfolg ist kein Zufall - die besten Investmentstrategien für das neue Jahrtausend*. FinanzBuch, München, 2000.

[R D17] R DEVELOPMENT CORE TEAM: *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, 2017.

[SB07] STOER, J. und R. BULIRSCH: *Numerische Mathematik 1*. Springer, Berlin, 2007.

[TVR99] TSITSIKLIS, J. N. und B. VAN ROY: *Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing high-dimensional financial derivatives*. IEEE Transactions on Automatic Control, 44(10):1840–1851, 1999.

Appendix

Der Programmcode ist wie folgt aufgebaut:

- `main.py` TBD.
- `data_gen.py` TBD.
- `help_neural_networks.py` fasst alle Hilfsfunktion zusammen.
- `new_neural_network.py` TBD.
- `fc_neural_network.py` TBD.
- `nearest_neighbor.py` TBD.
- `constant.py` TBD.

Listing 4.1: `main.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 23 15:08:26 2019

@author: adrian

Main Datei die die Simulation und damit den Vergleich der implementierten Schätzer
durchführt.
"""
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib . pyplot as plt
import pandas as pd
from scipy.stats import iqr
from data_gen import gen_data_Y, error_limit
from constant import constant_estimate
from new_neural_network import new_neural_network_estimate
from nearest_neighbor import nearest_neighbor_estimate
from fc_neural_network import fc_neural_1_estimate

'''
EINDIMENSIONALER FALL (d = 1) wird geplottet
'''
N = 3
q = 2
R = 10 ** 6
a = 2
M = 2
d = 1
```

```

sigma = 0.05

n_train = 100
n_test = 2000
# Parameter für unseren neuen Neuronale-Netze-Regressionschätzer
X_train = np.random.uniform(low=-2,high=2,size=(int(n_train),d))
m_X_train, Y_train = gen_data_Y(X_train,sigma)

X_test = np.random.uniform(low=-2,high=2,size=(int(n_test),d))

Y_pred_new_nn = new_neural_network_estimate(X_train, Y_train, X_test, N, q, R, d, M, a,)
#Y_pred_fc_nn_1 = fc_neural_1_estimate(X_train, Y_train, X_test)
#Y_pred_nearest_neighbor = nearest_neighbor_estimate(X_train, Y_train, X_test)
m_X_test, dummy = gen_data_Y(X_test,sigma)


#plt.plot(X_test, Y_pred_nearest_neighbor, '-r', label='nearest_neigbhor')
#plt.plot(X_test, Y_pred_fc_nn_1, '-g', label='fc_nn_1')
#colors = (0,0,0)
area = 4
plt.scatter(X_test, Y_pred_new_nn, s=area, color = 'red', alpha=0.5)
plt.title('Scatter plot pythonspot.com')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
#plt.plot(X_test, Y_pred_new_nn, 'ro-', label='new_nn')
#plt.plot(X_test, m_X_test, '-b', label='m_d')
#plt.legend(loc='upper left')
#plt.xlim(-2, 2)
#plt.xlim(-2,2)
#plt.show()
#plt.savefig('foo.png')

'''
ZEIDIMENSIONALER FALL (d = 2) wird geplottet
'''
N = 2
q = 2
R = 10 ** 6
a = 2
M = 2
d = 2

sigma = 0.05

n_train = 100
n_test = 2000
# Parameter für unseren neuen Neuronale-Netze-Regressionschätzer
X_train = np.random.uniform(low=-2,high=2,size=(int(n_train),d))
m_X_train, Y_train = gen_data_Y(X_train,sigma)

X_test = np.random.uniform(low=-2,high=2,size=(int(n_test),d))

Y_pred_new_nn = new_neural_network_estimate(X_train, Y_train, X_test, N, q, R, d, M, a,)
#Y_pred_fc_nn_1 = fc_neural_1_estimate(X_train, Y_train, X_test)
#Y_pred_nearest_neighbor = nearest_neighbor_estimate(X_train, Y_train, X_test)
m_X_test, dummy = gen_data_Y(X_test,sigma)



x = np.ravel(X_test[:,0])
y = np.ravel(X_test[:,1])

# so wie es sein soll
#z = m_X_test[:,0]
# was der SChätzer auswirft
z = Y_pred_new_nn[:,0]

ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
ax.view_init(40, 20)
fig

'''
```

```
ein Vergleich des emp. L2 Fehler gemacht für d = 1
'''
n_train = 100
n_test = 2000
# Parameter für unseren neuen Neuronale−Netze−Regressionschätzer

N = 3
q = 2
R = 10 ** 6
a = 2
M = 2
d = 1

sigma = 0.05

scaled_error = np.empty((5, 3,))
scaled_error[:] = np.nan

e_L2_avg = np.zeros(5)
e_L2_avg[:] = np.nan

for i in range(0,5,1):

    X_train = np.sort(np.random.uniform(low=−2,high=2,size=(int(n_train),d)), axis = 0)
    m_X_train, Y_train = gen_data_Y(X_train,sigma)

    X_test = np.sort(np.random.uniform(low=−2,high=2,size=(int(n_test),d)), axis = 0)

    #Y_pred_constant = constant_estimate(Y_train)
    Y_pred_new_nn = new_neural_network_estimate(X_train, Y_train, X_test, N, q, R, d, M, a,)
    Y_pred_fc_nn_1 = fc_neural_1_estimate(X_train, Y_train, X_test)
    Y_pred_nearest_neighbor = nearest_neighbor_estimate(X_train, Y_train, X_test)

    m_X_test, not_needed = gen_data_Y(X_test,sigma)

    e_L2_new_nn = np.mean(sum (abs(Y_pred_new_nn − m_X_test) ** 2))
    e_L2_fc_nn_1 = np.mean(sum (abs(Y_pred_fc_nn_1 − m_X_test) ** 2))
    e_L2_nearest_neighbor = np.mean(sum (abs(Y_pred_nearest_neighbor − m_X_test) ** 2))

    for j in range(0,5,1):

        X = np.sort(np.random.uniform(low=−2,high=2,size=(n_test,d)), axis = 0)
        m_X, Y = gen_data_Y(X,sigma)
        Y_pred_constant = constant_estimate(Y)

        e_L2_avg[j] = np.mean(sum(abs(Y_pred_constant − m_X) ** 2))

    scaled_error[i,0] = e_L2_new_nn / np.median(e_L2_avg)
    scaled_error[i,1] = e_L2_fc_nn_1 / np.median(e_L2_avg)
    scaled_error[i,2] = e_L2_nearest_neighbor / np.median(e_L2_avg)

iqr_new_nn = iqr(scaled_error[:,0])
iqr_fc_nn_1 = iqr(scaled_error[:,1])
iqr_nearest_neighbor = iqr(scaled_error[:,2])

median_new_nn = np.median(scaled_error[:,0])
median_fc_nn_1 = np.median(scaled_error[:,1])
median_nearest_neighbor = np.median(scaled_error[:,2])

rows = ["noise","e_L2_avg","approach","new_nn", "fc_nn_1", "nearest_neighbor"]

series_noise_1 = pd.Series([repr(sigma)+'%',np.median(e_L2_avg),"(Median, IQR)",(median_new_nn, iqr_new_nn), (median_fc_nn_1, iqr_fc_nn_1), (median_nearest_neighbor, iqr_nearest_neighbor)], index=rows)
series_noise_1.name = ""
#series_noise_2 = pd.Series([repr(sigma)+'%',np.median(e_L2_avg),"(Median, IQR)",(median_new_nn, iqr_new_nn), (median_fc_nn_1, iqr_fc_nn_1), (median_nearest_neighbor, iqr_nearest_neighbor)], index=rows)
#series_noise_2.name = ""

error_df = pd.concat([series_noise_1], axis=1)
print(error_df)
#error_df.to_csv('out.csv',index = True)

'''
ein Vergleich des emp. L2 Fehler gemacht für d = 2
'''
```

```
n_train = 100
n_test = 2000
# Parameter für unseren neuen Neuronale-Netze-Regressionschätzer

N = 2
q = 2
R = 10 ** 6
a = 2
M = 2
d = 2

sigma = 0.05

scaled_error = np.empty((5, 3,))
scaled_error[:] = np.nan

e_L2_avg = np.zeros(5)
e_L2_avg[:] = np.nan

for i in range(0,5,1):

    X_train = np.sort(np.random.uniform(low=-2,high=2,size=(int(n_train),d)), axis = 0)
    m_X_train, Y_train = gen_data_Y(X_train,sigma)

    X_test = np.sort(np.random.uniform(low=-2,high=2,size=(int(n_test),d)), axis = 0)

    #Y_pred_constant = constant_estimate(Y_train)
    Y_pred_new_nn = new_neural_network_estimate(X_train, Y_train, X_test, N, q, R, d, M, a,)
    Y_pred_fc_nn_1 = fc_neural_1_estimate(X_train, Y_train, X_test)
    Y_pred_nearest_neighbor = nearest_neighbor_estimate(X_train, Y_train, X_test)

    m_X_test, not_needed = gen_data_Y(X_test,sigma)

    e_L2_new_nn = np.mean(sum (abs(Y_pred_new_nn - m_X_test) ** 2))
    e_L2_fc_nn_1 = np.mean(sum (abs(Y_pred_fc_nn_1 - m_X_test) ** 2))
    e_L2_nearest_neighbor = np.mean(sum (abs(Y_pred_nearest_neighbor - m_X_test) ** 2))

    for j in range(0,5,1):

        X = np.sort(np.random.uniform(low=-2,high=2,size=(n_test,d)), axis = 0)
        m_X, Y = gen_data_Y(X,sigma)
        Y_pred_constant = constant_estimate(Y)

        e_L2_avg[j] = np.mean(sum(abs(Y_pred_constant - m_X) ** 2))

    scaled_error[i,0] = e_L2_new_nn / np.median(e_L2_avg)
    scaled_error[i,1] = e_L2_fc_nn_1 / np.median(e_L2_avg)
    scaled_error[i,2] = e_L2_nearest_neighbor / np.median(e_L2_avg)

iqr_new_nn = iqr(scaled_error[:,0])
iqr_fc_nn_1 = iqr(scaled_error[:,1])
iqr_nearest_neighbor = iqr(scaled_error[:,2])

median_new_nn = np.median(scaled_error[:,0])
median_fc_nn_1 = np.median(scaled_error[:,1])
median_nearest_neighbor = np.median(scaled_error[:,2])

rows = ["noise","e_L2_avg","approach","new_nn", "fc_nn_1", "nearest_neighbor"]

series_noise_1 = pd.Series([repr(sigma)+'%',np.median(e_L2_avg),"(Median, IQR)",(median_new_nn, iqr_new_nn), (median_
    fc_nn_1, iqr_fc_nn_1), (median_nearest_neighbor, iqr_nearest_neighbor)], index=rows)
series_noise_1.name = ""
#series_noise_2 = pd.Series([repr(sigma)+'%',np.median(e_L2_avg),"(Median, IQR)",(median_new_nn, iqr_new_nn), (median
    _fc_nn_1, iqr_fc_nn_1), (median_nearest_neighbor, iqr_nearest_neighbor)], index=rows)
#series_noise_2.name = ""

error_df = pd.concat([series_noise_1], axis=1)
print(error_df)
#error_df.to_csv('out.csv',index = True)
```

Listing 4.2: `data_gen.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
"""
Created on Fri Oct 11 12:01:42 2019

@author: adrian

Generieren der Daten die wir für einen Vergleich von Regressionsschätzern benötigen
"""
# Wir wählen x gleichverteilt auf [-1,1]^d, wobei d die dimension des Inputs ist
# n is die Größe der Stichprobe

import numpy as np
from scipy.stats import iqr

# Regressionsfunktionen
#
# x: Ein Vektor x \in [-1,-1]^d
# d: Dimension des Vektors x

def m_d (x, d):

    pi = np.pi
    cos = np.cos
    sin = np.sin
    exp = np.exp

    if d == 1:
        return sin(0.2 * x[0] ** 2) + exp(0.5 * x[0]) + x[0] ** 3

    elif d == 2:
        return np.sin(np.sqrt(x[0] ** 2 + x[1] ** 2))

    else:
        print("Your data has the wrong dimension!")

def error_limit (x, p, c, d):

        return c * (np.log(x) ** 3) * (x ** (-(2 * p)/(2 * p + d)))

# Generiert den Vektor Y_1,...,Y_n für den Datensatz (X_1,Y_1),...,(X_n,Y_n)
#
# X: Inputdaten der Form (X_1,...,X_n), wobei X_i \in [-1,-1]^d für i = 1,...,n
# sigma: Schwankung in den Werten (Noise) \in \{0.05,0.1\}

def gen_data_Y (X, sigma):

    n = np.size(X, 0)
    d = np.size(X, 1)

    m_X = np.zeros((n,1,))
    m_X[:] = np.nan

    S = np.random.standard_normal(size=(n,1))
    for t in range(0,n):
        m_X[t] = m_d(X[t], d)

    Y = m_X + sigma * iqr(m_X) * S
    return (m_X, Y)
```

Listing 4.3: `help_neural_networks.py`

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 15 11:22:02 2019

@author: adrian

Implementation von Neuronalen-Netzen welche wir für die Konstruktion unseres
Neuronale-Netze-Regressionschätzers benötigen
"""
import numpy as np

# Sigmoidfunktion
#
```

```
# x: x \in \R

def sigmoid (x):

    return 1 / (1 + np.exp(-x))

# Neuronales Netz welches die Funktion f(x) = x approximiert
#
# x: reelle Zahl
# R: reelle Zahl >= 1

def f_id (x, R):

    return 4 * R * sigmoid(x / R) - 2 * R

# Neuronales Netz welches die Funktion f(x, y) = x * y approximiert
#
# x: reelle Zahl
# y: reelle Zahl
# R: reelle Zahl >= 1

def f_mult (x, y, R):

    return (((R ** 2) / 4) * (((1 + np.exp(-1)) ** 3) / (np.exp(-2) - np.exp(-1)))) \
    * (sigmoid(((2 * (x + y)) / R) + 1) - 2 * sigmoid(((x + y) / R) + 1) \
    - sigmoid(((2 * (x - y)) / R) + 1) + 2 * sigmoid(((x - y) / R) + 1))

# Neuronales Netz welches die Funktion f(x) = max(x,0) approximiert
#
# x: reelle Zahl
# R: reelle Zahl >= 1

def f_relu (x, R):

    return f_mult(f_id(x, R), sigmoid(R * x),R)

# Neuronales Netz welches die Funktion f(x) = max(1 - (M/(2 * a)) * abs(x - y),0) approximiert
#
# x: reelle Zahl
# y: fixe reelle Zahl
# R: reelle Zahl >= 1
# M: fixe natürliche Zahl
# a: fixe Zahl > 0

def f_hat (x, y, R, M, a):

    return f_relu((M / (2 * a)) * (x - y) + 1, R) - 2 * f_relu((M / (2 * a)) * (x - y), R) + 2 * f_relu((M / (2 * a))
          * (x - y) - 1, R)
```

Listing 4.4: `new_neural_network.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 16 15:40:14 2019

@author: adrian

Um die Gewichte der Ausgabeschicht zu bestimmen lösen wir ein regularisiertes
Kleinste-Quadrate Problem.

"""
import scipy.special
import numpy as np
import itertools
from help_neural_networks import f_id, f_mult, f_hat
import math

# Neuronales Netz welches die Funktion f(x) = (x^(1)- x_ik^(1))^j1 * ... *
# (x^(d) - x_ik^(d))^jd * \prod_{j = 1}^d max((1 - (M/2a) * abs(x^(j) - x_ik^(j))),0)
#
# x: Eingabevektor für das Neuronale Netz x \in [-a,a]^d
# d: Ist die Dimension des Eingabevektors d > 0
# j_1_d: Ist ein d-dimensionaler Vektor j_1,...,j_d \in {0,1,...,N}
```

```
# X_i: Ist eine  d x (M+1)^d  Matrix.
# N: Natürliche Zahl >= q
# q:
# s: [log_2(N + d)]
# R: Zahl >= 1
# M: M \in N
# a: > 0

def f_net (x, d, j_1_d, X_i, N, q, s, R, M, a):
#initialize f_l_k
    #(2 ** s) + 1
    #((1 + M) ** d) + 1
    f_l_k = np.empty((s + 1, (2 ** s) + 1,))
    f_l_k[:] = np.nan


    for k in range(np.sum(j_1_d) + d + 1, (2 ** s) + 1, 1):
        f_l_k[s, k] = 1

    for k in range(1, d + 1, 1):
        f_l_k[s, np.sum(j_1_d) + k] = f_hat(x[k - 1], X_i[k - 1], R, M, a)

    for l in range(1, d + 1, 1):
        k = j_1_d[range(0, l - 1, 1)].sum() + 1
        while k in range(j_1_d[range(0, l - 1, 1)].sum() + 1, j_1_d[range(0, l, 1)].sum() + 1, 1):
            f_l_k[s, k] = f_id(f_id(x[l - 1] - X_i[l - 1], R), R)
            k += 1

    for l in range(s - 1, -1, -1):
        for k in range((2 ** l), 0, -1):
            f_l_k[l, k] = f_mult(f_l_k[l + 1, (2 * k) - 1], f_l_k[l + 1, 2 * k], R)

    return f_l_k[0,1]

# Bestimmung der Gewichte der Ausgabeschicht durch lösen eines regularisierten
# Kleineste-Quadrate Problems
#
# X: Eingabevektoren der Form (X_1,...,X_n) für das Neuronale Netz aus dem Datensatz (X_1,Y_1),...,(X_n,Y_n)
# Y: Eingabevektoren der Form (Y_1,...,Y_n) für das Neuronale Netz aus dem Datensatz (X_1,Y_1),...,(X_n,Y_n)
# N: Natürliche Zahl >= q
# q:
# R: Zahl >= 1
# d: Ist die Dimension des Eingabevektors d > 0
# M: M \in \N
# a: >0

def output_weights(X, Y, N, q, R, d, M, a):

    s = math.ceil(math.log2(N + d))

    # Anzahl der Eingabevektoren X_1,...,X_n

    n = np.size(X, 0)

    # Eine beliebige constante > 0

    #c_3 = np.random.randint(1,10)
    c_3 = 0.01

    # Anzahl der Spalten der Matrix für das Kleinste-Quadrate Problem
    # In den Spalten sind die Funktionswerte von f_net eingespeichert

    J = int(((1 + M) ** d) * scipy.special.binom(N + d, d))

    # Für die Konstruktion der Matrix brauchen wir erstmal alle Inputparameter
    # für f_net, da wir dort nur den Funktionswert für einen Vektor j_1,...,j_d einsetzen
    # müssen wir erstmals alle möglichen Vektoren dieser Art konstruieren die die Bedingung 0 <= j_1 + ... + j_d <= N
        erfüllen
    # X_ik hat in den Zeilen die Vektoren X_i aus dem Paper

   X_ik = np.transpose(np.empty((d, (1 + M) ** d,)))
   X_ik[:] = np.nan

    I_k = np.array(list(itertools.product(range(0, M + 1), repeat = d)))
   X_ik[:] = (I_k[:] * ((2 * a) / M)) - a
```

```
    all_j1_jd = np.array(list(itertools.product(range(0, N + 1), repeat = d)))
    all_j1_jd_by_cond = all_j1_jd[all_j1_jd.sum(axis=1) <= N]

    B = np.empty((n, J,))
    B[:] = np.nan

    for i in range(0, n):
        j = 0
        for k in range(0, ((M + 1) ** d)):
            for z in range(0, int(scipy.special.binom(N + d, d))):
                B[i,j] = f_net(X[i], d, all_j1_jd_by_cond[z], X_ik[k], N, q, s, R, M, a)
                j += 1

    #weights = np.linalg.solve((1 / n) * np.dot(np.transpose(B),B) + (c_3 / n) * np.identity(J), (1 / n) * np.dot(np.transpose(B),Y))
    weights = np.linalg.solve(np.dot(np.transpose(B),B) + (c_3) * np.identity(J), np.dot(np.transpose(B),Y))

    return (weights, J, all_j1_jd_by_cond, X_ik)

# Bestimmung des Funktionswert des Neuronale-Netzte-Regressionsschätzers
#
# x: Eingabe für einen Vektor der Form [-a,a]^d für den eine Schätzung bestimmt werden soll
# X: Eingabevektoren der Form (X_1,...,X_n) für das Neuronale Netz aus dem Datensatz (X_1,Y_1),...,(X_n,Y_n)
# Y: Eingabevektoren der Form (Y_1,...,Y_n) für das Neuronale Netz aus dem Datensatz (X_1,Y_1),...,(X_n,Y_n)
# N: Natürliche Zahl >= q
# q:
# s: [log_2(N + d)]
# R: Zahl >= 1
# d: Ist die Dimension des Eingabevektors d > 0
# M: M \in \N
# a: >0

def new_neural_network_estimate(X_train, Y_train, X_test, N, q, R, d, M, a):

    Y_pred = np.empty((len(X_test), 1,))
    Y_pred[:] = np.nan

    s = math.ceil(math.log2(N + d))

    weights, J, all_j1_jd_by_cond, X_ik = output_weights(X_train, Y_train, N, q, R, d, M, a)

    F_net = np.empty((1, J,))
    F_net[:] = np.nan

    for u in range (0,len(X_test),1):
        j = 0
        while j < J:
            for k in range(0, ((M + 1) ** d)):
                for z in range(0, int(scipy.special.binom(N + d, d))):
                    F_net[0,j] = f_net(X_test[u], d, all_j1_jd_by_cond[z], X_ik[k], N, q, s, R, M, a)
                    j += 1

        Y_pred[u] = np.sum(np.transpose(weights) * F_net)

    return Y_pred
```

Listing 4.5: `fc_neural_network.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 11 14:23:15 2019

@author: adrian
"""
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Fully-Connected Neuronales Netzt mit einer Verborgenen schicht welches die
# Anzahl der Neuronen adaptiv, durch minimierung des L2 fehlers, aus der Menge \{5, 10, 25, 50, 75\} auswählt.
#
# X: Eingabevektoren der Form (X_1,...,X_n) für das Neuronale Netz aus dem Datensatz (X_1,Y_1),...,(X_n,Y_n)
```

```
# Y: Eingabevektoren der Form (Y_1,...,Y_n) für das Neuronale Netz aus dem Datensatz (X_1,Y_1),...,(X_n,Y_n)

def fc_neural_1_estimate (X_train,Y_train,X_test):

    Ynew = np.empty((len(X_train), len([5,10,25,50,75]),))
    Ynew[:] = np.nan

    count = 0
    n_neurons = [5,10,25,50,75]

    d = np.size(X_train, 1)

    for j in n_neurons:
        model = Sequential()
        model.add(Dense(j, input_dim=d, activation='relu'))
        model.add(Dense(1, activation='linear'))
        model.compile(loss='mse', optimizer='adam')
        model.fit(X_train, Y_train, epochs=1000, verbose=0)

        Ynew[:,count] = model.predict(X_train)[:,0]
        count += 1

    Diff = Ynew[:] - Y_train[:]
    best_n_neurons = n_neurons[(1/len(X_train) *(Diff.sum(axis=0) ** 2)).argmax()]

    model = Sequential()
    model.add(Dense(best_n_neurons, input_dim=d, activation='relu'))
    model.add(Dense(1, activation='linear'))
    model.compile(loss='mse', optimizer='adam')
    model.fit(X_train, Y_train, epochs=1000, verbose=1)

    return model.predict(X_test)
```

Listing 4.6: `nearest_neighbor.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 11 11:14:47 2019

@author: adrian
K-Nearest-Neighbors Algorithm
"""

# Generate sample data
import numpy as np
from sklearn import neighbors
from sklearn.model_selection import GridSearchCV
import warnings

warnings.simplefilter(action='ignore', category=FutureWarning)

# Implementierung des k-Nächste-Nachbarn-Algorithmus. Dieser bestimmt auch selber bei einer Liste von Anzahlen an
      Nachbarn die betrachtet werden
# sollen welches die beste Wahl ist.
#
# X: Inputvektor für das Kalibirieren des Modells
# Y: Inputvektor für das Kalibirieren des Modells (Zielvektor an den die Gewichte angepasst werden)
# T: Inputvektor für den eine Vorhersage bestimmte werden soll

def nearest_neighbor_estimate (X_train,Y_train,X_test):

    params = {'n_neighbors':[2,3,4,5,6,7,8,9], 'weights': ['uniform', 'distance']}

    knn = neighbors.KNeighborsRegressor()

    knn_gridsearch_model = GridSearchCV(knn, params, cv=5)
    knn_gridsearch_model.fit(X_train,Y_train)

    return knn_gridsearch_model.predict(X_test)
```

Listing 4.7: `constant.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 23 15:26:19 2019

@author: adrian
Constant Estimator
"""
import numpy as np
from scipy import mean

# Gibt den Mittelwert der Funktionswerte einer Funktion als Schätzer zurück
#
# Y: Datensatz der Form (Y_1,...) wobei Y_i \in \R für i = 1,...

def constant_estimate(Y):
    m = np.zeros((len(Y),1,))
    m[:] = mean(Y)
    return m
```