

Technische Hochschule Rosenheim

Fakultät für Informatik

## Seminararbeit

im Masterstudiengang Informatik - Schwerpunkt Software Engineering

**Thema:** Optimierungsmethoden des Gradienten  
Abstiegverfahrens bei neuronalen Netzen

**Autor:** Victor Wolf victorwolf@outlook.de  
MatNr. 845615

**Version vom:** 27. Dezember 2019

**Betreuer:** Prof. Dr. Holaubek

## Zusammenfassung

Diese Arbeit wird sich mit verschiedenen Optimierungsmethoden neuronaler Netze beschäftigen und Sie auf Basis von Beispiel Datensätzen evaluieren, wie dem Boston House Price Datensatz. Hierbei wird eine Metrik definiert um die Ergebnis der einzelnen Optimierungsmethode zu vergleichen.

Eine Optimierungsmethode ist eine Möglichkeit die Konvergenz der Fehlerfunktion  $J(\theta)$  des neuronalen Netzes beim Lernen zu verbessern.

Hierbei wird auf den Lern Prozess des Neuronalen Netzes eingegangen. Besonderen Fokus wird der “Gradient Descent”, zu Deutsch Gradienten Abstiegsverfahren, einnehmen, da dies die Grundlage des Lernens darstellt. Dieser sucht im mehrdimensionalen Raum die Minima der nichtlinearen Fehlerfunktion und es gibt verschiedene Möglichkeiten diese Suche zu verbessern. Nach der theoretischen Aufarbeitung, werden wir ein paar Eigenschaften über diese Optimierungsmethoden annehmen und diese anhand der Test Daten überprüfen.

## Abstract

This work will focus on explaining the different optimization methods of neural networks and evaluating them on example datasets like the boston house price data. We will define a metric to be able to evaluate the performance of the different optimizer.

An optimization method is a way to improve the performance of the error function  $J(\theta)$  of the neural network.

Furthermore this work will give a detailed explanation of the learning process of Neural Networks especially focusing on the Gradient Descent, which is the foundation of learning in neural networks. This algorithm aims to find local minima in the Hyperplane of the non-linear error function and there are multiple ways to improve its search. After the Theory, we will assume some properties about those optimization methods and test those assumptions by evaluating the metrics of these neural networks.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>4</b>
<b>2</b>	<b>Theoretische Grundlagen</b>	<b>4</b>
2.1	Neuronale Netze . . . . .	4
2.2	Gradient Descent . . . . .	6
2.3	Optimierungsmethoden . . . . .	7
2.3.1	Stochastic Gradient Descent . . . . .	7
2.3.2	Adagrad . . . . .	8
2.3.3	Adam . . . . .	8
<b>3</b>	<b>Evaluation</b>	<b>9</b>
3.1	Test Datensatz . . . . .	9
3.1.1	Boston House Price . . . . .	9
3.1.2	Breast Cancer . . . . .	10
3.2	Metrik . . . . .	11
3.3	Programm . . . . .	12
3.3.1	Frameworks . . . . .	12
3.3.2	Aufbau . . . . .	13
3.4	Auswertung . . . . .	13
<b>4</b>	<b>Fazit</b>	<b>13</b>
	<b>Literaturverzeichnis</b>	<b>14</b>
	<b>Eidesstattliche Erklärung</b>	<b>15</b>

# 1 Einleitung

## 2 Theoretische Grundlagen

### 2.1 Neuronale Netze

Künstliche Neuronale Netze kurz KNNs sind der menschliche Versuch das biologische Nervensystem nachzuahmen. Sie basieren auf der Tatsache der Reizweitergabe. So wird ein Eingangsreiz von Rezeptoren aufgenommen und über verschiedene sogenannter Neuronen weitergegeben. Durch diese Weitergabe wird das Signal verändert, bis ein Ausgangssignal interpretiert werden kann. Diese Funktionsweise macht man sich bei künstlichen Neuronalen Netzen zu nutze. Der Eingangsreiz sind hier die sogenannten “features“, der Ausgangsreiz eine Klasse oder ein Wert der interpretiert werden kann. Wir wollen uns hier nun nur auf die “Feed Forward“ Netze fokussieren. Das bedeutet das Neuronen ihre Ausgabe nur in eine Richtung schicken dürfen.

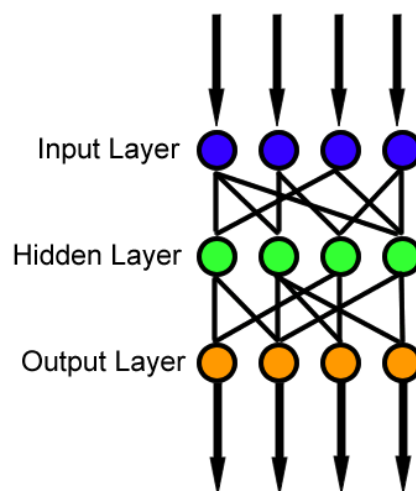


Abbildung 1: Beispiel eines Feed Forward KNNs

KNNs existieren in zwei Zuständen der Trainingsphase und der Arbeitsphase. Die Trainingsphase ist die interessantere und wird in dieser Arbeit beleuchtet. Hier werden durch Optimierung der Fehlerfunktion die Neuronen so “eingestellt”, dass sie einen möglichst gute Vorhersage treffen.

Im Folgenden soll nun der Begriff des Neurons formalisiert werden, um die Verbesserungsmöglichkeiten des Gradienten Abstiegsverfahrens in Abschnitt 2.3 nachvollziehen zu können.

**Definition 2.1** [Bur97, Kapitel 1.2] Ein (**formales**) **Neuron** ist eine Funktion  $\kappa : \mathbb{R}^n \rightarrow \mathbb{R}^m$  definiert durch:

- eine Aktivierungsfunktion  $T : \mathbb{R} \rightarrow \mathbb{R}$

- ein gewichteter Vektor  $\vec{w} = \{w_1, w_2, \dots, w_n\}$
- und eine Schwelle  $\Theta \in \mathbb{R}$ .

Der Vektor  $\vec{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$  wird auf den Vektor  $\vec{y} = (y, y, \dots, y) \in \mathbb{R}^m$  mit identischen Komponenten durch die folgende Rechenvorschrift abgebildet

$$\kappa(\vec{x}) := (T(\sum_{i=1}^n w_i x_i - \Theta), \dots, T(\sum_{i=1}^n w_i x_i - \Theta)) = \vec{y} \in \mathbb{R}^m \quad (1)$$

Hier seien ein paar Beispiele für Aktivierungsfunktionen angegeben

- Identität  $T_I$

$$T(x) := x = T_I(x)$$

- Binary step

$$T(x) := \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0 \end{cases} =: T_1(x)$$

- Sigmoid

$$T(x) := \frac{1}{1 + e^{-x}} =: T_S(x)$$

- Tangens hyperbolicus

$$T(x) := \frac{1 + \tanh(x)}{2} =: T_H(x)$$

Dies sind nur ein paar wenige Beispiele. Jede Funktion  $T : \mathbb{R} \rightarrow \mathbb{R}$  die  $\lim_{x \rightarrow -\infty} T(x) = 0$  and  $\lim_{x \rightarrow \infty} T(x) = 1$  erfüllt, kann als Aktivierungsfunktion genutzt werden.

**Definition 2.2** [Mic, Kapitel 2] Die **Fehlerfunktion**  $J(\theta)$  eines Neuronalen Netzes ist eine differenzierbare Funktion für die gilt:

- $J(\theta) = \frac{1}{n} \sum_x J_x$  wobei  $x$  ein Eingabe Datum beschreibt.
- $J(\theta)$  lässt sich aus der Summe der Elemente des Ausgabe Vektors darstellen.

Die erste Eigenschaft bedeutet, dass die gesamte Fehlerfunktion sich auch durch die Fehlerfunktion der einzelnen Eingabe Daten darstellen lässt. Diese Fehlerfunktion wird im nächsten Abschnitt minimiert werden, um eine optimale Parameterbelegung der Gewichte  $\vec{w}$  zu finden. Beispiel für eine solche Funktion wäre der mittlere quadratische Fehler.

## 2.2 Gradient Descent

Der Gradient Descent oder zu Deutsch Gradienten Abstiegsverfahren ist ein Weg eine Zielfunktion  $J(\theta)$  parametrisiert durch  $\theta \in \mathbb{R}^n$  zu minimieren. Man aktualisiert diese Parameter in Richtung des stärksten Abstiegs der Zielfunktion  $\nabla_{\theta} J(\theta)$ . Die Lern Rate  $\mu$  bestimmt dabei die Größe der Aktualisierungsschritte. Das Verfahren folgt also der Richtung des Abstiegs der Oberfläche der Zielfunktion in ein Tal, welches ein lokales Minimum beschreibt. [Rud, Kapitel 1]

Im Fall eines neuronalen Netzes ist die Zielfunktion  $J(\theta)$  die Fehlerfunktion des neuronalen Netzes. Wir brauchen einen solchen Algorithmus, da durch die Aktivierungsfunktion der Neuronen wie in 2.1 beschrieben, die Fehlerfunktion nichtlinear wird und somit die Minima sich nicht mehr analytisch berechnen lassen. Um die Richtung des stärksten Abstiegs der Fehlerfunktion zu bestimmen, benötigen wir den Gradienten.

**Definition 2.3** [Kön02] Der **Gradient**  $\nabla$  der total differenzierbaren Funktion  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  im Punkt  $a \in \mathbb{R}$  ist im Falle des Standard Skalar Produkts definiert durch:

$$\nabla f := \frac{\partial f}{\partial x_1} \hat{e}_1 + \dots + \frac{\partial f}{\partial x_n} \hat{e}_n \quad (2)$$

In einfachen Worte gefasst, ist der Gradient die Ableitung einer mehrdimensionalen Funktion, deren Funktionswerte man sich als Gebirge vorstellen kann. Hierbei ist der Gradient in einen Punkt ein Vektor der in die Richtung des stärksten Anstiegs.

Um die Richtung des stärksten Abstiegs zu erhalten, welche wir beim Gradienten Abstiegsverfahren benötigen, müssen wir nur den negativen Gradienten berechnen.

Mit diesem Wissen können wir nun den Standard gradient descent algorithmus definieren.

**Definition 2.4** [Rud, Kapitel 2.1] Der sogenannte **batch gradient descent**, berechnet den Gradienten der Kostenfunktion für den gesamten Datensatz. Jedes Update der Parameter ist definiert durch

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \quad (3)$$

wobei  $\eta$  die Lern Geschwindigkeit beschreibt.

Als Algorithmus würde der batch gradient descent folgendermaßen aussehen.

```

1 for i in range(nb_epochs):
2     params_grad = evaluate_gradient(loss_function, data, params)
3     params = params - learning_rate * params_grad

```

wobei `nb_epochs` die Anzahl der Iterationen beschreibt. Diese Implementierung beschreibt die grundsätzliche Idee aber hat mehrere Nachteile. Da wir den Gradienten für den gesamten Datensatz berechnen ist diese Methode sehr langsam und nicht möglich für Datensätze, die nicht in den Arbeitsspeicher passen. Außerdem konvergiert dieser Algorithmus nur sehr langsam gegen ein lokales Minimum. Zusätzlich ist die Wahl der perfekten Lern Geschwindigkeit oft schwierig.

Zusätzlich benötigt man den Backpropagation Algorithmus um dieses Verfahren auf ein gesamtes neuronales Netz anzuwenden. Dieser ist aber ebenfalls sehr komplex und wird deshalb hier nicht behandelt. Er kann in [Mic, Kapitel 2] nachgelesen werden.

## 2.3 Optimierungsmethoden

Im vorherigen Abschnitt haben wir die Grundlagen des Gradienten Abstiegsverfahren kennengelernt und gesehen, dass dieses Probleme mit sich bringt. Die folgenden Optimierungsmethoden verbessern das grundsätzliche batch gradient descent Verfahren in die ein oder andere Richtung.

### 2.3.1 Stochastic Gradient Descent

**Definition 2.5** [Rud, Kapitel 2.2] Der Aktualisierungsschritt des *stochastic gradient descent* ist definiert durch

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}) \quad (4)$$

Der Unterschied zum **batch gradient descent** ist nur, dass der Aktualisierungsschritt für jedes einzelne Datum ausgeführt wird. Das macht den Algorithmus wesentlich schneller, aber lässt ihn ebenfalls stärker schwanken, während er ein lokales Minimum sucht. Dies kann positiv wie negativ sein, da eine solche Schwankung den Algorithmus Ebenen schneller überwinden lässt, aber manchmal auch Minima überspringen lässt. Um eine gute Konvergenz zu gewährleisten, erweitert man den Algorithmus noch um zwei Eigenschaften.

- Die  $x^{(i)}, y^{(i)}$  werden jede Iteration zufällig angeordnet
- Die Lern Geschwindigkeit  $\eta$  wird linear verkleinert.

Der stochastic gradient descent kann durch folgende Eigenschaft noch erweitert werden.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \quad (5)$$

Nun wird nicht mehr für jedes einzelne Datum der Gradient berechnet, sondern für einen Teilmenge der Größe  $n$ . In mancher Literatur, wird dieser Algorithmus noch einmal extra als **Mini-batch gradient descent** benannt. [Rud, Kapitel 2.3]. In den Implementierungen wird hier jedoch meist keine Unterscheidung mehr getroffen. [C<sup>+</sup>15]

### 2.3.2 Adagrad

Der Adagrad Algorithmus erweitert den Stochastic Gradient Descent noch weiter. Bisher wurde die Lern Geschwindigkeit  $\eta$  entweder konstant gelassen oder linear verkleinert. Die Lern Geschwindigkeit hat somit keinen Bezug auf die Trainingsdaten. Hier setzt Adagrad an. Er verändert die Lern Geschwindigkeit im Verhältnis zur Dichte des jeweiligen Parameters.

Deshalb hängt der Algorithmus diesmal auch von jedem einzelnen Elemente von  $\theta$ , bezeichnet als  $\theta_i$  ab.

**Definition 2.6** [Rud, Kapitel 4.3] Der **Adagrad** Algorithmus ist definiert durch den Aktualisierungsschritt

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \quad (6)$$

wobei  $g_{t,i}$  definiert ist durch

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i}) \quad (7)$$

und  $G_t \in \mathbb{R}^{d \times d}$  ist eine diagonal Matrix, wobei die Diagonalelemente  $i,i$ , die Summe der Quadrate der Gradienten zum zugehörigen Parameter  $\theta_i$  bis zum Zeitpunkt  $t$  sind und  $\epsilon > 0$

Dies bringt eine große Erleichterung im Vergleich zu den bisherigen Algorithmen. Durch die automatische und variable Anpassung der Lern Geschwindigkeit gewinnt man eine einfachere Implementierung und bessere Ergebnisse.

Adagrad hat jedoch einen Nachteil, durch die quadrierten summierten Gradienten unter dem Bruch konvergiert die Veränderung der Parameter gegen null. Deshalb kann Adagrad situationsbedingt auch schlechter als die bisherig vorgestellten Algorithmen sein.

### 2.3.3 Adam

[Die15] Adaptive Moment Estimation (Adam) ist ein Weiterer Algorithmus, der sich wie Adagrad adaptiv an den jeweiligen Parameter anpasst. Er versucht aber gleichzeitig das Problem der summierten Gradienten zu umgehen.



**Definition 2.7** Der **ADAM** Algorithmus ist definiert durch den Aktualisierungsschritt

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad (8)$$

wobei

$$\hat{m}_t = \frac{\beta_1 m_{t-1} + (1 - \beta_1) g_t}{1 - \beta_1^t} \quad (9)$$

und

$$\hat{v}_t = \frac{\beta_2 v_{t-1} + (1 - \beta_2) g_t^2}{1 - \beta_2^t} \quad (10)$$

wobei  $\hat{m}_t$  den Mittelwert des Gradienten über die Zeit beschreibt und  $\hat{v}_t$  die Varianz.  $\beta_1$  und  $\beta_2$  sind dabei Zerfall Raten, die bestimmen, wieviel Einfluss die vergangenen Gradienten auf den neuen Wert nehmen.

Adam ist deswegen ein sehr beliebter Optimierungsalgorithmus, da er nicht nur adaptiv für jeden einzelnen Parameter arbeitet, sondern durch die Anpassung mit Mittelwert und Varianz der vergangenen Gradienten statistisches Wissen über die Beschaffenheit der Oberfläche mit einbringt.

## 3 Evaluation

In diesem Kapitel wollen wir uns jetzt mit der Analyse der einzelnen Optimierungsmethoden beschäftigen. Mithilfe des vorherigen Kapitel wollen wir nachweisen, dass ein theoretisch besserer Algorithmus auch wirklich besser arbeitet. Was in diesem Fall besser bedeutet soll ebenfalls geklärt werden.

### 3.1 Test Datensatz

Um ein neuronales Netz zu bewerten, brauchen wir zunächst einige Test Daten, hierbei werden die beiden frei verfügbaren Datensätze, der *Boston House Price* Datensatz und der *Breast Cancer* Datensatz. Beide sind im *scikit-learn* Paket [PVG<sup>+</sup>11] enthalten.

#### 3.1.1 Boston House Price

Der *Boston House Price* Datensatz besteht aus einem Regressionsproblem. Das bedeutet, dass wir eine reelle Zahl mit unserem Netz voraussagen wollen, nämlich den Preis eines Hauses in Boston. Damit das das Netz lernen kann gibt es einige explanatorische Variablen wie unter anderem

- **CRIM:** Die Verbrechensrate pro Kopf.

- **RM**: Die Anzahl der Räume im Haus.
- **TAX**: Steuer die auf das Grundstück gezahlt werden muss.
- **DIS**: gewichteter Abstand zum Zentrum von Boston.

Mit diesen, insgesamt 13 Hilfsvariablen soll das Netz den Haus Preis voraussagen.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03

Abbildung 2: Beispiel des Boston Datensatzes

### 3.1.2 Breast Cancer

Der *Breast Cancer* Datensatz hingegen besteht aus einem binären Klassifikationsproblem. Das bedeutet, dass das Netz nur “ja” oder “nein” kodiert als eins oder null, voraussagen soll. Hierbei bedeutet ein Ja, dass diese Person Brustkrebs hat. Hier gibt es wieder einige explanatorische Variablen, die dem Netz helfen sollen zu lernen. Diese sind Eigenschaften des Nukleus einer entommenen Brustzelle

- **radius**: Radius des Nukleus
- **texture**: Standardabweichung der Graustufenwerte
- **symmetry**: Symmetrie des Nukleus
- **smoothness**: Lokale Abweichung der Radius Länge

Insgesamt besteht der Datensatz aus 30 solchen Variablen.

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.07871
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.05667
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.05999

Abbildung 3: Beispiel des Breast Cancer Datensatzes

## 3.2 Metrik

Bevor wir mit dem Vergleich der verschiedenen Optimierungsmethoden der Gradienten beginnen können, brauchen wir ein Maß für die Güte eines Ergebnisses. Dafür müssen wir eine Metrik definieren. Sinnvoll ist es ein neuronales Netz daran zu messen, wieviel es richtig bewertet hat. Richtig ist im Falle des *Breast Cancer* Datensatzes einfach ob das Netz ja zu ja und nein zu nein gesagt hat. Im Falle des *Boston House Price* Datensatzes ist richtig, wenn der Abstand vom vorhergesagten zum echten Preis möglichst klein ist. Dies gibt uns eine Vielzahl von Metriken die diese Voraussetzungen erfüllen.

Günstiger Weise benötigen wir bereits durch das Training des neuronalen Netzes eine Gütefunktion die angibt ob sich die Parameter in die richtige Richtung bewegen. Diese ist die in Abschnitt 2.1 bereits besprochene Fehlerfunktion des Netzes. Diese sollte unsere erste Form einer Metrik sein um das Neuronale Netz zu bewerten.

Für den *Boston House Price* Datensatz ist diese Fehlerfunktion die *mittlere quadratische Abweichung*.

**Definition 3.1** [FHK<sup>+</sup> 16, S.344] Die **mittlere quadratische Abweichung** für eine Stichprobe  $x_1, \dots, x_n$  mit Schätzwerten  $\hat{x}_1, \dots, \hat{x}_n$  ist definiert durch

$$\frac{1}{n} \sum_{i=1}^n (\hat{x}_i - x_i)^2 \quad (11)$$

Dies ist die typische Fehlerfunktion für Regressionsprobleme. Man mag sich fragen warum diese quadriert wird und nicht der absolute Abstand zum echten Wert genommen wird, wie vorher überlegt. Diese Funktion ist aber diejenige aus der wir den Gradienten berechnen wollen, also muss sie differenzierbar sein, was das Quadrat gewährleistet.

Für den *Breast Cancer* Datensatz ist diese Funktion jedoch unzureichend, da nur null oder eins im Wertebereich der mittleren quadratischen Abweichung vorkommen würden. Da es sich hier um ein binäres Klassifikationsproblem handelt, eignet sich die *binäre Kreuzentropie*.

**Definition 3.2** [RK04] Die **binäre Kreuzentropie** für eine Stichprobe  $x_1, \dots, x_n$  mit Schätzwerten  $\hat{x}_1, \dots, \hat{x}_n$  ist definiert durch

$$\frac{1}{n} \sum_{i=1}^n x_i \log(\hat{x}_i) + (1 - x_i) \log(1 - \hat{x}_i) \quad (12)$$

wobei  $\log$  der natürliche Logarithmus ist.

Diese Fehlerfunktion ist geeignet für Wertebereiche zwischen null und eins. Also genau

passend für ein zwei Klassen Klassifikationsproblem.

Diese beiden Fehlerfunktionen geben uns eine erste Idee für die Güte des neuronalen Netzes und deren Prädiktion und damit auch einen Indikator, welches Netz den besseren Optimierungsalgorithmus besitzt, wenn sonst alle Parameter gleich bleiben. Zusätzlich nehmen wir um mehr Vergleichbarkeit zu schaffen eine weitere Fehlerfunktion hinzu, die nur der Evaluierung der besseren Optimierungsmethode dient und nicht für das Gradienten Abstiegsverfahren genutzt wird. Im Falle des *Boston House Price* Datensatzes benutzen wir noch den *Mittleren absoluten Fehler*.

**Definition 3.3** Der *Mittlere absolute Fehler* für eine Stichprobe  $x_1, \dots, x_n$  mit Schätzwerten  $\hat{x}_1, \dots, \hat{x}_n$  ist definiert durch

$$\frac{1}{n} \sum_{i=1}^n |\hat{x}_i - x_i| \quad (13)$$

Dies ist die intuitivere Definition des Fehlers und ist somit geeignet um die Optimierungsmethoden zu vergleichen.

Für den *Breast Cancer* Datensatz benutzen wir zusätzlich die *Accuracy*, um die Ergebnisse des neuronalen Netzes vergleichbarer zu machen.

**Definition 3.4** Die *Accuracy* einer Stichprobe der Größe  $n$  ist definiert durch

$$Accuracy = \frac{\text{Anzahl korrekter Voraussagen}}{n} \quad (14)$$

Diese ist ebenfalls das intuitivere Verständnis der Güte eines neuronalen Netzes und hilft somit bei der Auswertung der Optimierungsalgorithmen.

### 3.3 Programm

Für einen Aufbau eines solchen Netzes und die Anwendung des Gradienten Abstiegsverfahrens gibt es bereits verschiedenste Frameworks. Diese stehen vor Allem in *Python* zur Verfügung, da diese eine der vorrangigen KI Sprachen ist. Aus diesem Grund wird das folgende Programm für die Auswertung der Optimierungsalgorithmen, auf unseren zwei Test Datensätzen, ebenfalls in *Python* implementiert sein.

#### 3.3.1 Frameworks

Um uns die Arbeit zu erleichtern und um die Optimierungsalgorithmen, sowie das neuronale Netz nicht selbst implementieren zu müssen benutzen wir einige Frameworks.

- **Keras** [C<sup>+</sup>15]: Ein Framework, um neuronale Netze durch ein “Baustein” artiges Prinzip aufzubauen.

- **Scikit-learn** [PVG<sup>+</sup>11]: Ein Framework, welches viele machine learning- sowie Vorverarbeitungsalgorithmen als Funktionsaufruf zur Verfügung stellt.
- **Pandas** [McK10]: Ein Framework, was uns tabellenartige Datenstrukturen zur Verfügung stellt, welche die Weiterverarbeitung vereinfachen.
- **Numpy** [Oli ]: Ein Framework für wissenschaftliche Berechnungen in Python, wie zum Beispiel Matrizenmultiplikation.

### 3.3.2 Aufbau

## 3.4 Auswertung

# 4 Fazit

## Literaturverzeichnis

- [Bur97] BURKHARD LENZE: *Einführung in die Mathematik neuronaler Netze*. Berlin : Logos Verlag, 1997
- [C<sup>+</sup>15] CHOLLET, François u. a.: *Keras*. <https://keras.io>, 2015
- [Die15] DIEDERIK P. KINGMA AND JIMMY LEI BA: *Adam: a Method for Stochastic Optimization*. 2015
- [FHK<sup>+</sup>16] FAHRMEIR, Ludwig ; HEUMANN, Christian ; KÜNSTLER, Rita ; PI-GEOT, Iris ; TUTZ, Gerhard: *Statistik: Der Weg zur Datenanalyse*. 8., überarbeitete und ergänzte Auflage. Berlin and Heidelberg : Springer Spektrum, 2016 (Springer-Lehrbuch). <http://dx.doi.org/10.1007/978-3-662-50372-0>. <http://dx.doi.org/10.1007/978-3-662-50372-0>. – ISBN 9783662503713
- [Kön02] KÖNIGSBERGER: *Analysis 2*. Springer Berlin Heidelberg, 2002. – ISBN 3540435808
- [McK10] MCKINNEY, Wes: Data Structures for Statistical Computing in Python. In: WALT, Stéfan van d. (Hrsg.) ; MILLMAN, Jarrod (Hrsg.): *Proceedings of the 9th Python in Science Conference*, 2010, S. 51 – 56
- [Mic] MICHAEL NIELSEN: *Neural Networks and Deep Learning*. <http://neuralnetworksanddeeplearning.com/index.html>
- [Oli ] OLIPHANT, Travis: *NumPy: A guide to NumPy*. USA: Trelgol Publishing. <http://www.numpy.org/>. Version: 2006–. – [Online; accessed <today>]
- [PVG<sup>+</sup>11] PEDREGOSA, F. ; VAROQUAUX, G. ; GRAMFORT, A. ; MICHEL, V. ; THI-RION, B. ; GRISEL, O. ; BLONDEL, M. ; PRETTENHOFER, P. ; WEISS, R. ; DUBOURG, V. ; VANDERPLAS, J. ; PASSOS, A. ; COURNAPEAU, D. ; BRU-CHER, M. ; PERROT, M. ; DUCHESNAY, E.: Scikit-learn: Machine Learning in Python. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830
- [RK04] RUBINSTEIN, Reuven Y. ; KROESE, Dirk P.: *The cross-entropy method: A unified approach to combinatorial optimization, Monte-Carlo Simulation and machine learning*. New York, NY : Springer, 2004 (Information science and statistics). – ISBN 9780387212401
- [Rud] RUDER, Sebastian: *An overview of gradient descent optimization algorithms*. <http://arxiv.org/pdf/1609.04747v2>

## **Eidesstattliche Erklärung**

### Eidesstattliche Erklärung zur Seminararbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

*Unterschrift :*

*Ort, Datum :*