Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Titel der Arbeit

Bearbeiter: Bachvarov, Vladislav

Gruppe: GruppenID

Projektstudium

Ort, Abgabedatum

Kurzfassung

In dieser Ausarbeitung wird ein Programm, eher eine Klasse, erstellt, das ein ANFIS-Modell erstellt. Das ANFIS-Modell ist ein Neuro-Fuzzy-System, das in seinem Herzen ein TSK-Modell und ein Neuronales Netz kombiniert werden. Mit dem Modell, darf eine spezifische Lernaufgabe mit den Tools des Neuronalen Netzes gelernt werden. Ziel der Ausarbeitung war das Modell zu implementieren und seine Mglichkeiten zu testen. Im Fazit der letzten Kapitel werden alle Erkenntnisse infolge der Analyse geschrieben.

Vorerst wird aber mit einer kurzen Einfhrung in Fuzzy-Logik und Fuzzy-Systeme. In den nsten Kapitel wird auf Neuronale Netze eingegangen. Folglich beschige ich mich mit den unterschiedlichen Arten von Neuro-Fuzzy-Systeme, dabei werden zwei miteinander verglichen - ANFIS und NEFCON. In dem letzen Kapitel wird dann auf die Analyse eingegangen. Die Kapitel ist die umfangreichste und das wichtigste in dieser Ausarbeitung. Da aber nicht alle Testfe betrachtet und analysiert werden knnen, wird ein Anhang hinzugefgt, wo alle zuslichen Testfen mit Ergebnissen zu finden sind.

Inhaltsverzeichnis

1 Einleitung und Problemstellung 1

2 Grundlagen 2
2.1 Unscharfe Daten und Fuzzy-Logik 2
2.2 Fuzzy-Logik 2
2.2.1 Fuzzy-Sets 3
2.3 Operationen auf Fuzzy-Sets 5
2.3.1 Durchscnitt 5
2.3.2 Vereinigung 6
2.4 Fuzzy-Systeme 8
2.4.1 Mamdani-Regler 8
2.5 Takagi-Sugeno-Kang-Modell 10
2.6 Künstliche Neuronale Netze 12
2.7 Biologische Grundlagen 13
2.7.1 Neuron in Künstlichen Neuronalen Netzen 13
2.8 Lernen 14
2.8.1 Unsupervised Learning 14
2.8.2 Supervised Learning 14
2.8.3 Schlüsse 16
2.9 Neuro-Fuzzy-Systeme 16
2.10 Neuro-Fuzzy-Regler 16
2.10.1 Modell für feste Lernaufgaben 17
2.10.2 Modell mit verstärkendem Lernen 19
2.11 Erlernen einer Regelbasis 21
2.11.1 Top-Down- oder Reduktionsmethode zum Erlernen einer Regelbasis 21
2.11.2 Bottom-Up- oder Eliminationsmethode zum Erlernen einer Regelbasis 21
2.12 Optimierung der Regelbasis 22
2.13 Schlussworte 22

3 Neuro-Fuzzy-Modelle: Implementierung des ANFIS-Ansatzes . . 23
3.1 ANFIS-Klasse 23

3.1.1 Konstruktor . . . 23
3.1.2 Eingangsschicht (FirstLayer) . . . 24
3.1.3 Konklusionsparameter . . . 25
3.1.4 Zugehörigkeitsfunktion (Second and Third Layer) . . . 26
3.1.5 Fourth Layer . . . 29
3.1.6 Fifth Layer . . . 29
3.1.7 Sixth Layer . . . 30
3.2 Optimisierungsfunktion . . . 30
3.3 Trainingsfunktion . . . 30

4 Analyse der Ergebnisse . . . 32
4.1 Lernen der Sinusfunktion mit Stochastic Gradient Descent . . . 34
4.1.1 Schlussfolgerung . . . 41
4.2 Lernen der Sinusfunktion mit Mini-Batch Gradient Descent . . . 41
4.3 Lernen der Sinusfunktion mit Batch Gradient Descent . . . 46
4.3.1 Lernen mit 2 Fuzzy-Sets . . . 47
4.3.2 Lernen mit 3 Fuzzy-Sets . . . 48
4.4 Lernen der Parabelfunktion . . . 50
4.4.1 Lernen der Parabelfunktion mit Mini-Batch Gradient Descent 50
4.5 Fazit . . . 53

Literaturverzeichnis . . . 56

1

Einleitung und Problemstellung

Das Thema dieser Ausarbeitung ist Unscharfe Informationen, oder Fuzzy-Sets. Mit Fuzzy-Sets lassen sich schwammige Daten, wie “große” Zahlen oder “mittlere” Temperatur, für Maschinen, insbesondere Computern, beschreiben. Diese Mengen können dann von so genannten Fuzzy-Systeme interpretiert werden. Somit zieht man bestimmte logische Rückschlsse bezüglich einer Eingabe. Als Beispiel könnte man die Farben von Tomaten nehmen. Der Mensch kann mit höher Richtigkeit entscheiden, welche Tomate denn reif ist. Für eine Maschine jedoch ist die Interpretation roher Information(Tomate ist Rot, also reif) nicht möglich. Mit den Fuzzy-Sets und Regeln kann man in dem System, dieses definieren.

Ziel dieses Projektes ist es, unter Anwendung von Neuronalen Netzen, Parameter beliebiger Fuzzy-Modelle zu optimieren. Das würde hein, dass bei der Mathematische Funktion $y = a * x_1 + b * x_2$ die Parametern a und b von dem neuronalen Netz automatisch angepasst werden, sodass sich der Erwartungswert y' bei den Eingaben x_1 und x_2 ergibt.

In dieser Ausarbeitung werden einige Ansätze vorgestellt. Schließlich wird eine Entscheidung getroffen, welcher Ansatz verwenden wird, Neuro-Fuzzy-Systeme aufzubauen.

2

Grundlagen

2.1 Unscharfe Daten und Fuzzy-Logik

In dem normallen Alltag eines Menschen werden ständig Aussagen über Ereignisse getroffen. Diese Aussagen können möglichst exakt sein, oder auch nicht. In manchen Situationen werden solche genannt, die für den Menschen adäquat ein Ereignis beschreiben. Man würde zum Beispiel sagen, dass es gerade sehr stark regnet, oder es sehr heiß ist. Diese Information kann von dem menschlichen Gehirn angemessen vearbeitet werden. Maschinen, wie Komputern, sind jedoch nicht mit dieser Funktionalität ausgestattet. Die sogenannten unscharfe Daten müssen somit anders repräsentiert werden, damit auch Maschinen diese verarbeiten können. Auf dieser Weise können wir den Vorteil der Maschinen gegenüber Menschen ausnutzen - ihre große Kapazität und die Möglichkeit komplexere Strukturen und Systeme darzustellen. In diesem Kapitel wird eine Erweiterung der klassischen Logik vorgestellt, die es den Komputern ermöglicht, unscharfe Daten darzustellen und darauf Operationen durchzuführen. Diese Logik ist als Fuzzy-Logik bekannt.

2.2 Fuzzy-Logik

In der Literatur bezeichnet man die präzise Erfassung von unscharfen Daten als Fuzzy-Logic. Fuzzy-Logic unterscheidet sich von der klassischen Mengenlehre darin, dass Elemente graduell einer Menge gehören und nicht nur bivalente Zugehörigkeit erweisen können.

Zum Verdeutlichen betrachten wir die Menge M der reellen Zahlen, die viel größer als 1 sind.

$$M = \{x \mid x \in \mathfrak{R},\ x >> 1\} \tag{2.1}$$

Wird diese Menge M mit der klassischen Logik modelliert, ergibt sich die Problematik: Der Vergleich “viel größer” ist mathematisch nicht eindeutig definiert. Auf der Grafik 2.1 unten kann man die Modellierung mit der klassischen Logik sehen.

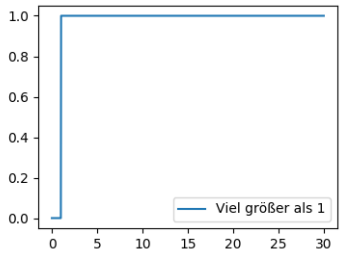

Abb. 2.1: “Viel größer als 1” in der klassischen Logik

Aus der Abbildung 2.1 kann man feststellen, dass alle Werte kleiner als 10 eine Zugehörigkeit von 0 und solche größer oder gleich 10 - eine Zugehörigkeit von 1 besitzen. Diese Repräsentation entspricht jedoch die Realität nicht. Es ist eindeutig, dass 9.9 als Wert schon größer als 1 ist, aber der Abbildung 2.1 nach wird das nicht klar. Würde man diese Menge in der Fuzzy-Logik darstellen, ergibt sich folgende Abbildung.

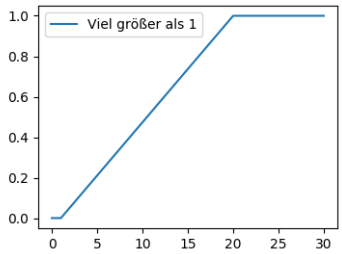

Abb. 2.2: “Viel größer als 1” in der Fuzzy-Logik

Hier beschreibt die Ausbildung 2.2 genauer, wie die Zahlen im Vergleich zu 1 stehen. Die Zugehörigkeit nimmt Werte zwischen 0 und 1. Zum Beispiel der Wert 10 hat die Zugehörigkeit von ca. 0.5 und für Werte größer als 20 liefert die Funktion $\mu(x)$ einen Wert von 1 (volle Zugehörigkeit).

2.2.1 Fuzzy-Sets

Fuzzy-Sets, oder Fuzzy-Mengen, beschreiben in der Fuzzy-Logik Eigenschaften von Elementen. Die Idee ist, dass Elemente zu einem rationallen Wert einer Menge gehören, beziehungsweise eine Eigenschaften besitzen. In der Literatur wird folgende Definition für Fuzzy-Mengen gegeben [RCC+15]:

Definition 2.1. *Eine Fuzzy-Menge oder Fuzzy-Teilmenge μ der Grundmenge X ist eine Abbildung $\mu : X \to [0, 1]$, die jedem Element $x \in X$ seinen Zugehorigkeitsgrad $\mu(x)$ zu μ zuordnet. Die Menge aller Fuzzy-Mengen von X bezeichnen wir mit $F(X)$. [RCC+15]*

Mengen aus der klassischen Logik können als spezielle Fuzzy-Mengen aufgefasst werden. Also sind Fuzzy-Sets verallgemeinerte charakteristische Funktionen [RCC+15].

In der Praxis werden mehrere Arten von Fuzzy-Mengen entwickelt. Die bekanntesten davon sind Dreiecksfunktion, Trapezfunktion und Gaußfunktion. Ihre Namensgebung ergibt sich aus der Funktion, die sie berechnet. Drei Beispielfunktionen sind unten gegeben.

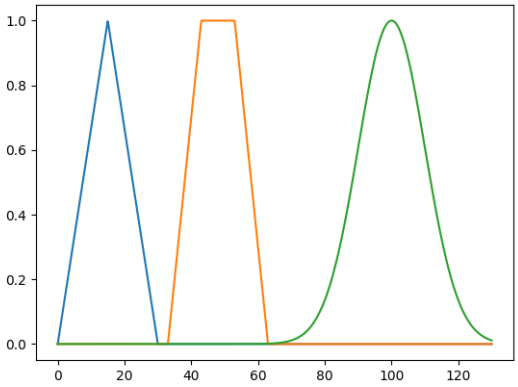

Abb. 2.3: Dreiecks-, Trapez- und Gaußfunktion von Fuzzy-Mengen

Die Form einer Fuzzy-Menge ist durch ihre Funktion bestimmt. Die Berechnung der Dreiecksfunktion wird in 2.2 .

$$\mu_{tri}(x) = \begin{cases} \frac{x-a}{b-a} & \text{falls } a \leq x \leq b \\ \frac{c-x}{c-b} & \text{falls } b \leq x \leq c \\ 0 & \text{sonst} \end{cases} \tag{2.2}$$

Der Index der Funktion verdeutlicht, dass es sich um die Dreiecksfunktion handelt. Die Funktion hat drei Parametern a, b und c, für die $a < b < c$ gelten muss. Die Dreiecksfunktion ist ein besonderer Fall der Trapezfunktion, wo die Punkte, bzw. Parametern, der oberen Grudseite gleich sind. Deswegen ist die Gleichung 2.3 für die Trapezfunktion sehr ähnlich [RCC+15].

$$\mu_{trap}(x) = \begin{cases} \frac{x-a}{b-a} & \text{falls } a \leq x \leq b \\ 1 & \text{falls } b \leq x \leq c \\ \frac{d-x}{d-c} & \text{falls } c \leq x \leq d \\ 0 & \text{sonst} \end{cases} \tag{2.3}$$

Die Glockefunktion ist, wie ihrer Name andeutet, eine Funktion, die die Form einer Glocke hat. Die Darstellung dieser Funktion ähnelt sich sehr der Gaußsche Funktion. Die Berechnung wird in 2.4 gegeben.

$$\mu_{bell}(x) = \frac{1}{1 + [(\frac{x-c}{a})^2]^b} \tag{2.4}$$

Die letzte Funktion, die vorgestellt werden soll, ist die gaußsche Funktion. Die Formel wird öfters in der Statistik für Darstellung von Nominalverteilungen, aber auch in der Fuzzy-Logik, benutzt. Die Gaußfunktion ist in 2.5 gegeben [RCC+15]:

$$\mu_{gauss}(x) = exp(\frac{-(x-m)^2}{s^2}) \tag{2.5}$$

Die Parametern *mundq* sind entsprechend der Mittelwert (Mittelpunkt) und die Abweichung von der Mitte, oder als σ (Sigma) in der Statistik bekannt.

2.3 Operationen auf Fuzzy-Sets

In dem vorherigen Unterkapitel 2.2 wurde die Fuzzy-Logik eingeführt, so wie auf die Repräsentation von Fuzzy-Mengen eingegangen. Um nun unscharfe Informationen verarbeiten zu können, wie Schlüsse daraus zu ziehen oder mehrere Fuzzy-Mengen zu kombinieren, brauchen wir eine Reihe von Operatoren. Da es um Mengen geht, eignen sich die Durchschnitt-, Vereinigung- und Komplementbildung aus der klassischen Logik gut. Im folgenden Kapitel werden die einzelnen Operationen beschrieben.

2.3.1 Durchscnitt

In der klassischen Logik ist der Durchschnitt durch einen Logischen-UND eingesetzt. die Menge aller Elementen, die zu einer Menge M_1 und einer Menge M_2 gehören, ist als Schnittmenge definiert. Gegeben seien die Mengen:

$$M_1 = \{x \mid x \in \mathfrak{R},\ 1 \leq x \leq 3\} \tag{2.6}$$

$$M_2 = \{x \mid x \in \mathfrak{R},\ 2 \leq x \leq 4\} \tag{2.7}$$

Der Durchschnitt der beiden Mengen ergibt sich aus:

$$M_1 \cap M_2 = \{x \mid x \in \mathfrak{R},\ 2 \leq x \leq 3\} \tag{2.8}$$

Der UND-Operator lässt sich analog auf die Fuzzy-Mengen anwenden. Es wird die Fläche bestimmt, die für beide Mengen erfüllt ist. Aus methematischer Sicht sprechen wir von dem Minimum-Operator(MIN).

Definition 2.2. *Seien μ_1 und μ_2 zwei Fuzzy-Mengen auf der Grundmenge G. Dann heißt:*

$$\mu_1 \cap \mu_2 : G \to [0,1] \text{ mit } (\mu_1 \cap \mu_2)(x) = MIN(\mu_1(x)\ ,\ \mu_2(x))$$

*der **Durchschnitt** der Fuzzy-Mengen μ_1 und μ_2.*

Zur Veranschaulichung wird unten die Grafik angegeben. Da sind zwei Fuzzy-Sets dargestellt. Die zwei Ausdrücke, die betrachtet werden, sind *mittlere* und *hohe* Temperatur. Der rote gestrichene Bereich stellt die Ergebnismenge dar.

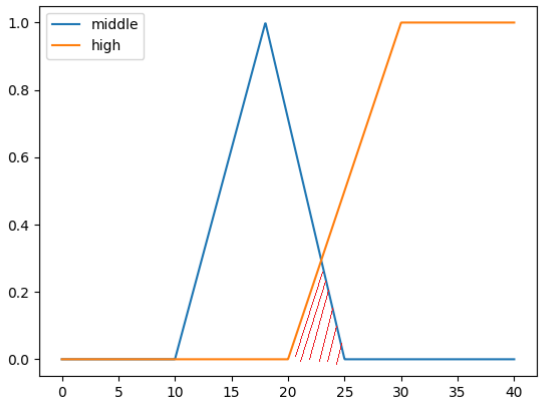

Abb. 2.4: Durchschnitt von zwei Fuzzy-Mengen (roter gestrichener Bereich)

2.3.2 Vereinigung

Nehmen wir ganz einfach die Definition der Vereinigung aus der klassischen Mengenlehre:

$$x \in M_1 \cup M_2 \Leftrightarrow x \in M_1 \vee x \in M_2$$

Ziemlich eindeutig und klar. Die Vereinigungsmenge enthält alle diese Elemente aus dem Grundbereich, die entweder in der Menge M_1 oder M_2 enthalten sind. Im nächsten Schritt wird dieser Operator an die Fuzzy-Mengen angepasst.

In der Mathematik ist dieser Operator als ODER-Verknüpfung angegeben. Die Anwendung der ODER-Operator auf Fuzzy-Mengen wird wie folgt dann definiert:

Definition 2.3. *Seien μ_1 und μ_2 zwei Fuzzy-Mengen auf der Grundmenge G. Dann heißt:*

$$\mu_1 \cup \mu_2 : G \rightarrow [0,1] \text{ mit } (\mu_1 \cap \mu_2)(x) = MIN(\mu_1(x)), \mu_2(x))$$

*die **Vereinigung** der Fuzzy-Mengen μ_1 und μ_2.*

Betrachten wir den selben Beispiel aus vorherigen Unterkapitel. Seien also wieder die Fuzzy-Mengen für *"mittlere"* und *"hohe"* Temperatur. Wenn wir den ODER-Operator auf die beiden Mengen anwenden, ergibt sich eine Ergebnismenge, die sich auf beiden Mengenflächen aufstellt. Dies wurde graphisch zunächst aufgezeichnet.

Komplement

Der dritte wichtige Operator ist das Komplement einer Menge. In der klassischen Mengenlehre ist dieser Operation ziemlich einfach anzuwenden. Der Operator beschreibt die Negation einer Aussage - zum Beispiel die Wahrscheinlichkeit eine 6 zu würfeln wäre $\frac{1}{6}$, die Gegenwahrscheinlichkeit, oder die Wahrscheinlichkeit etwas anderes als 6 zu würfeln wäre: (1 - $\frac{1}{6}$) = $\frac{5}{6}$. Das Komplement ist in der Fuzzy-Logik dann wie folgt definiert:

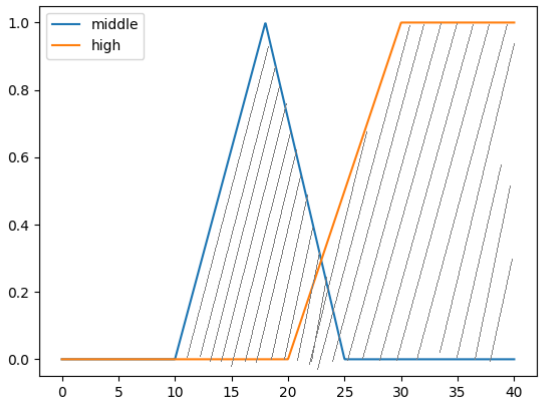

Abb. 2.5: Union von Fuzzy-Mengen (gestrichener mit Blau Bereich)

Definition 2.4. *Sei μ eine Fuzzy-Menge auf der Grundmenge G. Dann heißt:*

$$\mu^c : G \rightarrow [0,1] \text{ mit } (\mu^c)(x) = 1 - \mu(x))$$

die ***Vereinigung*** *der Fuzzy-Mengen μ_1 und μ_2.*

Zur Veranschaulichung die Menge *hohe* Temperatur negiert. Die Negierung ist in der Graphik unten zu sehen.

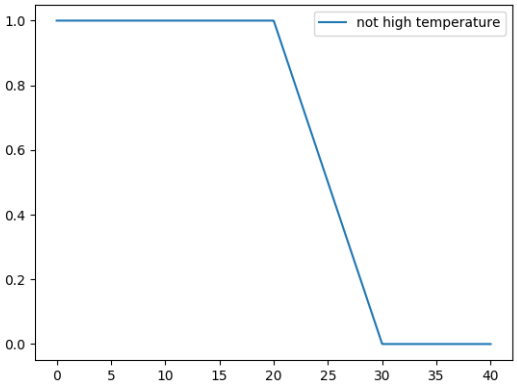

Abb. 2.6: Komplement der Menge hohe Temperatur

Zusammenfassung

In diesem Unterkapitel 2.3 habe ich die drei wichtigsten Operatoren auf Fuzzy-Mengen vorgestellt. Neben den Grundverknüpfungen gibt es eine weitere Sammlung von Verknüpfungsoperatoren.

1. Algebraisches Produkt: $(\mu_1\mu_2) = \mu_1(x) \cdot \mu_2(x)$
2. direkte Summe: $(\mu_1 \oplus \mu_2) = \mu_1(x) + \mu_2(\text{x})$ - $\mu_1(x) \cdot \mu_2(x)$
3. abgeschnittene Differenz: $(\mu_1 \dot{-} \mu_2) = \text{MAX}(0, \mu_1(\text{x}) + \mu_2(\text{x})$ - 1)
4. abgeschnittene Summe: $(\mu_1 \hat{+} \mu_2) = \text{MIN}(1, \mu_1(\text{x}) + \mu_2(\text{x}))$
5. $(\mu_1 \dot{-} \mu_2) = \text{MIN}(\mu_1(\text{x}), 1$ - $\mu_2(\text{x}))$
6. ...

Dies ist eine Vorbereitung auf die folgenden Kapiteln. Wir analysieren mehrere Fuzzy-Mengen und ziehen daraus bestimmte Schlüsse. Ein Beispiel dafür wäre: Wenn eine Tomate rot ist, dann ist die reif. Folgende Logische Aussagen können sehr einfach mit Fuzzy-Logic modelliert werden. In der Literatur taucht die Name Fuzzy-Regeln. Diese können dann zusammengestellt werden, um Fuzzy-Regel-Systeme aufzubauen. Das nächste Unterkapitel 2.4 stellt Fuzzy-Systeme vor.

2.4 Fuzzy-Systeme

Ein Fuzzy-System, auch Fuzzy-Regler oder Fuzzy-Inferenz-System, ist ein bekannter Framework basierend auf Fuzzy-Mengen-Theorie, Fuzzy Wenn-Dann-Regeln und Fuzzy Reasoning. Konzeptionell besteht ein Fuzzy-System aus drei Grundteile - die Regelbasis, welche die Wenn-Dann-Regeln beinhaltet; die Datenbank(oder das Wörterbuch), die alle Zugehörigkeitsfunktionen definiert; und der Entscheidungmechanismus, wer die Inferenz durchführt und eine angemessene Schlussfolgerung erkundet.

Folglich werden zwei Arten von Fuzzy-Systeme erläutert - Mamdani- und Takagi-Sugeno-Kang-Systeme.

2.4.1 Mamdani-Regler

Der Mamdani-Regler wurde im Jahr 1975 von Mamdani auf der Basis einer Veröffentlichung von Zadeh aus der Anfang der siebziger Jahren entwickelt.

Der Mamdani-Regler ist eine endliche Menge von Wenn-Dann-Regeln R der Form:

$$R : \text{If } x_1 \text{ is } A_1 \text{ and ... and } x_n \text{ is } A_n \text{ then } y' \text{ is } B \tag{2.9}$$

In der Regel sind x_1, ..., x_n die Eingangsgrößen und y' die Ausgabe. A_i und B sind linguistischen Werte. Jede Regel besteht aus zwei Teilen - Prämissen und Konklusionen. Jede Vorbedingung untersucht spezifische Eigenschaften. Es wird geprüft, ob der Eingangwert diese Eigenschaften erfüllt. Anhand diese Merkmale schließt sich eine Konklusion.

Sei W die Menge aller linguistischen Konklusionen B_i, so dass $B_1, \ldots, B_m \in W$ gilt. Der Regler kann als eine n-Stellige Funktion mit $f : \mathbb{G}^{\times} \mapsto W$ dargestellt werden. Die Funktion ordnet eine Eingabe zu einem linguistischen Term B_i.

$$f(x_1, ..., x_n) \approx \begin{cases} B_1 & \text{falls } x_1 \text{ is } A_1^{(1)} \text{ und ..., und } x_n \text{ is } A_n^{(1)} \\ \vdots & \\ B_m & \text{falls } x_1 \text{ is } A_1^{(m)} \text{ und ... und } x_n \text{ is } A_n^{(m)} \end{cases} \tag{2.10}$$

In der Funktion gibt es genau so viele Ausgaben, wie es Regeln gibt - m.

Mit einem Mamdani-Regler können viele Probleme aus der reellen Welt definiert werden. Als kleines Beispiel eignet sich die Farben der Tomaten. Wir betrachten die Farbe als Eingabe. Die Ausgabe würde uns sagen, wie reif eine Tomate ist. Wir teilen unsere Eingangsgrößen in drei Farben: rot, gelb und grün. Daraus lassen sich entsprechend 3 Fuzzy-Mengen definieren. Als Ausgabe wird die Reife einer Tomate bestimmt. Das Ergebnis kann in drei Mengen anfallen: reif, halbreif oder unreif. Auf dieser Weise haben wir ein Mamdani-Fuzzy-Model, das aus 3 Fuzzy-Sets und aus 3 Regeln besteht. Die Regelbasis sieht wie folgt aus:

- R_1: if x is rot, then y is reif.
- R_2: if x is gelb, then y is halbreif.
- R_3: if x is grün, then y is unreif.

Eine Tabelle für die Fuzzy-Relationen ist unten gegeben

x $\diagdown$ y	unreif	halbreif	reif
grün	1	0	0
gelb	0	1	0
rot	0	0	1

Tabelle 2.1: Beispiel für Tomaten

Wie man sieht die Tabelle ist ziemlich einfach zu lesen, wenn die Tomate grün ist, wird immer angenommen dass sie unreif ist. Man könnte entsprechend die Fuzzy-Mengen aus der Konklusion verfeinern. Das würde bedeuten, dass die Mengen sich überlappen. Unsere Tabelle könnte dann wie Tabelle 2.2 aussehen:

x $\diagdown$ y	unreif	halbreif	reif
grün	1	0.5	0
gelb	0.3	1	0.3
rot	0	0.5	1

Tabelle 2.2: Beispiel für Tomaten

Abhängig davon wie man die Ausgangsmengen definiert, wie weit die Mengen überlappen, ergibt sich eine unterschiedliche Interpretation der Werte. Je mehr Fuzzy-Sets für eine Größe(Maß) definiert sind, desto besser könnte ihr Zustand repräsentiert werden. Mit der Anzahl der Fuzzy-Sets steigt die Komplexität eines Systems proportional.

Den oberen kleinen Beispiel zeigt wie einfach Mamdani-Modelle zu modellieren sind. Mamdani-Reglern finden in der Praxis öfters Einsetzung.

2.5 Takagi-Sugeno-Kang-Modell

Das Takagi-Sugeno-Kang-Modell ist dem Mamdani-Modell sehr ähnlich. Der Unterschied erweist sich in der Konklusion. TSK-Regler verwenden Regeln der Form:

$$R: \text{ If } x_1 \text{ is } \mu_R^{(1)} \text{ and } \ldots \text{and } x_n \text{ is } \mu_R^{(n)} \text{ then } y = f_R(x_1, \ldots, x_n). \tag{2.11}$$

In den Prämissen der beiden Fuzzy-Systeme liegt kein Unterschied. Die Besonderheit des TSK-Modells liegt in der Konklusion. Da steht eine lineare Funktion, anstatt eine Fuzzy-Menge. In der Konklusion kann jede beliebige Funktion f mit belibiger Eingabe berechnet werden. Die Form der linearen Funktion ist in 2.12 gegeben:

$$f(x_1, \ldots, x_n) = p_0 + p_1 \cdot x_1 + \ldots + p_n \cdot x_n, \tag{2.12}$$

$p_0, \ldots, p_n$ sind die Parametern der Konklusionsfunktion. Für geeignet ausgewählte Parameterwerte beschreibt das TSK-Modell eine belibige mathematische Funktion, dies kann z.B. die Quadratfunktion sein. Für den Fall, dass die Parametern $p_1, \ldots p_n$ den Wert 0 haben, erhält man einen Mamdani-Modell mit scharfer Ausgabe. Solche Modelle heißen in der Literatur auch **zero-order Sugeno-Fuzzy-Modelle** [AM01, Han98, JCE97]. Als Beispiel kann folgende Regelbasis betrachtet werden:

$R_1:$ If x is "sehr klein" then $y = 0$
$R_2:$ If x is "klein" then $y = 1$
$R_3:$ If x is "groß" then $y = 2$
$R_4:$ If x is "sehr groß" then $y = 3$.

Die Fuzzy-Sets sind "sehr klein", "klein", "groß" und "sehr groß" und die Ergebnisswerte entsprechend *0, 1, 2 und 3*.

Fuzzifizierung und Defuzzifizierung

Der Vorgang eines Modells zerlegt sich in zwei Teilen - Fuzzifizierung und Defuzzifizierung. Der erste Begriff beschreibt den Prozess für die Evaluierung der Eingangswerte in Fuzzyinferenzsysteme. Es gibt mehrere Evaluierungsmethoden, beziehungsweise Fuzzifizierungsmechanismen, wie zum Beispiel die Gaußsche Funktion (2.2.1). Der zweite Begriff bezeichnet die Vorgehensweise bei der Berechnung der Ausgabe, oder Inferenz, eines Fuzzy-Systems. Die Reihenfolge der beiden Prozesse ist somit bestimmt. Da im vorrigen Kapitel schon Beispiele für Fuzzifizierungsverfahren vorgestellt werden, wird demnächst in den unterschiedlichen Arten von Defuzzifizierung eingestiegen.

Bei der Regelaktivierung unterscheidet man zwei Fällen. Wenn die Eingabe eine volle Zugehörigkeit zu einer Fuzzy-Menge ergibt, dann liefert das Modell ganz normal den Wert der entsprechenden Konklusionsfunktion. Bei partielle Aktivierung mehreren Regeln ergibt sich das Inferenzergebnis aus einem spezifischen Verfahren. Dafür werden bestimmte Inferenzmethoden angewendet, zum Beispiel das Center of Gravity, oder Center of Area. Einige dieser Methoden werden inden folgenden Unterkapiteln vorgestellt. In allen Verfahren handelt es sich, um Mamdani-Reglern, wo die Konklusion einer Regel aus einer Fuzzy-Mengen besteht. Die letzte vorgestellte Methode ist spezifisch für Takagi-Sugeno-Kang-Modelle anzuwenden.

Height Method

Bei dieser Methode, auch als *maximum membership principle* bekannt, wird der Regelaktievierung mit den größten Wert ausgewählt. Die Vorgehensweise könnte in Systeme verwendet werden, wo der größte Zutreffer nur wichtig ist. Als Beispiel könnte ein System mit entsprechenden Gegenmaßnahmen genannt werden, wo der größte Risiko Vorgang haben soll [AM01].

Center of Gravity

Center of Gravity, auch als Center of Area bekannt, ist die prominenteste Methode von allen, wo sich das Ergebniss als schärfer Wert ergibt. Das Endergebniss berechnet sich aus folgender Formel:

$$y^* = \frac{\int y \cdot \mu_R(y) dy}{\int \mu_R(y) dy} \tag{2.13}$$

Der Wert y hier ist das Ergebnis der aktivierte Regel R und y^* ist das Endergebniss. [AM01]

Weighted Average Method

Die gewichtete Durchschnittsmethode, oder Weighted Average Method, ist nur für Ausgangszugehörigkeitsfunktionen, die aus mehreren symmetrischen Zugehörigkeitsfunktionen μ_i besteht. Die Formel lautet:

$$y^* = \frac{\sum_i \bar{y} \cdot \mu_i(\bar{y})}{\sum \mu_i(\bar{y})} \tag{2.14}$$

Das Modus jeder Zugehörigkeitsfunktion μ_i wird durch den Wert von $\bar{y}$ beschrieben. [AM01]

Takagi-Sugeno-Kang-Defuzzifizierungsmethode

Die Defuzzifizierungsmethode von TSK-Modelle berechnet eine Interpolation zwischen den Ausgangswerten. Das Endergebnis ergibt sich aus dem Ausgabewert

jeder Regel und dessen Multilizierung mit dem Gewicht der entsprechenden Regel und geteilt durch die Summe der Wahrheitsrate jeder Regel. Die Formel ist gegeben:

$$y = \frac{\sum_i \mu_i \cdot f_i(x_1, ..., x_n)}{\sum \mu_i}, \tag{2.15}$$

wo μ_i die Wahrheitsrate, bzw. Aktievierungswert, einer Regel i ist und f_i liefert den Ergebniswert für die Regel bei Eingabe $x_1, ..., x_n$.

Zum Schluss wird eine Abbildung gegeben, die den gesammten Prozess, Fuzzifizierung und Defuzzifizierung, darstellt. Die Abbildung 2.7 beschreibt, wie das Endergebnis berechnet wird.

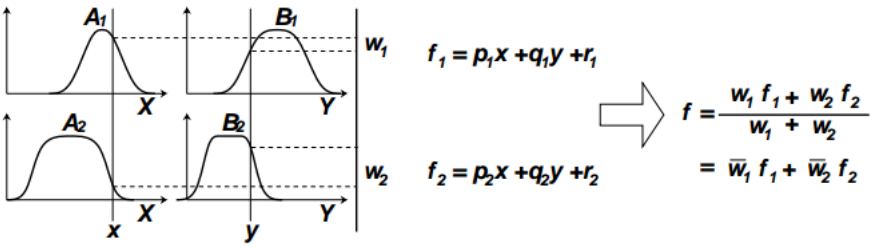

Abb. 2.7: Inferenzschritt eines TSK-Modells [Jan93]

In der Abbildung 2.7 wird ein Model mit zwei Eingangsgrößen und zwei Fuzzy-Mengen pro Eingabe dargestellt. Die Regelbasis besteht aus zwei Regeln. Die Fuzzy-Mengen werden durch die Glockenfunktion berechnet. Die Variablen w_1 und w_2 liefern die Regelaktievierungen für die Regeln mit entsprechenden Indexen. Das Endergebnis ergibt sich aus der oben definierten Funktion 2.15.

2.6 Künstliche Neuronale Netze

Künstliche Neuronale Netze (engl. artificial neural networks) sind Systeme, die auf dem Gehirn von Tieren und Menschen basieren und besteht aus endlich viele Einheiten, Neuronen. Der Datenaustausch geschieht über gerichtete Verbindungen zwischen den Neuronen. Forscher beschäftigen sich mit Neuronalen Netzen aus unterschiedlichen Gründen. In der Biologie simulieren Forscher diese, um sich den Prozess des Lernens und weitere Mechaniscmen zu untersuchen. In der Informatik wird aber versucht, die Lernfähigkeit des Gehirns nachzubilden und auszunutzen.

In den folgenden Kapiteln wird zuerst auf den Neuronen sowohl in der Biologie als auch in der Künstliche Intelligen eingegangen. Weiterhin werden die Eigenschaften von Neuronalen Netzen erklärt. Zum Schluss stelle ich den Gradientenvarfahren vor, anhand dessen diese Netze lernen.

2.7 Biologische Grundlagen

Das tierische Gehirn leitet jede einzelne Aktivität in dem Körper eines Tieres, inklusive die Verarbeitung von Daten. Laut [uP] besteht das Gehirn aus 86 Milliarden Neuronen, die in Cliquen verbunden sind. Neuronen in einer Clique komunizieren untereinander, indem sie Signale über ihre Axone weiterleiten und Signale über ihren Dendrit empfangen. Eine Verbindung zwischen Neuronen existiert, wenn die Axonterminale eines Neurons den Dendrit eines anderen berührt. Zur Veranschaulichung ist den Aufbau eines Neurons in Abbildung 2.8 gegeben. Jeder Signal wird in dem Soma verarbeitet und durch die Axone weitergeschickt.

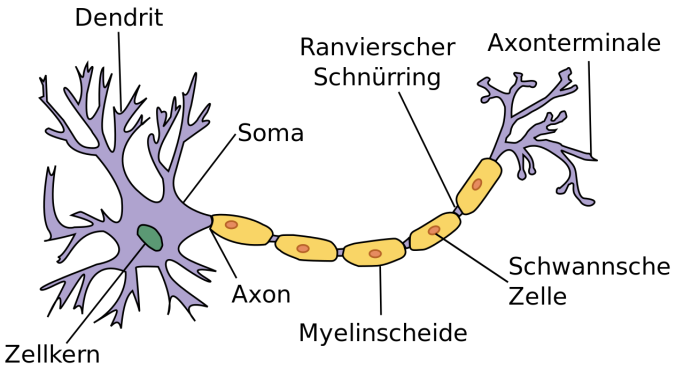

Abb. 2.8: Neuron [Wik]

Auf dem selben Prinzip funktionieren auch Neuronen in Künstlichen Neuronalen Netzen (KNN). Im folgenden Kapitel wird mehr darauf eingegangen. [Wik] [uP]

2.7.1 Neuron in Künstlichen Neuronalen Netzen

Neuronale Netze sind streng genommen gerichtete Graphen, wo jeder Knoten ein Neuron ist und jede Kante eine Verbindung zwischen Neuronen beschreibt. Die Neuronen können in drei Gruppen, oder auch als Schichten (*engl.* Layers) bekannt, unterteilt werden - Eingabe-, Ausgabe- und versteckte Neuronen. Die Ein- und Ausgabeknoten sind die Einheiten, die mit der Umgebung verbunden sind, dabei ist klar welche Knoten was sind. Die übrigen Elemente werden in dem Netz eingebaut und kommunizieren nur mit anderen Neuronen. Daraus ergibt sich auch ihren Namen “versteckt”.

Jede Verbindung zwischen Neuronen in KNN erhält ein Wert. Der Verbindungswert wird meistens als Gewicht bekannt. Bei einem Gewicht von 0 existiert keine Verbindung zwischen Neuronen. In einem KNN werden nur die Gewichte gelernt, während die Neuronen nur eine mathematische Funktion beschreiben. Eine übliche Repräsentation der Gewichte ist die Matrix 2.16.

$$\begin{array}{c} \\ u_1 \\ u_2 \\ \vdots \\ u_r \end{array} \begin{array}{c} \begin{array}{cccc} u_1 & u_2 & \dots & u_r \end{array} \\ \begin{pmatrix} w_{u_1u_1} & w_{u_1u_2} & \dots & w_{u_1u_r} \\ w_{u_2u_1} & w_{u_2u_1} & & w_{u_2u_1} \\ \vdots & & & \vdots \\ w_{u_ru_1} & w_{u_ru_2} & \dots & w_{u_ru_r} \end{pmatrix} \end{array} \tag{2.16}$$

Die Beziehungen zwischen Neuronen aus zwei Schichten wird außerdem ganz gut in der Matrix dargestellt. Sie ist von oben nach rechts zu lesen, also die Spalten sind es dem Neuronen zugeordnet, aus denen die Verbindungen ausgehen. Das heißt der Knoten u_1 hat eine Verbindung zu sich selbst und dem Neuron u_2 und die Gewichtwerte sind entsprechend $w_{u_1u_1}$ und $w_{u_1u_2}$. Die Darstellung als Matrix erlaubt es alle mathematischen Operationen durchzuführen, indem der Ausgabe eines Neurons, oder die Eingabe aus der Umgebung, rechts an der Matrix dranmultipliziert. [RCC+15] [Han98] [AM01] [OLBI98]

2.8 Lernen

Die erstaunlichste Eigenschaft Neuronaler Netze ist ihre Möglichkeit, eine Aufgabe, bzw. Fähigkeit, zu erlernen. Gewichtswerte und/oder weitere Parameter unterliegt Änderungen nach jeder Iterationsschritt während des Lernprozesses. In Bezug auf dem Iterationsschritt wird es zwischen "Offline-" und "Onlinelernen" unterschieden. Außerdem unterscheidet man zwei weiteren Gruppen in der Lernmethode eines Neuronalen Netzes - überwachtes (*engl.* supervised) und unüberwachtes (*engl.* unsupervised) Lernen. Demnächst werden die Begriffe einzeln untersucht. Danach werden nur beaufsichtigte Algorithmen vorgestellt, da die relevant für mein Projekt sind. Zum Schluss werden die Offline- und Online-Lernverfahren vorgestellt, die insgesammt drei sind. [AM01] [RCC+15] [JCE97]

2.8.1 Unsupervised Learning

Unsupervised Learning (*deutsch* unbeaufsichtigtes Lernen) besteht immer noch eine Ein-/Ausgabe beziehung, jedoch wird kein Fehlerfunktion eingesetzt. In diesem Fall muss das Netz Mustern aus den Ein-/Ausgabepaare erkennen und zusammengruppieren.

2.8.2 Supervised Learning

Unter Supervised Learning versteht man den Verfahren, bei dem das Netz anahand von Ein-/Ausgabepaare trainiert wird. Das Lernenprozess beinhaltet die allmäliche Anpassung der Gewichtparametern bei jedem Iterationsschritt, so dass bei gegebener Eingabe x der Fehler, der aus der Ausgabe und dem Erwartungswert berechnet wird, minimiert wird. Die Fehlerfunktion wird in vielen Fällen auch als "Kritiker" und das Modell als "Kritisierender" bekannt. Ein Speziallfall des überwachtes Lernen ist "reinforcement Learning". In dem Lernverfahren wird jede Berechnung "richtig" oder "falsch" bewertet, ohne dass ein Fehler unbedingt berechnet wird. [AM01] [RCC+15] [JCE97]

Least Mean Square Method

Die Kleinste Quadrate Methode (*engl.* Least Mean Square Method) wurde von den B. Widrow und M.Hoff für ein Projekt namens ADALINE oder ADAptive LINear Element entwickelt. Die Methode versucht die Fehlerrate mit dem Gradientenverfahren zu verringern. Die Fehlerrate wird mit dem Mean Squared Error (MSE) geteilt durch die Anzahl der Elementen im Trainingsdaten berechnet. Die Formel ist gegeben:

$$E(w) = \frac{1}{2}\sum_{i=1}^{m}(d_i - y_i)^2, \tag{2.17}$$

wobei w ist der Gewichtsvektor, und d_i und y_i entsprechend - Erwartungs- und Ausgabewert. Die Ableitung der Funktion ergibt den nächsten Iterationsschritt:

$$w^{t+1} = w^t + \mu(d_k^t - y_k^t)x_k^t. \tag{2.18}$$

Die Konstante μ kontrolliert die Konvergenz und Stabilität, **w** und textbfx sind Gewichtsvektor und Eingabe. Der Schritt wird durch den Exponent t bestimmt - $t + 1$ ist der nächste Schritt. Solange die Konstante entsprechend gewählt wird, konvergiert der Verfahren und der gesuchte Gewichtvektor kann gelernt werden. [AM01]

Backpropagation Algorithm

Der Backpropagation Algorithmus, oder auch als Gradient Descent bekannt, wurde laut [AM01] am Ende der 70er Jahren von Paul J. Warbos entwickelt. Dieses Verfahren hat die Interesse an Neuronalen Netzen wiederbelebt.

Der Algorithmus ist der bekannteste Lernalgorithmus für beaufsichtigtes Lernen. In dem Verfahren stecken zwei wichtige Konzepte. Zum einen liegt der Feedforwardphase vor, die dann folgendes Verhalten beschreibt: die Eingabe-/Erwartungswerte dem Neuronalen Netz gegeben werden und anschließend eine Ausgabe berechnet wird. Danach wird der Fehler berechnet, für den eine bestimmte Fehlerfunktion verwendet wird, z.B. der Mean Squared Error (siehe Gleichung 2.17). In der zweiten Phase, auch Rückwertsphase genannt, werden die Knoten rückwerts anfangend von der Ausgabe- über die versteckten Schichten bis zum Eingabeschicht angepasst. Das Ziel der Rückwertsschritt ist es den Fehlre zu minimieren. Hinter dem Verfahren stecken einfache Mathematische Operationen, wurde aber in dieser Ausarbeitung nicht betrachtet. In dem Buch [AM01] wird jedoch eine sehr ausführliche Beschreibung des Algorithmus in Schritten angegeben, welche auch hier figuriert.

1. Die Gewichte werden zufällig initialisiert.
2. Eine Ein-/Ausgabe paar wird dem Netz vorgestellt.
3. Liefer die Ausgabe des Netzes.

4. Berechne die Fehlerrate (z.B. durch die Mean Squared Error 2.17).
5. Beginnend mit der letzten Schicht berechne den Wert der Ableitung δ_j rückwerts.
6. Passe die Gewichte nach in dem Netz.
7. Wiederhole ab 2. Schritt für angegebene Anzahl an Iterationen, oder bis die Fehlerrate kleiner als ein Betrag wird.

2.8.3 Schlüsse

Es werden zwei Bibliotheken für Neuronale Netze berücksichtigt - Tensorflow und SciKit. Biobliotheken sind in der Regel sehr ähnlich. In diesem Fall bieten die betrachteten APIs die nötigten Funktionalitäten, um das Projekt umzusetzen. Jedoch das Ursprüngliche Gedacht war, dass ich die erstellten Neuronalen Modelle in C++ exportiere. Das würde erfordern, dass eventuell das ANFIS-Model mit Hilfe von andere Bibliotheken gegenüber C++ kompilieren müsste. Der Grund, warum ich dann zu Tensorflow geneigt habe, ist, weil Tensorflow geschriebene Modelle, einfach in Dateien zu exportieren sind. Was wiederum mich erlaubt das Model in C++ ohne weiteres zu importieren.

2.9 Neuro-Fuzzy-Systeme

Im vorletzten Kapitel wurden zwei Arten von Fuzzy-Systeme, oder Reglern, vorgestellt. Solche Systeme werden in der Praxis öfters mit Konzepte aus der Künstliche Intelligenz kombiniert. Zum einen bietet sich Kombinationen mit Neural Networks 2.6, Evolutionäralgorithmen und vielen weiteren. Im folgenden Kapitel wird besonders die Kombination mit Neuronalen Netzen untersucht. Die Zusammenarbeit von Fuzzy-Reglern und Neuronalen Netzen ist atraktiv, weil die Interpretierbarkeit von Fuzzy-Systeme mit den Lernmöglichkeit von Neuronalen Netzen effektiv zu verbinden ist. In den nächsten Unterkapiteln werden zwei Arten von Modelle mit jeweils einer Beispielstruktur - Modelle für feste Lernaufgaben und solche mit verstärkendem Lernen. Die Modelle werden außerdem in der Literatur als Neuro-Fuzzy-Systeme, bzw. Neuro-Fuzzy-Regler, bezeichnet. [RCC+15]

2.10 Neuro-Fuzzy-Regler

Neuro-Fuzzy-Regler, oder Neuro-Fuzzy-Systeme, sind Modelle, bei denen konzeptionelle Teile von Neuronalen Netzen und Fuzzy-Systeme kombiniert werden. Das Ziel dieser Reglern ist es das beste aus beiden Welten zu kombinieren - Lernbarkeit von Neuronalen Netzen und Interpretierbarkeit von Fuzzy-Systeme. Außerdem bietet Fuzzy-Systeme die Möglichkeit Vorwissen einzubrigen, was hingegen die Lernzeit der Neuronalen Netze verkürzt. Am Ende des Lernprozesses erhält man ein Modell, dessen Regelungsstragie interpretierbar ist und dessen Regelung überprüft und eventuell noch angepasst werden können [RCC+15]. Demnächst werden zwei Arten von Neuro-Fuzzy-Modelle, die unterschiedliche Anwendungsfälle haben.

2.10.1 Modell für feste Lernaufgaben

Neuro-Fuzzy-Modell für feste Lernaufgaben versuchen, Fuzzy-Mengen und, bei TSK-Modelln, die Parameter der Ausgabefunktion unter Einreichung einer Mengen von Ein-/Ausgabe-Tupeln zu optimieren. Diese Modell sind genau dann sinnvoll, wenn schon eine Fuzzy-Regelbasis vorliegt. Die Regelbasis unterliegt infolge des Lernens eine Verarbeitung, die als Ziel eine Optimierung hat.

Ein weiteres Anwendungsbeispiel ist bei bereits existierender Regelbasis, dass dieser mit einer neuen ausgetauscht wird. Falls die Basis schon errechnete Werte geliefert hat, können diese zusammen mit den zugehörigen Eingabewerten dem Neuro-Fuzzy-System zum Lernen gegeben werden. Am Ende erhält man eine optimierte Fuzzy-Regel-Basis, die die alte Basis "ersetzt".

Falls es keine angemessene Lernaufgabe bereits gegeben ist, dann eignet sich dieses Verfahren nicht. Es existieren natürlich Ansätze, die es ermöglichen, einen initialen Regelbasis aus Eingabedaten zu erstellen. Ein solcher Verfahren wird später noch vorgestellt.

Folglich wird ein Beispiel von einem Modell, das sich für feste Lernaufgaben eignet, vorgestellt. Es geht nämlich um das ANFIS-Modell.

Das ANFIS-Modell

Im Frühjahr von 1993 wurde das Neuro-Fuzzy-System ANFIS (Adaptive Neuro-Fuzzy Inference System oder Adaptive Network-based Fuzzy Inference System) entwickelt. Das Modell wurde in mehrere Software-Pakete schon eingesetzt. Die ANFIS basiert auf einer hybriden Struktur, sodass es sowohl als ein Neuronales Netz, als auch als ein Fuzzy-System interpretiert werden kann. In dem System sind Regeln angelegt, die nach dem TSK-Modell definiert sind (Takagi-Sugeno-Kang-Reglern 2.5). Die Abbildung 2.9 zeigt einen Modell mit folgender Regelbasis:

R_1: Falls x_1 ist A_1 und x_2 ist B_1, dann ist $y{=}f_1(x_1{,}x_2)$
R_2: Falls x_1 ist A_1 und x_2 ist B_2, dann ist $y{=}f_2(x_1{,}x_2)$
R_3: Falls x_1 ist A_2 und x_2 ist B_2, dann ist $y{=}f_3(x_1{,}x_2)$,

dabei sind A_1, A_2, B_1 und B_2 linguistische Termen, die den entsprechenden Fuzzy-Mengen $\mu_i^{(j)}$ zugeordnet sind. Die Funktionen f_i sind linear und sehen wie folgt aus (siehe 2.5):

$$f_i(x_1, x_2) = p_0^i + p_1^i \cdot x_1 + p_2^i \cdot x_2 \tag{2.19}$$

Der Konklusionsfunktion f_i entspricht die Regel R_i. Somit hat jede Regel eine eindeutige Ausgangsfunktionen mit eindeutigen Parametern p_0^i, p_1^i, p_2^i.

Die Ausgabe eines ANFIS-Modells berechnet sich genau so wie ein TSK-Modell (siehe 2.5). Für den genannten Beispiel lautet die Ausgabe:

$$f = \frac{\sum_i^3 \tilde{w}_i \cdot f_i(x_1, x_2)}{\sum_i^3 \tilde{w}_i} \tag{2.20}$$

Der ANFIS-Ansatz besteht aus 5 Schichten. Die Abbildung 2.9 veranschaulicht die Struktur.

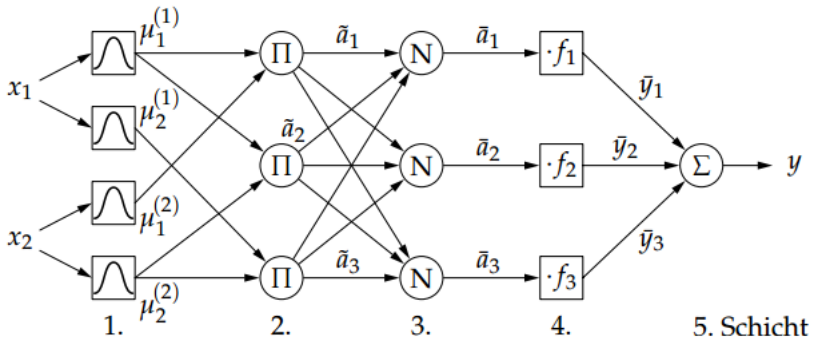

Abb. 2.9: ANFIS-Model [RCC+15]

In der **ersten Schicht** werden die Eingabewerte eingereicht und entsprechend die Zugehörigkeiten zu den Fuzzy-Sets ausgegeben. Weiterhin werden in der **zweiten Schicht** die Aktivierungswerte jeder Regel ausgewertet. Die Neuronen werden mit $\prod$ gekennzeichnet. Einzelnen Fuzzyzugehörigkeitswerte werden mittels Operatoren kombiniert, um die Aktivierungsgrad jeder Regel zu berechnen. Hier dürfen Operatoren zur Verknüpfung von Fuzzy-Mengen eingesetzt werden, üblicherweise der UND-Operator (siehe 2.3.1). Die Gleichung 2.21 ergibt die Berechnung bei gegebener UND-Verknüpfung:

$$\tilde{w}_i = \prod \mu_i^j(x_j) \tag{2.21}$$

Die Variable $\tilde{w}_i$ ergibt die Aktivierung, oder Erfüllungsgrad, des Regels R_i und μ_i^j ist die j. Zugehörigkeitsfunktion in der Regel R_i.

Im **dritten Schicht** findet die Normalisierung aller Aktivierungswerte $\tilde{w}_i$ statt. Mit einfachen Worten wird der Beitrag berechnet, den jeder Regel für den Gesamtausgabe beiträgt. Nach der Normalisierung erhält man Aktivierungsgrößen zwischen 0 und 1. Die Gleichung 2.22 berechnet die normalisierten Werten für jeden Regel R_i:

$$\bar{w}_i = \frac{\tilde{w}_i}{\sum_j \tilde{w}_j} \tag{2.22}$$

Im **vierten Schicht** berechnen die mit N markierten Neuronen die gewichteten Ausgabewerte. Das "Gewicht" (Ergebnis) aus dem letzten Layer wird mit der entsprechenden Ausgabefunktion multipliziert:

$$\bar{y}_i = net_i = \bar{w}_i \cdot f_i(x_1, ..., x_n). \tag{2.23}$$

Im **fünften Schicht** steht ein einziges Neuron, der mit $\sum$ beschriftet ist. Im letzten Schicht berechet man die Ausgabe, indem alle Werte aus dem **vierten Schicht** zusammenaddiert werden:

$$y = y_{out} = \sum_i \bar{y}_i = \frac{\tilde{w}_i \cdot f_i(x_1, ..., x_n)}{\sum_j \tilde{w}_j}. \tag{2.24}$$

Diese Struktur ähnelt sich der von TSK-Modell. Für die Optimierung von Parametern der Zugehörigkeitsfunktionen und Konklusionsfunktionen eignet sich der ANFIS-Ansatz.

Das ANFIS-Modell ermöglicht die Optimierung von Modellparametern - die Fuzzy-Mengen- und Ausgabefunktionsparametern. Diese können erlernt werden, wenn eine angemessene Lernaufgabe vorliegt. Außerdem muss eine ausreichende Menge von Ein-/Ausgabe-Werten zur Verfügung stehen. Es bieten sich mehrere Lernmethoden zur Optimierung der Parametern. Zwei davon sind Gradientenverfahren(analog zur Fehler-Rückpropagation-Verfahren aus Neuronalen Netzen) und die Kleinste-Quadrate-Methode. [RCC+15] [Jan93]

2.10.2 Modell mit verstärkendem Lernen

Bei Modellen mit verstärkendem Lernen wird versucht, die Menge der Daten für das Lernen möglichst gering zu halten. Der Unterschied zwischen Modell mit verstärkendem Lernen und solchen für feste Lernaufgaben besteht darin, dass bei dem Erstgenannten keine Vorwissen bekannt werden müssen, was öfters der Fall sein kann. Es reicht nur, wenn im Laufe des Lernens angegeben wird, ob die Richtung der Optimierung sinnvoll ist.

Ein großes Problem beim verstärkendem Lernen besteht darin, vorzusagen, wie groß der Einfluss einer Regelaktion auf das Gesamtsystem ist. Dieses Problem wird als *Credit Assignment Problem* bezeichnet.

Es existiert eine große Mengen von Modelln mit verstärkendem Lernen, alle aber basieren auf dem gleichen Prinzip. Das System wird zwei Teilsysteme aufgeteilt: zum einen der "Kritiker"(das "kritisierende" System) und der Aktor(zuständig für die Anwendung und Abspeicherung der Regelungsstrategie). Der Kritiker "äußert" seine Meinung über den jetzigen Zustand unter Berücksichtigung der vorhergehenden Zustände und somit entscheidet der Aktor anhand der Bewertung, ob eine Korrektur der Regelbasis gemacht werden soll. [RCC+15] [AR]

Das NEFCON-Modell

Ziel des NEFCON-Modell, Neuro-Fuzzy Control Modell, ist es, eine interpretierbare Fuzzy-Regelbasis mit möglichst kleinen Trainingsschritten zu erlernen. Das Modell unterscheidet sich von dem ANFIS, indem es erlaubt einen Regelbasis, ohne Vorwissen zu erlernen. Dieses Modell bietet natürlich auch die Möglichkeit, Vorwissen mitzubringen. Das heißt sowohl Fuzzy-Systeme mit vorhandenen Regelbasis, als auch solche mit unvollständiger Fuzzy-Regelbasis. Alle diese Vorteilen sprechen für das NEFCON gegenüber andere Modelle.

Das NEFCON-Modell basiert auf ein Mamdani-Regler. Zum Veranschaulichung wird hier einen kleinen Beispiel mit Grafik 2.10 und Regelbasis gegeben: [RCC+15]

R_1: IF x_1 in $A_1^{(1)}$ AND x_2 in $A_1^{(2)}$, THEN y is B_1
R_2: IF x_1 in $A_1^{(1)}$ AND x_2 in $A_2^{(2)}$, THEN y is B_1
R_3: IF x_1 in $A_2^{(1)}$ AND x_2 in $A_2^{(2)}$, THEN y is B_2
R_4: IF x_1 in $A_3^{(1)}$ AND x_2 in $A_2^{(2)}$, THEN y is B_3
R_4: IF x_1 in $A_3^{(1)}$ AND x_2 in $A_3^{(2)}$, THEN y is B_3

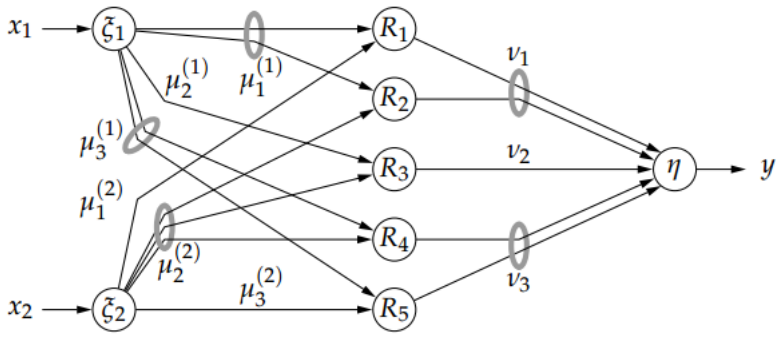

Abb. 2.10: NEFCON-Modell [RCC+15]

Das NEFCON-Modell basiert auf einem generischen Fuzzy-Perzeptron. Das Modell konnte in drei Schichten aufgeteilt werden. Die *erste Schicht* besteht natürlich aus den Eingangsneuronen. Die ist eindeutig und will nicht weiter darauf eingehen. In der *zweiten Schicht* befinden sich die inneren Neuronen, die die Regeln einer Fuzzy-System widerspiegeln. Im Beispiel sind insgesamt fünf Regeln gegeben. Die Fuzzy-Mengen $\mu_r^{(i)}$, die in Mehrere Regeln vorhanden sind, werden durch Ellipse zusammengeführt. Falls beim Lernen eine Anpassung an einem Gewicht durchgeführt werden soll, muss dies in allen Verbindungen gemacht werden, wo die Menge figuriert.

Der eigentliche Lernprozess besteht aus zwei Phasen. In der erste Phase wird versucht, eine Regelbasis erlernt zu werden. Diese Phase wurde weggelassen, falls schon eine existiert. Es lassen sich auch unvollständige sogar fehlende Regelbasen lernen. Für das Letztere ist ein weiterer Algorithmus erforderlich.

In der zweiten Phase findet die Optimierung statt. Dabei werden Fuzzy-Sets modifiziert oder selbst die Verbindungen zu den Regeln umgetauscht. In dem NEFCON-Modell wird als Bewertungsmaß, der "Kritiker", ein Fuzzy-Error verwendet. Damit die Optimierung optimal ausgeführt werden kann, sollte das Vorzeichen des Ausgabewertes bekannt sein. Darüber hinaus wird ein erweiterter Fuzzy-Error E^* berechnet:

$$E^* = sgn(y_{out}) \cdot E(x_1, \ \ldots, \ x_n) \tag{2.25}$$

[RCC+15] [AR]

2.11 Erlernen einer Regelbasis

Es existieren mehrere Algorithmen zum Erlernen von Regelbasis. Die Methoden können in drei Kategorien aufgeteilt werden: solche, die ohne vordefinierte Regelbasis startet; solche mit vollständiger Regelbasis und solche mit zufälliger Basis starten. In den folgenden zwei Unterkapiteln werden Methode für die ersten zwei Kategorien vorgestellt. Bei den Methoden wird keine feste Lernaufgabe benötigt. Tatsächlich geht es darum, dass eine Initialbasis aufgebaut wird, ohne eine große Datenmenge mit optimalen Werten bekannt zu sein. [RCC+15] [AR]

2.11.1 Top-Down- oder Reduktionsmethode zum Erlernen einer Regelbasis

Zum Einen wird die Methode der Top-Down-Methode (in der Literatur auch als NEFCON I bekannt) genannt. Das Verfahren erfordt, dass eine vollständige Regelbasis vorhanden ist. Die Regelbasis beinhaltet auch widersprüchliche Regeln, die in Laufe des Prozesses ausgefiltert werden.

Der Prozess kann in zwei Phasen aufgeteilt werden. In der ersten Phase werden alle die Regeln eliminiert, die bei ihrer Ausgabe den falschen Vorzeichen aufweisen. Die auszufilternden Regeln werden mit der erweiterten Fuzzy-Fehler-Funktion (siehe 2.25) bestimmt. In der zweiten Phase werden Regeln mit identischer Prämisse zufällig ausgewählt. Folglich wird der Fehler für die bestimmte Regel berechnet. Zum Schluss wird die Regel ausgewählt, die die kleinste Fehlerrate aufweist. Die restlichen Regeln werden verworfen.

Eins der Nachteile der Top-Down-Methode ist die Aufwändigkeit, weil es mit einer großen Regelbasis gestartet wird. Das nächste Verfahren ist das Gegenteil von Top-Down-Methode, zwar Bottom-Up-Methode. [RCC+15]

2.11.2 Bottom-Up- oder Eliminationsmethode zum Erlernen einer Regelbasis

Der Bottom-Up-Algorithmus beginnt mit einer leeren Regelbasis. Jedoch muss eine initiale Aufteilung(Intervall) der Ein- und Ausgabewerten gegeben sein. Analog zur Top-Down besteht diese Methode auch aus zwei Phasen.

Erste Phase beginnt mit der Bestimmung der Prämisse für die Regeln. Der Prozess evaluiert jede Fuzzy-Menge mit bestimmten Eingaben und die Mengen, die den höchsten Zugehörigkeitsgrad aufweisen, werden ausgewählt. Aus den ausgewählten Fuzzy-Mengen werden neue Regeln gebaut. Danach versucht der Algorithmus eine geeignete Ausgabe aus dem aktuellen Fuzzy-Fehler zu “raten”. Dabei wird vorausgesetzt, dass Eingaben mit ähnlichen Fehlerwerten ähnliche Ausgaben liefern.

In der zweiten Phase werden die Fuzzy-Mengen in den Konklusionen optimiert. Dabei werden nicht die Parametern der Konklusionen angepasst, sondern bei Bedarf die Fuzzy-Menge durch eine andere ersetzt.

Wegen des inkrementellen Lernen lässt sich einfach Vorwissen in dem Regelbasis einführen. Bei den Fällen mit unvollständigen Regelbasen werden passende

Regeln hinzugefügt. Das Verfahren garantiert jedoch nicht, dass immer eine geeignete Regelbasis aufgebaut wird. Aus diesem Grund ist es empfohlen, dass die Regelbasis manuell am Ende des Lernens überprüft und entsprechend angepasst wird. [RCC+15] [AR]

2.12 Optimierung der Regelbasis

Die Optimierung bei NEFCON verwendet die Methode "Back Propagtion Method", oder Rückpropagationsmethode. Der Fehler wird rückwirkend durch das Netz geführt und lokal bei jeder Fuzzy-Menge angewendet.

Eine Änderung darf sowohl in den Prämissen, als auch in den Konklusionen. Es wird das Prinzip des verstärkenden Lernens angewendet. Das bedeutet, dass für jede Änderung eine Fuzzy-Menge entweder "bestraft" oder "belohnt" wird. Bestrafung und Belohnung werde in Änderungen wie Verschiebung, Vergrößerung oder Verkleinerung des Bereichs einer Fuzzy-Menge ausgedrückt. Änderungen werden entsprechend iterativ in Beziehung zum Fuzzy-Fehler gemacht. [RCC+15] [AR]

2.13 Schlussworte

Die Methoden unterscheiden sich in ihrem Kern kaum. Beide Architekturen besitzen entsprechend Vorteile und Nachteile. Der größte Unterschied liegt in der Ausbau ihrer Struktur. ANFIS basiert auf TSK-Modellen, währen NEFCON auf die Mamdani-Reglern. Beide Modelle bieten sie sich gut für das Lernen von Fuzzy-Systeme, jedoch entspricht nur das ANFIS meinem Anwendungsfall - Optimierung von TSK-Modelle.

3

Neuro-Fuzzy-Modelle: Implementierung des ANFIS-Ansatzes

In der folgenden Kapitel stelle ich meine Aufsetzung des ANFIS-Ansatzes in Python vor. Während der Ausarbeitungszeit habe ich mich mit der Bibliothek für Neuronale Netze Tensorflow zusammengesetzt. Tensorflow ist ein Produkt von Google und eins der verbreitendsten APIs für Erstellung von Neuronalen Netzen. Wegen zahlreicher Nutzung ist das Web befüllt mit Guides und Tutorials über das Bauen von Neuronalen Netzen mit der Bibliothek Tensorflow. Aus diesem Grund habe ich mich für ihre Verwendung entschieden. Eine weitere Funktionalität, die die Bibliothek anbietet, ist die einfache Migration von erstellten Neuronalen Netzen. Tensorflow bietet die Möglichkeit, einfach Modelle nach den unterstützten Programmiersprachen zu exportieren, in disem Sinne auch nach C++.

In diesem Artikel wird so vorgegangen, dass zu jedem Programmcode eine Erklärung gegeben wird. Vorrigen Artikeln gehen auf die eigentliche Struktur und theoretische Aufsetzung eines ANFIS-Models in einem Neuronalen Netz. Laut einem dieser Artikeln besteht ein System aus sechs Schichten (zwei äußere und vier inneren Schichten). Über die erste Schicht erfolgt die Eingabe im Netz. Die weiteren fünf Schichten führen einfache mathematische Funktionen aus.

3.1 ANFIS-Klasse

Das Neuro-Fuzzy-Model wird nach dem bekannten ANFIS-Model eingerichtet. Das Model hat insgesammt 6 Schichten, 2 Außen- und 4 Innenschichten. Das ANFIS-Model wird in einer Klassendatei ausgelagert. Die Klasse verfügt über mehrere Methoden, einige davon Hilfsmethoden. Folglich werden die Wichtigsten davon in Unterkapiteln gestellt.

3.1.1 Konstruktor

Die ANFIS-Klasse vefügt über einen einzigen Konstruktor. Er hat einen Pflicht- und drei Optionalparameter - *num_sets* und entsprechend *path*, *mf_type*, *gradient_type*. Der Parameter *num_sets* besagt wie viele Fuzzy-Mengen pro Eingangsgröße zu erstellen sind. Der Pfad wird durch ein String - *path* - gegeben. Weiterhin unterscheide ich zwischen zwei Weisen der Berechnung von Zugehörigkeit, deren Vergleich wurde in der weitere Dokumentation erläutert. Anschliesend ergibt die

Variable *gradient_type* die Größe der Trainingsdaten, oder die Art des Gradient Descent. Es unterscheiden sich drei Arten von Gradient Descent Verfahren - Stochastic, Mini-Batch und Batch Gradient Descent Verfahren. Die gegebene Anordnung der Begriffe entspricht der in dem Programmcode, sprich *gradient_type=0* entspricht dem stochastischen Verfahren. Der Wert *1* heißt, dass das Mini-Batch ausgewählt wird und *2* - Batch Gradient Descent. Die drei Arten unterscheiden sich in dem Punkt, dass die Variablen zu Unterschiedlichen Zeitpunkten angepasst werden. Bei dem stochastischen Verfahren werden die Parameter nach jeder Bearbeitung von Trainingspaaren verändert. Bei dem Mini-Batch wird jeweils nach der Bearbeitung von gewisser Anzahl an Datenelementen. Eine übliche Anzahl ist meistens zwischen 30 und 500 Trainingsdaten pro Iteration, jedoch ist das weniger als der Gesamtmenge. Der Batch Gradient Verfahren nimmt die ganze Datenmenge und führt mit denen einen Lernzug. Die Erkenntnisse aus den Tests werde ich in einer weiteren Dokumentation beschreiben.

Der Konstruktor umfasst Definition der benötigten Variablen und die Erstellung des Models. Zu beginn werden die Trainingsdaten aus einer Datei gelesen. Durch die Struktur des Datensatzes wird die Anzahl der Eingangvariablen in dem Netz bestimmt. Die Datenpaar sind zeileinweise aufgebaut. In der Zeile werden zu bestimten Eingaben die Soll-Ergebnisse gegeben. Die letzte Spalte liefert die Sollwerte und in den Spalten davor werden die Eingangswerte geschrieben, sodass zum Beispiel in der ersten Spalte $\mathbf{x_1}$, in der zweiten $\mathbf{x_2}$ usw. und in der letzen der Erwartungswert steht. Üblicherweise arbeite ich mit einer Eingangsgröße, somit hat eine Datei zwei Spalten. Falls das gegebene Pfad gültig ist, werden vier Felder erstellt, zwei jeweils fürs Lernen und die Prüfung. Falls der Parameter *fulltrain* auf *wahr* gesetzt ist, werden die Daten nicht aufgeteilt. Bei dem Vorderen werden die Daten so getrennt, dass drei viertel der Datensätze fürs Training und ein viertel der Daten für die Bewertung des gelernten Modells verwendet werden. Anschließend wird mit der Konstruktion des Neuronalen Netzes fortgefahren. Folglich werden sechs weiteren Hilfsmethoden ausgeführt, die die Schichten des Neuronalen Netzes ausbauen. Diese werde ich etwas ausführlicher vorstellen.

Ein Beispiel für einen Aufruf des Konstruktors:

```
1 #...
2 # the default mf_type and gradient_type
3 # are set to 0, although gradient_type 0 is not the most optimal
4  f = Anfis(num_sets=num_sets,
5     path="../utils/sinus.out", gradient_type=0, mf_type=0)
```

3.1.2 Eingangsschicht (FirstLayer)

Die Definition der Eingangsschicht erlangt in der Funktion *first_layer()*. Die Methode erhält keine Parameter und initialisiert die zwei Eingangsgrößen. In Tensorflow können Eingaben an mehreren Stellen im Netzt eingeführt werden, was hier auch der Fall für die **Y-Variable** ist. Die **Y-Variable** wird in der Fehlerfunktion gebraucht und sie enthält alle Soll-Ergebnisse.

```
1 #...
2 # three different types for gradient descent type
```

```
3  # so we have three types of batch size,
4  # wich defines the amount of elements to be evaluated in one interation
5   if self.gradient_type == 0:
6      self.batch_size = self.num_inputs
7   elif self.gradient_type == 1:
8      self.batch_size = int((1 / 5) * (len(self.train_x_arr)))
9   elif self.gradient_type == 2:
10     self.batch_size = len(self.train_x_arr)
11 # input variable
12  self.x = tf.placeholder(name="x", shape=(self.num_inputs, self.batch_size),
13     dtype=tf.float64)
14 # variable for expected result
15  self.y = tf.placeholder(name="y", shape=(self.batch_size), dtype=tf.float64)
16 #...
```

Die **Variable X** ist ein Array, das eine Matrix mit n *(num_inputs)* Zeilen (Werten) und *batch_size* Spalten repräsentiert. Die Methode *placeholder* in Tensorflow erstellt einen Tensor, der keine zusätzliche Operation oder Funktion ausführt. *Placeholders* werden meistens als Eingangsvariablen verwendet. In der Variable *num_inputs* wird die Anzahl der Eingangsvariablen geschrieben. Im normalen Fall entspricht der Wert von n gleich 1. Weitere Größen für n werden nicht betrachtet. Der **Y-Placeholder** wird auch als Eingabe in dem Verlustfunktion genutzt, die später in dem Artikel gegeben wird. Die **Variable Y** enthält den Erwartungwert für gegebener **Eingangswert X**. Der eigentliche Ausgangswert(Istwert) wird von der letzen Schicht berechnet.

3.1.3 Konklusionsparameter

Nach der Definition der Eingagnsvariablen erfolgt die Erstellung von zwei Variablen, in dennen die Konklusionsparametern gespeichert werden. In diesem Fall handelt es sich nicht mehr um *Placeholdern*, sondern *Variablen*. Ihre Konstruktion erfolgt über die Hilfsfunktion *get_variables()*. Die Tensorflow-Dokumentation beschreibt diese sehr ausführlich. Die Funktion überprüft, ob eine Variable mit gegebener Name existiert. Variable wird erstellt, wenn das nicht der Fall ist, ansonsten wird die entsprechende Variable zurückgeliefert.

Im Kapitel 1 und in 2 aus der Dokumentation ist der Normalform der Konklusionsfunktionen eines Takagi-Sugeno-Kang-Modells definiert. Die Form der Gleichung ist hier nochmal gegeben:

$$f_i(x_1, ..., x_m) = w_0^i + w_1^i \cdot x_1 + ...w_m^i \cdot x_m \tag{3.1}$$

Jede Konklusionsfunktion hat ihre eigene Parameter. Damit keine einzelnen Variablen definiert werden, können diese in einem Vektor, bzw. Matrix, gespeichert werden. Die Stelligkeit der ersten Variable a_0, w_0^i in der Gleichung, hängt von der Anzahl der Regeln. Wir rechnen mit insgesammt $num\_sets^{num\_inputs}$-Viele Regeln (*num_rules*). Für die restlichen Konklusionsparametern a_y wird eine Matrix mit Dimensionen $num\_rules \times num\_inputs$ erstellt. Beide Variablen sind trainierbar, was aus dem Parameter *trainable* ersichtlich wird. Folgendes Programmabschnitt stellt die Umsetzung dar.

```
#...
    self.a_0 = tf.get_variable(name="a_0", dtype=tf.float64,
    initializer=np.ones(shape=(self.num_rules, 1)), trainable=1)
    self.a_y = tf.get_variable(name="a_y", dtype=tf.float64,
    initializer=np.ones(shape=(self.num_rules, self.num_inputs)),
trainable=1)
#...
```

3.1.4 Zugehörigkeitsfunktion (Second and Third Layer)

Die erste Aufgabe sei die Definition der **Zugehörigkeitsfunktionen (ZGF)**. Die Gesamtzahl der ZGFs ergibt sich aus dem Produkt der beiden Werten für Anzahl Fuzzy-Sets und Eingangsgrößen (*num_sets* und *num_inputs*). Die Erstellungsmethode wird *mfs()* genannt. Zwei Operationen werden in der Funktion ausgeführt - Erstellung der einzelnen ZGF und deren Kombination in Premissen.

Definition in Tensors aller Zugehörigkeitsfunktionen

In meinem Projektes habe ich mich nur mit Dreiecksfunktionen beschäftigt, in der Dokumentation werden weitere Zugehörigkeitsfunktionen vorgestellt. Wie die Name verraten würde, stellt die Dreiecksfunktion einen Bereich, der durch drei Punkten aufgespannt ist, dar. In der Methode *second_layer* werden zuerst alle Funktionsparametern iterativ mit einem Wert erstellt. Jeder Parameter wird in einem trainierbaren Tensor gespeichert. Dies erfolgt über die Hilffunktion *triangular_mf*, die entsprechend so aussieht:

```
def triangular_mf(self, x, par, name):
    # we define the triangular function
    if self.mf_type == 0:
    # we split the traingular space into two
    # the left side
    # we calculate the membership of x to the left side
    # we get a negative value if x is not part of the space
        dividend_left = tf.subtract(x, par[0])
        divider_left = tf.subtract(par[1], par[0])
        division_left = tf.divide(dividend_left, divider_left)
    # we calculate the membership of x to the right side
        dividend_right = tf.subtract(par[2], x)
        divider_right = tf.subtract(par[2], par[1])
        division_right = tf.divide(dividend_right, divider_right)
    # we get the lowest value
        minim = tf.minimum(division_left, division_right)
    # then we calculate the highest of 0 and the previous value
        maxim = tf.maximum(minim, tf.cast(0.0, tf.float64))
    elif self.mf_type==1:
    # first we calculate the dividend. We subtract the middle parameter
    #of the mf with x
    dividend = tf.abs(tf.subtract(par[1], x))
    # then we subtract the two boundary parameters of the membership function
        divider = tf.subtract(par[2], par[0])
    # we divide the two values to get the output of the memberhip function
        op = tf.divide(dividend, divider)
    # then we multiply it by 2, because we devide it by the whole length of
    #the triangle
    # and not just half of the length.
        mul = tf.multiply(tf.cast([2.], tf.float64), op)
    # we subtract 1 from it so we always either have a positive value when x
    #is a member
```

```
33      # of the set, or we get a negative value
34          sub = tf.subtract(tf.cast([1.], tf.float64), mul)
35      # we calculate the maximum of the value from the mf and 0, this way we
36      #don't take variables that are outside
37      # the range of the membership function
38          maxim = tf.maximum(tf.cast([0.], tf.float64), sub, name=name)
39      return maxim
```

Die Hilfsfunktion *triangular_mf()* nimmt drei Argumente ein. Die *Variable* x kann mehrere Einträge repräsentieren, abgesehen davon welcher Art der Gradienten Verfahren verwendet wird. Der zweite Methodenparameter speichert in sich die davor definierten Funktionsparametern. Schließlich wird eine Name als String gegeben. Jede Funktionsname ist eindeutig gewählt, so dass auf jede Funktion bei Bedarf über die *get_variable()*-Methode zugegriffen werden kann. Außerdem vereinfacht die Namensetzung das Debugging, weil alle Variablen eindeuting sind.

Erster Programmteil (Second Layer)

Im ersten Teil der Methode *second_layer()* erlangt die Definition aller Fuzzymengenparametern. Ich habe mich für diese Vorgehensweise entschieden, da die getrennte Erstellung der Variablen ermöglicht, dass jeder Parameter spezifisch definiert ist. Auf dieser Weise könnte ich mehr Einfluß auf einzelnen Ergebnisse und Vorgänge haben.

Algorithm 1 Definition of Fuzzy member parameters

Require: *value_scope*
Ensure: *val_inc* ← *increment*

```
for all par do
    if index = 0 then
        par ← tensor(val = value_scope[0], trainable = 0)
    else if index = last then
        par ← tensor(val = value_scope[1], trainable = 0)
    else
        par ← tensor(val_inc)
        increment val_inc
    end if
end for
```

Das Pseudocode beschreibt die Erstellung eines Parameters. Es wird unterschieden, welcher Parameter definiert wird. Die ersten und letzen Parameter sind nicht trainierbar, um Definitionslücken zu verhindern.

Es wurde schon erwähnt, dass alle Parametern aufsteigend angeordnet sind. Aus der Abbildung 3.1 wird ersichtlich, wie diese Parametern, insbesondere die ZGF, am Anfang aussehen.

Die Anordnung ist gleich bei allen Mengen außer die Erste. Damit keine Definitionslücken entstehen, verzichte ich auf die Trainierung der beiden Grenzwertparametern. Neben dem Trainingsausschluß des ersten Parameter der ersten Menge

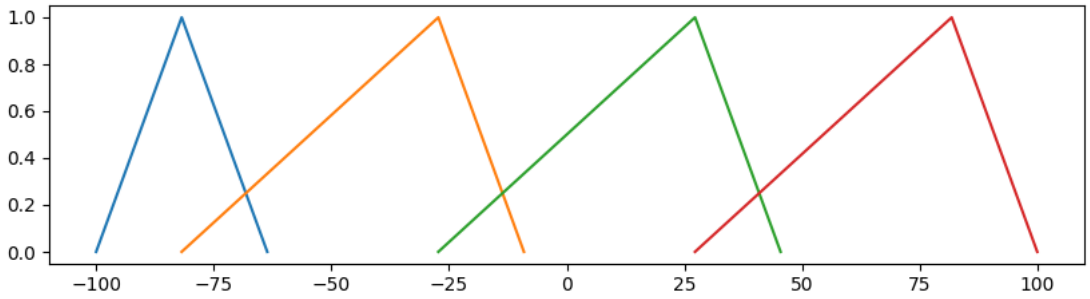

Abb. 3.1: Ursprüngliche Fuzzy-Mengen

und des letzten der letzen Mengen, die die Wertebereich bestimmen, werden auch weitere Einschränkungen angelegt. Zum einen wird der Wert der Spitzenparameter(mittlerer Parameter) jeder Funktion begrenzt, sodass er nicht kleiner als der linke Parameter, bzw. größer als der rechte Parameter, wird. Ohne diese Regel entstehen manchmal ungültige Fuzzy-Mengen.

Infolge des Austesten tauchten kritischen Fehler bezüglich der Undefiniertheit der Zugehörigkeitsfunktionen auf. Diese Mangeln könnte ich mit drei Einschränkungen aufheben. Die Beschränkungen fallen auf die drei Parametern, *a, m, b*, genauer gesagt die Dreiecksfunktionsparameter:

```
1 # ...
2  self.constraints.append(tf.assign(a, tf.clip_by_value(a,
3        self.range[0], m- 1e-2)))
4  self.constraints.append(tf.assign(b, tf.clip_by_value(b,
5       m + 1e-2, self.range[1])))
6  self.constraints.append(tf.assign(m, tf.clip_by_value(m,
7        self.range[0] +0.1, self.range[1] - 0.1)))
8 # ...
```

Zweiter Programmteil (Third Layer)

Der zweite Programmteil enthält weniger Programmcode, aber dauert unter Umständen länger. Die lange Laufzeit ergibt sich aus der Berechnung der Permutationen der ZGF für alle Eingaben. Die Bibliothek *itertools* stellt mehrer Funktion bereit. Eine der Methoden daraus berechnet das kartesische Produkt. Die Funktion kann mehrere Mengen und einen **repeat**-Parameter entgegennehmen und liefert eine Ergebnismenge. Die **repeat**-Variable gibt an, wie oft das kartesische Produkt ausgeführt wird. Die Iteratorfunktion ist in der Komplexitätsklasse $TIME_{product()} \in O(n^k)$, mit n gleich die Anzahl der Elemente in der Angegebene Menge und k gleich die Anzahl der Ausführungen. Ein kleines Beispiel zur Verdeutlichung als auch eine Implementierung des zweiten Programmteils wird unten gegeben.

```
1 # itertools.product(l, r) builds the cartesian product of
2 #the iterables (the list l).
3 # if r is equal to two, this would return the cartesian product
4 #of the list l with itself (l x l).
5 # if r is three - (l x l x l). We use it here to get all
6 #the combinations between the inputs
7 # Example: product((A, B), repeat=2) would result in the
8 #list [(A, A), (A, B), (B, A), (B, B)]
```

```
 9  for perm in it.product(range(self.num_sets), repeat=self.num_inputs):
10     tmp = tf.ones(shape=(), dtype=tf.float64)
11  # loop from 0 to self.num_inputs and multiply the MF of an
12  #input with index perm[i] with every other MF
13  #with index from the permutation list
14     for i in range(len(perm)):
15        tmp = tf.multiply(premises[i][perm[i]], tmp)
16     self.mf[index] = tmp
17     index += 1
18  end = t.process_time()
```

Aus der Komplexitätsklasse ergibt sich, wie oft die äußere Schleife ausgeführt wird. Genauer gesagt, der Loop braucht so viele Iterationen bis zum Schluss wie die Gesamtzahl der Regeln. In der innere Schleife werden die Elemente in der Liste durchlaufen. Pro Iteration wird ein Element aus der Productliste betrachtet. Jeder Eintrag aus der Liste repräsentiert ein Index **perm[i]** einer Zugehörigkeitsfunktion für die entsprechende Eingabe **i**. Die Funktionen werden der Reihe nach mit einem Dummy-Tensor multiplizert. Auf dieser Weise werden die Prämisse jeder Regel definiert.

3.1.5 Fourth Layer

Im dritten Layer laut der Struktur des ANFIS-Models, werden alle Ergebnisse aus dem vorrigen Schicht normiert. Die Funktion besteht nur aus zwei Zeilen Code:

```
 1  def third_layer(self):
 2     # Reshape the MFs
 3     # Reshape the array, to be in the form of an array with
 4     #num_rules items in it
 5     # sum the elements in the array along the 0th dimension
 6     self.mf_sum = tf.reduce_sum(tf.add(self.mf, [1e-10]), 0)
 7     # Normalize the MFs
 8     # Add all the items in the vector and divide every item in
 9     #it with the sum of the elements
10     # This way we get the normalized membership functions
11     self.normalizedMFs = tf.divide(self.mf, self.mf_sum)
```

Mit der Funktion **reduce_sum()** werden die Einträge in einem Array, bzw. Matrix oder Vektor, zusammensumiert. Es kann einen zusätzlichen Parameter angegeben werden, der in der Dokumentation als **axis** beschriftet ist und der angibt, welche Achse entlang die Elemente summiert werden sollen. Wie man am oberen Programmcode erkennt, werden hier die Elemente die 0. Achse entlang summiert. Wenn wir eine Matrix betrachten würden, heißt das für die Summe, dass die Elemente in einer Spalte summiert werden. Das Ergebnis der Summierung wird entsprechend in einer Variable abgespeichert, da wir einen zwei-Dimensionalen Array mit jeweils einen Eintrag in der zweite Dimension, erhalten wir eine Zahl. Schließlich werden alle Zugehörigkeitsfunktionen durch diesen Wert geteilt. Auf diesem Weg wird das Ergebnis der Prämissen normiert und ihre Summe gleich *1*.

3.1.6 Fifth Layer

Die fünfte Schicht soll das Ergebnis aus jeder Konklusionsfunktion berechnen. Die Ausgabe erhalten wird, in dem die Eingangsgrößen in jeden Konklusionsfunktion

eingesetzt werden und der Zutriffswert der Regeln an dem Funktionswert multipliziert werden. Durch die Normalisierung der Eintretungsgrößen erhalten wir eine Summe der Gewichte von 1, also ein Ergebnis mit Faktor kleiner 1 wird ausgegeben.

```
1  def fifth_layer(self):
2      # multiply every premis with its coresponding conclusion function
3      # we get a vektor of results with each result having a factor less than 1
4      self.outputs = tf.multiply(self.normalizedMFs, self.conclusions)
```

3.1.7 Sixth Layer

Die Summe aller Werten und das eigentliche Ergebnis für ein gegebenes Kandidat **X** wird in der letzen (sechsten) Schicht berechnet. Dies erfolgt in einem Einzeiler:

```
1  def sixth_layer(self):
2      self.result = tf.reduce_sum(self.outputs, 0)
```

3.2 Optimisierungsfunktion

Nachdem das Neuronale Model aufgebaut wird, folgt die Definition des Optimierungsverfahren. In der Literatur, auch in der Praxis, wird die Kleinste-Quadrate Funktion verwendet. Auch hier habe ich diese eingesetzt. Als nächstes muss der Optimiser definiert werden. Die zwei bekanntesten sind *GradientDescnetOptimizer* und *AdamOptimizer*. Bei der Auswahl gibt es keinen klaren besseren Algorithmus. Um den richtigen Optimisierer auszuwählen, muss man mit den unterschiedlichen Algorithmen gespielt haben, bis der Beste gefunden wird. In meinem Fall ist der AdamOptimizer besser geeignet. Unten ist die Definition der Funktion in Pseudocode gegeben.

```
1  def optimize_method(self):
2      self.loss = tf.losses.mean_squared_error(self.y, self.result)
3      # self.loss = tf.losses.huber_loss(self.y, self.result)
4
5      # self.optimizer =
6      #tf.train.GradientDescentOptimizer(learning_rate=0.01)
7      #.minimize(self.loss)
8      self.optimizer =
9          tf.train.AdamOptimizer(learning_rate=0.1).minimize(self.loss)
10     # self.optimizer =
11     # tf.train.RMSPropOptimizer(learning_rate=0.1).minimize(self.loss)
```

Wie üblich die Fehlerate ergibt sich aus dem Soll- und Istergebnis. Weiterhin erhalten wir den Optimierer durch einen simplen Codeaufruf, während dessen auch einen Lernrate gegeben wird. Normalerweise wird die Lernrate mit einem Wert von 0.1 gesetzt.

3.3 Trainingsfunktion

Mit den oben erklärten Funktionen kann einen ANFIS-Model mit beliebigen Eigenschaften erstellt werden. Nach dem Ausbau folgt das eigentliche Lernen. Darum habe ich eine separate Methode definiert, die den Ablauf beim Lernen beschreibt.

Algorithm 2 Training function

Require: initialize *statistic_arrays*
1: create *before_training_graphics*
2: **if** *gradient_type* = 0 **then**
3: *train_stocastic*()
4: **else if** *gradient_type* = 1 **then**
5: *train_mini_batch*()
6: **else**
7: *train_batch*()
8: **end if**
9: create *after_training_graphics*
10: save data

Die Trainingsfunktion kann in vier Teile abgegrenzt werden. In dem ersten Teil fallen alle Operationen für die Aufzeichnung der Grafiken vor dem Lernen. Im zweiten Teil fällt der Lernzug. Danach kommt entsprechend die Zeichnung des geänderten Models. Zum Schluss werden alle Daten in Dateien abgespeichert.

Im nächsten Kapitel wird das beschriebene Programm getestet. Die Tests haben zum Ziel einen Vertand zu verschaffen, wie schell das Modell lernt und wie das Modell am besten konfiguriert werden kann.

4

Analyse der Ergebnisse

In diesem Kapitel findet man Informationen über die durchgeführten Programmtests. Ziel der Testfälle ist es zu bestimmen, wie gut meine Implementierung des ANFIS-Models lernt. In jedem Unterkapitel wird gegen eine bestimmte Eigenschaft getestet. Bei der Untersuchung nehme ich in Betrachtung zwei mathematischen Funktionen, zwar die Parabel- und Sinusfunktion. Die Tests werden anhand Variation von drei Variablen - Anzahl Fuzzymengen und Iterationen, und Gradient-Descent-Arten.

Bei den Tests werden drei Arten von Gradient-Descent-Verfahren angesetzt - Stochastischen, Mini-Batch und Batch Gradienten Verfahren. Deren Eigenschaften werden in den nächsten Unterkapiteln erläutert.

Außerdem werde ich zwei unterschiedlichen Berechnungsweisen für die Zugehörigkeitsfunktionen verwenden. Das Endergebnis der Funktionen ist gleich, jedoch lernt das Modell unterschiedlich. Die zwei Arten nenne ich MF-Typ 0 und MF-Typ 1. Wenn ich vom MF-Typ 0, oder nur Typ 0, spreche, meine ich, dass die Zugehörigkeitsberechnung in zwei Teilen untergegliedert ist. Man kann das Dreieck, das durch die drei Parametern der Zugehörigkeitsfunktion bestimmt ist, durch die Mitte teilen. Dann entstehen zwei Bereiche, die separat betrachtet werden. Auf dieser Weise entstehen zwei Gleichungen (Teilen). Bei dem Typ 1 erfolgt die Berechnung in einer Gleichung. Also da wird nicht unterschieden, wo der X-Wert auf der Hypothenuse liegt - kleiner oder größer als der Mittelpunkt. Die zwei Berechnungsweisen für MF-Typ 0 4.1 und MF-Typ 1 4.2 gebe ich als Gesammtformel an.

$$\mu(x) = \max[\min(\frac{x-a}{m-a}, \frac{b-x}{b-m}), 0.0] \tag{4.1}$$

$$\mu(x) = \max[0, 1 - 2\frac{|x-m|}{b-a}] \tag{4.2}$$

Die erste Berechung ist von MF-Typ 0 und die Zweite von Typ 1. In der Gleichung sind die Größen *a, m und b* die Parametern der Zugehörigkeitsfunktion. Die Variable *a und b* sind die zwei Grenzwerte entsprechend links und rechts, und m ist der Gipfelpunkt, oder Mittelpunkt.

Alle unseren Lerndaten werden aus einer Datei, die einen spezifischen Aufbau hat, gelesen. Die Daten können in zwei Gruppen aufgeteilt werden - Eingaben und

Soll-Ergebnisse. Jede Spalte beinhaltet Elemente aus einer der beiden Gruppen, wobei die Letzte immer die Soll-Ergebnisse enthält. Da in diesem Projekt einstellige Funktionen in betracht genommen werden, gibt es in der Datei nur zwei Spalten.

Nachdem alle Verfahren vorgestellt wurden, wird mit den Tests begonnen.Wegen der größe Anzahl an Tests werden nur bestimmte ausgewählt. Eine Tabelle und alle Modellgrafiken werden später am Ende der Ausarbeitung angehängt.

Die Vorgehensweise bei der Beschriftung der Testfälle erfolgt für beide Funktionen gleich. Für ein Model wird seine Struktur und seine Eigenschaften gegeben. Weiterhin wird das Ergebnisgrafik gezeichnet. Zum Schluss werden die gelernten Konklusionsfunktionen vorgestellt. Zu jedem Teil wird eine Erläuterung zusätzlich aufgeführt.

Stochastic Gradient Descent

Bei dem stochastischen gradienten Verfahren (Stochastic Gradient Descent) handelt es sich um eine spezifische Eingabenweise. Bei dieser Vorgehensweise werden einzelnen Einträgen aus dem Datensatz als Eingabe in das Model gegeben. Somit beschreibt ein Element eine Iteration (Epoche) in dem Lernprozess. Nach jeder Iterationen werden die betroffenen Gewichten (Parametern) angepasst. Dieses Verfahren wird am seltesten von Allen (Stochastic, Mini-Batch und Batch) eingesetzt.

Mini-Batch Gradient Descent

Bei dem Mini-Batch Verfahren werden kleine Sets, normalerweise zwischen 30 und 500 Elemente, aus der Menge der Daten genommen und diese an dem Model zum Lernen gegeben. Das heißt eine Iterationen wird erst dann durchgeführt, wenn alle Elemente aus dem Batch abgearbeitet werden. Dieses Verfahren wird öfters bei Lernaufgaben verwendet, bei dennen der Datensatz extrem groß ist. In meinem Programm baue ich Zufälligkeit, sodass die Lernmenge zufällig aus der Gesamtmenge auszuwählen ist.

Batch Gradient Descent

Bei dem Batch Gradient Descent setzt man eine Architektur, bei der alle Daten in dem Lernprozess innerhalb eines Lernzugs fließen. Diese Art ist gut geeignet, bei Problemstellungen, wo die Datenbasis angemessen groß ist. In unseren zwei Fällen handelt es sich um relativ kleine Datenmengen (400 und 1000 Datensätze entsprechend für Sinus- und Parabelfunktion). Bei Lernaufgaben mit 100 Tausende von Datensätze ist diese Vorgehensweise ungeeignet. Da ist das Mini-Batch-Verfahren einzusetzen.

4.1 Lernen der Sinusfunktion mit Stochastic Gradient Descent

In diesem Kapitel wird die Sinusfunktion untersucht. Hier erläutere ich die Ergebnisse, die bei den Lernvorgängen erschaffen wurden. Zum Schluss verschaffe ich einen Überblick über die besten Eigenschaften zum Lernen der Sinusfunktion.

Die Sinusfunktion ist als Zeichnung (siehe 4.1) angegeben.

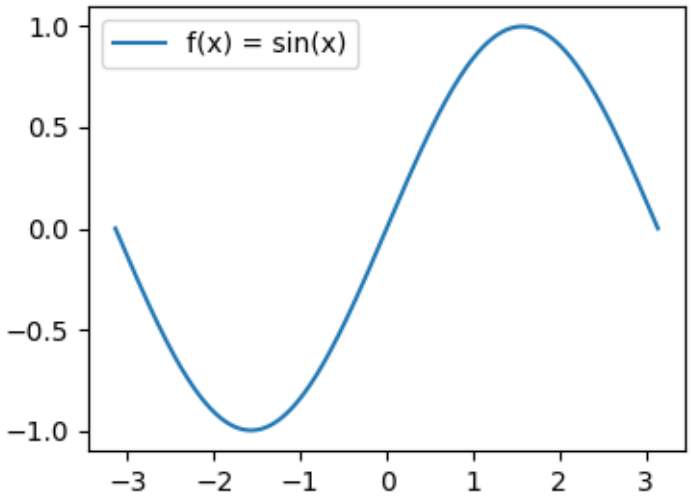

Abb. 4.1: Die Sinusfunktion.

Lernen eines Models mit 2 Fuzzy Sets und 1 Ablauf

Anfangend lasse ich das Model einen Lauf durchführen. Wobei hier wichtig zu erwähnen ist, dass eine Iteration und ein Lauf nicht das selbe ist. Iteration im stochastischen Verfahren bedeutet, dass ein Element aus der Datenmenge verarbeitet wurde und ein Lauf - dass alle Elemente erarbeitet sind. Also heißt es für unsere Laufzeit, dass das Model bereits nach einem Lauf 400 Iterationen ausgeführt hat. Die 400 Epochen haben ca. 1.16s gebraucht, wenn wir MF-Typ 0 verwenden. Bei dem MF-Typ 2 werden etwa 0.3s gespart - ca. 0.86s Lerndauer. Die Ergebnisse aus beiden Lerngänge werden unten in zwei Grafiken (4.2 und 4.3) zuerst und dann Daten in einer Tabelle gegeben.

Aus der beiden Abbildungen 4.2 und 4.3 ist es zu bestimmen, dass schon nach einem Lauf große Änderungen zu erkennen sind. In den Abbildungen sind die obersten Grafiken besonders wichtig. Jedoch aus der Abbildung 4.3 ist in der Grafik ein Fehler zu erkennen. Da hat die Funktion im Bereich zwischen -3.2 und -1.2 den 0-Wert. Dieses Fehler scheint weiter in den folgenden Testfällen aufzutauchen, aber verschwindet bei den Mini-Batch- und Batch-Lernverfahren. Interessanterweise erhalten wir zwei unterschiedlichen Endergebnisse, obwohl beide Zu-

1 Input 2 Sets 400 Epochs Stochastic Gradient Descent two equations mf.png

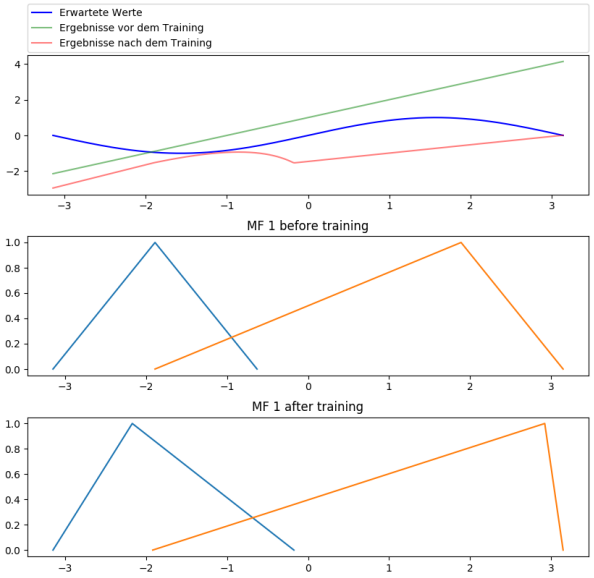

Abb. 4.2: Zwei Fuzzy-Sets, 400 Iterationen, MF-Typ 0

1 Input 2 Sets 400 Epochs Stochastic Gradient Descent one equation mf.png

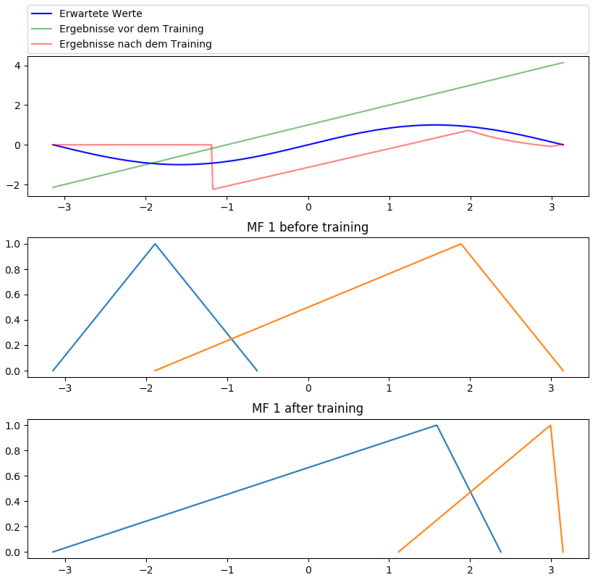

Abb. 4.3: Zwei Fuzzy-Sets, 400 Iterationen, MF-Typ 1

gehörigkeitsfunktionen die selbe Funktion berechnen. Man erkennt an den Grafiken weiter, dass die Fuzzy-Sets anders gestalltet sind.

In der Tabelle 4.1 unten ist die Fehlerrate und die Laufzeit abzulesen.

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 400 Epochs Stochastic Gradient Descent two equations mf	1.1651129310000004s	1.8500861	Stochastic Gradient Descent	two equations mf
sinus 1 Input 2 Sets 400 Epochs Stochastic Gradient Descent one equation mf	0.8567342689999995s	0.74594057	Stochastic Gradient Descent	one equation mf

Tabelle 4.1: Testergebnisse für Modelle mit 2 Fuzzy Sets und 400 Iterationen

Die interessantesten Spalten aus der Tabelle sind *Time* und *Error*. Die Zahlen deuten, dass MF-Typ 1 die bessere Vorgehensweise sein soll. Jedoch ist das wegen des Fehlers nicht ganz richtig.

Als letztes sind noch die Konklusionsfunktionen der beiden Lernabläufe - Konklusionsfunktionen 4.3 für das Modell mit 2 Sets, 400 Iterationen und MF-Typ 0 und 4.4 für das Modell mit einer Gleichung.

$$\begin{aligned} y_{mft0_1}(x) &= 0.64128985 + 1.14182867 \cdot x \\ y_{mft0_2}(x) &= -1.45790108 + 0.46766532 \cdot x \end{aligned} \tag{4.3}$$

$$\begin{aligned} y_{mft1_1}(x) &= -1.14182764 - 0.94000338 \cdot x \\ y_{mft1_2}(x) &= -2.1789615 + 0.35914132 \cdot x \end{aligned} \tag{4.4}$$

Aus der Konklusionsfunktionen kann man keine Rückschlüsse ziehen. Die Endkurven sind ähnlich, aber das ist kein Wunder, da beide Modelle versuchen, die selbe Funktion zu lernen.

Lernen eines Models mit 2 Fuzzy Sets und 10 Abläufe

Der nächste Test endet in 10 Läufe. Die Zwei Tests zeigen keine positiven Ergebnisse. Die Grafiken für die zwei MF-Typen sind sehr Unterschiedlich. Es werden insgesammt vier Tausend Iterationen pro Test durchgeführt, was etwa 7.3s und 6.9s für MF-Typ-0 und MF-Typ-1 entsprechend dauert. Die Abbildungen 4.4 und 4.5 stellen die Ergebnisse der Tests dar.

In der Abbildung 4.5 ist wieder der Fehler in der Berechnung zu bemerken. Der Fehler ergibt sich daraus, dass die Katete der gelbe Zugehörigkeitsfunktion zu lang ist. Was bei kleinen X-Werten dazu führt, dass der rechte Teil der Maximumoperation in der Gleichung 4.2 negativ wird.

1 Input 2 Sets 4000 Epochs Stochastic Gradient Descent two equations mf.png

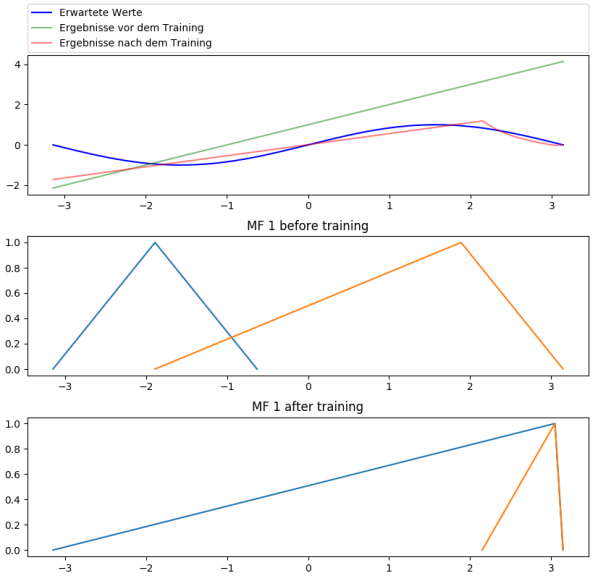

Abb. 4.4: Zwei Fuzzy-Sets, 4000 Iterationen, MF-Typ 0

1 Input 2 Sets 4000 Epochs Stochastic Gradient Descent one equation mf.png

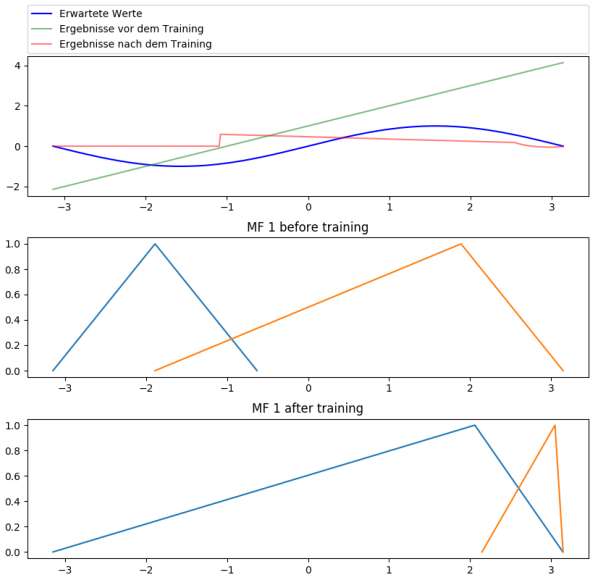

Abb. 4.5: Zwei Fuzzy-Sets, 4000 Iterationen, MF-Typ 1

Die Konklusionsfunktionen sind für beide Modelle gegeben (4.5 und

$$y_{mft1_1}(x) = 0.01042824 + 0.54973926 \cdot x \tag{4.5}$$
$$y_{mft1_2}(x) = -1.28822409 - 0.15157128 \cdot x$$

$$y_{mft2_1}(x) = 0.46553119 - 0.11269253 \cdot x \tag{4.6}$$
$$y_{mft2_2}(x) = -1.52831088 + 0.44669464 \cdot x$$

Im folgenden Fall gibt es wieder keine Ähnlichkeit zwischen den Konklusionsfunktionen der beiden Modelle.

Die Zeiten und Name als auch die Fehlerrate für beide Modell sind in der Tabelle 4.2 angegeben. Daraus können zwei Schlüsse gezogen werden.

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 4000 Epochs Stochastic Gradient Descent two equations mf	7.308665848s	0.21568511	Stochastic Gradient Descent	two equations mf
sinus 1 Input 2 Sets 4000 Epochs Stochastic Gradient Descent one equation mf	6.9275327529999995s	0.50803167	Stochastic Gradient Descent	one equation mf

Tabelle 4.2: Testergebnisse für 2 Sets und 4000 Iterationen

In der Tabelle können den Fehler und die Laufzeit ausgelesen werden. Anhand der Daten aus dieser Tabelle lässt sich sagen, dass die Berechnung für Modelle des Typs MF-1 kürzer ist, aber das Typ 0 Modell besser lernt.

Lernen eines Models mit 8 Fuzzy Sets und 10 Abläufe

Im folgenden Testfall ist zu untersuchen, wie die Laufzeit beieinflußt wird, wenn sich die Anzahl der Fuzzymengen erhöht. Im Folgenden Unterkapitel wird der Test mit **8** Fuzzy-Sets ausgeführt.

Die Ergebnissgrafiken sind zunächst gegeben (4.6 und 4.7).

An den Abbildugen erkennt man, dass sich die Funktion mit mehr Fuzzy-Mengen besser lernen lässt. Man sieht auch große Unterschiede in der Art der gelernten Fuzzymengen. Es ist zu bemerken, dass alle ab dem zweiten Fuzzyset in der ersten Abbildung in die zweite Hälfte des Wertebereichs vollgestopft werden. Während in der zweite Abbildung ab dem dritten Fuzzyset. Das ist nur deswegen geschehen, weil bei dem Lernen immer Einzelelemente aus der Datenmenge gezogen werden. Üblicherweise werden beim Lernen immer die betroffenen Parametern

1 Input 8 Sets 4000 Epochs Stochastic Gradient Descent two equations mf.png

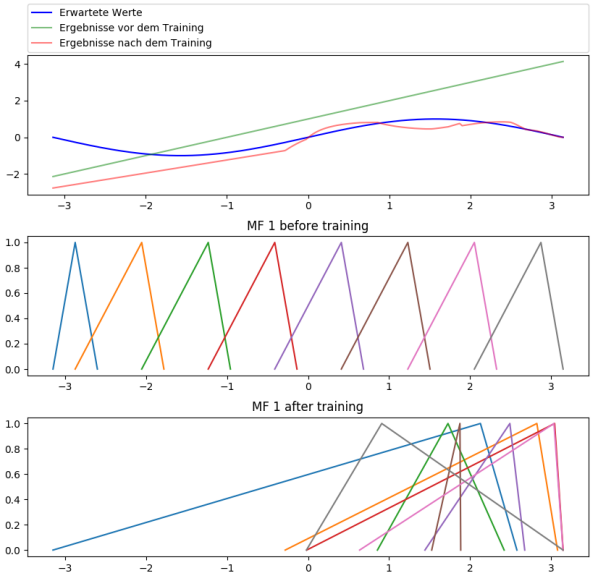

Abb. 4.6: Acht Fuzzy-Sets, 4000 Iterationen, MF-Typ 0

1 Input 8 Sets 4000 Epochs Stochastic Gradient Descent one equation mf.png

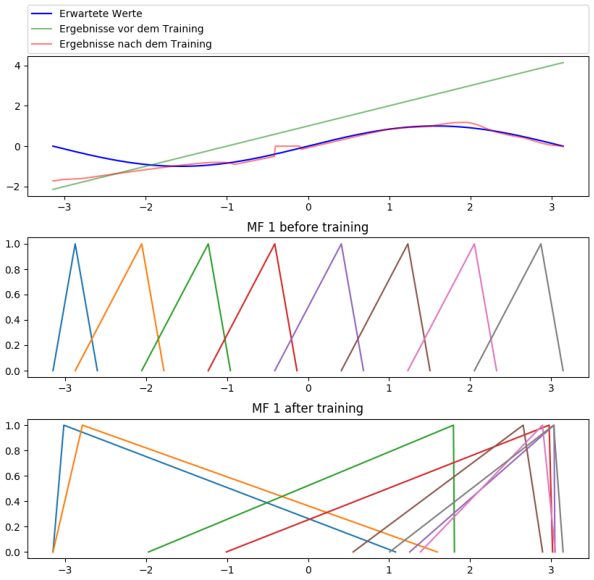

Abb. 4.7: Acht Fuzzy-Sets, 4000 Iterationen, MF-Typ 1

nach jeder Iteration angepasst. Also die erste Menge wird als erstes angesprochen und angepasst. Da der linke Parameter nicht geändert werden kann, bedeutet das, dass nur der Mittel- und rechten Grenzparameter nach rechts geschoben werden. Dies erklärt die Vollstopfung am rechten Rand des Wertebereichs.

Die Ergebnisse sind in Tabelle 4.3 abzulesen.

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 8 Sets 4000 Epochs Stochastic Gradient Descent two equations mf	17.413568215999998	0.80348945	Stochastic Gradient Descent	two equations mf
sinus 1 Input 8 Sets 4000 Epochs Stochastic Gradient Descent one equation mf	14.525632421000001	0.21442673	Stochastic Gradient Descent	one equation mf

Tabelle 4.3: Testergebnisse für 8 Fuzzy-Sets und 4000 Iterationen

Aus der Tabelle lässt sich herauslesen, dass der zweite Test (siehe 4.7) deutlich schneller und akkurater lernt. Auf der Abbildung 4.7 ist eine Stuffe im Mittleren Bereich zu erkennen, wo sich der Fehler zeigt. Es ist zu lesen, dass das Typ 1 *3* Sekunden schneller ist. Man sieht auch, dass das Verfahren deutlich näher an dem Sollfunktion ist als Typ 0.

Ich gebe die Konklusionsfunktionen für MF-Typ 0 4.7 und MF-Typ 1 4.8 nur informative an. Daran können keine weitere Rückschlüsse genannt werden.

$$
\begin{aligned}
y_{mft1_1}(x) &= -0.5235794 + 0.71489089 \cdot x \\
y_{mft1_2}(x) &= 2.93676464 - 0.78074253 \cdot x \\
y_{mft1_3}(x) &= -3.65265173 + 1.63942728 \cdot x \\
y_{mft1_4}(x) &= 2.85619127 - 0.87280814 \cdot x \\
y_{mft1_5}(x) &= 0.1644323 + 0.55701768 \cdot x \\
y_{mft1_6}(x) &= -0.92605422 + 1.26660038 \cdot x \\
y_{mft1_7}(x) &= 0.26038533 - 0.12738553 \cdot x \\
y_{mft1_8}(x) &= 0.63168423 - 0.15357252 \cdot x
\end{aligned} \tag{4.7}
$$

$$
\begin{aligned}
y_{mft1_1}(x) &= 0.46676819 + 0.41699012 \cdot x \\
y_{mft1_2}(x) &= -0.17691342 + 0.80520989 \cdot x \\
y_{mft1_3}(x) &= -0.07739224 + 0.93785577 \cdot x \\
y_{mft1_4}(x) &= 1.28046489 - 0.94272644 \cdot x \\
y_{mft1_5}(x) &= -1.72202347 + 0.71823673 \cdot x \\
y_{mft1_6}(x) &= 1.398913 + 0.30636544 \cdot x \\
y_{mft1_7}(x) &= 0.06231506 - 0.17987376 \cdot x \\
y_{mft1_8}(x) &= -0.04088159 - 0.28006114 \cdot x
\end{aligned} \tag{4.8}
$$

4.1.1 Schlussfolgerung

Hier ist schwehr die beste Konfiguration fürs Lernen zu nennen. Ich würde sogar sagen, dass das Lernen mit dem stochastischen Verfahren ungeeignet ist. Im nächsten Kapitel wird das Mini-Batch-Verfahren ausgetestet.

Außerdem wird in den nächsten Kapiteln nur auf MF-Typ 0 Modelle konzentriert. Da MF-Typ 1 Modelle nur für symmetrische Fuzzy-Mengen geeignet sind.

4.2 Lernen der Sinusfunktion mit Mini-Batch Gradient Descent

In diesem Kapitel handelt es sich um die Mini-Batch Gradient Descent Verfahren. Das Verfahren wurde am Anfang dieses 4. Kapitels vorgestellt. Ziel der folgenden Untersuchungen ist es zu zeigen, welche die beste Konfiguration fürs Model ist. Außerdem werden im Fazit die beiden Verfahren (Mini-Batch und Stochastic) verglichen.

Lernen der Sinusfunktion mit 2 Fuzzy-Sets und 1 Ablauf

Begonnen werden soll mit dem Lernen der Funktion mit zwei Fuzzymengen und der Ablauf verläuft über einen Gang. Die Testbeschreibung beinhaltet nur den MF-Typ 0.

In der Abbildung 4.8 sieht man keine große Änderung in den Fuzzy-Sets als auch in der Ergebnissfunktion. Das ist jedoch kein Wunder, da es nur 5 Iterationen durchgeführt werden. Die Dauer dieses Tests entspricht 14 ms.

1 Input 2 Sets 5 Epochs Mini-Batch Gradient Descent two equations mf.png

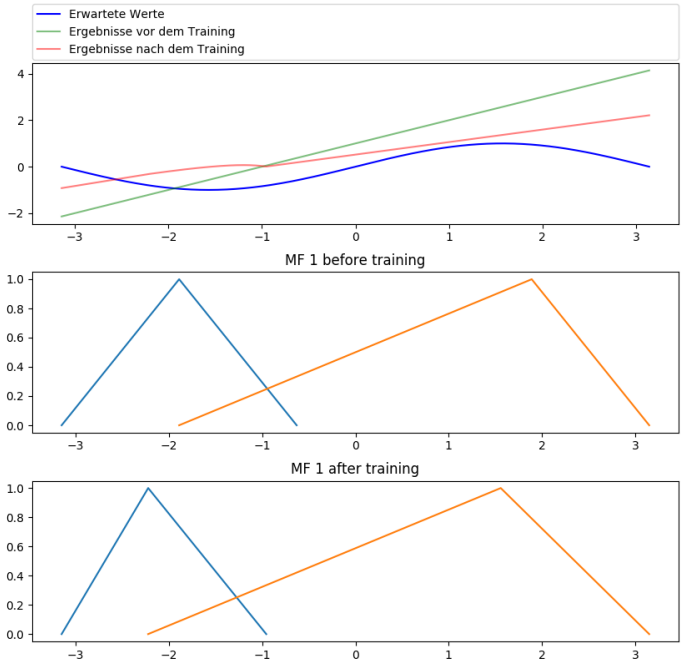

Abb. 4.8: 2 Fuzzy-Sets, 5 Iterationen, MF-Typ 0

Zur Abgleich habe ich die Ergebnisse aus dem vorrigen Unterkapitel und diesem Test in der Tabelle 4.4 angegben.

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 5 Epochs Mini-Batch Gradient Descent two equations mf	0.1438041160000001	0.70317644	Mini-Batch Gradient Descent	two equations mf
sinus 1 Input 2 Sets 400 Epochs Stochastic Gradient Descent two equations mf	1.1651129310000004s	1.8500861	Stochastic Gradient Descent	two equations mf

Tabelle 4.4: Testergebnisse für Mini-Batch und Stochastic

Die Fehlerrate ist wie erwartet sehr hoch, jedoch viel kleiner als der von Stochastischen Verfahren. Außerdem ist die Endfunktion sehr ähnlich wie aus der Vorrigenkapitel (siehe 4.2). Man erkennt aus den beiden Abbildungen, dass in Abbildung 4.2 die Funktion im Negativen Y-Bereich liegt.

Es ist eine schnellere Laufzeit als der stochastische Verfahren zu erkennen. Die kürze Lerndauer erklärt sich dadurch, da der Datensatz schneller abgearbeitet wird - in Fünf Schritte, im Vergleich zu Vierhundert. Es ist zu erwarten, dass nach 400 Iterationen das Modell noch besser (niedrigere Fehlerrate zu erweisen) sein soll. Im nächsten Kapitel wird diese Aussage untersucht.

Schließlich gebe ich die Konklusionsfunktionen für das Model in 4.9.

$$\begin{aligned} y_{mft1_1}(x) &= 1.12589365 + 0.65193717 \cdot x \\ y_{mft1_2}(x) &= 0.51933658 + 0.53854825 \cdot x \end{aligned} \tag{4.9}$$

Zwischen die Funktionen 4.9 und 4.3 ist in die Erste Gleichung zu erkennen, dass sich die Parametern ihren Platz getauscht haben. Ein weiterer Unterschied ist, dass es keine negativen Parametern gelernt wurden.

Lernen der Sinusfunktion mit 2 Fuzzy-Sets und 10 Ablauf

Als nächstes ist das Model mit 10 Abläufe zu betrachten. Die Frage, die in diesem Kapitel beantwortet werden soll, ist, ob das Lernen mit einem Bruchteil der Datenmenge wirklich besser ist. Deswegen vergleiche ich die Ergebnisse von Kapitel 4.1 mit den aus diesem.

Eine Ergebnissabbildung wird in 4.9 gezeichnet.

1 Input 2 Sets 50 Epochs Mini-Batch Gradient Descent two equations mf.png

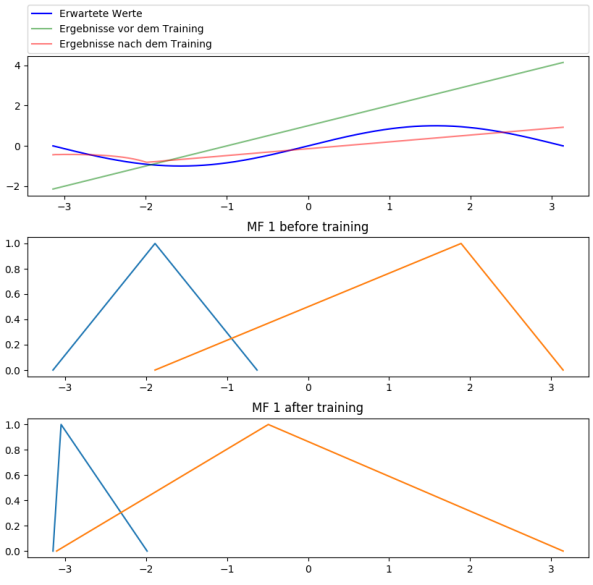

Abb. 4.9: 2 Fuzzy-Sets, 50 Iterationen, MF-Typ 0

Die Fuzzy-Mengen haben eine deutlich größere Änderung im Vergleich zu dem aus vorrigen Kapitel Modell unterlegen. Das Lernen hat 0.2s gedauert. Das ist

etwa 30 Mal schneller als der stochastische Verfahren. Die Daten für beide Modelle stehen mit Fehlerrate und Dauer in der Tabelle 4.5.

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 50 Epochs Mini-Batch Gradient Descent two equations mf	0.2727800499999997	0.1554767	Mini-Batch Gradient Descent	two equations mf
sinus 1 Input 2 Sets 4000 Epochs Stochastic Gradient Descent two equations mf	7.308665848s	0.21568511	Stochastic Gradient Descent	two equations mf

Tabelle 4.5: Erbenisvergleich zwischen dem Mini-Batch- und Stochastic-Modell

Man erzielt nicht nur ein schnelleres Lernen mit dem Mini-Batch Verfahren,sondern auch ein besseres. Die Fehlerrate ist von 0.22 um etwa 30% auf 0.16 abgestiegen.

Zum Schluss sind die Inferenzfunktionen 4.10 für das Modell gegeben.

$$\begin{aligned} y_{mft1_1}(x) &= 0.58986986 + 0.32787314 \cdot x \\ y_{mft1_2}(x) &= -0.14078197 + 0.3395195 \cdot x \end{aligned} \tag{4.10}$$

Lernen der Sinusfunktion mit 2 Fuzzy-Sets und 1000 Abläufe

In diesem Abschnitt wird das Ergebnis aus dem Test mit 1000 Abläufe vorgestellt. Das Model verfügt weiterhin über 2 Fuzzymengen und wird mit MF-Typ 0 gelernt.

Bei dieser Konfiguration lernt das Model mit nur einer Zugehörigkeitsgleichung auch sehr erfolgreich. Die Abbildungen der beiden Tests sind sehr ähnlich. Die Abbildungen 4.11 und 4.10 berichten das Ergebnis.

Das Endergebnis aus der Abbildung 4.11 scheint auf den ersten Blick besser zu sein. Der größte Unterschied liegt in den Fuzzymengen. Die gelernten Fuzzysets überlappen zur unterschiedlichen Stuffen. Die Mengen in der Abbildung 4.11 überschneiden sich deutlich mehr. Die Reichweite ist in der Abbildung 4.11 etwa 3 cm und auf der Abbildung 4.10 sind es ungefähr 2. Die zusätzlichen Daten, die in der Tabelle 4.6 zu finden sind, beantworten die Frage, welches Model besser ist.

1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent two equations mf.png

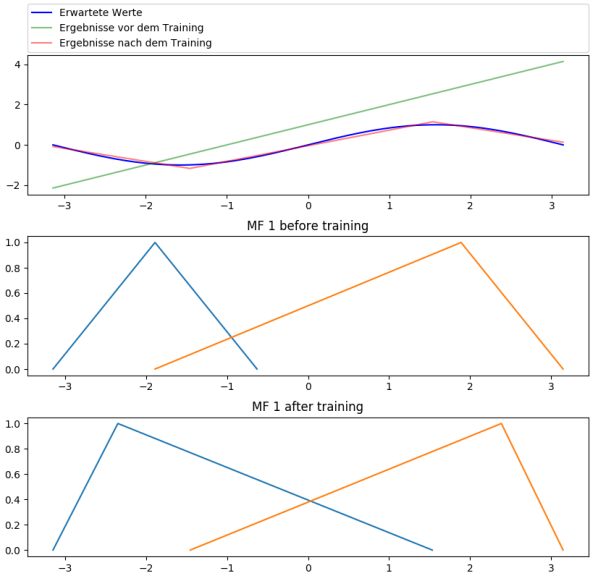

Abb. 4.10: 2 Fuzzy-Sets, 1000 Iterationen, MF-Typ 0

1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent one equation mf.png

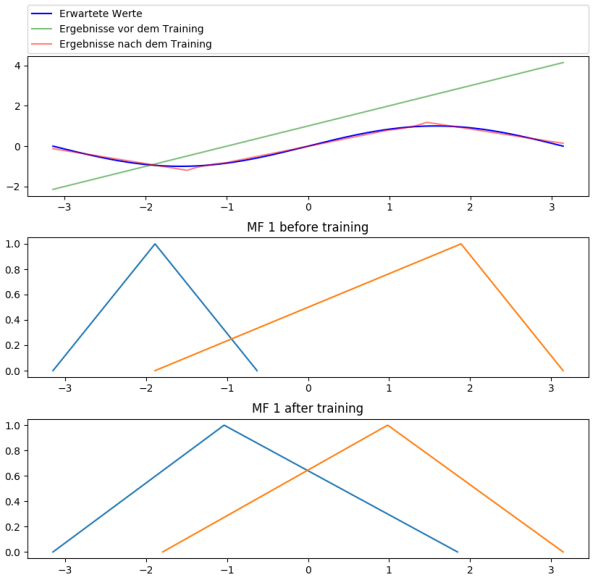

Abb. 4.11: 2 Fuzzy-Sets, 1000 Iterationen, MF-Typ 0

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent two equations mf	11.761141329s	0.006521431	Mini-Batch Gradient Descent	two equations mf
sinus 1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent one equation mf	10.697818779s	0.0048954873	Mini-Batch Gradient Descent	one equation mf

Tabelle 4.6: Testergebnisse für Modelle mit 2 Sets und 5000 Iterationen

Die Vermutung war richtig. Das zweite Model ist mit etwa 0,02 Einheiten besser und mit einer Sekunde schneller. Die Frage taucht jetzt auf, wo liegt der Unterschied zwischen die beiden Modellen? Ist es wegen des Unterschieds in der Fuzzymengen, oder sind die Inferenzfunktionen einfach unterschiedlich? Die Antwort wird erst klar, wenn die Gleichungen für die Konklusionen betrachtet werden. Analog gehören die ersten beiden Gleichung in 4.11 dem Modell mit zwei Gleichungen (MF-Typ 0) und entsprechend 4.12 dem Modell mit einer Gleichung (MF-Typ 1).

$$\begin{aligned} y_{mft1_1}(x) &= -2.10781751 - 0.64453965 \cdot x \\ y_{mft1_2}(x) &= 2.1163349 - 0.63109723 \cdot x \end{aligned} \tag{4.11}$$

$$\begin{aligned} y_{mft2_1}(x) &= -2.17036302 - 0.65073889 \cdot x \\ y_{mft2_2}(x) &= 2.09081218 - 0.62095912 \cdot x \end{aligned} \tag{4.12}$$

Die Antwort auf die Frage lautet, dass der Unterschied liegt in den Fuzzymengen. Die Gleichungen sind bis auf der zweiten Dezimalzahl identisch. Die Tatsache, dass die Werte überhaupt so ähnlich sind, ist ein Wunder.

Wenn man die Ergebnisse aus Unterkapitel 4.2 betrachtet, erkennt man wie schlecht das stochastische Verfahren lernt. Allein mit **10** Abläufe (4000 Iterationsschritte) bei dem Stochastischen-Test (siehe Tabelle 4.2) hat man fast die selbe Anzahl an Iterationsschritten (5000) bei dem Mini-Batch-Test (siehe 4.6) und trotzdem ist das Ergebniss **100** mal schlechter.

4.3 Lernen der Sinusfunktion mit Batch Gradient Descent

Weiterhin werden Modelle erstellt, die das Batch Gradient Descent Verfahren benutzen, um die Sinusfunktion zu erlernen. Jedes Testmodell wird 1000 Abläufe, oder 1000 Iterationen (siehe 4), laufen gelassen, aber jedes unterscheidet sich vom nächsten in der Anzahl der Fuzzy-Mengen. Es werden zwei Tests durchgeführt,

um zu bestimmen, welche ist die optimale, bzw. ausreichende, Anzahl an Fuzzy-Mengen. Die drei Tests werden entsprechen mit 2 und 3 Fuzzy-Sets trainiert. Außerdem es wird einen Vergleich mit Mini-Batch gezogen. Es ist interessant zu prüfen, ob die beiden Verfahren auch unterschiedliche Ergebnisse erzielen würden.

4.3.1 Lernen mit 2 Fuzzy-Sets

Das erste Testmodelle wird mit 2 Fuzzy-Mengen initialisiert und der Test wird 1000 Abläufe durchgeführt. Das Ergebnis wird in der Abbildung 4.12 und in der Tabelle 4.7 beschrieben.

1 Input 2 Sets 1000 Epochs Batch Gradient Descent two equations mf.png

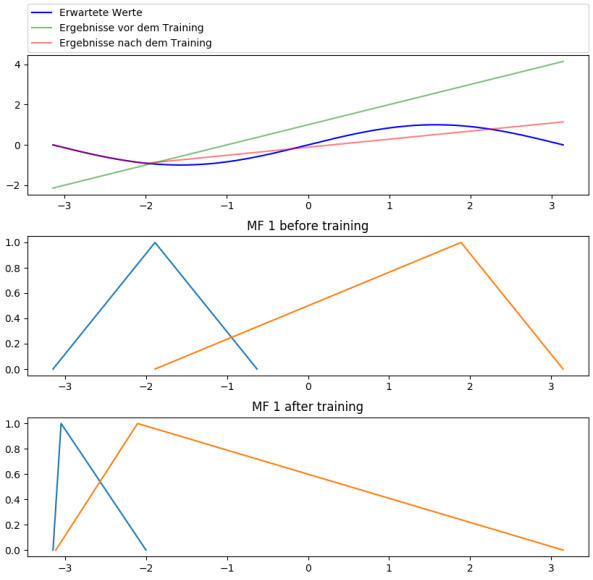

Abb. 4.12: Batch Modell mit 2 Fuzzy-Sets, 1000 Iterationen und MF-Typ 0

Der Abbildung 4.12 nach wird in dem Anfangsbereich zwischen den X-Werteb -3.2 und -2.0 scheint die Kurve gut angepasst zu werden. Im späteren Bereich (-2.0 bis 3.2) ist keine Kurve mehr, sondern eine Gerade. Das liegt daran, dass die Fuzzymengen nur zum kleinen Teil überlappen, zwar nur zwischen -3.2 und 2.0, darum auch die anliegende Kurve. Wenn dieses Ergebniss mit dem aus letzen Unterkapitel (siehe 4.2) verglichen wird, sieht man den Unterschied in den Fuzzymengen. Hier schon erkennt man, dass das Mini-Batch-Verfahren geeigneter ist als das Batch. Die Begründung dafür ist, dass mit dem Mini-Batch-Verfahren nach 1000 Abläufe ein gut tranierter Modell erhalten wird. Die Tabelle verschafft uns weitere Information darüber, wie gut das Modell ist.

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 1000 Epochs Batch Gradient Descent two equations mf	2.1207925870000004	0.13913395	Batch Gradient Descent	two equations mf
sinus 1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent two equations mf	11.761141329s	0.006521431	Mini-Batch Gradient Descent	two equations mf

Tabelle 4.7: Testergebnisse für die Modelle mit Batch- und Mini-Batch-Verfahren

Wenn die Zeilen aus der Tabelle 4.7 verglichen wird, erkennt man, dass das Mini-Batch Verfahren besser ist. In der Tabelle kann man lesen, dass es zwei unterschiedliche Werte für die Iterationen der Modelle figurieren. Dies entsteht aus den Unterschieden in der beiden Verfahren (Mini-Batch und Batch). In beiden Testfällen wird 1000 Abläufe durchgeführt, was bei dem Mini-Batch-Modell 5000 Iterationen sind, da einen Ablauf aus fünf Iterationen besteht. Daraus ergibt sich eine natürliche Verzögerung in der Berechnung(dem Lernen). Was aber nicht bestreiten kann ist die Fehlerrate. Hier erweist das Mini-Batch-Modell eine Verbesserung um zwei Stellen nach der Komma.

4.3.2 Lernen mit 3 Fuzzy-Sets

In dieser Unterkapitel wird ein Test mit 3 Fuzzy-Mengen durchgeführt. Das Modell besitzt die selben Eigenschaften wie das aus dem letzen Unterkapitel(siehe 4.3.1). Analog zur Unterkapitel 4.3.1 wird zuerst eine Abbildung und danach eine Tabelle analysier, jedoch wird jetzt einen Vergleich zwischen dem Modell aus 4.12 und dem Modell in der Abbildung 4.13 gezogen. Ziel hier ist es zu prüfen, ob eine Erhöhung der Fuzzy-Sets zum besseren Ergebniss führt. Die Abbildung 4.13 zeigt das Endergebniss.

Die Abbildung 4.13 zeigt eine deutliche Verbesserung in dem, wie die Sinusfunktion gelernt wird. Es kann sogar gesagt werden, dass die Funktion sehr gut gelernt ist. Hier ist es wichtig darauf zu weisen, dass alle drei Fuzzy Mengen zu einem gewissen Teil überlappen. Daraus entsteht auch die fast perfekte Kurve, die die gesuchte Funktion überdeckt. Weiterhin wird die Tabelle 4.8 untersucht

1 Input 3 Sets 1000 Epochs Batch Gradient Descent two equations mf.png

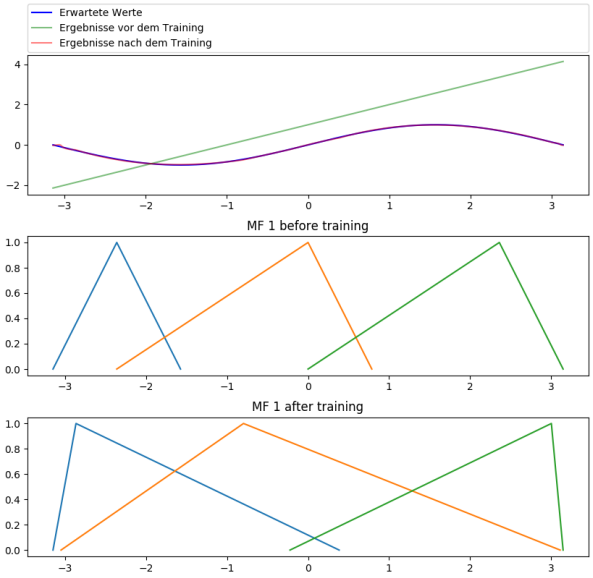

Abb. 4.13: Batch Modell mit 3 Fuzzy-Sets, 1000 Iterationen und MF-Typ 0

Type	Time	Error	Gradient Type	MF Type
sinus 1 Input 2 Sets 1000 Epochs	2.1207925870000004	0.13913395	Batch Gradient Descent	two equations mf
sinus 1 Input 3 Sets 1000 Epochs	2.528284161	0.00021747834	Batch Gradient Descent	two equations mf
sinus 1 Input 2 Sets 5000 Epochs	11.761141329s	0.006521431	Mini-Batch Gradient Descent	two equations mf
sinus 1 Input 3 Sets 5000 Epochs	12.479160914	0.00017558844	Mini-Batch Gradient Descent	two equations mf

Tabelle 4.8: Testergebnisse für die Modelle mit zwei und drei Fuzzy-Mengen und Mini-Batch- und Batch-Verfahren

Die Ausage, die sich aus dem Vergleich der Abbildungen 4.12 und 4.13 herausgestellt hat, wird durch die Tabelle 4.8 gestärkt. Was aber wichtiger zu betonen, ist die Verbesserung im Vergleich zu dem Mini-Batch-Verfahren Modell aus Unterkapitel 4.3.1. Das Endergebniss des drei Set Modells hat sich um etwa einen Dreißigstel verbessert. Außerdem trainiert das Modell schneller und zwar um etwa 5 Mal. Wenn man das Ergebniss in den Zeilen zwei und vier von der Tabelle 4.8 betrachtet, sieht man, dass das Mini-Batch-Verfahren etwas besser lernt, aber langsamer wegen der erhöhten Iterationen ist.

4.4 Lernen der Parabelfunktion

Die Struktur dieser Kapitel ähnelt dem aus Vorherigen. Im Laufe der Ausarbeitung bin ich einer sehr interessanten Frage gestoßen. Ich wollte untersuchen, ob das Vorzeichen der Trainingsdaten einen Einfluß auf das Lernen hat. Um das zu überprüfen, habe ich die Parabelfunktion verändert, so dass die positiven X-Werten einnimmt. In einem der Abschnitte unterscheide ich zwischen die beiden Funktionen.

Die eigentlichen Funktionen, die gelernt werden müssen, sind in zwei Abbildungen 4.14 und 4.15 zu sehen:

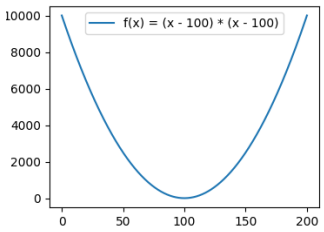

Abb. 4.14: positive Funktion.

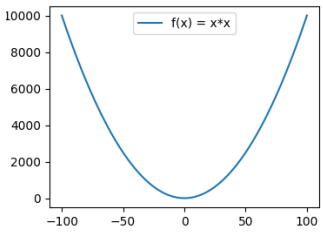

Abb. 4.15: quadratische Funktion.

Auf der ersten Grafik erkennt man, dass die X-Werten nur Positive sind. Ich werde jetzt einige Ergebnisse vorstellen und zeigen, ob die ANFIS-Modelle die Funktionen gleich lernen.

4.4.1 Lernen der Parabelfunktion mit Mini-Batch Gradient Descent

Zuerst betrachten wir zwei Modelle, die das Lernen über 10 Abläufe, bzw. 50 Iterationen, durchführen. Die Ergebnisse sind in den Abbildungen 4.16 und 4.17 zu lesen.

In diesem Fall ist die Anzahl der Durchläufe viel zu klein, um wesentliche Ergebnisse geliefert zu werden. Man erkennt, dass in der Grafik 4.16 die zwei Enden der Kurve nach oben geschoben sind, während die Mitte auf dem 0-Punkt liegt. In der Grafik 4.17 erkennt man keine Kurve wirklich, sondern eine Gerade, welche die selbe wie der Ursprungsgerade ist, aber versetzt. Es werden die nummerische Unterschiede der beiden Modellen in der Tabelle 4.9 betrachtet:

Type	Time	Error	Gradient Type	MF Type
parabola_1000	0.27263559800000037	17675046.0	Mini-Batch Gradient Descent	two equations mf
parabola_positive	0.26520074800000026	16615864.0	Mini-Batch Gradient Descent	two equations mf

Tabelle 4.9: Testergebnisse für die Modelle

Aus der Tebelle 4.9 erkennt man, dass das “positive” Modell sowohl schneller als auch “richtiger” ist. Die Bilder würden deuten, dass das “normale” Model besser

1 Input 2 Sets 50 Epochs Mini-Batch Gradient Descent two equations mf.png

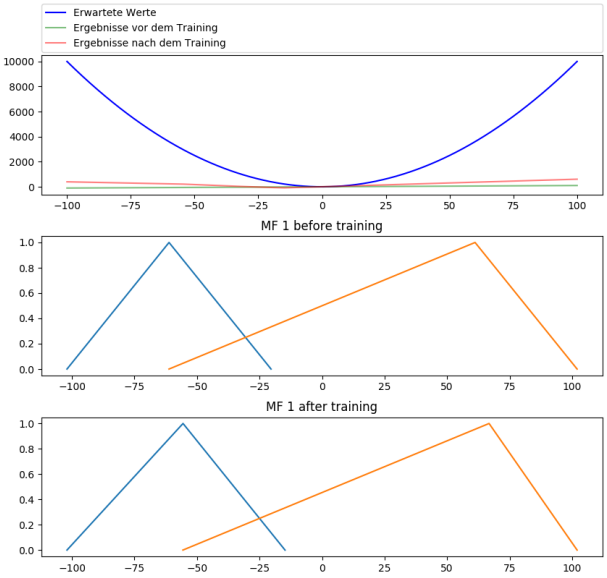

Abb. 4.16: 2 Fuzzy-Sets, 50 Iterationen, MF-Typ 0

1 Input 2 Sets 50 Epochs Mini-Batch Gradient Descent two equations mf.png

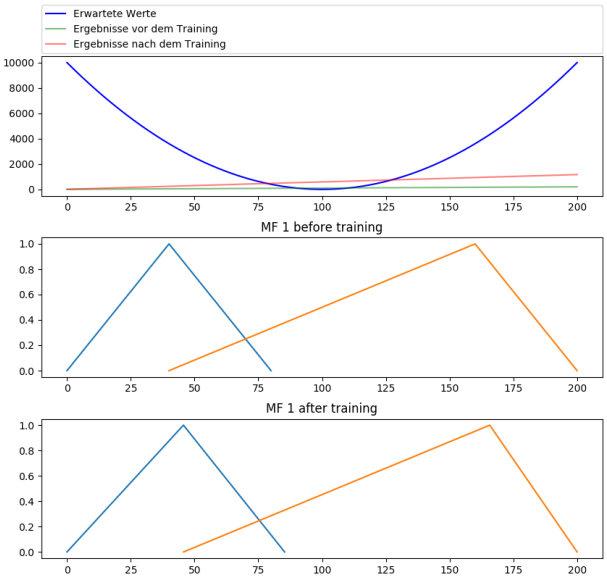

Abb. 4.17: 2 Fuzzy-Sets, 50 Iterationen, MF-Typ 0 mit nur Positiven X-Werte

ist, aber die nummerischen Daten sagen, dass das andere Modelle das bessere wäre. Das nächste Beipsiel beantwortet die Frage, welche Aufgabe besser gelernt werden kann.

Als nächstes werden wir die zwei Modellen 1000 Mal laufen lassen, das würde heißen das es insgesammt 5000 Iterationen durchgeführt werden. Die weiteren Konfigurationen verbleiben gleich. Zuerst werden die Abbildung 4.18 und 4.19 aus den beiden Tests gegeben:

1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent two equations mf.png

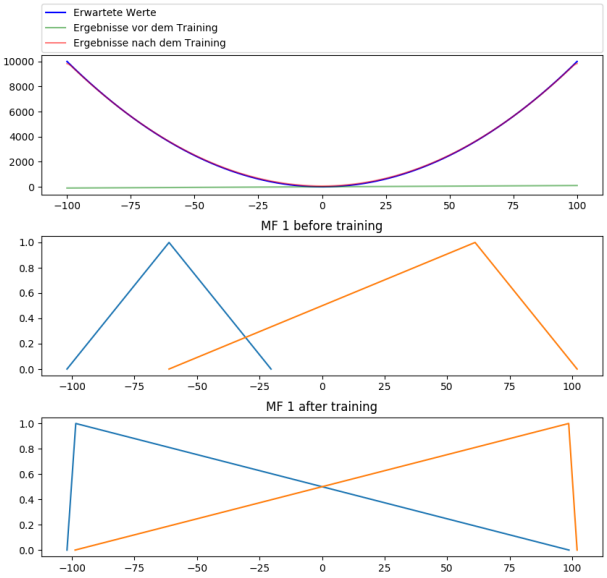

Abb. 4.18: 2 Fuzzy-Sets, 5000 Iterationen, MF-Typ 0

Aus den Abbildungen ist zu lesen, dass das Modell (siehe 4.18) mit positiven und negativen Trainingsdaten schneller den Optimalzustand erreicht als das positve Modell. Außerdem nach 5000 Iterationen ist das Netz sehr gut trainiert. Es ist zu erwarten, dass die nummerische Daten auch für das erste Beispiel sprechen. Die Tabelle 4.10 beinhaltet die nummerischen Daten.

Type	Time	Error	Gradient Type	MF Type
parabola_1000	12.538009995	1682.427	Mini-Batch Gradient Descent	two equations mf
parabola_positive	12.474743275000002	5956089.5	Mini-Batch Gradient Descent	two equations mf

Tabelle 4.10: Testergebnisee von dem Training der Funktionen “parabola” und parabola‘*positive*”

Wie auch schon aus den Grafiken ersichtlich geworden ist, eignet sich das gemischte Modell besser zum lernen. Auch die nummerischen Daten sprechen dafür,

1 Input 2 Sets 5000 Epochs Mini-Batch Gradient Descent two equations mf.png

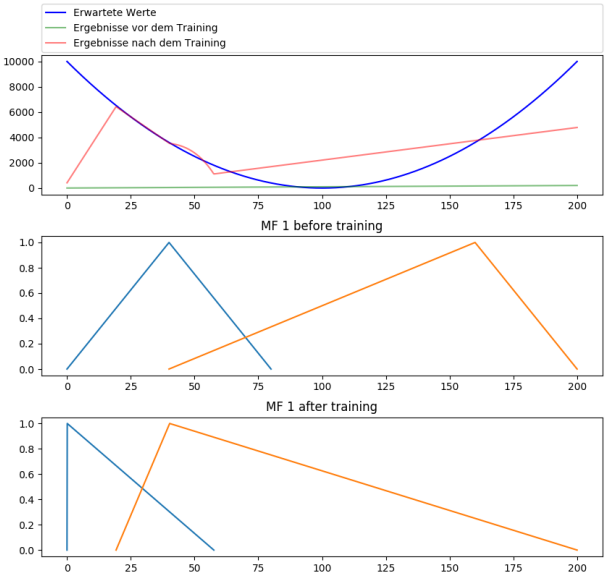

Abb. 4.19: 2 Fuzzy-Sets, 5000 Iterationen, MF-Typ 0 mit nur Positiven X-Werte

dass dies der Fall ist. Es besteht ein riesiger Unterschied in den Fehlerraten der beiden Testfälle.

4.5 Fazit

Die beschriebenen Testfälle wurden explizit ausgewählt, mit dem Ziel bestimmte Eigenschaften nachzuweisen. Der durchgeführten Tests zofolge kann beschlossen werden, dass Modelle mit dem MF-Typ 0 (siehe 4.1) in meinem Fall geeigneter sind. Das wurde festges aus dem Grund, weil die Berechnung der Zugehörigkeit mit MF-Typ 1 nur für symmetrische Fuzzymengen geeignet ist und der Fehler in den Tests (siehe z.B. Abbildungen 4.5 und 4.3) zu erkennen ist. Weiterhin sollte genannt werden, dass der MF-Typ 1 etwas schneller ist, weil er nur eine Berechnung, im Vergleich zu MF-Typ 0, der in zwei Schritten berechnet wird, erfordert.

Es ist sehr wichtig zu erwähnen, dass die Vergrößerung um eine Zusätzliche Menge etwa zwischen 1,2 und 1,3 Mal mehr Laufzeit erfordert. Während einer Verdopplung in Iterationen etwa die doppelte Laufzeit erfordert, was auch zu erwarten ist. Die Abbildung 4.20 zeigt diese Beziehung.

Weiterhin ist es sehr wichtig zu betonen, dass das Lernen mit Mini-Batch und Batch Gradient Descent die bessere Option im vergleich zum Stochastic-Verfahren sind. Einer der Gründe dafür ist, weil es eine bessere Fehlerrate in kleineren Laufzeiten erzielt wird. Diese Behauptung wird durch die nächsten zwei Abbildungen 4.21 und 4.22.

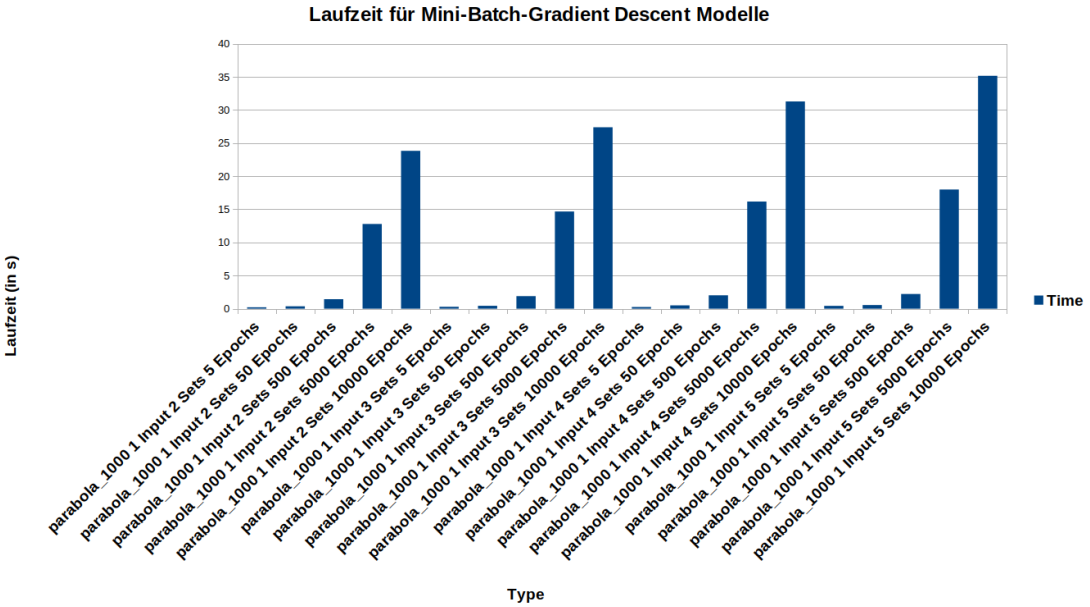

Abb. 4.20: Laufzeit für Modelle beim Erlernen der Parabel Funktion

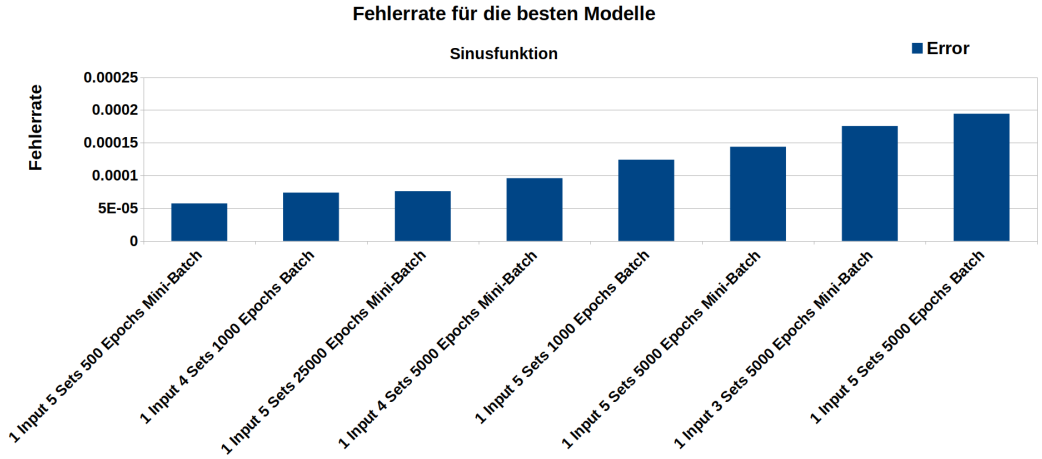

Abb. 4.21: Die besten Modelle beim Lernen der Sinusfunktion

Das weitere, was aus den Abbildung 4.21 und 4.22 zu entnehmen ist, ist, dass der Stochastische Verfahren in keinen der beiden Abbildungen figuriert. Das schließt dieses Verfahren zum Lernen der beiden Funktionen aus.

Aus den beiden Abbildungen 4.21 und 4.22 kommt man zu der Schlussfolgerung, dass das Mini-Batch-Verfahren bei dem Lernen von beiden Funktionen besser ist.

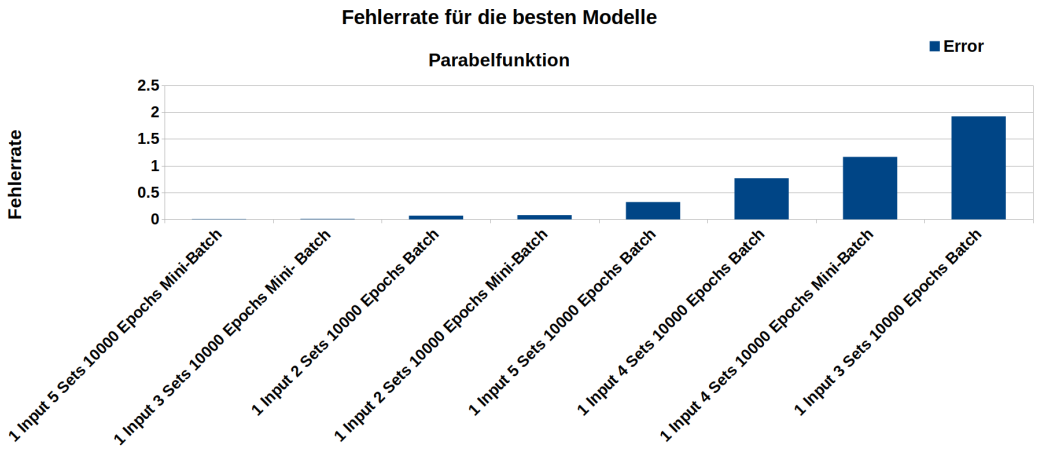

Abb. 4.22: Laufzeit für Modelle beim Erlernen der Parabel Funktion

Literaturverzeichnis

AM01. Andrea Tettamanzi und Marco Tomassini: *Soft Computing: Integrating Evolutionary, Neural, and Fuzzy Systems*. Springer-Verlag Berlin Heidelberg, 2001.

AR. Andreas Nrnberger, Detlef Nauck und Rudolf Kruse: *Neuro-Fuzzy Control Based on the NEFCON-Model Under MATLAB/SIMULINK*.
http://wwwiti.cs.uni-magdeburg.de/~nuernb/wsc2/.

Han98. Hans-Heinrich Bothe: *Neuro-Fuzzy-Methoden*. Springer-Verlag Berlin Heidelberg, 1998.

Jan93. Jang, Jyh-Shing Roger: *ANFIS: Adaptive-Network-Based Fuzzy Inference System*. Technischer Bericht, University of California, 1993.

JCE97. Jyh-Shing Roger Jang, Chuen-Tsai Sun und Eiji Mizutani: *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, 1997.

OLBI98. Okyay Kaynak, Lotfi A. Zadeh, Burhan Trk?en und Imre J. Rudas: *Computational Intelligence: Soft Computing and Fuzzy-Neuro Integration with Applications*. Springer-Verlag Berlin Heidelberg, 1998.

RCC+15. Rudolf Kruse, Christian Borgelt, Christian Braune, Frank Klawonn, Christian Moewes und Matthias Steinbrecher: *Computational Intelligence: Eine methodische Einfhrung in Knstliche Neuronale Netze, Evolution Algorithmen, Fuzzy-Systeme und Bayes-Netze*. Springer Fachmedien Wiesbaden, 2015.

uP. Prof. Dr. Oliver von Bohlen und Halbach, Leonie Seng und: *Zellen: spezialisierte Arbeiter des Gehirns*.
https://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/zellen-spezialisierte-arbeiter-des-gehirns.

Wik. Wikipedia: *Nervenzelle*.
https://de.wikipedia.org/wiki/Nervenzelle.