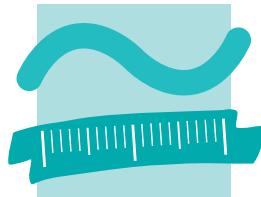


Modellbasierter Entwurf

---

# Neuronale Netze

---



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN  
University of Applied Sciences  
Fachbereich VI  
Technische Informatik

Teilnehmer	Matrikelnummer
Bryan Scheffner	888777
Toni Reichel	897991

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Aufgabenstellung . . . . .	1
1.2. Motivation neuronaler Netze . . . . .	1
1.3. Das menschliche Nervensystem . . . . .	2
1.4. Neuronale Netze und ihre Bedeutung . . . . .	4
<b>2. Aufbau des neuronalen Netzes</b>	<b>7</b>
<b>3. Erzeugung der Eingangsbilder</b>	<b>10</b>
<b>4. Gewichtsmatrizen</b>	<b>15</b>
4.1. Erzeugung der Gewichtsmatrizen . . . . .	16
4.2. Additive Überlagerung . . . . .	19
4.3. Multiplikative Überlagerung Typ 1 . . . . .	20
4.4. Multiplikative Überlagerung Typ 2 . . . . .	21
4.5. Additive und Multiplikative Überlagerung . . . . .	22
4.6. Spezielle Überlagerung . . . . .	23
<b>5. Das Matlab Tool</b>	<b>24</b>
5.1. Modulübersicht . . . . .	24
5.2. Erläuterung der Parameter . . . . .	27
<b>6. Auswertung</b>	<b>28</b>
6.1. AddMul Gewichtsmatrix . . . . .	29
6.1.1. H-Balken . . . . .	29
6.1.2. V-Balken . . . . .	32
6.1.3. Kreuz . . . . .	35
6.2. Special Gewichtsmatrix . . . . .	38
6.2.1. H-Balken . . . . .	38
6.2.2. V-Balken . . . . .	41
6.2.3. Kreuz . . . . .	44
<b>7. Fazit</b>	<b>47</b>
<b>A. Appendix</b>	<b>49</b>
A.1. GetPixelFeatureMatrix.m . . . . .	49
A.2. GetInputFeatureMatrix.m . . . . .	50
A.3. GetGaussWeights.m . . . . .	51

A.4. main . . . . .	53
A.5. SettingFileAddMul . . . . .	56
A.6. SettingFileSpecial . . . . .	59
A.7. GetNeuronOutput . . . . .	62
A.8. GetFeatureOfMatrix . . . . .	63
A.9. ConvMatrixToColumn . . . . .	63
A.10. GaussNormFunction . . . . .	64
A.11. SigmoidFunction . . . . .	68

# **1. Einleitung**

## **1.1. Aufgabenstellung**

In dieser Arbeit soll ein einfaches neuronales Netz aufgebaut werden, welches horizontale und vertikale Balken sowie ein Kreuz erkennen kann. Dafür werden die zentralen Themen sein, eine geeignete Gewichtsmatrix und eine passende Struktur für das Netz zu finden. Dies geschieht als Vorarbeit für eine spätere Hardwareimplementierung. In diesem Zuge sollen Simulationen durchgeführt werden, die die Funktion des neuronalen Netzes absichern. Darüber hinaus bietet die Simulation uns die Möglichkeit, empirisch eine optimale Konfiguration zu finden. Das Ziel dieser Arbeit ist es, das Prinzip hinter neuronalen Netzen besser zu verstehen und nicht bereits erforschte Netztopologie zu übernehmen. Für diese Arbeit sind wir wie folgt vorgegangen:

1. Allgemeine Informationssammlung zu neuronalen Netzen
2. Grundüberlegung der Gewichtung der einzelnen Pixel
3. Aufbau des Netzes
4. Erzeugung von Eingangsbildern
5. Auswertung der verschiedenen gefundenen Möglichkeiten

Alle Ergebnisse dieser Arbeit können mit dem von uns geschriebenen Matlab-Skripten nachvollzogen werden. Mithilfe dieser Skripte haben wir viele Simulationen durchgeführt und Erkenntnisse für das bessere Verständnis neuronaler Netze gewonnen.

## **1.2. Motivation neuronaler Netze**

Heutzutage findet man in nahezu allen Bereichen des Lebens Programme vor, sei es in Zahnbürsten, Autos oder in Robotern am Fließband. Mithilfe von Programmen sollen also sich wiederholende Aufgaben vereinfacht oder automatisiert werden. Dafür werden die Programme mit prozeduralen oder objektorientierten Methoden entwickelt. Bevor mit der Programmierung angefangen werden kann, muss zuerst ein Modell des Problems entworfen werden. Bei einfachen Anwendungen, wie zum Beispiel einer Zahnbürste ist es recht einfach. Möchte man allerdings ein Modell für eine Mustererkennung realisieren, stößt man mit herkömmlichen Methoden der Programmierung schnell an die Grenzen des Sinnvollem. Da sich besonders komplexe Zusammenhänge mittels der klassischen Programmierung zwar lösen lassen, aber diese Lösungen häufig sehr ressourcenintensiv sind bieten für solche Anwendungen neuronale Netze Vorteile. Neuronale Netze sollen die Vorteile des menschlichen Denkens und Verhaltens emulieren. Denn

das menschliche Denken und Verhalten unterscheidet sich sehr stark von dem einer Maschine. Bei einfachen Vorgängen wie dem Laufen, Sprechen oder anderen Körperbewegungen wie auch dem Sehen, Hören und Tasten laufen parallel und stellen uns meist vor keine großen Hürden. Des Weiteren kann eine Maschine in der Regel nicht kreativ sein, Lösungen interpolieren, aus Fehlern lernen oder gar phantasievoll agieren.

Bei der klassischen Programmierung spricht man auch von einer symbolischen Informationsverarbeitung. Alle Schritte sind dem Programmierer bewusst und er kann genau sagen, wie sich sein Programm verhalten wird. Vorausgesetzt er hat vorab ein präzises Modell erarbeitet, lässt sich sein Programm anhand des Modells überprüfen. Dem Gegenüber steht die Programmierung mit neuronalen Netzen. Bei neuronalen Netzen spricht man von einer sub-symbolischen Informationsverarbeitung. Der Programmierer des neuronalen Netzwerkes kann trotz eines genauen Modells nicht 100% sicher sagen, wie sich das neuronale Netzwerk verhalten wird. Erst wenn das neuronale Netzwerk konfiguriert wurde, ist es möglich genaueres Verhalten des neuronalen Netzes abzuschätzen. Die sub-symbolische Programmierung ist immer dann einzusetzen, wenn es schwer oder gar unmöglich ist die 'Realwelt' genau genug als Modell abzubilden.

Die Grundidee der sub-symbolischen Informationsverarbeitung ist das Auflösen der Symbole zur Beschreibung der Anwendungswelt in Mikrostrukturen auf der Basis primitiver Verarbeitungseinheiten (simulierter Neuronen). [2, S. 9]

Um besser zu verstehen, wie Nervenzellen (Neuronen) funktionieren, handelt der nächste Abschnitt über das menschliche Nervensystem.

### **1.3. Das menschliche Nervensystem**

Das menschliche Gehirn besteht aus etwa 86 Milliarden Nervenzellen, auch Neuronen genannt, die zum Zentralnervensystem miteinander verschaltet sind (im folgendem Nervensystem).[1] Nervenzellen im Gehirn sind hochspezialisierte Zellen, die sich im Gegensatz zu einfacheren Zellen nicht teilbar sind. Das bedeutet, dass sich das Nervensystem nicht über Zellteilung regenerieren kann. Allerdings schafft es das Gehirn durch die Verknüpfung anderer Nervenzellen den Defekt teilweise oder komplett auszugleichen. Jede Nervenzelle kann mit bis zu 10.000 anderen Nervenzellen in Verbindung stehen.[4]

Jede Nervenzelle ist im Grundaufbau gleich, kann aber unterschiedliche Formen und Größen haben. Das liegt unter anderem daran, dass sich Nervenzellen weiter spezialisieren.[4] So können die einen Nervenzellen für motorische und die anderen für sensorische Aufgaben bestimmt sein. Dadurch erreichen Nervenzellen unterschiedliche Geschwindigkeiten, wie sie Informationen

an andere Nervenzellen weitergeben.[1] Der Informationsfluss ist dabei immer in die gleiche Richtung.[4] In Abbildung 1 ist der Aufbau einer Nervenzelle dargestellt.

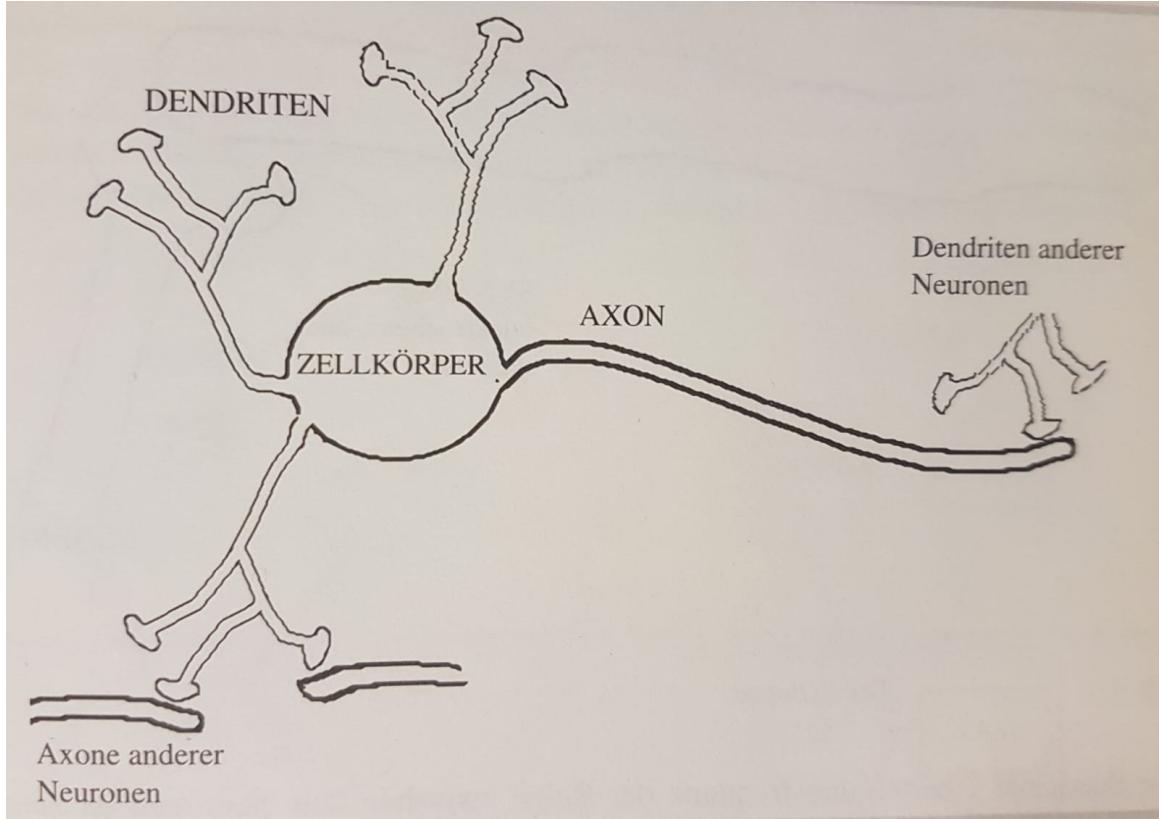


Abbildung 1: Aufbau Neuron [3, S. 11]

Der hier stark vereinfachte Aufbau einer Nervenzelle besteht aus dem Zellkörper, den Dendriten, dem Axon und den Synapsen. Die Synapsen liegen jeweils am Ende der Dendriten und des Axons. Der Zellkörper bildet die zentrale Einheit einer Nervenzelle und erhält seine Informationen über die Dendriten. Die Dendriten sind baumartig verzweigt und mit anderen Nervenzellen über Synapsen verbunden. Der Zellkörper bildet eine Summation über die Signale der vielen verschiedenen Dendriten. Dabei kann jedes Dendrit eine erregende oder hemmende Wirkung auf den Zellkörper haben. Bei ausreichender Stärke leitet der Zellkörper das Signal weiter. Wenn eine Erregungsweiterleitung stattfindet, wird das Signal über das Axon und die Synapsen an die nächste Nervenzelle weitergeleitet. Ein Axon hat einen Durchmesser von 0,002 - 0,01 Millimetern und kann bis zu einem Meter lang sein.[4] Umso dicker ein Axon ist, desto schneller findet auch die Weiterleitung statt. Jede Nervenzelle kann nur über ihre Synapsen mit anderen Nervenzellen kommunizieren. Dafür hat jede Nervenzelle bis zu 10.000, in manchen Extremfällen sogar mehr als 100.000 Synapsen.[1] Synapsen kommunizieren untereinander mit chemischen oder elektrischen Signalen. Wobei elektrische in chemische und chemische in elektrische Signale umgewandelt werden. In der Regel kommunizieren Synapsen über chemische Signale. Der Grund dafür ist, dass Synapsen meist keinen direkten Kontakt zueinander haben, sondern ein

kleiner Abstand von 20 bis 50 Nanometern bleibt.[1] Bei elektrischen Synapsen ist der Abstand so nah, dass über eine kleine Brücke (gap junction) kommuniziert wird. Dabei kann das Signal schneller weitergeleitet werden.[4]

Das Nervensystem ist ein großes Netz aus sehr vielen Nervenzellen, die unterschiedlich stark miteinander verbunden sind und unterschiedlich schnell Signale weiterleiten. Wenn eine Nervenzelle ein Signal auslöst, weil eine gewisse Schwelle überschritten wurde spricht man auch davon, dass das Aktionspotenzial erreicht wurde. Das Aktionspotenzial kann nur erreicht werden, wenn vorher genügend vorgesetzte Nervenzellen ein Signal gesendet haben. Der Anfang einer Verkettung von Nervenzellen kann zum Beispiel über sehen, schmecken, spüren oder ähnliches stattfinden. Wird ein Aktionspotenzial in den dafür verantwortlichen Nervenzellen erreicht, werden sie wieder Signale an andere Nervenzellen senden. Am Ende kann das Signal bei Nervenzellen ankommen, die für eine Bewegung verantwortlich sind, wie zum Beispiel eine Gehbewegung. Das ist ein sehr stark vereinfachtes Beispiel und soll nur dazu dienen die Mechanismen eines Nervensystems zu verstehen.

In Abbildung 2 ist zu sehen, wie ein visueller Eindruck in einem neuronalen Netzwerk verarbeitet wird. Im neuronalen Netzwerk werden bei verschiedenen visuellen Eindrücken verschiedene Neuronen unterschiedlich stark stimuliert. Umso mehr sich Objekte voneinander unterscheiden, desto mehr wird sich auch das neuronale Netzwerk in der Stimulation der einzelnen Neuronen unterscheiden. Dabei gibt es nicht nur zwei Stufen der Unterscheidung, sondern mehrere. Kennt man die Stimulationen der einzelnen Neuronen auf verschiedene geometrische Referenz-Objekte, wie eines 'Kreises', 'Dreiecks' oder 'Trapezes' kann man neue unbekannte geometrische Objekte mit bekannten vergleichen und eine Aussage darüber treffen, um was es sich für ein Objekt handelt. Die Entscheidung ist dabei mit Wahrscheinlichkeiten verbunden. Ist die Übereinstimmung mit einem Referenz-Objekt sehr hoch, wird es auch möglich sein eine gute Aussagekraft über das unbekannte Objekt zu erhalten. Umso mehr sich ein unbekanntes Objekt von den Referenz-Objekten unterscheidet, desto geringer wird auch die Aussagekraft über das unbekannte Objekt sein.

## **1.4. Neuronale Netze und ihre Bedeutung [2]**

Um besser zu verstehen, wie das menschliche Gehirn funktioniert wurden von Biologen, Neurophysiologen und Kognitionswissenschaftler neuronale Verarbeitungsmodelle entwickelt. Bei den entwickelten Verarbeitungsmodelle ist darauf zu achten, dass sie sehr stark vereinfacht sind. Sie sollen viel mehr genutzt werden, um Theorien zu untermauern. Die Verarbeitungsmodelle umfassen eine Vielzahl von Annahmen der menschlichen Gehirnstruktur, die nicht eindeutig belegt sind.

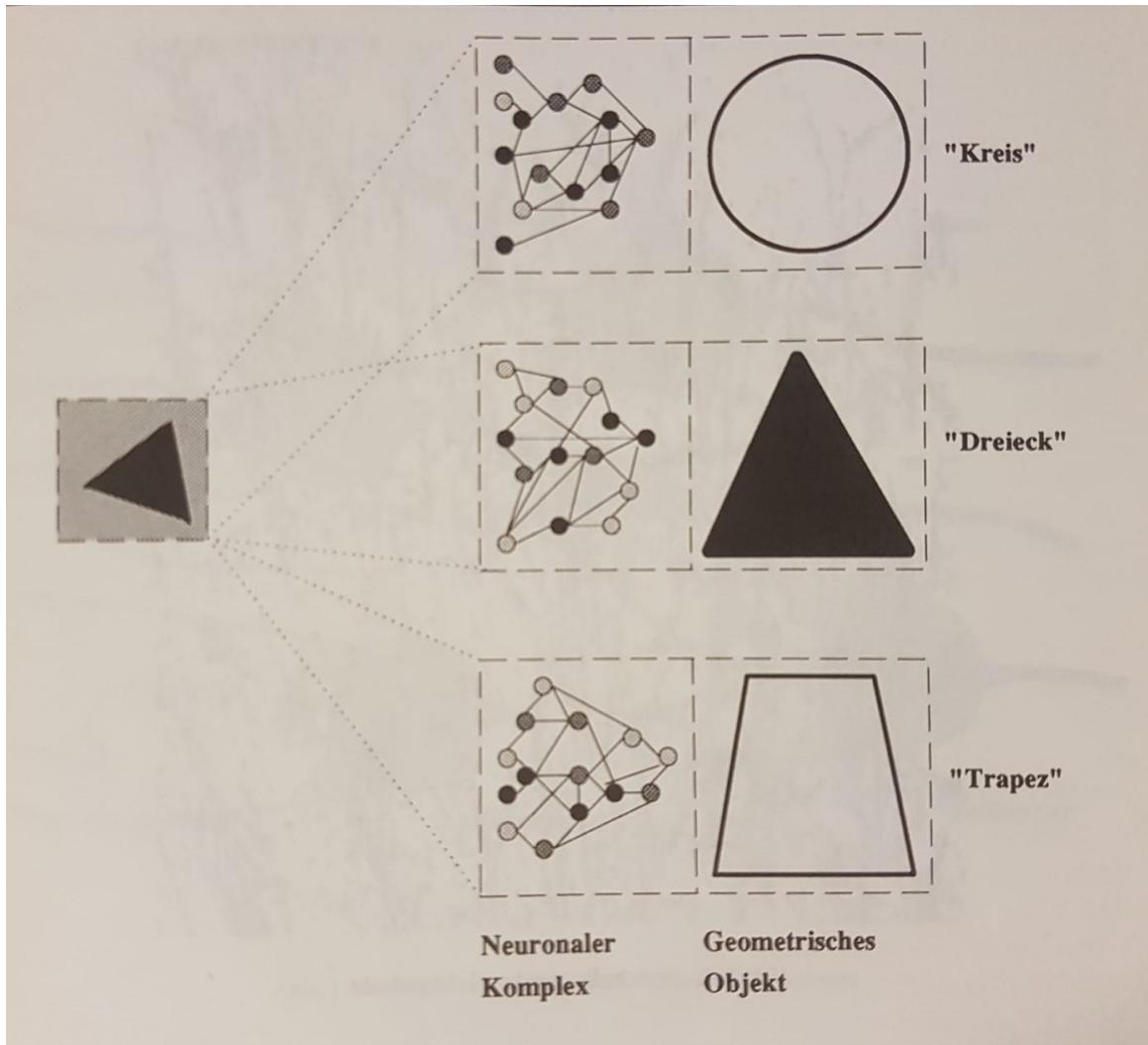


Abbildung 2: Vearbeitung eines visuellen Eindrucks [2, S. 9]

Neuronale Netzwerke sind Modelle der Gehirnfunktion. Sie versuchen, in Struktur und Funktionsweise Gehirnzellkomplexe nachzubilden und dadurch eine tragfähige Simulation komplexer menschlicher Denkvorgänge zu erzielen. [2, S. 12]

Am menschlichen Gehirn sind nicht nur die kreativen und phantasievollen Fähigkeiten erstaunlich. Auch die Fähigkeit etwas neues zu lernen und neue Lösungen für Probleme zu finden. Wenn Teile des menschlichen Gehirns beschädigt werden, wie zum Beispiel durch einen Schlaganfall, schafft es das menschliche Gehirn die auftretenden neurologischen Ausfälle teilweise bzw. komplett zu regenerieren. All diese Vorteile können nicht mit einem herkömmlichen Programm nachgebildet werden, können jedoch teilweise mit der Implementierung eines künstlichen neuronalen Netzwerks erreicht werden. Dabei müssen nicht alle Facetten des menschlichen Gehirns nachgebildet werden, sondern nur grundlegende Funktionen.

Ein Vorteil, den ein künstliches neuronales Netz von einem menschlichen Gehirn übernehmen kann ist unter anderem die Robustheit. Wenn bei einem hinreichend großem Netz ein Neuron

ausfällt, dann können umliegende Neuronen die Aufgabe übernehmen, wenn man das neuronale Netzwerk neu anlernen würde. Aber auch bei nicht neu anlernen des Netzwerkes würde zwar die Qualität der Aussage schlechter werden, aber es könnten immer noch Aussagen getroffen werden. Dies gilt bei Aufgabenstellungen, die assoziierenden, interpolierenden, klassifizierenden oder beurteilenden Charakter haben.

Ein zweiter Vorteil von künstlichen neuronalen Netzen in der modernen Mehrprozessor-Rechner-Architektur sind die Modellinhärenzen Parallelisierungsmöglichkeiten. So können Neuronen zu bestimmten Gruppen zusammengefasst werden, ähnlich wie es auch im menschlichen Nervensystem realisiert wird, und sehr stark miteinander vernetzt sein. Jede Gruppe wird dann auf verschiedenen Prozessoren ausgeführt und nur notwendige Verbindungen über die Prozessoren hinaus verbunden. Das gilt natürlich nicht exklusiv für Software-, sondern auch für Hardware-implementierungen.

Ein dritter Vorteil von künstlichen neuronalen Netzen ist die Adaptivität, also die Fähigkeit etwas zu erlernen. Dafür gibt es für fast jedes Netzmodell unterschiedliche Verfahren, die es dem Netzwerk erlauben sich selbst zu konfigurieren. Soll ein neuronales Netzwerk beispielsweise Muster erkennen, dann werden dem neuronalen Netzwerk vorab Beispiele für zu erkennende Muster gegeben und damit das Netzwerk konfiguriert. Man spricht hierbei auch von Trainingsbeispielen. Diese Trainingsbeispiele sollten möglichst reproduzierbar sein. Am Ende des Trainings soll das neuronale Netzwerk Muster in unbekannten Bildern durch Assoziation bzw. Interpolation erkennen.

Trotz dieser ganzen Vorteile künstlicher neuronaler Netze ist es nicht möglich zukünftig ohne die klassische Programmierung auszukommen. Für jede neue Anwendung muss immer eine Ein- und Ausgabecodierung vorliegen. Dies ist sowohl zur Schaffung einer sinnvollen Schnittstelle also Eingang zum Neuronalen Netz notwendig als auch für die Ausgabe. Somit ist es dem Anwender am Ende auch möglich das Ergebnis zu interpretieren. Bei den Ein- und Ausgaben handelt es sich um teils komplexe Transformationen. Neuronale Netzwerke bieten einen großen Einsatzbereich, sind aber nicht in der Lage die herkömmliche Programmierung komplett zuersetzen. Es kommt vielmehr auf das Einsatzgebiet an, wo man mehr mit neuronalen Netzwerken, mit herkömmlichen Programmiermethodiken oder mit einer Kombination arbeitet.

## 2. Aufbau des neuronalen Netzes

In diesem Abschnitt wird erklärt, wie unser neuronales Netz aufgebaut ist. Vor allem die Abbildung 3 ist für das bessere Verständnis der Matlab-Skripte wichtig. In den Matlab-Skripten, welche unter anderem im A zu finden sind, muss immer die Pixelanzahl pro Merkmal (in Abbildung 3 N) wie auch die Anzahl der Merkmale (in Abbildung 3 M) angegeben werden. Dabei handelt es sich immer um einen symmetrischen Aufbau. Ist die Merkmalanzahl gleich 5, dann gibt es 25 Merkmale (5 in x-Richtung und 5 in y-Richtung). Das gleiche gilt für die Pixelanzahl. Bei einer Pixelanzahl von 8 werden für jedes Merkmal so 64 Pixel angenommen. Bei einer Merkmalanzahl von 5 und einer Pixelanzahl von 8 wird also eine Pixelmatrix von 40 mal 40 Pixel erzeugt und jeweils anhand der Merkmale behandelt.

$1 \ 2 \ \dots \ 4 \ \dots \ N$ 2 ⋮ 4 ⋮ $N$	1	2	⋮	⋮	M
	2				
	⋮				
	⋮				
	M				

Abbildung 3: Anordnung der Merkmale und Pixel

In Abbildung 4 ist ein beispielhaftes Neuron abgebildet. Dieses Neuron der 1. Ebene fasst ein komplettes Merkmal mit all seinen Pixeln zusammen. Es verfügt also in unserem Beispiel über 64 Eingänge und es gibt 25 dieser Neuronen in der ersten Ebene des Netzes.

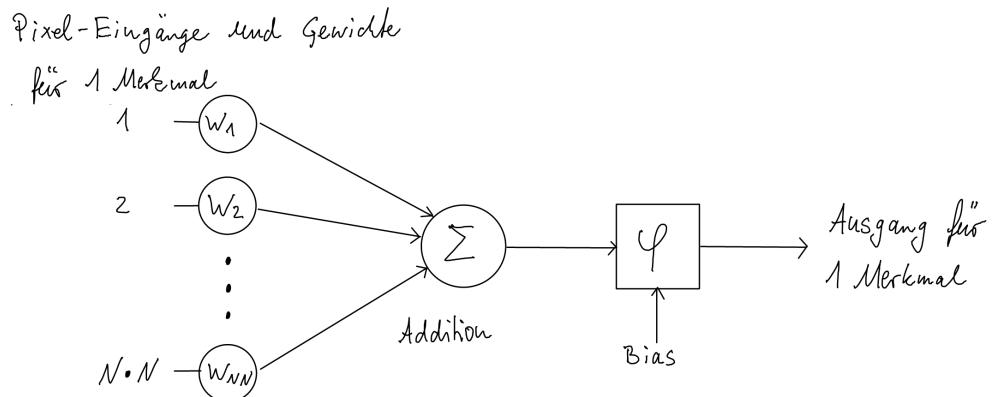


Abbildung 4: Neuron der 1. Ebene

Mit einer einzigen Ebene von Neuronen kann bereits viel erreicht werden. Allerdings ist für die Gesamtauswertung eine zweite Ebene von Vorteil, um zum Beispiel bei starkem Rauschen Fehler besser zu detektieren. Bevor wir die zweite Ebene der Neuronen betrachten, werfen wir einen Blick auf die Abbildung 5. Hier sind drei Arten von Merkmalen zu erkennen. Die fünf schwarz hervorgehobenen Merkmale detektieren einen horizontalen Balken, die blauen hervorgehobenen einen vertikalen Balken und die roten Merkmale können ein hohes Rauschen beziehungsweise auch zur Detektion von Fehlern ausgewertet werden. Die roten Merkmale erlauben es, eine bessere Konfiguration des neuronalen Netzes zu finden. Durch die Kombination der blau wie auch der schwarz hervorgehobenen Merkmale ergibt somit zusätzlich die Möglichkeit ein Kreuz zu detektieren.

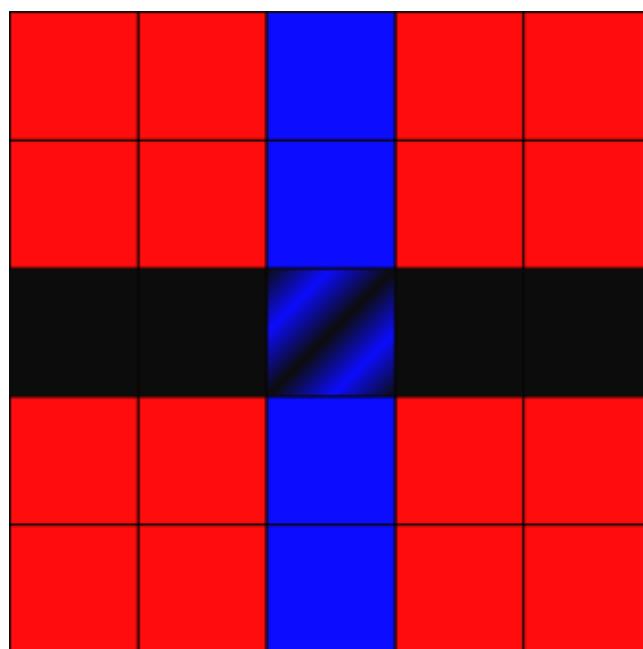


Abbildung 5: Erläuterung der Anordnung für die 2. Ebene

In Abbildung 6 ist die zweite Ebene des Neuronalen Netzes dargestellt. Die Ausgänge der einzelnen Merkmale dienen anhand ihres jeweiligen Merkmalstyps den Neuronen der zweiten Ebene als Input. In diesem Fall sind alle Gewichte auf eins gesetzt. Es wird lediglich der Bias eingestellt und somit die Schwelle zur Detektion festgelegt.

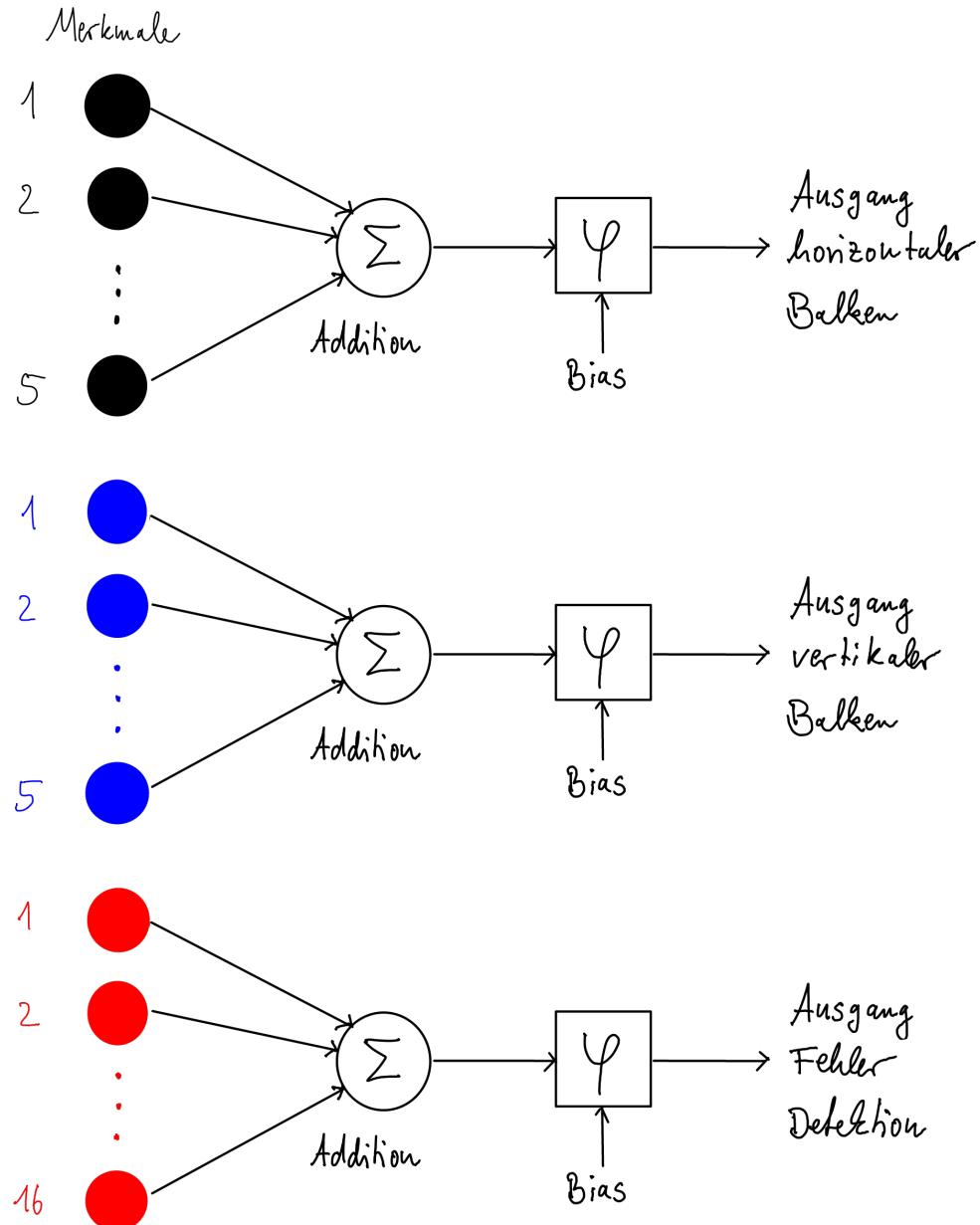


Abbildung 6: Neuronen der 2. Ebene

Für die Auswertung der Neuronen der erste Ebene nutzen wir eine Sigmoid-Funktion. Für die zweite Ebene werden die addierten Werte werden ohne eine weiteren Verarbeitung weiter gegeben. Das entspricht einer linearen Aktivierungsfunktion mit dem Anstieg eins.

### 3. Erzeugung der Eingangsbilder

Die Erzeugung von Eingangsbildern für unser neuronales Netz, war eines der ersten Probleme, welches wir gelöst haben. Testbilder zu suchen oder selbst zu erstellen kann sehr zeitaufwendig sein. Aus diesem Grund haben wir zwei Matlab-Skripte geschrieben, die uns dieses Problem zukünftig abnehmen. Die beiden Dateien heißen 'GetPixelFeatureMatrix.m' A.1 und 'GetInputFeatureMatrix.m' A.2. Die Funktion 'GetPixelFeatureMatrix' erwartet als Eingang eine Merkmale-Matrix, die dann auf eine Pixel-Matrix hochskaliert wird. Darüber hinaus kann auch ein Rauschwert angegeben werden, um das neuronale Netz auf Rauschempfindlichkeit zu testen. Im Anschluss sind ein paar Bilder erzeugt worden. Im Sinne der Lesbarkeit sind im Weiteren alle angegebenen Werte für Merkmale, Pixel oder Pixel pro Merkmal lediglich für eine Dimension angegeben, da diese stets symmetrisch sind. Die ersten beiden Abbildungen sollen die Flexibilität der Funktion verdeutlichen und verfügen über 7 Merkmale, 64 Pixel pro Merkmal und einen Rauschanteil von 10%. Die meisten der nachfolgenden Abbildungen verfügen über fünf Merkmale, 128 Pixel pro Merkmal und somit insgesamt über 640 Pixel und unterscheiden sich neben des dargestellten Musters hauptsächlich in ihrer Rauschintensität. Eine Ausnahme bildet lediglich die Abbildung 10 mit 9 Merkmalen, 65 Pixeln pro Merkmal und einem Rauschanteil von 10%. Somit wird bei den Bildern lediglich die Besonderheiten beziehungsweise der jeweilige Rauschanteil genannt.

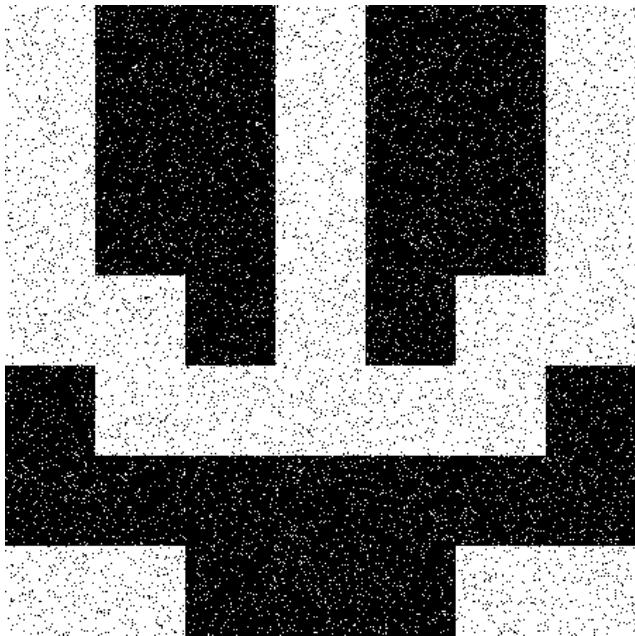


Abbildung 7: Beispielbild 1 mit 10% Rauschen

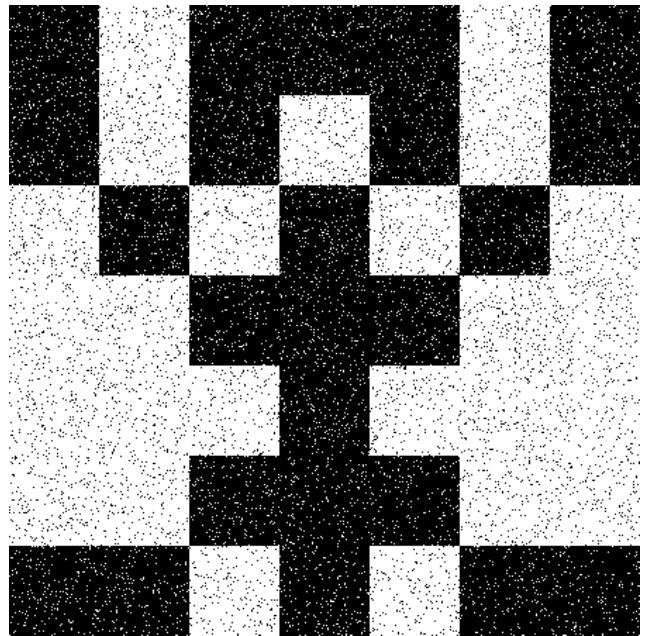


Abbildung 8: Beispielbild 2 mit 10% Rauschen

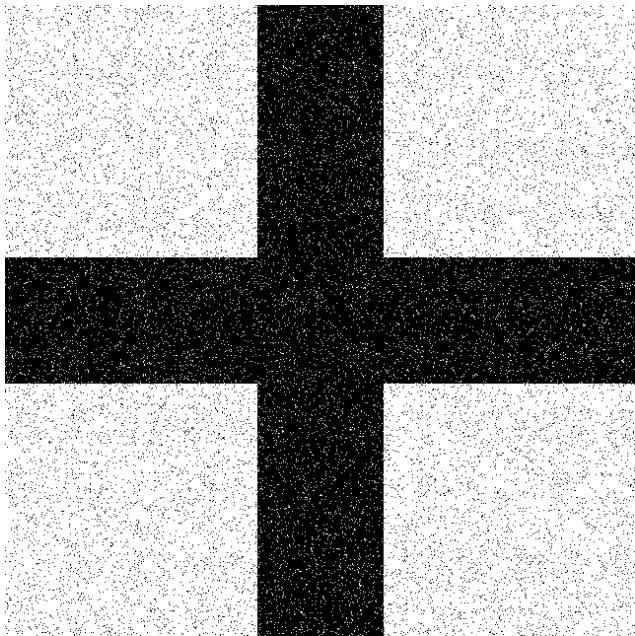


Abbildung 9: Kreuz mit 10% Rauschen

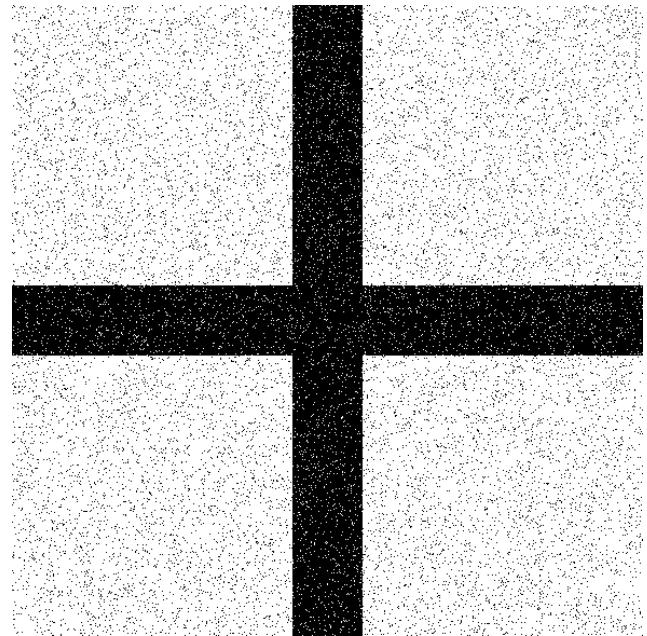


Abbildung 10: 9 Merkmale & 10% Rauschen

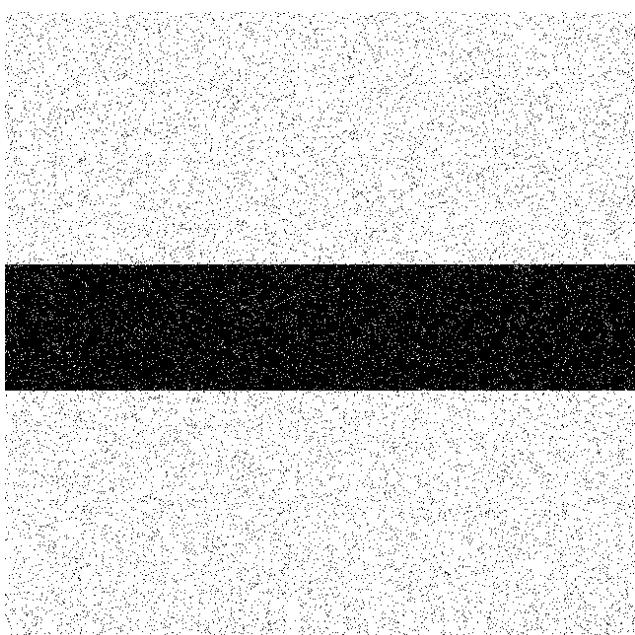


Abbildung 11: Horizontal mit 10% Rauschen

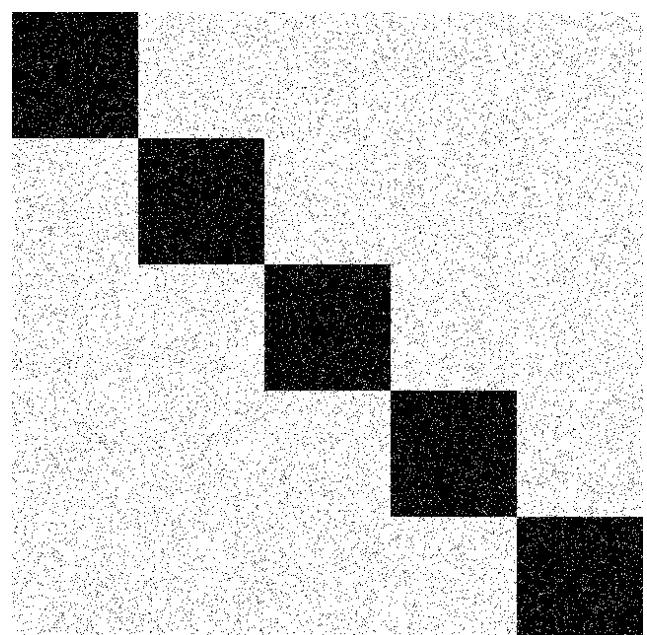


Abbildung 12: Diagonale mit 10% Rauschen

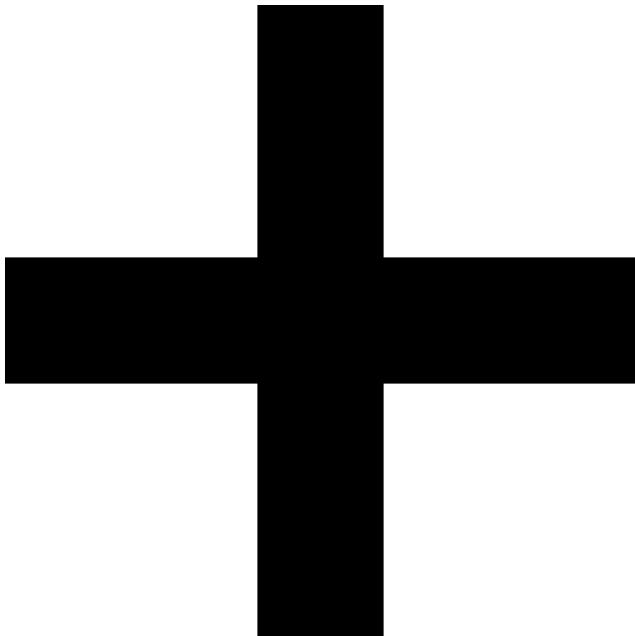


Abbildung 13: Kreuz ohne Rauschen

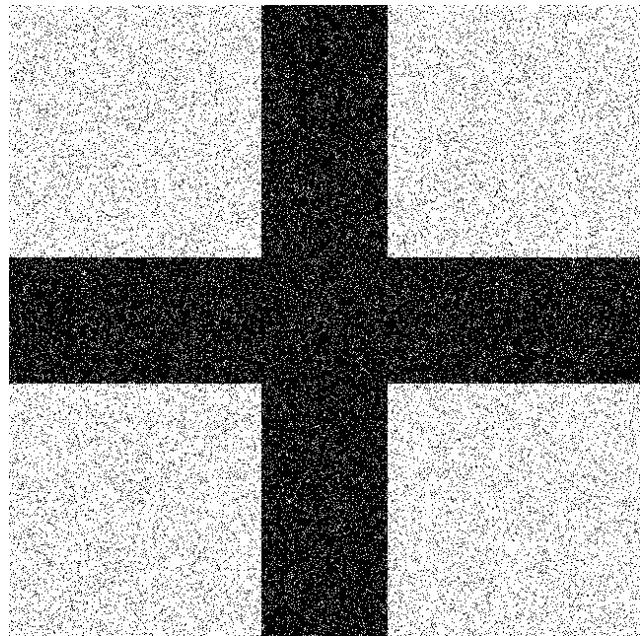


Abbildung 14: Kreuz mit 20% Rauschen

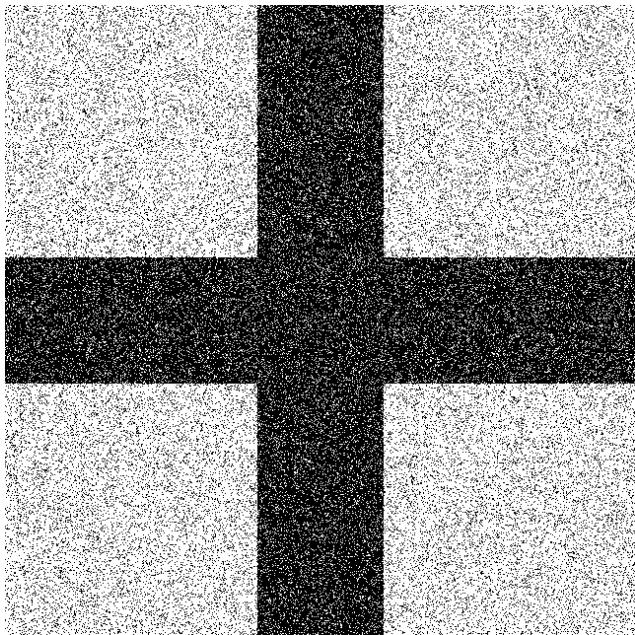


Abbildung 15: Kreuz mit 30% Rauschen

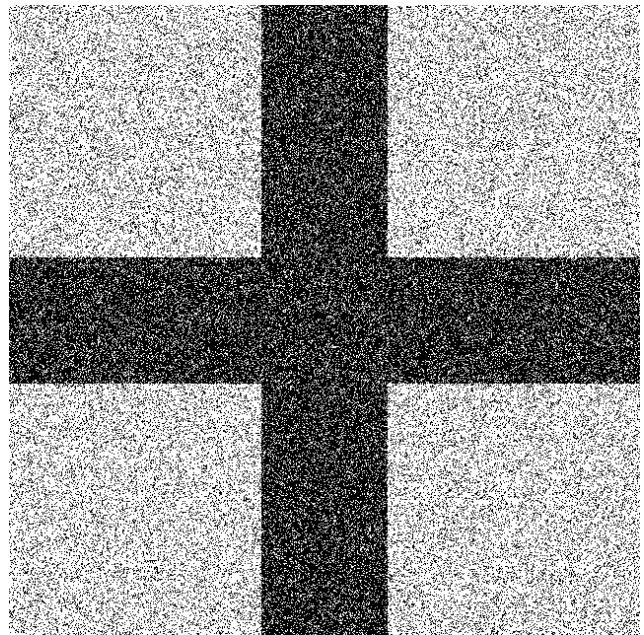


Abbildung 16: Kreuz mit 40% Rauschen

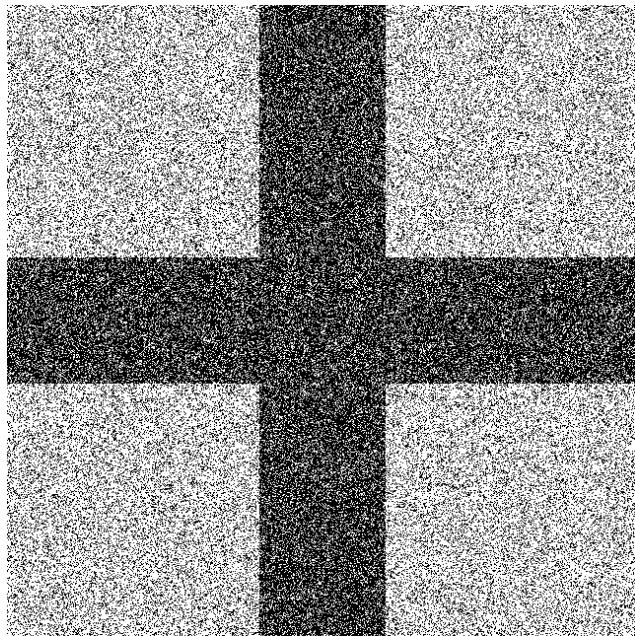


Abbildung 17: Kreuz mit 50% Rauschen

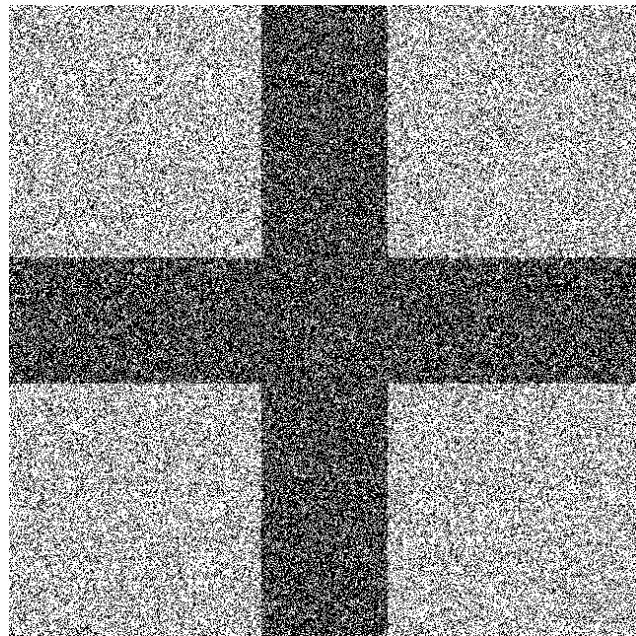


Abbildung 18: Kreuz mit 60% Rauschen

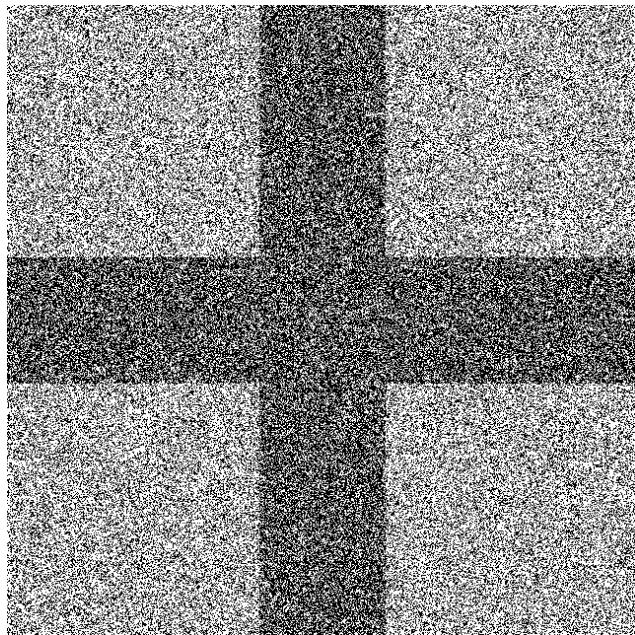


Abbildung 19: Kreuz mit 70% Rauschen

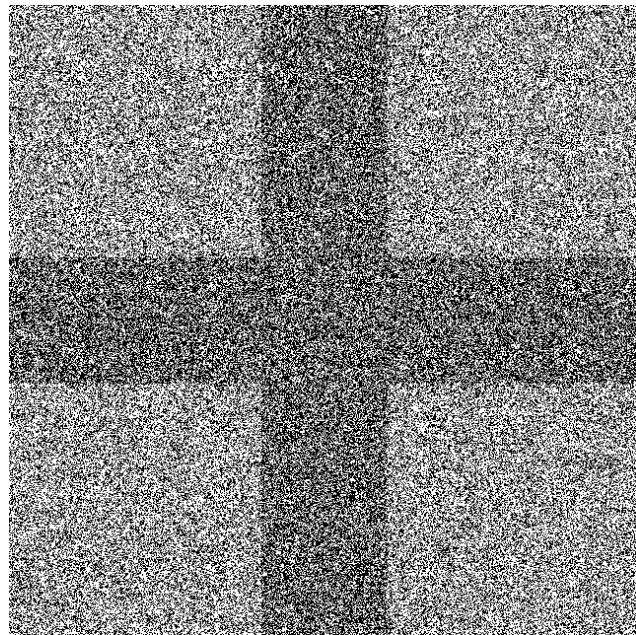


Abbildung 20: Kreuz mit 80% Rauschen

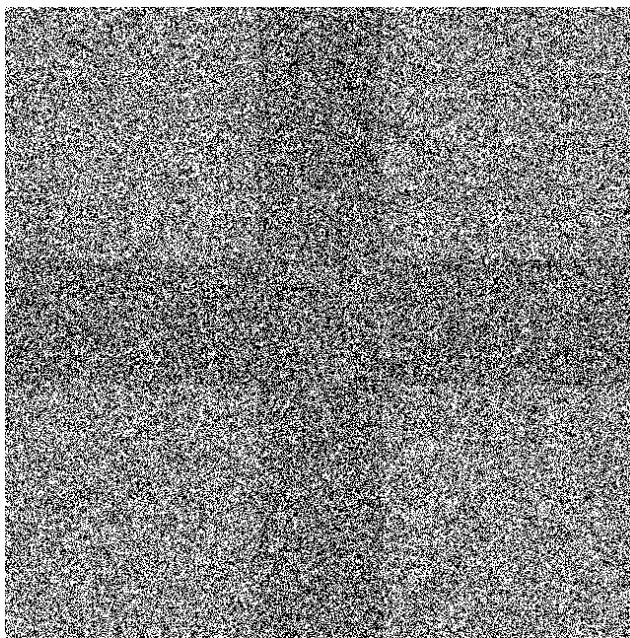


Abbildung 21: Kreuz mit 90% Rauschen

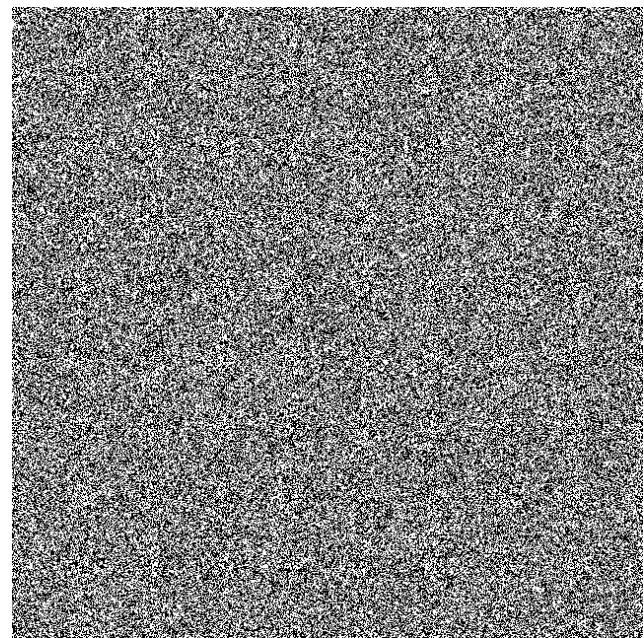


Abbildung 22: Kreuz mit 100% Rauschen

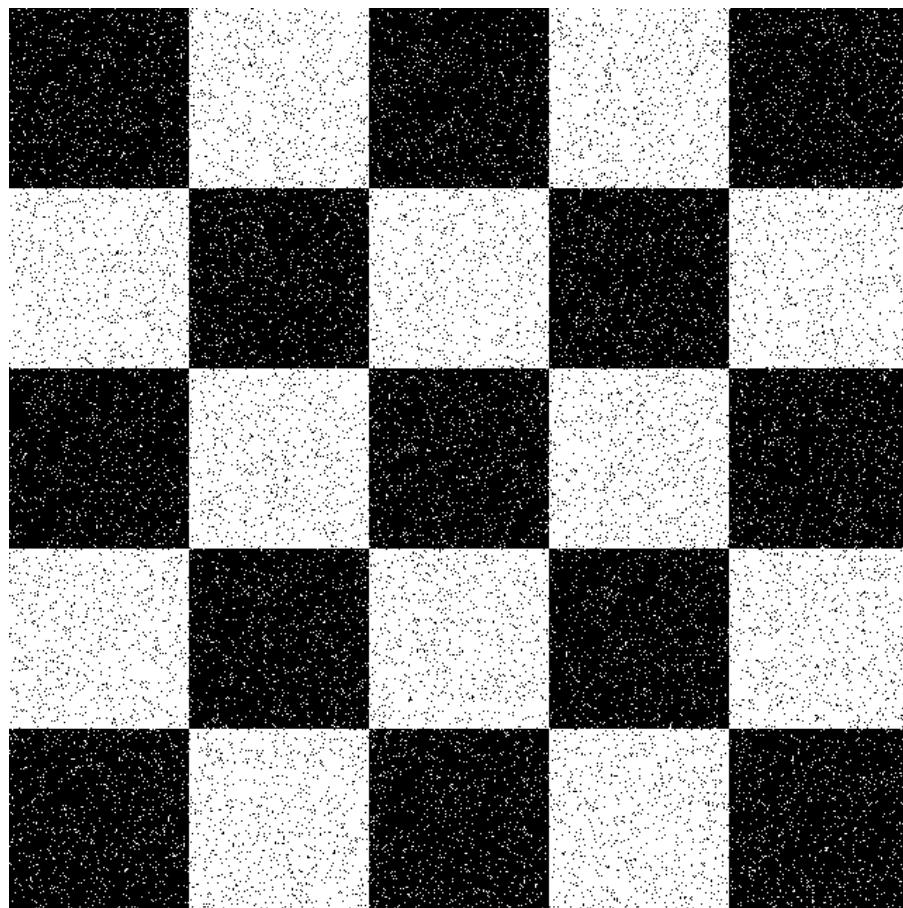


Abbildung 23: Schachbrettmuster mit 10% Rauschen

## 4. Gewichtsmatrizen

In diesem Abschnitt werden verschiedene Gewichtsmatrizen vorgestellt und wie diese erzeugt wurden. Wir haben uns zunächst auf die Erzeugung mit der Gauß-Verteilung konzentriert. Für spätere Anpassungen oder Versuche wurden weitere Funktionen implementiert, wie zum Beispiel die Rayleigh-Funktion oder der Tangenshyperbolicus, jedoch nicht ausgewertet.

Zunächst soll erläutert werden, wie die Gewichtsmatrizen grob erstellt wurden. Dafür werden im Abschnitt 4.1 3 Beispiele mit jeweils einem Slope, beziehungsweise einer Steigung, von 75 gezeigt. Es soll außerdem verdeutlicht werden, dass es sich immer nur um einen Gauß in x- und y-Richtung handelt. Des Weiteren ist das Endergebnis immer zwischen -1 und 1 definiert, obwohl auch dies variabel eingestellt werden kann. Jedoch kann somit später der Ergebnisbereich einfacher skaliert werden. Die vollständige Erzeugung der Gewichtsmatrizen für diese Beispiele ist im Appendix A.3 aufgeführt. In den anderen Unterabschnitten sind für jede Art der Gewichtsmatrizen unterschiedliche Steigungen eingestellt.

## 4.1. Erzeugung der Gewichtsmatrizen

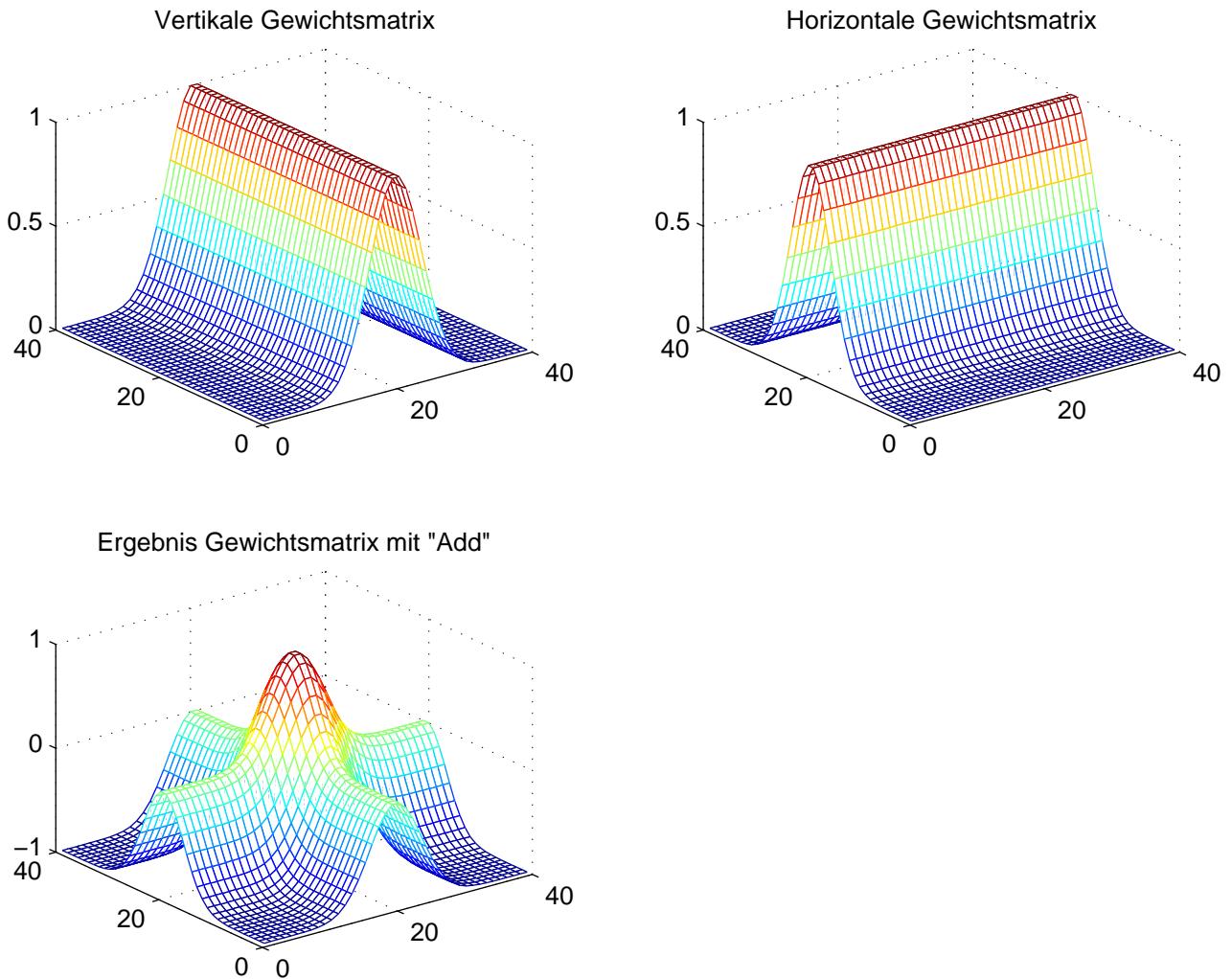


Abbildung 24: Additive Überlagerung mit Slope von 75

In Abbildung 24 wurde der 3D-Gauß in x- und y-Richtung additiv Überlagert, mit 2 multipliziert und am Ende um 1 heruntergesetzt. Siehe Listing 4.1.

Listing 1: Auszug GetGaussWeights: Additive Überlagerung A.3

```

74 elseif strcmp(type, 'Add')
75     for i = 1:vN
76         tempWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) + (hGauss./max(hGauss)))./2;
77         tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) .* 2 - 1;
78     end

```

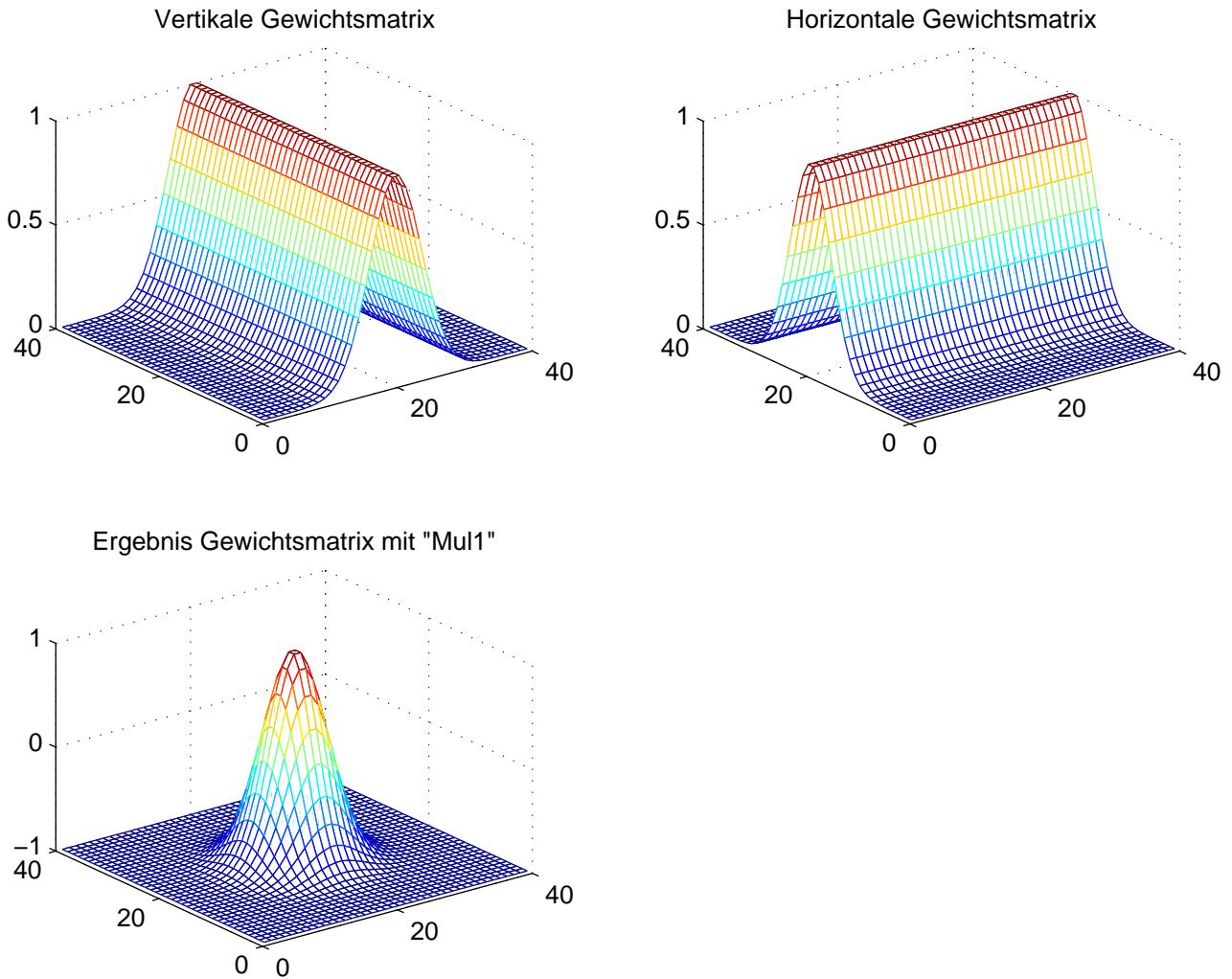


Abbildung 25: Multiplikative Überlagerung Typ 1 mit Slope von 75

In Abbildung 25 wurde der 3D-Gauß in x-Richtung mit dem 3D Gauß in y-Richtung multiplikativ Überlagert, mit 2 multipliziert und um 1 heruntergesetzt. Siehe Listing 2.

Listing 2: Auszug GetGaussWeights: Multiplikative Überlagerung Typ 1 A.3

```

62 if strcmp(type, 'Mul1')
63   for i = 1:vN
64     tempWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* (hGauss./max(hGauss));
65     tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) .* 2 - 1;
66   end

```

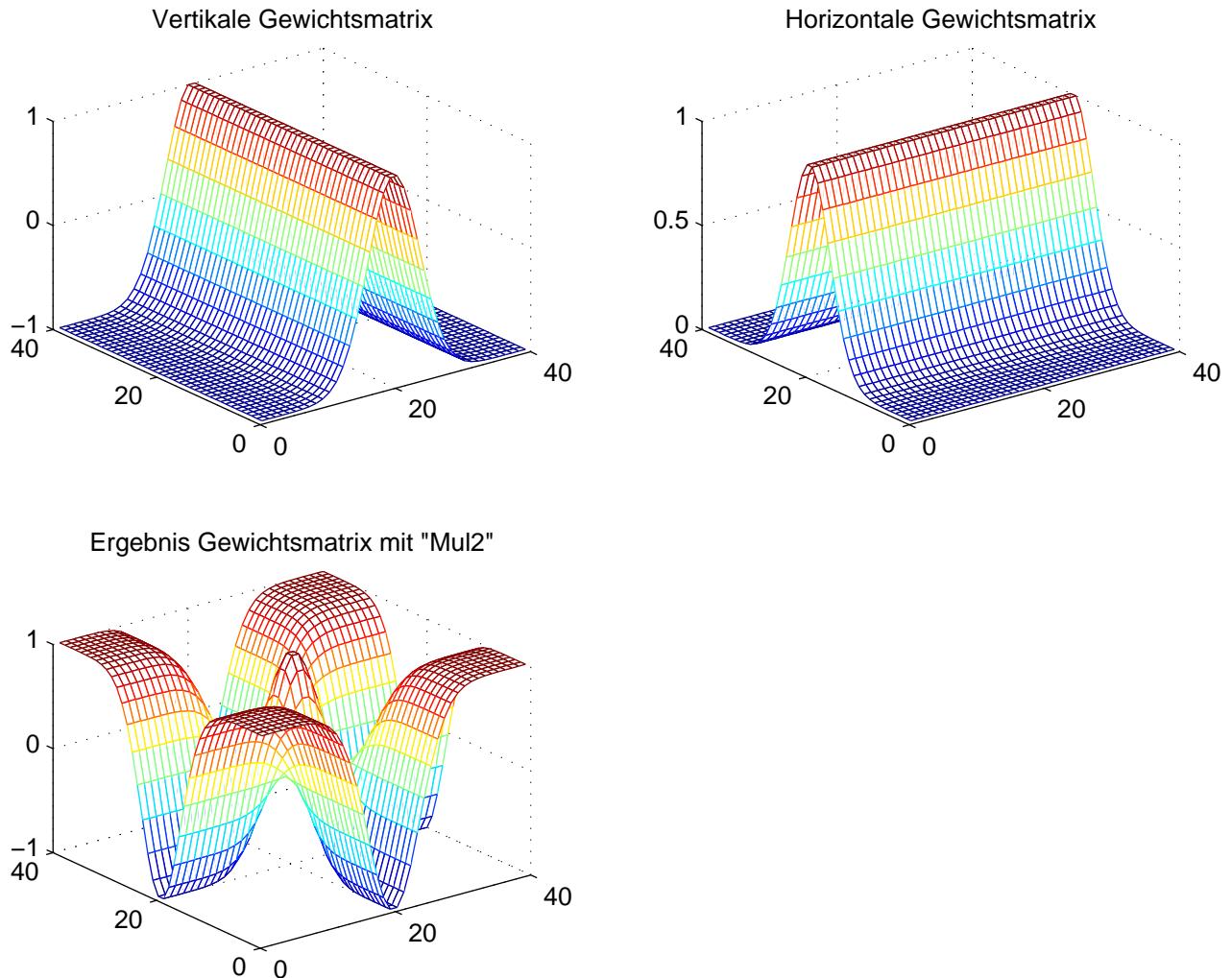


Abbildung 26: Multiplikative Überlagerung Typ 2 mit Slope von 75

In Abbildung 26 wurde der 3D-Gauß in y-Richtung erst einmal zwischen -1 und 1 skaliert. Anschließend wird der 3D-Gauß in x-Richtung ebenfalls zwischen -1 und 1 skaliert. Am Ende werden beide multiplikativ überlagert. Siehe auch das Listing 3.

Listing 3: Auszug GetGaussWeights: Multiplikative Überlagerung Typ 2 A.3

```

67 elseif strcmp(type, 'Mul2')
68     for i = 1:vN
69         % skalieren auf -1 bis 1
70         vWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* 2 - 1;
71         % v-Gauss und h-Gauss multiplizieren
72         tempWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* ((hGauss./max(hGauss)) * 2 - 1)';
73     end

```

## 4.2. Additive Überlagerung

Ergebnisse der Additiven Überlagerung mit unterschiedlichen Slope-Werten.

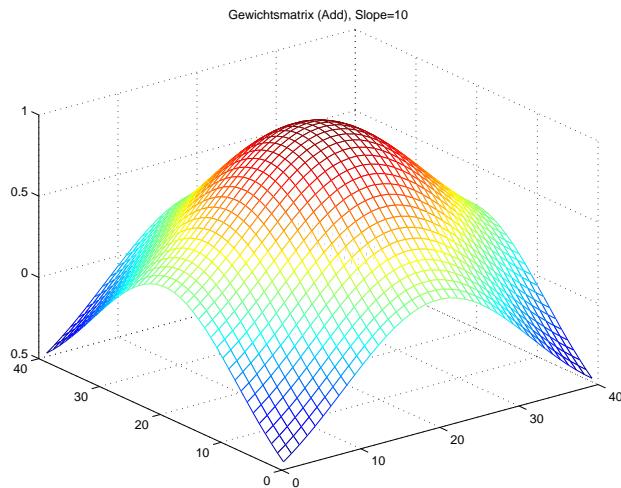


Abbildung 27: Additive mit Slope 10

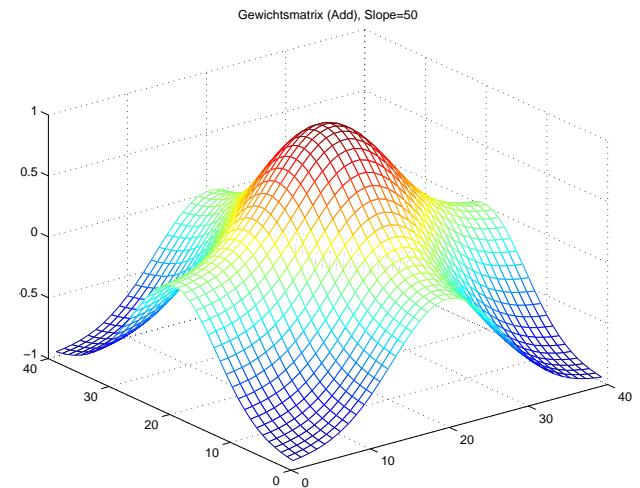


Abbildung 28: Additive mit Slope 50

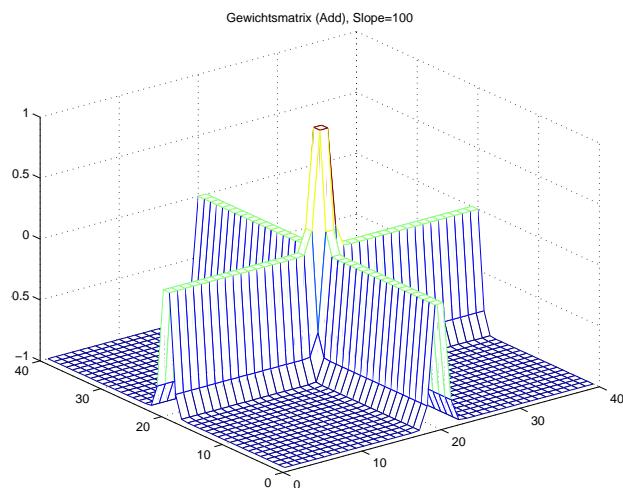


Abbildung 29: Additive Überlagerung mit Slope von 100

### 4.3. Multiplikative Überlagerung Typ 1

Ergebnisse der Multiplikativen Überlagerung Typ 1 mit unterschiedlichen Slope-Werten.

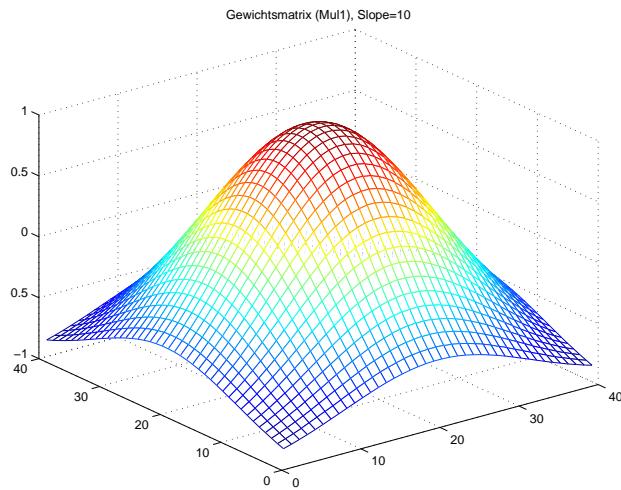


Abbildung 30: Multi-Typ 1 mit Slope von 10

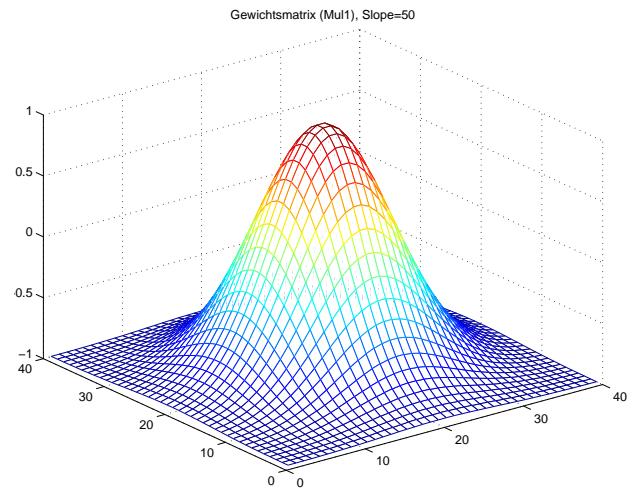


Abbildung 31: Multi-Typ 1 mit Slope von 50

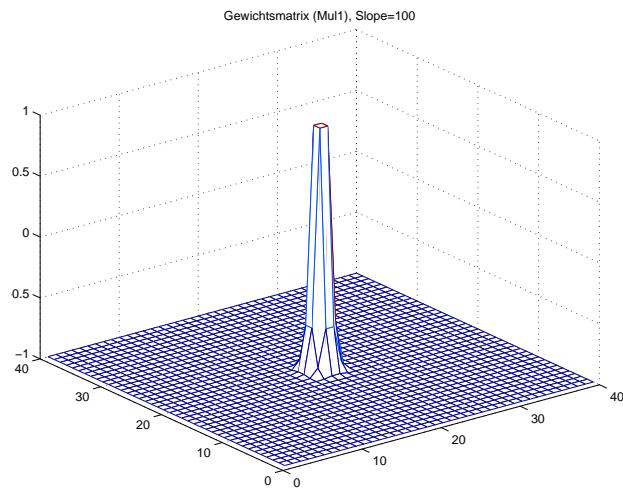


Abbildung 32: Multiplikative Überlagerung Typ 1 mit Slope von 100

## 4.4. Multiplikative Überlagerung Typ 2

Ergebnisse der Multiplikativen Überlagerung Typ 1 mit unterschiedlichen Slope-Werten.

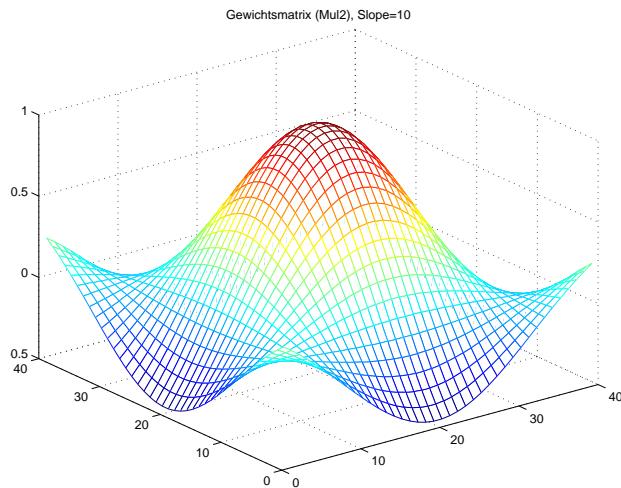


Abbildung 33: Multi-Typ 2 mit Slope von 10

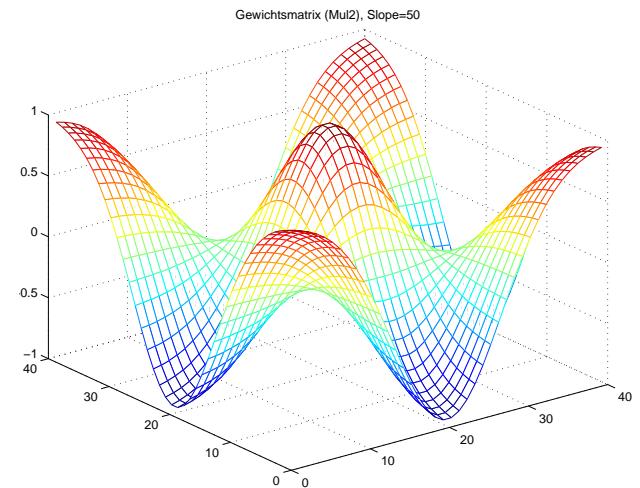


Abbildung 34: Multi-Typ 2 mit Slope von 50

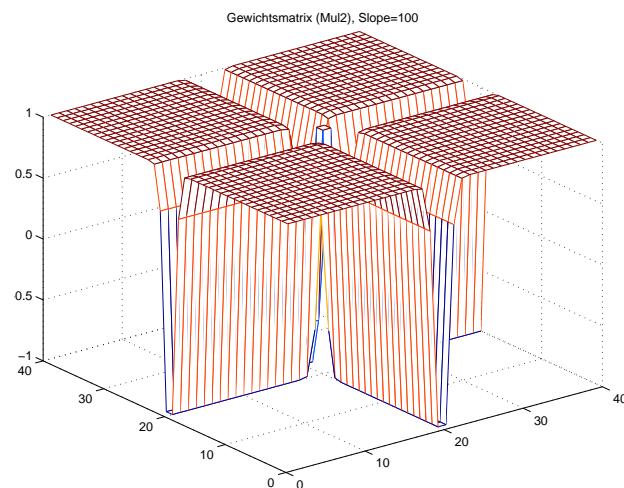


Abbildung 35: Multiplikative Überlagerung Typ 2 mit Slope von 100

## 4.5. Additive und Multiplikative Überlagerung

Diese Art der Überlagerung entstand als Idee die Überhöhung im Zentrum der Gewichtsmatrix, siehe Abbildung 24, zu vermeiden und zusätzlich die Schulternäuf 1 anzuheben. Hierfür werden die beiden Gauß-Matrizen zunächst additiv überlagert und anschließend wird die multiplikative Überlagerung abgezogen. Siehe Listing 4. Die Beschriftung wurde mit 'AddMul' abgekürzt.

Listing 4: Auszug GetGaussWeights: 'AddMul'-Überlagerung A.3

```

79 elseif strcmp(type, 'AddMul')
80     for i = 1:vN
81         % v-Gauss und h-Gauss multiplizieren
82         addMulWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) .* (hGauss./ max(hGauss))') - 1;
83         % v-Gauss und h-Gauss addieren
84         tempWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) + (hGauss./ max(hGauss))') - 1;
85         % mittlere Erhöhung entfernen
86         tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) - addMulWeightMatrix(1:end, i);
87         % skalieren
88         tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) .* 2 - 1;
89     end

```

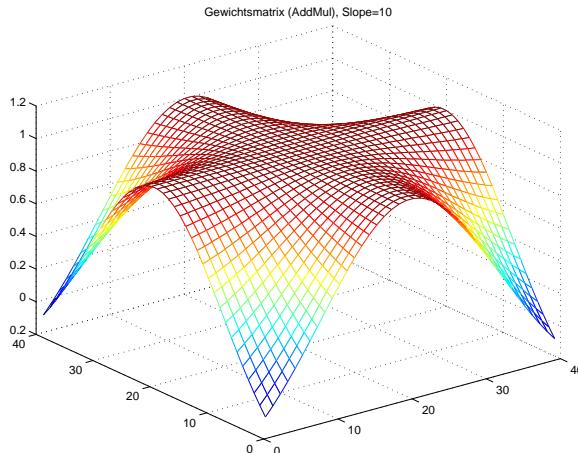


Abbildung 36: AddMul mit Slope 10

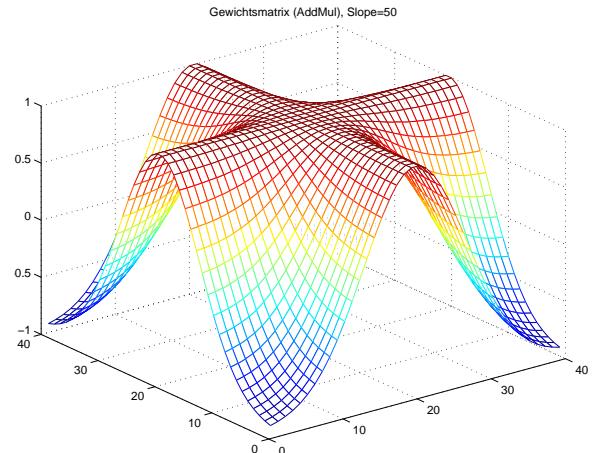


Abbildung 37: AddMul mit Slope 50

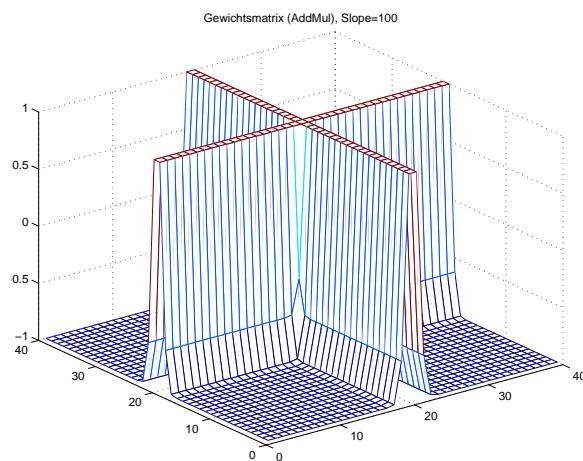


Abbildung 38: Additive und Multiplikative Überlagerung mit Slope von 100

## 4.6. Spezielle Überlagerung

Bei der Speziellen Gewichtsmatrix hat der Slope keine Auswirkungen. Das war viel mehr ein Versuch, die Auswirkung von erhöhten Gewichten in den Randbereichen zu untersuchen. Sie wurde empirisch aus den Typen: 'Mul1', 'Mul2' und 'AddMul' ermittelt, siehe Listing 5.

Listing 5: Auszug GetGaussWeights: Spezielle Überlagerung A.3

```

104     elseif strcmp(type, 'Special')
105         weightMatrix1 = GetGaussWeights(pixelCnt, featureCnt, 45, 'Mul2', -2, 2);
106         weightMatrix2 = GetGaussWeights(pixelCnt, featureCnt, 70, 'AddMul', -2, 2);
107         weightMatrix3 = GetGaussWeights(pixelCnt, featureCnt, 45, 'Mul1', -2, 2);
108         tempWeightMatrix = weightMatrix1 + weightMatrix2 - weightMatrix3 - 1;

```

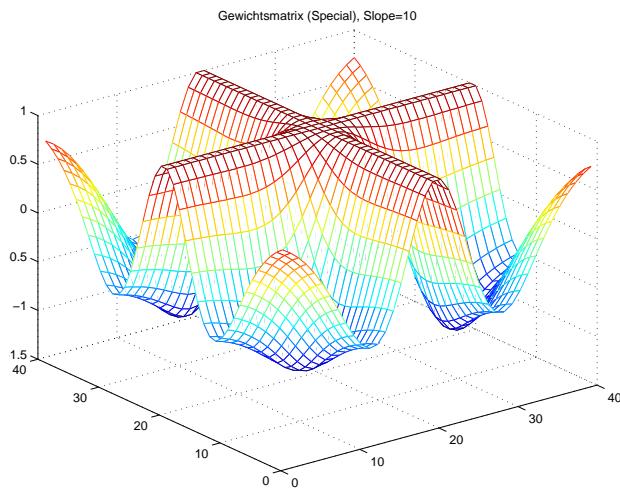


Abbildung 39: Spezielle mit Slope 10

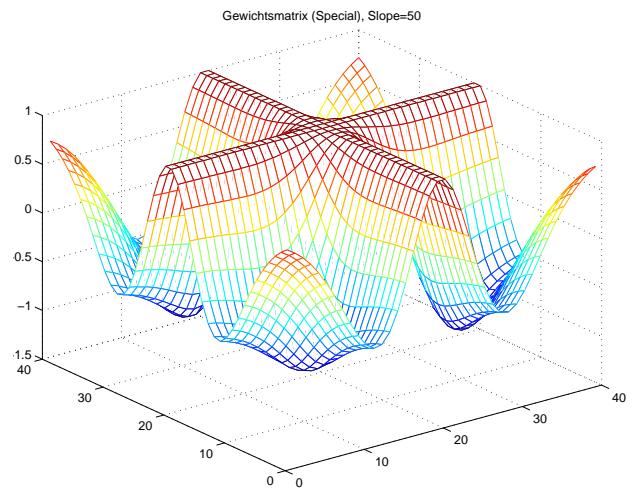


Abbildung 40: Spezielle mit Slope 50

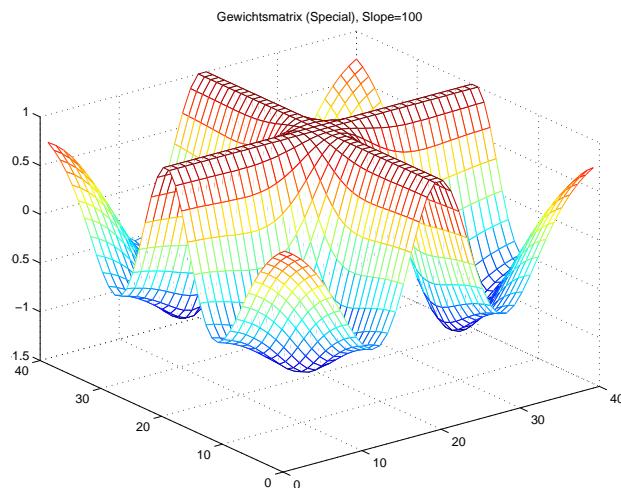


Abbildung 41: Spezielle Überlagerung mit Slope von 100

## 5. Das Matlab Tool

Dieser Abschnitt soll dazu dienen, die Matlab-Skripte besser verstehen zu können. Es soll keine komplette Anleitung sein, aber die wichtigsten Zusammenhänge darstellen und Erfahrungswerte vermitteln. In diesem Projekt sind viele Zeilen Code in Matlab geschrieben worden. Es sollte erreicht werden, dass der Aufbau der einzelnen Dateien möglichst generisch ist. Die Entwicklung hat mithilfe von GitHub stattgefunden. Über diesen Link stehen dort alle, auch die hier nicht erläuterten, Dateien in dem zu dieser Dokumentation passenden Stand zur Verfügung.

### 5.1. Modulübersicht

Die Matlab-Skripte können in Kategorien eingeteilt werden. Die folgende Übersicht soll darüber Aufschluss geben.

#### Ablauf und Einstellungen:

In diesen Dateien werden alle benötigten Module aufgerufen. In der Datei 'main.m' ist der komplette Ablauf hinterlegt. Für eine bessere Reproduzierbarkeit haben wir zwei weitere Dateien angelegt, in denen wir Grundeinstellungen für die Gewichtsmatrizen getroffen haben, die wir in dieser Dokumentation auswerten.

- main.m A.4 Die Hauptdatei unseres Tools.
- SettingFileAddMul.m A.5 Einstellungen für die Gewichts-Matrix 'AddMul'.
- SettingFileSpecial.m A.6 Einstellungen für die Gewichts-Matrix 'Special'.

#### Module:

In dieser Auflistung werden alle relevanten Module gezeigt, welche für die Grundfunktionen verantwortlich sind.

- GetPixelFeatureMatrix.m A.1

Diesem Modul wird eine Merkmale-Matrix übergeben und das Modul skaliert dann die Merkmale-Matrix auf eine Pixel-Matrix. Außerdem kann dem Modul ein Wert für das Rauschen zwischen 0 und 100 übergeben werden, um die Bilder mit Rauschen zu versehen. Dieser Wert wird als Prozentanteil interpretiert. Das Modul gibt eine Pixel-Matrix zurück und kann die erzeugten Bilder optional als Datei speichern.

- GetNeuronOutput.m A.7

Dieses Modul ist das eigentliche Neuron. Es müssen als Parameter die Eingangszustände und Gewichte übergeben werden. Es berechnet anschließend über die Aktivierungsfunktion das Ergebnis. Es werden darüber hinaus auch Zwischenergebnisse zurückgegeben.

- GetInputFeatureMatrix.m A.2

Dieses Modul gibt vordefinierte Merkmale-Matrizen zurück, die wir häufig verwendet haben.

- GetGaussWeights.m A.3

Dieses Modul gibt eine Gewichts-Matrix zurück. Welche hinterlegt sind wurde bereits im Abschnitt 4 erläutert.

- GetFeatureOfMatrix.m A.8

Diesem Modul kann eine Pixel-Matrix übergeben werden und gibt eine Merkmale-Matrix zurück.

- ConvMatrixToColumn.m A.9

Das Modul konvertiert eine Merkmale-Matrix in einen Spaltenvektor.

## Funktionen:

Diese Funktionen wurden zusätzlich von uns implementiert. Im akutellen Stand wurde allerdings nur die Gauß- beziehungsweise Sigmoid-Funktion verwendet. Deshalb sind auch lediglich diese im Appendix A aufgeführt.

- GaussNormFunction.m A.10
- SigmoidFunction.m A.11
- LinearFunction.m
- RayleighFunction.m
- TangHFunction.m

## Hilfsfunktionen:

Die Hilfsfunktionen haben für das eigentliche Tool keine oder kaum Bedeutung. Sie werden nur genutzt, um Module zu testen oder Ausgaben zu generieren und sind ebenfalls im Appendix nicht aufgeführt jedoch im Release enthalten.

- TestTangHFunction.m
- TestSigmoidFunction.m
- TestRayleighFunction.m
- TestLinearFunction.m
- TestGetPixelFeatureMatrix.m
- TestGetGaussWeights.m
- TestGetFeatureOfMatrix.m
- TestGaussNormFunction.m
- SummaryWeights.m
- SummaryFunctions.m
- savePic.m

## 5.2. Erläuterung der Parameter

Im Matlab-Skript 'main.n' und die abgeleiteten Skripte 'SettingFileAddMul' und 'SettingFileSpecial' sind viele Parameter enthalten, die eingestellt werden können. Dieser Abschnitt soll einen Überblick geben, was die einzelnen Parameter bewirken.

Zeile	Parameter	Erläuterung
5	pixelCnt	Pixel Anzahl in x- und y-Richtung
6	featureCnt	Merkmal Anzahl in x- und y-Richtung
7	weightType	Gewichts-Matrix Typ (AddMul, Special, Add, Mul1, Mul2)
8	inFeatureType	voreingestellten Merkmale-Matrizen (Cross, H_Line, V_Line)
9	noise	Verrauschungsgrad zwischen 0 und 100 in Prozent
10	slope	Steigung der Aktivierungs-Funktion
13	bias	Verschiebung der Aktivierungsfunktion (negativ nach rechts)
14	threshold	Auswertungsschwelle des Ergebnisses
15	domainOfDefinition	Gueltigkeitsbereich der Neuronenfunktion (+/-)
18	lowerBound	Untere Grenze der Gewichts-Matrix
19	upperBound	Obere Grenze der Gewichts-Matrix
125	domainOfDefinition	2. Neuronen Ebene, h-Balken
126	bias	2. Neuronen Ebene, h-Balken
127	threshold	2. Neuronen Ebene, h-Balken
151	domainOfDefinition	2. Neuronen Ebene, v-Balken
152	bias	2. Neuronen Ebene, v-Balken
153	threshold	2. Neuronen Ebene, v-Balken
177	domainOfDefinition	2. Neuronen Ebene, Fehler Detektion
178	bias	2. Neuronen Ebene, Fehler Detektion
179	threshold	2. Neuronen Ebene, Fehler Detektion

Tabelle 1: Übersicht der Parameter

## 6. Auswertung

Für die Auswertung haben wir zwei von den vorgestellten Gewichtsmatrizen untersucht. Wir haben uns für die Gewichtsmatrizen 'AddMul' und 'Special' entschieden. Um reproduzierbare Ergebnisse liefern zu können, haben wir aus der 'main.m' zwei Dateien abgeleitet in denen die Konfigurationen der nachfolgenden Bilder hinterlegt sind. Die beiden Dateien heißen 'SettingFileAddMul.m' A.5 und 'SettingFileSpecial.m' A.6. Beide Systeme wurde empirisch eingestellt und es wurde versucht ab einem Rauschen von 50% keine vernünftigen Ergebnisse mehr zu liefern. Außerdem wurden die Bias- und Threshold-Werte voreingestellt.

Als nächstes soll für ein Beispiel besprochen werden, wie die Grafiken zu lesen sind. Dafür benutzen wir die Abbildungen 42 und 43 heran. Untersucht wurde die AddMul-Gewichtsmatrix mit horizontalen Balken. Der Rauschwert ist auf 40% eingestellt. Die Farbe der eingezeichneten Punkte gibt an, ob das jeweilige Merkmal richtig detektiert wurde oder nicht. In Abbildung 42 sehen wir das Ergebnis der ersten Neuronen Ebene. In der Sigmoid-Grafik sind 25 grüne Punkte eingezeichnet. Es sind also alle Merkmale dem gewünschten Ergebnis entsprechend richtig detektiert worden. An dieser Stelle werden die addierten Werte der einzelnen Pixel auf eine Sigmoid-Funktion gegeben und grafisch ausgewertet. Zur Verdeutlichung ist zusätzlich noch die Ausgabe der ersten Ebene als Bild aufgeführt.

In Abbildung 43 ist die zweite Neuronen-Ebene dargestellt. In jeder Grafik ist jeweils ein Punkt eingezeichnet. In der Grafik der 'h-Balken Detektion' ist ein grüner Punkt eingezeichnet, d.h. es wurde ein 'h-Balken' detektiert. In der Grafik der 'v-Balken Detektion' ist ein roter Punkt eingezeichnet, d.h. es wurde kein vertikaler Balken detektiert. Die letzte Grafik zeigt das Ergebnis der 'Fehler Detektion'. In diesem Fall ist der Punkt grün, also wurde keinen Fehler detektiert. Die 'Fehler Detektion' soll angeben, wenn der voreingestellte Rauschwert überschritten wurde. Es ist zu beachten, dass die Werte aus der ersten Ebene nicht mit der Sigmoid-Funktion bearbeitet wurden, sondern lediglich mit einer linearen Aktivierungsfunktion mit dem Anstieg 1, d.h. die addierten Werte der Pixel pro Merkmal werden direkt weitergereicht. Wie zu erwarten zeigt die Grafik für den horizontalen Balken einen grünen Punkt und somit wurde ein solcher erkannt.

## 6.1. AddMul Gewichtsmatrix

### 6.1.1. H-Balken

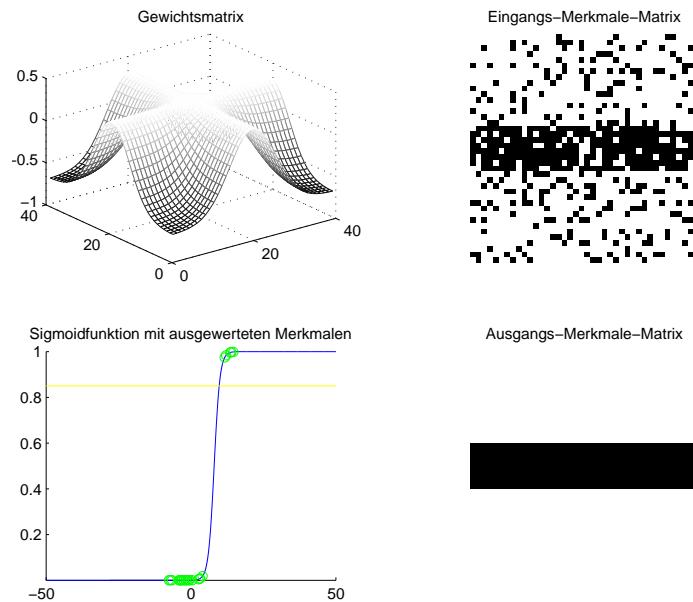


Abbildung 42: AddMul, H-Balken, 40% Rauschen, 1. Neuronen Ebene

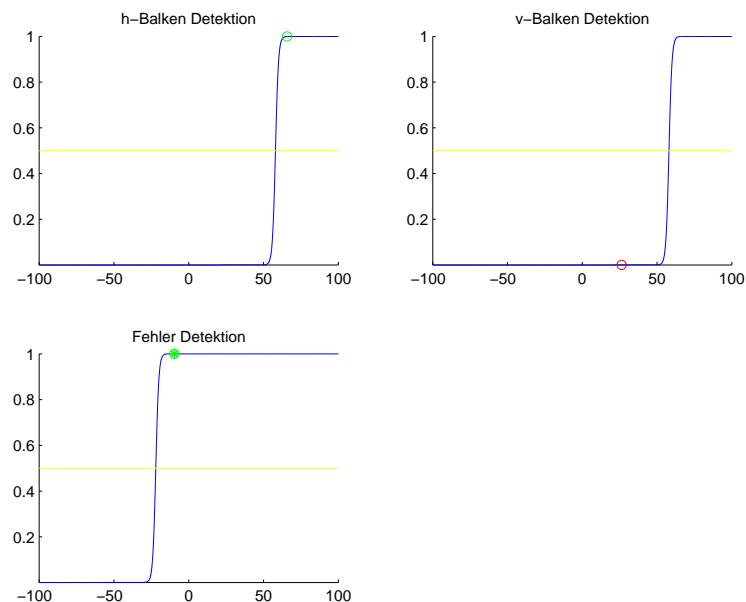


Abbildung 43: AddMul, H-Balken, 40% Rauschen, 2. Neuronen Ebene

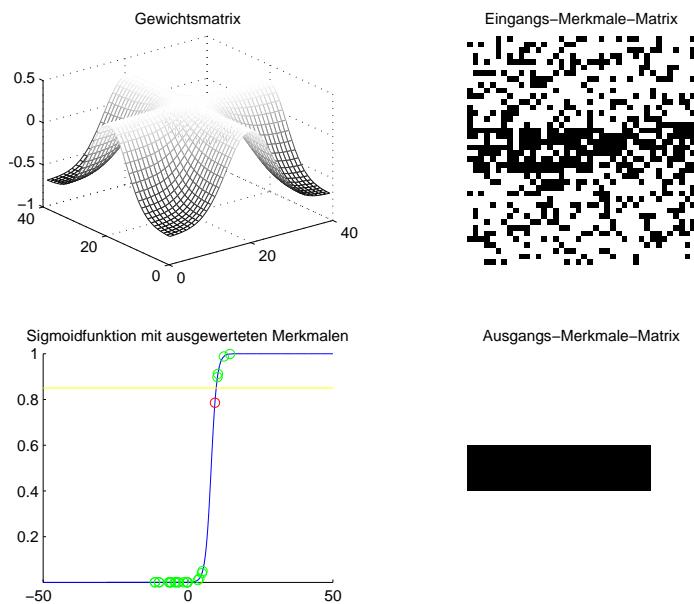


Abbildung 44: AddMul, H-Balken, 60% Rauschen, 1. Neuronen Ebene

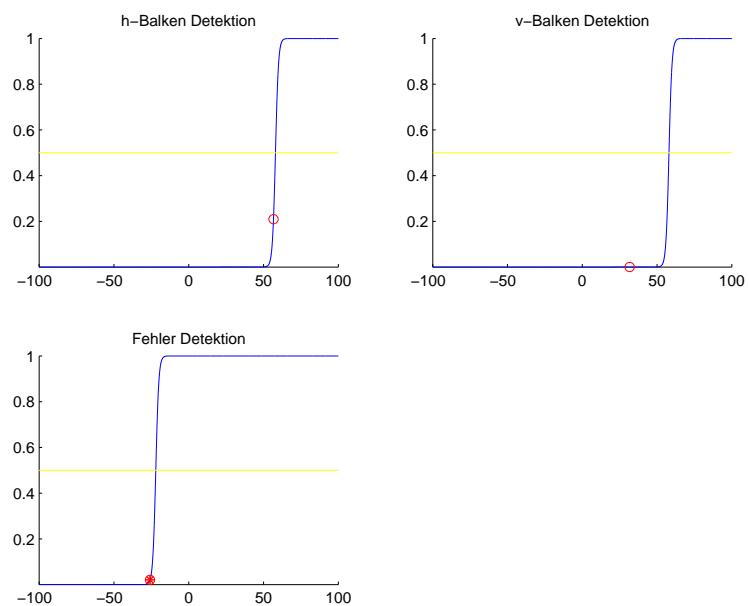


Abbildung 45: AddMul, H-Balken, 60% Rauschen, 2. Neuronen Ebene

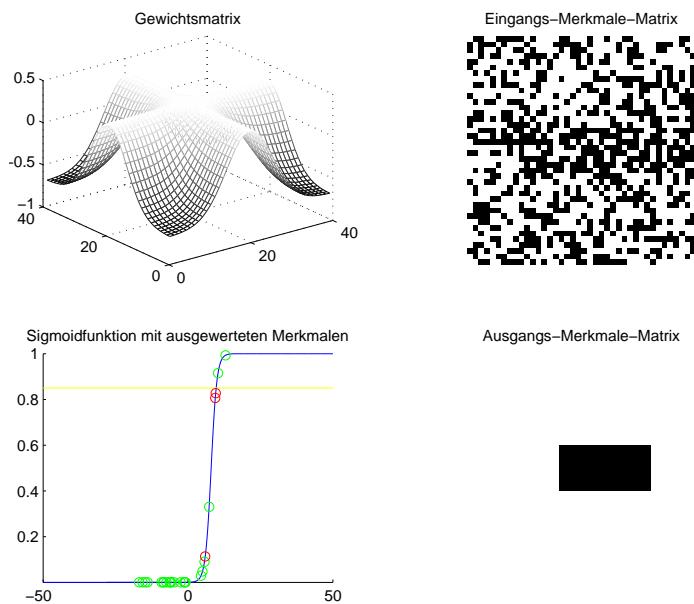


Abbildung 46: AddMul, H-Balken, 80% Rauschen, 1. Neuronen Ebene

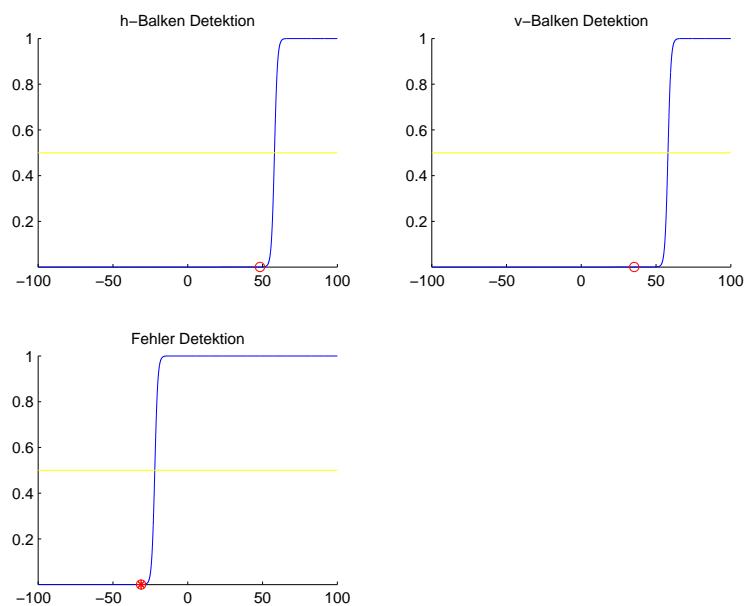


Abbildung 47: AddMul, H-Balken, 80% Rauschen, 2. Neuronen Ebene

### 6.1.2. V-Balken

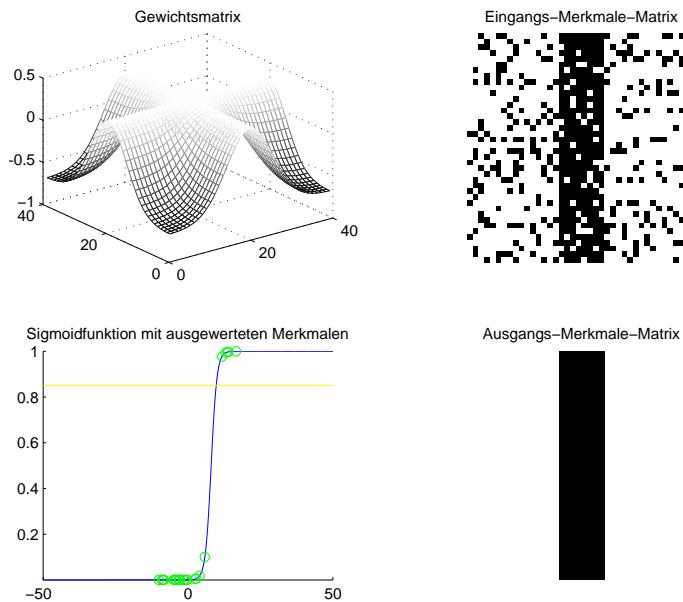


Abbildung 48: AddMul, V-Balken, 40% Rauschen, 1. Neuronen Ebene

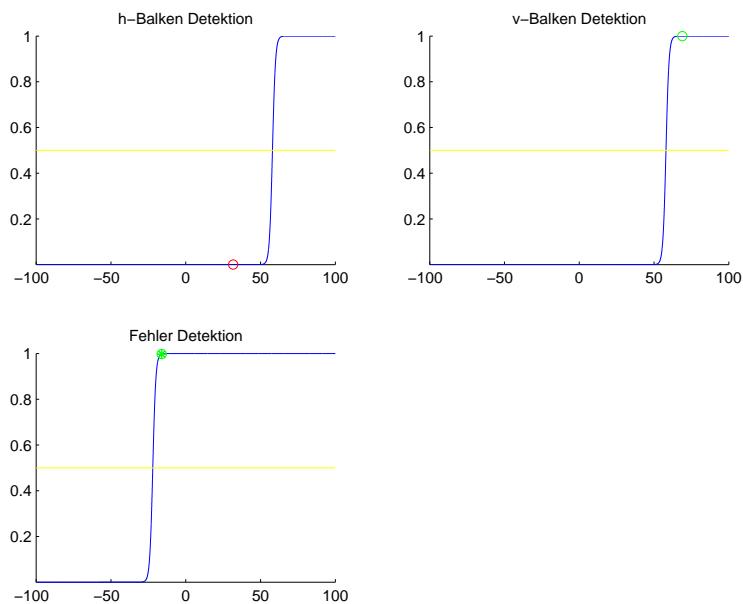


Abbildung 49: AddMul, V-Balken, 40% Rauschen, 2. Neuronen Ebene

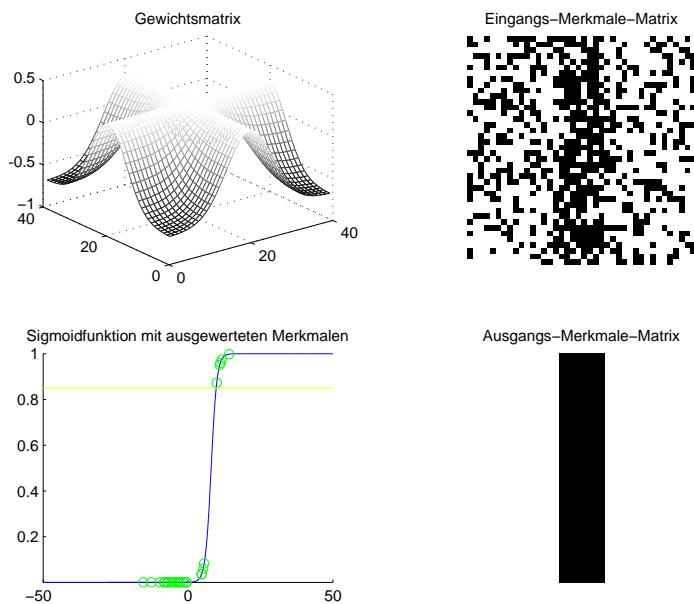


Abbildung 50: AddMul, V-Balken, 60% Rauschen, 1. Neuronen Ebene

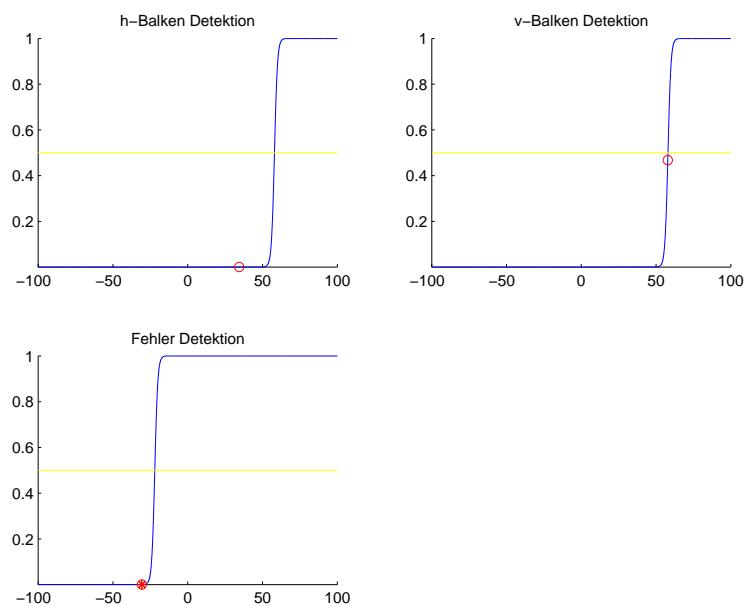


Abbildung 51: AddMul, V-Balken, 60% Rauschen, 2. Neuronen Ebene

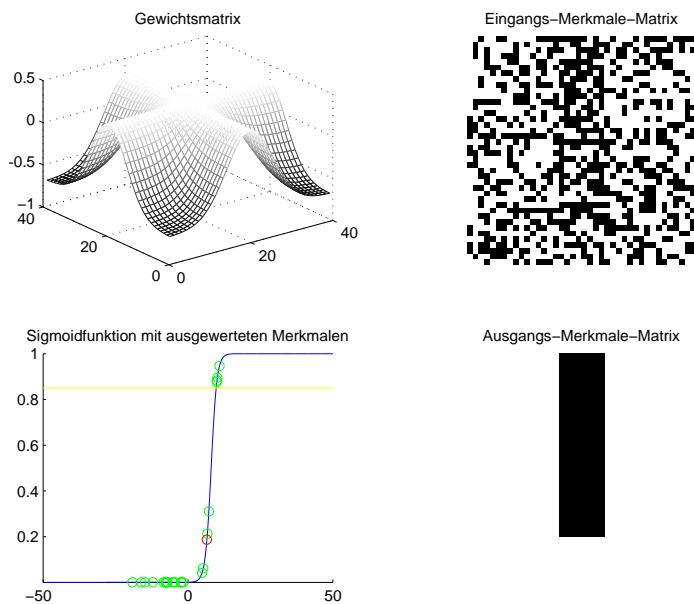


Abbildung 52: AddMul, V-Balken, 80% Rauschen, 1. Neuronen Ebene

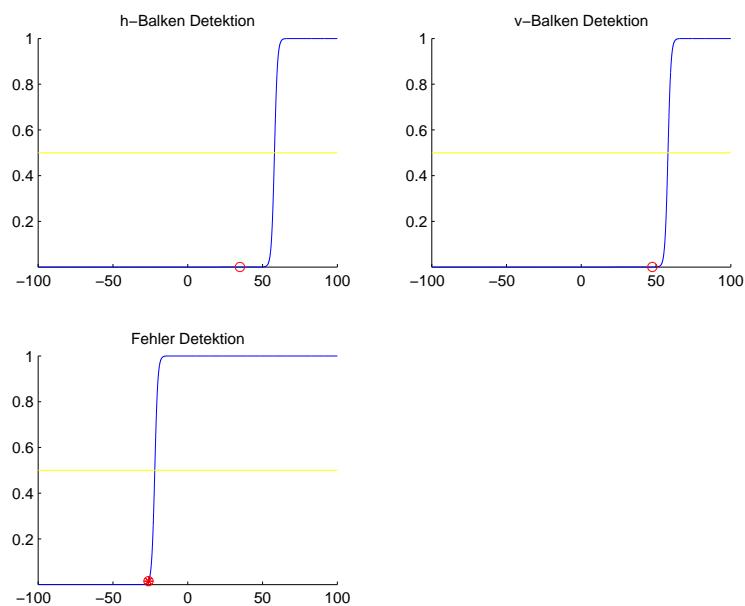


Abbildung 53: AddMul, V-Balken, 80% Rauschen, 2. Neuronen Ebene

### 6.1.3. Kreuz

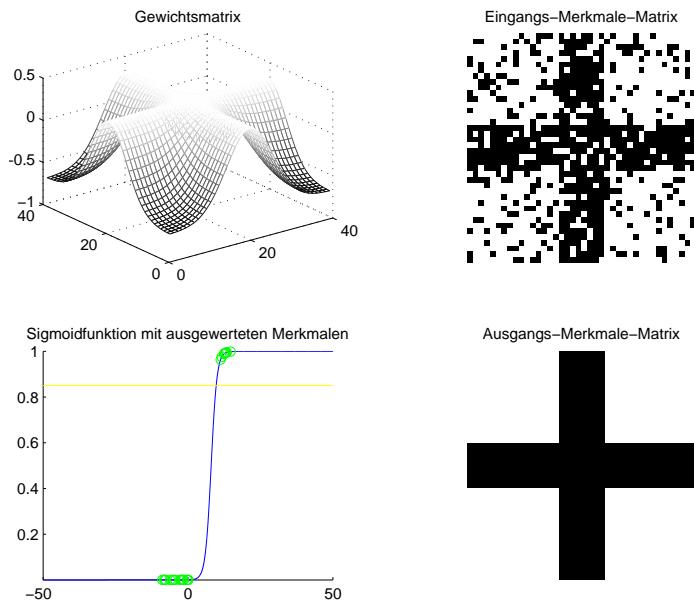


Abbildung 54: AddMul, Kreuz, 40% Rauschen, 1. Neuronen Ebene

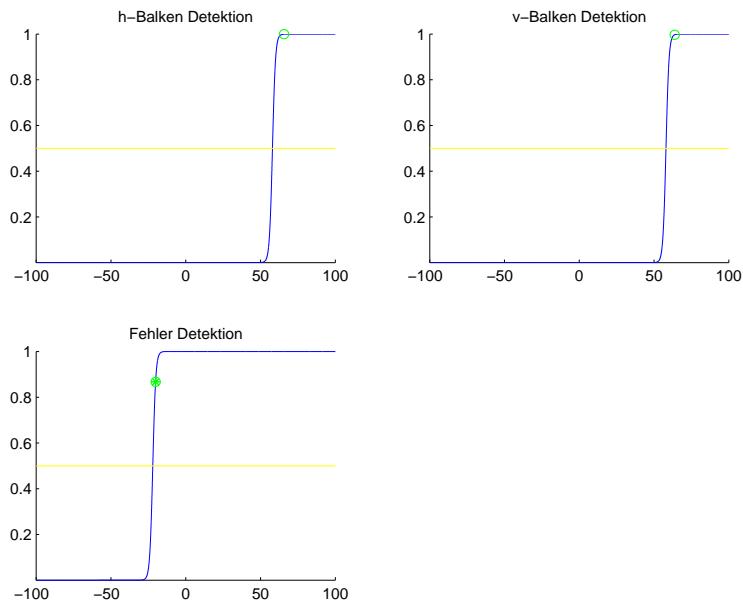


Abbildung 55: AddMul, Kreuz, 40% Rauschen, 2. Neuronen Ebene

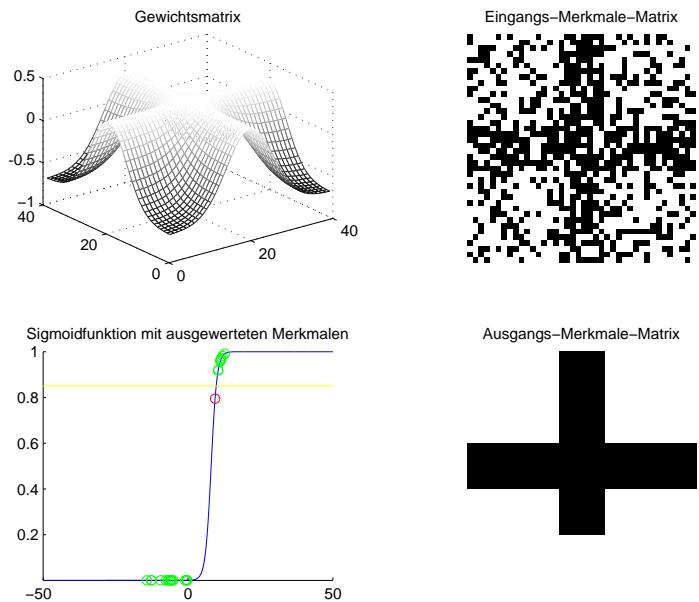


Abbildung 56: AddMul, Kreuz, 60% Rauschen, 1. Neuronen Ebene

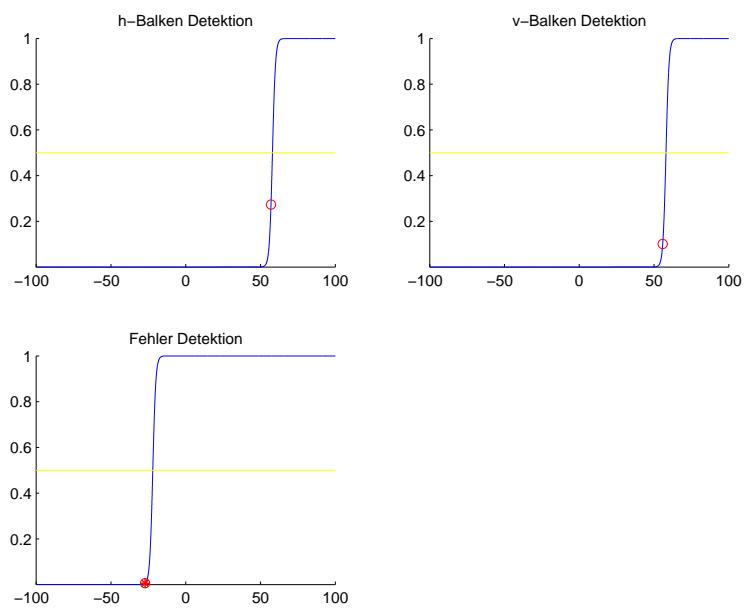


Abbildung 57: AddMul, Kreuz, 60% Rauschen, 2. Neuronen Ebene

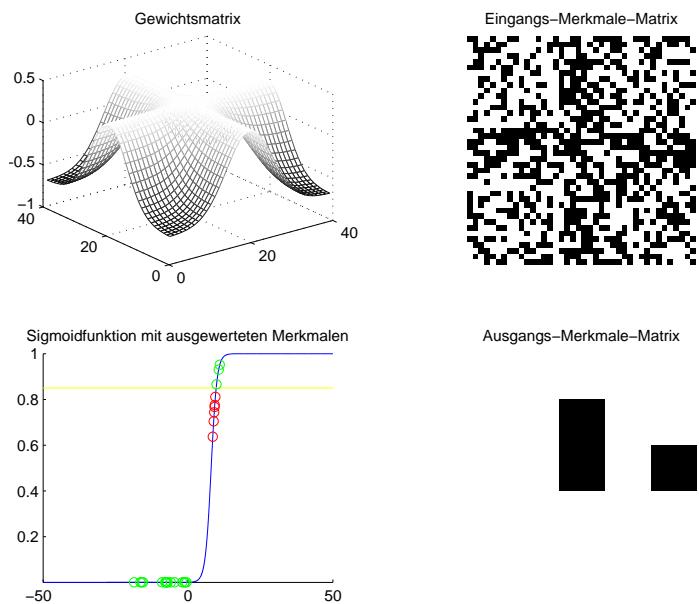


Abbildung 58: AddMul, Kreuz, 80% Rauschen, 1. Neuronen Ebene

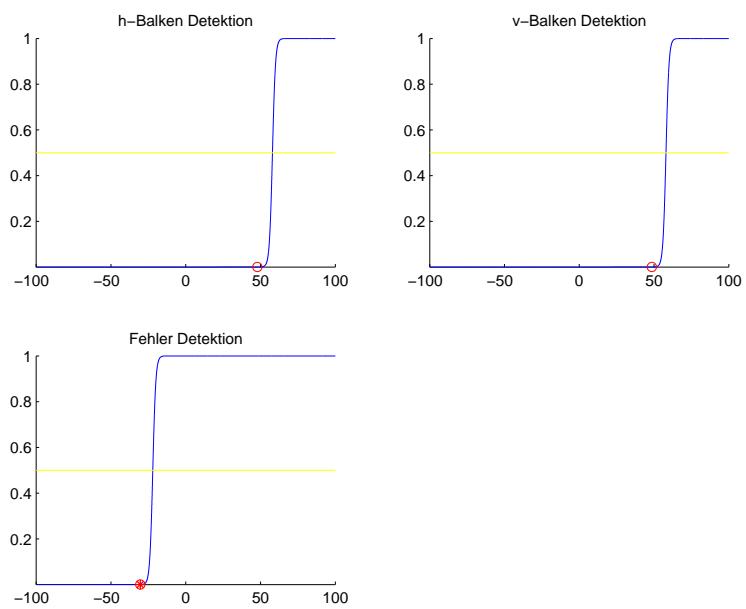


Abbildung 59: AddMul, Kreuz, 80% Rauschen, 2. Neuronen Ebene

## 6.2. Special Gewichtsmatrix

### 6.2.1. H-Balken

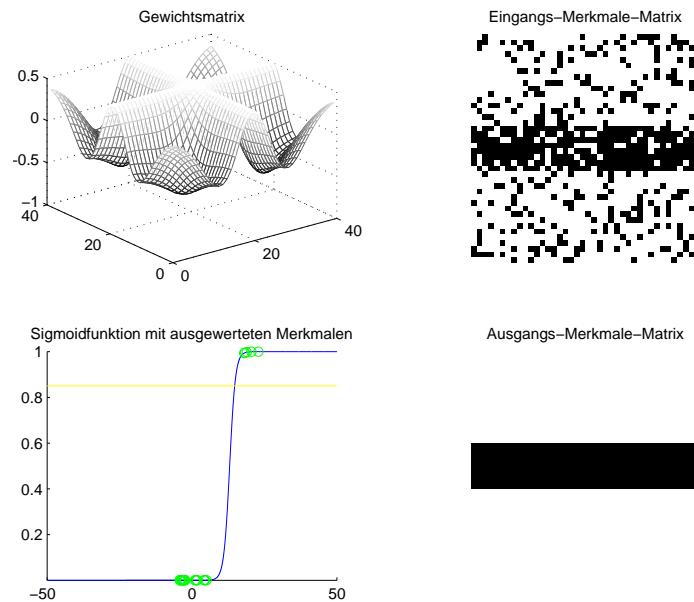


Abbildung 60: Special, H-Balken, 40% Rauschen, 1. Neuronen Ebene

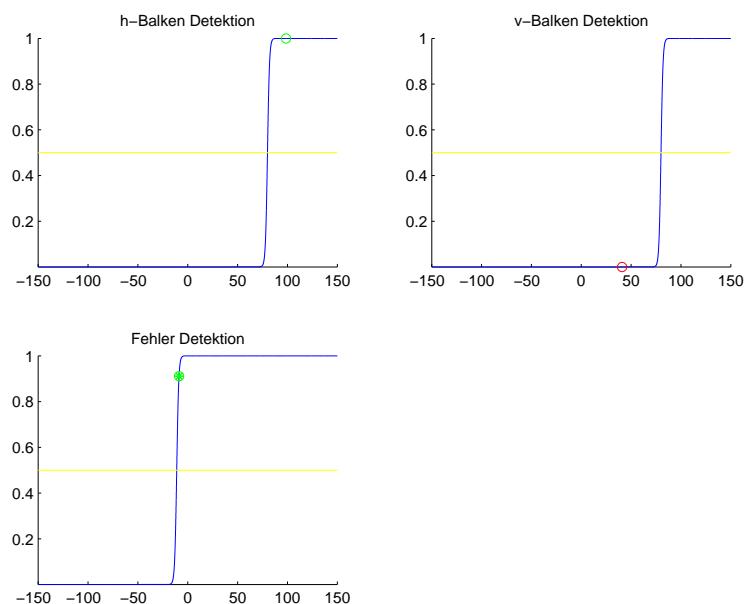


Abbildung 61: Special, H-Balken, 40% Rauschen, 2. Neuronen Ebene

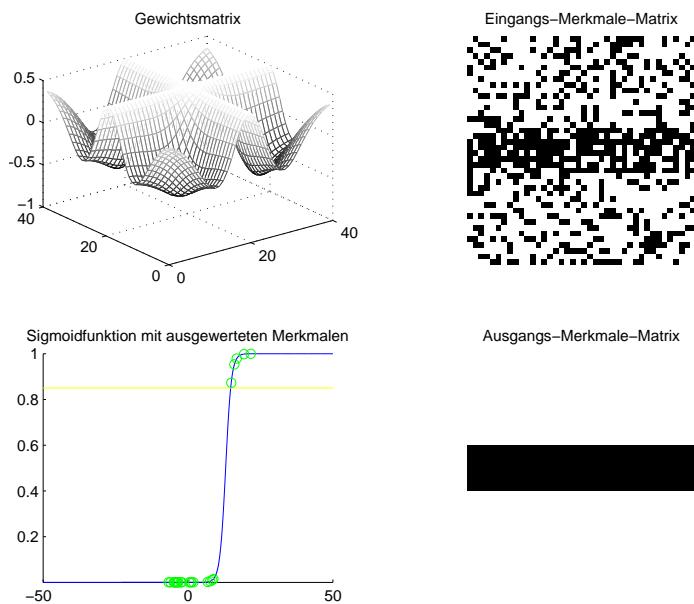


Abbildung 62: Special, H-Balken, 60% Rauschen, 1. Neuronen Ebene

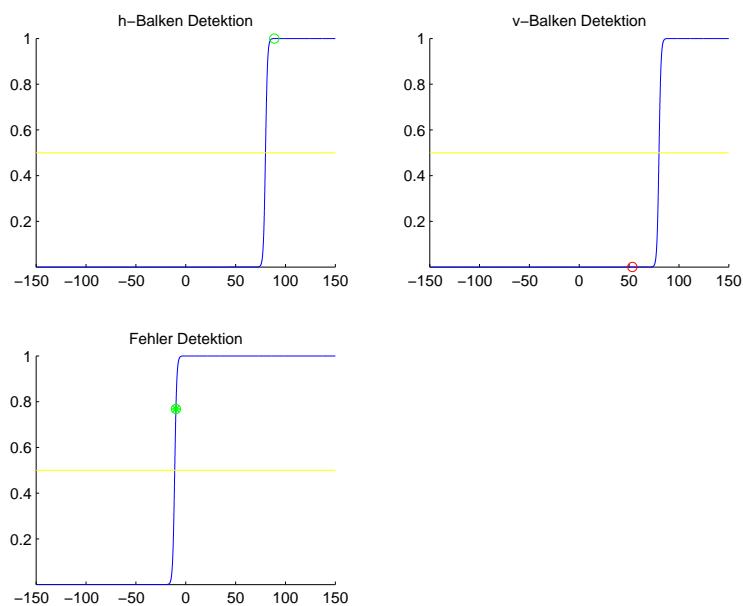


Abbildung 63: Special, H-Balken, 60% Rauschen, 2. Neuronen Ebene

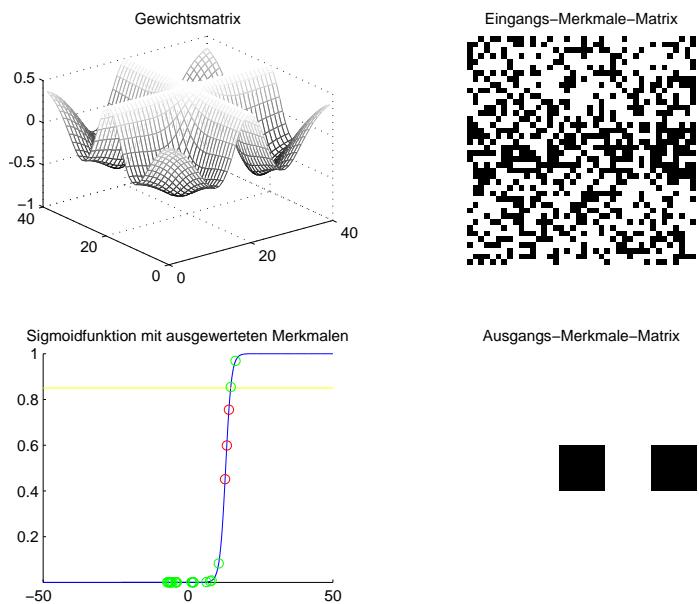


Abbildung 64: Special, H-Balken, 80% Rauschen, 1. Neuronen Ebene

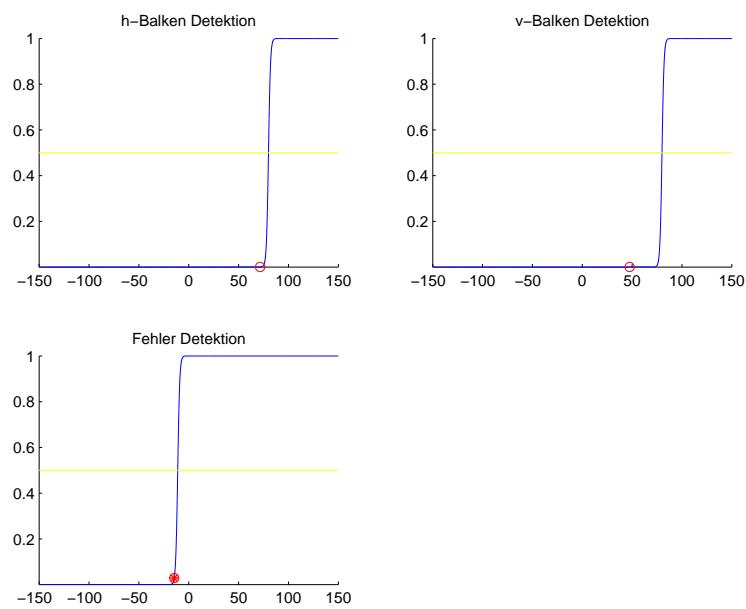


Abbildung 65: Special, H-Balken, 80% Rauschen, 2. Neuronen Ebene

### 6.2.2. V-Balken

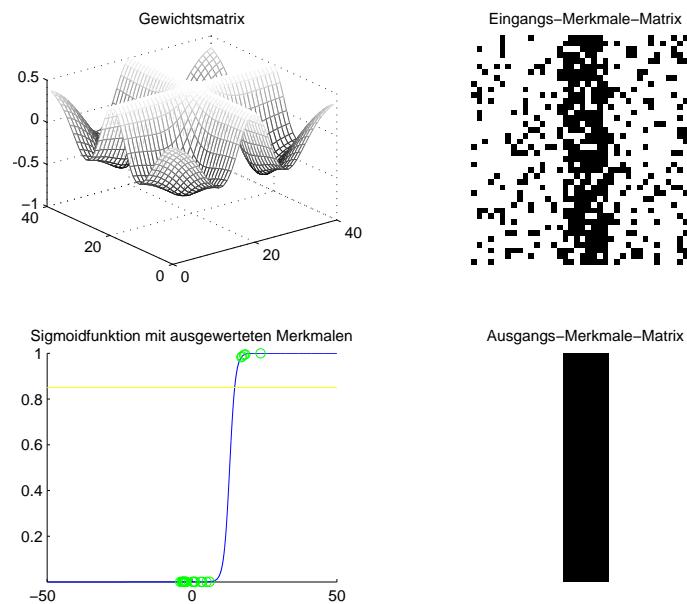


Abbildung 66: Special, V-Balken, 40% Rauschen, 1. Neuronen Ebene

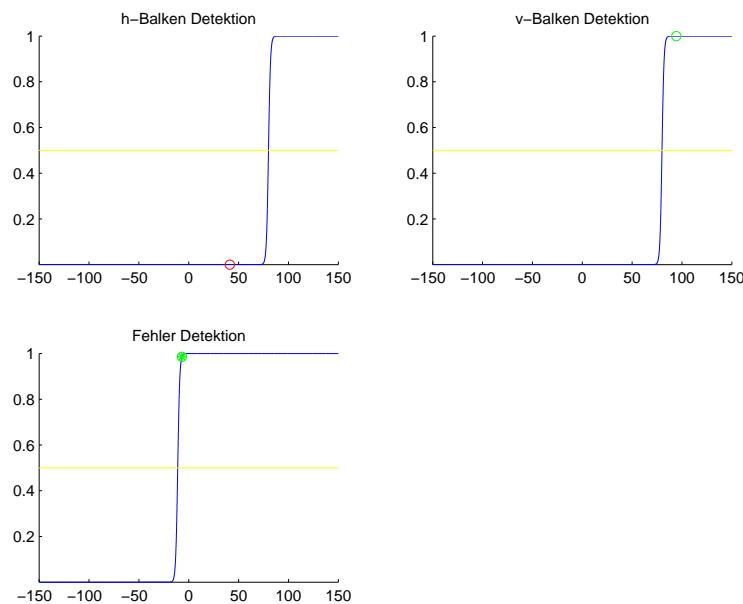


Abbildung 67: Special, V-Balken, 40% Rauschen, 2. Neuronen Ebene

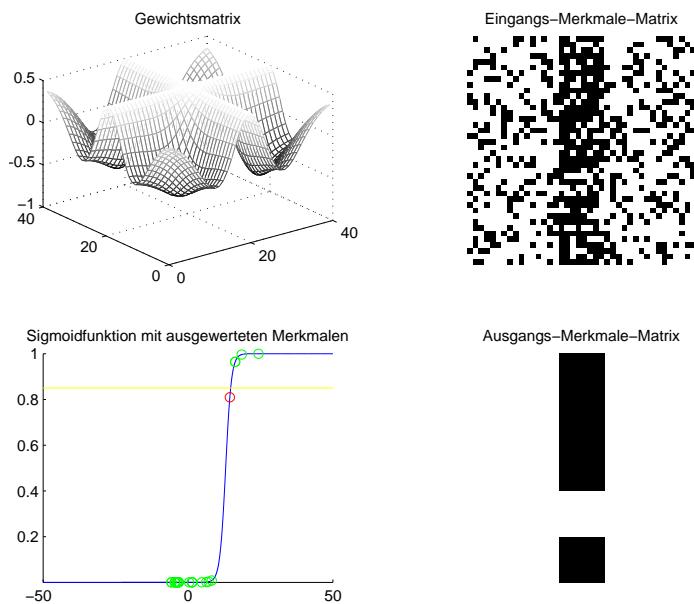


Abbildung 68: Special, V-Balken, 60% Rauschen, 1. Neuronen Ebene

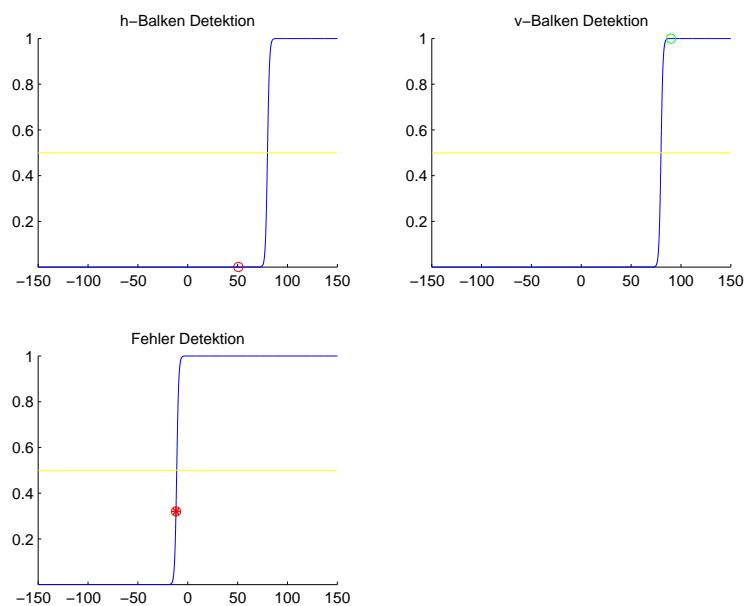


Abbildung 69: Special, V-Balken, 60% Rauschen, 2. Neuronen Ebene

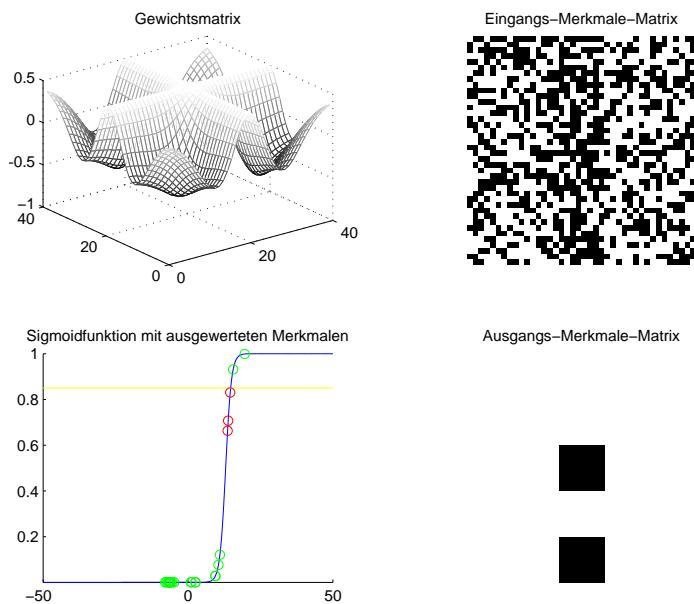


Abbildung 70: Special, V-Balken, 80% Rauschen, 1. Neuronen Ebene

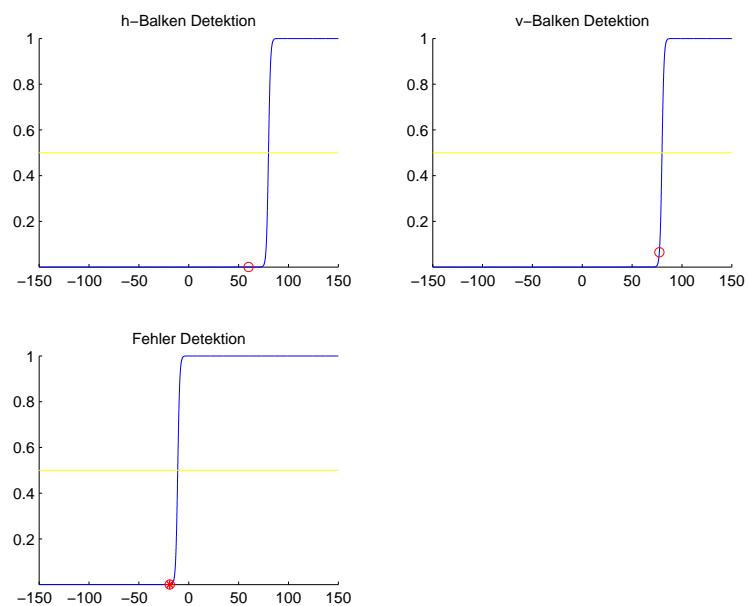


Abbildung 71: Special, V-Balken, 80% Rauschen, 2. Neuronen Ebene

### 6.2.3. Kreuz

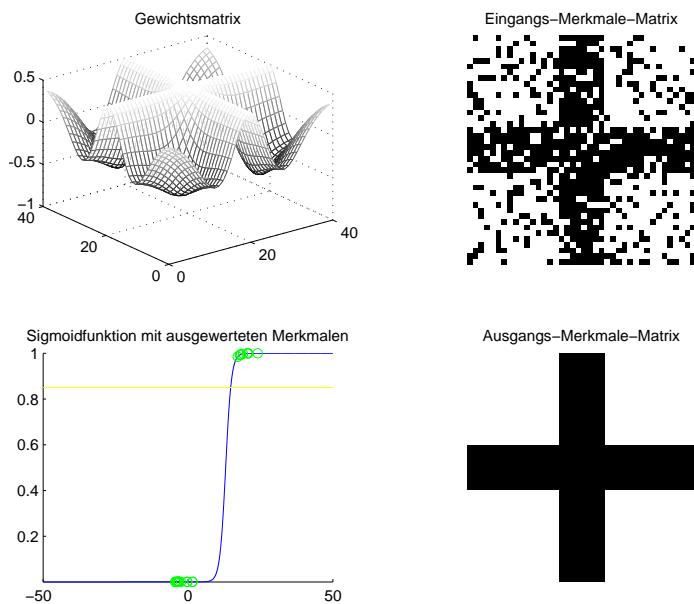


Abbildung 72: Special, Kreuz, 40% Rauschen, 1. Neuronen Ebene

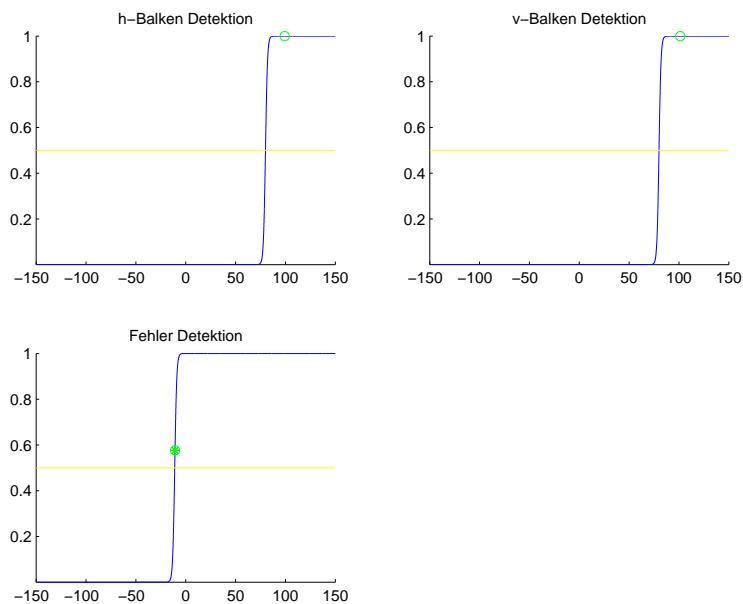


Abbildung 73: Special, Kreuz, 40% Rauschen, 2. Neuronen Ebene

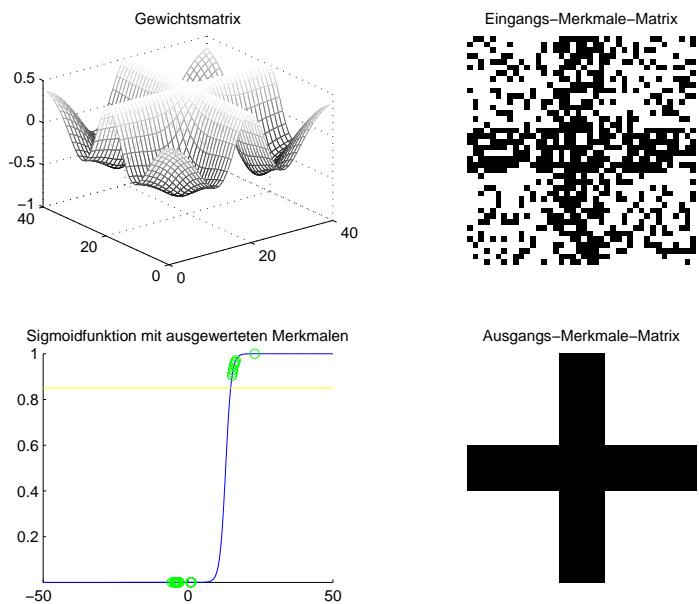


Abbildung 74: Special, Kreuz, 60% Rauschen, 1. Neuronen Ebene

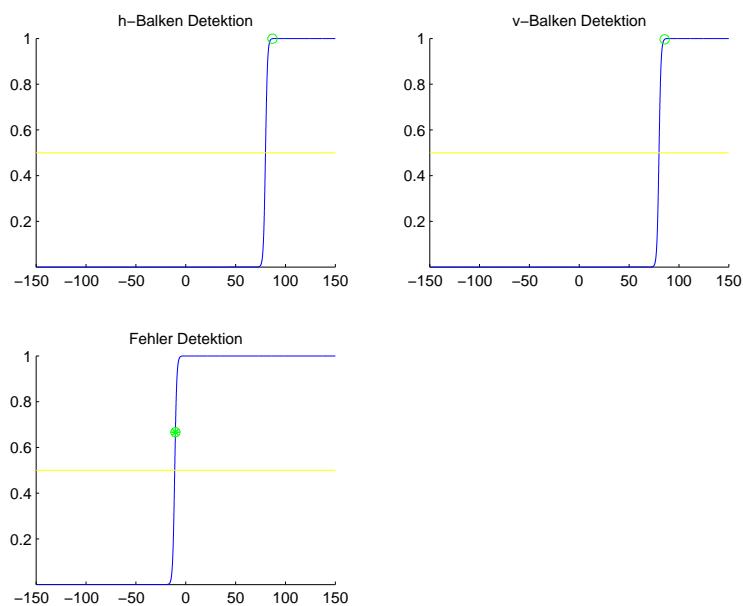


Abbildung 75: Special, Kreuz, 60% Rauschen, 2. Neuronen Ebene

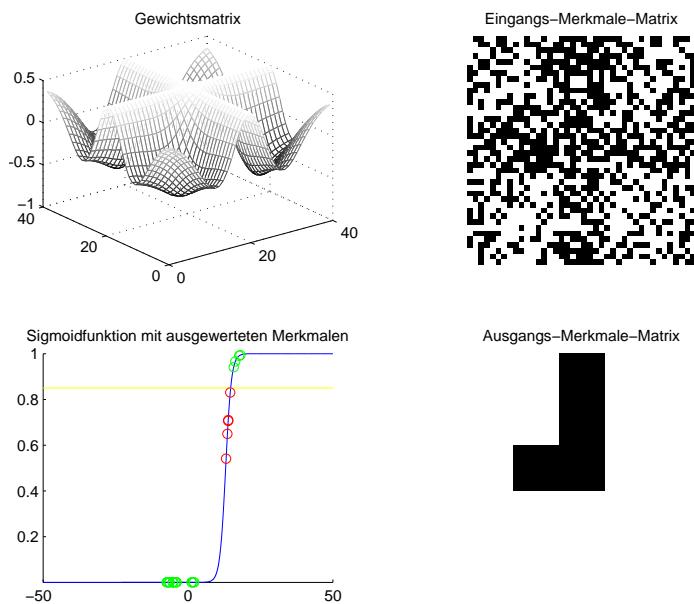


Abbildung 76: Special, Kreuz, 80% Rauschen, 1. Neuronen Ebene

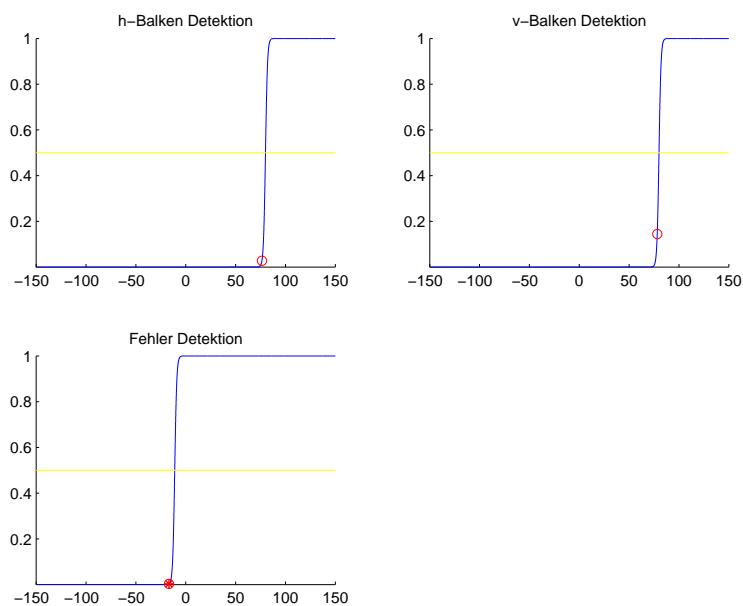


Abbildung 77: Special, Kreuz, 80% Rauschen, 2. Neuronen Ebene

## 7. Fazit

In diesem Projekt haben wir ein neuronales Netz zur Detektion von horizontalen und oder vertikalen Balken entwickelt. Das neuronale Netz besteht dabei aus zwei Neuronen Schichten. Die Anzahl der benötigten Neuronen hängt dabei von der Anzahl der Merkmale ab. Die Anzahl der Neuronen in der ersten Ebene wächst Linear mit der Anzahl der Merkmale. Die erste Neuronen Ebene fasst dabei alle Pixel zu den jeweiligen Merkmalen zusammen. Jedes Pixel wird dabei über die Gewichts-Matrix bewertet. In der zweiten Neuronen Ebene findet eine Verfeinerung des Ergebnisses statt. Erst die zweite Neuronen Ebene wertet die Informationen aus den Merkmalen aus.

Entwickelt haben wir verschiedene Möglichkeiten von Gewichts-Matrizen. Untersucht haben wir am Ende die zwei erfolgversprechendsten Gewichts-Matrizen 'AddMul' und 'Special'. 'AddMul' hatte dabei die maximale Möglichkeit Parameter einzustellen. Dies ist bei der Special-Gewichtsmatrix nicht möglich. Vergleicht man beide Typen von Gewichts-Matrizen fällt kaum ein Unterschied im Ergebnis auf. Aus diesem Grund ist für uns die AddMul-Gewichts-Matrix besser geeignet, weil sie einfacher aufgebaut und variabler ist. Bei der Special-Gewichts-Matrix konnte sich ein Vorteil der erhöhten Randbereiche nicht bestätigen. Durch die Erhöhung haben sich lediglich die Schwellen der Detektion verändert.

Um die richtige Konfiguration der verschiedenen Parameter zu finden, haben wir länger gebraucht als erwartet. Es hat sich teilweise als sehr schwierig erwiesen vernünftige Werte zu finden. Die Ergebnisse hängen sehr stark davon ab, wie viel Rauschen man in den Bildern hat. Erste Einstellungen des neuronalen Netzes haben in komplett verrauschten Bildern ein Kreuz detektiert. Durch sukzessive Anpassung der Schwellen haben wir diesen Problem in den Griff bekommen.

Ein weiteres Problem war es, die Merkmale der ersten Neuronen Ebene in zwei Gruppen zu unterteilen und damit separierbar zu machen. Jede Gruppe soll dabei möglichst zusammenhängend sein. Für das bessere Verständnis siehe Abbildung 78. In diesem Fall muss die Flanke der Sigmoid-Funktion zwischen den beiden Gruppen liegen. Die Verschiebung der Sigmoid-Funktion wird über den jeweiligen Bias-Parameter erreicht. Mit dem Threshold-Parameter wird die Schwelle festgelegt, ab wann ein aktives Merkmal detektiert werden soll. Werden die Parameter 'pixelCnt', 'featureCnt', 'lowerBound', 'upperBound' oder 'slope' geändert, dann müssen auch die Parameter 'bias', 'threshold' und 'domainOfDefinition' neu eingestellt werden.

Im Gegensatz zur ersten Ebene sind die richtigen Einstellungen für die zweite Ebene deutlich leichter zu finden, weil mal statt einer Sigmoid-Funktion drei Sigmoid-Funktionen zum Einstellen hat. Zur Erinnerung die zweite Neuronen Ebene erhält von der ersten Neuronen Ebene

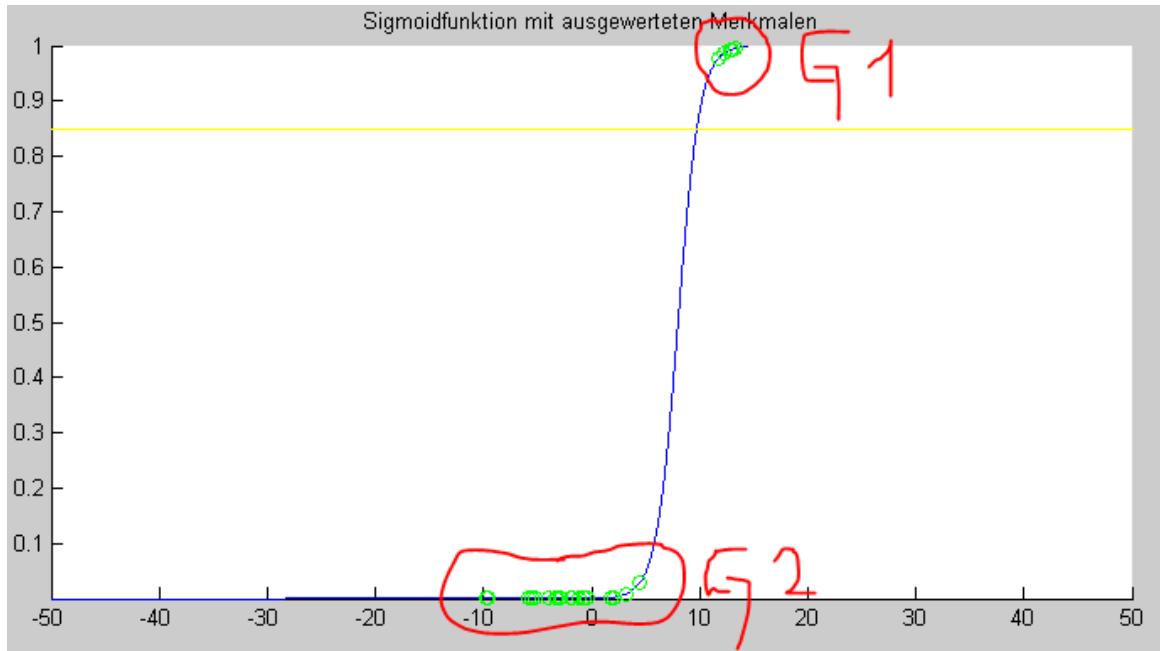


Abbildung 78: Merkmale-Gruppen

nicht die Werte der Sigmoid-Funktion, sondern die addierten Pixel-Gewichte ohne weitere Verarbeitung. Bei der zweiten Ebene können die Einstellungen für die Detektion des 'H-Balken', 'V-Balken' und des 'Fehlers' unabhängig voneinander eingestellt werden.

Mit dieser Arbeit haben wir einen Meilenstein gelegt, um ein Neuronales Netz in Hardware zu implementieren. Wir haben Erfahrungen gesammelt, wie ein selbst lernendes neuronales Netz die Gewichte findet und wie es aufgebaut sein muss, um vernünftige Ergebnisse zu liefern. Im Weiteren werden wir uns damit befassen unsere Ergebnisse in Hardware zu implementieren und weitere Funktionalitäten, wie zum Beispiel das selbstständige Lernen, zu ermöglichen.

# A. Appendix

Hier sind lediglich die wichtigsten Dateien aufgeführt, für den kompletten Umfang der Arbeit wird ein Blick in das Release empfohlen.

## A.1. GetPixelFeatureMatrix.m

```
1 function pixelFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, noise, featureMatrix, name)
2 % Funktion skaliert uebergebene Merkmale-Matrix in eine Pixel-Matrix
3 %
4 % pixelCnt - Anzahl der Pixel in x-Richtung pro Merkmal - mindestens 1
5 % featureCnt - Anzahl der Merkmale in x-Richtung - mindestens 1
6 % noise - Verrauschungsgrad zwischen 0 und 100%
7 % featureMatrix - Merkmale-Matrix, welche Bereiche aktiv sein sollen
8 % filename - optional - Name der Ausgabedatei
9
10 % Ueberpruefung der Merkmale-Matrix
11 sizeInput = size(featureMatrix);
12 if (sizeInput(1) ~= featureCnt)
13     error('featureCnt passt nicht zu featureMatrix')
14 end
15 clear sizeInput
16 if max(max(featureMatrix)) > 1
17     error('in the featureMatrix are only allowed 0 and 1')
18 end
19
20 % Ueberpruefung der maximal erlaubten Bildbreite
21 % es koennen nur uint16-Werte beruecksichtigt werden
22 if pixelCnt*featureCnt > 2^16-1
23     error('Pixelbreite des Bildes ist zu gross.')
24 end
25
26 % Ueberpruefung des erlaubten Noise-Level-Bereichs -> 0 bis 100
27 if ((noise > 100) && (0 < noise))
28     error('noise level is not allowed')
29 end
30 noise = uint8(noise/2); % 50 ist maximal verrauscht
31
32 % Umbenennungen fuer eventuelle Erweiterung in x- und y-Richtung
33 pixelCntX = pixelCnt; % Anzahl der Pixel pro Merkmal in x-Richtung
34 pixelCntY = pixelCnt; % Anzahl der Pixel pro Merkmal in y-Richtung
35 featureCntX = featureCnt; % Anzahl der Merkmale horizontal
36 featureCntY = featureCnt; % Anzahl der Merkmale vertikal
37
38 % Berechnen der maximalen Werte
39 pixelCntX_N = pixelCntX * featureCntX; % Zaehler in x-Richtung
40 pixelCntY_N = pixelCntY * featureCntY; % Zaehler in y-Richtung
41
42 % Erstellen der Pixel-Matrix und Rausch-Matrix
43 pixelFeatureMatrix = uint8(ones(pixelCntY_N, pixelCntX_N) .* 255);
44 noiseMatrix = uint8(randi([0 99], pixelCntY_N, pixelCntX_N));
45
46 % Umwandlung der Merkmale-Matrix in S/W-Bild
47 featureMatrix = uint8(featureMatrix);
48 featureMatrix = 255 - (featureMatrix .* 255);
49
50 % Befuellung der Pixel-Matrix
51 % Skalieren der Merkmale-Matrix auf Pixel-Matrix
52 for iPixY = 0:(pixelCntY_N - 1)
53     iFeatY = idivide(uint16(iPixY), pixelCntY, 'floor');
54     for iPixX = 0:(pixelCntX_N - 1)
55         iFeatX = idivide(uint16(iPixX), pixelCntX, 'floor');
56         pixelFeatureMatrix(iPixY + 1, iPixX + 1) = featureMatrix(iFeatY + 1, iFeatX + 1);
57     end
58 end
59 % Rauschen hinzufuegen
60 for iPixY = 1:pixelCntY_N
61     for iPixX = 1:pixelCntX_N
62         if (noiseMatrix(iPixY, iPixX) < noise)
63             pixelFeatureMatrix(iPixY, iPixX) = 255 - pixelFeatureMatrix(iPixY, iPixX);
64         end
65     end
66 end
```

```

65     end
66 end
67
68 % optional Bild speichern
69 if (~isempty(name))
70     imwrite(pixelFeatureMatrix, [name '.bmp']);
71 end
72 end

```

## A.2. GetInputFeatureMatrix.m

```

1 function inputFeatureMatrix = GetInputFeatureMatrix( featureCnt, type )
2 %GetInputFeatureMatrix Erzeugt die Eingangs-Merkmale-Matrix (featureCnt x featureCnt)
3 % featureCnt - Anzahl der Merkmale
4 % Art der Eingangs-Merkmale ((default)'Cross', 'V_Line', 'H_Line' & 'Cal')
5
6 if ( nargin < 2) % default ('Cross') vorgeben
7     type = 'Cross';
8     warning('Kein Merkmalstyp angegeben: Es wurde "Cross" als default eingestellt!');
9 end
10
11 if ( featureCnt > 2)
12     if (mod(featureCnt, 2) == 0) % Test ob featureCnt gerade ist
13         warning('Anzahl der Merkmale ist gerade und es werden die beiden innersten Merkmale gewaehlt!');
14         type = [type '_even']; % type um '_even' ergaenzen
15     end
16 else
17     error('Anzahl der Merkmale zu gering');
18 end
19
20 inputFeatureMatrix = zeros(featureCnt, featureCnt);
21
22 switch type % je nach type die Matrix mit 1 besetzen
23     case 'V_Line'
24         feature = ((featureCnt - 1)/2)+1;
25         inputFeatureMatrix(1:featureCnt, feature) = 1;
26     case 'V_Line_even'
27         feature = featureCnt/2;
28         inputFeatureMatrix(1:featureCnt, feature:(feature+1)) = 1;
29     case 'H_Line'
30         feature = ((featureCnt - 1)/2)+1;
31         inputFeatureMatrix(feature, 1:featureCnt) = 1;
32     case 'H_Line_even'
33         feature = featureCnt/2;
34         inputFeatureMatrix(feature:(feature+1), 1:featureCnt) = 1;
35     case 'Cross'
36         feature = ((featureCnt - 1)/2)+1;
37         inputFeatureMatrix(1:featureCnt, feature) = 1;
38         inputFeatureMatrix(feature, 1:featureCnt) = 1;
39     case 'Cross_even'
40         feature = featureCnt/2;
41         inputFeatureMatrix(1:featureCnt, feature:(feature+1)) = 1;
42         inputFeatureMatrix(feature:(feature+1), 1:featureCnt) = 1;
43     % alle Merkmale auf 1
44     case 'Cal'
45         inputFeatureMatrix(1:featureCnt, 1:featureCnt) = 1;
46     case 'Cal_even'
47         inputFeatureMatrix(1:featureCnt, 1:featureCnt) = 1;
48     otherwise
49         error('Unbekannter Merkmalstyp! Bitte aus "Cross", "V_Line", "H_Line" oder "Cal" waehlen');
50     end
51
52 inputFeatureMatrix = uint8(inputFeatureMatrix);
53
54 end

```

### A.3. GetGaussWeights.m

```

1 function [ weights , vWeightMatrix , hWeightMatrix ] = GetGaussWeights(pixelCnt , featureCnt , slope , type , lower , upper)
2 % weights - Rueckgabe der Gewichtsmatrix
3 % pixelCnt - Anzahl der Pixel pro Merkmal
4 % featureCnt - Anzahl der Merkmale
5 % slope - Steilheit der Fkt, Randrauschen unterdruecken 1 bis 100%
6 % type - Art wie die Gewichte erstellt werden - Mul, Add oder AddMul
7 % lower - Die untere Grenze der Gewichte (default = -1)
8 % upper - Die obere Grenze der Gewichte (default = 1)
9
10 if nargin == 4
11     lower = -1;
12     upper = 1;
13 elseif nargin == 5
14     error('Bitte zweite Grenze fuer die Gewichte angeben oder auf die Vorgabe von Gewichten verzichten');
15 end
16 if ((upper - lower) <= 0)
17     error('Reichweite der Gewichtsgrenzen ist negativ oder gleich null => GetWeights(..., upper, lower) vertauscht?')
18 end
19 if (slope < 1) || (slope > 100)
20     error('Rauschwert ist nicht erlaubt');
21 end
22
23 vN = pixelCnt * featureCnt;
24 hN = pixelCnt * featureCnt;
25 sigma = -0.0019*(slope - 1) + 0.2;
26 %GaussFunction = @(x, s, x0)(1/(sqrt(2*pi).*s))*exp(-((x-x0).^2)/(2.*s.^2));
27
28 % Einteilung bestimmen
29 v = linspace(-0.3,0.3,vN);
30 h = linspace(-0.3,0.3,hN);
31
32 % 2 Gaussfunktionen mit festem Sigma im Raum
33 % Sigma zwischen 0.01 und 0.2 waehlen
34 vGauss = GaussNormFunction(v, sigma, 0); % x-Achse, v-Balken
35 hGauss = GaussNormFunction(h, sigma, 0); % y-Achse, h-Balken
36
37 % initiale Gewichts-Matrix erstellen
38 vWeightMatrix = zeros(hN, vN);
39
40 % ersten Gauss in Gewichts-Matrix schreiben
41 for i = 1:hN
42     % auf 1 Normieren und in Matrix schreiben
43     vWeightMatrix(i, 1:end) = vGauss./max(vGauss);
44 end
45 if (nargout == 3)
46     hWeightMatrix = zeros(hN, vN);
47     for j = 1:hN
48         hWeightMatrix(1:end, j) = hGauss./max(hGauss);
49     end
50 end
51
52 % zweite temporaere Matrix erzeugen zur Ueberlagerung von (v_Gauss & h_Gauss)
53 tempWeightMatrix = zeros(hN, vN);
54
55 % fuer (type == MulAdd) zusaetzliche Matrix erzeugen
56 if strcmp(type , 'AddMul') || strcmp(type , 'AddMul2')
57     addMulWeightMatrix = zeros(hN, vN);
58 end
59
60 % Unterscheidung in for-Schleife je nach Art der Ueberlagerung (weightType)
61
62 if strcmp(type , 'Mul1')
63     for i = 1:vN
64         tempWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* (hGauss./max(hGauss))';
65         tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) .* 2 - 1;
66     end
67 elseif strcmp(type , 'Mul2')
68     for i = 1:vN
69         % skalieren auf -1 bis 1
70         vWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* 2 - 1;
71         % v-Gauss und h-Gauss multiplizieren
72         tempWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* ((hGauss./max(hGauss)) * 2 - 1)';
73     end
74 elseif strcmp(type , 'Add')
75     for i = 1:vN
76         tempWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) + (hGauss./max(hGauss))') ./ 2;
```

```

77      tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) .* 2 - 1;
78  end
79 elseif strcmp(type, 'AddMul')
80   for i = 1:vN
81     % v-Gauss und h-Gauss multiplizieren
82     addMulWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) .* (hGauss./max(hGauss)))' - 1;
83     % v-Gauss und h-Gauss addieren
84     tempWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) + (hGauss./max(hGauss)))' - 1;
85     % mittlere Erhöhung entfernen
86     tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) - addMulWeightMatrix(1:end, i);
87     % skalieren
88     tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) .* 2 - 1;
89   end
90 elseif strcmp(type, 'AddMul2')
91   for i = 1:vN
92     % skalieren auf -1 bis 1
93     vWeightMatrix(1:end, i) = vWeightMatrix(1:end, i) .* 2 - 1;
94     % v-Gauss und h-Gauss multiplizieren
95     addMulWeightMatrix(1:end, i) = (vWeightMatrix(1:end, i) .* ((hGauss./max(hGauss)) * 2 - 1))' - 1;
96     % v-Gauss und h-Gauss addieren
97     %tempWeightMatrix(i:end, i) = (vWeightMatrix(i:end, i) + ((hGauss./max(hGauss)) * 2 - 1))' - 1;
98     % mittlere Erhöhung entfernen
99     %tempWeightMatrix(i:end, i) = tempWeightMatrix(i:end, i) - addMulWeightMatrix(i:end, i);
100    % skalieren
101    %tempWeightMatrix(i:end, i) = tempWeightMatrix(i:end, i) .* 2 - 1;
102    tempWeightMatrix(1:end, i) = addMulWeightMatrix(1:end, i);
103  end
104 elseif strcmp(type, 'Special')
105   weightMatrix1 = GetGaussWeights(pixelCnt, featureCnt, 45, 'Mul2', -2, 2);
106   weightMatrix2 = GetGaussWeights(pixelCnt, featureCnt, 70, 'AddMul', -2, 2);
107   weightMatrix3 = GetGaussWeights(pixelCnt, featureCnt, 45, 'Mul1', -2, 2);
108   tempWeightMatrix = weightMatrix1 + weightMatrix2 - weightMatrix3 - 1;
109 else
110   error('Weighttype for generation unknown, use Mul, Add or AddMul')
111 end
112
113 if(lower == 0 && upper == 1)
114   tempWeightMatrix = tempWeightMatrix + 1;
115   tempWeightMatrix = tempWeightMatrix ./ 2;
116 elseif(not(lower == -1 && upper == 1)) % fuer andere Grenzen als 0..1 oder -1..1
117   if(-lower == upper) % fuer symmetrische Grenzen um Null (e.g. -2..2)
118     tempWeightMatrix = tempWeightMatrix.*upper;
119   else % auf 0..2 verschieben und anschliessend mit (half_range) skalieren und verschieben
120     tempWeightMatrix = tempWeightMatrix + 1;
121     halfRange = ((upper - lower)/2);
122     tempWeightMatrix = tempWeightMatrix .* halfRange;
123     tempWeightMatrix = tempWeightMatrix + lower;
124   end
125 end
126
127 weights = tempWeightMatrix;
128
129 end

```

## A.4. main

```

1 close all, clear all
2
3 %% Parameter
4 % Parameter fuer
5 pixelCnt = 8; % Anzahl der Pixel in x-Richtung pro Merkmal - mindestens 1
6 featureCnt = 5; % Anzahl der Merkmale in x-Richtung - mindestens 1
7 weightType = 'Special'; % Typ der Gewichtsmatrix ('Add', 'AddMul' & 'Mul')
8 inFeatureType = 'Cross'; % Arten der Eingangs-Merkmale-Matrix ((default)'Cross', 'V_Line', 'H_Line' & 'Cal')
9 noise = 50; % Verrauschungsgrad zwischen 0 und 100%
10 slope = 30; % Steigung der Aktivierungs-Funktion (gauss) [50]
11
12 % Parameter fuer Aktivierungsfunktion
13 bias = -13; % Verschiebung in x-Richtung -> Neg (rechts), Pos (links)
14 threshold = 0.35; % Auswertungsschwelle des Ergebnisses
15 domainOfDefinition = 50; % Gueltigkeitsbereich der Neuronenfunktion -> (+/- domainOfDefinition)
16
17 % Parameter fuer Gewichtsmatrix
18 lowerBound = -0.5; % (optional) Untere Grenze der Gewichts-Matrix (default = -1)
19 upperBound = 0.5; % (optional) Obere Grenze der Gewichts-Matrix (default = 1)
20
21 %% Grundeinstellungen
22 % Erstellen der Eingangs-Merkmale-Matrix
23 inputFeatureMatrix = GetInputFeatureMatrix(featureCnt, inFeatureType);
24
25 I1 = [1 1 1 1 1]; % Zeile 1
26 I2 = [1 1 1 1 1]; % Zeile 2
27 I3 = [1 1 1 1 1]; % Zeile 3
28 I4 = [1 1 1 1 1]; % Zeile 4
29 I5 = [1 1 1 1 1]; % Zeile 5
30 % inputFeatureMatrix = uint8([I1; I2; I3; I4; I5]);
31
32 % Erstellen der Ausgangs-Merkmale-Matrix
33 outputFeatureMatrix = zeros(5, 5); % Erstelle Merkmale-Ausgangs-Matrix
34 outputFeatureMatrixDebug = zeros(5,5); % Erstelle Merkmale-Ausgangs-Matrix mit Summe aus Pixeln pro Merkmal
35 outputFeatureMatrixDebug2 = zeros(5,5); % Erstelle Merkmale-Ausgangs-Matrix mit Summe aus Pixeln pro Merkmal
36
37 % Erstellen der Gewichts-Matrix
38 weightMatrix = GetGaussWeights(pixelCnt, featureCnt, slope, weightType, lowerBound, upperBound);
39 % Pixel, Merkmale, Stielheit, Art, (untere, obere Grenze)
40
41
42 %% Erstelle Plot der Gewichts-Matrix
43 figure
44 hold on
45 subplot(2,2,1)
46 mesh(weightMatrix)
47 title('Gewichtsmatrix')
48
49 %% Erstelle Plot der Eingangs-Merkmale-Matrix
50 subplot(2,2,2)
51 inputPixelFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, noise, inputFeatureMatrix, ',');
52 imshow(inputPixelFeatureMatrix)
53 title('Eingangs-Merkmale-Matrix')
54
55 inputMatrix = (255 - inputPixelFeatureMatrix)/255;
56
57 %% Erstelle Plot der Sigmoidfunktion
58 subplot(2,2,3)
59 hold on
60 x = -domainOfDefinition:0.01:domainOfDefinition;
61 y = SigmoidFunction(x, bias);
62 plot(x,y)
63 % line for bias
64 %line([-domainOfDefinition domainOfDefinition],[SigmoidFunction(0, bias) SigmoidFunction(0, bias)], 'color', 'y')
65 % line for threshold
66 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
67 axis([x(1) x(end) min(y) max(y)])
68 title('Sigmoidfunktion mit ausgewerteten Merkmalen')
69
70 %% Erste Neuronen Ebene - Iteration ueber alle Merkmale
71 for yi=1:1:(featureCnt)
72     for xi=1:1:(pixelCnt)
73         % Bestimmung der Gewichte in Form eines Spaltenvektors
74         weights = GetFeatureOfMatrix(weightMatrix, xi, yi, pixelCnt, featureCnt+1);
75         weights = ConvMatrixToColumn(weights);
76

```

```

77 % Bestimmung der Eingaenge in Form eines Spaltenvektors
78 inputs = GetFeatureOfMatrix(inputMatrix, xi, yi, pixelCnt, featureCnt+1);
79 inputs = ConvMatrixToColumn(inputs);
80
81 % Berechne Neuronenausgang
82 [neuronNetTerms, neuronOutput] = GetNeuronOutput(inputs, weights, bias, domainOfDefinition, 'sigmoid');
83
84 % Werte Teilergebnis aus und trage es in Ausgang-Merkmale-Matrix ein
85 if neuronOutput > threshold
86     outputFeatureMatrix(yi, xi) = 1;
87 else
88     outputFeatureMatrix(yi, xi) = 0;
89 end
90
91 % Zeichne Merkmal in Sigmoidfunktion ein
92 hold on
93 subplot(2,2,3)
94 if (sum(neuronNetTerms) == 0)
95     % Alle Eingangswerte des Neurons sind 0
96     plot(sum(neuronNetTerms), neuronOutput, '*', 'color', 'b')
97 elseif outputFeatureMatrix(yi, xi) == inputFeatureMatrix(yi, xi)
98     % Merkmal richtig detektiert
99     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
100 else
101     % Merkmal falsch detektiert
102     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
103 end
104
105 % zum Debugging
106 outputFeatureMatrixDebug(yi, xi) = sum(neuronNetTerms);
107 outputFeatureMatrixDebug2(yi, xi) = neuronOutput;
108 end
109 end
110
111 %% Erstelle Plot der Ausgangs-Merkmale-Matrix
112 subplot(2,2,4)
113 outputFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, 0, outputFeatureMatrix, '');
114 imshow(outputFeatureMatrix)
115 title('Ausgangs-Merkmale-Matrix')
116
117 %% Debug-Ausgabe
118
119 outputFeatureMatrixDebug(1:end, 1:end)
120 outputFeatureMatrixDebug2(1:end, 1:end)
121
122 figure
123 %% Zweite Neuronen Ebene - Auswertung h-Balken
124 % Parameter
125 domainOfDefinition = 150;
126 bias = -60;
127 threshold = 0.5;
128 row = 3;
129
130 [neuronNetTerms, neuronOutput] = GetNeuronOutput(outputFeatureMatrixDebug(row, 1:end)', ones(1, featureCnt)', bias, domainOfDefinition);
131
132 % Erstelle Plot der Sigmoidfunktion
133 subplot(2,2,1)
134 hold on
135 x = -domainOfDefinition:0.01:domainOfDefinition;
136 y = SigmoidFunction(x, bias);
137 plot(x,y)
138 % line for threshold
139 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
140 axis([x(1) x(end) min(y) max(y)])
141 title('h-Balken Detektion')
142
143 if (neuronOutput > threshold)
144     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
145 else
146     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
147 end
148
149 %% Zweite Neuronen Ebene - Auswertung v-Balken
150 % Parameter
151 domainOfDefinition = 150;
152 bias = -60;
153 threshold = 0.5;
154 column = 3;

```

```

155
156 [neuronNetTerms, neuronOutput] = GetNeuronOutput(outputFeatureMatrixDebug(1:end, column), ones(featureCnt, 1), bias, domain
157
158 % Erstelle Plot der Sigmoidfunktion
159 subplot(2,2,2)
160 hold on
161 x = -domainOfDefinition:0.01:domainOfDefinition;
162 y = SigmoidFunction(x, bias);
163 plot(x,y)
164 % line for threshold
165 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
166 axis([x(1) x(end) min(y) max(y)])
167 title('v-Balken Detektion')
168
169 if (neuronOutput > threshold)
170 plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
171 else
172 plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
173 end
174
175 %% Zweite Neuronen Ebene - Auswertung Error-Field - Nicht generisch
176 % Parameter
177 domainOfDefinition = 150;
178 bias = 11;
179 threshold = 0.5;
180
181 errorField1 = ConvMatrixToColumn(outputFeatureMatrixDebug(1:2, 1:2));
182 errorField2 = ConvMatrixToColumn(outputFeatureMatrixDebug(4:5, 1:2));
183 errorField3 = ConvMatrixToColumn(outputFeatureMatrixDebug(1:2, 4:5));
184 errorField4 = ConvMatrixToColumn(outputFeatureMatrixDebug(4:5, 4:5));
185 sz = size(errorField1);
186
187 [neuronNetTerms1, neuronOutput1] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
188 [neuronNetTerms2, neuronOutput2] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
189 [neuronNetTerms3, neuronOutput3] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
190 [neuronNetTerms4, neuronOutput4] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
191
192 % Erstelle Plot der Sigmoidfunktion
193 subplot(2,2,3)
194 hold on
195 x = -domainOfDefinition:0.01:domainOfDefinition;
196 y = SigmoidFunction(x, bias);
197 plot(x,y)
198 % line for threshold
199 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
200 axis([x(1) x(end) min(y) max(y)])
201 title('Fehler Detektion')
202
203 if (neuronOutput1 < threshold)
204 plot(sum(neuronNetTerms1), neuronOutput1, 'o', 'color', 'r')
205 else
206 plot(sum(neuronNetTerms1), neuronOutput1, 'o', 'color', 'g')
207 end
208 if (neuronOutput2 < threshold)
209 plot(sum(neuronNetTerms2), neuronOutput2, '*', 'color', 'r')
210 else
211 plot(sum(neuronNetTerms2), neuronOutput2, '*', 'color', 'g')
212 end
213 if (neuronOutput3 < threshold)
214 plot(sum(neuronNetTerms3), neuronOutput3, 'x', 'color', 'r')
215 else
216 plot(sum(neuronNetTerms3), neuronOutput3, 'x', 'color', 'g')
217 end
218 if (neuronOutput4 < threshold)
219 plot(sum(neuronNetTerms4), neuronOutput4, '+', 'color', 'r')
220 else
221 plot(sum(neuronNetTerms4), neuronOutput4, '+', 'color', 'g')
222 end

```

## A.5. SettingFileAddMul

```

1 close all, clear all
2
3 %% Parameter
4 % Parameter fuer
5 pixelCnt = 8; % Anzahl der Pixel in x-Richtung pro Merkmal - mindestens 1
6 featureCnt = 5; % Anzahl der Merkmale in x-Richtung - mindestens 1
7 weightType = 'AddMul'; % Typ der Gewichtsmatrix ('Add', 'AddMul' & 'Mul')
8 inFeatureType = 'V_Line'; % Arten der Eingangs-Merkmale-Matrix ((default)'Cross', 'V_Line', 'H_Line' & 'Cal')
9 noise = 40; % Verrauschungsgrad zwischen 0 und 100%
10 slope = 60; % Steigung der Aktivierungs-Funktion (gauss) [50]
11
12 % Parameter fuer Aktivierungsfunktion
13 bias = -8; % Verschiebung in x-Richtung -> Neg (rechts), Pos (links)
14 threshold = 0.85; % Auswertungsschwelle des Ergebnisses
15 domainOfDefinition = 50; % Gueltigkeitsbereich der Neuronenfunktion -> (+/- domainOfDefinition)
16
17 % Parameter fuer Gewichtsmatrix
18 lowerBound = -0.7; % (optional) Untere Grenze der Gewichts-Matrix (default = -1)
19 upperBound = 0.3; % (optional) Obere Grenze der Gewichts-Matrix (default = 1)
20
21 %% Grundeinstellungen
22 % Erstellen der Eingangs-Merkmale-Matrix
23 inputFeatureMatrix = GetInputFeatureMatrix(featureCnt, inFeatureType);
24
25 I1 = [1 1 1 1 1]; % Zeile 1
26 I2 = [1 1 1 1 1]; % Zeile 2
27 I3 = [1 1 1 1 1]; % Zeile 3
28 I4 = [1 1 1 1 1]; % Zeile 4
29 I5 = [1 1 1 1 1]; % Zeile 5
30 % inputFeatureMatrix = uint8([I1; I2; I3; I4; I5]);
31
32 % Erstellen der Ausgangs-Merkmale-Matrix
33 outputFeatureMatrix = zeros(5, 5); % Erstelle Merkmale-Ausgangs-Matrix
34 outputFeatureMatrixDebug = zeros(5,5); % Erstelle Merkmale-Ausgangs-Matrix mit Summe aus Pixeln pro Merkmal
35 outputFeatureMatrixDebug2 = zeros(5,5); % Erstelle Merkmale-Ausgangs-Matrix mit Summe aus Pixeln pro Merkmal
36
37 % Erstellen der Gewichts-Matrix
38 weightMatrix = GetGaussWeights(pixelCnt, featureCnt, slope, weightType, lowerBound, upperBound);
39 % Pixel, Merkmale, Stielheit, Art, (untere, obere Grenze)
40
41
42 %% Erstelle Plot der Gewichts-Matrix
43 bild1 = figure;
44 hold on
45 subplot(2,2,1)
46 mesh(weightMatrix)
47 title('Gewichtsmatrix')
48
49 %% Erstelle Plot der Eingangs-Merkmale-Matrix
50 subplot(2,2,2)
51 inputPixelFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, noise, inputFeatureMatrix, '');
52 imshow(inputPixelFeatureMatrix)
53 title('Eingangs-Merkmale-Matrix')
54
55 inputMatrix = (255 - inputPixelFeatureMatrix)/255;
56
57 %% Erstelle Plot der Sigmoidfunktion
58 subplot(2,2,3)
59 hold on
60 x = -domainOfDefinition:0.01:domainOfDefinition;
61 y = SigmoidFunction(x, bias);
62 plot(x,y)
63 % line for bias
64 %line([-domainOfDefinition domainOfDefinition],[SigmoidFunction(0, bias) SigmoidFunction(0, bias)], 'color', 'y')
65 % line for threshold
66 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
67 axis([x(1) x(end) min(y) max(y)])
68 title('Sigmoidfunktion mit ausgewerteten Merkmalen')
69
70 %% Erste Neuronen Ebene - Iteration ueber alle Merkmale
71 for yi=1:1:(featureCnt)
72     for xi=1:1:(featureCnt)
73         % Bestimmung der Gewichte in Form eines Spaltenvektors
74         weights = GetFeatureOfMatrix(weightMatrix, xi, yi, pixelCnt, featureCnt+1);
75         weights = ConvMatrixToColumn(weights);
76

```

```

77 % Bestimmung der Eingaenge in Form eines Spaltenvektors
78 inputs = GetFeatureOfMatrix(inputMatrix, xi, yi, pixelCnt, featureCnt+1);
79 inputs = ConvMatrixToColumn(inputs);
80
81 % Berechne Neuronenausgang
82 [neuronNetTerms, neuronOutput] = GetNeuronOutput(inputs, weights, bias, domainOfDefinition, 'sigmoid');
83
84 % Werte Teilergebnis aus und trage es in Ausgang-Merkmale-Matrix ein
85 if neuronOutput > threshold
86     outputFeatureMatrix(yi, xi) = 1;
87 else
88     outputFeatureMatrix(yi, xi) = 0;
89 end
90
91 % Zeichne Merkmal in Sigmoidfunktion ein
92 hold on
93 subplot(2,2,3)
94 if (sum(neuronNetTerms) == 0)
95     % Alle Eingangswerte des Neurons sind 0
96     plot(sum(neuronNetTerms), neuronOutput, '*', 'color', 'b')
97 elseif outputFeatureMatrix(yi, xi) == inputFeatureMatrix(yi, xi)
98     % Merkmal richtig detektiert
99     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
100 else
101     % Merkmal falsch detektiert
102     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
103 end
104
105 % zum Debugging
106 outputFeatureMatrixDebug(yi, xi) = sum(neuronNetTerms);
107 outputFeatureMatrixDebug2(yi, xi) = neuronOutput;
108 end
109 end
110
111 %% Erstelle Plot der Ausgangs-Merkmale-Matrix
112 subplot(2,2,4)
113 outputFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, 0, outputFeatureMatrix, '');
114 imshow(outputFeatureMatrix)
115 title('Ausgangs-Merkmale-Matrix')
116
117 %% Debug-Ausgabe
118
119 outputFeatureMatrixDebug(1:end, 1:end)
120 outputFeatureMatrixDebug2(1:end, 1:end)
121
122 bild2 = figure;
123 %% Zweite Neuronen Ebene - Auswertung h-Balken
124 % Parameter
125 domainOfDefinition = 100;
126 bias = -58;
127 threshold = 0.5;
128 row = 3;
129
130 [neuronNetTerms, neuronOutput] = GetNeuronOutput(outputFeatureMatrixDebug(row, 1:end)', ones(1, featureCnt)', bias, domainOfDefinition);
131
132 % Erstelle Plot der Sigmoidfunktion
133 subplot(2,2,1)
134 hold on
135 x = -domainOfDefinition:0.01:domainOfDefinition;
136 y = SigmoidFunction(x, bias);
137 plot(x,y)
138 % line for threshold
139 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
140 axis([x(1) x(end) min(y) max(y)])
141 title('h-Balken Detektion')
142
143 if (neuronOutput > threshold)
144     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
145 else
146     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
147 end
148
149 %% Zweite Neuronen Ebene - Auswertung v-Balken
150 % Parameter
151 domainOfDefinition = 100;
152 bias = -58;
153 threshold = 0.5;
154 column = 3;

```

```

155
156 [neuronNetTerms, neuronOutput] = GetNeuronOutput(outputFeatureMatrixDebug(1:end, column), ones(featureCnt, 1), bias, domain
157
158 % Erstelle Plot der Sigmoidfunktion
159 subplot(2,2,2)
160 hold on
161 x = -domainOfDefinition:0.01:domainOfDefinition;
162 y = SigmoidFunction(x, bias);
163 plot(x,y)
164 % line for threshold
165 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
166 axis([x(1) x(end) min(y) max(y)])
167 title('v-Balken Detektion')
168
169 if (neuronOutput > threshold)
170 plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
171 else
172 plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
173 end
174
175 %% Zweite Neuronen Ebene - Auswertung Error-Field - Nicht generisch
176 % Parameter
177 domainOfDefinition = 100;
178 bias = 22;
179 threshold = 0.5;
180
181 errorField1 = ConvMatrixToColumn(outputFeatureMatrixDebug(1:2, 1:2));
182 errorField2 = ConvMatrixToColumn(outputFeatureMatrixDebug(4:5, 1:2));
183 errorField3 = ConvMatrixToColumn(outputFeatureMatrixDebug(1:2, 4:5));
184 errorField4 = ConvMatrixToColumn(outputFeatureMatrixDebug(4:5, 4:5));
185 sz = size(errorField1);
186
187 [neuronNetTerms1, neuronOutput1] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
188 [neuronNetTerms2, neuronOutput2] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
189 [neuronNetTerms3, neuronOutput3] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
190 [neuronNetTerms4, neuronOutput4] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
191
192 % Erstelle Plot der Sigmoidfunktion
193 subplot(2,2,3)
194 hold on
195 x = -domainOfDefinition:0.01:domainOfDefinition;
196 y = SigmoidFunction(x, bias);
197 plot(x,y)
198 % line for threshold
199 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
200 axis([x(1) x(end) min(y) max(y)])
201 title('Fehler Detektion')
202
203 if (neuronOutput1 < threshold)
204 plot(sum(neuronNetTerms1), neuronOutput1, 'o', 'color', 'r')
205 else
206 plot(sum(neuronNetTerms1), neuronOutput1, 'o', 'color', 'g')
207 end
208 if (neuronOutput2 < threshold)
209 plot(sum(neuronNetTerms2), neuronOutput2, '*', 'color', 'r')
210 else
211 plot(sum(neuronNetTerms2), neuronOutput2, '*', 'color', 'g')
212 end
213 if (neuronOutput3 < threshold)
214 plot(sum(neuronNetTerms3), neuronOutput3, 'x', 'color', 'r')
215 else
216 plot(sum(neuronNetTerms3), neuronOutput3, 'x', 'color', 'g')
217 end
218 if (neuronOutput4 < threshold)
219 plot(sum(neuronNetTerms4), neuronOutput4, '+', 'color', 'r')
220 else
221 plot(sum(neuronNetTerms4), neuronOutput4, '+', 'color', 'g')
222 end

```

## A.6. SettingFileSpecial

```

1 close all, clear all
2
3 %% Parameter
4 % Parameter fuer
5 pixelCnt = 8; % Anzahl der Pixel in x-Richtung pro Merkmal - mindestens 1
6 featureCnt = 5; % Anzahl der Merkmale in x-Richtung - mindestens 1
7 weightType = 'Special'; % Typ der Gewichtsmatrix ('Add', 'AddMul' & 'Mul')
8 inFeatureType = 'Cross'; % Arten der Eingangs-Merkmale-Matrix ((default)'Cross', 'V_Line', 'H_Line' & 'Cal')
9 noise = 40; % Verrauschungsgrad zwischen 0 und 100%
10 slope = 30; % Steigung der Aktivierungs-Funktion (gauss) [50]
11
12 % Parameter fuer Aktivierungsfunktion
13 bias = -13; % Verschiebung in x-Richtung -> Neg (rechts), Pos (links)
14 threshold = 0.85; % Auswertungsschwelle des Ergebnisses
15 domainOfDefinition = 50; % Gueltigkeitsbereich der Neuronenfunktion -> (+/- domainOfDefinition)
16
17 % Parameter fuer Gewichtsmatrix
18 lowerBound = -0.5; % (optional) Untere Grenze der Gewichts-Matrix (default = -1)
19 upperBound = 0.5; % (optional) Obere Grenze der Gewichts-Matrix (default = 1)
20
21 %% Grundeinstellungen
22 % Erstellen der Eingangs-Merkmale-Matrix
23 inputFeatureMatrix = GetInputFeatureMatrix(featureCnt, inFeatureType);
24
25 I1 = [1 1 1 1 1]; % Zeile 1
26 I2 = [1 1 1 1 1]; % Zeile 2
27 I3 = [1 1 1 1 1]; % Zeile 3
28 I4 = [1 1 1 1 1]; % Zeile 4
29 I5 = [1 1 1 1 1]; % Zeile 5
30 % inputFeatureMatrix = uint8([I1; I2; I3; I4; I5]);
31
32 % Erstellen der Ausgangs-Merkmale-Matrix
33 outputFeatureMatrix = zeros(5, 5); % Erstelle Merkmale-Ausgangs-Matrix
34 outputFeatureMatrixDebug = zeros(5,5); % Erstelle Merkmale-Ausgangs-Matrix mit Summe aus Pixeln pro Merkmal
35 outputFeatureMatrixDebug2 = zeros(5,5); % Erstelle Merkmale-Ausgangs-Matrix mit Summe aus Pixeln pro Merkmal
36
37 % Erstellen der Gewichts-Matrix
38 weightMatrix = GetGaussWeights(pixelCnt, featureCnt, slope, weightType, lowerBound, upperBound);
39 % Pixel, Merkmale, Steilheit, Art, (untere, obere Grenze)
40
41
42 %% Erstelle Plot der Gewichts-Matrix
43 bild1 = figure;
44 hold on
45 subplot(2,2,1)
46 mesh(weightMatrix)
47 title('Gewichtsmatrix')
48
49 %% Erstelle Plot der Eingangs-Merkmale-Matrix
50 subplot(2,2,2)
51 inputPixelFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, noise, inputFeatureMatrix, '');
52 imshow(inputPixelFeatureMatrix)
53 title('Eingangs-Merkmale-Matrix')
54
55 inputMatrix = (255 - inputPixelFeatureMatrix)/255;
56
57 %% Erstelle Plot der Sigmoidfunktion
58 subplot(2,2,3)
59 hold on
60 x = -domainOfDefinition:0.01:domainOfDefinition;
61 y = SigmoidFunction(x, bias);
62 plot(x,y)
63 % line for bias
64 %line([-domainOfDefinition domainOfDefinition],[SigmoidFunction(0, bias) SigmoidFunction(0, bias)], 'color', 'y')
65 % line for threshold
66 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
67 axis([x(1) x(end) min(y) max(y)])
68 title('Sigmoidfunktion mit ausgewerteten Merkmalen')
69
70 %% Erste Neuronen Ebene - Iteration ueber alle Merkmale
71 for yi=1:1:(featureCnt)
72     for xi=1:1:(featureCnt)
73         % Bestimmung der Gewichte in Form eines Spaltenvektors
74         weights = GetFeatureOfMatrix(weightMatrix, xi, yi, pixelCnt, featureCnt+1);
75         weights = ConvMatrixToColumn(weights);
76

```

```

77 % Bestimmung der Eingaenge in Form eines Spaltenvektors
78 inputs = GetFeatureOfMatrix(inputMatrix, xi, yi, pixelCnt, featureCnt+1);
79 inputs = ConvMatrixToColumn(inputs);
80
81 % Berechne Neuronenausgang
82 [neuronNetTerms, neuronOutput] = GetNeuronOutput(inputs, weights, bias, domainOfDefinition, 'sigmoid');
83
84 % Werte Teilergebnis aus und trage es in Ausgang-Merkmale-Matrix ein
85 if neuronOutput > threshold
86     outputFeatureMatrix(yi, xi) = 1;
87 else
88     outputFeatureMatrix(yi, xi) = 0;
89 end
90
91 % Zeichne Merkmal in Sigmoidfunktion ein
92 hold on
93 subplot(2,2,3)
94 if (sum(neuronNetTerms) == 0)
95     % Alle Eingangswerte des Neurons sind 0
96     plot(sum(neuronNetTerms), neuronOutput, '*', 'color', 'b')
97 elseif outputFeatureMatrix(yi, xi) == inputFeatureMatrix(yi, xi)
98     % Merkmal richtig detektiert
99     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
100 else
101     % Merkmal falsch detektiert
102     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
103 end
104
105 % zum Debugging
106 outputFeatureMatrixDebug(yi, xi) = sum(neuronNetTerms);
107 outputFeatureMatrixDebug2(yi, xi) = neuronOutput;
108 end
109 end
110
111 %% Erstelle Plot der Ausgangs-Merkmale-Matrix
112 subplot(2,2,4)
113 outputFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, 0, outputFeatureMatrix, '');
114 imshow(outputFeatureMatrix)
115 title('Ausgangs-Merkmale-Matrix')
116
117 %% Debug-Ausgabe
118
119 outputFeatureMatrixDebug(1:end, 1:end)
120 outputFeatureMatrixDebug2(1:end, 1:end)
121
122 bild2 = figure;
123 %% Zweite Neuronen Ebene - Auswertung h-Balken
124 % Parameter
125 domainOfDefinition = 150;
126 bias = -80;
127 threshold = 0.5;
128 row = 3;
129
130 [neuronNetTerms, neuronOutput] = GetNeuronOutput(outputFeatureMatrixDebug(row, 1:end)', ones(1, featureCnt)', bias, domainOfDefinition);
131
132 % Erstelle Plot der Sigmoidfunktion
133 subplot(2,2,1)
134 hold on
135 x = -domainOfDefinition:0.01:domainOfDefinition;
136 y = SigmoidFunction(x, bias);
137 plot(x,y)
138 % line for threshold
139 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
140 axis([x(1) x(end) min(y) max(y)])
141 title('h-Balken Detektion')
142
143 if (neuronOutput > threshold)
144     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
145 else
146     plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
147 end
148
149 %% Zweite Neuronen Ebene - Auswertung v-Balken
150 % Parameter
151 domainOfDefinition = 150;
152 bias = -80;
153 threshold = 0.5;
154 column = 3;

```

```

155
156 [neuronNetTerms, neuronOutput] = GetNeuronOutput(outputFeatureMatrixDebug(1:end, column), ones(featureCnt, 1), bias, domain
157
158 % Erstelle Plot der Sigmoidfunktion
159 subplot(2,2,2)
160 hold on
161 x = -domainOfDefinition:0.01:domainOfDefinition;
162 y = SigmoidFunction(x, bias);
163 plot(x,y)
164 % line for threshold
165 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
166 axis([x(1) x(end) min(y) max(y)])
167 title('v-Balken Detektion')
168
169 if (neuronOutput > threshold)
170 plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'g')
171 else
172 plot(sum(neuronNetTerms), neuronOutput, 'o', 'color', 'r')
173 end
174
175 %% Zweite Neuronen Ebene - Auswertung Error-Field - Nicht generisch
176 % Parameter
177 domainOfDefinition = 150;
178 bias = 11;
179 threshold = 0.5;
180
181 errorField1 = ConvMatrixToColumn(outputFeatureMatrixDebug(1:2, 1:2));
182 errorField2 = ConvMatrixToColumn(outputFeatureMatrixDebug(4:5, 1:2));
183 errorField3 = ConvMatrixToColumn(outputFeatureMatrixDebug(1:2, 4:5));
184 errorField4 = ConvMatrixToColumn(outputFeatureMatrixDebug(4:5, 4:5));
185 sz = size(errorField1);
186
187 [neuronNetTerms1, neuronOutput1] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
188 [neuronNetTerms2, neuronOutput2] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
189 [neuronNetTerms3, neuronOutput3] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
190 [neuronNetTerms4, neuronOutput4] = GetNeuronOutput(errorField1, ones(sz(1, 1), 1), bias, domainOfDefinition, 'sigmoid');
191
192 % Erstelle Plot der Sigmoidfunktion
193 subplot(2,2,3)
194 hold on
195 x = -domainOfDefinition:0.01:domainOfDefinition;
196 y = SigmoidFunction(x, bias);
197 plot(x,y)
198 % line for threshold
199 line([-domainOfDefinition domainOfDefinition],[threshold threshold], 'color', 'y')
200 axis([x(1) x(end) min(y) max(y)])
201 title('Fehler Detektion')
202
203 if (neuronOutput1 < threshold)
204 plot(sum(neuronNetTerms1), neuronOutput1, 'o', 'color', 'r')
205 else
206 plot(sum(neuronNetTerms1), neuronOutput1, 'o', 'color', 'g')
207 end
208 if (neuronOutput2 < threshold)
209 plot(sum(neuronNetTerms2), neuronOutput2, '*', 'color', 'r')
210 else
211 plot(sum(neuronNetTerms2), neuronOutput2, '*', 'color', 'g')
212 end
213 if (neuronOutput3 < threshold)
214 plot(sum(neuronNetTerms3), neuronOutput3, 'x', 'color', 'r')
215 else
216 plot(sum(neuronNetTerms3), neuronOutput3, 'x', 'color', 'g')
217 end
218 if (neuronOutput4 < threshold)
219 plot(sum(neuronNetTerms4), neuronOutput4, '+', 'color', 'r')
220 else
221 plot(sum(neuronNetTerms4), neuronOutput4, '+', 'color', 'g')
222 end

```

## A.7. GetNeuronOutput

```
1 function [netTerms, output] = GetNeuronOutput(inputs, weights, bias, domainOfDefinition, type)
2 % Berechnet ein Neuron und gibt das Zwischen- und Endergebnis zurueck.
3 %
4 % returns:
5 % netTerms - Zwischenergebnisse als Spaltenvektor -> inputs * weights
6 % output - Ausgang des Neurons
7 %
8 % inputs - Zustende der Eingaenge als Spaltenvektor
9 % weights - Gewichte der Eingaenge als Spaltenvektor
10 % bias - Verschiebung in x-Richtung -> Neg (rechts), Pos (links)
11 % domainOfDefinition - Ergebnis wird auf Gueltigkeitsbereich begrenzt
12 % -> (+/- domainOfDefinition)
13 % type - Typ der Aktivierungsfunktion - akutell nur Sigmoidfunktion
14
15 % Ueberpruefe, ob Eingaenge als Spaltenvektor vorliegen
16 sizeInputs = size(inputs);
17 if sizeInputs(2) > 1
18     error('Eingaenge und Gewichte muessen Spaltenvektoren sein')
19 end
20 clear sizeInputs
21
22 % Ueberpruefen, ob die Anzahl der Eingaenge und Gewichte zusammen passen
23 if (length(inputs) ~= length(weights))
24     error('Anzahl der angegebenen Eingaenge und der Eingaenge muessen gleich sein.')
25 end
26
27 % Eingaenge und Gewichten multiplizieren
28 netTerms = weights .* inputs;
29
30 % Summe ueber alle Zwischenergebnisse
31 netOutput = sum(netTerms);
32
33 % Gueltigkeitsbereich der Zwischenergebnisse einschraenken
34 while (netOutput > domainOfDefinition) || (-domainOfDefinition > netOutput)
35     netOutput = netOutput ./ 2;
36     netTerms = netTerms ./ 2;
37     error('Zahlenbereich ueberschritten')
38 end
39
40 % Funktion zuordnen
41 if strcmp(type, 'sigmoid')
42     output = SigmoidFunction(netOutput, bias);
43 else
44     error('type unknown, use sigmoid')
45 end
46
47 end
```

## A.8. GetFeatureOfMatrix

```
1 function feature = GetFeatureOfMatrix(matrix, featureX, featureY, pixelCnt, featureCnt)
2 % Funktion teilt die uebergebene Matrix in Merkmale ein und gibt nur
3 % ein Merkmal zurueck
4 %
5 % matrix - Matrix mit allen Merkmalen
6 % featureX - Position des Merkmals in X-Richtung - startet mit 1
7 % featureY - Position des Merkmals in Y-Richtung - startet mit 1
8 % pixelCnt - Anzahl der Pixel in x-Richtung pro Merkmal - mindestens 1
9 % featureCnt - Anzahl der Merkmale in x-Richtung - mindestens 1
10
11 % Ueberpruefen der Uebergabeparamter
12 % featureX und featureY muessen mit 1 beginnen
13 if (featureX <= 0)
14     error('featureX muss mit 1 beginnen');
15 end
16 if (featureY <= 0)
17     error('featureY muss mit 1 beginnen');
18 end
19 % pixelCnt und featureCnt muessen mindestens 1 sein
20 if (pixelCnt <= 0)
21     error('pixelCnt muss mindestens 1 sein');
22 end
23 if (featureCnt <= 0)
24     error('featureCnt muss mindestens 1 sein');
25 end
26 % Merkmalverfuegbarkeit testen
27 if (featureX > featureCnt) || (featureY > featureCnt)
28     error('Merkmal ist nicht verfuegbar');
29 end
30
31 featureX = featureX - 1;
32 featureY = featureY - 1;
33
34 rangeMinY = 1+pixelCnt*featureY;
35 rangeMaxY = pixelCnt+pixelCnt*featureY;
36 rangeMinX = 1+pixelCnt*featureX;
37 rangeMaxX = pixelCnt+pixelCnt*featureX;
38
39 featureMatrix = matrix(rangeMinY:rangeMaxY, rangeMinX:rangeMaxX);
40 feature = featureMatrix;
41 end
```

## A.9. ConvMatrixToColumn

```
1 function res = ConvMatrixToColumn(matrix)
2 % Funktion liest Matrix zeilenweise ein und gibt matrix als
3 % Spaltenvektor aus
4 % res - gibt einen Spaltenvektor zurueck
5 % matrix - zu konvertierende Matrix
6
7 sizeMatrix = size(matrix);
8 res = zeros(sizeMatrix(1)*sizeMatrix(2), 1);
9
10 i=1;
11 for y=1:sizeMatrix(1)
12     for x=1:sizeMatrix(2)
13         res(i) = matrix(y,x);
14         i = i+1;
15     end
16 end
17 end
```

## A.10. GaussNormFunction

```

1 function [gauss, y] = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma, mue)
2 %% Funktionsdefinition
3 %% y = (1/(sqrt(2*pi).*sigma))*exp(-((x-mue).^2)/(2.*sigma.^2))
4 % y = (1/(sqrt(2*pi.*(sigma.^2))))*exp(-((x-mue).^2)/(2.*sigma.^2))
5
6 % y - (optional) gibt einzelnen Wert im Punkt x zurueck
7 % gauss - Vektor der Dimension (nice) mit den abgetasteten Funktionswerte
8
9 % nice - Feinheit der Abtastung der Funktion, bestimmt Laenge von (gauss) (Empfehlung >> 3)
10 % x1 - untere Grenze der Definitionsbreite (x-Werte) fuer Funktionswerte von (gauss) (default: -6)
11 % x2 - obere Grenze der Definitionsbreite (x-Werte) fuer Funktionswerte von (gauss) (default: 6)
12 % y1 - untere Grenze der Funktionswerte, gilt fuer y und/oder gauss
13 % y2 - obere Grenze der Funktionswerte, gilt fuer y und/oder gauss
14 % wenn y1 && y2 == 0 wird keine Skalierung vorgenommen
15 % x - (optional) Variable fuer einzelnen Rueckgabewert der Funktion
16 % sigma - Standardabweichung, beeinflusst den Grad der Zentrierung bzw. Steilheit von gauss (default: 1)
17 % mue - (optional) Verschiebung in x-Richtung, gilt fuer y und/oder gauss (default: 0) -> Neg (rechts), Pos (links)
18
19 %% Wichtigste vollstaendige Funktionen
20 % gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma, mue) % vollparametrisierter gauss
21 % y = GaussNormFunction(y1, y2, x, sigma, mue)
22 % gauss OHNE Skalierung mittels "y1 && y2 = 0" in 1:7
23 % gauss = GaussNormFunction(nice, x1, x2, 0, 0, sigma, mue)
24 % y OHNE Skalierung
25 % y = GaussNormFunction(x, sigma, mue)
26
27 % Moegliche Nutzungen von GaussNormFunction: (Aeusserere Aufrufe)
28
29 % 1:1 % y = GaussNormFunction(x)                                     == (1:2 default: sigma)
30 % 1:2 % y = GaussNormFunction(x, sigma)                                == (1:3 default: mue)
31 % 2:2 % [gauss, y] = GaussNormFunction(nice, x)                      == (1:4 default: x1, x2, sigma)
32 % 1:3 % y = GaussNormFunction(x, sigma, mue)
33 % 2:3 % [gauss, y] = GaussNormFunction(nice, x, sigma)                == 1:2 && (1:4 default: x1, x2)
34 % 1:4 % gauss = GaussNormFunction(nice, x1, x2, sigma)               == (1:6 mit: y1 && y2 == 0)
35 % 2:4 % [gauss, y] = GaussNormFunction(nice, x1, x2, x)              == 1:1 && (1:4 default: sigma)
36 % 1:5 % y = GaussNormFunction(y1, y2, x, sigma, mue)
37 % 2:5 % [gauss, y] = GaussNormFunction(nice, x1, x2, x, sigma)        == 1:4 && 1:2
38 % 1:6 % gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma)       == (1:7 default: mue)
39 % 2:6 % [gauss, y] = GaussNormFunction(nice, x1, x2, x, sigma, mue)   == 1:5 && (1:7 mit: y1 && y2 == 0)
40 % 1:7 % gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma, mue)
41 % 2:7 % [gauss, y] = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma) == 1:6 && (1:5 default: mue)
42 % 1:8 % y = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma, mue)    == 1:5
43 % 2:8 % [gauss, y] = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma, mue) == 1:5 && 1:7
44
45 % default-Werte
46 x1Default = -6;
47 x2Default = 6;
48 mueDefault = 0;
49 sigmaDefault = 1;
50
51 %% Ab hier Unterscheidung je nach Anzahl der gegebenen Input- und Output-Werte
52
53 if nargin == 1
54     %% 1 Input-Wert gegeben
55     %% 1 Output
56     if nargout == 1
57         % 1:1 % y = GaussNormFunction(x)
58         x = nice;
59         y = (1/(sqrt(2*pi).*sigmaDefault))*exp(-((x-mueDefault).^2)/(2.*sigmaDefault.^2));
60         gauss = y;
61     else
62         error('GaussNormFunction kann mit einem Input "x" nur einen Output "y" erzeugen!');
63     end
64
65 elseif nargin == 2
66     %% 2 Input-Wert gegeben
67     %% 1 Output
68     if nargout == 1
69         % 1:2 % y = GaussNormFunction(x, sigma)
70         x = nice;
71         sigma = x1;
72         y = (1/(sqrt(2*pi).*sigma))*exp(-((x-mueDefault).^2)/(2.*sigma.^2));
73         gauss = y;
74     %% 2 Outputs
75     elseif nargout == 2
76         % 2:2 % [gauss, y] = GaussNormFunction(nice, x)

```

```

77         x = x1;
78         y = GaussNormFunction(x);
79         gauss = zeros(nice,1);
80         j=1;
81         for i=linspace(x1Default, x2Default, nice)
82             gauss(j) = (1/(sqrt(2*pi).*sigmaDefault))*exp(-((i-mueDefault).^2)/(2.*sigmaDefault.^2));
83             j = j + 1;
84         end
85     end
86
87 elseif nargin == 3
88 %% 3 Input-Wert gegeben
89 % 1 Output
90 if nargout == 1
91 % 1:3 % y = GaussNormFunction(x, sigma, mue)
92     x = nice;
93     sigma = x1;
94     mue = x2;
95     y = (1/(sqrt(2*pi).*sigma))*exp(-((x-mue).^2)/(2.*sigma.^2));
96     gauss = y;
97 %% 2 Outputs
98 elseif nargout == 2
99 % 2:3 % [gauss, y] = GaussNormFunction(nice, x, sigma)
100    x = x1;
101    sigma = x2;
102    y = (1/(sqrt(2*pi).*sigma))*exp(-((x-mueDefault).^2)/(2.*sigma.^2));
103    gauss = zeros(nice,1);
104    j=1;
105    for i=linspace(x1Default, x2Default, nice)
106        gauss(j) = (1/(sqrt(2*pi).*sigma))*exp(-((i-mueDefault).^2)/(2.*sigma.^2));
107        j = j + 1;
108    end
109 end
110
111 elseif nargin == 4
112 %% 4 Input-Wert gegeben
113 % 1 Output
114 if nargout == 1
115 % 1:4 % gauss = GaussNormFunction(nice, x1, x2, sigma)
116     sigma = y1;
117     gauss = GaussNormFunction(nice, x1, x2, 0, 0, sigma);
118 %% 2 Outputs
119 elseif nargout == 2
120 % 2:4 % [gauss, y] = GaussNormFunction(nice, x1, x2, x)
121     x = y1;
122     y = GaussNormFunction(x);
123     gauss = GaussNormFunction(nice, x1, x2, 0, 0, sigmaDefault);
124 end
125
126 elseif nargin == 5
127 %% 5 Input-Wert gegeben
128 % 1 Output
129 if nargout == 1
130 % 1:5 % y = GaussNormFunction(y1, y2, x, sigma, mue)
131     sigma = y1;
132     mue = y2;
133     x = x2;
134     y2 = x1;
135     y1 = nice;
136
137     y = GaussNormFunction(x, sigma, mue);
138     if y1 == 0 && y2 == 0
139         gauss = y;
140     else
141         if y2 < y1
142             error('Grenzen von GaussNormFunction vertauscht da obere Grenze "y2" kleiner als untere Grenze "y1" gewaehlt');
143         elseif y2 == y1
144             error('Grenzen von GaussNormFunction falsch: da obere Grenze "y2" gleich unterer Grenze "y1" gewaehlt');
145         end
146         maxGauss = (1/(sqrt(2*pi).*sigma))*exp(-(0)/(2.*sigma.^2));
147         y = y./maxGauss; % auf 0..1 normieren
148         if y1 == 0
149             y = y.*y2; % auf 0..y2 normieren
150         elseif -y1 == y2 % symetrisch definiert
151             y = y.*2 - 1; % auf -1..1 normieren
152             y = y.*y2; % auf y1..y2 bzw. -y2..y2 normieren
153         else
154             range = y2 - y1;

```

```

155         y = y.*range;
156         y = y + y1;
157     end
158     gauss = y;
159 end
160
161 % 2 Outputs
162 elseif nargin == 2
163 % 2:5 % [gauss, y] = GaussNormFunction(nice, x1, x2, x, sigma)
164     x = y1;
165     sigma = y2;
166     gauss = GaussNormFunction(nice, x1, x2, sigma);
167     y = GaussNormFunction(x, sigma);
168 end
169
170 elseif nargin == 6
171 %% 6 Input-Wert gegeben
172 % 1 Output
173 if nargout == 1
174 % 1:6 % gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma)
175     sigma = x;
176     gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma, mueDefault);
177 % 2 Outputs
178 elseif nargin == 2
179 % 2:6 % [gauss, y] = GaussNormFunction(nice, x1, x2, x, sigma, mue)
180     % mue = x;
181     % sigma = y2;
182     % x = y1;
183     % y = GaussNormFunction(y1, y2, x, sigma, mue);
184     % y = GaussNormFunction(nice, x1, x2, x, sigma);
185     mue = x;
186     sigma = y2;
187     x = y1;
188     y = GaussNormFunction(0, 0, x, sigma, mue);
189     gauss = GaussNormFunction(nice, x1, x2, 0, 0, sigma, mue);
190 end
191
192 elseif nargin == 7
193 %% 7 Input-Wert gegeben
194 % 1 Output
195 if nargout == 1
196 % 1:7 % gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma, mue)
197     mue = sigma;
198     sigma = x;
199     if mod(nice, 1) ~= 0
200         error('Wert fuer "nice" muss ganzzahlig sein!');
201     elseif nice <= 0
202         error('Wert fuer "nice" darf nicht kleiner oder gleich null sein!');
203     elseif nice <= 3
204         warning('Wert fuer "nice" wurde kleiner gleich 3 gewaehlt -> sehr geringe Aufloesung! ');
205     end
206     if x1 > x2
207         error('SigmoidFunction untere Grenze "x1" darf nicht groesser als obere Grenze "x2" sein!');
208     elseif x1 == x2
209         error('SigmoidFunction untere Grenze "x1" darf nicht gleich oberer Grenze "x2" sein!');
210     end
211     if y1 == 0 && y2 == 0
212         gauss = zeros(nice,1);
213         j=1;
214         for i=linspace(x1, x2, nice)
215             gauss(j) = (1/(sqrt(2*pi).*sigma))*exp(-((i-mue).^2)/(2.*sigma.^2));
216             j = j + 1;
217         end
218     else
219         if y2 < y1
220             error('Grenzen von GaussNormFunction vertauscht da obere Grenze "y2" kleiner als untere Grenze "y1" gewaehlt');
221         elseif y2 == y1
222             error('Grenzen von GaussNormFunction falsch: da obere Grenze "y2" gleich unterer Grenze "y1" gewaehlt wurden');
223         end
224         gauss = zeros(nice,1);
225         j=1;
226         for i=linspace(x1, x2, nice)
227             gauss(j) = (1/(sqrt(2*pi).*sigma))*exp(-((i-mue).^2)/(2.*sigma.^2));
228             j = j + 1;
229         end
230         maxGauss = max(gauss);
231         gauss = gauss./maxGauss; % auf 0..1 normieren
232         if y1 == 0

```

```

233         gauss = gauss.*y2; % auf 0..y2 normieren
234     elseif -y1 == y2 % symetrisch definiert
235         gauss = gauss.*2 - 1; % auf -1..1 normieren
236         gauss = gauss.*y2; % auf y1..y2 bzw. -y2..y2 normieren
237     else
238         range = y2 - y1;
239         gauss = gauss.*range;
240         gauss = gauss + y1;
241     end
242 end
243 % 2 Outputs
244 elseif nargin == 2
245     % 2:7 % [gauss, y] = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma)
246     y = GaussNormFunction(y1, y2, x, sigma, mueDefault);
247     gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma);
248 end
249
250 elseif nargin == 8
251     %% 8 Input-Wert gegeben
252     % 1 Output
253     if nargin == 1
254         % 1:8 % y = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma, mue)
255         y = GaussNormFunction(y1, y2, x, sigma, mue);
256         gauss = y;
257     % 2 Outputs
258     elseif nargin == 2
259         % 2:8 % [gauss, y] = GaussNormFunction(nice, x1, x2, y1, y2, x, sigma, mue)
260         y = GaussNormFunction(y1, y2, x, sigma, mue);
261         gauss = GaussNormFunction(nice, x1, x2, y1, y2, sigma, mue);
262     end
263
264 else
265     error('Ungueltige Anzahl Inputs fuer GaussNormFunction!');
266 end
267
268 end

```

## A.11. SigmoidFunction

```
1 function [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2, x, bias)
2 % y - (optional) gibt einzelnen Wert im Punkt x zurueck
3 % sigmoid - Vektor der Dimension (nice) mit den abgetasteten Funktionswerte
4
5 % nice - Feinheit der Abtastung der Funktion, bestimmt Laenge von (sigmoid) (Empfehlung >> 3)
6 % x1 - untere Grenze der Definitionsbreite (x-Werte) fuer Funktionswerte von (sigmoid) (default: -12)
7 % x2 - obere Grenze der Definitionsbreite (x-Werte) fuer Funktionswerte von (sigmoid) (default: 12)
8 % y1 - untere Grenze der Funktionswerte, gilt fuer y und/oder sigmoid (y-Werte default: -1)
9 % y2 - obere Grenze der Funktionswerte, gilt fuer y und/oder sigmoid (y-Werte default: 1)
10 % x - (optional) Variable fuer einzelnen Rueckgabewert der Funktion
11 % bias - (optional) Verschiebung in x-Richtung, gilt fuer y und/oder sigmoid (default: 0) -> Neg (rechts), Pos (links)
12
13 %% Wichtigste vollstaendige Funktionen
14 % sigmoid = SigmoidFunction(nice, x1, x2, y1, y2, bias)
15 % y = SigmoidFunction(y1, y2, x, bias)
16
17 % Moegliche Nutzungen von SigmoidFunction: (Aeusserere Aufrufe)
18 % y = SigmoidFunction(x)
19 % y = SigmoidFunction(x, bias)
20 % [sigmoid, y] = SigmoidFunction(nice, x)
21 % sigmoid = SigmoidFunction(nice, x1, x2)
22 % [sigmoid, y] = SigmoidFunction(nice, x, bias)
23 % y = SigmoidFunction(y1, y2, x, bias)
24 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, bias)
25 % y = SigmoidFunction(x1, x2, y1, y2, x)
26 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, x, bias)
27 % sigmoid = SigmoidFunction(nice, x1, x2, y1, y2, bias)
28 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2, x)
29 % y = SigmoidFunction(nice, x1, x2, y1, y2, x, bias)
30 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2, x, bias)
31
32 % default-Werte
33 biasDefault = 0;
34 x1Default = -12;
35 x2Default = 12;
36
37 %% Ab hier Unterscheidung je nach Anzahl der gegebenen Input- und Output-Werte
38
39 if nargin == 1
40 % 1 Input-Wert gegeben
41 % Funktion wird lediglich zur Ermittlung eines Wertes (y) auf Grundlage von (x) genutzt
42 % Bei einem Input und Output-Wert wird keine Meldung ausgegeben
43 % es werden fuer die nicht angegeben Argumente werden default-Werte gesetzt
44 % y = SigmoidFunction(x) -> Aeusserer Aufruf
45 % sigmoid = SigmoidFunction(nice) -> Innerer Aufruf
46 if nargout == 1
47 x = nice;
48 bias = biasDefault;
49 sigmoid = 1./(1+exp(-(x+bias)));
50
51 % Mit einem Input-Wert koennen nicht beide Output-Werte sinnvoll bedient werden
52 % [sigmoid, y] = SigmoidFunction(nice, x1)
53 elseif nargout == 2
54 error('SigmoidFunction benoetigt mehr als einen Input-Wert fuer zwei Rueckgabewerte!');
55
56 % Nicht definierte Anzahl an Rueckgabewerten
57 % [sigmoid, y, ...] = SigmoidFunction(nice)
58 else
59 error('SigmoidFunction kann bei einem Input-Wert nicht mehr als einen Output-Wert erzeugen!');
60 end
61
62 elseif nargin == 2
63 % 2 Input-Werte gegeben
64 % 2 Input-Werte -> Unterscheidung nach Output-Anzahl notwendig
65
66 % 1 Output-Wert -> Ermittlung eines Wertes (y)
67 % auf Grundlage von (x) und (bias) mit default-Werten
68 % y = SigmoidFunction(x, bias) -> Aeusserer Aufruf
69 % sigmoid = SigmoidFunction(nice, x1) -> Innerer Aufruf
70 if nargout == 1
71 x = nice;
72 bias = x1;
73 sigmoid = 1./(1+exp(-(x+bias)));
74
75 % 2 Output-Werte -> Rueckgabe eines Wertes (y) und des Vektors (sigmoid)
76 % auf Grundlage von (x) und (nice) mit default-Werten
```

```

77 % [sigmoid, y] = SigmoidFunction(nice, x) -> Auesserer Aufruf
78 % [sigmoid, y] = SigmoidFunction(nice, x1) -> Innerer Aufruf
79 elseif nargout == 2
80
81     % Test ob nice-Wert eine ganze Zahl und groesser null ist
82     if mod(nice, 1) ~= 0
83         error('Wert fuer "nice" muss ganzzahlig sein! ');
84     elseif nice <= 0
85         error('Wert fuer "nice" darf nicht kleiner oder gleich null sein! ');
86     elseif nice <= 3
87         warning('Wert fuer "nice" wurde kleiner gleich 3 gewaehlt -> sehr geringe Aufloesung! ');
88     end
89
90     bias = biasDefault;
91     x = x1;
92     x1 = x1Default;
93     x2 = x2Default;
94     y = 1./(1+exp(-(x+bias)));
95     sigmoid = zeros(nice,1);
96     j=1;
97     for i=linspace(x1, x2, nice)
98         sigmoid(j) = 1./(1+exp(-(i+bias)));
99         j = j + 1;
100    end
101
102    else % obsolet da Matlab schon vorher eine Fehlermeldung erzeugt -> wird ab hier weggelassen
103        error('SigmoidFunction kann bei zwei Input-Werten nicht mehr als zwei Output-Werte erzeugen! ');
104    end
105
106 elseif nargin == 3
107     %% 3 Input-Werte gegeben
108     %% 3 Input-Werte -> Unterscheidung nach Output-Anzahl notwendig
109     %% Fehlermeldung fuer mehr als 2 Outputs wird vernachlaessigt
110
111     % Test ob nice-Wert eine ganze Zahl und groesser null ist
112     if mod(nice, 1) ~= 0
113         error('Wert fuer "nice" muss ganzzahlig sein! ');
114     elseif nice <= 0
115         error('Wert fuer "nice" darf nicht kleiner oder gleich null sein! ');
116     elseif nice <= 3
117         warning('Wert fuer "nice" wurde kleiner gleich 3 gewaehlt -> sehr geringe Aufloesung! ');
118     end
119
120     % 1 Output-Wert -> Ermittlung des sigmoid-Vektors
121     % auf Grundlage der Grenzen (x1) & (x2) mit Abtastung (nice)
122     % sigmoid = SigmoidFunction(nice, x1, x2) -> Auesserer & Innerer Aufruf
123     if nargout == 1
124
125         % Gueltigkeit von x1 und x2 testen
126         if x1 > x2
127             error('SigmoidFunction untere Grenze "x1" darf nicht groesser als obere Grenze "x2" sein! ');
128         elseif x1 == x2
129             error('SigmoidFunction untere Grenze "x1" darf nicht gleich oberer Grenze "x2" sein! ');
130         end
131
132         bias = biasDefault;
133         sigmoid = zeros(nice,1);
134         j=1;
135         for i=linspace(x1, x2, nice)
136             sigmoid(j) = 1./(1+exp(-(i+bias)));
137             j = j + 1;
138         end
139
140     % 2 Output-Werte -> Ermittlung des sigmoid-Vektors und eines Funktionswertes y
141     % auf Grundlage der Feinheit (nice) und eines (bias) mit default-Werten und
142     % eines Eingangswertes (x) und eines (bias) mit default-Werten
143     % [sigmoid, y] = SigmoidFunction(nice, x, bias) -> Auesserer Aufruf
144     % [sigmoid, y] = SigmoidFunction(nice, x1, x2) -> Innerer Aufruf
145     elseif nargout == 2
146         bias = x2;
147         x = x1;
148         x1 = x1Default;
149         x2 = x2Default;
150         sigmoid = zeros(nice,1);
151         j=1;
152         for i=linspace(x1, x2, nice)
153             sigmoid(j) = 1./(1+exp(-(i+bias)));
154             j = j + 1;

```

```

155         end
156         y = 1./(1+exp(-(x+bias)));
157     end
158
159 elseif nargin == 4
160 % 4 Input-Werte gegeben
161
162 % erste Zuweisung der uebergebenen Argumente zur Ueberpruefung bei
163 % einem Output notwendig
164 bias = y1;
165 x = x2;
166 y2 = x1;
167 y1 = nice;
168
169 % 1 Output-Wert -> Ermittlung des Wertes y
170 % im Punkt von (x) skaliert auf die Grenzen (y1) & (y2) mit (bias)
171 % y = SigmoidFunction(y1, y2, x, bias) -> Aeußerer Aufruf
172 % sigmoid = SigmoidFunction(nice, x1, x2, y1) -> Innerer Aufruf
173 if nargout == 1
174     if y2 < y1
175         error('Grenzen von SigmoidFunction vertauscht da obere Grenze "y2" kleiner als untere Grenze "y1" gewählt wurde!');
176     elseif y2 == y1
177         error('Grenzen von SigmoidFunction falsch: da obere Grenze "y2" gleich unterer Grenze "y1" gewählt wurde!');
178     end
179
180     y = 1/(1+exp(-(x+bias))); % y-Wert fuer nachfolgende Skalierung erzeugen
181
182 if -y1==y2 % Test ob symetrisch definiert -> einfache Verschiebung in y notwendig & skalieren
183     if y == 0.5 % y liegt im Koordinatenursprung nach Verschiebung
184         y = 0;
185     else
186         y = y - 0.5; % Verschiebung auf -0.5..0.5
187         y = y * 2; % Skalierung auf -1..1
188         y = y * y2; % gewünschte Skalierung
189     end
190 else
191     range = y2 - y1;
192     y = y * range; % Skalierung auf 0..range bzw. 0..(y2-y1)
193     y = y + y1; % Verschiebung auf y1..y2 bzw. (0+y1)..((y2-y1)+y1)
194 end
195 sigmoid = y;
196
197 % 2 Output-Werte -> Ermittlung des sigmoid-Vektors auf Grundlage der Feinheit (nice) und den Grenzen (x1) und (x2)
198 % Und Ermittlung eines Funktionswertes (y) auf Grundlage von (x) mit default-Werten
199 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, x) -> Aeußerer Aufruf
200 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1) -> Innerer Aufruf
201 elseif nargout == 2
202
203     y1 = bias; % Erste Zuweisung vor 'nargout == 1' rückgängig machen
204     x = y1;
205     bias = biasDefault; % default-Wert einstellen
206
207     % Test ob nice-Wert eine ganze Zahl und grösser null ist
208     if mod(nice, 1) ~= 0
209         error('Wert fuer "nice" muss ganzzahlig sein!');
210     elseif nice <= 0
211         error('Wert fuer "nice" darf nicht kleiner oder gleich null sein!');
212     elseif nice <= 3
213         warning('Wert fuer "nice" wurde kleiner gleich 3 gewählt -> sehr geringe Auflösung!');
214     end
215
216     % Gültigkeit von x1 und x2 testen
217     if x1 > x2
218         error('SigmoidFunction untere Grenze "x1" darf nicht grösser als obere Grenze "x2" sein!');
219     elseif x1 == x2
220         error('SigmoidFunction untere Grenze "x1" darf nicht gleich oberer Grenze "x2" sein!');
221     end
222
223     y = 1./(1+exp(-(x+bias)));
224     sigmoid = zeros(nice,1);
225     j=1;
226     for i=linspace(x1, x2, nice)
227         sigmoid(j) = 1./(1+exp(-(i+bias)));
228         j = j + 1;
229     end
230 end
231
232 elseif nargin == 5 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2, x, bias)

```

```

233 %% 5 Input-Werte gegeben
234
235 % Ueberpruefung von nice, x1 & x2 fuer beide Output-Varianten notwendig
236 % Test ob nice-Wert eine ganze Zahl und groesser null ist
237 if mod(nice, 1) ~= 0
238     error('Wert fuer "nice" muss ganzzahlig sein!');
239 elseif nice <= 0
240     error('Wert fuer "nice" darf nicht kleiner oder gleich null sein!');
241 elseif nice <= 3
242     warning('Wert fuer "nice" wurde kleiner gleich 3 gewaehlt -> sehr geringe Aufloesung!');
243 end
244
245 % Gueltigkeit von x1 und x2 testen
246 if x1 > x2
247     error('SigmoidFunction untere Grenze "x1" darf nicht groesser als obere Grenze "x2" sein!');
248 elseif x1 == x2
249     error('SigmoidFunction untere Grenze "x1" darf nicht gleich oberer Grenze "x2" sein!');
250 end
251
252 % 1 Output-Wert -> Ermittlung des sigmoid-Vektors
253 % auf Grundlage der Grenzen (x1), (x2), (y1) & (y2) mit Abtastung (nice)
254 % sigmoid = SigmoidFunction(nice, x1, x2, y1, y2) -> Aeusserer & Innerer Aufruf
255 if nargin == 1
256
257     if y2 < y1
258         error('Grenzen von SigmoidFunction vertauscht da obere Grenze "y2" kleiner als untere Grenze "y1" gewaehlt wurde!');
259     elseif y2 == y1
260         error('Grenzen von SigmoidFunction falsch: da obere Grenze "y2" gleich unterer Grenze "y1" gewaehlt wurde!');
261     end
262
263     bias = biasDefault;
264     sigmoid = zeros(nice,1);
265     j=1;
266     for i=linspace(x1, x2, nice)
267         sigmoid(j) = 1./(1+exp(-(i+bias)));
268         j = j + 1;
269     end
270
271     if -y1==y2 % Test ob symetrisch definiert -> einfache Verschiebung in y notwendig & skalieren
272         sigmoid = sigmoid - 0.5; % Verschiebung auf -0.5..0.5
273         sigmoid = sigmoid * 2; % Skalierung auf -1..1
274         sigmoid = sigmoid * y2; % gewuenschte Skalierung
275     else
276         range = y2 - y1;
277         sigmoid = sigmoid * range; % Skalierung auf 0..range bzw. 0..(y2-y1)
278         sigmoid = sigmoid + y1; % Verschiebung auf y1..y2 bzw. (0+y1)..((y2-y1)+y1)
279     end
280
281 % 2 Output-Werte -> Ermittlung des sigmoid-Vektors und eines Funktionswertes y
282 % auf Grundlage der Feinheit (nice), eines Eingangswertes (x) und eines (bias)
283 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, x, bias) -> Aeusserer Aufruf
284 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2) -> Innerer Aufruf
285 elseif nargin == 2
286
287     x = y1;
288     bias = y2;
289     y = 1./(1+exp(-(x+bias)));
290     sigmoid = zeros(nice,1);
291     j=1;
292     for i=linspace(x1, x2, nice)
293         sigmoid(j) = 1./(1+exp(-(i+bias)));
294         j = j + 1;
295     end
296 end
297
298 elseif nargin == 6
299 %% 6 Input-Werte gegeben
300
301 % Ueberpruefung von nice, x1 & x2 fuer beide Output-Varianten notwendig
302 % Test ob nice-Wert eine ganze Zahl und groesser null ist
303 if mod(nice, 1) ~= 0
304     error('Wert fuer "nice" muss ganzzahlig sein!');
305 elseif nice <= 0
306     error('Wert fuer "nice" darf nicht kleiner oder gleich null sein!');
307 elseif nice <= 3
308     warning('Wert fuer "nice" wurde kleiner gleich 3 gewaehlt -> sehr geringe Aufloesung!');
309 end
310 % Gueltigkeit von x1 und x2 testen

```

```

311     if x1 > x2
312         error('SigmoidFunction untere Grenze "x1" darf nicht groesser als obere Grenze "x2" sein!');
313     elseif x1 == x2
314         error('SigmoidFunction untere Grenze "x1" darf nicht gleich oberer Grenze "x2" sein!');
315 end
316
317
318 % 1 Output-Wert -> Ermittlung des sigmoid-Vektors (mit allen notwendig Parametern)
319 % auf Grundlage der Grenzen (x1), (x2), (y1) & (y2) mit Abtastung (nice) und (bias)
320 % sigmoid = SigmoidFunction(nice, x1, x2, y1, y2, bias) -> Aeußerer Aufruf
321 % sigmoid = SigmoidFunction(nice, x1, x2, y1, y2, x) -> Innerer Aufruf
322 if nargout == 1
323
324     if y2 < y1
325         error('Grenzen von SigmoidFunction vertauscht da obere Grenze "y2" kleiner als untere Grenze "y1" gewählt wurde!');
326     elseif y2 == y1
327         error('Grenzen von SigmoidFunction falsch: da obere Grenze "y2" gleich unterer Grenze "y1" gewählt wurde!');
328 end
329
330 bias = x;
331 sigmoid = zeros(nice,1);
332 j=1;
333 for i=linspace(x1, x2, nice)
334     sigmoid(j) = 1./(1+exp(-(i+bias)));
335     j = j + 1;
336 end
337
338 if -y1==y2 % Test ob symetrisch definiert -> einfache Verschiebung in y notwendig & skalieren
339     sigmoid = sigmoid - 0.5; % Verschiebung auf -0.5..0.5
340     sigmoid = sigmoid * 2; % Skalierung auf -1..1
341     sigmoid = sigmoid * y2; % gewünschte Skalierung
342 else
343     range = y2 - y1;
344     sigmoid = sigmoid * range; % Skalierung auf 0..range bzw. 0..(y2-y1)
345     sigmoid = sigmoid + y1; % Verschiebung auf y1..y2 bzw. (0+y1)..((y2-y1)+y1)
346 end
347
348 % 2 Output-Werte -> Ermittlung des sigmoid-Vektors und eines Funktionswertes y
349 % auf Grundlage der Feinheit (nice) in den Grenzen (x1), (x2), (y1) & (y2) und eines Eingangswertes (x)
350 % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2, x) -> Aeußerer & Innerer Aufruf
351 elseif nargout == 2
352
353     % default bias setzen
354     bias = biasDefault;
355     % y & sigmoid fuer nachfolgende Skalierung erzeugen
356     y = 1/(1+exp(-(x+bias)));
357     sigmoid = zeros(nice,1);
358     j=1;
359     for i=linspace(x1, x2, nice)
360         sigmoid(j) = 1./(1+exp(-(i+bias)));
361         j = j + 1;
362     end
363     if y2 < y1
364         error('Grenzen von SigmoidFunction vertauscht da obere Grenze "y2" kleiner als untere Grenze "y1" gewählt wurde!');
365     elseif y2 == y1
366         error('Grenzen von SigmoidFunction falsch: da obere Grenze "y2" gleich unterer Grenze "y1" gewählt wurde!');
367 end
368
369 if -y1==y2 % Test ob symetrisch definiert -> einfache Verschiebung in y notwendig & skalieren
370     if y == 0.5 % y liegt im Koordinatenursprung nach Verschiebung
371         y = 0;
372     else
373         y = y - 0.5; % Verschiebung auf -0.5..0.5
374         y = y * 2; % Skalierung auf -1..1
375         y = y * y2; % gewünschte Skalierung
376     end
377     sigmoid = sigmoid - 0.5; % Verschiebung auf -0.5..0.5
378     sigmoid = sigmoid * 2; % Skalierung auf -1..1
379     sigmoid = sigmoid * y2; % gewünschte Skalierung
380 else
381     range = y2 - y1;
382     y = y * range; % Skalierung auf 0..range bzw. 0..(y2-y1)
383     y = y + y1; % Verschiebung auf y1..y2 bzw. (0+y1)..((y2-y1)+y1)
384     sigmoid = sigmoid * range; % Skalierung auf 0..range bzw. 0..(y2-y1)
385     sigmoid = sigmoid + y1; % Verschiebung auf y1..y2 bzw. (0+y1)..((y2-y1)+y1)
386 end
387 end
388 end

```

```

389 elseif nargin == 7
390     %% 7 Input-Werte gegeben
391
392     % 1 Output-Wert -> Ermittlung eines Funktionswertes y mit allen Parametern angegeben
393     % y = SigmoidFunction(nice, x1, x2, y1, y2, x, bias) -> Aeußerer Aufruf
394     % sigmoid = SigmoidFunction(nice, x1, x2, y1, y2, x, bias) -> Innerer Aufruf
395     if nargout == 1
396
397         % gleich zu (4 Inputs -> 1 Output) da nice, x1 & x2 nicht
398         % berücksichtigt werden bzw. fuer y keine Rolle spielen
399         sigmoid = SigmoidFunction(y1, y2, x, bias);
400
401     % 2 Output-Werte -> Ermittlung des sigmoid-Vektors und eines Funktionswertes y
402     % auf Grundlage der Feinheit (nice), eines Eingangswertes (x) und eines (bias)
403     % [sigmoid, y] = SigmoidFunction(nice, x1, x2, y1, y2, x, bias) -> Innerer & Aeußerer Aufruf
404     elseif nargout == 2
405
406         % gleich zu (4 Inputs -> 1 Output) da nice, x1 & x2 nicht
407         % berücksichtigt werden bzw. fuer y keine Rolle spielen
408         y = SigmoidFunction(y1, y2, x, bias);
409
410     % gleich zu (6 Inputs -> 1 Output) da x in sigmoid keine Rolle spielt
411     sigmoid = SigmoidFunction(nice, x1, x2, y1, y2, bias);
412 end
413
414 else
415     %% Andere Inputs ausser 1..7 sind ungültig
416     error('Ungültige Anzahl Inputs fuer SigmoidFunction!');
417 end
418
419 end

```

## Literatur

- [1] Anna Corves. [www.dasgehirn.info, 2012. <https://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/nervenzellen-im-gespraech>](http://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/nervenzellen-im-gespraech), Letzter Zugriff 21, Oktober 2017.
- [2] Klaus Peter Kratzer. Neuronale Netze. Hanser, 1991.
- [3] Alessandro Mazzetti. Praktische Einführung in Neuronale Netze. Heise, 1996.
- [4] Andrea Schäfers. [www.gehirnlernen.de. <https://www.gehirnlernen.de/gehirn/die-einzelne-nervenzelle-und-wie-sie-mit-anderen-kommuniziert/>](http://www.gehirnlernen.de/gehirn/die-einzelne-nervenzelle-und-wie-sie-mit-anderen-kommuniziert/), Letzter Zugriff 21. Oktober 2017.