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

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 Matalb-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 ressourcenineffizient 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.

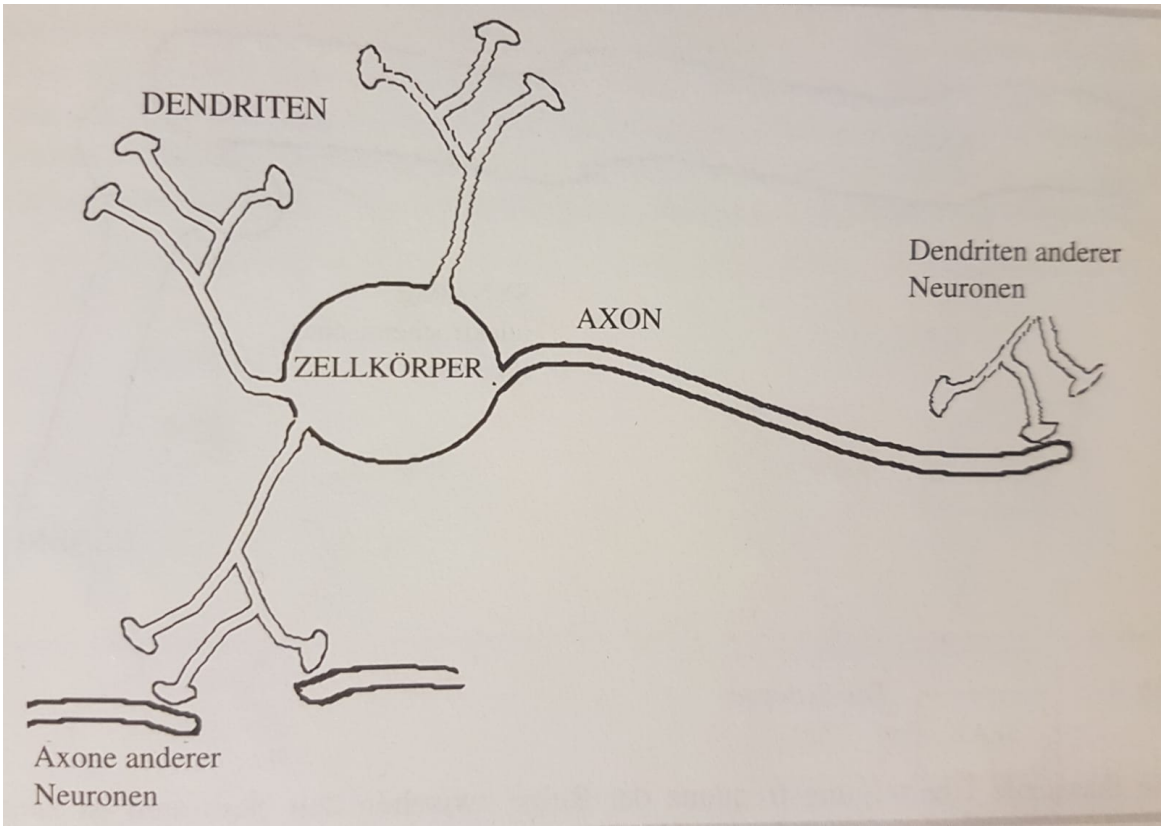

Abbildung 1: Aufbau Neuron [3, S. 11]

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 verschieden 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 Extremfälle 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 vorgeschaltete 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 untertermauern. Die Verarbeitungsmodelle umfassen eine Vielzahl von Annahmen der menschlichen Gehirnstruktur, die nicht eindeutig belegt sind.

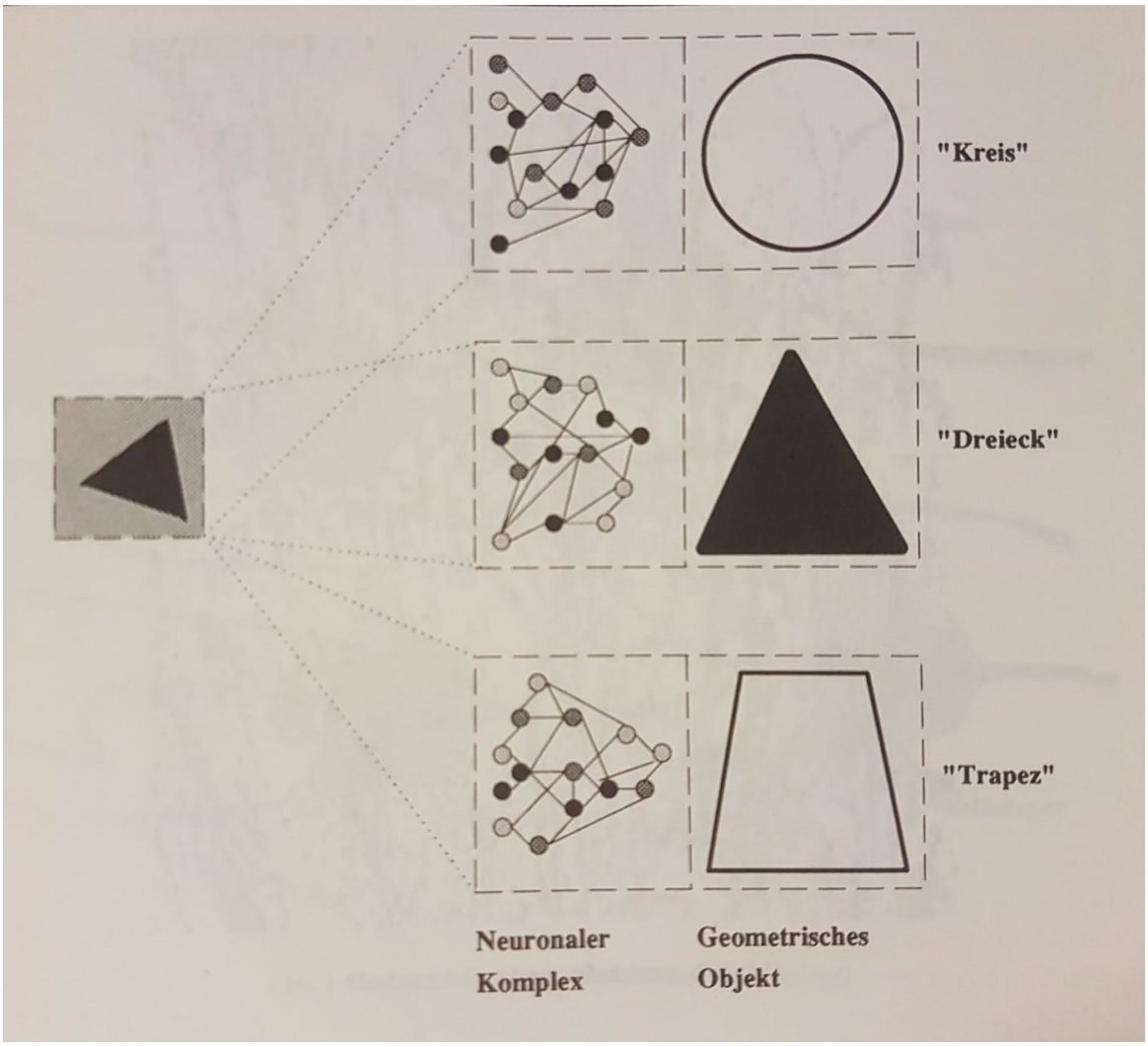

Abbildung 2: Vearbeitung eines visuellen Eindrucks [2, S. 9]

> Neuronale Netzwerke sind Modelle der Gehirnfunktion. Sie versuchen, in Struktur und Funktionsweise Gehirnzellkomplexe nachzubilden und dadruch 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 Modellinhärenten 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 Hardwareimplementierungen.

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 zu ersetzen. 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.

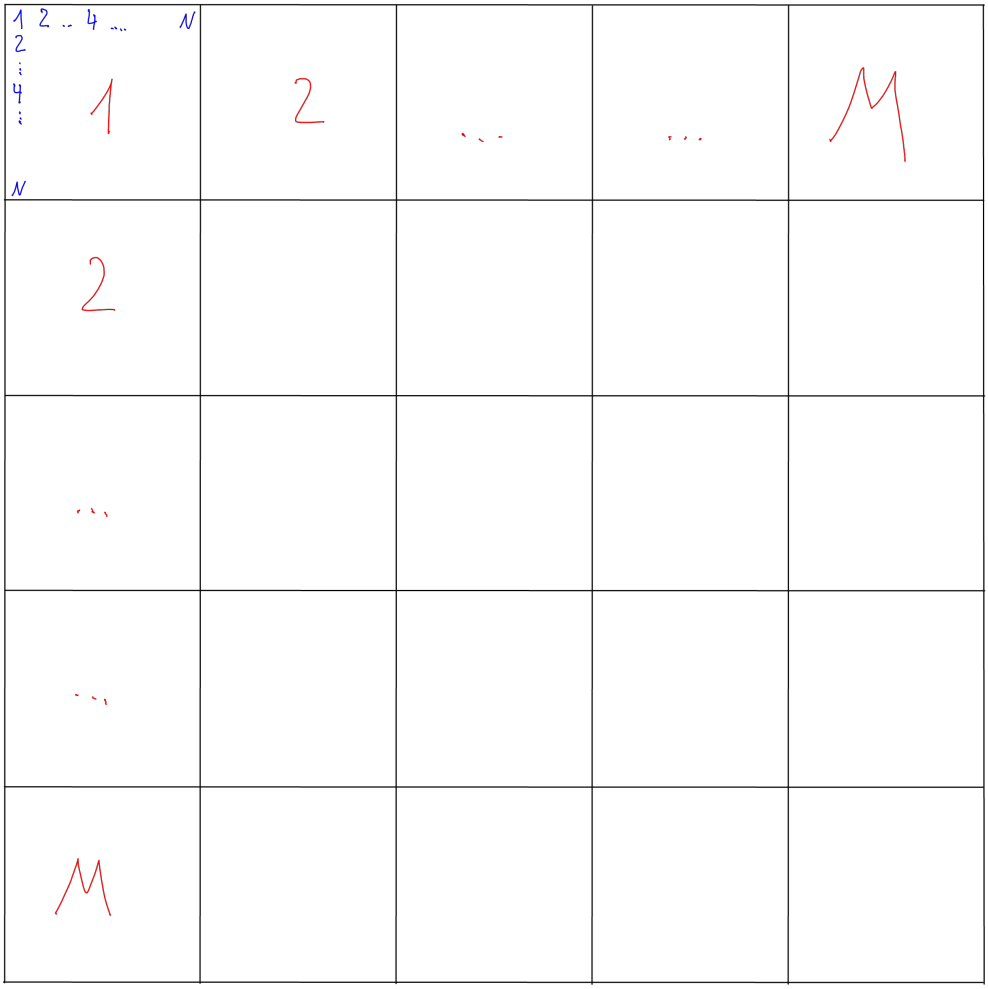

Abbildung 3: Anordnung der Merkmale und Pixel

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.

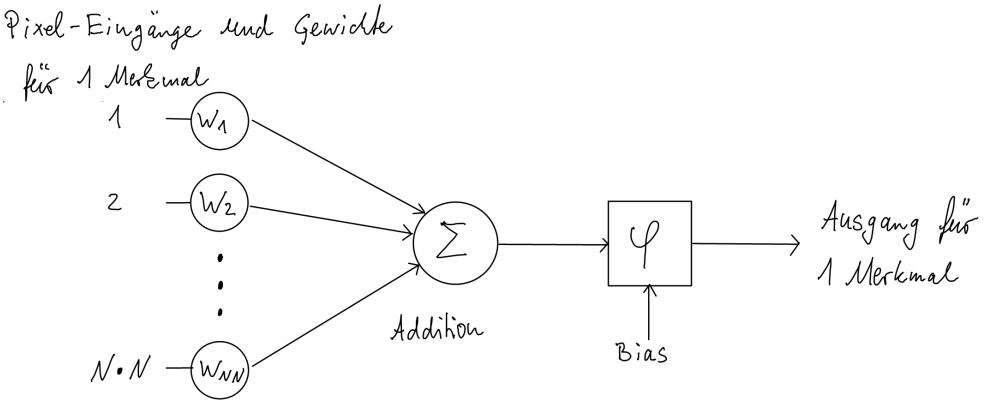

Abbildung 4: Neuron der 1. Ebene

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.

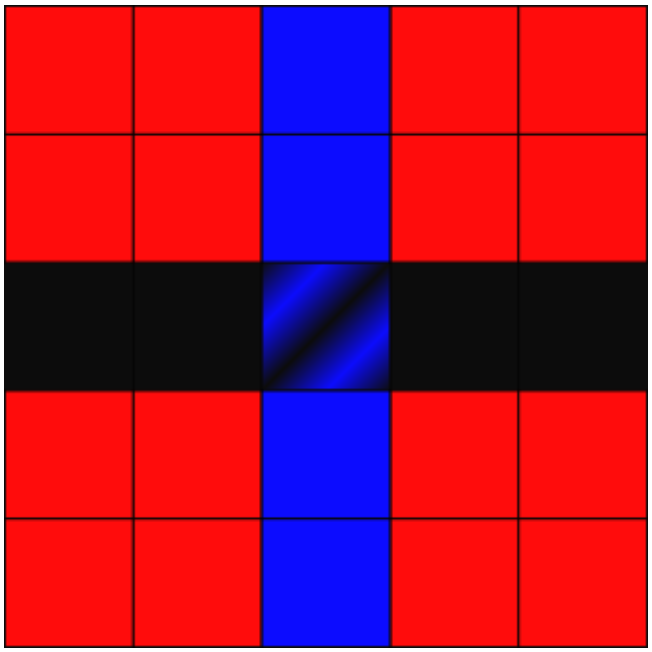

Abbildung 5: Erläuterung der Anordnung für die 2. Ebene

In Abbildung 6 ist die zweite Ebene des Neuronalen Netzes dargestellt. Die Ausgänge der einzelnen Merkmale dienen anhand ihres jeweiligen Merkmaltyps 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.

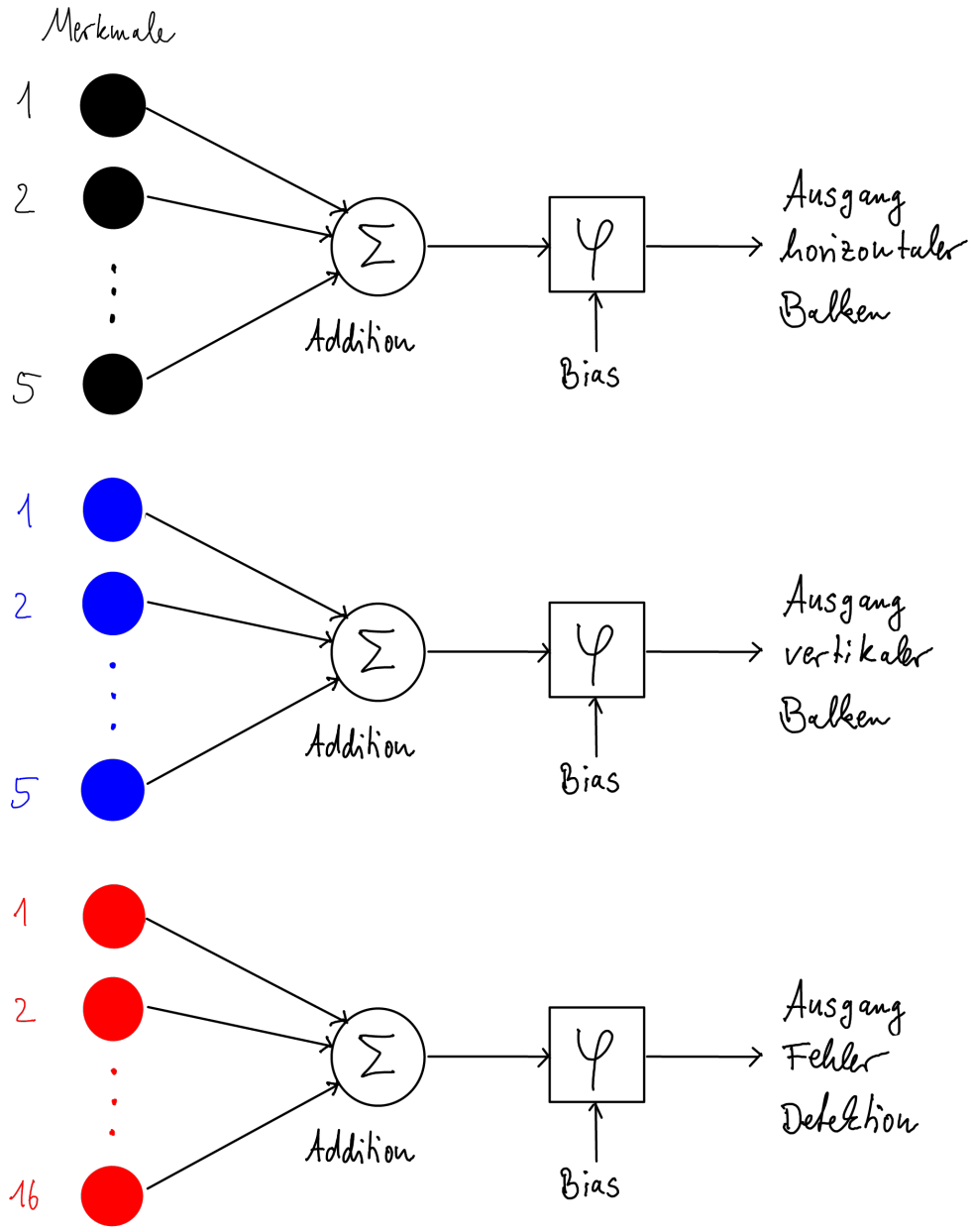

Abbildung 6: Neuronen der 2. Ebene

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 angebenen 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.

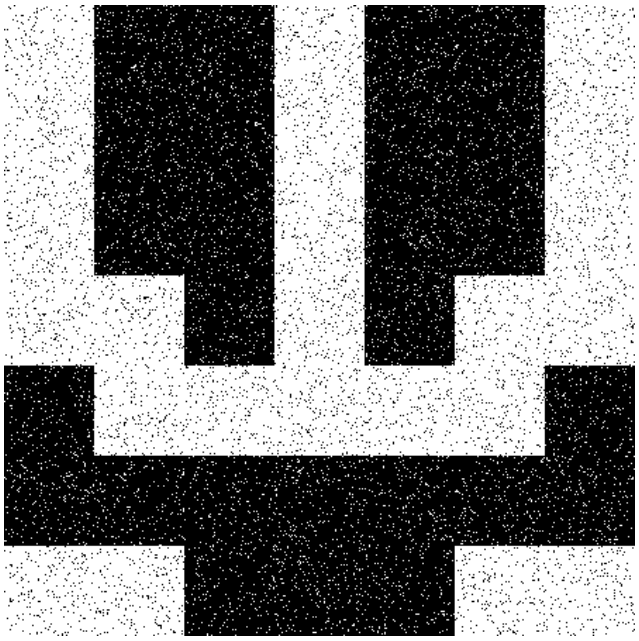

Abbildung 7: Beispielbild 1 mit 10% Rauschen

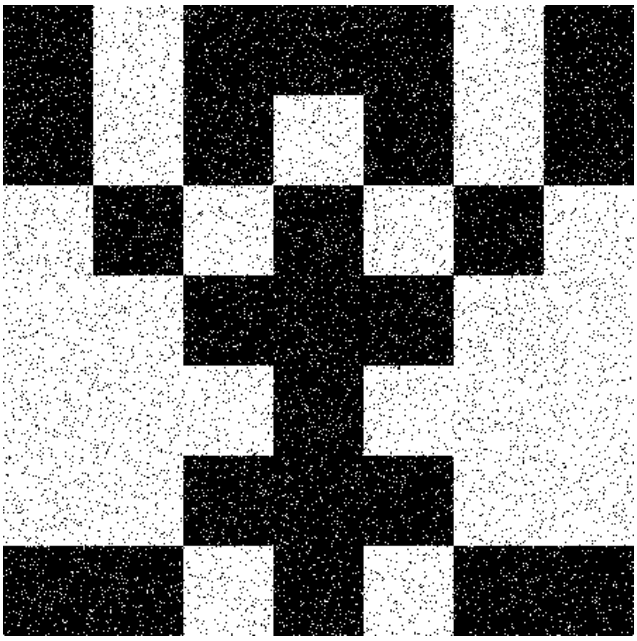

Abbildung 8: Beispielbild 2 mit 10% Rauschen

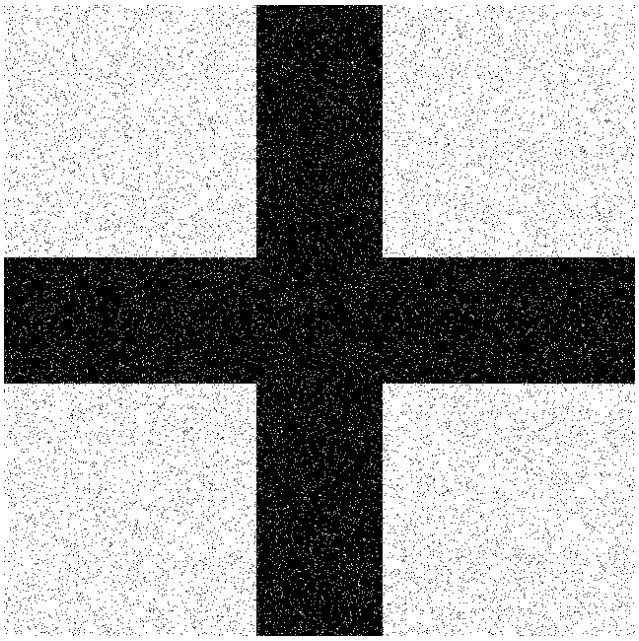

Abbildung 9: Kreuz mit 10% Rauschen

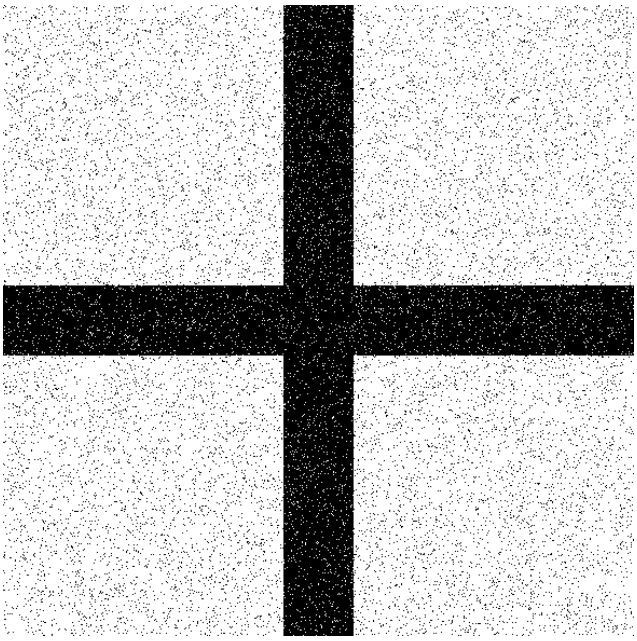

Abbildung 10: 9 Merkmale & 10% Rauschen

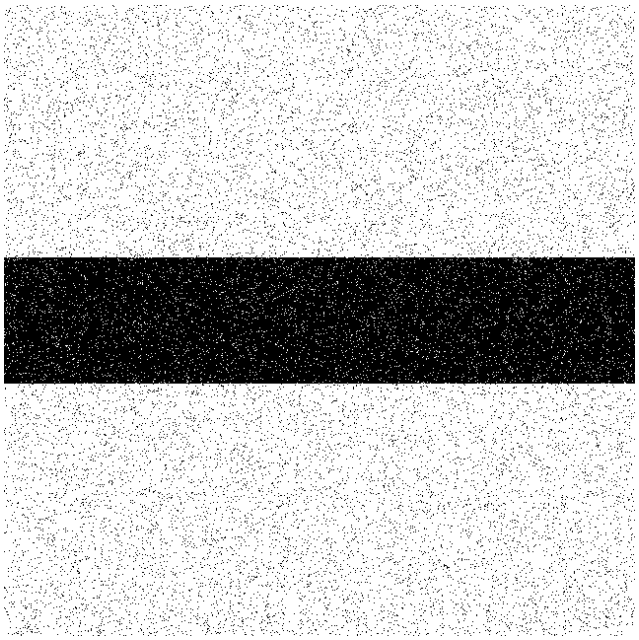

Abbildung 11: Horizontal mit 10% Rauschen

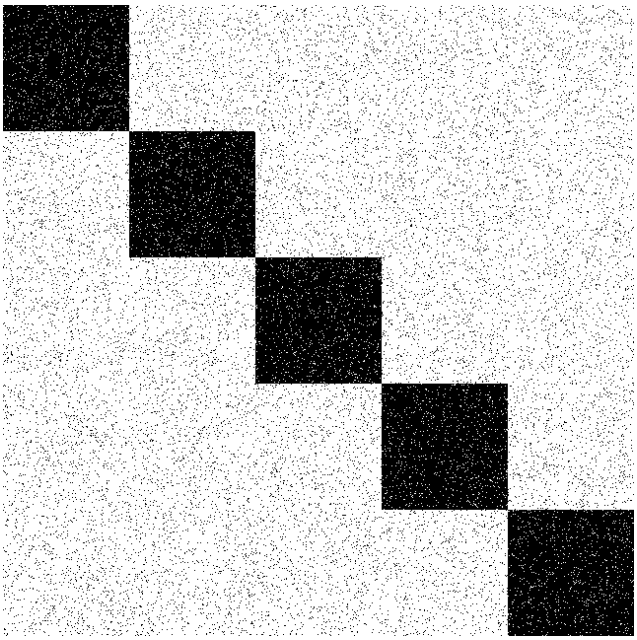

Abbildung 12: Diagonale mit 10% Rauschen

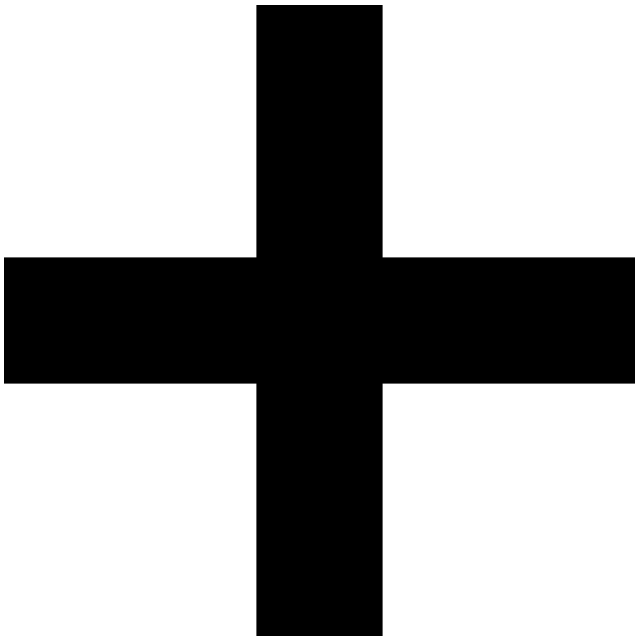

Abbildung 13: Kreuz ohne Rauschen

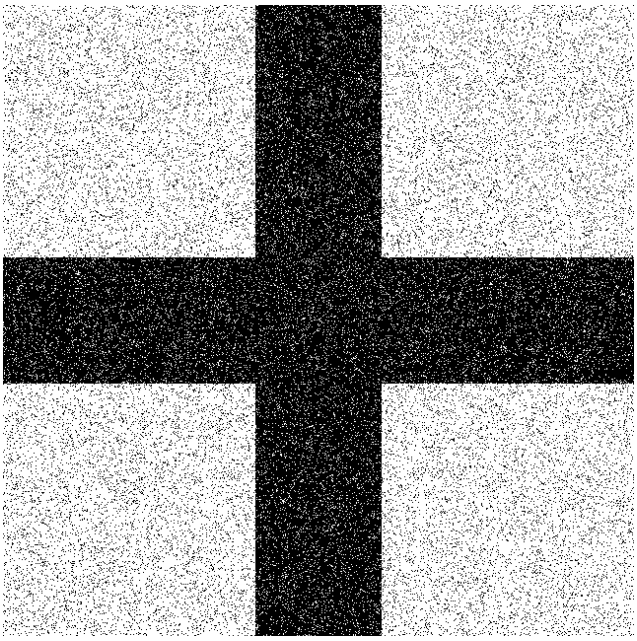

Abbildung 14: Kreuz mit 20% Rauschen

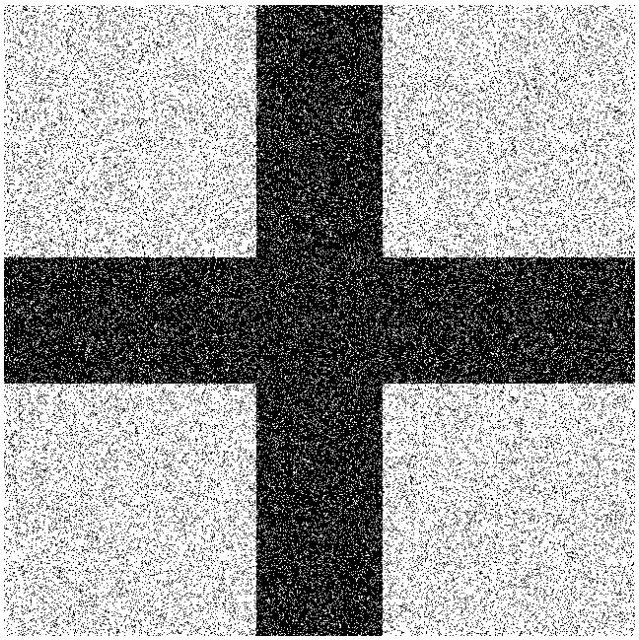

Abbildung 15: Kreuz mit 30% Rauschen

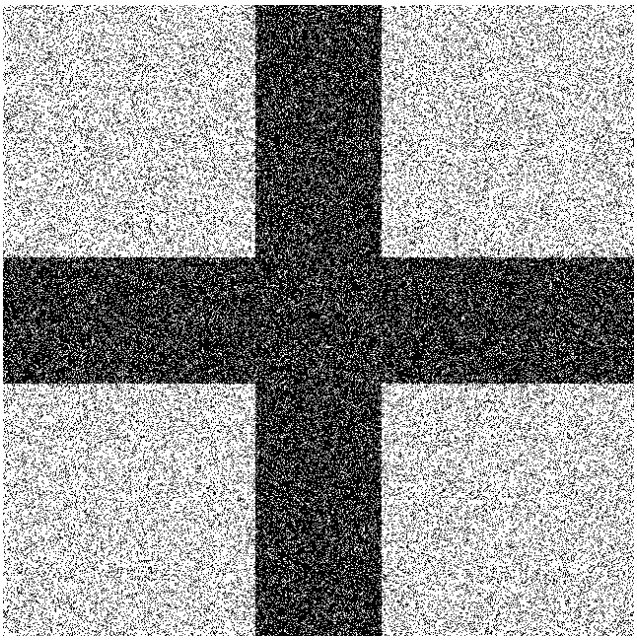

Abbildung 16: Kreuz mit 40% Rauschen

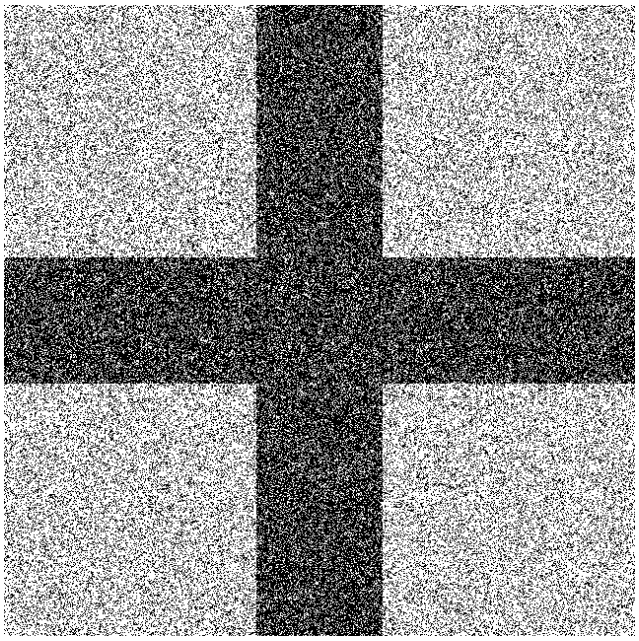

Abbildung 17: Kreuz mit 50% Rauschen

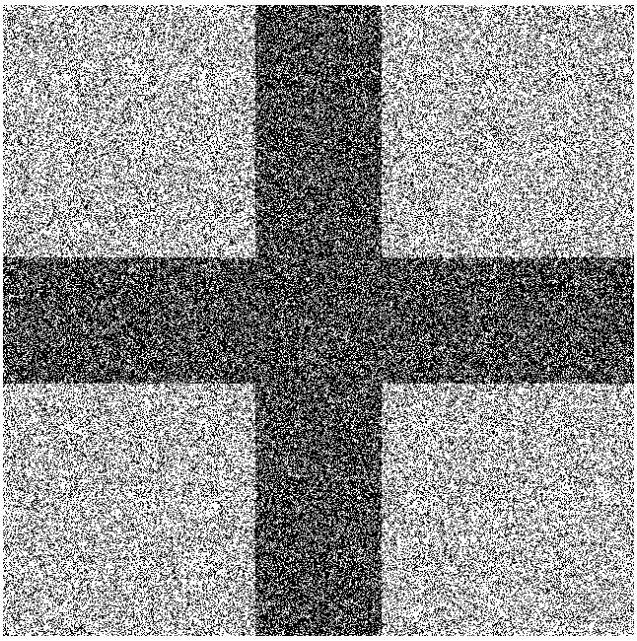

Abbildung 18: Kreuz mit 60% Rauschen

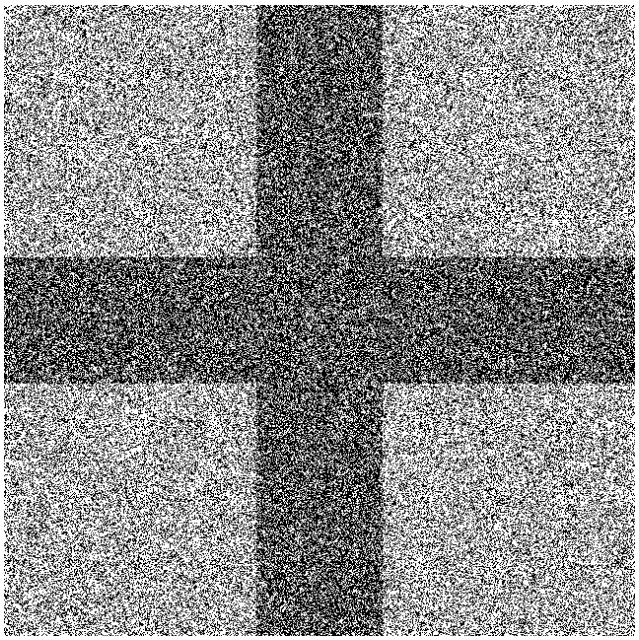

Abbildung 19: Kreuz mit 70% Rauschen

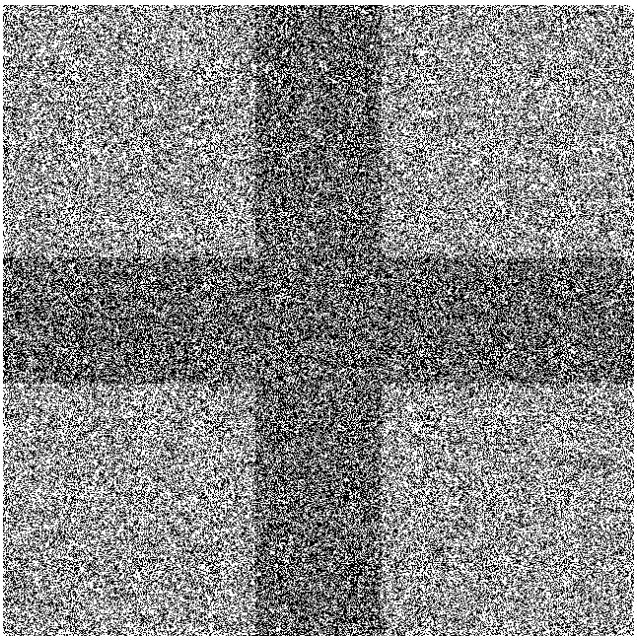

Abbildung 20: Kreuz mit 80% Rauschen

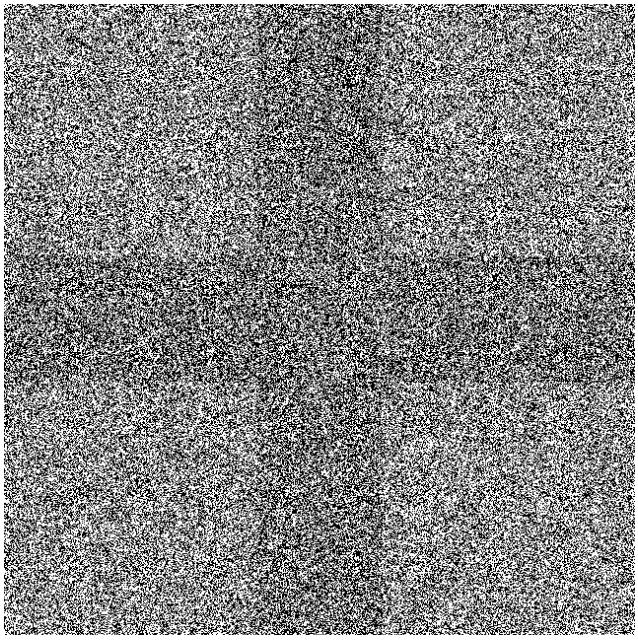

Abbildung 21: Kreuz mit 90% Rauschen

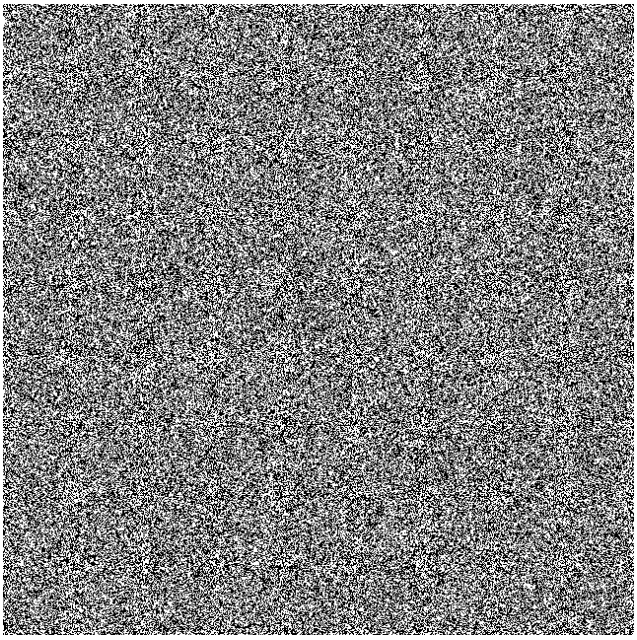

Abbildung 22: Kreuz mit 100% Rauschen

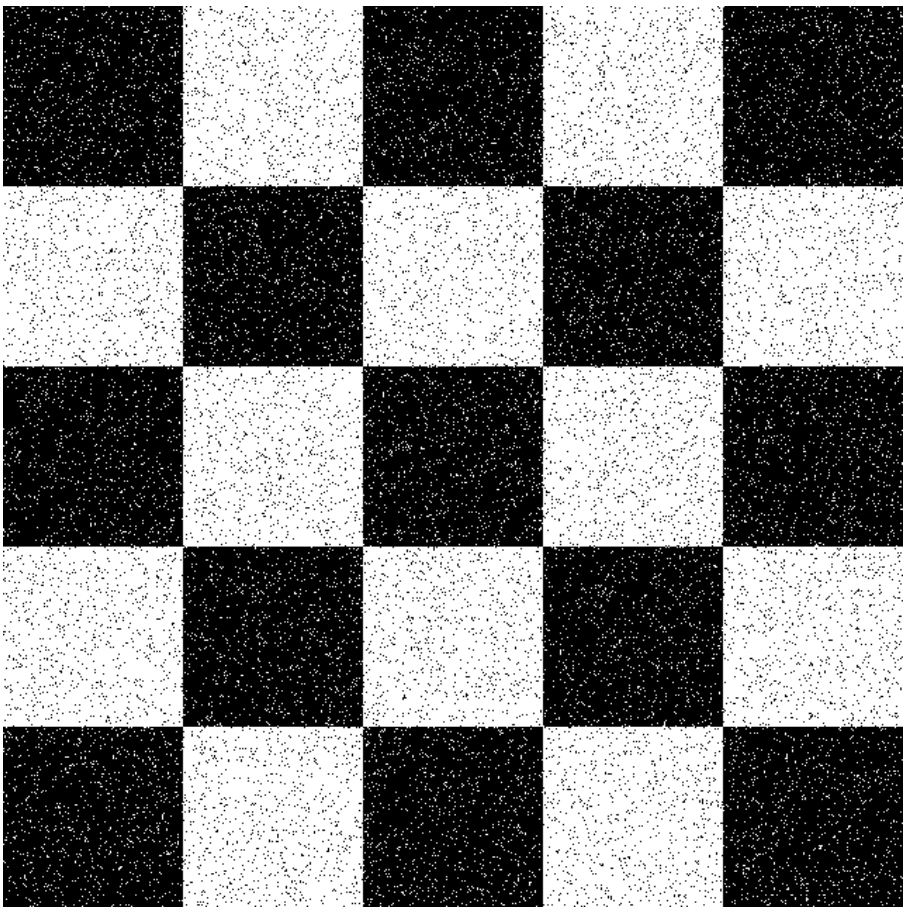

Abbildung 23: Schachbrettmuster mit 10% Rauschen

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 Beipiele ist im Appendix A.3 aufgeführt. In den anderen Unterabschnitten sind für jede Art der Gewichtsmatrizen unterschiedliche Steigungen eingestellt.

4.1. Erzeugung der Gewichtsmatrizen

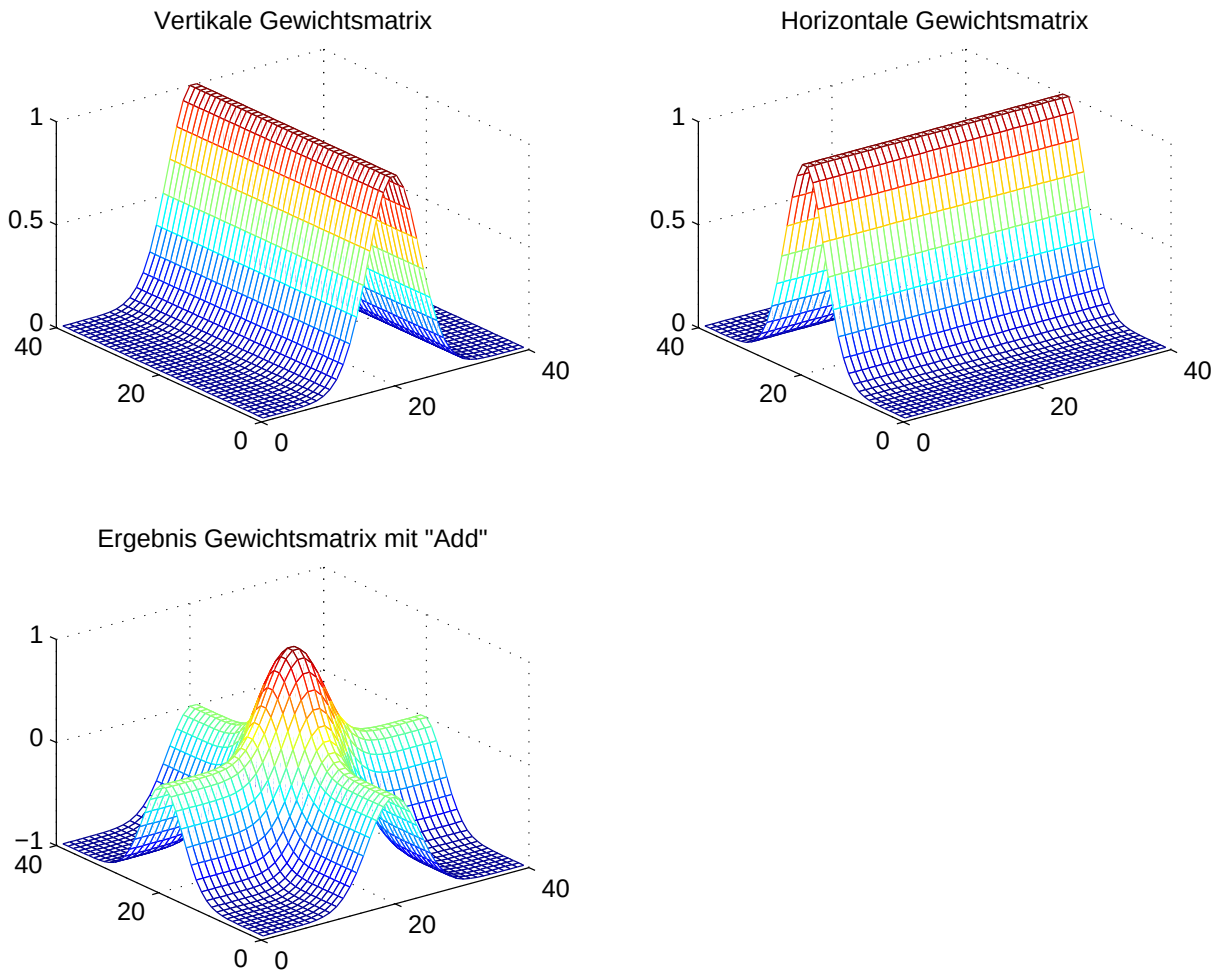

Abbildung 24: Additive Überlagerung mit Slope von 75

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
```

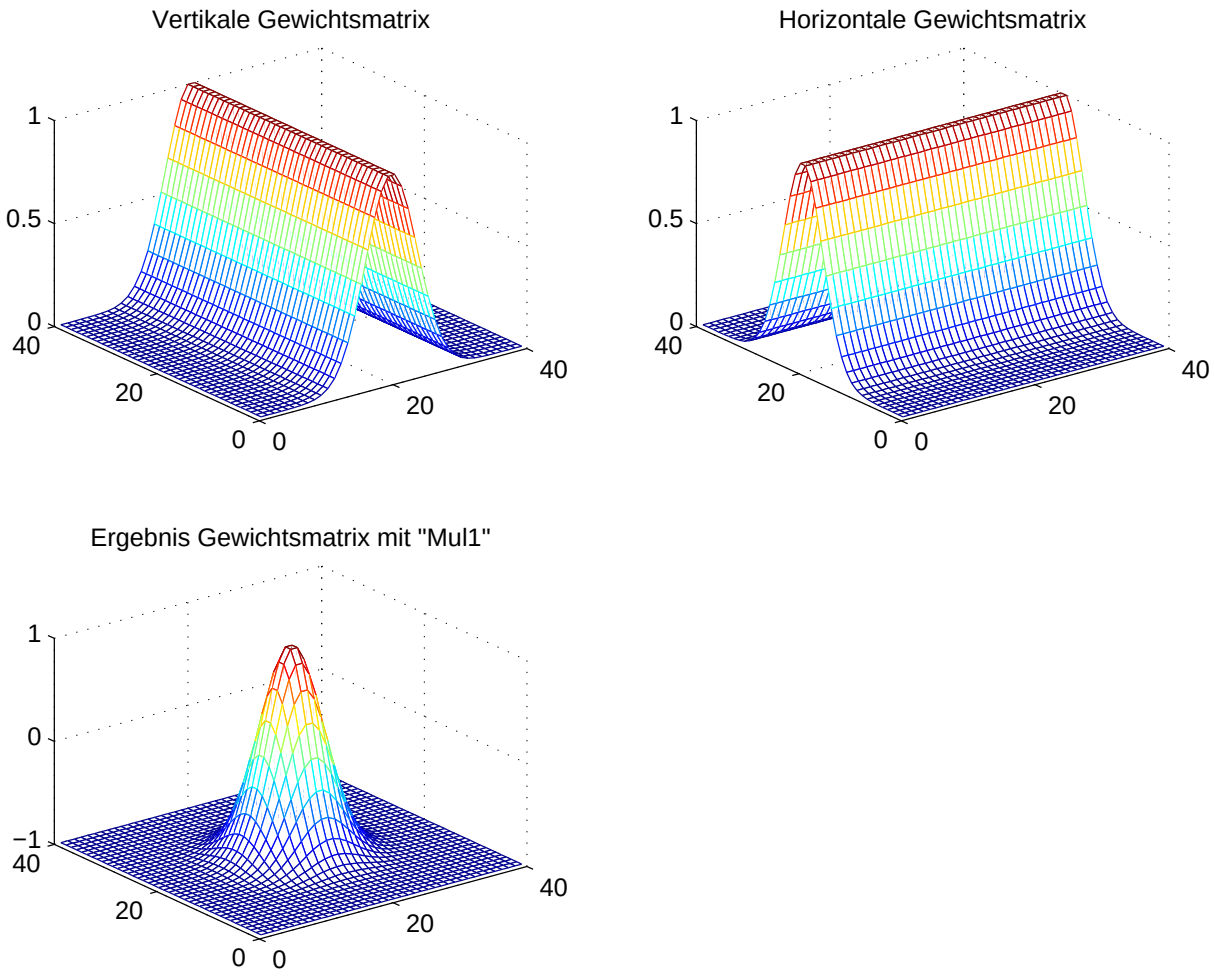

Abbildung 25: Multiplikative Überlagerung Typ 1 mit Slope von 75

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
```

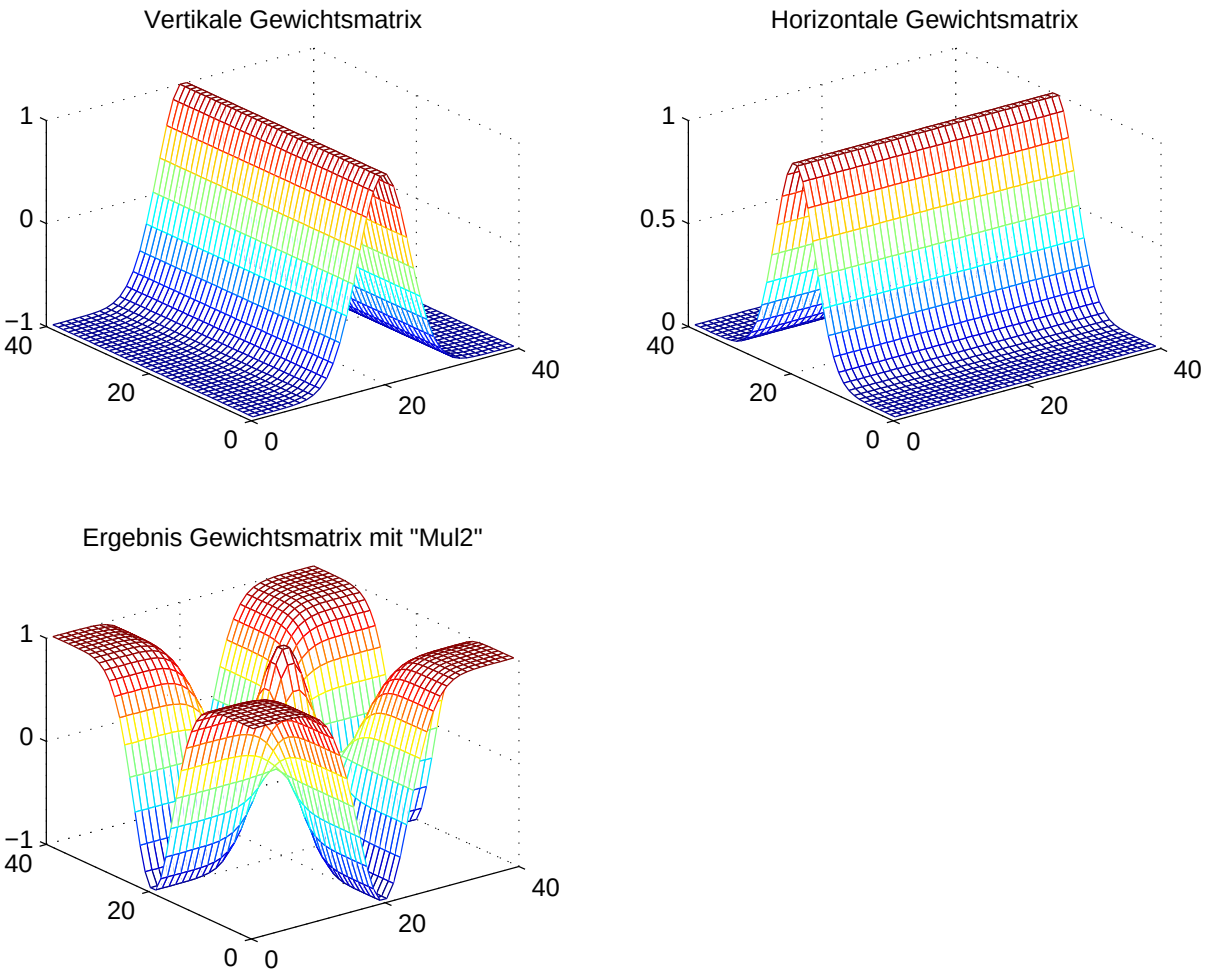

Abbildung 26: Multiplikative Überlagerung Typ 2 mit Slope von 75

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.

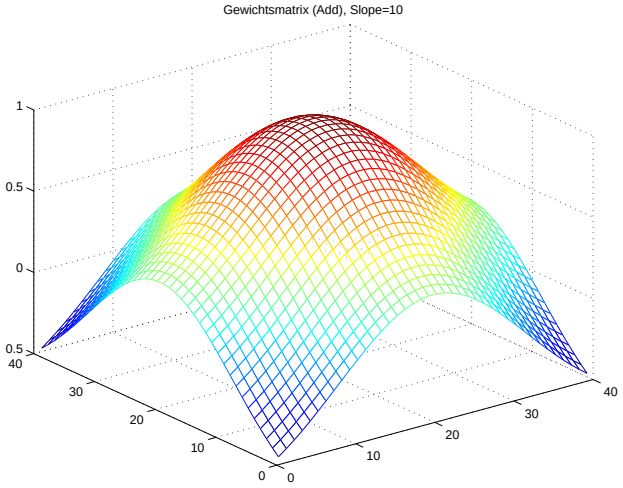

Abbildung 27: Additive mit Slope 10

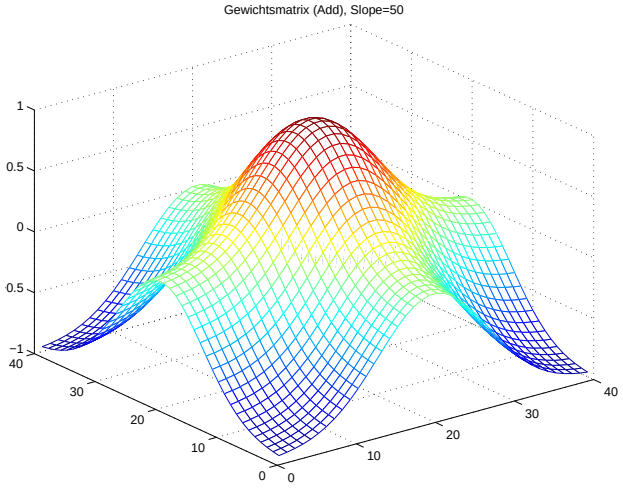

Abbildung 28: Additive mit Slope 50

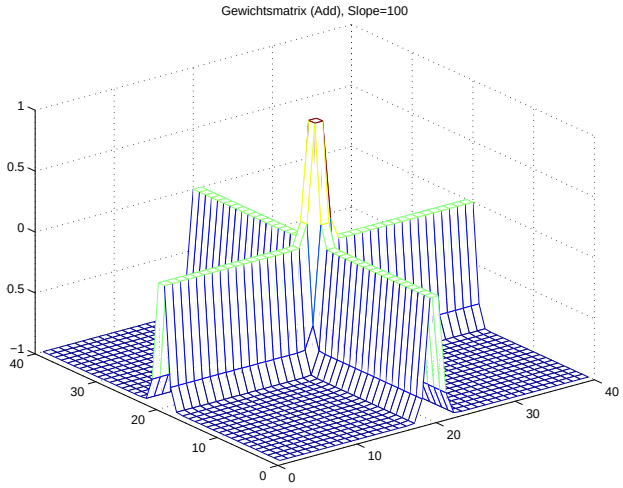

Abbildung 29: Additive Überlagerung mit Slope von 100

4.3. Multiplikative Überlagerung Typ 1

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

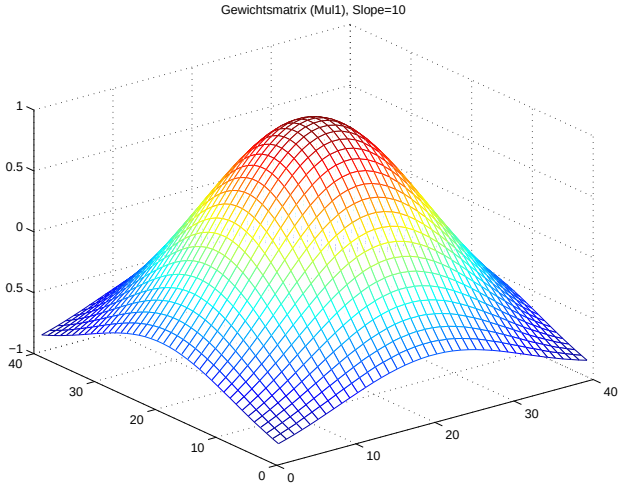

Abbildung 30: Multi-Typ 1 mit Slope von 10

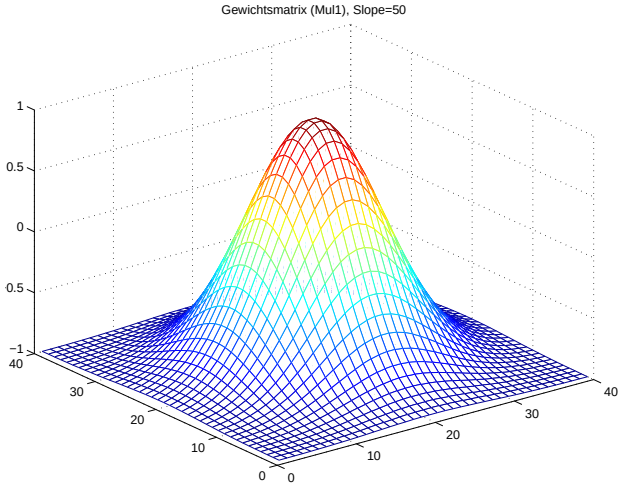

Abbildung 31: Multi-Typ 1 mit Slope von 50

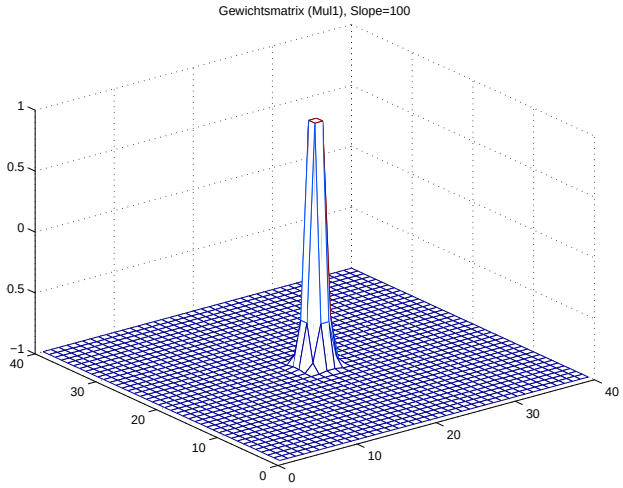

Abbildung 32: Multiplikative Überlagerung Typ 1 mit Slope von 100

4.4. Multiplikative Überlagerung Typ 2

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

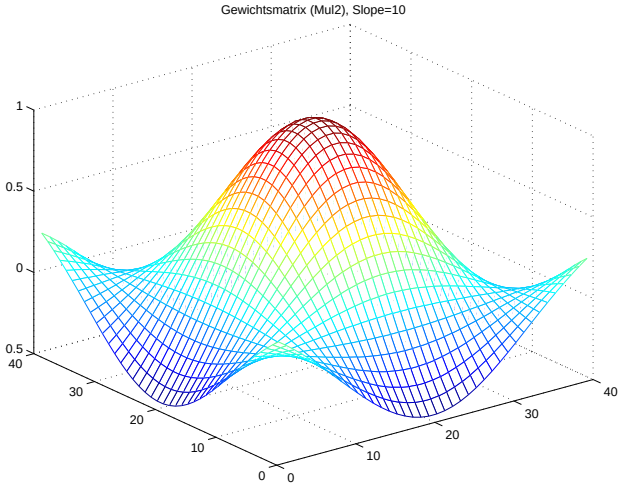

Abbildung 33: Multi-Typ 2 mit Slope von 10

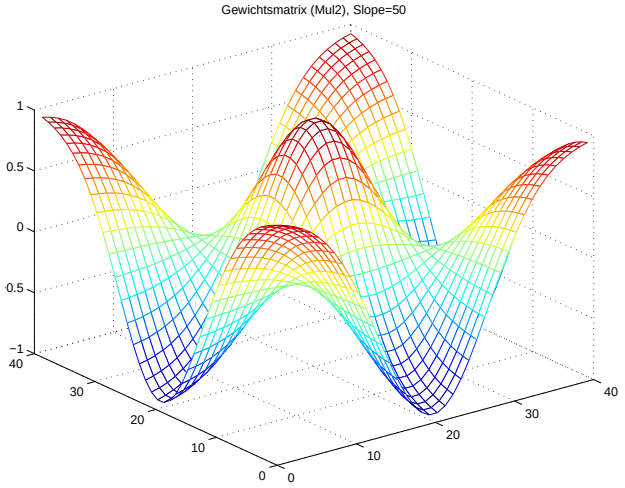

Abbildung 34: Multi-Typ 2 mit Slope von 50

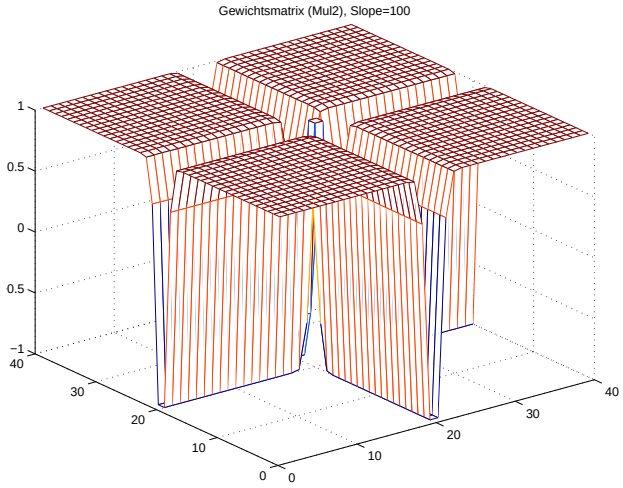

Abbildung 35: Multiplikative Überlagerung Typ 2 mit Slope von 100

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 Erhoehung 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
```

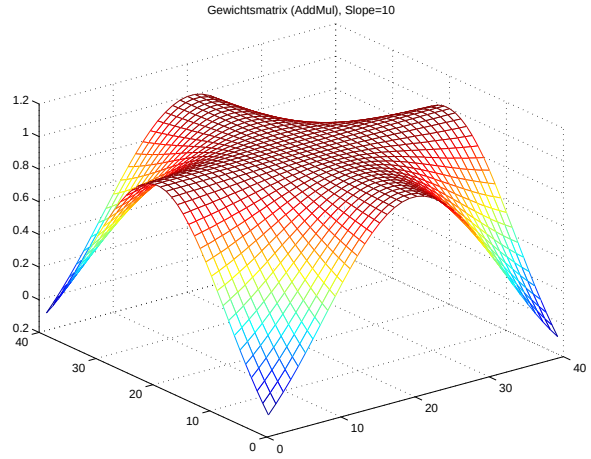

Abbildung 36: AddMul mit Slope 10

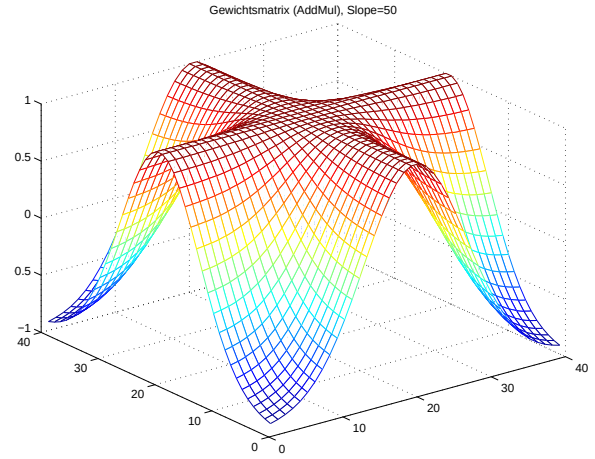

Abbildung 37: AddMul mit Slope 50

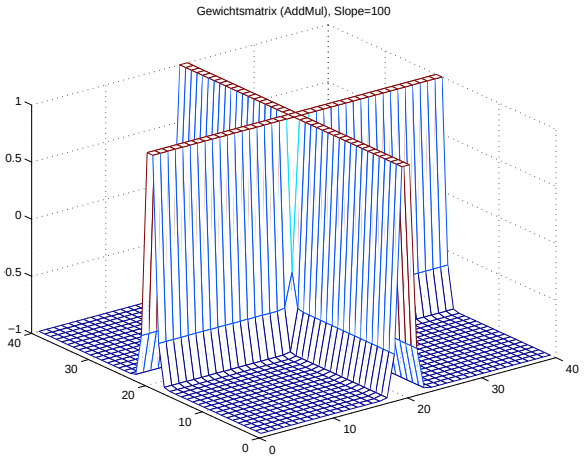

Abbildung 38: Additive und Multiplikative Überlagerung mit Slope von 100

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;
```

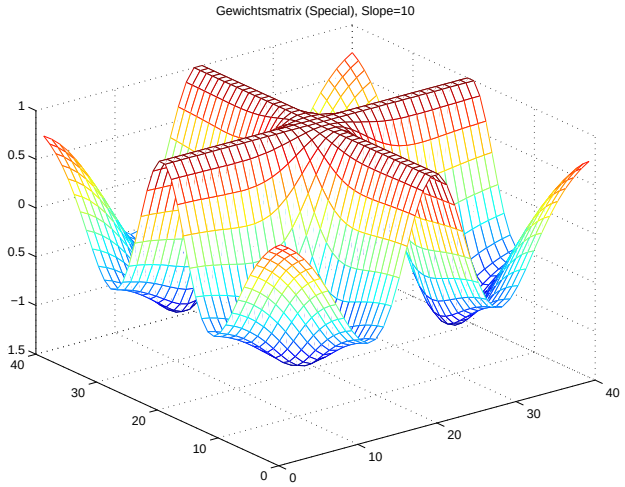

Abbildung 39: Spezielle mit Slope 10

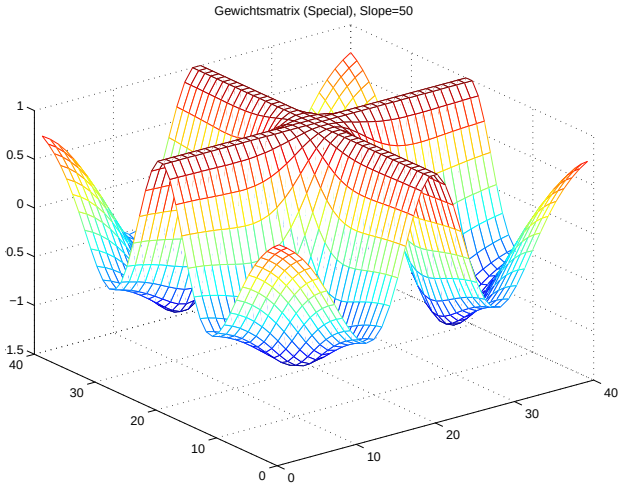

Abbildung 40: Spezielle mit Slope 50

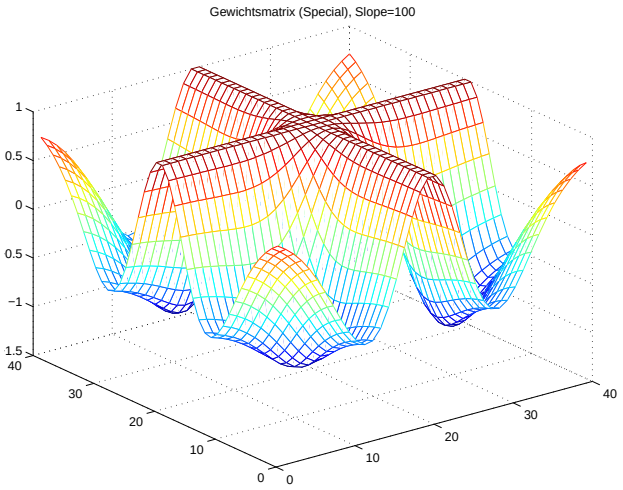

Abbildung 41: Spezielle Überlagerung mit Slope von 100

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 darlegen. In diesem Projekt sind viele Zeile 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. Darüber können auch die Skripte und Dokumentation heruntergeladen werden.

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 'main.m' ist der komplette Ablauf hinterlegt. Für eine bessere Reproduzierbarkeit haben wir zwei weitere Dateien angelegt, in denen wir Grundeinstellungen getroffen haben für Gewichtsmatrizen, die wir in der Dokumentation auswerten.

  - main.m
  - SettingFileAddMul.m Einstellungen für die Gewichts-Matrix 'AddMul'.
  - SettingFileSpecial.m Einstellungen für die Gewichts-Matrix 'Special'.

- Module:

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

  - GetPixelFeatureMatrix.m Diesem Modul wird eine Merkmale-Matrix übergeben und das Modul skaliert dann die Merkmale-Matrix auf eine Pixel-Matrix. Außerdem kann dem Modul ein Rauschwert zwischen 0% und 100% übergeben werden, um die Bilder mit Rauschen zu versehen. Das Modul gibt die Pixel-Matrix zurück und kann die erzeugten Bilder auch gleich als Datei speichern.

- GetNeuronOutput.m 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 Dieses Modul gibt vordefinierte Merkmale-Matrizen zurück, die wir häufig verwendet haben.
- GetGaussWeights.m Dieses Modul gibt eine Gewichts-Matrix zurück. Welche hinterlegt sind kann im Abschnitt 4 nachgeschaut werden.
- GetFeatureOfMatrix.m Diesem Modul kann eine Pixel-Matrix übergeben werden und gibt eine Merkmale-Matrix zurück.
- ConvMatrixToColumn.m Das Modul konvertiert eine Merkmale-Matrix in einen Spaltenvektor.

- Funktionen:

  Diese Funktionen wurden von uns implementiert. Im akutellen Stand wurde allerdings nur die Gauß- und Sigmoid-Funktion verwendet.

  - GaussNormFunction.m
  - SigmoidFunction.m
  - 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.

  - 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

In dem 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	Typ der Gewichts-Matrix (AddMul, Special, Add, Mul1, Mul2)
8	inFeatureType	voreingestellten Merkmale-Matrizen (Cross, H_Line, V_Line)
9	noise	Verrauschungsgrad zwischen 0% und 100%
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 haben 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' und 'SettingFileSpecial.m'. Beide Systeme wurde empirisch eingestellt und es wurde versucht ab 50% Rauschen keine vernünftigen Ergebnisse mehr zu liefern. Außerdem wurden die Bias-Werte 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 richtig detektiert worden. An dieser Stelle werden die addierten Werte der einzelnen Pixel auf eine Sigmoid-Funktion gegeben und grafisch ausgewertet.

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 'v-Balken' detektiert. Die letzte Grafik der zweiten Neuronen Ebene ist die '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.

6.1. AddMul Gewichtsmatrix

6.1.1. H-Balken

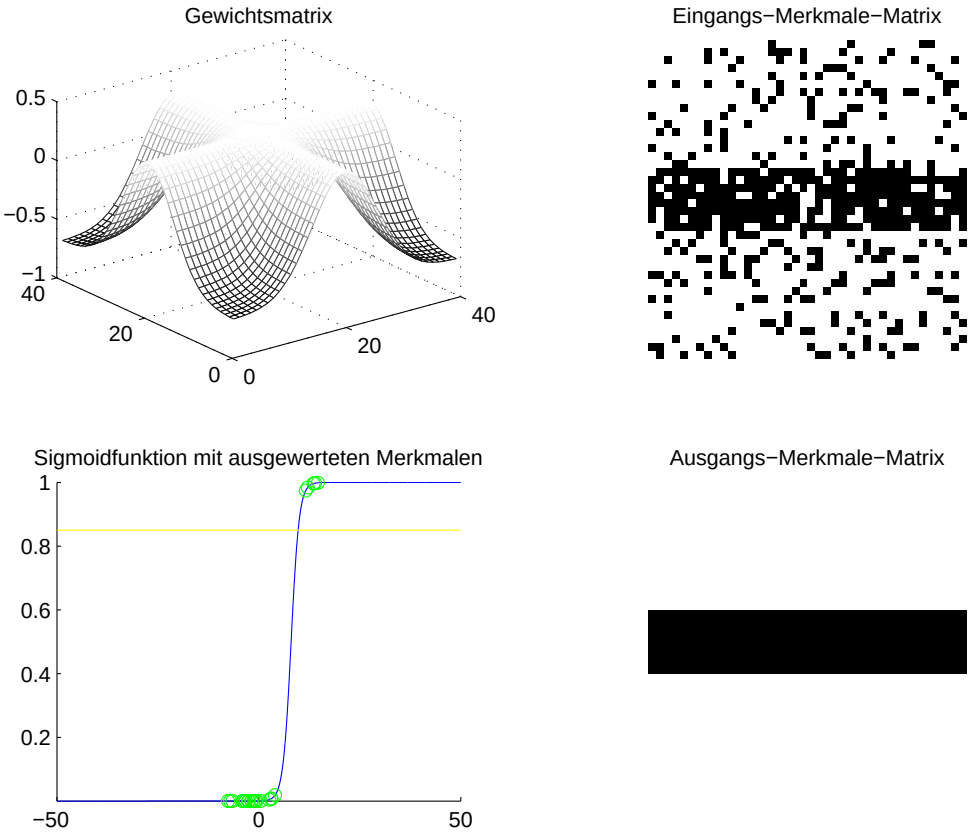

Abbildung 42: AddMul, H-Balken, 40% Rauschen, 1. Neuronen Ebene

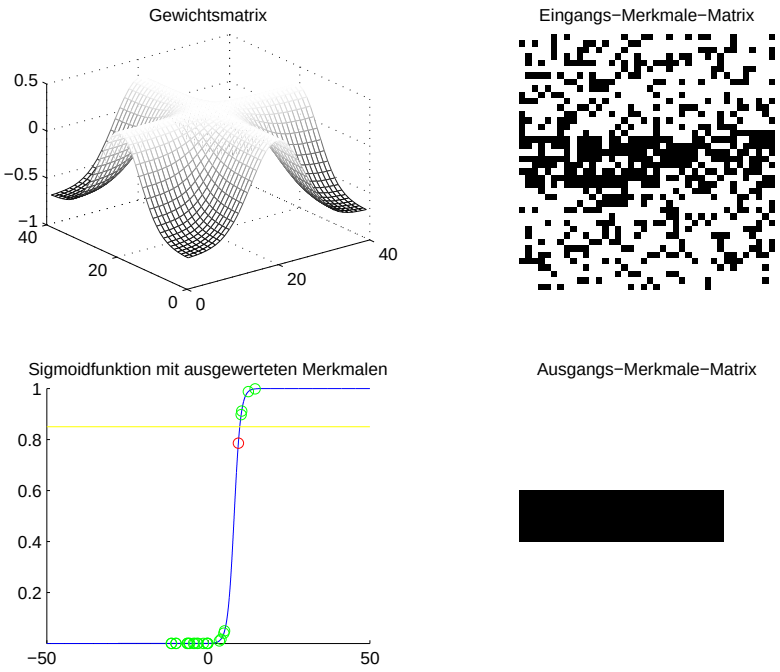

Abbildung 44: AddMul, H-Balken, 60% Rauschen, 1. Neuronen Ebene

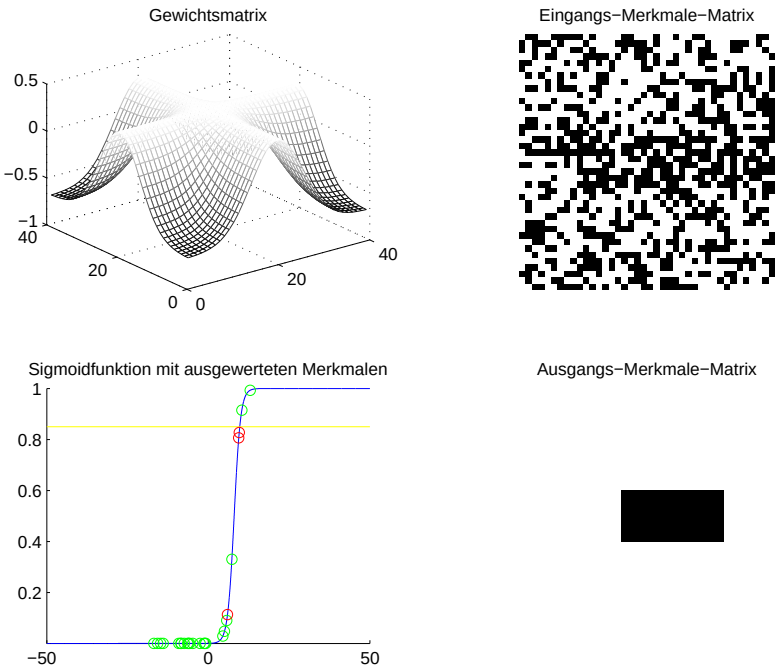

Abbildung 46: AddMul, H-Balken, 80% Rauschen, 1. Neuronen Ebene

6.1.2. V-Balken

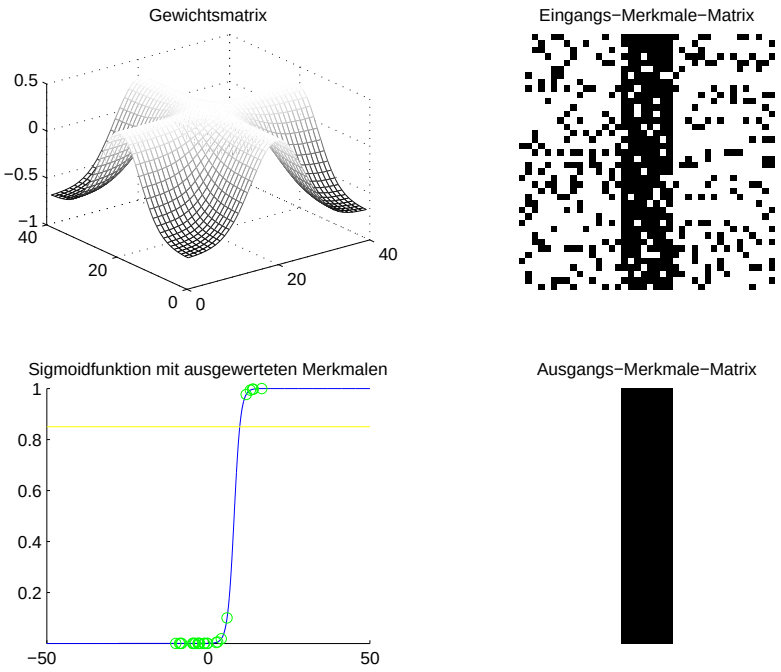

Abbildung 48: AddMul, V-Balken, 40% Rauschen, 1. Neuronen Ebene

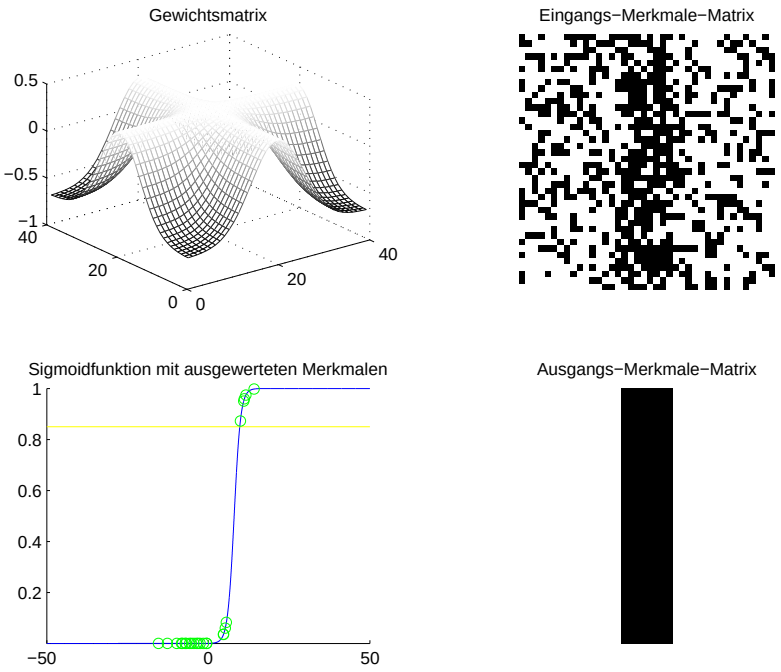

Abbildung 50: AddMul, V-Balken, 60% Rauschen, 1. Neuronen Ebene

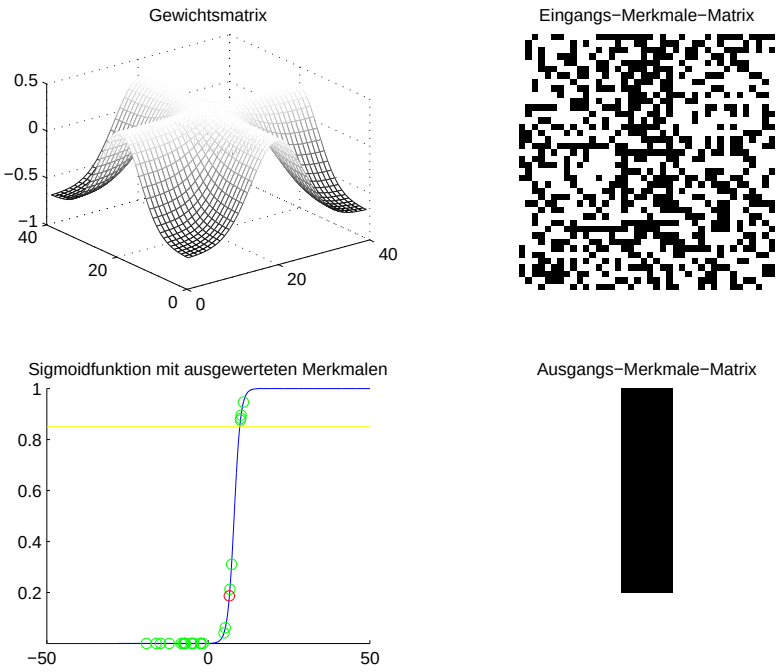

Abbildung 52: AddMul, V-Balken, 80% Rauschen, 1. Neuronen Ebene

6.1.3. Kreuz

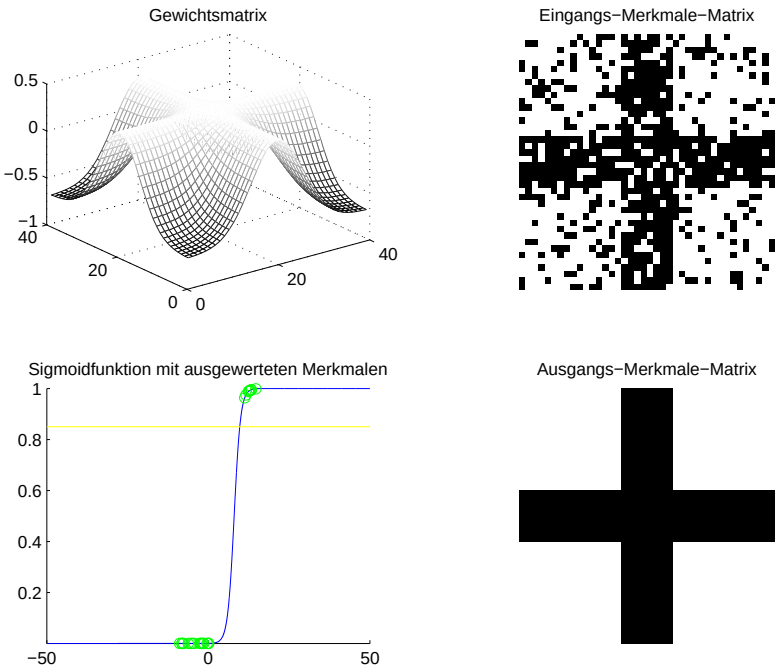

Abbildung 54: AddMul, Kreuz, 40% Rauschen, 1. Neuronen Ebene

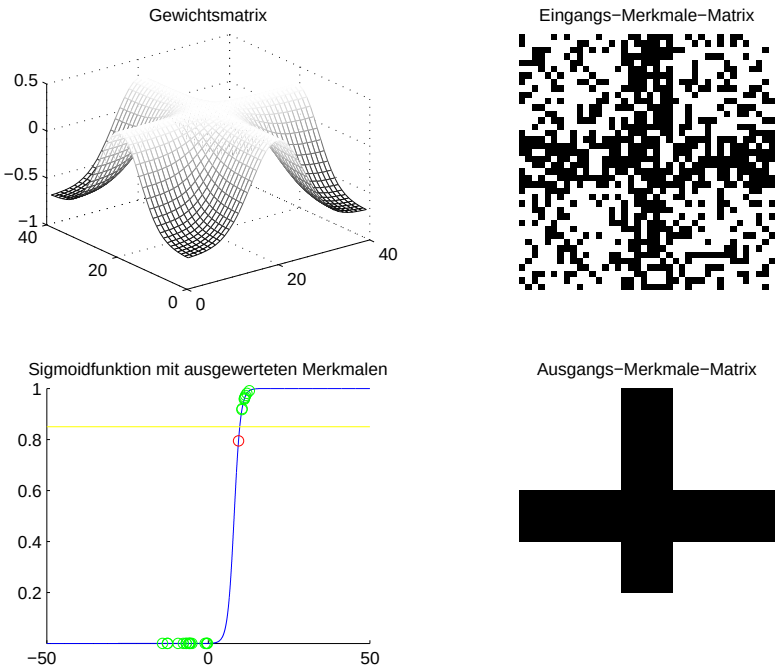

Abbildung 56: AddMul, Kreuz, 60% Rauschen, 1. Neuronen Ebene

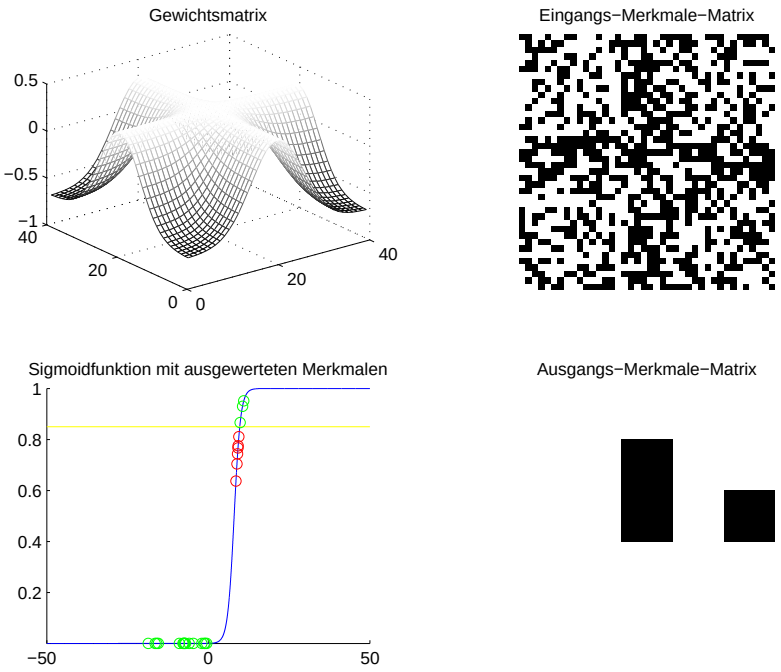

Abbildung 58: AddMul, Kreuz, 80% Rauschen, 1. Neuronen Ebene

6.2. Special Gewichtsmatrix

6.2.1. H-Balken

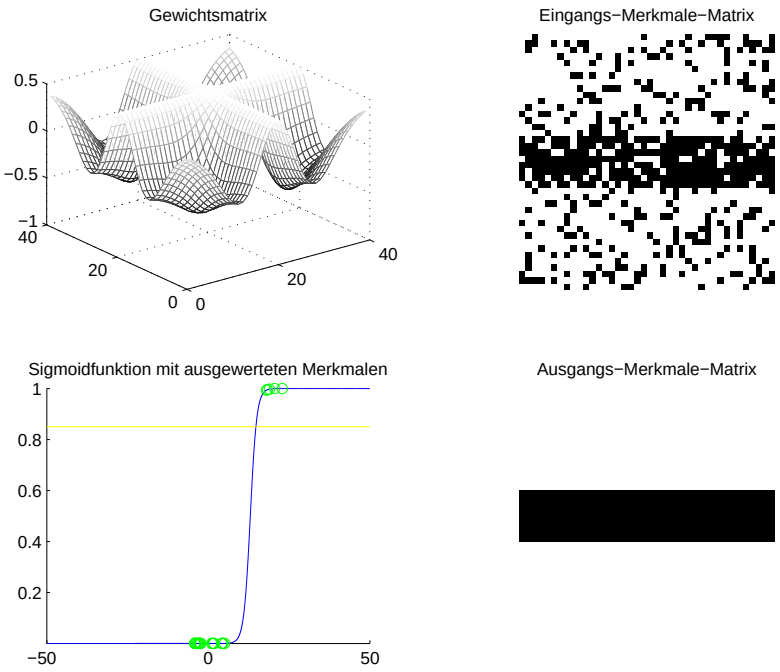

Abbildung 60: Special, H-Balken, 40% Rauschen, 1. Neuronen Ebene

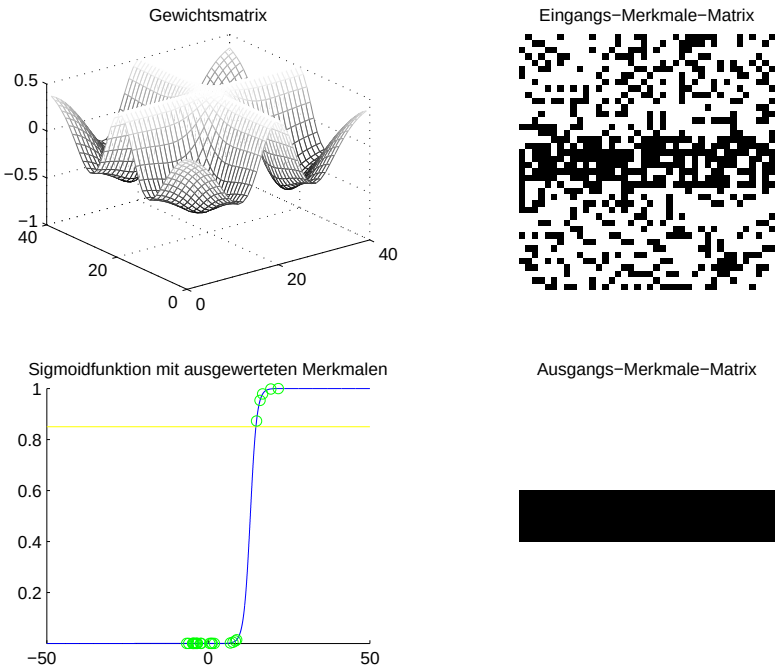

Abbildung 62: Special, H-Balken, 60% Rauschen, 1. Neuronen Ebene

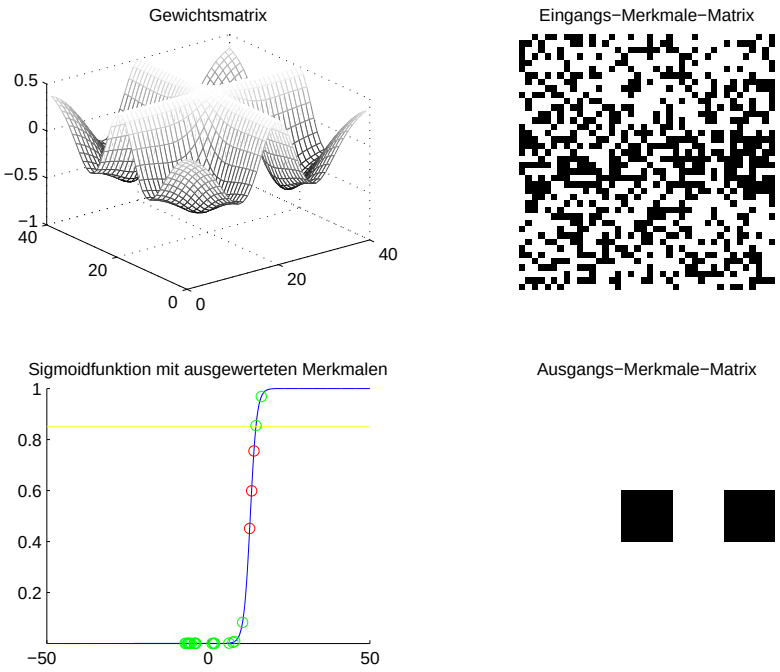

Abbildung 64: Special, H-Balken, 80% Rauschen, 1. Neuronen Ebene

6.2.2. V-Balken

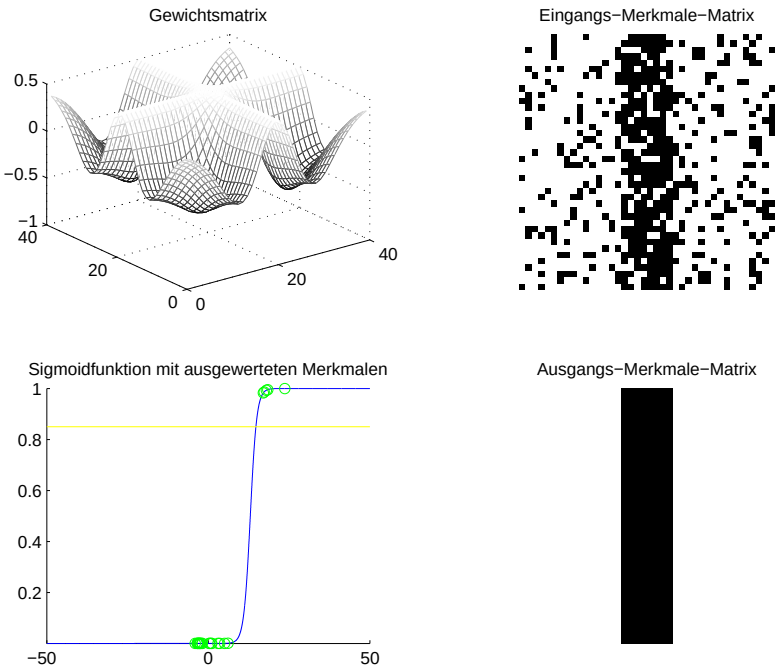

Abbildung 66: Special, V-Balken, 40% Rauschen, 1. Neuronen Ebene

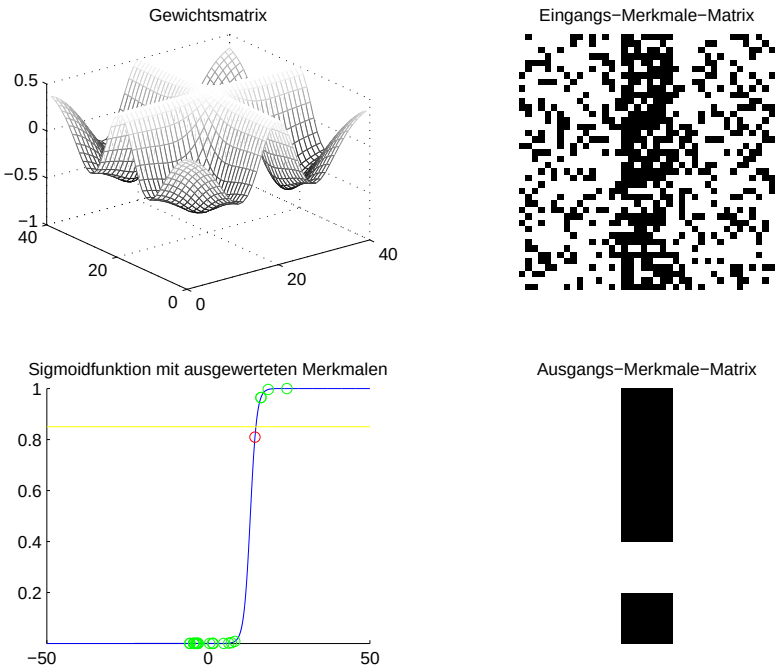

Abbildung 68: Special, V-Balken, 60% Rauschen, 1. Neuronen Ebene

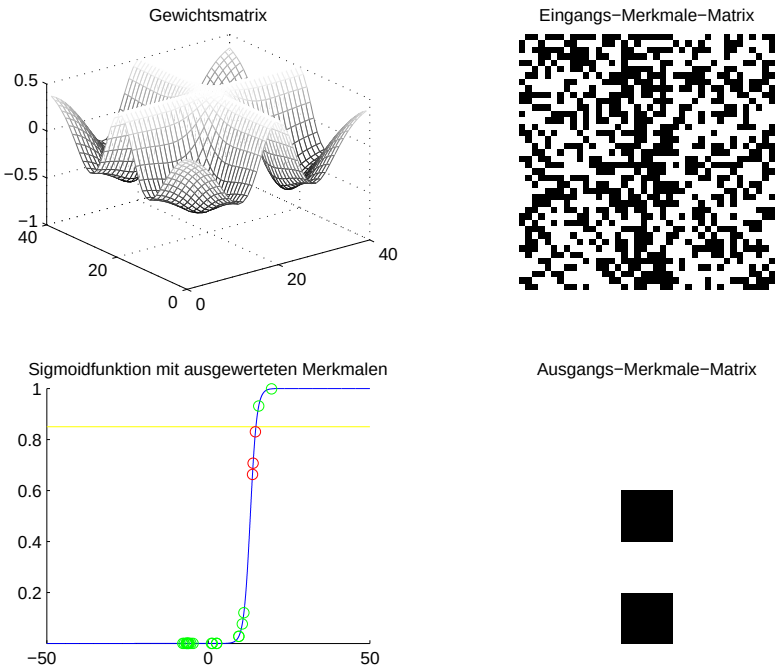

Abbildung 70: Special, V-Balken, 80% Rauschen, 1. Neuronen Ebene

6.2.3. Kreuz

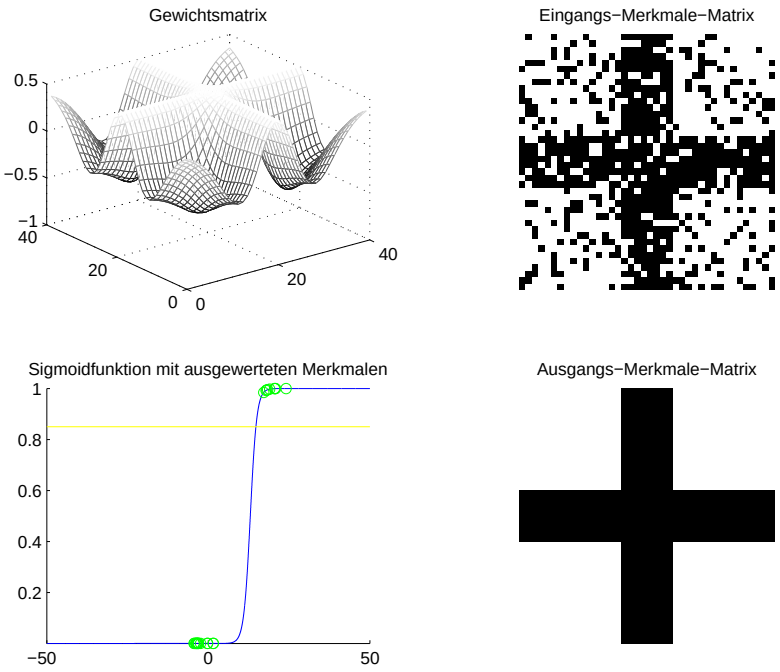

Abbildung 72: Special, Kreuz, 40% Rauschen, 1. Neuronen Ebene

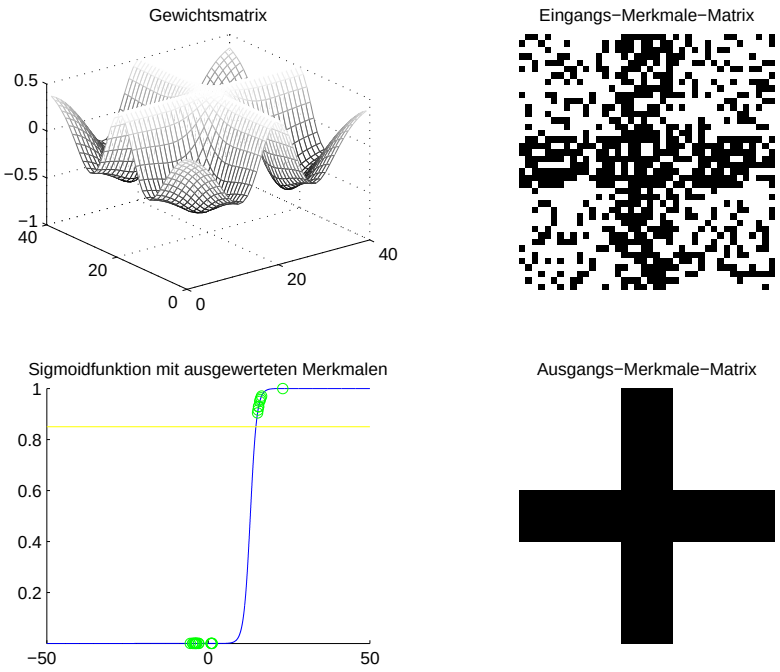

Abbildung 74: Special, Kreuz, 60% Rauschen, 1. Neuronen Ebene

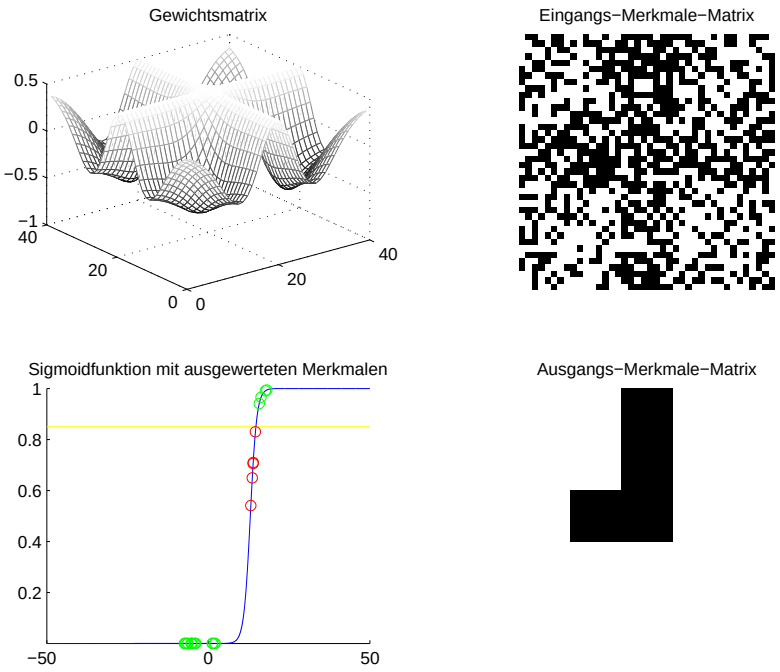

Abbildung 76: Special, Kreuz, 80% Rauschen, 1. Neuronen Ebene

7. Fazit

In diesem Projekt haben wir ein neuronales Netz zur Detektion von horizontalen und 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 wächst Linear mit der Anzahl der Merkmalen. 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 besser einstellbar 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.

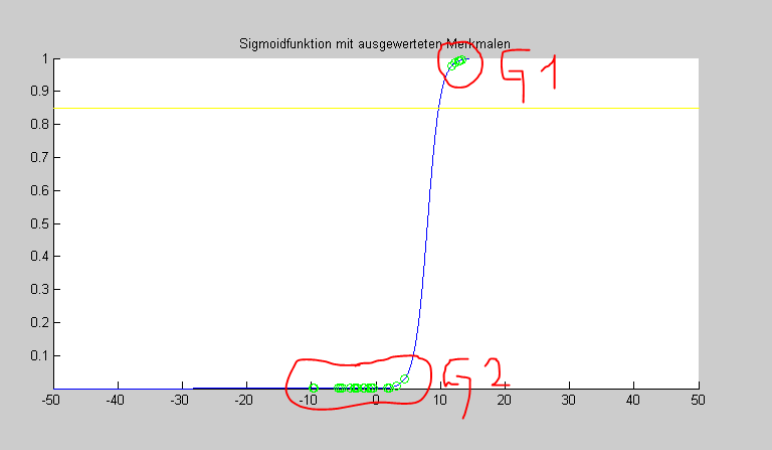

Abbildung 78: Merkmale-Gruppen

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 zusammen-

hängend 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 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.

A. Appendix

A.1. GetPixelFeatureMatrix.m

```
function pixelFeatureMatrix = GetPixelFeatureMatrix(pixelCnt, featureCnt, noise, featureMatrix, name)
    % Funktion skaliert uebergebene Merkmale-Matrix in eine Pixel-Matrix
    %
    % pixelCnt - Anzahl der Pixel in x-Richtung pro Merkmal - mindestens 1
    % featureCnt - Anzahl der Merkmale in x-Richtung - mindestens 1
    % noise - Verrauschungsgrad zwischen 0 und 100%
    % featureMatrix - Merkmale-Matrix, welche Bereiche aktiv sein sollen
    % filename - optional - Name der Ausgabedatei

    % Ueberpruefung der Merkmale-Matrix
    sizeInput = size(featureMatrix);
    if (sizeInput(1) ~= featureCnt)
        error('featureCnt passt nicht zu featureMatrix')
    end
    clear sizeInput
    if max(max(featureMatrix)) > 1
        error('in the featureMatrix are only allowed 0 and 1')
    end

    % Ueberpruefung der maximal erlaubten Bildbreite
    % es koennen nur uint16-Werte beruecksichtigt werden
    if pixelCnt*featureCnt > 2^16-1
        error('Pixelbreite des Bildes ist zu gross.')
    end

    % Ueberpruefung des erlaubten Noise-Level-Bereichs -> 0 bis 100
    if ((noise > 100) && (0 < noise))
        error('noise level is not allowed')
    end
    noise = uint8(noise/2); % 50 ist maximal verrauscht

    % Umbenennungen fuer eventuelle Erweiterung in x- und y-Richtung
    pixelCntX = pixelCnt; % Anzahl der Pixel pro Merkmal in x-Richtung
    pixelCntY = pixelCnt; % Anzahl der Pixel pro Merkmal in y-Richtung
    featureCntX = featureCnt; % Anzahl der Merkmale horizontal
    featureCntY = featureCnt; % Anzahl der Merkmale vertikal

    % Berechnen der maximalen Werte
    pixelCntX_N = pixelCntX * featureCntX; % Zaehler in x-Richtung
    pixelCntY_N = pixelCntY * featureCntY; % Zaehler in y-Richtung

    % Erstellen der Pixel-Matrix und Rausch-Matrix
    pixelFeatureMatrix = uint8(ones(pixelCntY_N, pixelCntX_N) .* 255);
    noiseMatrix = uint8(randi([0 99], pixelCntY_N, pixelCntX_N));

    % Umwandlung der Merkmale-Matrix in S/W-Bild
    featureMatrix = uint8(featureMatrix);
    featureMatrix = 255 - (featureMatrix .* 255);

    % Befuellung der Pixel-Matrix
    % Skalieren der Merkmale-Matrix auf Pixel-Matrix
    for iPixY = 0:(pixelCntY_N - 1)
        iFeatY = idivide(uint16(iPixY), pixelCntY, 'floor');
        for iPixX = 0:(pixelCntX_N - 1)
            iFeatX = idivide(uint16(iPixX), pixelCntX, 'floor');
            pixelFeatureMatrix(iPixY + 1, iPixX + 1) = featureMatrix(iFeatY + 1, iFeatX + 1);
        end
    end
    % Rauschen hinzufuegen
    for iPixY = 1:pixelCntY_N
        for iPixX = 1:pixelCntX_N
            if (noiseMatrix(iPixY, iPixX) < noise)
                pixelFeatureMatrix(iPixY, iPixX) = 255 - pixelFeatureMatrix(iPixY, iPixX);
            end
        end
    end

    % optional Bild speichern
    if (~isempty(name))
        imwrite(pixelFeatureMatrix, [name '.bmp']);
    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 Merkmaltyp 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 Merkmaltyp! 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 Erhoehung 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(1:end, i) = (vWeightMatrix(1:end, i) + ((hGauss./max(hGauss)) * 2 - 1)') - 1;
98              % mittlere Erhoehung entfernen
99              %tempWeightMatrix(1:end, i) = tempWeightMatrix(1:end, i) - addMulWeightMatrix(1:end, i);
100             % skalieren
101             %tempWeightMatrix(1:end, i) = tempWeightMatrix(1: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
```

Literatur

[1] Anna Corves. www.dasgehirn.info, 2012. https://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/, Letzter Zugriff 21. Oktober 2017.