Analyse verschiedener Varianten neuronaler Netzwerke für den Einsatz auf FPGAs

Nikolaus Haminger-Huber

BACHELORARBEIT

Nr. 1410306015-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Hardware-Software-Design

in Hagenberg

im Juli 2017

Diese Arbeit entstand im Rahmen des Gegenstands

Seminar Bachelorarbeit

im

Wintersemester 2016/Sommersemester 2017

Betreuer:

Prof. (FH) Dipl.-Ing. Dr. Markus Pfaff

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 1. Juli 2017

Nikolaus Haminger-Huber

Inhaltsverzeichnis

Erklärung **iii**

Kurzfassung **vi**

Abstract **vii**

1 Einleitung **1**
1.1 Motivation 1
1.2 FPGAs als Plattformen für Anwendungen der KI 2
1.3 Gliederung 3

2 Grundlagen **4**
2.1 Was ist künstliche Intelligenz? 4
2.2 Methoden der künstlichen Intelligenz 5

3 Neuronale Netze **7**
3.1 Das Gehirn als biologisches Vorbild 7
3.2 Mathematisches Modell 9
3.3 Lernen 11
3.4 Hopfield-Netze 12
3.5 Multi-Layer-Perceptron 13
3.6 Backpropagation 15
3.7 Ausblick 17

4 Realisierung der Varianten auf einem FPGA **18**
4.1 Grundsätzliche Überlegungen 18
4.2 Reelle Zahlen und arithmetische Operationen 19
4.3 Das Neuron 20
4.4 Die Synapse 21
4.5 Hopfield-Netze 22
4.6 Multi-Layer-Perceptron 24
4.7 Backpropagation 24

5 Testimplementierungen **28**

5.1 Von der Simulation zum Hardware-Modell 28
5.2 Die Hardware-Implementierung 29
5.3 Verwendete Arbeitsumgebung 32

6 Testergebnisse und Vergleiche 34
6.1 Das XOR-Problem 34
6.2 Simulationsergebnisse 34
6.2.1 Software-Simulationen mit C++ 35
6.2.2 Hardware-Simulationen 37
6.3 Syntheseergebnisse 39

7 Ausblick 41

Kurzfassung

Das Verhalten von nicht-lernfähigen Programmen im Umgang mit ihren Eingangsdaten muss von ihren Entwicklerinnen und Entwicklern bis ins kleinste Detail festgelegt werden. Im Laufe der Zeit werden Systeme jedoch mit einer wachsenden Flut an Daten konfrontiert. Da das Verhalten von nicht-lernfähigen Algorithmen für jeden einzelnen Datensatz genau definiert werden muss, steigt der Entwicklungsaufwand dadurch enorm an.

Daher wurden Verfahren zur Implementierung von lernfähigen Systemen entwickelt, welche 'Künstliche Intelligenz' (KI) genannt werden. Systeme dieser Art werden mit vorhandenen Daten trainiert und können dann neue Daten klassifizieren, Prognosen abgeben oder Entscheidungen treffen.

Es gibt viele verschiedene Methoden der KI. Eine der bekanntesten und heute vielfach in der Industrie eingesetzten Methoden ist jene der neuronalen Netze. Die Struktur der neuronalen Netze entspricht dem bisher bekannten Aufbau des Gehirns.

Zur Implementierung neuronaler Netze gibt es verschiedene Ansätze. In dieser Arbeit werden einige dieser Ansätze analysiert und ihre Anwendbarkeit auf FPGAs untersucht.

Abschließend werden verschiedene Prototypen neuronaler Netze als FPGA-Implementierungen vorgestellt und anhand der Simulations- und Syntheseergebnisse verglichen. Die Arbeit liefert als Ergebnis einen praktischen Ansatz zur Implementierung neuronaler Netze auf FPGAs.

Abstract

The workflow of a traditional program not having the ability to learn from its processed data has to be defined down to the smallest detail. Once unknown data has to be processed, the programmer has to change the implementation of such a traditional program manually. An increasing data volume will result in a higher effort on handling such data.

As a result of the previously mentioned issues, algorithms with the ability to learn from processed data have been developed. These algorithms are called 'Artificial Intelligence' (AI) and they are able to be trained with currently existing data. After a successful training session they are able to classify unknown data or make predictions and decisions very accurately.

A huge variety of AI have been developed by now, where the so-called *Artificial Neural Networks* (ANN), frequently applied in industry, show an outstanding success. The structure of an ANN is roughly equal to the known physiology for learning in our brains.

There are many different ways to implement ANNs. Some of these will be analyzed for their usage on FPGAs in this thesis. Furthermore, some prototypes for such FPGA implementations will be presented which will then be compared by the results of simulation and synthesis. The goal of the thesis is to introduce a practical and easily understandable approach for implementing ANNs on an FPGA.

Kapitel 1

Einleitung

1.1 Motivation

Der Begriff 'Künstliche Intelligenz' (KI) oder im Englischen 'Artificial Intelligence' (AI) hat längst Einzug in die Gesellschaft gehalten. Oftmals ist der Begriff von diversen Science-Fiction-Filmen negativ geprägt. Jedoch verwenden große Firmen wie Google, Microsoft oder Tesla bereits Methoden der KI (Beispiele: Google Assistant, Microsoft Cortana, Teslas Autopilot). Alltägliche Anwendungen wie die Spracherkennung, Suchfunktion oder Bilderkennung in verschiedenen Systemen haben Methoden der KI im Hintergrund. Das Ziel vieler Forscher auf dem Gebiet der KI ist, Systeme zu erzeugen, die über eine menschenähnliche oder darüber hinausgehende Intelligenz verfügen [RN05].

Es gibt viele verschiedene Methoden der KI. Die am häufigsten eingesetzte Methode ist jene der neuronalen Netze, welche heute vor allem als *Deep Neural Networks* (DNN) implementiert werden. Der Ansatz entspringt aus der Idee, das Neuronen-Synapsen-System eines Gehirns nachzubilden. Neuronale Netze weisen demnach eine ähnliche Struktur auf, wie die derzeit bekannte Physiologie des Lernsystems im Gehirn (siehe Kapitel 3).

In den späten 1980er- und frühen 1990er-Jahren wurde versucht, spezielle Hardware (ASICs) für neuronale Netzwerke zu entwickeln - sogenannte Neurocomputer. Allerdings hatten diese wenig Erfolg, da die zur der Zeit verwendeten ASIC-Technologien nicht genügend leistungsfähig für eine ernsthafte Anwendung von neuronalen Netzen waren [OR06].

Heute werden für große neuronale Netze vielfach GPUs verwendet, da diese aus sehr vielen parallelen Recheneinheiten bestehen (siehe Abbildung 1.1). Sie können damit viele Matrixoperationen gleichzeitig ausführen und eignen sich geradezu optimal für die Beschleunigung von neuronalen Netzen [OJ04]. GPU-Hersteller wie nVidia entwickeln inzwischen eigene GPUs für die Be-

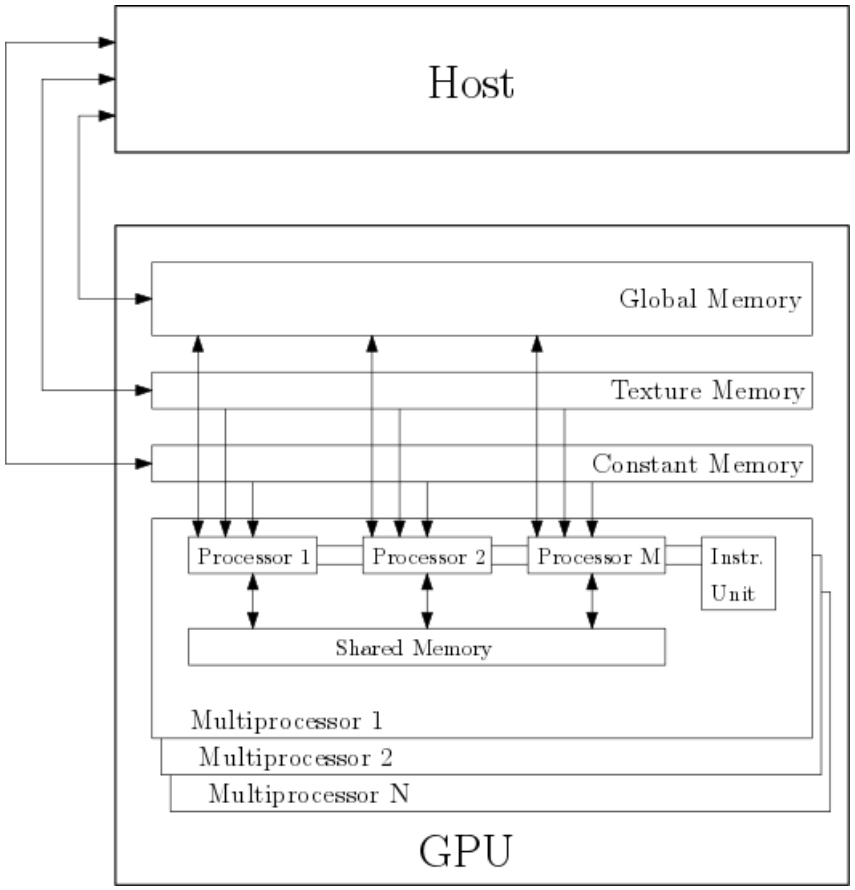


Abbildung 1.1: *Schematischer Aufbau einer GPU.*

schleunigung von neuronalen Netzen (Beispiel: nVidia Tesla P100 [nVi16]).

1.2 FPGAs als Plattformen für Anwendungen der künstlichen Intelligenz

Nun stellt sich die Frage, warum FPGAs als Plattformen für neuronale Netze in Betracht gezogen werden können. Diese Überlegung entstammt aus der Betrachtung einiger Eigenschaften eines FPGAs, die es für die Implementierung von neuronalen Netzen geeignet erscheinen lassen:

- **Konfigurierbarkeit**
  Durch die Konfigurierbarkeit eines FPGAs können anwendungsspezifische neuronale Netze erstellt und nahezu ohne Verzögerung getestet werden, ohne ASICs dafür entwickeln zu müssen.

- **Parallelität**
  Die Struktur eines FPGAs ist prädestiniert für parallele Anwendungen. Die Implementierung von neuronalen Netzen kann auf mehreren Ebenen parallelisiert werden und die Verarbeitungsgeschwindigkeit wird dadurch massiv beschleunigt.
- **Leistungsaufnahme**
  Im Vergleich zu GPUs haben FPGAs einen geringeren Leistungsumsatz. Dieser Vorteil wird gerade bei größeren neuronalen Netzen tragend, wenn mehrere Hardwareeinheiten zusammengeschaltet werden.

In Kapitel 4 werden die angeführten Eigenschaften für die Erstellung eines Hardware-Modells für FPGAs genutzt.

1.3 Gliederung

In Kapitel 2 wird auf die Grundlagen der Methoden der künstlichen Intelligenz eingegangen. Die Unterschiede des maschinellen Lernens im Vergleich zur klassischen Programmierung werden aufgegriffen sowie die verschiedenen Methoden der künstlichen Intelligenz in ihren Grundzügen erläutert.

Anschließend werden in Kapitel 3 die Grundlagen der neuronalen Netze analysiert und erklärt. Dies ist nötig, um die im Kapitel 4 entwickelten Hardware-Modelle von neuronalen Netzen zu verstehen.

Die Kapitel 5 und 6 betreffen die selbst implementierten Prototypen von neuronalen Netzen und beschreiben den Aufbau sowie deren Testdurchläufe.

Zum Schluss erfolgt in Kapitel 7 noch eine Zusammenfassung der gesamten Arbeit.

Kapitel 2

Grundlagen

2.1 Was ist künstliche Intelligenz?

In Kapitel 1 wurde bereits das Potential von Anwendungen der künstlichen Intelligenz erläutert. Doch was ist eigentlich die Definition der KI? Schlägt man in diversen Fachbüchern nach, so lassen sich die darin vorhandenen Definitionen in vier Kategorien einteilen [RN05, Kap. 1]:

- **Systeme, die menschlich denken**
  Dabei handelt es sich um den kognitiven Ansatz. Das menschliche Denksystem muss dabei zuerst durchschaut werden, bevor es nachgebildet werden kann. Dies kann entweder durch die Beobachtung der physiologischen Vorgänge im Gehirn oder durch eine psychologische Herangehensweise erfolgen. Aus der Beobachtung der physiologischen Vorgänge entstammt etwa das Prinzip der neuronalen Netze.

- **Systeme, die menschlich handeln**
  Dieser Ansatz ist die Grundlage des Turing-Tests. Dieser ist nach seinem Erfinder *Alan Turing* (1950) benannt. Bei diesem Test kommuniziert ein Mensch mit einem System, welches versucht, das Verhalten eines Menschen vorzutäuschen. Es wird als intelligent bezeichnet, wenn der menschliche Befragende erfolgreich getäuscht wurde.

- **Systeme, die rational denken**
  Systeme dieser Art agieren nach den Gesetzen der Logik. Die Logik entspringt dem Versuch, 'richtiges Denken' in Gesetzen auszudrücken. Damit ein solches System ein sinnvolles Resultat erarbeiten kann, müssen alle Eingaben und ihre Beziehungen zueinander in eine formale logische Notation gebracht werden. Dies ist eine große Hürde bei der Implementierung solcher Systeme.

- **Systeme, die rational handeln**
  Diese Variante scheint nah mit Systemen verwandt zu sein, die rational denken. Tatsächlich benötigen rational handelnde Systeme eine rationale Entscheidungsfindung. Allerdings geraten Systeme in Situationen, wo keine unmittelbare rationale Lösung vorhanden ist und trotzdem reagiert werden muss. Die darauf folgende Handlung wird auch als rational angesehen, obwohl sie nicht der Lösung eines logischen Systems entstammt. Dazu sei das Beispiel eines Menschen angeführt, der auf eine heiße Herdplatte greift. Das unmittelbare Zurückziehen der Hand ist eine logische Konsequenz, obwohl diese Aktion nicht aus einer logischen Entscheidungsfindung stammt.

2.2 Methoden der künstlichen Intelligenz

Die KI-Forschung nahm ihren Anfang in den 1930er-Jahren mit der Prädikatenlogik des Österreichers Kurt Gödel sowie dem Halteproblem von Alan Turing [Ert16]. Seitdem wurden unzählige Methoden entwickelt. Im folgenden werden die einzelnen Teilbereiche kurz erklärt:

- **Neuronale Netze**
  Neuronale Netze sind eine Nachbildung des Neuronen-Synapsen-Systems des Gehirns. Ihre Stärken sind der Umgang mit schwer klassifizierbaren Daten sowie die Fähigkeit des Lernens. Sie werden eingehend im Kapitel 3 untersucht.

- **Logik**
  Dieses Teilgebiet der künstlichen Intelligenz folgt den Gesetzen der *Prädikatenlogik*. Eigene Programmiersprachen wie *LISP* und *Prolog* wurden zur Logikprogrammierung entwickelt. Bis in die 1980er-Jahre wurden große Erfolge mit diesen symbolverarbeitenden Programmen erzielt. Allerdings funktionierten die Logikprogramme nur bei kleinen Problemen gut, da in der Logik zwischen *wahr* und *falsch* keine anderen Werte möglich sind. Die Programme konnten zudem nur mit Daten umgehen, welche vorher in die richtigen logischen Ausdrücke übersetzt wurden. Als der Erfolg dieser Programme hinter den Erwartungen zurückblieb, wurden die Anstrengungen in diesem Teil der künstlichen Intelligenz vermindert.

- **Schließen mit Unsicherheit**
  In dieser Zeit wurde das *probabilistische Schließen* als vielversprechender Ausweg aus dem Dilemma der Logikprogramme entdeckt. Man versuchte, die Logik mit den schon länger bekannten *neuronalen Netzen* zu verknüpfen. So wurden die *Bayes-Netze* entwickelt, die bis heute als Diagnose-Systeme eingesetzt werden. Ein weiteres Beispiel ist die *Fuzzy-Logik*, welche unendlich viele Werte zwischen null und eins

einführte. Sie wird heute gerne in der Regelungstechnik eingesetzt.

- **Lernen**
  Ein großer Vorteil verschiedener Methoden der KI gegenüber klassischen Programmen ist die Fähigkeit, aus den prozessierten Daten zu lernen. Auch das Lernen wird in Kapitel 3 behandelt.

- **Suche/Problemlösen**
  Dieser Teil der KI-Forschung beschäftigt sich mit der effizienten Suche von Lösungen in sogenannten Suchbäumen. Da heutige Systeme mit einer wachsenden Flut an Daten konfrontiert werden, werden Algorithmen zur schnellen Suche der relevanten Daten und Lösungen entwickelt.

Das Buch [Ert16] beschäftigt sich eingehender mit all diesen Themen. In dieser Arbeit werden nur die *Neuronalen Netze* und damit verbunden das *Lernen* näher behandelt.

Kapitel 3

Neuronale Netze

Dieses Kapitel ist eine Einführung in die umfangreiche Thematik der neuronalen Netze und dient als Grundlage für das Kapitel 4. Der Inhalt ist angelehnt an das gleichnamige Kapitel des Werks *Grundkurs Künstliche Intelligenz* von W. Ertel [Ert16, Kap. 9]. Aus anderen Werken entnommene Passagen sind entsprechend gekennzeichnet.

3.1 Das Gehirn als biologisches Vorbild

Wie schon mehrfach erwähnt, ist das Modell der neuronalen Netze ein Nachbau des Neuronen-Synapsen-Systems des menschlichen Gehirns. Dieses besitzt etwa 10 bis 100 Milliarden Nervenzellen (Neuronen), wie sie in Abbildung 3.1 vereinfacht dargestellt werden. Jedes *Neuron* besitzt ein Axon und mehrere Dendriten. Axone leiten elektrische Impulse vom Zellkörper weg, während Dendriten die Impulse zum Zellkörper hinleiten. Axone und Dendriten sind über Synapsen verbunden. Ein Neuron kann eine kleine elektrische Ladung speichern; es ist vergleichbar mit einem kleinen Kondensator. Die Nervenzelle wird durch die Impulse von anderen Neuronen über die eingehenden synaptischen Verbindungen aufgeladen. Umso mehr verbundene Neuronen Impulse übermitteln, desto höher ist die Spannung im Neuron. Überschreitet die interne Spannung einen Grenzwert, so 'feuert' das Neuron. Das heißt, es setzt über das Axon einen Impuls ab und entlädt so seinen Speicher.

Jedes Neuron im menschlichen Gehirn ist mit etwa 1000 bis 10.000 anderen Neuronen verbunden. Interessant ist an dieser Stelle eine nähere Betrachtung der Verbindungsstellen. Diese *Synapsen* (siehe Abbildung 3.2) besitzen einen kleinen Spalt. Der elektrische Impuls wird hier durch chemische Substanzen, die sogenannten *Neurotransmitter*, weitergegeben. Je nach Häufigkeit einer Aktivität an einer Synapse wird die Verbindung nach und nach leitfähiger

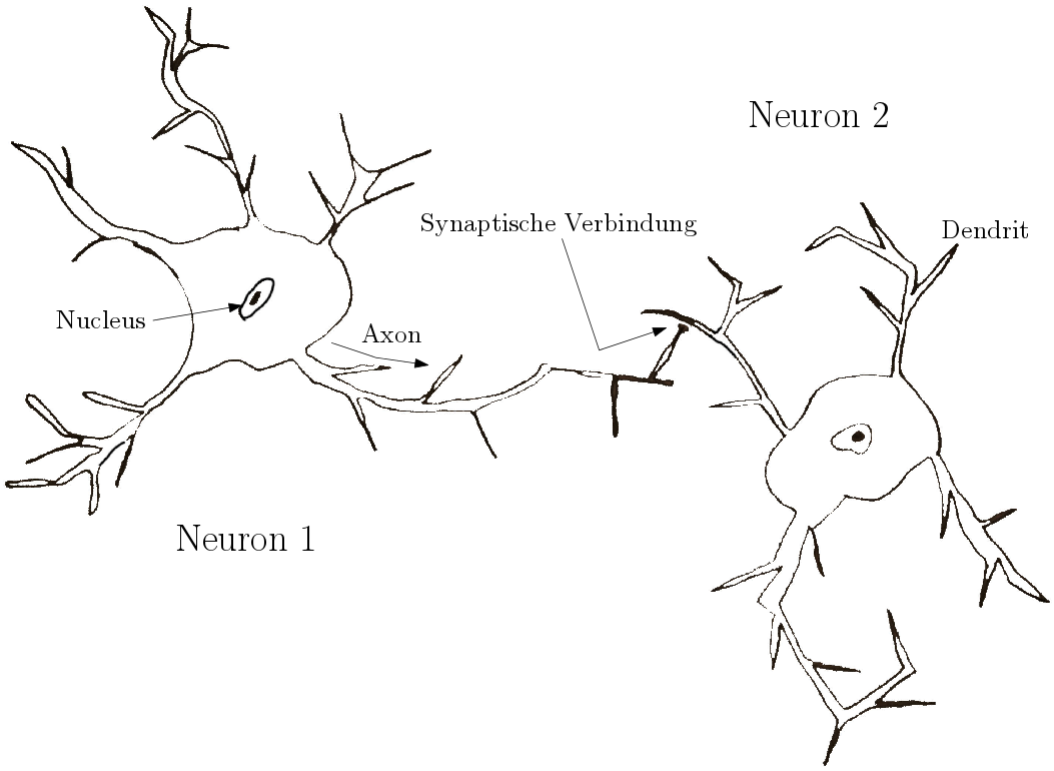


Abbildung 3.1: *Schematische Darstellung der Neuronen unseres Gehirns.*

oder -unfähiger[1]. Der Impuls wird also gewichtet durch diese Leitfähigkeit an das anliegende Neuron weitergegeben. Diese Tatsache spielt eine wichtige Rolle für die *Lernfähigkeit* des Gehirns. Da das Gewicht adaptiv ist, sind die Synapsen speichernde Elemente.

Interessant ist die Frage nach der Verarbeitungsgeschwindigkeit der Neuronen. Diese liegt bei etwa einer Millisekunde im Neuron und einer weiteren Millisekunde bei der Überwindung des synaptischen Spaltes. Das Gehirn arbeitet also mit einem 'Takt' unter einem Kilohertz. Diese geringe Geschwindigkeit wird jedoch durch die asynchrone und parallele Arbeitsweise der Neuronen ausgeglichen. Bedenkt man nun die unglaublich hohe Anzahl an Verbindungen (etwa 10^{13} bis 10^{15}), so kann die Fähigkeit der komplexen Mustererkennung und Lernfähigkeit unseres Gehirns durchaus nachvollzogen werden.

[1] Die Leitfähigkeit ist allerdings auch durch verschiedene Substanzen wie Alkohol oder Drogen beinflussbar.

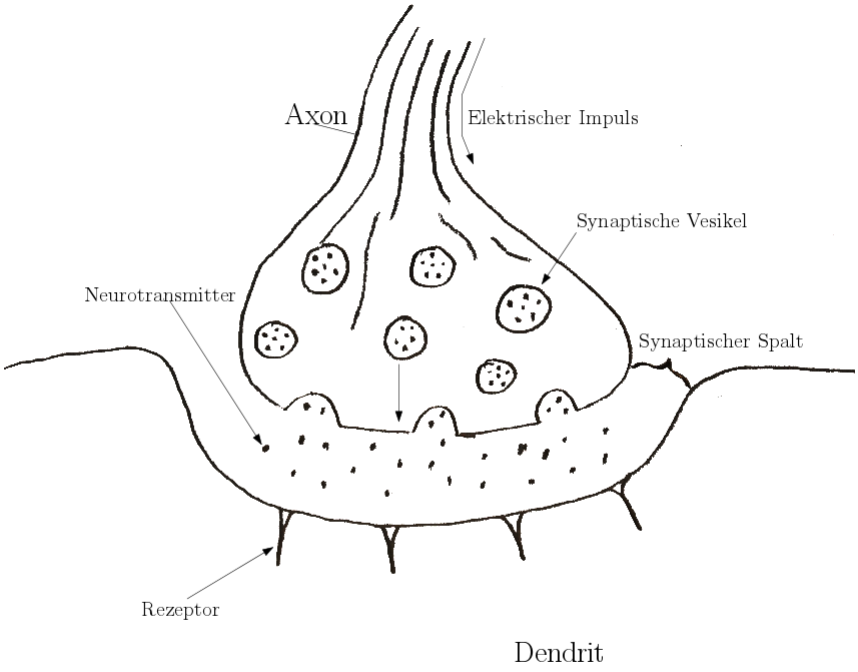


Abbildung 3.2: *Schematische Darstellung einer Synapse.*

3.2 Mathematisches Modell

1943 wurde von McCulloch und Pitts erstmals ein mathematisches Modell des Neurons aufgestellt [MP43]: Ein Neuron i wird in einem Zeitschritt 'aufgeladen' durch die Summe der eingehenden Verbindungen $x_1, ..., x_n$, gewichtet mit der 'Leitfähigkeit' der jeweiligen Synapsen w_{ij}. Formal kann dieser Prozess mit

$$\sum_{j=1}^{n} w_{ij} x_j$$

beschrieben werden. Das 'Feuern' eines Neurons wird mithilfe einer sogenannten *Aktivierungsfunktion* $f(x)$ modelliert. Der Ausgangswert eines Neurons i berechnet sich somit zu

$$y_i = f(\sum_{j=1}^{n} w_{ij} x_j) \tag{3.1}$$

Die mathematische Modellierung eines Neurons wird in Abbildung 3.3 grafisch dargestellt.

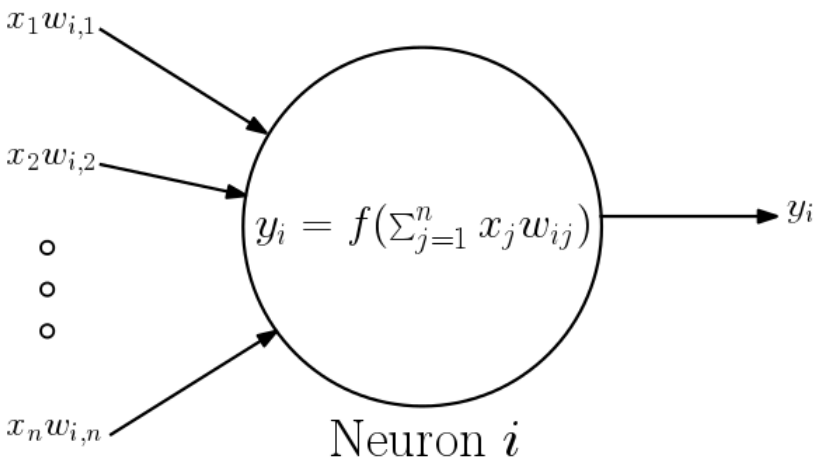


Abbildung 3.3: *Mathematisches Modell eines Neurons.*

Für die Realisierung der **Aktivierungsfunktion** gibt es mehrere Varianten, abgebildet in 3.4:

- **Identität**
  Dies ist die einfachste Form einer Aktivierungsfunktion mit $f(x) = x$. Es werden also prinzipiell nur die gewichteten Eingänge aufsummiert und unverändert weitergegeben. Das Problem eines neuronalen Netzes mit einer linearen Aktivierungsfunktion ist, dass die Ausgänge mit der Zeit über alle Grenzen wachsen aufgrund der Unbeschränktheit einer solchen Funktion.

- **Einheitssprung**
  Der Einheitssprung (auch *Heavisidesche Stufenfunktion*) ist definiert als

$$H_\theta(x) = \begin{cases} 1 & \text{wenn } x > \theta \\ 0 & \text{sonst} \end{cases} \tag{3.2}$$

  Diese Funktion kann für binäre Neuronen verwendet werden. Ein binäres Neuron wird auch Perzeptron genannt. Dieses kann jedoch nur linear separable Daten trennen (siehe Abschnitt 6.1 auf Seite 34). Daher werden für nicht-lineare Probleme stetige Neuronen mit Ausgangswerten zwischen 0 und 1 verwendet.

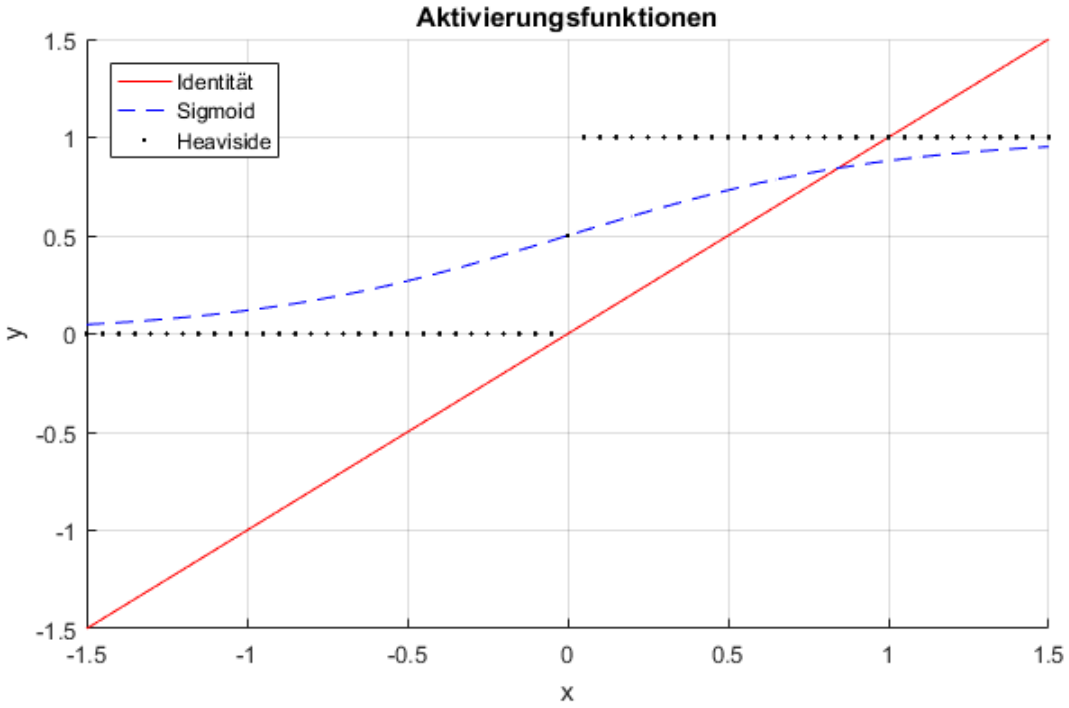


Abbildung 3.4: *Vergleich verschiedener Aktivierungsfunktionen.*

- **Sigmoid**
  In stetigen Neuronen werden oftmals sigmoide Aktivierungsfunktionen angewendet, wie zum Beispiel

$$f(x) = \frac{1}{1 + e^{-2x}}$$

  Diese sind asymptotisch beschränkt und um den Wendepunkt annähernd linear.

3.3 Lernen

Eine zentrale Fähigkeit des Gehirns ist das Lernen. Weiter oben wurde festgestellt, dass die Lernfähigkeit in der variablen synaptischen Leitfähigkeit verankert liegt. Je mehr Impulse über einen synaptischen Spalt übertragen werden, desto stärker wirken sich die Impulse auf ein nachfolgendes Neuron aus.

In neuronalen Netzen wird dieser Umstand durch die Veränderung der Gewichte w_{ij} nachgebildet. D. Hebb modellierte 1959 die heute bekannte *Hebb-Regel* [Heb49]. Diese besagt, dass bei gleichzeitiger Aktivität zweier Neuro-

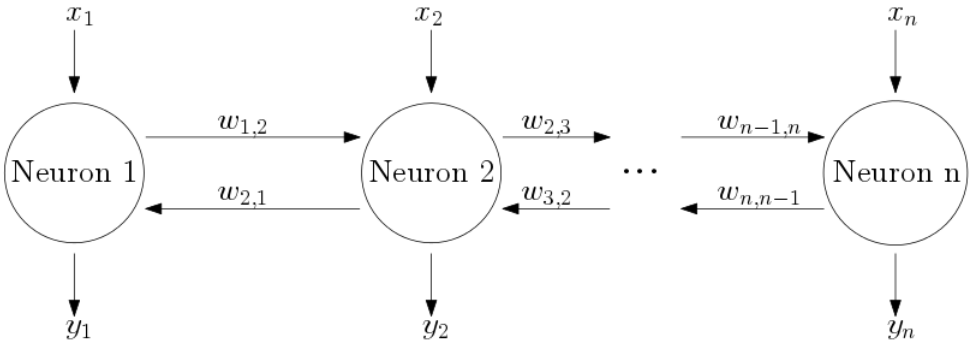


Abbildung 3.5: *Vollvernetzte Hopfield-Neuronenkette.*

nen i und j, welche beispielsweise wie in Abbildung 3.5 miteinander verknüpft sind, das Gewicht w_{ij} erhöht wird. Eine gängige Formel für die Gewichtsveränderung Δw_{ij} ist

$$\Delta w_{ij} = \eta y_i y_j \tag{3.3}$$

wobei mit η die Größe der einzelnen Lernschritte festgelegt wird. Die Konstante η wird auch *Lernrate* genannt.

3.4 Hopfield-Netze

Wird die Hebb-Regel auf die bisher betrachteten Neuronen mit Ausgangswerten zwischen 0 und 1 angewendet, so können die Gewichte nur stärker werden. Um dieses Problem zu beheben, stellte Hopfield 1982 ein Modell binärer Neuronen auf, welche die Werte -1 und 1 annehmen können [Hop82]. Wendet man nun die Gleichung 3.3 auf diese Neuronen an, so wachsen die Gewichte nur bei gleichzeitiger Aktivität der beiden Neuronen i und j. Ist eine der beiden Neuronen inaktiv, wird Δw_{ij} negativ und das Gewicht wird abgeschwächt.

Nach diesem Prinzip arbeiten die *Hopfield-Netze*. Diese kann man sich als eine Kette von Neuronen vorstellen, welche jeweils voll mit ihren Nachbarneuronen vernetzt sind (siehe Abbildung 3.5). Diese vollständige Vernetzung bringt komplexe Rückkopplungen mit sich, welche *Rekurrenzen* genannt werden.

Hopfield-Netze eignen sich zur Mustererkennung beispielsweise von handgeschriebenen Buchstaben. Zuerst werden dem Netz hierzu verschiedene Bilder von Buchstaben mit den gleichen Dimensionen durch Gewichtsanpassung beigebracht. Ein solches Muster entspricht einem Vektor $\boldsymbol{q}$, wobei die Elemente $q_i^j \in \{-1, 1\}$ jeweils ein Pixel eines Musterbildes darstellen. Das

Lernen erfolgt nun mit N Mustervektoren mit jeweils n Elementen gemäß der Formel

$$w_{ij} = \frac{1}{n} \sum_{k=1}^{N} q_i^k q_j^k \tag{3.4}$$

An ein derart angelerntes Netz legt man anschließend ein neues Bild in Form eines Vektors $\boldsymbol{x}$ an. Wichtig ist dabei, dass alle Eingangswerte gleichzeitig angelegt werden und somit alle Neuronen parallel und asynchron arbeiten. Die nach Gleichung 3.1 prozessierten Ausgangswerte $\boldsymbol{y}$ werden wiederholt an den Eingang des Netzes rückgeführt, bis sich keine Wertänderungen an beliebigen y_i mehr ergibt. Das Bild sollte nun gegen eines der im Netz gespeicherten Muster konvergiert sein.

Die gelernten Muster neuronaler Netze im Allgemeinen und des Hopfield-Netzes im Speziellen stellen globale Minima dar (genauer in [Ert16, Kap. 9.2]). Ein Problem liegt in der begrenzten Kapazität der Netze: Je mehr Muster gelernt werden, desto mehr verschwimmen die Grenzen der Minima und desto höher ist die Wahrscheinlichkeit, dass angelegte Vektoren gegen 'falsche' lokale Minima konvergieren.

3.5 Multi-Layer-Perceptron

Durch die imponierenden Erfolge der Hopfield-Netze in der Mustererkennung wurde in den 1980er-Jahren die Forschung im Bereich der neuronalen Netze vorangetrieben. Man wollte die Kapazität der Netze verbessern sowie die schwer verständlichen rekurrenten Verbindungen eliminieren. Daraus erwuchs unter vielen anderen das Konzept der *mehrschichtigen Perzeptren* (Multi-Layer-Perceptron), welches im Verbund mit der *Backpropagation* bis heute das gängigste Modell der neuronalen Netze darstellt.

Abbildung 3.6 zeigt das grundlegende Konzept der mehrschichtigen Netze. Es beinhaltet eine Eingabeschicht (Input Layer), eine oder mehrere versteckte Schicht(en) (Hidden Layer) sowie eine Ausgabeschicht (Output Layer). Das Netz hat folgende Eigenschaften:

- **Signalfluss in eine Richtung**
  Verarbeitet das Netz einen Input (auch *Feed Forward* oder *Vorwärtspropagation* genannt), so ist die Richtung des Signalflusses von den Eingangsneuronen zu den Ausgangsneuronen vorgegeben. Rekurrente Verbindungen sind in diesem Netzwerk keine vorhanden.
- **Veränderung der Kapazität**
  Durch die Änderung der Anzahl an verdeckten Schichten und die Größenvariation derselben kann die Kapazität eines solchen Netzes verändert werden.

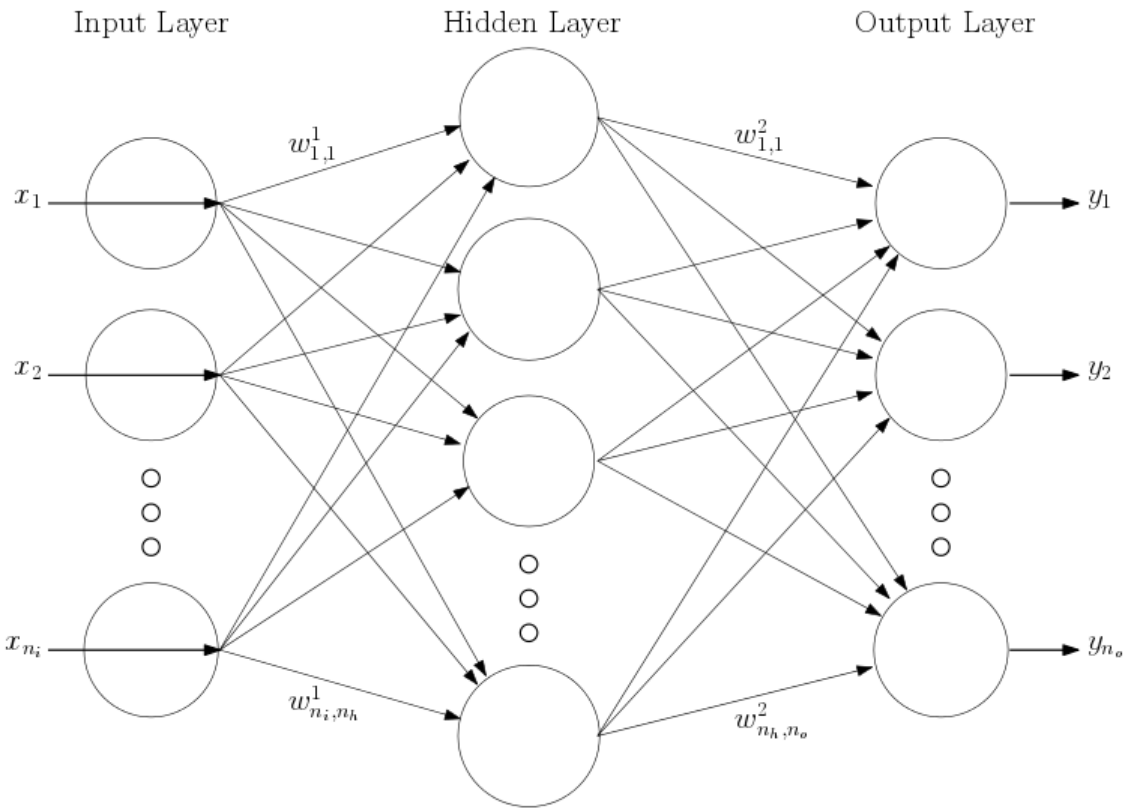


Abbildung 3.6: *Dreischichtiges neuronales Netz. Anzahl der Eingangsneuronen = n_i, Anzahl der Ausgangsneuronen = n_o, Anzahl der versteckten Neuronen = n_h.*

- **Verschiedene Größen der Eingangs- und Ausgangsvektoren**
  In einem Hopfield-Netz sind die Eingangs- auch gleichzeitig die Ausgangsneuronen, wodurch ein Eingangsvektor $\boldsymbol{x}$ auch nur auf einen gleich großen Ausgangsvektor $\boldsymbol{y}$ abgebildet werden kann. Diese Einschränkung existiert bei mehrschichtigen neuronalen Netzen nicht. So kann beispielsweise einem mehrschichtigen Netz beigebracht werden, aus einem Bild einer Zahl eine binäre Zahl zu produzieren.

Wie in Abbildung 3.6 dargestellt, verarbeiten die Eingangsneuronen den Eingangsvektor $\boldsymbol{x}$ direkt mit der Funktion $f(x) = x$. Alle anderen Neuronen berechnen ihren Ausgabewert y wie bereits dargestellt mit

$$y_i^p = f(\sum_{j=1}^{n_p} w_{ij}^p x_j^p)$$

wobei p die Nummer der gerade betrachteten Schicht darstellt (ohne Ein-

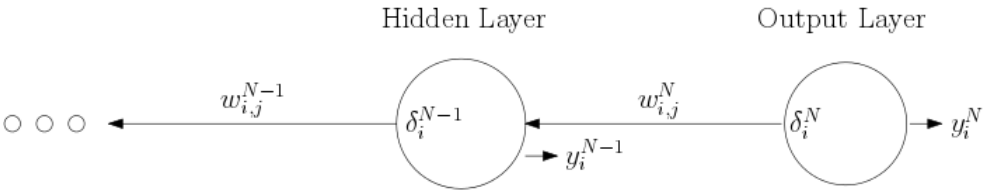


Abbildung 3.7: *Signalverlauf einer Backpropagation mit N Schichten.*

gangsschicht). Als Aktivierungsfunktion $f(x)$ wird zumeist eine Sigmoidfunktion verwendet.

3.6 Backpropagation

Dem Leser stellt sich nun die Frage, wie ein mehrschichtiges neuronales Netz lernen kann. Dies geschieht in einem Prozess, der *Backpropagation* genannt wird. In der Umgangssprache sagt man oft, dass ein Mensch aus seinen Fehlern lernt. Genau dieses Prinzip wird auch auf die mehrschichtigen neuronalen Netze angewendet.

Aufgrund der Ausgabe des Netzes wird für jedes Ausgabeneuron ein Fehlerwert (*Gradient* δ) berechnet. Dieser Fehler wird dann gewichtet an die Neuronen der nächst höheren Schicht weitergegeben. Der Vorgang wiederholt sich Schicht für Schicht bis zur ersten verdeckten Schicht nach der Eingangsschicht (Abbildung 3.7). Anschließend werden die Gewichte der Verbindung zwischen den Neuronen i und j gemäß

$$\Delta w_{j,i}^{p} = \eta \delta_j^p y_i^p \tag{3.5}$$

aktualisiert, wobei p die Schicht des Neurons i darstellt.

Der Gradient jedes verdeckten Neurons j berechnet sich zu

$$\delta_j^p = y_j^p (1 - y_j^p) \sum_{k=1}^{n_{p+1}} \delta_k w_{k,j}$$

Der Term $1 - y$ verhindert, dass der Wert der Gewichte w_{ij} nur anwächst, nicht aber kleiner werden kann. Für die Berechnung des Gradienten der Ausgabeneuronen gibt es verschiedene Möglichkeiten, von denen zwei nachfolgend angeführt werden.

- **Supervised Learning**
  Beim *Supervised Learning* (überwachtes Lernen) werden die vom neuronalen Netz berechneten Ausgangswerte $\boldsymbol{y}$ mit zuvor festgelegten Sollwerten $\boldsymbol{t}$ (engl. *target*) verglichen. Der Gradient des Ausgabeneurons i

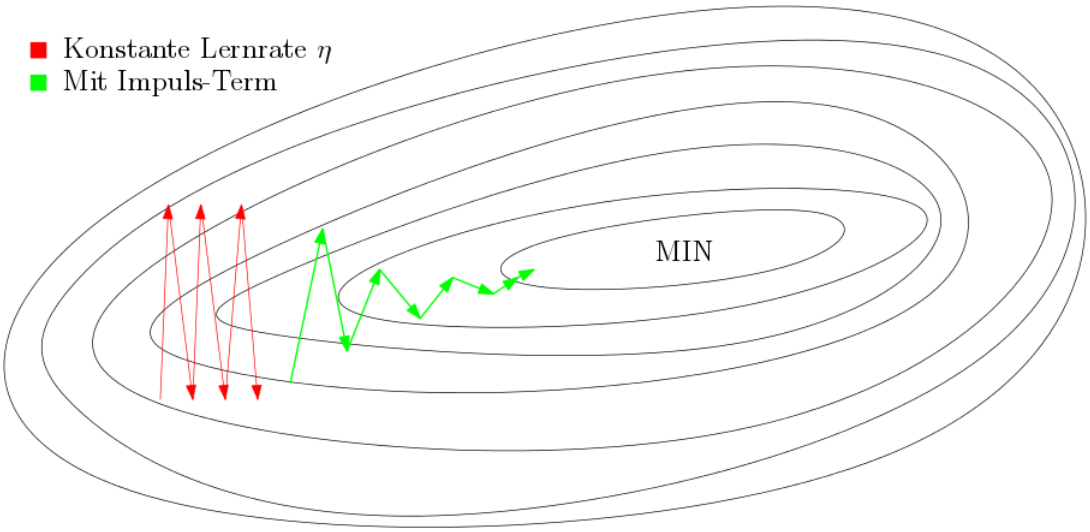


Abbildung 3.8: *Lernen ohne (links) und mit (rechts) Impuls-Term.*

berechnet sich zu

$$\delta_i = y_i(1 - y_i)(t_i - y_i)$$

Diese Variante des Lernens kann zum Beispiel bei einer Ziffernerkennung aus Bildern angewendet werden: Als Eingang des Netzes wird ein Bild angelegt, worauf das Netz die Zahl binär ausgeben muss. Als Target dient jeweils die entsprechende Binärzahl.

- **Reinforcement Learning**
  Bei dieser Variante wird das neuronale Netz pro Iteration für gute Aktionen 'belohnt' und für schlechte Aktionen 'bestraft'. Dies wird erreicht, indem bei einer absolut richtigen Aktion der Gradient des entsprechenden Ausgabeneurons auf einen Wert nahe 0 und bei einer absolut falschen Aktion auf einen Wert nahe 1 gesetzt wird. Wie auch beim Supervised Learning wird das Netz im Zuge jeder Iteration versuchen, den Gradienten zu minimieren.

Mit einer konstanten Lernrate η konvergieren neuronale Netze oft nur sehr langsam, wie Abbildung 3.8 anhand eines zweidimensionalen Problems veranschaulicht. Um ein Minimum schneller zu erreichen und damit die Lerngeschwindigkeit eines Netzes zu erhöhen, gibt es verschiedene Verbesserungen der ursprünglichen Lernregel. Die zwei nachfolgend angeführten Methoden werden in der Praxis gerne angewendet.

- **Veränderung der Lernrate η**
  Bei dieser Methode wird η bei jeder Iteration um einen konstanten Wert verringert. Das Ziel ist eine Verhinderung der Schwingung des Ausgabewertes um das Minimum.

- **Impuls-Term**
  Durch diese Maßnahme wird eine drastische Veränderung des Gradienten verhindert, wie in Abbildung 3.8 veranschaulicht wird. Die Lernregel gemäß der Gleichung 3.5 erweitert sich zu

$$\Delta w_{ij}^p(t) = \eta \delta_j^p x_i^p + \alpha \Delta w_{ij}^p(t-1) \tag{3.6}$$

  Die Konstante α bestimmt den Anteil der Gewichtsveränderung der letzten Iteration und kann je nach Anwendungsfall auf einen Wert zwischen 0 und 1 festgelegt werden.

3.7 Ausblick

Heute werden für komplexe Mustererkennungs- und Lernaufgaben wie die Echtzeit-Erkennung von Objekten in Videos zumeist neuronale Netze eingesetzt, welche unter dem Begriff *Deep Learning* zusammengefasst werden. Diese Netze sind eine Kombination aus mehreren verschiedenen neuronalen Netzen. Zur Bilderkennung werden beispielsweise sogenannte *Convolutional Neural Networks (CNN)* eingesetzt. Dabei werden in erster Instanz jeweils mehrere Eingangsneuronen auf ein Neuron in der nächsten Schicht abgebildet. Diese sogenannten *Feature-Maps* sind oftmals mehrschichtig und bilden hoch-dimensionale Eingangsvektoren auf niedriger-dimensionale Ebenen ab. Danach wird ein klassisches mehrschichtiges neuronales Netz mit Backpropagation in Reihe geschaltet, welches die Objekte aus den Feature-Maps erkennt und ausgibt.

Mit Deep Learning wurden in den vergangenen Jahren beachtliche Erfolge erzielt. Varianten davon werden in teil- und vollautonomen Fahrzeugen zur Objekterkennung eingesetzt. Ein anderes Deep-Learning-Netz schlug etwa den besten menschlichen Spieler in Go [Sil+16]. Andere neuronale Netze dieser Art können kreative Werke wie Musik und Bilder produzieren.

Da Deep-Learning-Netze meist eine Zusammenschaltung verschiedener Varianten neuronaler Netze sind, werden sie in dieser Arbeit nicht explizit behandelt. Diese verschiedenen Varianten basieren großteils auf den bisher behandelten Grundlagen.

Kapitel 4

Realisierung der Varianten auf einem FPGA

In diesem Kapitel werden anhand der theoretischen Grundlagen und mathematischen Modelle aus Kapitel 3 reale Modelle entwickelt, welche in Hardware-Beschreibungs-Sprachen implementierbar sind.

4.1 Grundsätzliche Überlegungen

Die neuronalen Verbindungen eines Gehirns (siehe die Abbildungen 3.1 und 3.2) übertragen die Impulse analog. Die Neuronen reagieren auf eine gewisse Intensität der über die Dendriten zugeführten Spannungswerte, welche durch die Leitfähigkeit der synaptischen Verbindungen definiert werden.

Auf einem FPGA können diese Verbindungen von Neuron zu Neuron nicht analog realisiert werden. Auch ein Gewicht $w_{ij} \in \mathbb{R}$ einer Verbindung zwischen den Neuronen i und j muss als reelle Zahl abgebildet werden. Ein wichtiges Detail bei der Implementierung eines neuronalen Netzes auf einem FPGA ist also der Umgang mit reellen Werten.

Die Gleichung 3.1 des mathematischen Modells eines Neurons sowie die *Hebb-Regel* 3.3 zur Aktualisierung der Gewichte enhalten einige Multiplikationen. Auch die Bestimmung des Gradienten zur Backpropagation innerhalb eines mehrschichtigen neuronalen Netzes beinhaltet mehrere Multiplikationen. Diese sind in der Hardware zeit- und flächenintensiv.

In Kapitel 6 wird anhand der Synthese des Prototypen sichtbar, dass die gerade erläuterten Umstände die am FPGA benötigte Fläche eines synthetisierbaren neuronalen Netzes entscheidend beinflussen.

4.2 Reelle Zahlen und arithmetische Operationen

Grundsätzlich gibt es verschiedene Möglichkeiten, reelle Zahlen in digitaler Hardware abzubilden. In Abbildung 4.1 werden die zwei meist eingesetzten Methoden dargestellt.

Eine Möglichkeit ist die Fixkomma-Darstellung (*Fixed-Point*), bei der jeweils eine festgelegte Anzahl an Bits für die Vorkomma- und die Nachkommastellen verwendet werden. Das Komma steht also immer an einer bestimmten Stelle. Um auch negative Zahlen darstellen zu können, wird die Vorkomma-Zahl als Zweierkomplement angenommen.

Eine andere Variante ist die Gleitkomma-Darstellung (*Floating-Point*) nach dem IEEE-754-Standard [IEE08]. Diese setzt sich aus einem Bit s für das Vorzeichen, der Mantisse m und dem Exponenten e zusammen zu

$$x = (-1)^s \cdot m \cdot 2^e$$

Der Exponent e wird dabei als Zweierkomplement dargestellt und die Mantisse m wird auf einen Wert zwischen 1 und $1,\bar{9}$ *normalisiert*.

Durch die Gleitkomma-Darstellung können sowohl der Wertebereich als auch die Genauigkeit des darstellbaren Zahlenraums gegenüber der Fixkomma-Darstellung erhöht werden. Um diese Behauptung zu belegen, werden zwei 16-Bit-Zahlen der beiden Darstellungen verglichen. Für die Fixkomma-Zahl werden jeweils 8 Bit für die Vorkomma- und die Nachkommastellen angenommen. Der Wertebereich liegt zwischen -129 und $+128$ und die Genauigkeit bei $2^{-8} = 0,00390625$. Die Gleitkomma-Zahl besitzt nach [IEE08] 5 Bits für den Exponenten und 10 Bits für die Mantisse. Der Wertebereich liegt zwischen -65.536 und $+65.536$ und die Genauigkeit bei $2^{-16} = 0,000015259$. Allerdings bleibt die Genauigkeit der Fixkomma-Zahl über den gesamten Wertebereich konstant, während sie bei der Gleitkomma-Zahl mit dem Exponenten skaliert.

Holt und Baker bewiesen, dass für die Gewichte eines neuronalen Netzes mindestens 16-Bit-Fixkomma-Zahlen verwendet werden müssen, damit es konvergieren kann [HB91]. Um die Fläche der in dieser Arbeit erstellten Prototypen zu optimieren, werden daher 16-Bit-Fixkomma-Zahlen verwendet, zumal mit diesen arithmetische Operationen auf dem eingesetzten FPGA hardwaremäßig durchgeführt werden können.

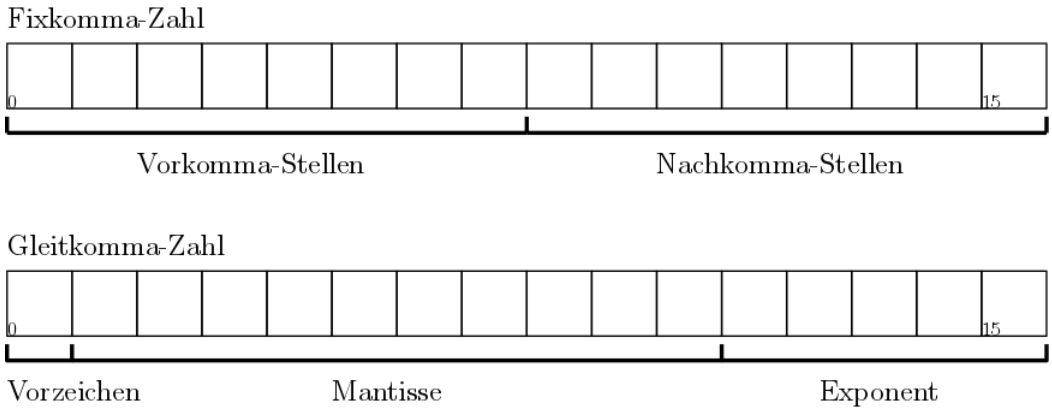


Abbildung 4.1: *Darstellung der Fixkomma- und Gleitkomma-Zahlen jeweils als 16-Bit-Werte.*

Für jede Multiplikation innerhalb des neuronalen Netzes wird eine DSP- oder Multiplikator-Einheit benötigt. Alternativ kann eine Add-and-Shift-Operation durchgeführt werden, welche jedoch sehr zeit- und flächen-aufwändig und damit unvorteilhaft ist. Selbst bei einem simplen mehrschichtigen Netz ohne Backpropagation ist für jede Gewichtung eines Signals und damit für jede einzelne Verbindung zweier Neuronen eine Multiplikation nötig. Daraus folgt, dass die Größe eines synthetisierbaren neuronalen Netzes stark von der Anzahl der verfügbaren DSP-Blöcke beziehungsweise der Multiplikatoren eines FPGAs abhängt.

4.3 Das Neuron

Die Modellierung eines Neurons als Hardware-Baustein kann vom mathematischen Modell im Abschnitt 3.2 abgeleitet werden und wird in Abbildung 4.2 veranschaulicht.

Die Eingänge werden mit den gewichteten Ausgängen der vorgeschalteten Neuronen verbunden. Der Ausgang ist das Ergebnis der durch eine Aktivierungsfunktion veränderte Summe der Eingänge. Zu beachten ist, dass sowohl jeder einzelne Eingang als auch der Ausgang in Form von Fix- oder Gleitkomma-Zahlen anliegen. So liegen beispielsweise bei 5 vorgeschalteten Neuronen und 16-Bit-Zahlen schon 80 Bits am Eingang des Neurons an.

An dieser Stelle ist die Realisierung der *Aktivierungsfunktion* in der Hardware interessant. Bei der Untersuchung des mathematischen Modells eines Neurons in Kapitel 3 wurde erläutert, dass eine Sigmoidfunktion für stetige Neuronen die besten Resultate liefert. Sollen neuronale Netze auch negative Ergebnisse berechnen, wird oftmals die *Tangens-Hyperbolicus-Funktion*

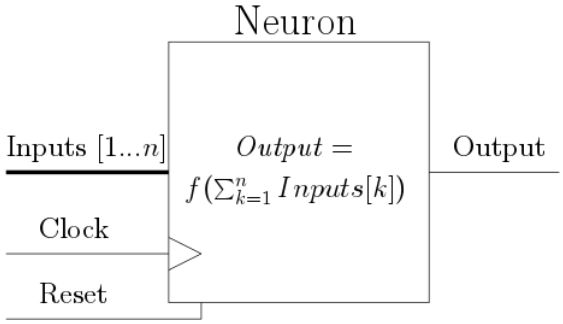


Abbildung 4.2: *Hardware-Modell eines Neurons mit n Eingängen und der Aktivierungsfunktion f.*

tanh(x) verwendet. Diese Funktion ist jedoch sehr zeit- und flächenintensiv, weshalb sie in dieser Realisierung mithilfe einer streckenweise linearen Funktion angenähert wird (siehe Abbildung 4.3):

$$f(x) = \begin{cases} 1 & \text{wenn } x > 1 \\ -1 & \text{wenn } x < -1 \\ x & \text{sonst} \end{cases}$$

So sind im Neuron keine aufwändigen mathematischen Berechnungen notwendig.

Das Neuron besitzt Clock- und Reset-Eingänge. So können in den Neuronen Register für die Forward-Propagation verwendet werden. Diese Maßnahme ist wichtig, um kombinatorische Pfade möglichst kurz zu halten und einen einheitlichen Takt bei verschiedenen Größen neuronaler Netze verwenden zu können. Die Registergrenze wird nach der Addition der Eingänge und vor der Aktivierungsfunktion gesetzt.

4.4 Die Synapse

Für jede Verbindung von einem Neuron zum nächsten muss ein Gewicht gespeichert werden. Diese Gewichte können direkt in den Neuronen gespeichert werden. Allerdings müssen sie im Zuge einer Backpropagation aktualisiert werden, wie in Abschnitt 4.7 angegeben. Dieser Umstand rechtfertigt die Auslagerung der Gewichtslogik in eine eigene Komponente. Diese wird in Abbildung 4.4 dargestellt und hat vorerst nur die Aufgabe, den Ausgang des vorgeschalteten Neurons zu gewichten und diesen Wert an das nächste Neuron weiterzugeben.

Um kombinatorische Pfade zu kürzen und eine höhere Taktrate zu erhalten, können auch in den Verbindungen Register verwendet werden. Da dies

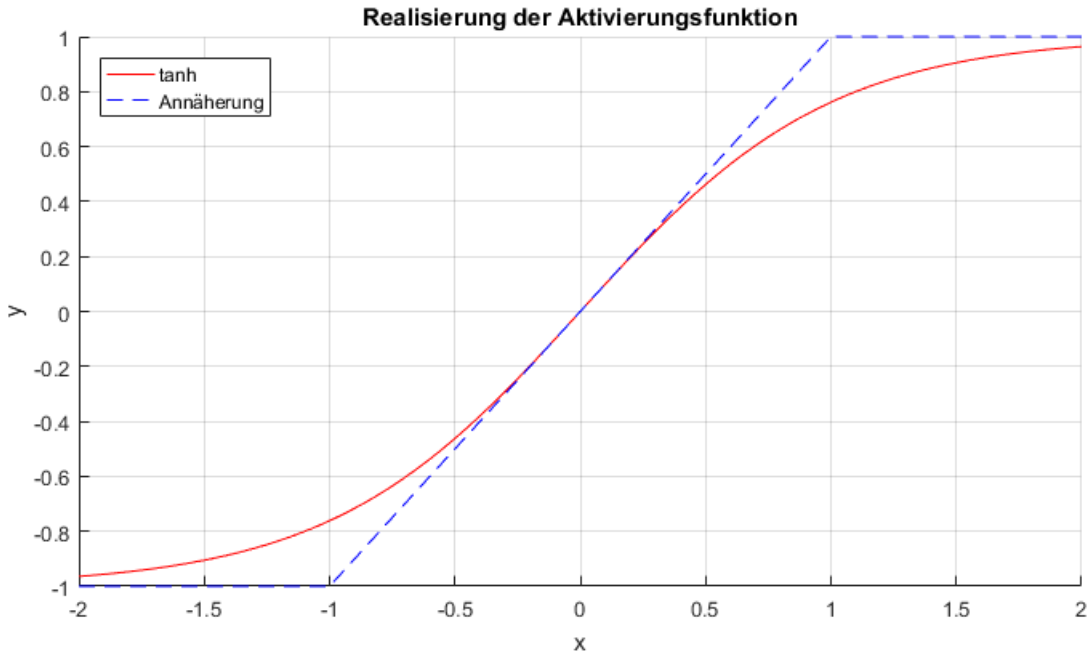


Abbildung 4.3: *Vergleich der tanh-Funktion mit der streckenweise linearen Funktion.*

jedoch in den folgenden einfachen neuronalen Netzen nur die Komplexität erhöht und keinen entscheidenden Vorteil mit sich bringt, bekommt die Verbindungskomponente erst im Zuge der Backpropagation Clock- und Reset-Leitungen.

4.5 Hopfield-Netze

Mit den in den vorherigen Abschnitten vorgestellten Modellen der Neuronen und Verbindungen kann nun ein Hopfield-Netz erstellt werden. Die Verschaltung der einzelnen Komponenten wird in Abbildung 4.5 dargestellt.

Die Gewichte der Verbindungen werden gemäß Gleichung 3.4 im Vorhinein berechnet und als Konstanten gespeichert. Da die Verbindung zwischen den Neuronen i und j bidirektional ist, gilt $w_{ij} = w_{ji}$. Ist dieses Netz einmal in Hardware synthetisiert, kann es aufgrund der konstanten Gewichte nur für eine Aufgabe verwendet werden. Als Vorteil dieses Netzes kann die geringe Verarbeitungszeit erachtet werden. Um aus einem Vektor $\boldsymbol{x}$ den Ausgabevektor $\boldsymbol{y}$ zu erzeugen, ist nur ein Taktzyklus notwendig. Allerdings muss die prozessierte Ausgabe je nach Beispiel in bis zu mehreren hundert Iterationen wieder an den Eingang angelegt werden, damit das Netz gegen eines der

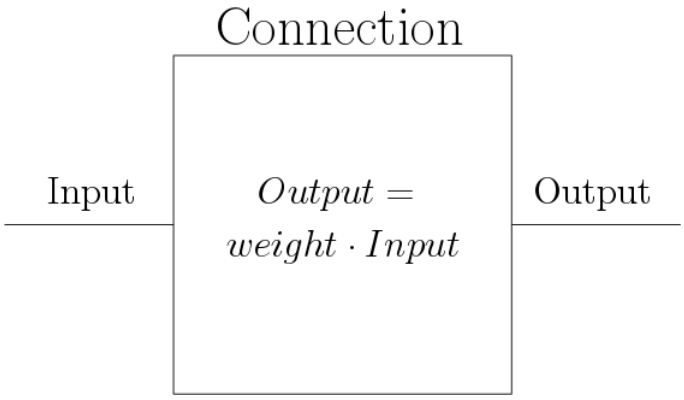


Abbildung 4.4: *Komponente für die Verbindungen zwischen den Neuronen.*

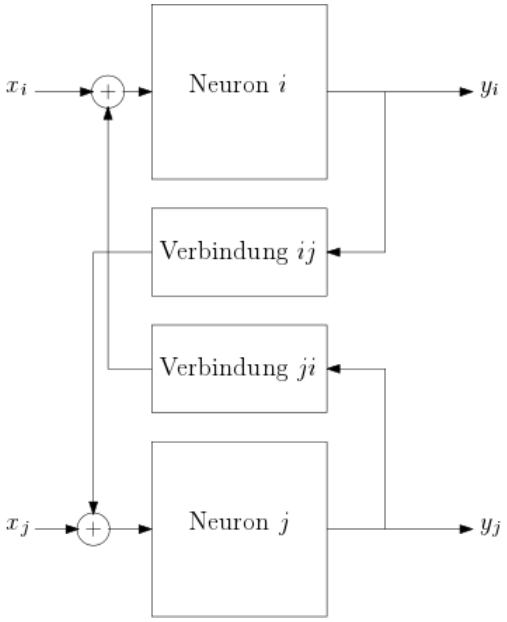


Abbildung 4.5: *Verschaltung der Neuronen i und j in Form eines Hopfield-Netzes.*

gespeicherten Muster konvergiert [Ert16, Kap. 9.2].

In Kapitel 3 wurden bereits einige Nachteile der Hopfield-Netze in dieser Form aufgezeigt. Problematisch sind die komplexen Rückkopplungen und die geringe Lernkapazität. Daher konzentriert sich diese Arbeit auf die mehrschichtigen Netze in den folgenden Abschnitten.

4.6 Multi-Layer-Perceptron

Mit den Hardware-Modellen des Neurons und der Synapse kann in weiterer Folge ein mehrschichtiges neuronales Netz erstellt werden. Für dieses Konzept wird zunächst das Modell des Multi-Layer-Perceptrons aus Kapitel 3 als Grundlage verwendet. Da die Eingangsneuronen des theoretischen Modells den Eingangswert unverändert weitergeben und auch sonst in keiner Weise die Funktion des neuronalen Netzes beinflussen, werden sie im Hardware-Modell nicht berücksichtigt. In weiterer Folge werden die verschiedenen Schichten des Netzes als eigene Komponenten entworfen (siehe Abbildung 4.6).

Die Eingangsschicht (*InputLayer*) besteht nunmehr allein aus Verbindungen, da keine Eingangsneuronen benötigt werden. Die verdeckten Schichten (*HiddenLayer*) enthalten jeweils eine Neuronenschicht und die darauf folgenden Verbindungen zur nächsten Schicht. Die Ausgangsschicht (*OutputLayer*) besteht schließlich aus einer einzigen Schicht von Neuronen.

In Abbildung 4.6 sind der kritische Pfad und die Anzahl der benötigten Takte einer Forward-Propagation ersichtlich. Der kritische Pfad läuft, ausgehend von den Registern eines Neurons, durch deren Aktivierungsfunktion und die Multiplikation der Verbindung sowie die Addition am Eingang eines Neurons zu den Registern des nächsten Neurons. Die Anzahl der benötigten Takte ist gleich der Anzahl der Schichten des Netzes.

Ein mehrschichtiges neuronales Netz in dieser Form ist nicht lernfähig, da die Gewichte der Verbindungen nicht zur Laufzeit aktualisiert werden können. Die Gewichte müssen also vor der Synthese berechnet und als Konstanten gespeichert werden. Alternativ können die Verbindungen um eine Speicherschnittstelle erweitert werden. So könnte ein Zweitsystem die Gewichte berechnen und bereitstellen. Um ein selbstständig lernfähiges neuronales Netz zu erzeugen und die Notwendigkeit des Einsatzes eines Zweitsystems zu unterbinden, ist der Einsatz der *Backpropagation* nötig.

4.7 Backpropagation

Der Einsatz der Backpropagation in einem Hardware-Modell eines mehrschichtigen neuronalen Netzes bedingt die Einführung von Rückwärtszweigen zur Rückführung der Gradienten. Um diese Rückwärtszweige realisieren zu können, werden zuerst die Modelle der Neuronen und Verbindungen aus den Abbildungen 4.2 und 4.4 überarbeitet. Die beiden erweiterten Komponenten werden in Abbildung 4.7 auf Seite 27 dargestellt.

Das Neuron besitzt nun Eingänge für die gewichteten Gradienten (*Dow*)

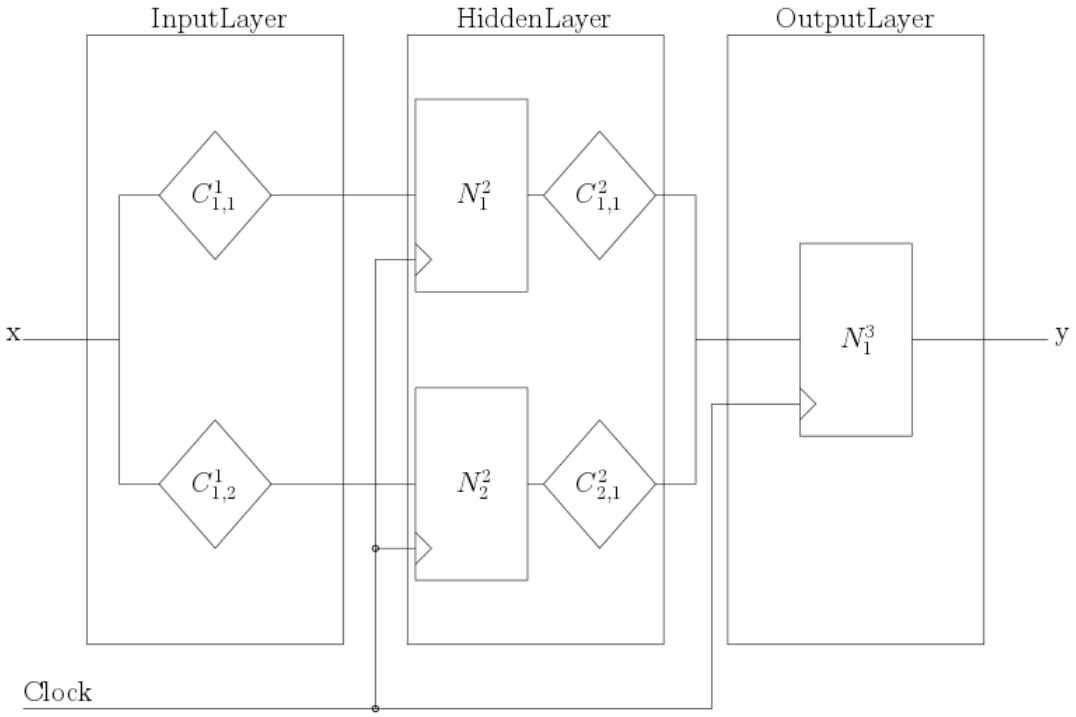


Abbildung 4.6: *Hardware-Modell eines mehrschichtigen neuronalen Netzes mit einem Eingang, zwei Neuronen in der verdeckten Schicht und einem Ausgang. Die Verbindungen sind mit $C^p_{i,j}$ und die Neuronen mit N^p_i gekennzeichnet.*

der Neuronen der nächsten Schicht. Aus diesen Eingängen wird der Gradient des Neurons berechnet, welcher am Ausgang weitergegeben wird. Die Berechnung des Gradienten erfolgt je nach Art des Neurons nach den Regeln aus Abschnitt 3.6. Allerdings werden die Fehlerfunktionen statt des Terms $1 - y_i$ mit der nach y abgeleiteten Aktivierungsfunktion $\frac{d}{dy}f(y) = f'(y)$ gewichtet[1]. Dabei wird unterschieden zwischen einem Ausgabeneuron und einem Neuron in einer verdeckten Schicht. Ein Ausgabeneuron bekommt nur einen Target- oder Fehlerwert als Eingangswert, während ein verdecktes Neuron alle gewichteten Gradienten der Neuronen der Folgeschicht als Eingangswerte bekommt. Der Gradient eines Ausgabeneurons i ergibt sich zu

$$(Dows[1] - Output) \cdot f'(Output) \tag{4.1}$$

[1]Diese Maßnahme berücksichtigt die Tatsache, dass bei diesem Entwurf eines neuronalen Netzes auch der negative Wertebereich mit einbezogen wird.

und der Gradient eines verdeckten Neurons zu

$$\sum_{k=1}^{N}(Dows[k]) \cdot f'(Output) \tag{4.2}$$

Das Ergebnis der Berechnung des Gradienten wird in einem Register gespeichert.

Die aktualisierte Variante der Verbindung bekommt zusätzlich den Gradienten des nachgeschalteten Neurons und gibt diesen gewichtet an das vorgeschaltete Neuron weiter. Weiterhin bekommt die Komponente Eingänge für die Lernrate η, den Parameter α aus dem Impuls-Term von Gleichung 3.6 und ein Signal *UpdateWeight*. Letzteres ist notwendig, um den Verbindungen den Zeitpunkt der Aktualisierung des Gewichtes zu signalisieren. Dies muss für alle Gewichte gleichzeitig geschehen, da sonst die Ergebnisse verfälscht werden. Um die Gewichte in Registern speichern zu können, werden nun auch für die Verbindungskomponente Clock- und Reset-Leitungen benötigt.

Die beiden erweiterten Modelle werden wie in Abbildung 4.6 verschaltet. Zusätzlich werden der Eingangsschicht und jeder verdeckten Schicht ein sogenanntes *Bias-Neuron* hinzugefügt (siehe Abbildung 4.8). Dieses spezielle Neuron hat keine Eingangswerte und gibt immer den Wert 1 am Ausgang aus. Es ist somit nur mit den Neuronen der nachfolgenden Schicht verbunden. Durch ein solches Bias-Neuron können die Aktivierungsfunktionen der nachfolgenden Neuronen virtuell verschoben werden, was die Wahrscheinlichkeit der Konvergenz des neuronalen Netzes erhöht [Ert16, Kap. 8.2].

Die Anzahl der benötigten Takte für eine Backpropagation entspricht wieder der Anzahl der Schichten des neuronalen Netzes. Für die Aktualisierung der Gewichte ist ein weiterer Taktzyklus notwendig. Es sei N die Anzahl der Schichten des Netzes, so sind für dieses Modell $2N + 1$ Takte für einen kompletten Lernzyklus inklusive Forward-Propagation nötig.

Durch die zusätzlichen Multiplikationen bei der Berechnung und Gewichtung der Gradienten sowie der Aktualisierung der Gewichte wird die Anzahl der benötigten DSP-Blöcke vervielfacht. Aus diesem Grund schrumpft die Größe eines synthetisierbaren neuronalen Netzes mit Backpropagation beträchtlich gegenüber einem nicht-lernfähigen mehrschichtigen Netz. In Kapitel 6 wird diese Einschränkung anhand der synthetisierten Prototypen verdeutlicht.

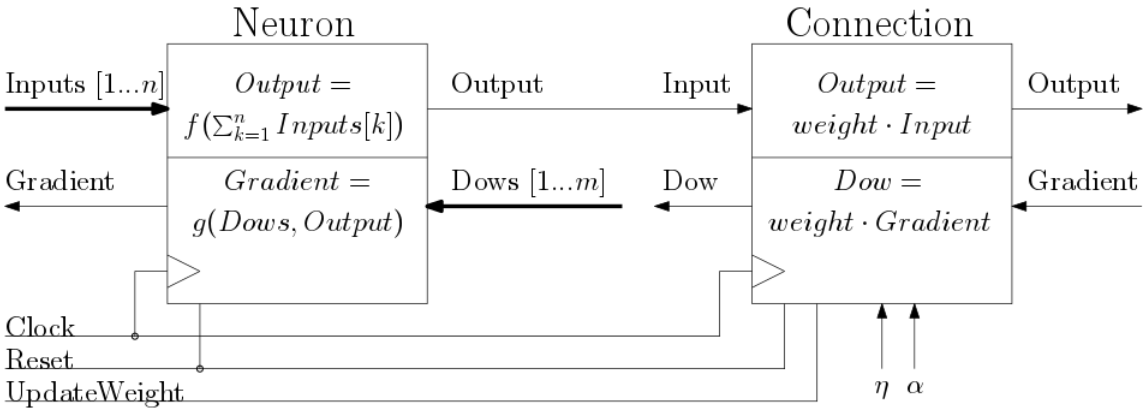


Abbildung 4.7: *Für die Backpropagation erweiterte Modelle der Neuronen und Synapsen.*

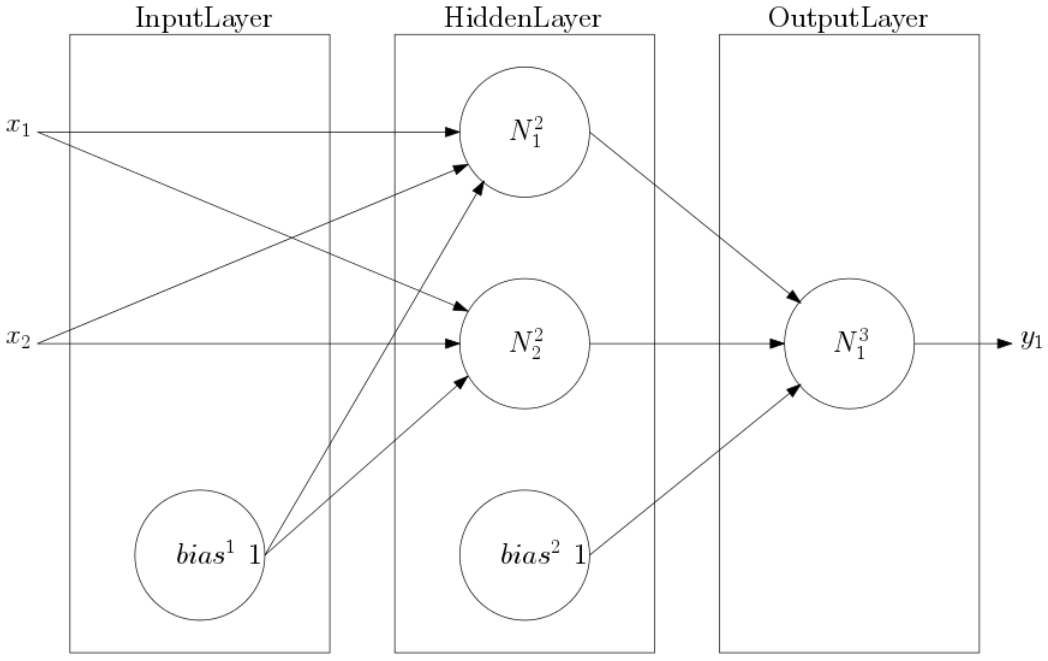


Abbildung 4.8: *Mehrschichtiges neuronales Netz mit Bias-Neuronen. Die Verbindungen werden durch Pfeile dargestellt.*

Kapitel 5

Testimplementierungen

In diesem Kapitel werden die entwickelten Prototypen und die Arbeitsumgebung vorgestellt.

5.1 Von der Simulation zum Hardware-Modell

Im Zuge der Arbeit wurde ein Software-Prototyp in C++ entwickelt, um die grundlegende Struktur des später entwickelten Hardware-Modells festzulegen und zu testen. Die Implementierung ist angelehnt an [Mil11]. Der entworfene Prototyp ist ein mehrschichtiges neuronales Netz mit Backpropagation. Ein Klassendiagramm der Implementierung wird in Abbildung 5.1 gezeigt. Im Unterschied zu dem in Kapitel 4 entwickelten neuronalen Netz sind die Verbindungen mit den Gewichten hier Teil der Neuronen. Ansonsten ist die Struktur ähnlich der des Hardware-Entwurfs. So können beispielsweise verschiedene Aktivierungsfunktionen bereits im Software-Entwurf getestet werden.

Bei der Instantiierung der Klasse *NeuralNet* werden im Konstruktor die Dimensionen des neuronalen Netzes sowie eine Funktion zur Aufbereitung der Ausgangswerte des neuronalen Netzes als Parameter mitgegeben. Anschließend können mit *NeuralNet.ForwardPropagate(Data)* die mitgegebenen Daten prozessiert werden. In der Funktion *NeuralNet.BackPropagate(Data)* wird mithilfe des mitgegebenen Target-Vektors eine Backpropagation durchgeführt. Mit *NeuralNet.Train(Data, Data)* werden schließlich Forwardpropagation und Backpropagation direkt hintereinander ausgeführt. *NeuralNet.getResults()* liefert die zuletzt prozessierten Ausgangsdaten des neuronalen Netzes, während *NeuralNet.getRecentError()* den durchschnittlichen quadratischen Fehler der letzten Durchläufe zurückgibt. Im Code-Ausschnitt 5.1 wird als Beispiel einem neuronalen Netz das logische Verhalten eines XOR-Gatters beigebracht.

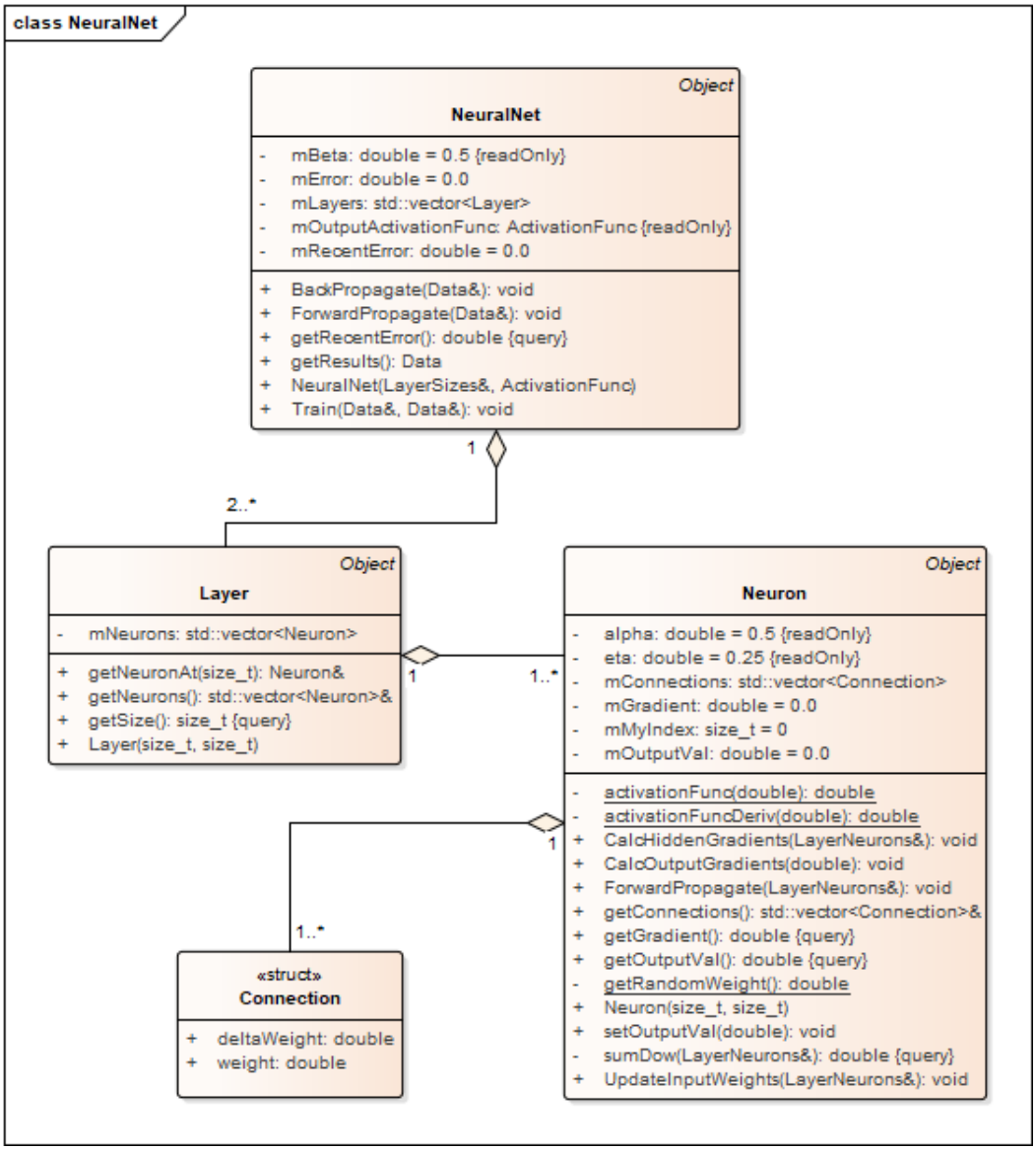


Abbildung 5.1: *Klassendiagramm des in C++ entwickelten Software-Prototypen eines mehrschichtigen neuronalen Netzes mit Backpropagation.*

5.2 Die Hardware-Implementierung

Die in Kapitel 4 spezifizierten Modelle wurden in VHDL umgesetzt. Dabei wurde einerseits ein nicht lernfähiges mehrschichtiges neuronales Netz entworfen, das die benötigten Gewichte als Eingänge bekommt. Andererseits wurde ein neuronales Netz mit Backpropagation implementiert. Die verschiedenen Implementierungen werden in Kapitel 6 in verschiedenen Variationen

Programm 5.1: Beispiel eines neuronalen Netzes in C++

```
#include "NeuralNet.h"

double PrepareResults(double const& x) {
  return x;
}

int main() {
  // Neuronales Netz mit zwei Eingangsneuronen, einer
  // verdeckten Schicht mit fünf Neuronen und einem Ausgangsneuron
  NeuralNet net({ 2, 5, 1 }, PrepareResults);
  // Eingangs- und Target-Vektoren
  Data test1 = { 0 0 }; Data target1 = { 0 };
  Data test2 = { 1 0 }; Data target2 = { 1 };
  Data test3 = { 0 1 }; Data target3 = { 1 };
  Data test4 = { 1 1 }; Data target4 = { 0 };

  // Die Anzahl der benötigten Durchläufe variiert je nach
  // Art der Aktivierungsfunktion, Größe des Netzes und Umfang
  // der prozessierten Daten zwischen mehreren hundert
  // und mehreren tausend Durchläufen.
  for (int i = 0; i < 1000; ++i) {
    net.Train(test1, target1);
    net.Train(test2, target2);
    net.Train(test3, target3);
    net.Train(test4, target4);
  }
}
```

miteinander verglichen. Dabei werden sowohl die Simulations- als auch die Synthese-Ergebnisse betrachtet. Die Netze werden in verschiedenen Größen sowie mit verschiedenen Aktivierungsfunktionen und Parametern getestet.

Die Schnittstelle des mehrschichtigen neuronalen Netzes ohne Backpropagation wird im Code-Ausschnitt 5.2 gezeigt. Über die *Generics* kann die Anzahl der Eingangs- und Ausgangsneuronen sowie die Anzahl der verdeckten Schichten und deren Neuronen spezifiziert werden. Als Eingänge bekommt das neuronale Netz ein Clock-Signal, ein Reset-Signal, die Eingänge, die Gewichte sowie das Start-Signal für einen Durchlauf. Als Ausgänge liefert das Netz die prozessierten Ausgangswerte und ein Signal für das Ende eines Durchlaufes. Für alle reellen Werte innerhalb der neuronalen Netze wurde ein eigener Datentyp *neuro_real* definiert, welcher aus einer Fixkomma-Zahl mit sechs Vorkomma- und zehn Nachkomma-Stellen besteht.

Programm 5.2: Schnittstelle des MLPs ohne Backpropagation in VHDL

```
1 subtype neuro_real is sfixed(5 downto -10);
2 type neuro_real_vector is array (natural range <>) of neuro_real;
3
4 entity MLP_Net is
5   generic(
6     gNumberInputs          : natural   := 2;
7     gNumberOutputs         : natural   := 1;
8     gNumberHiddenLayers    : natural   := 1;
9     gNumberNeuronsPerLayer : natural   := 5
10  );
11  port(
12    iClk              : in  std_ulogic;
13    inRst             : in  std_ulogic;
14    -- Neural net inputs
15    iInputs           : in  neuro_real_vector(...);
16    iInputWeights     : in  neuro_real_vector(...);
17    iHiddenWeights    : in  neuro_real_vector(...);
18    iOutputWeights    : in  neuro_real_vector(...);
19    iStart            : in  std_ulogic;
20    -- Neural net outputs
21    oOutputs          : out neuro_real_vector(...);
22    oFinishedAll      : out std_ulogic
23  );
24 end entity;
```

Der Code-Ausschnitt in Programm 5.3 zeigt die Schnittstelle des neuronalen Netzes mit Backpropagation. Die *Generics* beinhalten zusätzlich einen Parameter zur Spezifizierung des Lerntyps. Dieser kann die Werte *Supervised* oder *Reinforced* annehmen. Statt der Gewichte wird ein Target-Vektor am Eingang angelegt. Des weiteren werden die Parameter η und α für die Aktualisierung der Gewichte angelegt. Außerdem wird dem Netz durch *iLearn* signalisiert, ob die nächste Iteration mit oder ohne Backpropagation erfolgen soll. Das Netz signalisiert am Ausgang jeweils der Ende der Forward- und Backpropagation und gibt den Fehler der letzten Iteration aus.

Programm 5.3: Schnittstelle des MLPs mit Backpropagation in VHDL

```
1 entity BP_Net is
2   generic(
3     gLearning              : tLearning := Supervised;
4     gNumberInputs          : natural   := 2;
5     gNumberOutputs         : natural   := 1;
6     gNumberHiddenLayers    : natural   := 1;
7     gNumberNeuronsPerLayer : natural   := 5
8   );
9   port(
10    iClk              : in  std_ulogic;
11    inRst             : in  std_ulogic;
12    -- Neural net inputs
13    iInputs           : in  neuro_real_vector(...);
14    iTargets          : in  neuro_real_vector(...);
15    iStart            : in  std_ulogic;
16    -- Weight update inputs
17    iEta              : in  neuro_real;
18    iAlpha            : in  neuro_real;
19    iLearn            : in  std_ulogic;
20    -- Neural net outputs
21    oOutputs          : out neuro_real_vector(...);
22    oFinishedForward  : out std_ulogic;
23    oFinishedBackward : out std_ulogic;
24    oFinishedAll      : out std_ulogic;
25    oError            : out neuro_real
26  );
27 end entity;
```

5.3 Verwendete Arbeitsumgebung

In der folgenden Übersicht werden die im Rahmen dieser Arbeit verwendeten Hard- und Software-Werkzeuge, deren Hersteller sowie die verwendeten Sprachen aufgelistet.

- **Software-Prototyp**
  Sprache: C++
  Hersteller: Microsoft Corporation
  Bezeichnung: Visual Studio Community 2015
  Version: 14.0.25431.01 Update 3
- **Prototyp für den FPGA**
  Sprache: VHDL
  Hersteller: Sigasi
  Bezeichnung: Sigasi Studio
  Version: 3.3.0.201612051610

- **Simulation der Hardware-Beschreibung**
  Hersteller: Mentor Graphics, Inc.
  Bezeichnung: ModelSim PE Student Edition
  Version: 10.4a 2015.03
- **Synthese der Hardware-Beschreibung**
  Hersteller: Intel Corporation
  Bezeichnung: Quartus Prime Lite Edition
  Version: 16.1.0
- **Verwendeter FPGA**
  Hersteller: Intel Corporation
  Bezeichnung: Cyclone V
  Version: 5CSEMA5F31C6

Kapitel 6

Testergebnisse und Vergleiche

Nachfolgend werden die Simulations- und Syntheseergebnisse von mehrschichtigen neuronalen Netzen mit verschiedenen Parametern und Aktivierungsfunktionen dargestellt und verglichen.

6.1 Das XOR-Problem

Um die richtige Funktionsweise eines neuronalen Netzes zu verifizieren, eignet sich die Abbildung des Verhaltens eines XOR-Gatters. Das Beibringen dieses logischen Verhaltens ist eine besondere Herausforderung, da ein Datensatz dieser Art nicht *linear separabel* ist. In der Abbildung 6.1 werden die Datensätze eines OR- und eines XOR-Gatters zweidimensional dargestellt. Die Ausgangswerte eines OR-Gatters sind durch eine lineare Funktion trennbar, während die Ausgangswerte eines XOR-Gatters nur durch eine nichtlineare Funktion trennbar sind. Die Trennung von linear separablen Daten kann durch ein Neuron bewerkstelligt werden, während für nicht linear separable Daten mindestens eine verdeckte Schicht zur korrekten Klassifizierung benötigt wird [Ert16, Kap. 8.2]. Daher wurden die Netze in den folgenden Simulationen anhand des XOR-Problems getestet.

6.2 Simulationsergebnisse

Nachfolgend werden verschiedene Simulationsdurchläufe grafisch dargestellt und diskutiert. Die Diagramme zeigen den durchschnittlichen Fehler ΔRMS des Netzes über der Anzahl der Iterationen i. Der durchschnittliche Fehler berechnet sich aus dem RMS (Root Mean Square, quadratisches Mittel) zu

$$\Delta RMS(t) = \frac{\Delta RMS(t-1) \cdot \beta + RMS(t)}{\beta + 1}$$

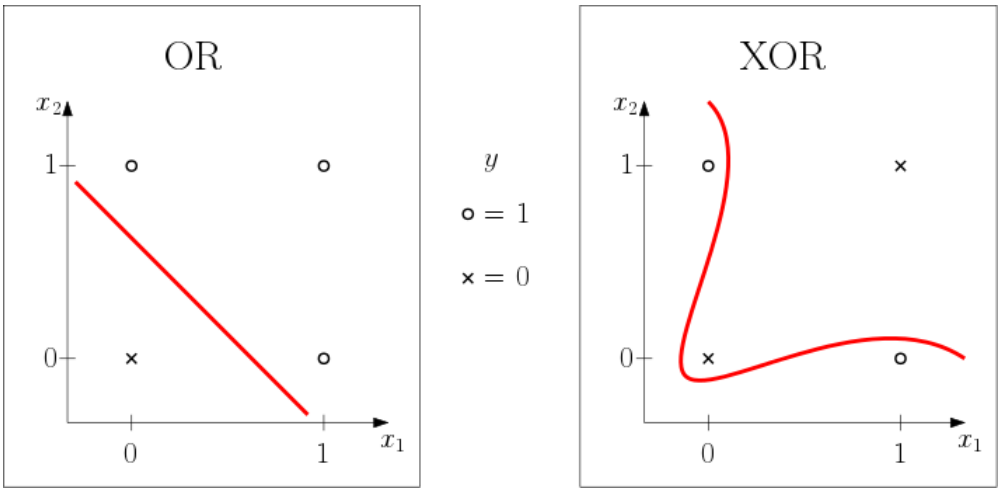


Abbildung 6.1: *Das XOR-Problem zweidimensional dargestellt. Auf der Abszisse und Ordinate sind jeweils die Eingangswerte x_1 und x_2 dargestellt.*

wobei β einen konstanten Gewichtungsfaktor des Fehlers der letzten Iteration darstellt. Das *RMS* selbst ergibt sich aus den Werten des Target-Vektors $\boldsymbol{t}$ und den Ausgangswerten $\boldsymbol{y}$ der Ausgangsneuronen sowie der Anzahl der Ausgangsneuronen N zu

$$RMS = \sqrt{\frac{\sum_{k=1}^{N}(t_k - y_k)^2}{N}}$$

6.2.1 Software-Simulationen mit C++

In den Abbildungen 6.2, 6.3 und 6.4 werden jeweils drei Lernzyklen mit zufällig initialisierten Gewichten dargestellt. Das simulierte mehrschichtige neuronale Netz hat zwei Eingangsneuronen, eine verdeckte Schicht mit fünf Neuronen und ein Ausgangsneuron sowie jeweils ein Bias-Neuron in der Eingangs- und verdeckten Schicht. Für die Berechnung der Gradienten werden die Gleichungen 4.1 und 4.2 mit verschiedenen abgeleiteten Aktivierungsfunktionen verwendet.

Abbildung 6.2 zeigt drei Testdurchläufe des MLPs mit der Aktivierungsfunktion $f(x) = tanh(x)$. Für die Berechnung der Gradienten wird die an $\frac{d}{dx}tanh(x)$ angenäherte Funktion $f'(x) = 1 - x^2$ verwendet. Die Lernrate η ist bei 0.15 und α bei 0.5. Der durchschnittliche Fehler fällt bei zwei Durchläufen innerhalb von 500 Iterationen auf unter drei Prozent. Bei einem Durchlauf ist diese Grenze erst bei etwa 700 Iterationen erreicht. Beim ersten Durchlauf schwingt sich die Fehlerrate bei etwa 625 Iterationen noch einmal auf über 15 Prozent auf.

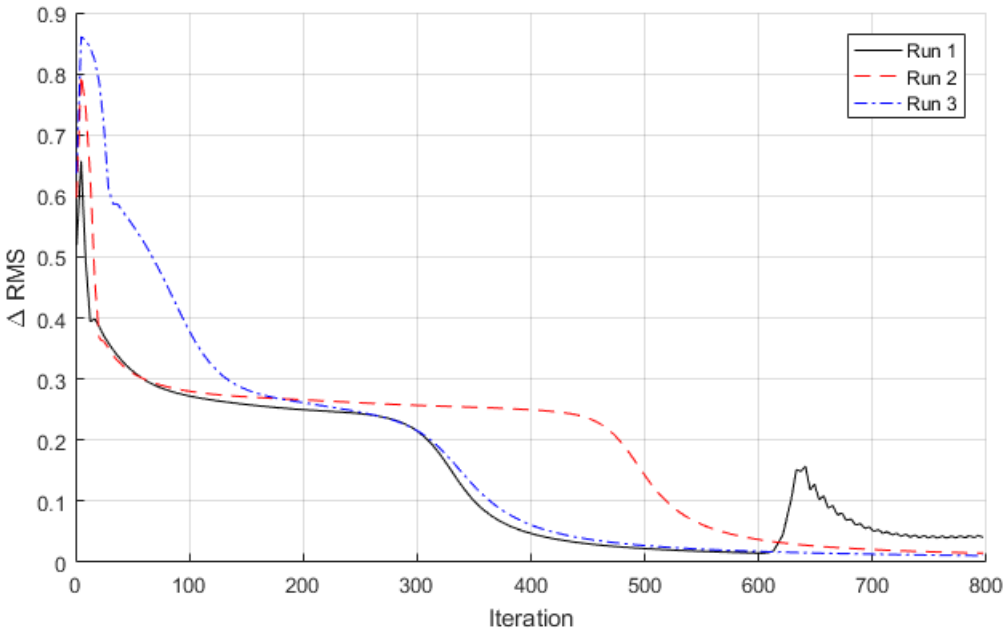


Abbildung 6.2: *Verlauf der Fehlerrate mit der Aktivierungsfunktion* $tanh(x)$, $\eta = 0.15$ *und* $\alpha = 0.5$.

Abbildung 6.3 zeigt das Lernverhalten des neuronalen Netzes mit der Aktivierungsfunktion aus der Gleichung 4.3. Für die Berechnung der Gradienten wird die von $tan^{-1}(x)$ abgeleitete Funktion

$$f'(x) = \frac{d}{dx} tan^{-1}(x) = \frac{1}{1+x^2} \tag{6.1}$$

verwendet. Diese zeichnete sich in mehreren Versuchen als beste Funktion für diese Berechnung aus. η und α sind wie in den vorherigen Testläufen konstant bei 0.15 und 0.5. Bei den drei Testläufen können große Unterschiede bei der Konvergenzdauer der Fehlerrate beobachtet werden. Allerdings nähert sich die Fehlerrate schnell an die Null-Linie an, wenn etwa 40 Prozent unterschritten sind.

In drei weiteren Testläufen werden ebenfalls die Funktionen aus den vorherigen Tests verwendet. Allerdings wird hier für die Lernrate η die mit einem konstanten Faktor γ gewichtete durchschnittliche Fehlerrate ΔRMS verwendet. Die Lernregel ergibt sich nun zu

$$\Delta w_{ij}^p(t) = \gamma \delta_j^p x_i^p \Delta RMS(t-1) + \alpha \Delta w_{ij}^p(t-1) \tag{6.2}$$

Bei den Durchläufen in Abbildung 6.4 ist $\alpha = 0$ und $\gamma = 0.25$. Die Fehlerrate nähert sich mit dieser Aktualisierungsfunktion nur sehr langsam an die

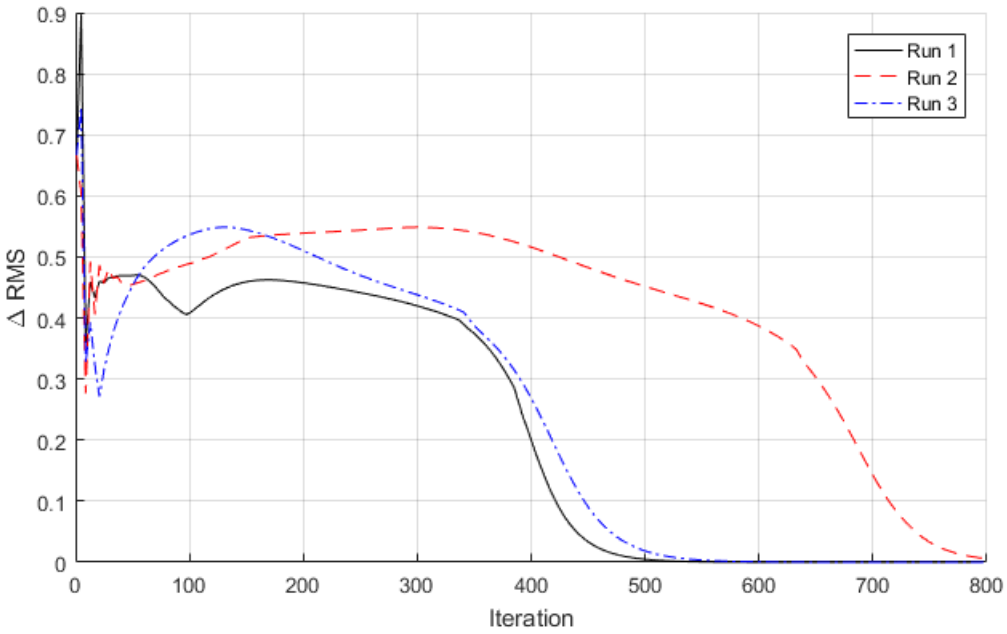


Abbildung 6.3: *Verlauf der Fehlerrate mit der an $\tanh(x)$ angenäherten Aktivierungsfunktion aus Gleichung 4.3, $\eta = 0.15$ und $\alpha = 0.5$.*

Null-Linie an, jedoch ist das Lernverhalten des neuronalen Netzes bei allen Durchläufen ähnlich. Dadurch kann die Anzahl der benötigten Iterationen bis zur Unterschreitung einer gewissen Grenze besser vorhergesagt werden. Ein weiterer Vorteil ist die Elimination einer Multiplikation pro Verbindung, da mit $\alpha = 0$ der Impulsterm wegfällt.

6.2.2 Hardware-Simulationen

Für die Hardware-Simulationen wurde ein mehrschichtiges neuronales Netzes mit Backpropagation verwendet. Dieses besitzt ebenfalls zwei Eingangsneuronen, eine verdeckte Schicht mit fünf Neuronen und ein Ausgangsneuron sowie jeweils ein Bias-Neuron in der Eingangs- und verdeckten Schicht.

Abbildung 6.5 zeigt den Vergleich der Konvergenz von jeweils zwei Simulationen des Hardware-Modells mit unterschiedlichen Datentypen für alle reellen Werte innerhalb des Netzes. Für beide Simulationen wurden die Gewichte jeweils mit den selben Werten initialisiert. Die gestrichelten Kurven sind das Ergebnis der Simulationen des Netzes mit dem Typ *REAL* aus der *IEEE.MATH_REAL* Bibliothek, die durchgezogenen Kurven sind das Ergebnis der Verwendung von 16-Bit-Fixkomma-Zahlen mit 6 Vorkomma- und 10 Nachkomma-Stellen. Als Aktivierungsfunktion wird jeweils die Funktion

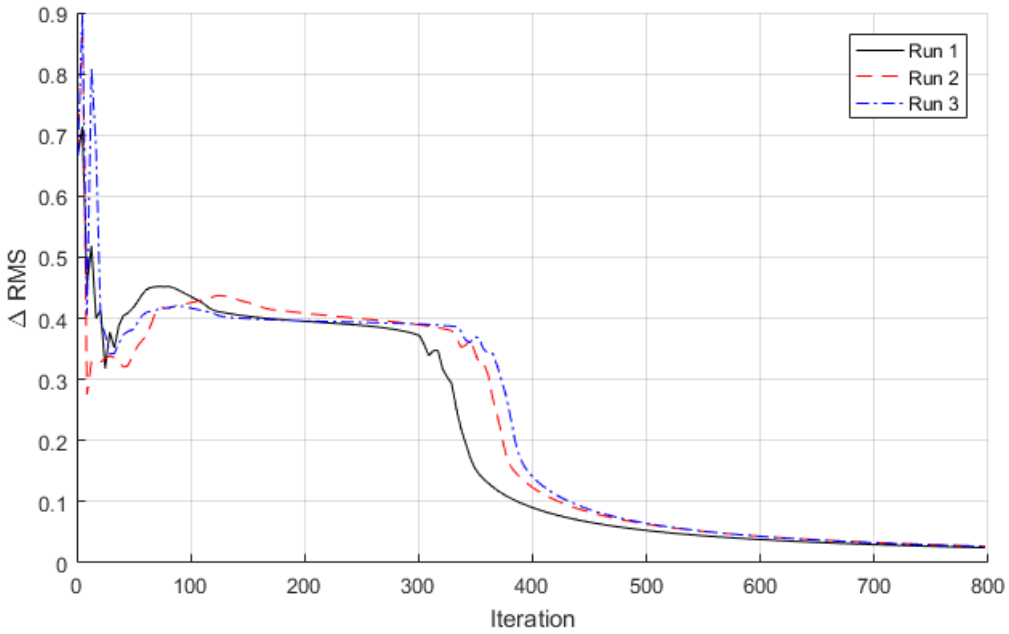


Abbildung 6.4: *Verlauf der Fehlerrate mit der Verwendung des ΔRMS für die Lernrate η.*

aus Gleichung 4.3 verwendet, für die Berechnung der Gradienten wird die Funktion aus Gleichung 6.1 verwendet.

Im Bereich zwischen 0 und $0,2$ auf der Ordinate sind kleine Abweichungen ersichtlich, die sich aus der Ungenauigkeit der Fixkomma-Zahlen ergeben. Durch den Einsatz der Fixkomma-Zahlen entstehen allerdings keine Nachteile, was die Konvergenz der Gradienten betrifft.

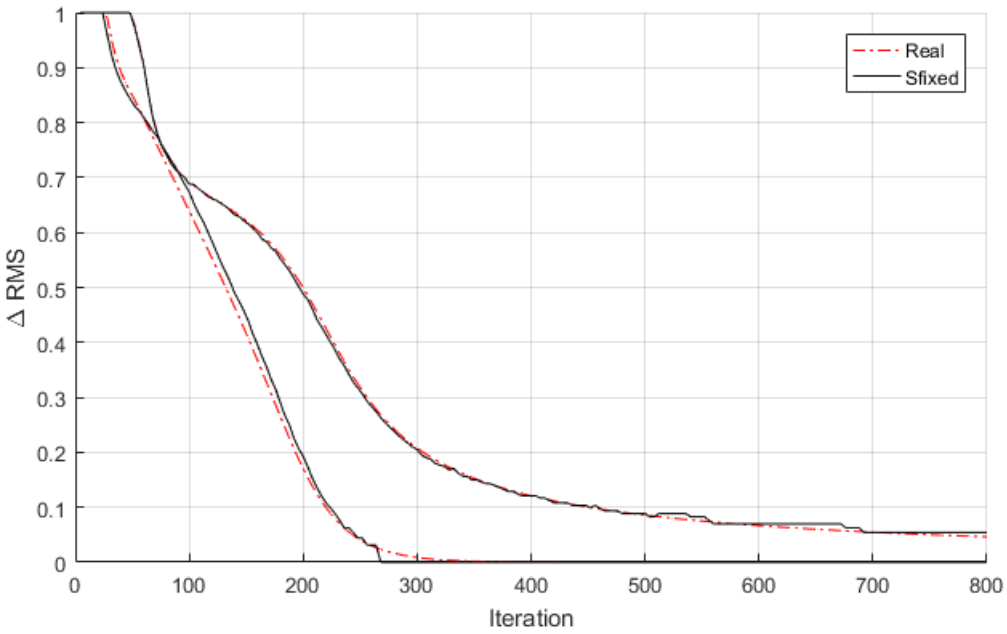


Abbildung 6.5: *Vergleich der Simulationen des VHDL-Modells mit Real- und Sfixed-Werten. Links wird die Aktualisierungsfunktion aus Gleichung 3.6 mit $\eta = 0.15$ und $\alpha = 0.5$ verwendet, rechts wird die Funktion aus Gleichung 6.2 mit $\gamma = 0.25$ und $\alpha = 0$ verwendet.*

6.3 Syntheseergebnisse

Alle synthetisierten Netze weisen die gleiche Topologie wie die Netze in den Hardware-Simulationen auf. Die Anzahl der Neuronen $N = 8$, die Anzahl der Verbindungen $V = 21$. Die Netze wurden für ein FPGA vom Typ Intel Cyclone V 5CSEMA5F31C6 synthetisiert. Dieser FPGA besitzt 87 DSP-Blöcke und 32 070 *Adaptive Logic Modules* (ALMs) [Alt16]. In Tabelle 6.1 werden die wichtigsten Kennzahlen der Syntheseergebnisse verschiedener mehrschichtiger neuronaler Netze dargestellt.

In einem ersten Durchlauf wurde ein neuronales Netz ohne Backpropagation synthetisiert. Da in diesem Netz keine Gradienten berechnet werden und auch keine Gewichtsaktualisierung durchgeführt wird, sind nur DSP-Einheiten für die Gewichtung der Ausgangssignale der Neuronen nötig. Diese Gewichtung findet bei jeder Verbindung von einem Neuron zum nächsten statt. Da auch die Verbindungen von den Bias-Neuronen zu den nachfolgenden Neuronen gewichtet werden, ergeben sich $V = 21$ Multiplikationen. Der maximale Takt von $38,37$ MHz lässt sich durch die Betrachtung des kritischen Pfades erklären, welcher von einem Neuron zum nächsten durch

Tabelle 6.1: Vergleich der Syntheseergebnisse verschiedener Multi-Layer-Perceptrons (MLP) mit und ohne Backpropagation (BP).

Eigenschaft	**MLP ohne BP**	**MLP mit BP,** $\alpha = 0$	**MLP mit BP,** $\alpha = 0.5$
Anzahl DSPs	21	87 (102)	87 (123)
Anzahl ALMs	715	11 915	12 542
Anzahl Register	93	737	1044
Max. Takt [MHz]	38.37	5.45	5.31

jeweils eine DSP-Einheit führt.

Im nächsten Durchlauf wurde ein mehrschichtiges Netz mit Backpropagation und der Lernregel aus Gleichung 6.2 mit $\alpha = 0$ synthetisiert. Mit dieser Lernregel entfällt im Vergleich zum dritten synthetisierten Netz eine Multiplikation pro Verbindung. Für die Berechnung der Gradienten werden pro Neuron jeweils zwei DSP-Blöcke benötigt. Pro Verbindung sind jeweils zwei DSP-Einheiten für die Gewichtung der Signale und für die Aktualisierung der Gewichte nötig. Die Gesamtanzahl der benötigten Multiplikationen berechnet sich zu $2N + 4V = 102$. Da das verwendete FPGA nur 87 DSP-Einheiten besitzt, wurden einige Multiplikationen durch *Add-And-Shift*-Operationen ersetzt. Diese Tatsache und der durch mehrere Multiplikationen verlängerte kritische Pfad erklären den niedrigeren maximalen Takt von $5,45$ MHz sowie die hohe Anzahl der benötigten ALMs.

Ein mehrschichtiges Netz mit Backpropagation und der Lernregel aus Gleichung 3.6 sowie $\alpha = 0,5$ wurde im dritten Durchlauf synthetisiert. Im Vergleich zum zweiten Durchlauf benötigt dieses Netz eine Multiplikation pro Verbindung zusätzlich, weshalb die Anzahl der benötigten Multiplikationen auf $2N + 5V = 123$ steigt.

Kapitel 7

Ausblick

Methoden der KI im Generellen und neuronale Netze im Speziellen sind wichtige Werkzeuge für aktuelle und zukünftige intelligente Systeme. FPGAs sind durch ihre parallele Struktur und ihren geringeren Energieumsatz im Vergleich zu Hochleistungs-GPUs als Plattformen für neuronale Netze prädestiniert.

In dieser Arbeit wurden verschiedene Hardware-Modelle mehrschichtiger neuronaler Netze für FPGAs entwickelt. Die Funktionsweise wurde anhand der Simulation des XOR-Problems und anhand des Vergleichs mit einer Software-Simulation eines mehrschichtigen neuronalen Netzes verifiziert.

Durch die Analyse der Funktionen und arithmetischen Operationen innerhalb eines neuronalen Netzes wurde festgestellt, dass die häufigen Multiplikationen die Größe eines synthetisierbaren Netzes stark limitieren. Diese Erkenntnis wurde durch die Vergleiche der Syntheseergebnisse der Hardware-Modelle untermauert. Allerdings können auf modernen Hochleistungs-FPGAs größere neuronale Netze instantiiert werden, welche beispielsweise für eine Objekterkennung in Bildern eingesetzt werden können.

Als abschließende Erkenntnis kann gesagt werden, dass moderne FPGAs mit ihren mehrfach erwähnten Eigenschaften eine sinnvoll einsetzbare Plattform für schnelle neuronale Netze sind. Die maximale Größe und die Lerngeschwindigkeit hängen überwiegend von der Anzahl und der Geschwindigkeit der vorhandenen Multiplikatoren beziehungsweise DSP-Blöcken ab.

Literatur

[Alt16] Altera. *Cyclone V Specifications*. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_51002.pdf. [Online; Stand vom 21.05.2017]. 2016 (siehe S. 39).

[Ert16] Wolfgang Ertel. *Grundkurs Kunstliche Intelligenz*. Springer, 2016 (siehe S. 5–7, 13, 23, 26, 34).

[HB91] Jordan L Holt und Thomas E Baker. „Back propagation simulations using limited precision calculations“. In: *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*. Bd. 2. IEEE. 1991, S. 121–126 (siehe S. 19).

[Heb49] Donald Olding Hebb. *The organization of behavior: A neuropsychological approach*. John Wiley & Sons, 1949 (siehe S. 11).

[Hop82] John J Hopfield. „Neural networks and physical systems with emergent collective computational abilities“. *Proceedings of the national academy of sciences* 79.8 (1982), S. 2554–2558 (siehe S. 12).

[IEE08] IEEE. „IEEE Standard for Floating-Point Arithmetic“. *IEEE Std 754-2008* (Aug. 2008), S. 1–70 (siehe S. 19).

[Mil11] David Miller. *Make a Neural Net Simulator in C++*. http://www.millermattson.com/dave/?p=54. [Online; Stand vom 21.05.2017]. 2011 (siehe S. 28).

[MP43] Warren S McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. *The bulletin of mathematical biophysics* 5.4 (1943), S. 115–133 (siehe S. 9).

[nVi16] nVidia. *NVIDIA TESLA P100*. http://www.nvidia.com/object/tesla-p100.html. [Online; Stand vom 21.05.2017]. 2016 (siehe S. 2).

[OJ04] Kyoung-Su Oh und Keechul Jung. „GPU implementation of neural networks“. *Pattern Recognition* 37.6 (2004), S. 1311–1314 (siehe S. 1).

[OR06] Amos R Omondi und Jagath Chandana Rajapakse. *FPGA implementations of neural networks*. Bd. 365. Springer, 2006 (siehe S. 1).

[RN05] Stuart Russell und Peter Norvig. „Ai a modern approach“. *Learning* 2.3 (2005), S. 4 (siehe S. 1, 4).

[Sil+16] David Silver u. a. „Mastering the game of Go with deep neural networks and tree search“. *Nature* 529.7587 (2016), S. 484–489 (siehe S. 17).

Abbildungsverzeichnis

1.1 GPU-Schematic 2

3.1 Neuronen im Gehirn 8
3.2 Synapse 9
3.3 Mathematisches Neuron 10
3.4 Aktivierungsfunktionen 11
3.5 Hopfield-Netz 12
3.6 Mehrschichtiges neuronales Netz 14
3.7 Backpropagation 15
3.8 Konvergenzproblem 16

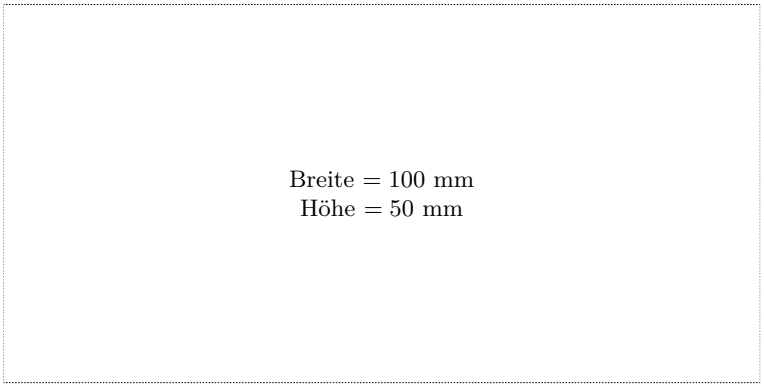
4.1 Reelle Zahlen 20
4.2 Hardware-Modell eines Neurons 21
4.3 Aktivierungsfunktion in der Hardware 22
4.4 Verbindung in der Hardware 23
4.5 Hopfield-Netz in der Hardware 23
4.6 Multi-Layer-Perceptron in der Hardware 25
4.7 Backpropagation-Modell Neuron und Verbindung 27
4.8 MLP mit Bias-Neuronen 27

5.1 Cpp-Klassendiagramm 29

6.1 XOR-Problem 35
6.2 Cpp-Simulation mit tanh(x) 36
6.3 Cpp-Simulation mit Annäherung an tanh(x) 37
6.4 Cpp-Simulation mit Rückführung des RMS 38
6.5 VHDL-Simulation Vergleich Real und Sfixed 39

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —