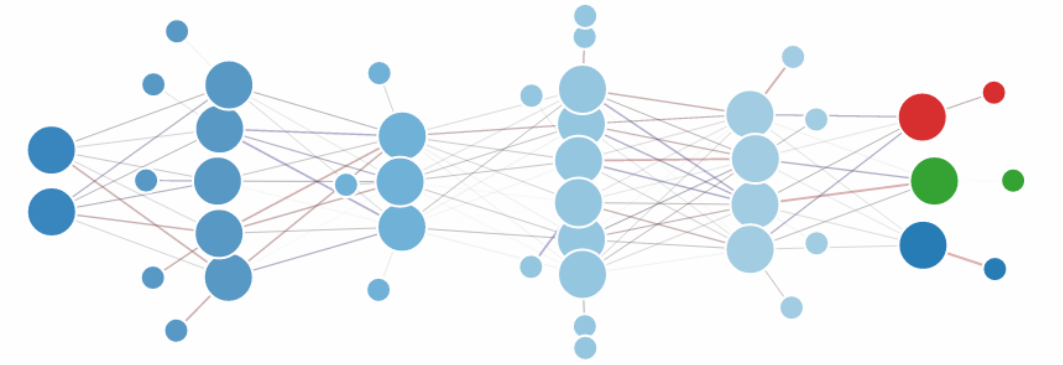
# Realisierung eines interaktiven künstlichen neuronalen Netzwerks

BACHELORTHESIS
zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)
Studiengang Media Systems




Finn Ole Koenecke
2151235


Erstprüfer: Prof. Dr. Edmund Weitz
Zweitprüfer: Prof. Dr. Andreas Plaß


Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik


29. Februar 2016

## Abstract

The goal of this thesis is to make structures and concepts behind artificial neural networks comprehendable. Therefore a network is being implemented, that can be controlled by a graphical user interface. Simultaneously, it provides information on internal states and processes to the user, that remain concealed under normal circumstances. This interaction enables users to learn the funcionality of neural networks by example.



## Zusammenfassung

Diese Arbeit hat zum Ziel, die Strukturen und Konzepte hinter künstlichen neuronalen Netzwerken begreifbar zu machen. Dazu wird ein Netzwerk so implementiert, dass es über eine graphische Oberfläche gesteuert werden kann. Gleichzeitig liefert das Netzwerk Informationen zu internen Zuständen und Abläufen, die normalerweise verborgen bleiben, an den Benutzer zurück. Durch diese Interaktion können Anwender die Funktionsweise von neuronalen Netzwerken direkt am Beispiel erlernen.

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Begriffsklärung

Der Ausdruck *neuronales Netz* stammt ursprünglich aus der Biologie. Der Themenkomplex müsste in der Informatik korrekterweise immer mit *künstliches neuronales Netz* oder *simuliertes neuronales Netz* beschrieben werden. Um diese sperrigen Formulierungen zu umgehen, wird auch in der Informatik nur von *neuronalen Netzen*, oder *ann (artificial neural networks)*, gesprochen.

Auch diese Arbeit verwendet die kürzere Form, solange der Kontext eindeutig ist. Bei Vergleichen zwischen biologischen und digitalen Netzen wird eine sprachliche Trennung vorgenommen. Des Weiteren werden die Begriffe *Netz* und *Netzwerk* synonym für den Verbund neuronaler Einheiten verwendet.

## 1.2 Künstliche Intelligenz

Künstliche Intelligenz (KI) ist wohl das Thema der Informatik, das Fachleute, wie Außenstehende gleichermaßen fasziniert. Maschinen, die ihr eigenes Bewusstsein entwickelt haben, spielen in etlichen Zukunftsszenarien, utopischen wie dystopischen, eine tragende Rolle. Die allgemeine Vorstellung ist, dass Roboter das menschliche Verhalten irgendwann so gut imitieren, dass kein Unterschied mehr festgestellt werden kann. Beispielsweise wird in *Ridley Scotts* prominenter Science Fiction Dystopie *Blade Runner* diskutiert, wie sich solche künstlichen Menschen in die Gesellschaft einfügen könnten. Aber auch in der realen Forschung erleben das Thema Künstliche Intelligenz und die damit verbundenen Zukunftsvisionen immer wieder Popularitätsschübe. Das ist nur konsequent, wenn man bedenkt, dass schon Alan Turing in den Anfängen der modernen Informationstechnologie die Frage stellte: “Können Maschinen denken?” [Tur50, 433].

Um auch eine Antwort auf diese Frage finden zu können, schlug er den mittlerweile nach ihm benannten *Turing-Test* vor. Dabei kommuniziert eine Testperson per Chat mit einem anonymen Gegenüber. Anhand des Gesprächsverlaufs muss die Testperson

entscheiden, ob sie mit einem Menschen oder einer Maschine verbunden wurde. Der Test gilt als bestanden, wenn die Testperson das nicht unterscheiden kann. Obwohl er schon 1950 formuliert wurde, hat es bis heute kein Programm geschafft, den Test zu bestehen. Neben Problemlösungskompetenzen sind nämlich auch kreative Denkprozesse und Selbstbestimmtheit Anforderungen an eine solche sogenannte *starke KI*, die menschliches Verhalten tatsächlich imitiert.

Die Erschaffung so eines künstlichen Bewusstseins liegt nach allgemeiner Auffassung noch weit entfernt [Mue13]. Wird heute von künstlicher Intelligenz gesprochen, die bereits Einzug in unseren Alltag gehalten hat, ist damit die *schwache KI* gemeint. Diese stellt eine Simulation intelligenten menschlichen Verhaltens bei bestimmten Problemstellungen dar. Sie hat dabei nicht den Anspruch eines autonomen Bewusstseins, sondern ahmt lediglich menschliche Wahrnehmung und Entscheidungen bei speziellen Aufgabenstellungen mit Mitteln der Mathematik und der Informatik nach. Zwar kann hierbei nicht von tatsächlicher Intelligenz gesprochen werden, trotzdem sind die jüngeren Entwicklungen beachtlich.

Am bekanntesten sind wohl künstliche Assistenten in Mobiltelefonen, wie *Apples Siri* oder *Micosofts Cortana*, die auf verbal formulierte Fragen und Anforderungen mit Hilfe von Mustererkennung und Wahrscheinlichkeitsberechnungen sinnvoll reagieren können. Sie ahmen damit die menschliche Fähigkeit, etwas zu verstehen, nach. Akustische Signale werden also nicht nur aufgenommen, sondern auch in einen Zusammenhang gesetzt.

### 1.2.1 methodische Ansätze

Für die Umsetzung einer künstlichen Intelligenz gibt es verschiedene Herangehensweisen. Es stellt sich die Frage nach dem eingesetzten Mittel, bzw. wie eine Simulation technisch ablaufen soll. Es wird unterschieden zwischen *Neuronaler KI*, die den Aufbau eines Gehirns als Interaktion einer Vielzahl einfacher Einheiten nachahmen will, und *Symbolischer KI*, die ein Modell mit Hilfe von Zeichen (Symbolen) und Verarbeitungsvorschriften definiert. Der erste Ansatz wird Konnektionismus [Gar15] genannt und kommt beispielsweise bei neuronalen Netzwerken zum Einsatz. Der zweite Ansatz basiert auf der *physical symbol system hypothesis (PSSH)* [Wik15], die davon ausgeht, dass sich Intelligenz als formales Regelwerk ausdrücken lässt und damit auch von einer Maschine erlernbar sei.

Außerdem muss entschieden werden, was imitiert werden soll. Auch hier gibt es generell zwei Ausprägungen. Bei der ersten wird versucht, kognitive Abläufe im Gehirn nachzubilden, um daraus konkretes Verhalten abzuleiten. Diese Methode wird häufig in der Bioinformatik eingesetzt, um Gehirnsimulationen zur Forschung zu entwickeln. Hierbei geht es nicht primär um die Erschaffung künstlicher Intelligenz, sondern um einen Erkenntnisgewinn in Fragen der Neurobiologie. Trotzdem besteht die Theorie, dass bei einer

möglichst exakten Nachbildung von Hirnaktivitäten automatisch eine Intelligenz entsteht. Dieser *Bottom–Up*–Ansatz wird beispielsweise im umstrittenen *Human Brain Project* [Hum16] verfolgt.

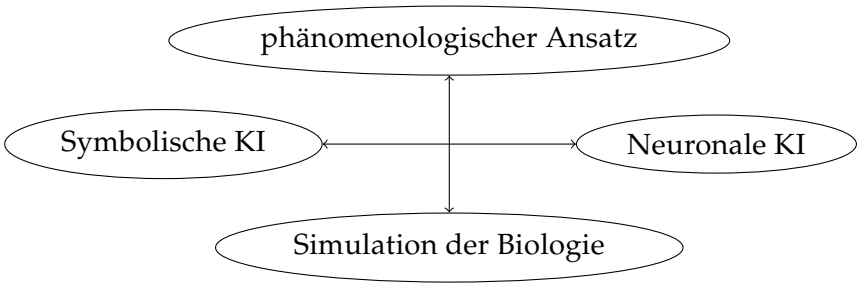



**Abbildung 1.1:** Spannungsfeld der methodischen Ansätze in der künstlichen Intelligenz

Die andere Methode basiert auf der Phänomenologie. Es werden beobachtete Zusammenhänge als gewünschte Ergebnisse definiert und versucht, diese zu imitieren. In dieser *Top–Down*–Herangehensweise wird die Korrektheit der Modelle vernachlässigt, solange die richtigen Resultate erzielt werden. Der Einsatz von neuronalen Netzen im Zusammenhang mit schwacher KI fällt in diese Kategorie. Die Netzwerke werden bei ihrer Anwendung als *Black Box* betrachtet, deren primäres Ziel es ist, aus bestimmten Eingaben passende Ausgaben zu generieren. Interne Abläufe können deshalb auch durch Algorithmen hergeleitet werden. In hochautomatisierten Szenarien kann das ein entscheidender Vorteil sein.

Die vorgestellten Kategorien sind dabei nicht als exklusiv zu verstehen. Ein konnektionistischer Ansatz ist nicht komplett ohne Algorithmen und Symbole zu realisieren. Auch ist es sinnvoll, die Prinzipien und Mechanismen eines Systems zu verstehen und sich nicht nur auf dessen Ergebnisse zu verlassen. Das aufgezeigte Spannungsfeld (Abb. 1.1) dient zur Einordnung von Themen im Rahmen der künstlichen Intelligenz. Diese Arbeit befasst sich mit dem Aufbau von künstlichen neuronalen Netzen als Beispiel für ein konnektionistisches Modell. Dabei steht nicht die Simulation eines Nervensystems im Vordergrund, sondern wie gesammelte Phänomene einem Netzwerk präsentiert werden müssen, damit es daraus entsprechende Verhaltensweisen ableitet.

## 1.3 Zielsetzung

Eine effiziente Art zu lernen ist es, Dinge selbst auszuprobieren, damit herumzuexperimentieren, neugierig zu sein und Schlüsse zu ziehen. Sich aktiv damit zu beschäftigen und darüber auszutauschen ist wichtig, um ein tiefes Verständnis für ein Thema zu erlangen [Kok15]. In der Informatik und der Mathematik ist dies oft schwierig, weil die Konzepte für viele

abstrakt und wenig greifbar sind. Alle Regeln eines Modells im Kopf zu behalten, und gleichzeitig ständig zu überprüfen, ist schwierig.

Um Themen aus den genannten Bereichen erfahrbarer zu machen, kann es deshalb helfen, das Modell zu vereinfachen und ein Werkzeug zu finden, das einerseits Interaktion ermöglicht und andererseits Vorstellungsarbeit abnimmt. So kann ein Lernender ausprobieren und Implikationen erleben, muss aber nicht direkt alle grundlegenden Mechanismen selbst kennen.

Grundschüler lernen die Addition beispielsweise auch leichter mit ihren Fingern. Dieses Hilfsmittel vereinfacht das Rechnen auf unterschiedliche Arten. Es begrenzt den Zahlenraum und verringert damit die Komplexität des Modells. Alle nötigen Regeln werden implizit durch den Aufbau der Hand vorgegeben. Außerdem macht das Abzählen der Finger den Ablauf der Berechnung transparent. Das Kind kann jederzeit eingreifen, selbst Vermutungen aufstellen und diese direkt überprüfen. Im Idealfall stößt es an die Grenzen des Werkszeugs und stellt weiterführende Fragen.

Das Ziel dieser Arbeit ist es, künstliche neuronale Netze im Hinblick darauf aufzuarbeiten. Die Grundidee besteht daraus, ein neuronales Netz zu implementieren, dessen Abläufe zu jeder Zeit angehalten und dessen interner Status eingesehen werden kann. Über eine grafische Oberfläche ist es dann möglich, diese Funktionen aufzurufen, um das Netzwerk zu bedienen. Währenddessen können die internen Mechanismen beobachtet werden. So kann ein Anwender testweise Eingaben tätigen, deren Auswirkungen beobachten und daraus Schlüsse ziehen. Die Interna des Netzwerks werden transparent. Anhand von bekannten und interessanten Problemen kann sich so auch ein unerfahrener Nutzer die Funktionsweise neuronaler Netze erschließen.

Die Implementierung ist dabei auf eine Weise gewählt, die Nutzern zwar möglichst große Freiheit bei der Manipulation der Netze ermöglicht, bestimmte Detailaspekte allerdings auch streng vorgibt. Dementsprechend kommen eher gängige Netzwerkmodelle zum Einsatz. Diese sind gegebenfalls nicht für jeden möglichen Anwendungsfall ausreichend. Auch die grafische Oberfläche soll schlicht gehalten sein, um Benutzer nicht zu überfordern oder abzuschrecken. Für diesen Punkt ist eine ansprechende Gestaltung wichtig. Das bedeutet unter anderem, dass Eingaben intuitiv getätigt werden können und nicht erst viel Zeit zum Erlernen der Anwendung aufgewendet werden muss.

## 1.4 Motivation

Die Forschung an künstlichen neuronalen Netzen wurde in der Vergangenheit mehrfach als Sackgasse bezeichnet. Es wurde angenommen, dass das Modell nicht dafür geeignet sei,

Probleme einer gewissen Komplexität abbilden oder lösen zu können. Neue Impulse bei der Struktur und der Verarbeitung haben allerdings immer wieder dafür gesorgt, dass das Modell erweitert werden konnte, um eine größere Menge an Problemen abzudecken. Aktuell steht das Thema unter dem Modewort *Deep Learning* wieder im Fokus (Abb.1.2). Es häufen sich Nachrichten über Anwendungsgebiete, in denen damit deutlich bessere Ergebnisse erzielt werden als mit traditionellen Algorithmen.

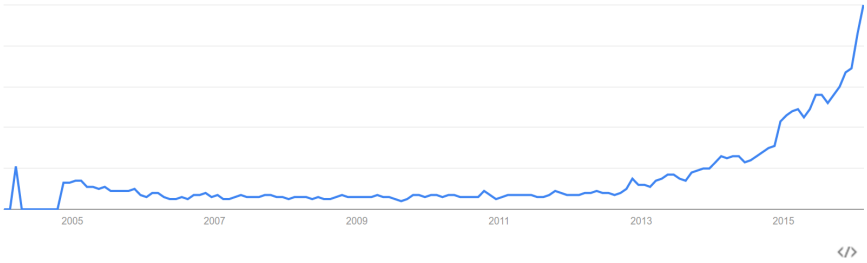



**Abbildung 1.2:** Google Trends zum Begriff *Deep Learning*. Relative Zahlen im Bezug auf den Suchzeitraum von 2004 bis heute.

*Georg Hotz*, der damit bekannt wurde, Sicherheitsmechanismen von mobilen Betriebssystemen und Spielekonsolen auszuhebeln, sorgte 2015 für Schlagzeilen, indem er den großen Automobilhersteller Tesla mit einem selbstentwickelten autonomen Fahrsystem herausforderte [Van15]. Dieses hatte er in seiner Garage entwickelt, indem er sich in die Elektronik eines Autos einklinkte. Er konnte so Kameraaufnahmen der Straße mit seinen Steuerkommandos als Fahrer verknüpfen. Mit Hilfe dieser Daten war er in der Lage, neuronale Netzwerke so zu trainieren, dass sie seine Reaktionen auf die Umgebung und damit seine Fahrweise simulieren.

Ein anderes Beispiel ist das Google-Unternehmen *DeepMind*, das sich auf die Entwicklung künstlicher Intelligenzen spezialisiert hat. Dieses meldete Anfang 2016, dass es ihrem Programm *AlphaGo* gelungen sei, den europäischen Meister *Fan Hui* im Brettspiel Go in fünf aufeinanderfolgenden Partien zu schlagen [Gib16]. Bisherige Programme erreichten gerade einmal das Niveau eines Anfängers. Go gilt durch die Vielzahl der möglichen Spielzüge als deutlich komplexer als Schach.

Auch *DeepMind* ist der Durchbruch mit Hilfe von neuronalen Netzwerken gelungen. Erst wurde ein Netz mit Millionen von gesammelten Spieldaten trainiert, um dann wiederholt gegen sich selbst anzutreten. So konnte das Netz erfolgreiche Spielzüge für Brettpositionen erlernen und ist in der Lage, aus den durchschnittlich 200 Möglichkeiten pro Zug die besten herauszufiltern. Auf dieser Basis kann dann mit Hilfe eines klassischen Entscheidungsfindungsalgorithmus, der *Monte Carlo tree search*, der erfolgversprechendste

Zug ausgewählt werden.

Neuronale Netze haben sich offensichtlich in der Praxis etabliert und werden dort eingesetzt, wo eine algorithmische Abarbeitung von Daten an technische Grenzen stößt. Die Motivation für diese Arbeit besteht darin, einen Überblick über die Grundlagen des Themas zusammenzustellen. Durch die Realisierung der interaktiven Anwendung soll dieses Wissen einerseits gefestigt, andererseits auch für andere zugänglich gemacht werden. Außerdem soll die Frage, für welche Zwecke sich neuronale Netze anbietet, geklärt werden. So entsteht eine Einordnung des Werkzeugs in den Themenkomplex der Softwareentwicklung. Lesern und Anwendern dieser Arbeit soll so ein Großteil des Rechercheaufwands abgenommen und die Grundlage dafür geschaffen werden, dass sie neuronale Netze in eigenen Projekten einsetzen können.

Die Relevanz dafür scheint durchaus gegeben. In den letzten Jahren stellten immer mehr große Akteure im Bereich Informationstechnologie *Frameworks* bereit, die die Arbeit mit künstlichen neuronalen Netzen erleichtern. Beispiele hierfür sind *Googles TensorFlow* oder *Microsofts Project Oxford*, die beide das Arbeiten im Bereich *Maschinelles Lernen* vereinfachen, indem sie eine abstrahierte Schnittstelle anbieten [Mic16, Goo16]. Diese Arbeit soll auch einen Grundstein dafür legen, überhaupt verstehen zu können, was diese Werkzeuge leisten und wie sie eingesetzt werden können.

## 1.5 Abgrenzung

Diese Arbeit hat nicht den Anspruch, eine vollständige Simulation neuronaler Netze und deren kompletter Möglichkeiten zu sein. Es gibt Software, auch frei verfügbare, die diesen Anspruch verfolgt und besser umsetzt als es eine Arbeit mit diesem Umfang könnte. Der Anspruch besteht darin, einen einfachen Einstieg in die Thematik mit Hilfe einer simplifizierten Simulation zu ermöglichen. Daraus ergeben sich Anforderungen an das Ergebnis, die es von den gängigen Umsetzungen unterscheidet.

Normalerweise werden Bibliotheken so implementiert, dass sie interne Abläufe möglichst verbergen. Das hat in der Praxis viele Vorteile wie Benutzbarkeit und Effizienz, hilft beim Verstehen aber nur bedingt. Daher ist ein Teilziel dieser Arbeit die Implementierung eines neuronalen Netzwerks, die nicht nur dessen Funktionalitäten abbildet, sondern den Ablauf der internen Algorithmen einsehbar macht. Die Implementierung ist zwar von anderen Bibliotheken inspiriert, muss aber an die speziellen Anforderungen angepasst und daher neu entwickelt werden.

Als Beispiel für eine solche Software sei hier *OpenNN (Open Neural Networks Library)* [Art16] genannt. Die freie Bibliothek implementiert neuronale Netzwerke mit dem

Ziel, gängige Aufgaben aus dem Bereich *Maschinelles Lernen* bearbeiten zu können. Sie ist in C++ geschrieben und legt den Schwerpunkt auf Effizienz. Von der gleichen Firma wurde außerdem die grafische Oberfläche *Neural Designer* entwickelt, die auf der Programmierschnittstelle von *OpenNN* aufbaut. Mit dieser ist es möglich die Bibliothek auch ohne Programmierkenntnisse einzusetzen.

Die meisten Simulationen von neuronalen Netzen bieten eine Möglichkeit der Visualisierung. Dabei geht es allerdings immer um deren Aufbau und die Vernetzung der einzelnen Komponenten. Die reine Visualisierung eines trainierten Netzes hilft jedoch nicht beim Verständnis der Funktionsweise. Ein Projekt, das auch die elementare Phase des Trainings abbildet, konnte nicht gefunden werden.

# 2 künstliche neuronale Netzwerke

Dieses Kapitel beschäftigt sich mit den theoretischen Grundlagen von neuronalen Netzen. Zuerst wird ihr Modell und genereller Aufbau anhand der beteiligten Komponenten beschrieben. Danach wird erklärt, welche Auswirkungen verschiedene Konfigurationen auf die Eigenschaften eines Netzes haben. Schließlich geht es darum, wie einem Netz konkrete Funktionen mit Hilfe von maschinellen Lernverfahren beigebracht werden können. Das Kapitel legt dabei den Fokus auf eine Erläuterung der grundlegenden Mechanismen von neuronalen Netzen und stellt keinen Anspruch auf Vollständigkeit. Netzwerktypen, die stark vom Aufbau des *Perzeptrons* abweichen, werden nicht behandelt.

## 2.1 Grundlagen

Computer sind dazu optimiert, arithmetische Berechnungen möglichst effizient durchzuführen. Durch Algorithmen, also strikte Vorgaben zur Abarbeitung von Aufgaben, ist es möglich, aufwändige Informationsverarbeitung zu automatisieren. Strukturierte Daten können so in einem Bruchteil der Zeit, die ein Mensch dafür benötigen würde, fehlerfrei verarbeitet werden.

Sobald es aber darum geht, mit unbekannten oder fehlerhaften Informationen umzugehen, stößt diese klassische Vorgehensweise an ihre Grenzen. Menschen können beispielsweise auch dann noch einem Gespräch folgen, wenn Umgebungsgeräusche dafür sorgen, dass Satzfragmente unverständlich werden. Bei Maschinen führt eine fehlerhafte Eingabe zu unerwünschten Ausgaben oder zum Abbruch der Verarbeitung.

Das menschliche Gehirn ist offensichtlich in der Lage, auf analoge und damit fehleranfällige Informationen dynamisch zu reagieren. Auch kann es unbekannte Informationen zumindest thematisch einordnen. Künstliche neuronale Netzwerke sind ein Versuch, die dafür nötigen biologischen Abläufe und Strukturen als Modell zu beschreiben. Die digitale Umsetzung dieses Modells soll auch Computern ermöglichen, Informationen zu verarbeiten, für die vorab keine konkreten Verarbeitungsvorschriften durch einen Entwickler festgelegt wurden.

### 2.1.1 biologisches Vorbild

Für die Erstellung eines Modells muss die reale Vorlage nachvollzogen werden. In der Neurobiologie wurden zwar schon viele Abläufe des Gehirns erforscht, ein vollständiges Verständnis der Funktionsweise gibt es aber noch nicht. Glücklicherweise reduziert sich die abzubildende Komplexität, weil viele der Prozesse, wie die chemische Erzeugung elektrischer Impulse, für die maschinelle Verarbeitung nicht relevant sind. Das Grundmodell neuronaler Netze hat auch nicht den Anspruch, das komplette Gehirn zu simulieren, sondern macht sich eher dessen elementare Prinzipien zunutze.

Die Funktion eines Nervensystems wie dem Gehirn basiert auf dem Zusammenwirken einer Vielzahl von Nervenzellen, die *Neuronen* genannt werden. Deren Interaktion sorgt dafür, dass wir unsere Umwelt erfassen, verarbeiten und darauf reagieren können. Neuronen nehmen über ein fein verästeltes System von Fortsätzen, den *Dendriten*, Erregungen aus anderen Zellen auf. Das können beispielsweise Muskel–, Sinnes– oder andere Nervenzellen sein. Die Verbindung zwischen Dendrit und einer anderer Zelle ist die *Synapse*. Die über Synapsen empfangenen Erregungen werden im Zellkern zusammengefasst und zu einem eigenen neuen Impuls weiterverarbeitet.

Diese Erregung wird über einen sich verzweigenden Informationsleiter, das *Axon*, an andere Zellen weitergegeben. Auf diesem einfachen Prinzip basiert die komplette Informationsverarbeitung des Gehirns. Die Basis in Form von Eingabe, Verarbeitung und Ausgabe entspricht passenderweise auch dem gleichnamigen Konzept *input*, *process*, *output* der Datenverarbeitung in der Softwareentwicklung [Wik16a].

In Anlehnung an das biologische Vorbild wird auch die kleinste Einheit in künstlichen Netzen *Neuron* genannt. Zwischen den Neuronen bestehen Verbindungen, deren Funktion auf den Synapsen des Vorbilds basiert. Diese sind gewichtet und leiten Impulse entsprechend verstärkend oder abschwächend weiter. Diese Gewichte sind für die Flexibilität neuronaler Netze von entscheidender Bedeutung. Jedes Neuron verarbeitet die gewichteten Impulse mit Hilfe mathematischer Funktionen zu einem neuen Signal, das wiederum von nachfolgenden Neuronen empfangen wird. Ein Signal bewegt sich auf diese Weise durch das Netzwerk, bis ein Neuron ohne Nachfolger den Wert ausgibt. Jede einzelne Einheit ist für sich genommen nicht für komplexe Aufgaben geeignet. Die Vielseitigkeit des Systems ergibt sich erst aus der Kombination vieler Neuronen.

### 2.1.2 Topologie

Die Anordnung der Neuronen untereinander wird *Topologie* genannt. Sie und die Ausprägung der Gewichte bestimmen, welchen Zweck ein konkretes Netzwerk erfüllt.

Die Forschung hofft beispielsweise mit dem Zusammenschluss vieler Millionen Neuronen, komplexe Denkmuster abbilden zu können. Allerdings ist es schon mit einem einzelnen Neuron möglich, einfache mathematische Funktionen zu realisieren. Die Topologie des Netzwerks muss also passend zum Einsatzzweck gewählt werden. Prinzipiell entspricht der Aufbau eines neuronalen Netzwerks dem eines gewichteten Graphen und kann ebenso viele Formen annehmen. Es haben sich allerdings zweckgebundene Strukturen etabliert, denen die meisten Netze folgen.

Im Gehirn sind Neuronen als *rekurrentes Netzwerk* (Abb. 2.1) angeordnet. Im Gegensatz zu sogenannten *Feedforward Netzwerken* (Abb. 2.2) haben die erzeugten Impulse auch Einfluss auf Neuronen, die ihrerseits das ursprüngliche Neuron beeinflussen. So entstehen zirkuläre Beziehungen. In Feedforward-Netzwerken werden dagegen Signale immer nur in eine Richtung weitergegeben.

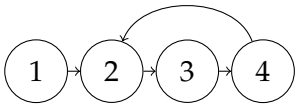



**Abbildung 2.1:** Schematische Darstellung eines rekurrenten Netzwerks

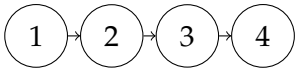



**Abbildung 2.2:** Schematische Darstellung eines Feedforward Netzwerks

Rekurrente Netze werden generell als leistungsfähiger angesehen und können beispielsweise auch zeitabhängige Informationen verarbeiten. Sie sind aufgrund ihrer erhöhten Komplexität aber auch schwieriger zu trainieren und zu beschreiben und werden hier deshalb nicht weiter behandelt.

In der Praxis werden die Neuronen meistens in Schichten organisiert. Die Menge an Neuronen, die die initialen Reize einbringt, wird als *Eingabeschicht* bezeichnet. Entsprechend gehören die Neuronen, die das Ergebnis präsentieren, zur *Ausgabeschicht*. Dazwischen kann es eine beliebige Anzahl von *verdeckten Schichten* geben, die für die äußere Betrachtung des Netzwerks nicht relevant sind. Dieser Aufbau ist in Abbildung 2.3 dargestellt.

Die Neuronen einer Schicht sind dabei immer nur mit denen der nächsten Schicht verbunden und nicht untereinander. Netze mit dieser Art von Topologie werden *Perzeptron* genannt und sind mittlerweile gut erforscht und beschrieben.

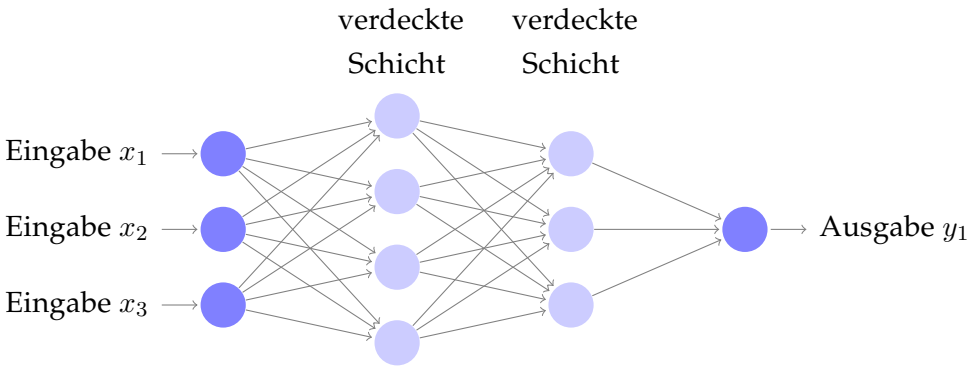



**Abbildung 2.3:** Aufbau der Schichten in einem neuronalen Netzwerk

### 2.1.3 Perzeptron

Der Begriff *Perzeptron* beschreibt nicht nur eine konkrete Netztopologie, sondern wird für eine ganze Klasse von Modellen neuronaler Netze verwendet. Das Konzept dazu wurde 1958 von *Frank Rosenblatt* veröffentlicht und später von *Minsky* und *Papert* in ihrem Buch *Perceptrons* analysiert und mathematisch definiert [Zel97, S. 97]. Dieses ursprüngliche Modell des Perzeptrons beschreibt mehrschichtige Netzwerke, in denen nur die Gewichte der Ausgabeschicht modifizierbar sind. Nach heutiger Definition ist damit effektiv ein einlagiges Perzeptron beschrieben.

#### einlagiges Perzeptron

Um die Funktionsweise von komplexen Netzwerken zu verstehen, ist es sinnvoll zuerst einfache Topologien und deren Eigenschaften zu betrachten. Der Aufbau von einlagigen Perzeptronen lässt sich aufgrund ihrer geringen Komplexität gut dazu heranziehen.

Einlagigkeit bedeutet, dass die Eingabeneuronen direkt mit den Ausgabeneuronen verbunden sind und sich damit nur eine Lage an Gewichten ergibt. Es gibt also keine verdeckten Schichten. Da die Neuronen in der Ausgangsschicht nicht untereinander interagieren, kann so ein Netzwerk mit $m$ Ausgangsneuronen zur besseren Übersicht auch als $m$ verschiedene Perzeptronen mit jeweils nur einem Ausgangsneuron betrachtet werden. Die Ergebnisse $y_i$ der Teilperzeptronen werden dann später zur Ausgabe des gesamten Perzeptrons zusammengefasst (Abb. 2.4).

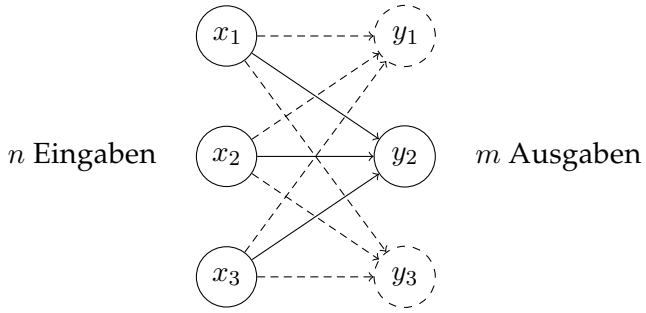



**Abbildung 2.4:** Spaltung eines Perzeptrons mit mehreren Ausgängen in mehrere Perzeptronen mit jeweils einem Ausgang

Ein solches Teilperzeptron besteht also aus einem Ausgabeneuron und mindestens einem Eingabeneuron. Die Eingabeneuronen sind nicht an der Verarbeitung von Informationen beteiligt, sondern stellen diese nur dem restlichen Netzwerk bereit. Die Berechnung wird durch das Ausgabeneuron durchgeführt, indem die Summe der gewichteten Eingaben $u = \sum_i w_i x_i$ an eine *Aktivierungsfunktion* $\varphi$ übergeben wird (Gleichung 2.1). Jede Eingabe wird also mit dem Gewicht ihrer Verbindung multipliziert und alle so erzeugten Werte vom Ausgabeneuron aufsummiert und weiterverarbeitet (Abb. 2.5.

$$y = \varphi\left(\sum_i w_i x_i\right) \tag{2.1}$$

In der Regel werden die Eingaben $x_1, \ldots, x_n$ und die Gewichte $w_1, \ldots, w_n$ als Vektoren $x$ und $w$ zusammengefasst. Dadurch kann die Berechnung von $u$ in ihrer Darstellung vereinfacht werden (Gleichung 2.2).

$$u = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x} \tag{2.2}$$

In älteren Modellen waren noch andere Funktionen an der Verarbeitung beteiligt, die allerdings mittlerweile zur Aktivierungsfunktion zusammengefasst wurden. Das Ergebnis $y = \varphi(u)$ beschreibt die Ausgabe des einzelnen Neurons und somit die $i$-te Ausgabe $y_i$ des gesamten Perzeptrons. Das komplette Perzeptron bildet somit $n$ Eingabewerte auf $m$ Ausgabewerte ab.

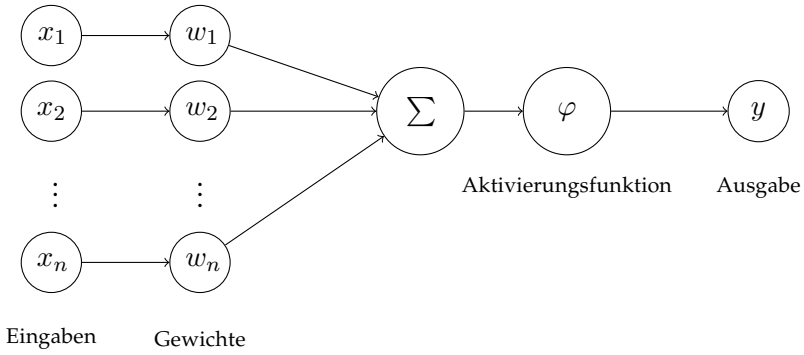



**Abbildung 2.5:** Aufbau eines einlagigen Perzeptons mit einem Ausgabeneuron

### Klassifizierung

Ein häufiger Verwendungszweck von Perzeptronen ist die *Klassifizierung*, also die Zuordnung von Eingaben zu einer von mehreren Kategorien. Diese Funktion ergibt sich direkt aus der Beobachtung der Prozesse im biologischen Vorbild. Dort sendet ein Neuron immer nur dann ein Signal, wenn ein bestimmtes Schwellenpotential durch eingehende Impulse überschritten wird. Um dieses Verhalten zu simulieren, wird auch bei künstlichen neuronalen Netzwerken eine *Schwellenwertfunktion* (auch Heaviside– oder Stufenfunktion) (Gleichung 2.3) als Aktivierungsfunktion $\varphi(u)$ eingesetzt. Der summierte Wert $u$ muss eine Schwelle $\theta$ überschreiten, um ein Signal zu erzeugen (Abb. 2.6). Solche Neuronen werden *linear threshold unit (LTU)* genannt. Sie erzeugen per Definition nur binäre Ausgaben und liefern keine kontinuierlichen Werte.

$$f(u) = \begin{cases} 0 & u \leq \theta \\ 1 & u > \theta \end{cases} \tag{2.3}$$

Dadurch ist es dem Neuron möglich zu entscheiden, ob die Eingaben in ihrer Kombination bestimmte Eigenschaften haben und sie so zu klassifizieren.

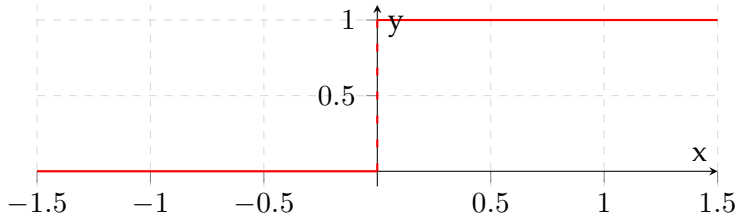



**Abbildung 2.6:** Verlauf einer Schwellenwertfunktion mit 0 als Schwelle $\theta$

Mathematisch begründet sich das Verhalten wie folgt. Eingaben repräsentieren einen

Punkt im $n$-dimensionalen Raum, wobei $n$ der Anzahl der Eingaben entspricht. Alle Eingabevektoren, für die gilt, dass $u = \theta$, beschreiben in diesem Raum eine *Hyperebene*, die ihn in zwei Bereiche trennt. Hyperebenen sind Objekte mit einer Dimension weniger als der Raum, in dem sie sich befinden. Beispielsweise trennt eine Linie den zweidimensionalen Raum.

Gegeben sei ein Perzeptron mit zwei Eingabeneuronen, den Gewichten $\binom{1}{1}$ und einer Schwellenwertfunktion mit $\theta = 0$. Der Schwellenwert wird nur unterschritten, wenn mindestens eine Eingabe negativ ist und sie nicht durch die andere ausgeglichen wird. Dadurch wird eine Linie, die Hyperebene, durch $P_1 = (-1/1)$ und $P_2 = (1/-1)$ definiert (Abb. 2.7).

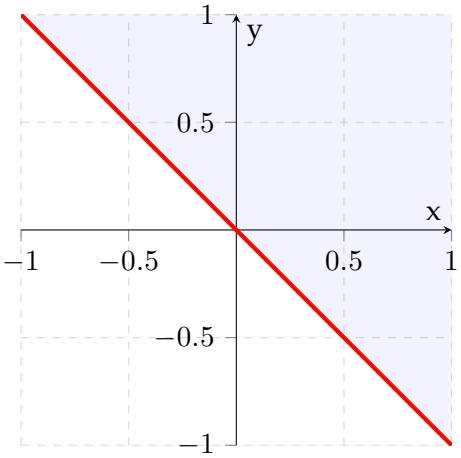



**Abbildung 2.7:** Die Hyperebene trennt die Fläche in zwei Klassen. Alle Eingabevektoren innerhalb des blauen Bereichs erzeugen einen Impuls.

Alle Eingabewerte, die auf der einen Seite dieser Linie liegen, ergeben einen Reiz in Form einer $1$, alle anderen eine $0$. Eingaben werden so einer der beiden Kategorien zugeordnet, also klassifiziert.

Es ist möglich, die Hyperebene durch Anpassungen des Schwellenwerts zu verschieben. Beispielsweise kann die Konjunktion aus der Logik (AND-Verknüpfung) abgebildet werden, indem bei zwei binären Eingaben $x_i \in \{0, 1\}$ ein $\theta$ von $1,5$ angenommen wird und so nur eine $1$ ausgegeben wird, wenn beide Eingaben $1$ sind (Abb 2.8). Alle anderen Eingaben erzeugen eine $0$. Somit entspricht die durch das Perzeptron abgebildete Funktion der Wahrheitstabelle der Konjunktion.

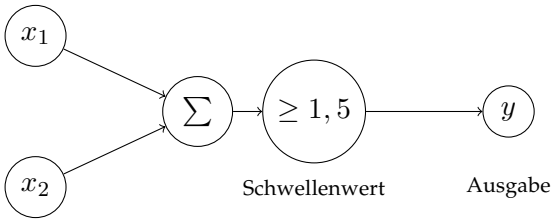



**Abbildung 2.8:** Umsetzung einer AND-Verknüpfung als einlagiges Perzeptron durch Schwellenwertänderung

In der Praxis ist es allerdings üblich, die Aktivierungsfunktionen für das komplette Netzwerk vorab festzulegen und während der Verarbeitung nicht mehr zu verändern. Um die Funktionalität des Netzwerks trotzdem flexibel zu halten, werden die Gewichte zwischen den einzelnen Neuronen verändert. Die Hyperebene aus dem vorherigen Beispiel kann also beeinflusst werden, ohne Schwellenwerte ändern zu müssen.

Unter der Annahme der Gewichte $\binom{0,25}{-1}$ muss der Eingangswert des ersten Neurons bei über $4$ liegen, wenn beim zweiten eine $1$ anliegt, um den Schwellenwert von $0$ zu übersteigen. Die Änderung der Gewichte hat die Steigung der Trennlinie verändert (Abb. 2.9). Die Ausrichtung der Hyperebene lässt sich also durch Anpassungen der Gewichte abwandeln. Allgemein gilt, dass diese Ebene immer senkrecht auf dem Vektor der Gewichte $w$ steht [Ree98, S. 15-17].

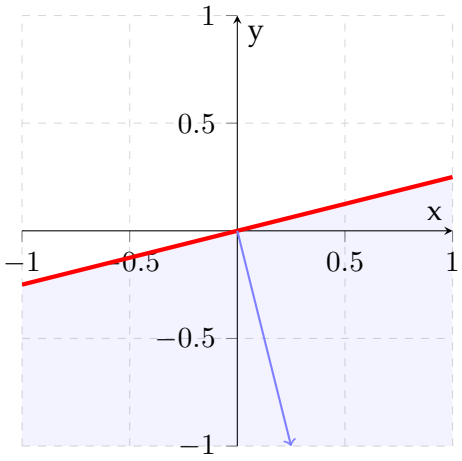



**Abbildung 2.9:** Die Hyperebene trennt die Fläche senkrecht zum blauen Vektor $w$ in zwei Klassen. Dieses Mal erzeugen alle Eingaben unterhalb der Linie Impulse.

### Bias-Neuronen

Es fällt auf, dass sich so nur Hyperebenen darstellen lassen, die durch den Ursprung des $n$-dimensionalen Raums verlaufen. Es ist zwar möglich deren Steigung zu verändern, jedoch

erzeugt die Eingabe des Nullvektors, unabhängig von den Eingabegewichten, immer eine $0$ in der Ausgabe. Um diesen Effekt zu umgehen, wird jedem Neuron ein weiterer Eingang hinzugefügt, der *Bias–Neuron (bias)* genannt wird und immer den Wert $1$ liefert. Sein Gewicht ist ebenso variabel, wie die der anderen Eingänge. Zur Summe $u$ wird also zusätzlich das Bias–Gewicht addiert. (Gleichung 2.4) Die Darstellung wurde hier allerdings nur zur Veranschaulichung gewählt. Aus praktischen Gründen wird nämlich nicht zwischen dem Bias-Neuron und den anderen Eingängen unterschieden.

$$u = \sum_i w_i x_i + w_{bias} \tag{2.4}$$

Das zusätzliche Neuron hat den selben Effekt, als würde der Schwellenwert um den negativen Wert des Gewichts verschoben (Gleichung 2.5). Die resultierende Hyperebene entfernt sich dadurch vom Ursprung. Es ist also möglich, durch Veränderung der Gewichte ein Perzeptron zu konstruieren, das zwei beliebige linear trennbare Mengen unterscheidet. Welcher der beiden Mengen eine Eingabe zugeordnet wird, drückt der generierte Ausgabewert aus.

$$\theta = -w_{bias} \tag{2.5}$$

**weitere Aktivierungsfunktionen**

Solange nur binäre Ausgaben gefordert sind, reichen Schwellenwertfunktionen für die Klassifizierung von Eingaben aus. In manchen Fällen werden allerdings kontinuierliche Werte benötigt. Beispielsweise kann es für nachfolgende Verarbeitungsschritte relevant sein, ob eine Eingabe eindeutig zugeordnet werden konnte, oder eher nahe der Hyperebene lag. Man spricht dann von der Wahrscheinlichkeit, dass eine Eingabe zu einer Klasse gehört.

Eine Möglichkeit ist es, stückweise lineare Funktionen einzusetzen. Diese begrenzen den resultierenden Wertebereich weiterhin auf ein Intervall, z.B. $[0, 1]$, ermöglichen aber kontinuierliche Werte. Summen $u$, die nahe am Schwellenwert liegen, erzeugen somit Zwischenwerte, die ausdrücken, dass sie im Grenzbereich des Raums liegen (Abb. 2.10).

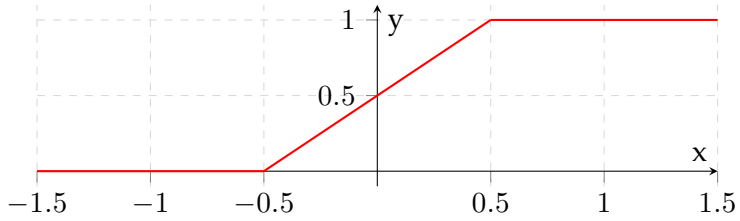
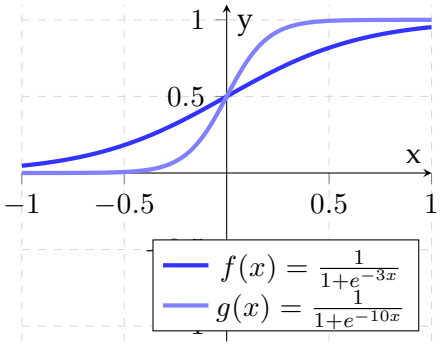
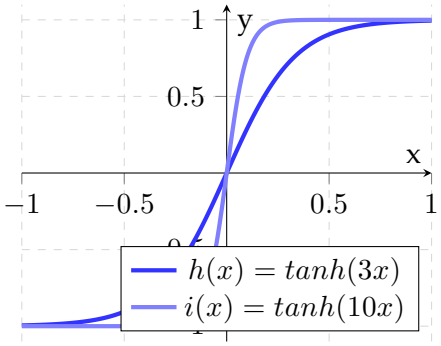



**Abbildung 2.10:** Verlauf einer stückweise linearen Funktion

In der Praxis kommen meistens *sigmoide* Funktionen zum Einsatz, die nach ihrer Ähnlichkeit zum Buchstaben $S$ (Abb. 2.11) benannt sind. Auch durch sie lässt sich der Wertebereich begrenzen. Im Gegensatz zu stückweise linearen Funktionen sind sie aber differenzierbar und damit besser für spätere Automatisierungen geeignet. Über Parameter lässt sich steuern, ob sie eher flach anlaufen oder einer steilen Schwellenwertfunktion gleichen. Im Ergebnis lässt sich so die Größe des Grenzbereichs um die Hyperebene kontrollieren. Damit wird bestimmt, wie strikt Eingaben klassifiziert werden.




**(a)** logistische Aktivierungsfunktion




**(b)** tanh Aktivierungsfunktion

**Abbildung 2.11**

### Regression

Neben Klassifizierungsproblemen, bei denen Eingaben einer bestimmten Gruppe zugeordnet werden sollen, gibt es noch Regressionsprobleme. Bei diesen geht es darum, Zusammenhänge zwischen Ein- und Ausgabe, ähnlich einer Funktion, abzubilden. Hierzu wird die Identität als Aktivierungsfunktion gewählt. Da die Ausgabe dabei nicht auf ein Intervall begrenzt ist, können Funktionszusammenhänge direkt abgebildet werden. Ein Neuron mit nur einem Eingang kann die Steigung der erzeugten linearen Funktion durch das einzige Gewicht verändern. Beispielsweise erzeugt ein Neuron mit dem Gewicht 3 die Funktion $f(x) = 3x$. Mit Hilfe von Bias–Neuronen kann auch der Verlauf von Regressionsfunktionen vom Ursprung verschoben werden.

Ein ähnliches Verhalten lässt sich auch als Klassifizierung abbilden. Die Hyperebene soll in diesem Fall nicht den Raum in zwei Hälften unterteilen, sondern ihr nahe gelegene Eingaben durch hohe Ausgabewerte hervorheben. Statt der schwellenwertähnlichen Funktionen kommt die Gaußsche Normalverteilung zum Einsatz. Diese sorgt dafür, dass Eingaben, die in der Nähe der Hyperebene liegen, starke Reize erzeugen.

Angenommen, ein Neuron mit zwei Eingängen und den Gewichten $\binom{1}{1}$ empfängt den Eingangsvektor $\vec{x_1} = \binom{-1}{1}$. Dadurch, dass die Summe $u = 0$ ist, ergibt die Aktivierungsfunktion einen hohen Wert. Eine Eingabe von $\vec{x_2} = \binom{1}{1}$ und damit $u = 2$ erzeugt einen Wert nahe $0$. In der Praxis wird diese Methode allerdings nur selten eingesetzt, weil die Normalverteilung mathematisch nicht für eine spätere Automatisierung des Netzwerks geeignet ist.

**Einschränkungen**

Einlagige Perzeptronen sind aufgrund ihrer Topologie nur in der Lage, bestimmte Probleme abzubilden. Bei der Klassifizierung können nur Eingaben voneinander getrennt werden, die linear separierbar sind. Alle Datenpunkte einer Klasse müssen dafür durch die Hyperebene von den Punkten der anderen Klassen getrennt sein. Im zweidimensionalen Raum bedeutet das, dass eine Linie zwischen den Datenpunkten gezogen werden kann, die dafür sorgt, dass Punkte unterschiedlicher Klassen auf gegenüberliegenden Seiten der Linie liegen. Auch bei Regressionsproblemen können mit einer Lage von Gewichten nur lineare Lösungen umgesetzt werden.

Als Konsequenz daraus lassen sich schon einfache Funktionen, wie die Kontravalenz aus der Logik (XOR), in einem einlagigen Perzeptron nicht abbilden (Abb. 2.12). Da deren vier Datenpunkte jeweils abwechselnd zu einer anderen Klasse gehören, lässt sich keine Linie finden, die die Klassen voneinander trennt. 1969 sorgte diese Erkenntnis von *Minsky* und *Papert* dafür, dass die Forschung im Bereich künstlicher neuronaler Netzwerke kaum noch finanzielle Unterstützung erhielt. Zwar war damals schon bekannt, dass dieses Problem mit Hilfe von mehrlagigen Perzeptronen zu lösen ist, allerdings war deren Einsatz noch nicht genügend erforscht.

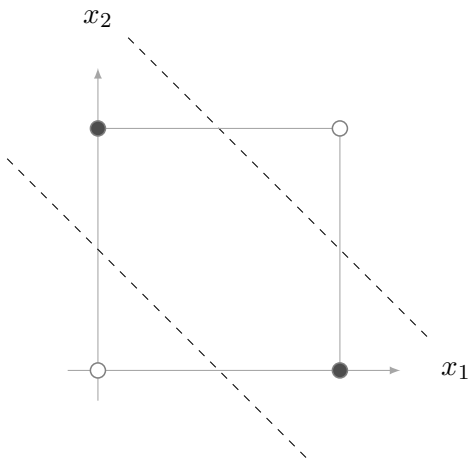



**Abbildung 2.12:** Die Datenpunkte der Kontravalenz (XOR) können nicht linear getrennt werden

### 2.1.4 mehrlagiges Perzeptron

Ein *mehrlagiges Perzeptron (MLP)* besteht aus einer Abfolge von einlagigen Perzeptronen, bei denen die Ausgaben eines Perzeptrons jeweils die Eingaben des Perzeptrons der nächsten Schicht sind. Die Bezeichnung eines Perzeptrons als $n$-lagig oder -schichtig bezieht sich bei den meisten Autoren auf die Anzahl der aktiven Schichten. Eine Schicht ist dann aktiv, wenn sie mit Hilfe einer Aktivierungsfunktion einen neuen Wert generiert. Die Eingabeschicht wird demzufolge nicht zu den aktiven Schichten gezählt. Ein Perzeptron mit Eingabe-, Ausgabe- und einer verdeckten Schicht wird entsprechend als zweischichtig bezeichnet, obwohl drei Schichten beteiligt sind.

Mehrlagige Perzeptons kommen zum Einsatz, wenn einlagige Perzeptronen nicht in der Lage sind, eine gegebene Funktion abzubilden. Sie können sowohl bei Klassifizierungs- als auch bei Regressionsproblemen nichtlineare Zusammenhänge modellieren.

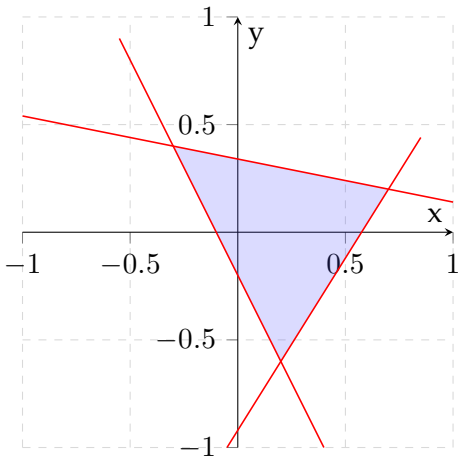



**Abbildung 2.13:** Klassifizierung in Form eines Dreiecks

Soll beispielsweise ein Dreieck vom Rest der Fläche getrennt werden (Abb. 2.13), werden in der ersten Schicht des Perzeptrons drei Hyperebenen definiert, die den Verlängerungen der Seiten des Dreiecks entsprechen. Eingaben lösen nur einen Impuls aus, wenn sie auf der korrekten Seite der Hyperebene liegen. Die zweite Schicht löst darauf aufbauend nur einen Impuls aus, wenn alle drei Neuronen der ersten Schicht eine hohe Aktivierung aufweisen (Abb. 2.14). Entsprechende Eingaben befinden sich dann im Dreieck, also dem Schnittbereich aller Flächen, die durch die erste Schicht definiert wurden.

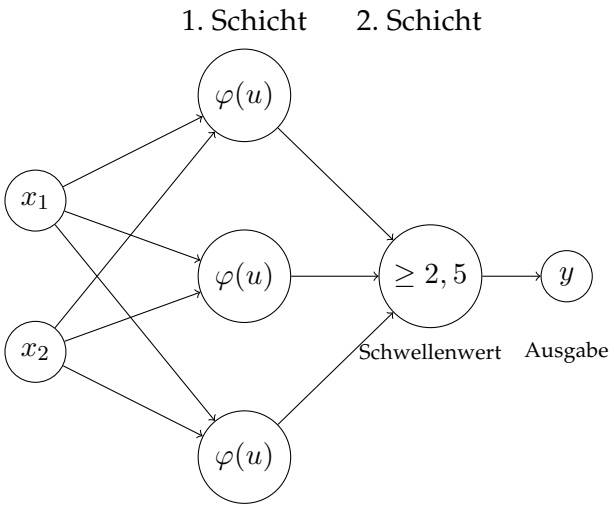



**Abbildung 2.14:** Aufbau eines mehrlagigen Perzeptrons zur nichtlinearen Klassifizierung

Zweilagige Perzeptronen sind offensichtlich in der Lage, konvexe Polygone darzustellen. Es ist auch möglich, konkave und unzusammenhängende Polygone abzubilden. Allerdings geschieht dies über eine Art Raster, auf dem Zeilen und Spalten mehr oder weniger starke

Werte generieren. Es ergibt sich dadurch eine Auflösung, die bei wachsender Anzahl von Neuronen in der verdeckten Schicht genauere Klassifizierungen zulässt. Übersteigt der Wert einer Zelle in diesem Raster den Schwellenwert der Ausgabeschicht, lösen Eingaben einen Impuls aus. Diese Darstellung ist allerdings sehr umständlich und deckt zwar viele, aber nicht alle möglichen Funktionen ab.

Durch eine dritte Schicht an Neuronen wird es möglich, beliebige Vereinigungen oder Differenzen aus den konvexen Polygonen der zweiten Schicht des Perzeptrons zu bilden. So entfällt die umständliche Rasterbildung und Flächen können in beliebiger Genauigkeit definiert werden. Es ist dadurch möglich, jede beliebige Funktion mit Hilfe eines dreilagigen Perzeptrons zu repräsentieren [Ree98, S. 31-38].

#### Overfitting

Größere Netze mit mehr Lagen und Neuronen können komplexere Funktionen abbilden. Es liegt nahe, Netzwerke jedes Mal zu erweitern, wenn eine gewünschte Funktion nicht zur Zufriedenheit abgebildet wird. Dabei muss beachtet werden, dass mehr Neuronen in einem Netz auch immer mehr Freiheitsgrade bedeuten.

Zu viele Freiheiten können dafür sorgen, dass das Netz nicht den generellen Zusammenhang zwischen Eingabe- und Ausgabe, sondern nur konkrete Datenpunkte abbildet. Das Netzwerk ist dann zwar in der Lage, aus bekannten Eingaben perfekte Ergebnisse zu generieren, kann neue Daten aber nicht einordnen. Diese Überanpassung des Netzwerks wird *overfitting* genannt.

## 2.2 Training

Es ist zwar interessant, dass neuronale Netze all die beschriebenen Funktionen umsetzen können, allerdings lassen diese sich mit anderen Mitteln deutlich einfacher realisieren. Das besondere Merkmal neuronaler Netze ist, dass ihre Funktionalität automatisiert angepasst werden kann. Dazu werden die Gewichte der Neuronenverbindungen durch einen Algorithmus solange optimiert, bis das gewünschte Verhalten angenommen wird. Dieser Vorgang wird auch *Training* genannt und ist ein essenzieller Bestandteil maschinellen Lernens.

An dieser Stelle ist die Unterscheidung zwischen zwei Konzepten wichtig. Aufgaben, die ein Netzwerk aufgrund seines Aufbaus abbilden kann, sind von ihm repräsentierbar. Neben der *Repräsentierbarkeit* gibt es die *Lernfähigkeit*. Sie beschreibt die Fähigkeit eines Lernalgorithmus, ein Netzwerk auf die Umsetzung einer repräsentierbaren Funktion zu trainieren. Auch

wenn ein Netzwerk eine Funktion zwar theoretisch abbilden kann, kann es eventuell nicht dahingehend trainiert werden. Für einlagige Perzeptronen konnte Rosenblatt 1962 in seinem Konvergenztheorem [Ros62] beweisen, dass es einen Algorithmus gibt, der einem Netzwerk in endlicher Zeit alle Funktionen beibringen kann, die es repräsentieren kann.

Die Entwicklung eines neuronalen Netzes für eine bestimmte Aufgabe besteht also aus zwei Phasen. Zuerst muss ein Aufbau in Form von Topologie und Aktivierungsfunktionen gewählt werden. Danach werden die Gewichte des Netzwerks über Training an die Aufgabenstellung angepasst. Hierfür werden dem Netzwerk Beispieldaten präsentiert, aus denen es ein bestimmtes Verhalten ableiten soll. Um den Erfolg des Trainings überprüfen zu können und Probleme wie das *overfitting* zu vermeiden, werden meistens Testdaten bereitgestellt. Diese sind eine Teilmenge der Trainingsdaten, die dem Netzwerk allerdings nicht zu Trainingszwecken präsentiert werden. Dadurch können sie dem trainierten Netz als unbekannte Eingabedaten übergeben werden, um die Genauigkeit der ermittelten Funktion zu testen.

Sprachlich wird nicht zwischen Netzwerk und Trainingsverfahren unterschieden, sondern der Lernalgorithmus als Netzwerkkomponente betrachtet. Für das Training gibt es unterschiedliche Methoden, deren Auswahl hauptsächlich darauf beruht, welche Informationen vorab zur Verfügung stehen.

#### Überwachtes Lernen

Beim überwachten Lernen besteht ein Trainingsdatensatz immer aus der Netzeingabe und der dazugehörigen erwarteten Ausgabe. Es werden also Aufgabe und Lösung gleichzeitig präsentiert. Offensichtlich muss dazu zum Trainingszeitpunkt schon eine Lösung, bzw. ein erwartetes Verhalten vorliegen. Das Netzwerk verändert bei Abweichung zwischen erwarteter und tatsächlicher Ausgabe die Gewichte, um diese Differenz zu reduzieren. Es gibt zwar viele verschiedene Trainingsalgorithmen, allerdings basieren die meisten auf den gleichen Grundprinzipien.

1. Dem Netzwerk werden die Eingabedaten an seiner Eingabeschicht präsentiert.
2. Das Netzwerk berechnet auf Basis der aktuellen Gewichte die Ausgabe für diese Eingabe. Dieser Schritt heißt *Propagierung*.
3. Das Delta zwischen der ermittelten und der erwarteten Ausgabe wird berechnet. Dieses wird auch als Fehler bezeichnet.
4. Auf Basis dieses Deltas werden die Gewichte des Netzwerks so verändert, dass der Fehler reduziert wird.

Das Netz soll durch die Wiederholung dieser Schritte mit vielen Datensätzen selbstständig den Zusammenhang zwischen den Ein- und Ausgaben herstellen. Dadurch kann es auch auf unbekannte Daten, die den bekannten Eingaben ähneln, mit dem erwarteten Ergebnis reagieren. Die Fähigkeit, auch neue Daten mit Hilfe des präsentierten Wissens einordnen zu können, nennt sich *Generalisierung*.

Überwachtes Lernen ist am weitesten verbreitet, weil es gut erforscht ist und ein Netzwerk generell schnell für eine Aufgabe trainiert. Problematisch ist allerdings, dass Ein- und Ausgaben vom Anwender in einer Art vorgegeben sein müssen, die es dem Lernalgorithmus einfach macht, daraus Gewichte abzuleiten. Je nach Aufgabenstellung kann deshalb eine aufwändige Vorverarbeitung oder eine anwendungsbezogene Wahl des Algorithmus nötig sein.

#### Bestärkendes Lernen

Bestärkendes Lernen entspricht dem überwachten Lernen in den meisten Punkten. Anstatt allerdings konkrete Ausgaben vorzugeben, wird dem Netzwerk nur mitgeteilt, ob das ermittelte Ergebnis erwünscht war oder nicht. Das Netzwerk wird für angestrebtes Verhalten belohnt und im Fehlerfall bestraft. Diese Methode braucht prinzipiell länger, um ein Netzwerk zu trainieren.

Es wird dadurch allerdings auch möglich, abstraktere Verhaltensmuster zu trainieren. Beispielsweise wird die Funktionsweise eines Netzwerks durch Punkte bewertet. Die Gewichte des Netzwerks werden dann durch das Lernverfahren so verändert, dass mehr Punkte bei der Aufgabe erreicht werden. Dabei ist es egal, wie das Netz konkret agiert und welche Ausgaben es erzeugt, solange die Resultate eine Punktsteigerung bewirken. Dieses Vorgehen kann für Einsatzzwecke interessant sein, in denen konkrete Netzwerkausgaben zum Zeitpunkt des Trainings nicht bekannt sind.

#### Unüberwachtes Lernen

Im Gegensatz zu den überwachten Verfahren werden dem Netzwerk beim unüberwachten Lernen keine erwarteten Ergebnisse übermittelt. Für das Training werden nur Eingaben präsentiert. Der Lernalgorithmus sorgt dann dafür, dass ähnliche Eingabemuster der gleichen Klasse von Ausgaben zugeordnet werden. Da es allerdings keine Vorgaben gibt, sucht der Algorithmus selbstständig nach Gemeinsamkeiten. Welche Regelmäßigkeiten erkannt werden können, hängt von der Topologie des Netzwerks und dem Lernalgorithmus ab.

Der größte Vorteil dieses Verfahrens ist es, dass vorab keine Ausgaben des Netzwerks

bekannt sein müssen. So können ihm auch unklassifizierte Daten präsentiert werden. Solche Daten kommen deutlich häufiger vor, als klassifizierte. Handschriftlichen Texten liegen beispielsweise eher selten digitale Informationen zur Einordnung der einzelnen Zeichen bei. Die Ergebnisse eines so trainierten Netzwerks werden teilweise von anderen Netzwerken eingesetzt, damit diese nicht mit den rohen, sondern bereits optimierten Daten umgehen müssen.

### 2.2.1 Lernalgorithmen

Zur Veränderung der Gewichte des neuronalen Netzwerks bedarf es einer Lernregel, die die einzelnen Anpassungen herleitet. An dieser Stelle werden ausschließlich Lernregeln des überwachten Lernens vorgestellt. Diese Verfahren eignen sich besonders gut für die Veranschaulichung der Grundlagen, weil eine direkte Untersuchung von Ein- und Ausgaben stattfinden kann.

#### Delta-Regel

Das Ziel der Lernregel ist es, den Fehler, also die Differenz zwischen tatsächlichem und erwartetem Ergebnis, zu minimieren. Im ersten Schritt muss dazu ein Richtwert definiert werden, der angibt, wie hoch die Abweichung bei den Trainingsdaten ausfällt. Es ist üblich, dafür die *Summe der Fehlerquadrate (sum of squared errors - SSE)* zu verwenden (Gleichung 2.6) [Ree98, S. 50].

$$E_{SSE} = \sum_{p} \sum_{i} (t_{pi} - y_{pi})^2 \tag{2.6}$$

Damit wird für jedes Ausgabeneuron $i$ bei jedem Trainingsdatensatz $p$ das Delta zwischen erwartetem Ergebnis $t_{pi}$ und eigentlicher Ausgabe $y_{pi}$ zu einem Gesamtfehler aufsummiert. Der quadrierte Wert wird verwendet, um den Fehler bei negativen Differenzen nicht wieder zu reduzieren. Für das eigentliche Verfahren macht der höhere Wert keinen Unterschied. Der Zusammenhang zwischen den Gewichten des Netzes und der Fehlerquadratsumme beschreibt in einlagigen Netzen damit eine Funktion $L$ (Gleichung 2.7).

$$L(\mathbf{w}) = \sum_{p} \sum_{i} \left(\varphi(\mathbf{w}^T \mathbf{x}_p) - y_{pi}\right)^2 \tag{2.7}$$

Diese Fehlerfunktion stellt in Abhängigkeit der Gewichte den Gesamtfehler des Netzes dar. Das Minimum der Funktion entspricht also der Konfiguration an Gewichten, die die Zusammenhänge der Trainingsdaten am besten repräsentieren. Für die Minimierung dieses Fehlers wird ein *Gradientenabstiegsverfahren* eingesetzt. Das generelle Prinzip des Verfahrens

ist es, von der aktuellen Position des Netzwerks im Fehlerraum zu prüfen, in welche Richtung sich das Netz, also die Gewichte, verändern muss, um den Fehler zu verringern [Dol13].

In einem Netz mit zwei Gewichten entspricht die Fehlerfunktion einer Oberfläche im dreidimensionalen Raum (Abb. 2.15). Angenommen die Gewichte repräsentieren einen Punkt auf einer Erhebung, so sorgt das Verfahren dafür, dass der Punkt sich Schritt für Schritt den Hang hinab bewegt. Das Verhalten ist vergleichbar mit dem einer Kugel, die auf einer Oberfläche platziert wird und automatisch zu einem tieferen Punkt rollt.

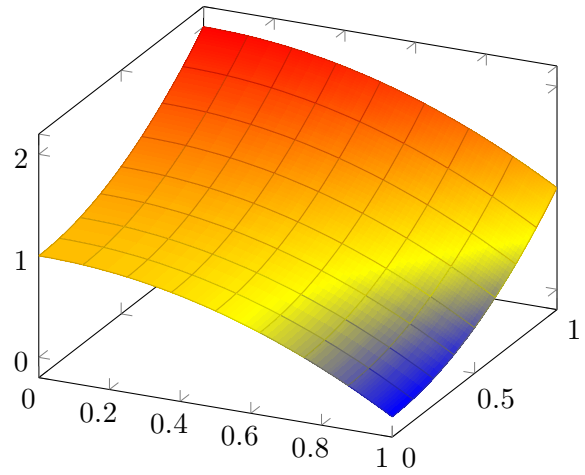



**Abbildung 2.15:** Beispiel einer Fehlerfunktion $L$ im Raum

Es wird hierbei unterschieden, ob die Gewichte nach jedem Datensatz verändert werden, oder erst sobald alle Trainingsdatensätze dem Netzwerk einmal präsentiert wurden. Beim kontinuierlichen Training spricht man von *online learning*. Die Verarbeitung aller Ergebnisse, bei der ein Durchlauf *Epoche* genannt wird, heißt *batch* oder *offline learning*.

Mathematisch beschreibt der *Gradient* einen Vektor mit den Werten aller Ableitungen nach den einzelnen Achsen des aufgespannten Raums. So wird die Steigung in Richtung der Achsen, also der einzelnen Gewichte, errechnet. Je stärker die Steigung an der entsprechenden Stelle, desto weiter bewegt sich das Gewicht in diese Richtung. Verschwindet der Gradient, ist also $\nabla = 0$, wurde ein lokales Minimum erreicht [Tes07, S. 161]. An dieser Stelle wird ein Vorteil von Bias–Neuronen deutlich. Anstatt für jede Aktivierungsfunktion eine andere Ableitung berechnen zu müssen, kann der Algorithmus ohne Anpassung verwendet werden. Außerdem wird klar, warum manche Funktionen nicht als Aktivierungsfunktionen geeignet sind. Da an jeder Stelle der Funktion eine Ableitung existieren muss, müssen Aktivierungsfunktionen differenzierbar sein.

Um die Veränderung der Gewichte zu bestimmen, muss der Gradient negiert werden, weil das Verfahren sonst in der Fehlerfunktion aufsteigen würde. So ergeben sich die Gleichungen

für einzelne Gewichte (Gleichung 2.8) und den gesamten Gewichtsvektor (Gleichung 2.9).

$$\Delta \mathbf{w_{j \to k}} = -\eta \frac{\partial}{\partial w_{j \to k}} L(\mathbf{w}) \tag{2.8}$$

$$\Delta \mathbf{w} = -\eta \nabla L(\mathbf{w}) \tag{2.9}$$

Der *Lernfaktor (Lernrate)* $\eta$ ist dabei ein Parameter, der die Veränderung pro Lernzyklus beschränken soll. Dies geschieht, um in der Fehlerfunktion nur langsam abzusteigen. Zu große Sprünge könnten unter anderem dafür sorgen, dass ein Minimum wiederholt übergangen wird, oder die Gewichte in Extreme laufen. Aufgrund der Abhängigkeit von Netz und Fehlerfunktion lässt sich kein allgemeingültiger Wert für die Lernrate angeben. In der Literatur wird ein Richtwert von $0,05 \leq \eta \leq 0,75$ empfohlen, wobei $\eta = 0,1$ oft als Standard eingesetzt wird [Ree98, S. 77].

Neben dem Lernfaktor kommt bei manchen Trainingsverfahren noch ein *Moment* $\alpha$ hinzu, das die Veränderung der Gewichte des letzten Lernzyklus mit einbezieht. Es wird eingesetzt, um das Lernverfahren zu beschleunigen, indem die Veränderung eine generelle Richtung beibehält und nicht zu stark durch lokale Abweichungen beeinflusst wird. Die Kugel aus der Analogie bekommt so Schwung, um über Unebenheiten hinwegzurollen. Teilweise werden beide Werte während des Trainings dynamisch verändert, um optimal auf die Fehlerfunktion reagieren zu können. Bei der hier behandelten einfachen Delta–Regel kommen diese Techniken allerdings nicht zum Einsatz.

### Backpropagation

Die Delta–Regel leitet die Veränderung der Gewichte direkt aus dem Fehler der Ausgabeschicht ab. Dadurch ist es zwar möglich einlagige Netzwerke zu trainieren, allerdings lässt sich das Vorgehen nicht auf die verdeckten Schichten eines mehrlagigen Netzes übertragen. Es gab deshalb bis zur Mitte der 1980er Jahre nur die Möglichkeit, Netze triviale Aufgaben lernen zu lassen, die mit einer Schicht an Neuronen repräsentierbar sind.

Das *Backpropagation–Verfahren* ist eine Erweiterung der Delta–Regel, die es erlaubt, auch mehrlagige Netze zu trainieren. Die Grundidee dahinter ist es, den Fehler jedes Trainingsdatensatzes rückwärts durch das Netzwerk zu propagieren. Dabei wird allerdings nicht mehr der Gesamtfehler betrachtet, sondern die konkrete Auswirkung jedes Neurons. Das Verfahren gleicht prinzipiell der vorwärtsgerichteten Propagierung der Eingaben. Es läuft in drei Schritten ab:

1. Die Eingaben der Trainingsdatensätze werden durch das Netzwerk propagiert. Für jedes Neuron $j$ werden dabei die gewichtete Summe seiner Eingaben $u_{pj}$ und seine Ausgabe $o_{pj}$ pro Datensatz $p$ gespeichert.
2. Die Fehler $\delta_{pj}$ aus der Ausgangsschicht werden rückwärts durch das Netzwerk verbreitet und für jedes Neuron abgeleitet.
3. Die Gewichte $w_{j\to k}$ werden entsprechend ihres Anteils am Fehler angepasst.

Die Berechnung des Fehlers (Gleichung 2.10) unterscheidet dabei zwischen Neuronen der Ausgangsschicht und der verdeckter Schichten.

$$\delta_{pj} = \begin{cases} f'_j(u_{pj})(t_{pj} - y_{pj}), & \text{falls j in der Ausgangsschicht liegt,} \\ f'_j(u_{pj}) \sum_k \delta_{pk} w_{j\to k}, & \text{falls j in einer verdeckten Schicht liegt.} \end{cases} \tag{2.10}$$

In der Ausgangsschicht findet die Berechnung statt, indem zuerst die Differenz zwischen dem tatsächlichen Ergebnis $t$ und dem erwarteten Ergebnis $y$ ermittelt wird. Diese wird dann mit der Ableitung der Aktivierungsfunktion $f'$ an der Stelle der ursprünglichen gewichteten Eingabesumme $u$ multipliziert. Bei sigmoiden Aktivierungsfunktionen ergibt sich daraus, dass eine unsichere Klassifizierung (bei $u \approx 0$) einen größeren Einfluss auf den Fehler des Neurons $\delta_{pj}$ hat als eine eindeutige.

Da bei Neuronen aus verdeckten Schichten nicht der direkte Fehler des Ausgangs verwendet werden kann, wird die Summe der Fehler der nachfolgenden Neuronen verwendet. Die Berechnung der Fehler baut sich so rekursiv über das Netzwerk auf, ändert sich aber ansonsten nicht. Die Gewichtsveränderung wird dann basierend auf den berechneten Fehlern ermittelt (Gleichung 2.11).

$$\Delta w_{i\to j} = -\frac{\eta}{|P|} \sum_p o_{pi} \delta_{pj} \tag{2.11}$$

Die Veränderung wird hauptsächlich durch den Ausgabewert $o_{pi}$ des aktivierenden $i$ und den Fehlerwert $\delta_{pj}$ des aktivierten Neurons $j$ beeinflusst. Das größte Delta entsteht also bei Verbindungen, die eine starke Aktivierung erfahren, aber keine eindeutige Klassifizierung nach sich ziehen. Schließlich wird der Durchschnitt der Fehlerwerte aller Trainingsdatensätze $P$ gebildet und mit der Lernrate $\eta$ angepasst. Das Verfahren gilt dann als abgeschlossen, wenn die Länge des Gradienten unter einen Grenzwert fällt. Es ist dann nicht mehr zu erwarten, dass die Gewichte sich noch in relevanter Weise verändern. Man spricht davon, dass das Verfahren *konvergiert*.

Auch das Backpropagation–Verfahren kann kontinuierlich, also online, durchgeführt werden. Hierzu müssen die Gewichte lediglich nach jedem Datensatz angepasst werden (Gleichung 2.12).

$$\Delta w_{i \to j} = -\eta o_{pi} \delta_{pj} \tag{2.12}$$

Es existieren generell viele Abwandlungen und Erweiterungen für die Backpropagation. Sie beheben Probleme beim Gradientenabstieg im Fall von speziellen Fehlerfunktionen oder machen das Training von komplexeren Topologien, wie rekurrenten Netzen, möglich. Unter anderem durch diese Flexibilität ist Backpropagation das populärste überwachte Lernverfahren.

Dieser Abschnitt folgt den Ausführungen im Blogartikel [Dol14a] und dem jeweils 5. und 8. Kapitel über Backpropagation in [Zel97] und [Ree98]. Hier wird auch die Herleitung der Formeln zur Berechnung der Fehler und der Gewichtsänderung beschrieben.

### 2.2.2 Merkmale

Ein ursprüngliches Ziel neuronaler Netzwerke ist, wie eingangs beschrieben, Repräsentationen analoger Daten verarbeiten zu können. Die Beispiele dieser Arbeit beschäftigten sich bisher allerdings meistens mit Netzen, die nur wenige Eingangsneuronen bereitstellten. Bei der Verarbeitung von Bildern wäre es allerdings notwendig, mindestens einen Eingang pro Pixel zu verwenden. Je nach Auflösung und Farbkanälen kann das bedeuten, dass Tausende oder Millionen von Neuronen allein für die Aufnahmen der Daten nötig wären. Technisch ist das zwar machbar, bringt aber einige Probleme mit sich.

Neben dem enormen Aufwand, ein solches Netzwerk zu trainieren, entsteht vor allem ein Problem, das *Fluch der Dimensionalität (curse of dimensionality)* genannt wird [Bis95, S. 7-8]. Für die Klassifizierung durch ein Netzwerk müssen genug Trainingsdaten vorhanden sein, damit Bereiche im Raum definiert werden können. Steigt die Anzahl der Eingänge und damit die Anzahl der Dimensionen, gibt es immer mehr Möglichkeiten für potenzielle Punkte im Raum. Die Anzahl der nötigen Trainingsdaten steigt deshalb exponentiell mit den Eingängen. Dazu kommt, dass die räumliche Nähe zweier ähnlicher Datensätze nicht mehr gegeben ist. Bilder können ähnliche Motive zeigen, aber trotzdem durch sehr verstreute Punkte repräsentiert werden.

Um dieses Problem zu umgehen, werden Netzen nicht die umfangreichen Rohdaten präsentiert. Es findet eine Vorverarbeitung statt, die sogenannte *Merkmale (features)* extrahiert, die die ursprünglichen Daten durch deutlich weniger Informationen beschreiben. Teil der Vorverarbeitung ist in den meisten Fällen auch eine Normalisierung, die nur Werte im Intervall $[0, 1]$ zulässt. Bei der Texterkennung wäre ein solches Merkmal beispielsweise das Verhältnis zwischen Höhe und Breite eines Buchstaben. In diesem Beispiel macht das Merkmal das Netz gleichzeitig auch robust gegen Skalierungen. Das Extrahieren von

Merkmalen aus Bildern ist ein Teilgebiet der Bildverarbeitung, für das unter anderem der *Scale-invariant feature transform (SIFT)* Algorithmus zum Einsatz kommt [Wik16b].

Offensichtlich wird zur Unterscheidung von Buchstaben mehr als dieses eine Merkmal benötigt. Trotzdem können mit verhältnismäßig wenig Informationen schon Klassen von Zeichen voneinander getrennt werden, ohne dem Netzwerk das komplette Bild eines Buchstabens präsentieren zu müssen. Die Auswahl von geeigneten Merkmalen und die entsprechende Vorverarbeitung werden *Feature Engineering* genannt und sind für die Leistung eines Netzwerks von entscheidender Bedeutung.

## 2.3 Deep Learning

Feature Engineering kann ein aufwändiger Prozess sein, der viel Erfahrung voraussetzt. Vor allem müssen die Daten von einem Entwickler aufbereitet werden, damit der Computer sie verarbeiten kann. Das *Deep Learning* geht an dieser Stelle einen Schritt weiter. Rohdaten werden sehr umfangreichen Netzen präsentiert, die allerdings nicht zum Ziel haben, direkt eine Klassifizierung durchzuführen. Die Schichten dieser Netze extrahieren Eigenschaften aus den Rohdaten und abstrahieren sie Schicht für Schicht.

Die erste Schicht würde beispielsweise Kanten in einem Bild finden. Die zweite Schicht setzt diese dann zu Formen zusammen, damit nachfolgende Schichten konkrete Objekte, wie Text oder Gesichter, aus diesen Formen erkennen können. Die Ergebnisse eines solchen Netzes dienen dann als Eingabe für die eigentliche Klassifizierung. Im Optimalfall kann die Kombination aus mehreren Netzen so Merkmale aus Rohdaten selbst extrahieren und verarbeiten.

Auf der Basis einer Kombination einfacher Komponenten und Prinzipien bieten künstliche neuronale Netzwerke mittlerweile die Möglichkeit auch komplexe Verhaltensweisen zu simulieren. Auch wenn die Forschung von vielen Zielen in Bezug auf die Umsetzung von echten Nervensystemen noch weit entfernt ist, sollte der Einsatz von neuronalen Netzen generell für Projekte in Erwägung gezogen werden. Besonders Systeme, in denen die Vielseitigkeit der auszuwertenden Daten groß oder die explizite Modellierung eines Problems schwierig sind, eignen sich für die Umsetzung in einem neuronalen Netz.

# 3 Realisierung

Dieses Kapitel beschreibt die Realisierung des praktischen Teils der Arbeit. Erst wird diskutiert, warum welche Konzepte neuronaler Netze für die Umsetzung eingesetzt werden. Dann werden die grobe Architektur mit den eingesetzten Technologien und die Besonderheiten in der Implementierung vorgestellt. Gleichzeitig wird auf die Funktionsweise der entstandenen Anwendung eingegangen. So soll ein grober Überblick über die handwerkliche Vorgehensweise gegeben werden. Weiterführende Details können dem mitgelieferten Quellcode im *src* Verzeichnis entnommen werden. Die Anwendung selbst wurde für die gängigsten Plattformen und Architekturen vorkompiliert und liegt im *bin* Verzeichnis. Nach dem Start stellt die Anwendung ihre Weboberfläche unter der Adresse `http://localhost:8080` bereit.

## 3.1 Auswahl der Netzparameter

Das Ziel des praktischen Teils ist es, eine interaktive Bedienoberfläche zu realisieren, über die auch Benutzer ohne viele Vorkenntnisse neuronale Netze erleben können. Anwender sollen ein Netzwerk konfigurieren und trainieren können, ohne komplizierte Detailentscheidungen treffen zu müssen. Daher werden einige Parameter der Konfiguration vorab strikt festgelegt.

Neben der Reduzierung seiner Optionen soll dem Benutzer der Zugang auch durch eine klare Darstellung vereinfacht werden. Eine ansprechende, übersichtliche Visualisierung beschränkt allerdings die Auswahlmöglichkeiten bei der Netzkonfiguration noch weiter. Netzwerkeingaben mit mehr als drei Merkmalen lassen sich beispielsweise nicht grafisch abbilden, weil die räumlichen Dimensionen dafür nicht ausreichen. Wird stattdessen auf eine nüchterne, tabellarische Eingabeform zurückgegriffen, leidet die Benutzbarkeit. Schon die Umsetzung dreidimensionaler Inhalte sollte vermieden werden, weil sich die Komplexität der Erstellung und der Bedienung gegenüber flachen Designs deutlich erhöht.

Die wichtigste Anforderung an die Konfiguration des Netzwerks ist, dass sie möglichst universell einsetzbar sein soll. Im Idealfall kann sie alle unterschiedlichen Probleme lösen, deren Eingabe über die Bedienoberfläche möglich ist. Im Folgenden wird deshalb für die wichtigsten Netzparameter diskutiert, ob und in welchem Umfang sie über die

Oberfläche steuerbar sein sollen. Außerdem werden nicht steuerbare Optionen mit möglichst allgemeingültigen Parametern vorbelegt.

### Problemstellung

Die Wahl der Netztopologie hängt stark davon ab, welche Probleme das Netz lösen können soll. Deshalb muss zuerst entschieden werden, welche Art von Problemen über die Oberfläche eingegeben werden kann. Zur Auswahl stehen dabei Regressions- und Klassifizierungsprobleme. Beide Problemtypen eignen sich zur Visualisierung in zweidimensionalen Koordinatensystemen. Die Klassifizierung ist allerdings greifbarer als die reine Zuordnung von Zahlenwerten, weil sich Klassen einfacher in der Darstellung kodieren lassen. Im praktischen Teil der Arbeit werden verschiedene Klassen beispielsweise durch Farben kodiert. Auch ist es einfacher für Klassifizierungsprobleme universelle Netzwerke bereitzustellen, weil alle Ein- und Ausgaben immer nur im festen Wertebereich $[0, 1]$ liegen. Die Oberfläche ist deshalb nur darauf ausgelegt, Klassifizierungen zu realisieren.

### Netzwerktopologie

Die Visualisierung vieler Neuronen und Gewichte kann schnell unübersichtlich werden. Neben der Verarbeitungsdauer ist dies ein entscheidender Grund für die Begrenzung der Menge an Schichten und Neuronen, die einem Netzwerk hinzugefügt werden können. Es besteht wegen der zweidimensionalen Darstellung ohnehin eine Limitierung der Eingabeschicht auf zwei Neuronen. Jeder Datensatz setzt sich nämlich aus zwei Merkmalen in Form der x– und y–Koordinate zusammen, denen eine Klasse zugeordnet wird. Die einsetzbaren Klassen und damit die Anzahl der Neuronen der Ausgabeschicht können theoretisch konfigurierbar gehalten werden. Es erleichtert die Darstellung der Netzausgaben allerdings, wenn auch hier nach einem festen Konzept mit wenigen Neuronen vorgegangen wird.

Trotzdem soll der Benutzer die Kontrolle über die Dimensionen des Netzwerks behalten. Deshalb sind Anzahl und Größe der verdeckten Schichten — im Rahmen einer sinnvollen Darstellung — frei wählbar. Für die Repräsentierbarkeit der meisten sich daraus ergebenden Probleme reicht eine einfache Netztopologie aus. Aufgrund seiner Einfachheit in der Struktur und bei der Umsetzung bietet sich das Perzeptron als genereller Netzaufbau an. Durch das klare Schichtenmodell und die begrenzte Anzahl von Verbindungen hat es außerdem den Vorteil, übersichtlich dargestellt werden zu können.

**Aktivierungsfunktionen und Lernverfahren**

Aktivierungsfunktionen und Lernverfahren hängen eng miteinander zusammen. Von den beschriebenen Lernverfahren eignet sich bei mehrlagigen Perzeptronen nur die Backpropagation. Um diese nutzen zu können, muss eine differenzierbare Aktivierungsfunktion gewählt werden. Für Klassifizierungsprobleme kommt daher eine der sigmoiden Funktionen zum Einsatz. Eine Auswahl des Benutzers ist an dieser Stelle nicht zwingend notwendig und wird daher ausgelassen. Beide Aspekte sind außerdem sehr komplex und würden unerfahrene Anwender überfordern. Das Training wird deshalb nur implizit durch die Veränderung der Gewichte dargestellt. Um trotzdem einen Einblick in die internen Abläufe des Lernverfahrens zu geben, werden Informationen wie die aktuelle Gewichtsveränderung und die bisher durchlaufenen Trainingsdaten angegeben.

## 3.2 Architektur

Die Programmierung des praktischen Teils der Arbeit umfasst zwei Bestandteile. Die Implementierung eines künstlichen neuronalen Netzwerks bildet den ersten. Darauf aufbauend stellt eine grafische Oberfläche, mit deren Hilfe das Netzwerk überwacht, verändert und trainiert werden kann, den zweiten Teil dar. Die Komponenten sind unabhängig voneinander umgesetzt worden, weshalb zusätzlich eine Schnittstelle zur Kommunikation realisiert wurde.

### 3.2.1 künstliches neuronales Netzwerk

Das neuronale Netz ist als Bibliothek (Paket) der Programmiersprache *Google Go (golang)* entstanden. Ihr Hauptbestandteil ist die Repräsentation eines Netzwerks in Form der Klasse *MLP (multilayer perceptron)*. Die Initialisierung eines Objekts dieser Klasse entspricht der Erstellung eines neuen Netzwerks. Dieses hat standardmäßig nur eine Eingabeschicht. Weitere Schichten können über die Funktion *AddLayer()* hinzugefügt werden. Die Anzahl der Neuronen in der Schicht muss zusammen mit der Aktivierungsfunktion zum Zeitpunkt dieses Aufrufs festgelegt werden. Aktivierungsfunktionen liegen immer gekapselt, zusammen mit ihrer Ableitungsfunktion, vor. Neue Funktionen können auf diese Weise leicht hinzugefügt werden. Die Gewichte zwischen den Neuronen werden bei der Initialisierung der Schichten pseudozufällig erstellt. Das geschieht, um Symmetrie im Netzwerk zu vermeiden. Dieses Verfahren wird auch *symmetry breaking* [Ree98, S. 97] genannt. Werden alle Gewichte mit den gleichen Werten initialisiert, verändern sie sich auch während

des Lernverfahrens immer gleichförmig, was die Lernfähigkeit des Netzwerks drastisch einschränkt.

Netzwerkeingaben können einem *MLP–Objekt* über die Funktion *Propagate()* präsentiert werden, die die zugehörigen Ausgaben zurückgibt. Dieser Teil bildet die Klassifizierung von Daten ab. Gleichzeitig speichert das Objekt intern Ein- und Ausgabewerte an jeder Schicht, um diese dem Lernverfahren bereitstellen zu können. Das Backpropagation–Verfahren ist fest in der Klasse *MLP* implementiert. Über die Funktion *Backpropagate()* werden die Fehlerwerte anhand der erwarteten Ergebnisse rückwärts durch das Netzwerk verbreitet. Schließlich lassen sich die Gewichte, basierend auf den ermittelten Fehlerwerten, mit *UpdateWeights()* anpassen. Gewichte, sowie interne Zwischenergebnisse und Informationen zum bisherigen Training lassen sich zu jeder Zeit aus dem Objekt auslesen. Die Implementierung stellt damit die Grundfunktionen eines Perzeptrons bereit und genügt allen Anforderungen, die durch die Zielsetzung der Arbeit gestellt werden.

Handwerklich berücksichtigt der Quellcode wichtige Prinzipien der Softwareentwicklung wie Modularisierung und Wartbarkeit. Die Verarbeitungsgeschwindigkeit wurde nicht mit der von anderen Implementierungen verglichen. Zu einer professionellen Lösung fehlen außerdem Aspekte wie Fehlerbehandlung, Plausibilitätsprüfungen oder Modultests. Um die Funktionalität des Pakets trotzdem bewerten zu können, wurde damit eine Erkennung von handschriftlichen Ziffern umgesetzt.

#### MNIST Ziffernerkennung

Die *MNIST–Datenbank* [LeC16] ist eine frei verfügbare Sammlung von vorverarbeiteten Bildern handgeschriebener Ziffern. Sie wird eingesetzt, um die Qualität und Funktionalität von Klassifizierungsmethoden zu überprüfen. Sie umfasst 60.000 Trainingsdaten- und 10.000 Testdatensätze. Die Bilder haben eine Auflösung 28x28 Pixeln. Damit sind sie klein genug, um jedes Pixel als Merkmal zu benutzen.

Diese Daten wurden einem dreischichtigen Netzwerk mit $784 - 100 - 100 - 10$ Neuronen präsentiert. Dabei wurden immer 100 der 60.000 Datensätze auf einmal propagiert, um die Zahl der Epochen künstlich zu erhöhen. Die Rahmenbedingungen hierfür sind dem Blogartikel über Matrixberechnungen [Dol14b] im *Backpropagation*-Verfahren entnommen worden. Als Ergebnis konnten Fehlerraten von unter 3% erreicht werden, wenn alle Datensätze dem Netzwerk oft genug (30-40x) präsentiert wurden. Schon nachdem dem Netzwerk alle Trainingsdatensätze einmal präsentiert worden waren, erreichte es eine Fehlerrate von circa 10%.

In Tabelle 3.1 werden Ergebnisse von Netzwerken aufgelistet, die teilweise über deutlich

komplexere Verfahren trainiert wurden. Im Vergleich erzielt die naive Implementierung dieser Arbeit gute Ergebnisse. Der Quellcode für dieses Training ist als Modultest in der Bibliothek enthalten. Zum Laden der MNIST–Daten wird dabei das Paket *github.com/petar/GoMNIST* verwendet.

| Jahr | Netzwerk | Fehlerrate (%) |
|---|---|---|
| 1998 | 2 Schichten, 300 verdeckte Neuronen | 4,70 |
| 1998 | 2 Schichten, 1000 verdeckte Neuronen | 4,50 |
| 1998 | 3 Schichten, 300+100 verdeckte Neuronen | 3,05 |
| 1998 | 3 Schichten, 500+150 verdeckte Neuronen | 2,95 |
| 2005 | 3 Schichten, 500+300 verdeckte Neuronen | 1,53 |
| 2003 | 2 Schichten, 800 verdeckte Neuronen | 0,70 |
| 2010 | 6 Schichten, 784-2500-2000-1500-1000-500-10 | 0,35 |

**Tabelle 3.1:** Auszug aus der MNIST Fehlerraten–Tabelle

#### Google Go

Die Programmiersprache *golang* hat zum Ziel, die Stärken klassischer Systemsprachen wie *C++* mit modernen Programmierkonzepten zu vereinen. Sie kommt in dieser Arbeit hauptsächlich aus Interesse und aufgrund ihrer Einfachheit zum Einsatz. Außerdem ist sie plattformunabhängig und wichtige Konzepte, wie die Parallelisierung von Programmteilen, sind direkt im Sprachkern enthalten. Zum Zeitpunkt der Arbeit war allerdings noch kein komfortabler Debugger verfügbar. Außerdem gestaltete sich die Arbeit mit der eingesetzten externen Matrix-Bibliothek *(github.com/gonum/matrix/)* als umständlich. Wenn sich das Ökosystem der Sprache allerdings weiterhin entwickelt, ist sie für Serveranwendungen durchaus empfehlenswert.

### 3.2.2 Kommunikation

Die Visualisierung des Netzwerks wird nicht in *golang* umgesetzt, weil die Sprache nicht gut für die Umsetzung grafischer Oberflächen geeignet ist. Deshalb muss die Funktionalität des Servers über eine Schnittstelle verfügbar gemacht werden, um deren Daten extern abzubilden. Dies geschieht über einen Webserver, der zusätzlich zur Weboberfläche, mit der das Netzwerk gesteuert wird, eine *Websocket–Schnittstelle* zur Verfügung stellt. Über diese findet die Kommunikation zwischen der grafischen Oberfläche und der Netzwerk–Bibliothek statt. Für beides wurde das Paket *"gopkg.in/igm/sockjs-go.v2/sockjs"* eingesetzt.

Die per *Websocket* ausgetauschten Daten werden mittels *json (JavaScript Object Notation)* formatiert. So können einerseits Steuerkommandos entgegengenommen und andererseits

Informationen zum künstlichen Netz versendet werden. Allgemein bietet die Schnittstelle nur zwei Methoden an. Die eine kann ein neues Netzwerk, basierend auf den Informationen zu den einzelnen Schichten, erzeugen. Die andere kann das bestehende Netzwerk anhand von Trainingsdaten lernen lassen. Zwischen den verschiedenen Aufrufen wird mit Hilfe von eindeutigen Bezeichnern unterschieden, die im Steuerkommando enthalten sind.

Der Server verwaltet immer nur ein Netzwerk gleichzeitig. Wird ein neues Netzwerk angefordert, überschreibt der Server das alte. Nach jeder dieser Operationen werden Testdaten in Form regelmäßiger Proben aus allen möglichen Eingaben $\{(x_1, x_2) \mid x_1 \in [0,1],\ \ x_2 \in [0,1]\}$ propagiert, um so die Klassifizierung des Netzwerks für alle Punkte der Fläche auszugeben. Im Anschluss sendet der Server die gesammelten Informationen über das Netz zurück an die Quelle des Steuerkommandos.

## 3.3 grafische Oberfläche

Der zweite Teil der Anwendung ist die Weboberfläche zur Steuerung des Netzwerks. Diese ist als *HTML5–Anwendung* realisiert. Die Logik ist in *JavaScript* implementiert und die Gestaltung über *HTML* und *CSS* definiert. Diese Kombination von Technologien ermöglicht einen immensen Gestaltungsfreiraum bei der Erstellung von browserbasierten Oberflächen. Die Webanwendung besteht aus mehreren Bedienelementen, die jeweils eine Funktion abbilden und in eigenen Dateien gekapselt sind.

### Konfiguration

Das erste Element ist die Konfiguration der Netzwerkschichten. Hierüber lässt sich ein neues Netzwerk definieren und vom Server anfordern. Der Nutzer kann verdeckte Schichten hinzufügen und entfernen. Die Ein- und Ausgabeschicht sind dabei statisch. Über ein Rastersystem kann die Anzahl der Neuronen pro Schicht eingestellt werden. Schichten werden über die Zeilen definiert und Neuronen über die Spalten (Abb. 3.1). Beide Werte sind begrenzt, um die Übersichtlichkeit der Darstellung zu gewährleisten.

### Training

Über das zweite Steuerelement lässt sich das erstellte Netzwerk trainieren (Abb. 3.2). Der Benutzer kann Trainingsdatensätze in Form von Punkten in eine Fläche eintragen, die einem Koordinatensystem entspricht. Jedem Punkt wird dabei eine Farbe, die vorher ausgewählt werden kann, zugeordnet. Die Farbe entspricht der Ausgabe, die vom Netzwerk bei Eingabe

dieses Punktes erwartet wird. Die Merkmale des Netzwerks sind also die beiden Koordinaten $x$ und $y$. Das Training des Netzwerks soll bewirken, dass nahegelegene Koordinaten gleich klassifiziert werden. Es wurden rot, grün und blau zur Kodierung der Klassen gewählt, weil mit Hilfe dieser Farben offensichtlich eine Darstellung des RGB–Farbraums optimal möglich ist. Diese Wahl ist auch der Grund, warum die Ausgangsschicht auf genau drei Neuronen begrenzt wurde.

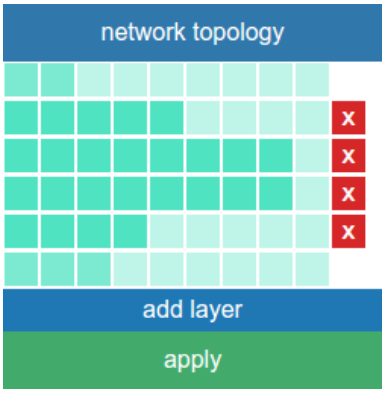



**Abbildung 3.1:** Konfiguration und schematische Darstellung der Schichten und Neuronen im Netz

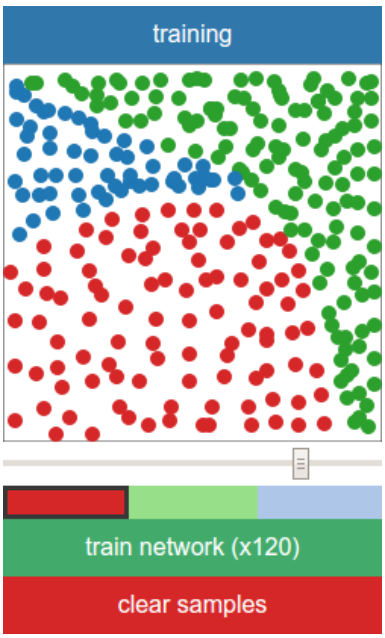



**Abbildung 3.2:** Element zur Erzeugung von Trainingsdaten und zur Steuerung des Trainings

### Vorschau

Eine Vorschau über die aktuellen Ergebnisse des Netzwerks gibt das dritte Element. Nach jedem Trainingszyklus wird hier ein Bild generiert, das die Ausgaben des Netzwerks für jeden Punkt des Eingabekoordinatensystems zeigt (Abb. 3.3). Hierfür kommt der Testlauf mit jeder möglichen Kombination von Merkmalen zum Einsatz.

Punkte, die das Netz eindeutig klassifizieren kann, werden gesättigt in der entsprechenden Farbe angezeigt. Bei unsicheren Klassifizierungen sind Mischfarben zu sehen. Der Effekt basiert darauf, dass das Netzwerk Wahrscheinlichkeiten für die Zuordnung zu den einzelnen Klassen liefert. Liegt die Eingabe nahe einer Hyperebene, die einen blauen von einem roten Bereich trennt, wird ein Lilaton ausgegeben. Die Anwendung sieht an dieser Stelle beide Klassifizierungen als wahrscheinlich an und erzeugt einen Farbton mit hohem Rot- und hohem Blauanteil. Das Training eines Netzwerks hat zum Ziel, die gesamte Fläche möglichst eindeutig zu klassifizieren und auch Bereiche ohne Testdatensätze plausibel abzudecken. Austrainierte Netze erzeugen daher gesättigte Farben und scharfen Kanten in der Vorschau.

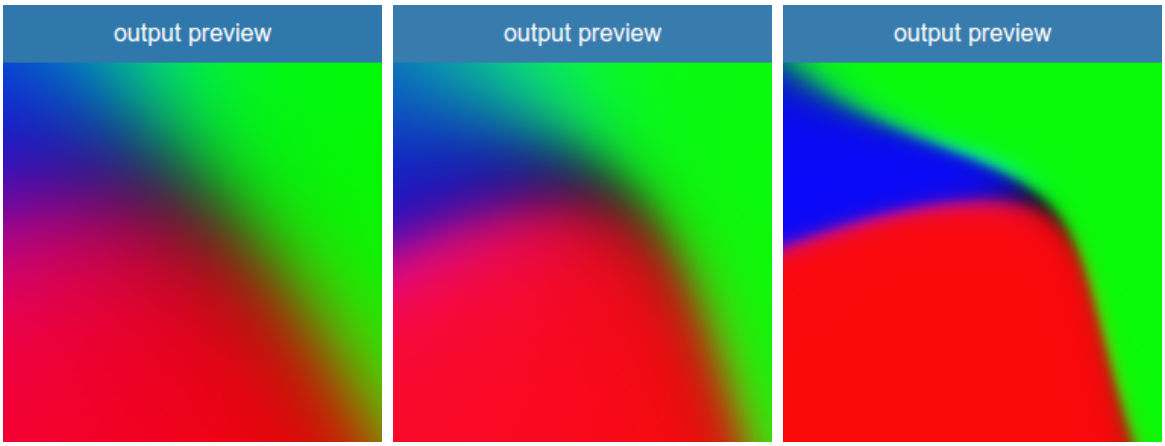



**Abbildung 3.3:** Vorschau des Trainingsergebnisses nach einer, zwei und fünf Millionen Beispieldaten

### Visualisierung

Das räumlich größte Element repräsentiert die Struktur des neuronalen Netzwerks (Abb. 3.5). Die einzelnen Neuronen werden zu Schichten zusammengefasst. Zwischen den Schichten werden die Gewichte der verbundenen Neuronen mit Hilfe von mehr oder weniger stark ausgeprägten Linien dargestellt. Dieses Element wurde mit *D3.js* umgesetzt. *D3.js* ist eine Bibliothek zur Visualisierung und Manipulation von Daten, die einfache *JavaScript*–Objekte so verarbeitet, dass aus diesen innerhalb des *D3.js*–Kontexts komfortabel manipulierbare Grafiken werden, ohne dabei das ursprüngliche Objekt zu

verändern. Für die simulierte Gravitation im System kommt das mitgelieferte Paket *force* zum Einsatz. Es ermöglicht, spezielle Bereiche des Netzes genauer zu betrachten, indem einzelne Neuronen herausgezogen werden. Die Darstellung des Netzes aktualisiert sich automatisch bei Anpassungen durch Training oder der Erstellung eines neuen Netzwerks.

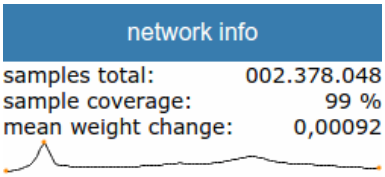
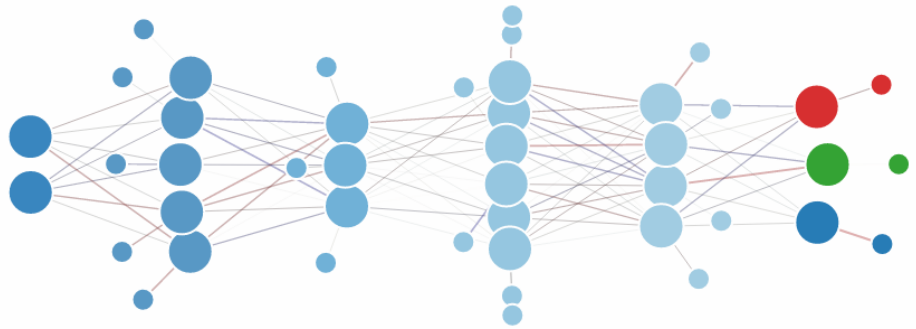
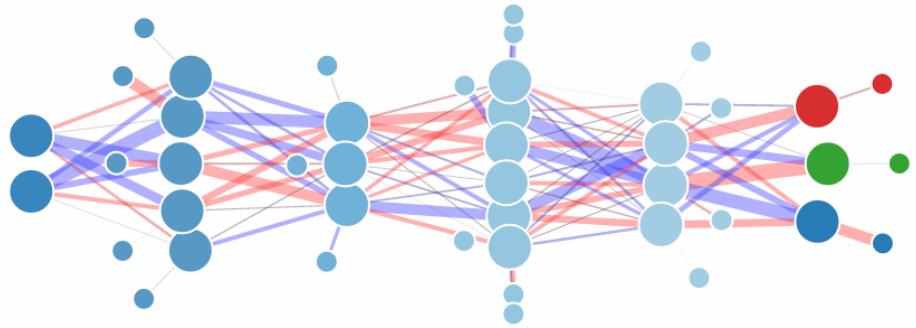



**Abbildung 3.4:** Darstellung verschiedener Kennzahlen zum Trainingsverlauf

Informationen zum Trainingsfortschritt des Netzes werden dem Benutzer in Textform präsentiert. Die Anzeige umfasst die Anzahl der bisher propagierten Datensätze, den Prozentwert der korrekt klassifizierten Trainingspunkte, sowie den Verlauf der durchschnittlichen Gewichtsveränderung (Abb. 3.4). Der Anwender kann damit zumindest oberflächlich den Verlauf des Lernverfahrens nachvollziehen.



**(a)**



**(b)**

**Abbildung 3.5:** Interaktive Darstellung eines untrainierten (a) und eines trainierten (b) Netzwerks mit gewichteten Verbindungen zwischen den Neuronen

## 3.4 Besonderheiten

Die Implementierung der Anwendung folgt in den meisten Punkten den Ausführungen über neuronale Netze des theoretischen Teils der Arbeit. Die größte Abweichung ergibt sich daraus, dass der Lernalgorithmus nicht als prozedurale Abarbeitung der einzelnen Neuronen realisiert wurde, sondern als Folge von Matrixmultiplikationen. Die Idee und Umsetzungsansätze folgen dem Blogartikel von *Brian Dolhansky* [Dol14b].

Anstatt jedes Neuron als einzelnen Knoten zu betrachten, werden alle Gewichte und die Ein- und Ausgaben pro Schicht als Matrizen behandelt. Dadurch wird es möglich, mit einer Operation mehrere Trainingsdatensätze auf einmal zu propagieren. Mittels Matrixmultiplikation der Eingaben des Netzwerks $X$ mit den Gewichten zwischen der ersten und der zweiten Schicht $W_{j\to k}$ werden beispielsweise die gewichteten Summen $U$, die den Aktivierungsfunktionen der zweiten Schicht übergeben werden, berechnet ($U = XW_{j\to k}$). Der Hauptvorteil dieses Vorgehens ist, dass die zugrundeliegende *BLAS (Basic Linear Algebra Subprograms)* Bibliothek die Berechnungen parallelisieren und somit die Verarbeitungsgeschwindigkeit beschleunigen kann.

Der Nachteil dieser Anpassung zeigt sich in der Verwaltung des Netzwerks. Während des Trainings müssen beim Berechnen der gewichteten Eingaben die Matrizen immer manipuliert werden, um auch Bias–Neuronen abzubilden. Diese verhalten sich nämlich anders als die restlichen Neuronen, weil ihnen die Verbindung zur vorherigen Schicht fehlt. Der gleiche Effekt tritt bei der Backpropagation und dem Berechnen der Gewichtsänderungen auf.

Das für *D3.js* verwendete Datenmodell geht außerdem davon aus, dass jedes Element einzeln vorliegt und Gewichte den Neuronen direkt zugeordnet werden können. Da *Bias–Neuronen* allerdings im Server–Datenmodell nur implizit auftreten, findet bei jeder Kommunikation eine komplexe Umwandlung der beiden Modelle statt.

#### Gesamteindruck

Insgesamt ist die Implementierung des praktischen Teils der Arbeit allerdings geradlinig. Der Quellcode ist strukturiert und gut verständlich. Die Schnittstelle zwischen Bibliothek und Oberfläche ist als zweckmäßig zu betrachten und müsste für eine Erweiterung überarbeitet werden. Rückblickend wäre es eine gute Idee gewesen, auch das neuronale Netz in *JavaScript* zu implementieren. Die Notwendigkeit einer Schnittstelle und das mühsame umwandeln der Datenmodelle wäre damit entfallen. Positiv zu sehen ist, dass gute Ergebnisse bei der Handschriftenerkennung erzielt werden konnten und tatsächlich eine intuitive Bedienoberfläche für neuronale Netze entstanden ist.

# 4 Fazit

Die Idee hinter dieser Arbeit war, die Grundlagen des Themas künstliche neuronale Netzwerke zusammenzutragen, im theoretischen Teil abzubilden, praktisch umzusetzen und damit nachvollziehbar zu machen. Schon bei der Recherche wurde klar, wie umfangreich der gesamte Themenkomplex ist. Informationen im Internet sind häufig nur sehr oberflächlich, oder verweisen auf Abstrahierungen durch Bibliotheken, die beim tieferen Verständnis nicht weiterhelfen. Informationen aus der Literatur sind im Gegensatz dazu zwar äußerst detailliert, erklären dann aber komplexere Zusammenhänge sehr mathematisch und erfordern Kenntnisse des Maschinellen Lernens. Außerdem haben sie selten Bezug zur Softwareumsetzung. Durch die lang andauernde Forschung zum Thema können viele der empfohlenen Texte aufgrund ihres Alters nicht auf die jüngeren Entwicklungen eingehen.

Zusammen mit der in der Einleitung beschriebenen Erwartungshaltung in Bezug auf künstliche Intelligenz ist es deshalb mühsam, relevante Informationen zu finden, die nicht entweder trivial sind oder sehr fortgeschrittene Konzepte und Spezialisierungen beschreiben. Der Großteil der Arbeit beschäftigt sich deshalb mit den elementaren Netztopologien und Verarbeitungsverfahren. Selbst ein Verständnis zu diesen Grundlagen aufzubauen, war zeitintensiv. Auf weiterführende Methoden wird in der Arbeit deshalb nur am Rande eingegangen.

## 4.1 Zielüberprüfung

Das Hauptziel der Arbeit, die Realisierung eines interaktiven neuronalen Netzwerks, ist gelungen. Ein einfaches Netz, in Form eines Perzeptrons, kann über eine Weboberfläche bedient werden. Warum welche Eingaben getätigt werden können und welche Auswirkungen sie haben, erschließt sich einem Nutzer ohne Vorwissen jedoch nicht. Jemandem mit Basiswissen kann die Anwendung aber helfen, tiefer in die Thematik einzusteigen. Dieses Wissen hätte noch expliziter über die Oberfläche transportiert werden können, kann aber durch den theoretischen Teil dieser Arbeit erlangt werden. Insgesamt stellt das Ergebnis der Arbeit einen soliden Einstieg in das Thema dar. Außerdem ist die Anwendung gut dazu geeignet, Restriktionen bestimmter Topologien, wie das XOR–Problem, zu visualisieren.

Die Oberfläche ist dabei so gestaltet, dass keine komplizierten Eingaben getätigt werden müssen, um das Netzwerk zu konfigurieren und zu trainieren. Dieser Punkt sorgt dafür, dass wichtige, aber komplizierte Konzepte, wie das Lernverfahren, nicht direkt abgebildet werden. Ein Ziel der Arbeit war es, Einblick in das Netzwerk während jedes einzelnen Programmschrittes zu bieten. Ein Konzept zur einfachen Darstellung dieses umfangreichen Ablaufs war insbesondere beim Training zeitlich nicht mehr möglich.

Deshalb waren die Optionen, das Lernverfahren entweder nüchtern über die sehr abstrakten Zahlenwerte zu präsentieren, oder es nicht abzubilden. Um eine übersichtliche Gestaltung weiterhin zu gewährleisten, wurde deshalb auf die Darstellung verzichtet. Das generelle Abwägen zwischen einer präzisen und einer einfachen Darstellungsweise war eine der größten Herausforderungen dieser Arbeit. In den meisten Fällen wurde zu Gunsten der Einfachheit entschieden. Trotzdem hat der Benutzer noch genügend Freiheiten, um die Reaktionen unterschiedlicher Netze auf seine Eingaben zu testen.

Ein weiteres Ziel war es, die Bedienung von Netzwerken im Hinblick auf umfangreichere Bibliotheken und abstrakte Werkzeuge zu erklären. Dieser Punkt ist teilweise gelungen. Wichtige Konzepte wie Topologien und Trainingsdaten werden abgebildet. Es fehlen zwar Beispiele von Netzen, die reale Probleme lösen, das konkrete Training von echten Anwendungsfällen kann über eine rein exemplarische oder theoretische Arbeit aber ohnehin nicht abgedeckt werden. Dazu muss der Anwender selbst Erfahrungen mit Testdaten, wie der *MNIST–Datenbank* sammeln. Erfahrungen abseits von echten Daten kann das Programm vermitteln und erzeugt damit eine Vorstellung von der Arbeitsweise umfangreicherer Werkzeuge und Bibliotheken.

## 4.2 Ausblick

Die Frage, ob und wann Maschinen denken können, kann auch nach dieser Arbeit nicht beantwortet werden. Es zeigt sich allerdings, dass der Trend, der das Thema zurzeit wieder umgibt, überzogen ist. Auch wenn die Entwicklungen im Bereich selbstlernender Netze spannend sind, ist ihr Funktionsumfang schon länger bekannt. Nur weil der Anstieg an Rechenleistung dafür sorgt, dass immer komplexere Netzwerke trainiert werden können, bedeutet das nicht, dass dadurch direkt eine perfekte Gehirnsimulation entsteht.

Fest steht aber, dass neuronale Netze ein interessantes Werkzeug sind, um Probleme auf eine alternative Weise anzugehen. In besonderem Maße lohnt es sich, den Einsatz bei komplexen Mechanismen zu erwägen, bei denen Ursache und Wirkung zwar bekannt sind, die entscheidenden Prozesse dahinter jedoch nicht. Künstliche Intelligenz in Videospielen ist ein Beispiel, bei dem bisher oft endliche Automaten eingesetzt werden, um vielseitiges,

intelligentes Verhalten zu simulieren. Dabei handelt es sich aber immer um einen festen Satz von Verhaltensregeln. Der Unterschied zum realen Gegenspieler ist deshalb noch immer deutlich spürbar. Neuronale Netze könnten die Eingaben von Spielern auf der Basis von Spielsituationen beobachten und aus ihnen lernen, um echtes Verhalten von Spielern nachzuahmen.

Wie jedes Werkzeug und jede Datenstruktur haben auch neuronale Netze Einsatzzwecke, für die sie gut oder weniger gut geeignet sind. Deshalb sollten sie zwar im Repertoire eines Informatikers nicht fehlen, aber trotzdem nur als weiteres Werkzeug eingesetzt und nicht zur Patentlösung stilisiert werden.

# Literaturverzeichnis


[Art16] ARTELNICS: *Open Neural Networks Library*, 2016. http://www.opennn.net/ [online; abgerufen am 02.02.2016].

[Bis95] BISHOP, CHRISTOPHER M.: *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., 1995.

[Dol13] DOLHANSKY, BRIAN: *Artificial Neural Networks: Linear Regression (Part 1)*, 2013. http://briandolhansky.com/blog/artificial-neural-networks-linear-regression-part-1 [online; abgerufen am 02.02.2016].

[Dol14a] DOLHANSKY, BRIAN: *Artificial Neural Networks: Mathematics of Backpropagation (Part 4)*, 2014. http://briandolhansky.com/blog/2013/9/27/artificial-neural-networks-backpropagation-part-4 [online; abgerufen am 02.02.2016].

[Dol14b] DOLHANSKY, BRIAN: *Artificial Neural Networks: Matrix Form (Part 5)*, 2014. http://briandolhansky.com/blog/2014/10/30/artificial-neural-networks-matrix-form-part-5 [online; abgerufen am 02.02.2016].

[Gar15] GARSON, JAMES: *Connectionism*, 2015. http://plato.stanford.edu/entries/connectionism/ [online; abgerufen am 02.02.2016].

[Gib16] GIBNEY, ELIZABETH: *Google AI algorithm masters ancient game of Go*, 2016. http://www.nature.com/news/google-ai-algorithm-masters-ancient-game-of-go-1.19234 [online; abgerufen am 02.02.2016].

[Goo16] GOOGLE: *TensorFlow*, 2016. https://www.tensorflow.org/ [online; abgerufen am 02.02.2016].

[Hum16] HUMAN BRAIN PROJECT: *Human Brain Project*, 2016. https://www.humanbrainproject.eu/ [online; abgerufen am 02.02.2016].

[Kok15] KOKCHAROV, IGOR: *Hierarchy of Skills*, 2015. http://de.slideshare.net/igorkokcharov/kokcharov-skillpyramid2015 [online; abgerufen am 02.02.2016].

[LeC16] LECUN, YANN: *THE MNIST DATABASE of handwritten digits*, 2016. http://yann.lecun.com/exdb/mnist/ [online; abgerufen am 16.02.2016].

[Mic16] MICROSOFT: *Project Oxford*, 2016. https://www.projectoxford.ai/ [online; abgerufen am 02.02.2016].

[Mue13] MUEHLHAUSER, LUKE: *When Will AI Be Created?*, 2013. https://intelligence.org/2013/05/15/when-will-ai-be-created/ [online; abgerufen am 02.02.2016].

[Ree98] REED, RUSSELL D. UND MARKS, ROBERT J.: *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1998.

[Ros62] ROSENBLATT, FRANK: *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[Tes07] TESCHL, GERALD UND TESCHL, SUSANNE: *Mathematik Für Informatiker – Analysis und Statistik*, Band 2 der Reihe *eXamen.press*. Springer-Verlag, Berlin, Heidelberg, New York, 2 Auflage, September 2007.

[Tur50] TURING, ALAN M.: *Computing Machinery and Intelligence*. Mind, 59, 1950.

[Van15] VANCE, ASHLEE: *The First Person to Hack the iPhone Built a Self-Driving Car. In His Garage*, 2015. http://www.bloomberg.com/features/2015-george-hotz-self-driving-car/ [online; abgerufen am 02.02.2016].

[Wik15] WIKIPEDIA: *Physical symbol system — Wikipedia, The Free Encyclopedia*, 2015. https://en.wikipedia.org/w/index.php?title=Physical_symbol_system&oldid=688526513 [online; abgerufen am 02.02.2016].

[Wik16a] WIKIPEDIA: *IPO model — Wikipedia, The Free Encyclopedia*, 2016. https://en.wikipedia.org/w/index.php?title=IPO_model&oldid=704008085 [online; abgerufen am 14.02.2016].

[Wik16b] WIKIPEDIA: *Scale-invariant feature transform — Wikipedia, The Free Encyclopedia*, 2016. https://en.wikipedia.org/w/index.php?title=Scale-invariant_feature_transform&oldid=699439255 [online; abgerufen am 02.02.2016].

[Zel97] ZELL, ANDREAS: *Simulation neuronaler Netze*. Oldenbourg, Deutschland, 1. Auflage, 1997.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Eigenständigkeitserklärung

Ich versichere, die vorliegende Arbeit selbständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Finn Ole Koenecke