

Realisierung eines interaktiven künstlichen neuronalen Netzwerks in Javascript mit spielerischen Elementen

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Xuan Linh Do

2138196



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

Erstprüfer: Prof. Dr. Edmund Weitz

Zweitprüfer: Prof. Dr. Andreas Pläß

vorläufige Fassung vom 30. August 2017

Inhaltsverzeichnis

1	Einleitung	5
1.1	Motivation	5
1.2	Begriffsdefinition	5
1.3	Zielsetzung	6
1.4	Anmerkungen im Bezug auf Koeneckes Arbeit	7
1.5	Struktur dieser Arbeit	8
2	Grundlagen	9
2.1	Künstliche Intelligenz	9
2.1.1	Begriffserklärung	9
2.1.2	Turing Test	10
2.1.3	Starke und schwache Künstliche Intelligenz (KI)	10
2.2	Maschinelles Lernen	10
2.2.1	Begriffserklärung	10
2.2.2	Prinzip und Verfahren	11
2.3	Einordnung der KNN	12
2.4	Bereiche der KNN	12
3	Vorbild der KNN und das Prinzip eines künstlichen Neurons	13
3.1	Natürliche neuronale Netze	13
3.2	Funktionsweise und Aufbau eines künstlichen Neurons	14
3.2.1	Übertragungsfunktion	15
3.2.2	Aktivierungsfunktion	15
3.2.3	Ausgangsfunktion	18
4	Struktur der KNN	19
4.1	Aufbaustruktur	19
4.2	Einlagiges Perzeptron	21
4.2.1	Darstellung der booleschen Funktionen als einfaches Perzeptron	21
4.2.2	Klassifizierung und lineare Separierbarkeit	22
4.2.3	Das XOR-Problem	24
4.3	Mehrlagiges Perzeptron	24
4.3.1	Darstellung der XOR-Funktion als MLP	25
4.3.2	Bias-Neuronen	25

5	Lernen in KNN	27
5.1	Lernen durch Modifikation der Gewichte	27
5.2	Lernarten	27
5.3	Lernverfahren	28
5.4	Lernregel	29
5.4.1	Hebbsche Lernregel	29
5.4.2	Delta-Regel	30
5.4.3	Backpropagation-Regel	30
5.5	Auswahl der Aktivierungsfunktion	34
6	Das ursprüngliche Programm von Koenecke	35
6.1	Problemstellung und Netzwerktopologie	35
6.2	Technologie	35
6.3	Festgelegte Parameter	36
6.4	Aufbau der GUI	36
6.4.1	Netzwerktopologie	36
6.4.2	Training	37
6.4.3	Präsentation der Netzwerkergebnisse	37
7	Eigene Implementierungen	39
7.1	Implementierung des KNN in Javascript	39
7.1.1	Wahl der passenden Javascript Bibliothek für das KNN	39
7.1.2	Performancevorteile durch Nutzung von Web Workern	40
7.1.3	Trainieren eines MLP mit Synaptic	41
7.1.4	Schnittstelle zwischen der Netzworkebibliothek und der GUI	43
7.1.5	Erweiterungen an der GUI	45
8	Konfiguration eines KNN passend zur Problemstellung	46
	Abbildungsverzeichnis	48
	Tabellenverzeichnis	49
	Literaturverzeichnis	50

Abstract

English

Zusammenfassung

Diese Arbeit ist eine Fortführung der von Finn Ole Koenecke angefertigten Bachelorarbeit mit dem Titel „Realisierung eines interaktiven künstlichen neuronalen Netzwerks“ aus dem Jahre 2016. Im Rahmen seiner Arbeit wurde ein Programm geschrieben, in dem der Anwender interaktiv die Arbeitsweise und Strukturen von neuronalen Netzen visualisieren und erlernen kann. Das ursprüngliche Programm von Koenecke ist so aufgebaut, dass die graphische Oberfläche über den Browser und die Rechenarbeit für das neuronale Netz im Hintergrund über einen Server läuft.

Ziel dieser Bachelorarbeit ist es, dieses Programm komplett in Javascript, HTML und CSS zu portieren, sodass dieses auch ohne einen Server im Browser läuft. Zudem wird das Programm so erweitert, dass das neuronale Netz spielerisch eingesetzt werden kann. Die Erweiterungen sind darauf ausgelegt, noch mehr den didaktischen Charakter des Programms sowie die Einsatzmöglichkeiten von neuronalen Netzen hervorzuheben.

1 Einleitung

1.1 Motivation

Bei vielen Aufgaben ist es von großem Vorteil auf technische Hilfsmittel zuzugreifen, da sie so erheblich schneller gelöst werden können. Zum Beispiel kann für die Berechnung komplexerer Rechenaufgaben der Taschenrechner genutzt werden. Jedoch soll nicht außer Acht gelassen werden, dass ein Rechner darauf beschränkt ist, nur eine begrenzte Anzahl von Aufgaben erledigen zu können. Der Mensch kann sich dagegen mit seiner Flexibilität und Vielseitigkeit auszeichnen. Zudem gibt es immer noch viele Aufgaben, in denen der Computer dem Menschen deutlich unterlegen ist. *„Wahrnehmung und Erkennen, Lernen und Speichern von Informationen, Anpassung an die Umwelt, Steuerung von Verhalten sowie Kreativität sind Leistungen des menschlichen Nervensystems, die Maschinen bisher erst in Ansätzen in der Lage sind nachzuahmen.“* (Kramer 2009: S. 119)

Zu den Ansätzen, die im Zitat genannten Fertigkeiten auch dem Computer zu ermöglichen, zählen unter anderem künstliche neuronale Netze (KNN). Dank der jüngsten technischen Fortschritte in diesem Bereich können immer mehr komplexere Aufgaben auch vom Computer gelöst werden. Im Bereich der Spiele können sie zum Beispiel beim Jump'n'Run-Videospiel *Super Mario World* eingesetzt werden, wie der Youtuber *SethBling* in einem seiner Videos zeigt (SethBling 2015). Zudem gibt es nicht nur Computerprogramme, die in der Lage sind, mithilfe von KNN die besten Spieler in komplexen Spielen zu schlagen, in denen alle Informationen offen gelegt sind wie beim Brettspiel *Go*. Auch bei *Texas Hold'em*, einer Variante des Kartenspiels Poker, die aufgrund der verdeckten Karten auf die Intuition und Glück aufbauen, haben KNN dazu beigetragen, dass professionelle Spieler von einem Computerprogramm geschlagen werden konnten (Riley 2017). Die Abbildung 1.1 zeigt, in welchen Bereichen KNN noch eingesetzt werden können.

1.2 Begriffsdefinition

KNN sind die stark abstrahierte, technische Umsetzung der biologischen neuronalen Netze des menschlichen Gehirns. Mithilfe des Wissens über die Struktur und

Bereiche	Anwendungsgebiete		
Industrie	Qualitätskontrolle	Kapazitätsplanung	Optimierung
	Sortierung	Robotersteuerung	Materialsynthese
	Steuerungskontrolle	Mitgliederanswahl	Bildverarbeitung
Finanzen	Bonitätsvorhersage	Wertpapierbewertung	Kursprognose
	Buchstabenerkennung	Unterschriftenerkennung	Schätzungen
Telekommunikation	Datenkompression	Routingstrategien	adaptive Filter
	Optimierung des Signalverkehrs		Netzoptimierung
Medizin	Atemanalyse	Blutdruckanalyse	Klinikmanagement
	Diagnose	Bakterienidentifikation	Gewebeanalyse
Marketing	Erkennung von Mustern	Zielgruppenbestimmung	Konsumentenanalyse
Künstliche Intelligenz	Dateimanagement	Wissensgewinnung	Vorstrukturierung
	Spracherkennung	implizite Regeln	
Öffentlicher Dienst	Formularverarbeitung	Dienstplanoptimierung	Handschriftlesen
	Automatisierung/Optimierung des Postverkehrs		
Verkehr	Hinderniserkennung	Fahrplanoptimierung	Ampelschaltung
	Routenplanung für autonome Fahrzeuge		

Abbildung 1.1: Anwendungsgebiete für den Einsatz von KNN, entnommen aus <http://vieta.math.tu-cottbus.de/~kolb/ml-nn/node10.html>

Funktionsweise vom Nervensystem können die biologischen Prinzipien der Informationsverarbeitung als systematisches Modell für den Rechner übertragen werden.

1.3 Zielsetzung

Das Hauptziel dieser Arbeit ist, dem Leser und Anwender einen leichten Einstieg in das Thema KNN und deren Einsatzmöglichkeiten zu bieten. Damit wird dasselbe Ziel verfolgt, das schon Herr Koenecke mit seiner Arbeit hatte. Mit dem theoretischen Teil soll eine gewisse Wissensbasis über KNN geschaffen werden. Beim praktischen Teil kann der Anwender mithilfe eines Programms, das durch den theoretischen Teil erworbene Wissen interaktiv veranschaulichen und verfestigen. Folgender Textauszug aus Herr Koeneckes Arbeit zeigt, was die Anforderungen an seinem geschriebenen Programm sind:

„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.“ (Koenecke 2016: S. 7)

Weitestgehend sind die Anforderungen mit dem ursprünglichen Programm erreicht worden. Ausgehend von diesem Programm als Basis sollen für diese Arbeit zwei größere Änderungen stattfinden:

1. *Portieren seines Programms komplett in Javascript, HTML und CSS*: Dies setzt voraus, dass die Berechnungen für das KNN nicht mehr wie bisher im Hintergrund über einen Server laufen, sondern komplett lokal im Browser in Javascript durchgeführt werden. Von der Portierung ist zu erwarten, dass das Programm zwar nicht so schnell wie das ursprüngliche Programm laufen wird, aber dennoch performant genug, dass es sich flüssig in einem aktuellen Browser benutzen lässt.
2. *Änderung des User Interfaces durch Hinzufügen spielerischer Elemente*: Die graphische Oberfläche des Programms soll verändert werden, sodass spielerische Interaktionen möglich sind. Dabei ist es wichtig, dass das Näherbringen der Grundlagen von KNN immer noch im Vordergrund des Programms stehen soll. Daher sind die Veränderungen am Programm vor allem als Erweiterungen zu betrachten. Die Idee für die Erweiterungen fokussiert sich dabei auf den letzten Satz der oben von Herr Koenecke zitierten Textpassage, also die Möglichkeit anhand bekannter Probleme die Arbeitsweise KNN zu erlernen. Im ursprünglichen Programm ist der Anwender noch dazu gezwungen, sich die Probleme selbst herauszusuchen oder auszudenken. Um ihm diesen Schritt zu ersparen, soll es zusätzlich eine Option vom Programm geben, anhand gestellter Aufgaben die Probleme kennenzulernen. Zum Lösen dieser Aufgaben, muss der Anwender die passenden Konfigurationen des KNN vornehmen. So erhält der Anwender die Möglichkeit spielerisch zu erlernen, wie und wozu KNN eingesetzt werden können.

1.4 Anmerkungen im Bezug auf Koeneckes Arbeit

Zum Lesen dieser Arbeit soll keine zwingende Voraussetzung sein, Herrn Koeneckes Bachelorthesis durchzulesen. Daher werden sich einige Überschneidungen und Verweise zu Koeneckes Arbeit finden. Dies lässt sich sowieso nicht komplett aufgrund der Tatsache vermeiden, dass diese Arbeit und der dazugehörige praktische Teil auf Grundlage seiner Arbeit und Anwendung entstanden ist. Die größten Überschneidungen finden sich vor allem in den Kapiteln 2 - 5, in denen die Theorie behandelt wird, die mit der Künstlichen Intelligenz und den KNN in Verbindung stehen ([Koenecke 2016](#): nachzulesen in seiner Thesis auf den Seiten S. 4-5 sowie 11-32). Im Unterschied zu seiner Arbeit wurde die Theorie der KNN auf mehr Kapitel unterteilt und eine andere Strukturierung ausgewählt, mit dem Ziel eine bessere Übersicht über das Thema zu bieten. Ob dieser Weg wirklich der bessere ist, hängt im Grunde genommen

vom Leser ab. Bevorzugt er es lieber kompakt in einem Kapitel sich die Theorie anzueignen, wäre die bessere Entscheidung sich Herr Koenekes Arbeit durchzulesen. Möchte er lieber stückchenweise sich über verschiedene Kapitel die Theorie aneignen, kann diese Arbeit die bessere Wahl sein.

1.5 Struktur dieser Arbeit

KOMMT AM ENDE

2 Grundlagen

In diesem Kapitel soll eine grundlegende Wissensbasis zur Einordnung der Begriffe KI, Maschinelles Lernen, Deep Learning und KNN geschaffen werden. So kommt es bei Laien häufiger zu Verwechslungen dieser Begriffe, deswegen sollen in diesem Kapitel genauer auf sie eingegangen werden, bevor im Kapitel 3 der tiefere Einblick in die Theorie der KNN erfolgt. Auf diese Weise lässt sich besser verstehen, was KNN sind und was sie nicht sind, für welchen Bereich sie eine Bedeutung spielen und wie sie in der Informatik einzuordnen sind. Es wird sich zeigen, dass das Thema KI so breit gefächert ist, dass für die Arbeit nur ein oberflächlicher Einschnitt in das Thema erfolgt. So werden vor allem die Grundlagen behandelt, die für die Thematik der KNN und das im Rahmen der Bachelorthesis geschriebene Programm von Bedeutung sind. Den Lesern, die tiefer in die Materie gehen möchten, steht es nach der Einführung mithilfe dieser Bachelorthesis offen, sich danach anderweitig über das Thema KI zu informieren.

2.1 Künstliche Intelligenz

2.1.1 Begriffserklärung

Ob der Mensch ein Lebewesen oder eine Maschine für intelligent hält oder nicht, beruht meist auf seine relativ gute Intuition. So beurteilt er wie ausgeprägt die kognitiven Fähigkeiten sind, um sich an ungewohnte Situationen anzupassen. Aus wissenschaftlicher Sicht gibt es jedoch keine allgemein gültige Definition für den Begriff *Intelligenz* (Rimscha 2014: S. 139). „Befragt man heute 20 Intelligenzforscher nach einer Definition, erhält man 20 unterschiedliche Antworten“, ist die Aussage von Thomas Grüter, einem Arzt und Sachbuchautor, der für die Neurophysiologie forschte (Hillmann: S. 139). Dass der Begriff Intelligenz schwer zu definieren ist, zeigt ebenfalls, wie schwierig es ist, die KI zu definieren. Grob formuliert wird unter KI *die Nachbildung menschlicher Intelligenz durch Maschinen* verstanden (Quelle nochmal raussuchen :P). Dies betrifft auch die menschliche Wahrnehmung und das menschliche Handeln.

2.1.2 Turing Test

Um festzustellen, ob eine Maschine dem Anspruch der KI genügt, hat Alan M. Turing als Vorschlag in seinem Paper (Turing 1950) den nach ihm benannten Turing-Test ausgedacht. Über längere Zeit kommuniziert ein Mensch als Tester parallel mit zwei Gesprächspartnern. Bei dem einen Gesprächspartner handelt es sich um einen Menschen, bei dem anderen um eine Maschine. Die Kommunikation erfolgt ohne Sicht- und Hörkontakt (beispielsweise über ein Chat-Programm). Sowohl Mensch als auch Maschine versuchen den Tester zu überzeugen, dass sie denkende Menschen sind. Kann der Tester nach der Unterhaltung nicht bestimmen, welcher von den Gesprächspartnern die Maschine ist, gilt der Test als bestanden und die Maschine kann als intelligent bezeichnet werden.

Bisherige Versuche diesen Test zu bestehen, gab es bereits mit Programmen wie *ELIZA* (1966) oder dem Chatbot *Eugene Goostman* (2014). Doch bei keinem dieser Versuche kann von Experten wirklich überzeugt bestätigt werden, dass sie den Turing Test bestanden haben. Und trotz der intensiven Forschung im Bereich der KI wird mit großer Wahrscheinlichkeit auch in absehbarer Zeit dies nicht passieren.

2.1.3 Starke und schwache KI

Aufgrund der Komplexität der KI ist die Idee gekommen, eine Unterscheidung vorzunehmen (Russell & Norvig 2010). Zum einen die *starke KI*, die dem Menschen ebenbürtig ist und dazu instande ist, komplexe kognitive Aufgaben autonom zu bewältigen, die sonst nur der Mensch lösen kann. Zum anderen die *schwache KI*, die sich auf die Lösung der Probleme konzentriert, die konkretisiert und stark eingrenzt sind. In dieser Arbeit wird es ausschließlich um die Lösungsansätze von Problemen aus dem Gebiet der schwachen KI gehen. Beispielsweise zählen in das Gebiet der schwachen KI Suchmaschinen, Spracherkennung und Bilderkennung, in denen in letzter Zeit beachtliche Erfolge erzielt werden konnten. Als Beispiele sind das 2014 von Amazon veröffentlichte, sprach-gesteuerte Gerät Amazon Echo oder die Internet-Suchmaschine Google zu nennen. Sowohl die starke als auch die schwache KI haben gemeinsam, dass bei beidem das Maschinelle Lernen eine immer größere Bedeutung spielt.

2.2 Maschinelles Lernen

2.2.1 Begriffserklärung

Beim Maschinellen Lernen (englisch: *artificial intelligence* - *AI*) geht es um den Gewinn von Informationen und Wissen aus Erfahrung in Form von konkreten Beispielen.

daten (Rimscha 2014: S. 143). Im Gegensatz zur *Symbolischen KI* löst der Computer also die Aufgabe nicht durch eine ihm vorher vorgegebene Lösungsvorschrift, sondern muss sich die Lösungsvorschrift selbst ableiten. Dadurch ist es für ihn auch möglich, Lösungen für neue oder unbekannte Probleme zu finden.

Die abgeleitete Lösungsvorschrift und damit die Ergebnisse dieser sind nur so gut, wie die Beispieldaten. Wenn die Beispieldaten nur einen bestimmten Bereich des Problems abdecken, aber nicht das Ganze, so werden die Daten als schlecht betrachtet. Um zu verdeutlichen, was damit gemeint ist, folgendes Beispiel: Es soll das Konsumverhalten von Pizza untersucht werden und es handelt sich bei den Testpersonen zufälligerweise um welche, die auf Diät sind. Die Repräsentativität der daraus entstehenden Ergebnisse wird unter diesen Bedingungen stark in Zweifel genommen werden können. Ebenfalls führen zu wenige zur Verfügung stehende Daten zu schlechten Ergebnissen, deswegen können diese auch als schlechte Beispieldaten betrachtet werden.

2.2.2 Prinzip und Verfahren

Das Maschinelle Lernen funktioniert im Grunde ähnlich wie das menschliche Lernen (Manhart 2017). So wird versucht, anhand der gegebenen Daten diese intelligent miteinander zu verknüpfen, durch Ausprobieren und Abgleichen bestimmte Zusammenhänge sowie Ergebnisse zu erschließen und anhand dieser Vorhersagen treffen zu können. Bei der Erkennung von Objekten auf Bildern lernt ein Mensch im Kindesalter zum Beispiel wie ein Hund aussieht, indem ihm Beispielbilder gezeigt werden, mit dem Hinweis, dass es sich um Hunde handelt. Auch beim Maschinellen Lernen füttert und trainiert der Programmierer das Lernprogramm mit Informationen in Form von Daten. Bei jeder Datei gibt er zusätzlich an, ob es sich um einen Hund handelt oder nicht. Je mehr Beispieldaten das Programm bekommt, desto besser kann er unterscheiden, auf welchen Bildern Hunde zu sehen sind und auf welchen nicht.

Um aus den Beispieldaten Wissen zu gewinnen, gibt es verschiedene Verfahren, die mathematische und statistische Algorithmen und Methoden verwenden. Davon nennenswerte wären zum Beispiel Entscheidungsbäume, Reinforcement Learning, Genetische/Evolutionäre Algorithmen, das Bayes'sche Lernen, der Monte Carlo Tree Search oder KNN.ⁱ Die verschiedenen Verfahren sind dabei nicht völlig getrennt voneinander zu betrachten. Beispielsweise finden sich Überschneidungen zwischen genetischen Algorithmen und KNN. Zudem sei zu erwähnen, dass für die Lösung eines Problems auch oftmals nicht nur einer der Verfahren verwendet wird, sondern eine Kombination der verschiedenen Verfahren.

ⁱDie beiden letztgenannten Verfahren wurden unter anderem auch in dem von der Google-Tochter *Deepmind* stammendem Programm *Alpha Go* verwendet, der in der Lage war, im Mai 2017 den damaligen Weltranglistenersten *Ke Jie* in drei von drei Go-Partien zu schlagen (Wunderlich-Pfeiffer 2017). Das ist insofern beachtlich, dass Go als sehr komplexes Brettspiel gilt. So bietet es mit ca. $2,08 \cdot 10^{170}$ möglichen Stellungen sogar mehr als Schach mit 10^{43} möglichen Stellungen.

2.3 Einordnung der KNN

Anhand der vorherigen Erläuterungen sollte nun klar zu unterscheiden sein, wie die verschiedenen Begriffe im Zusammenhang stehen. So sind KNN ein Unterthema des Maschinellen Lernens, das wiederum ein Unterthema der KI ist. Die Abbildung 2.1 soll diesen Zusammenhang veranschaulichen.

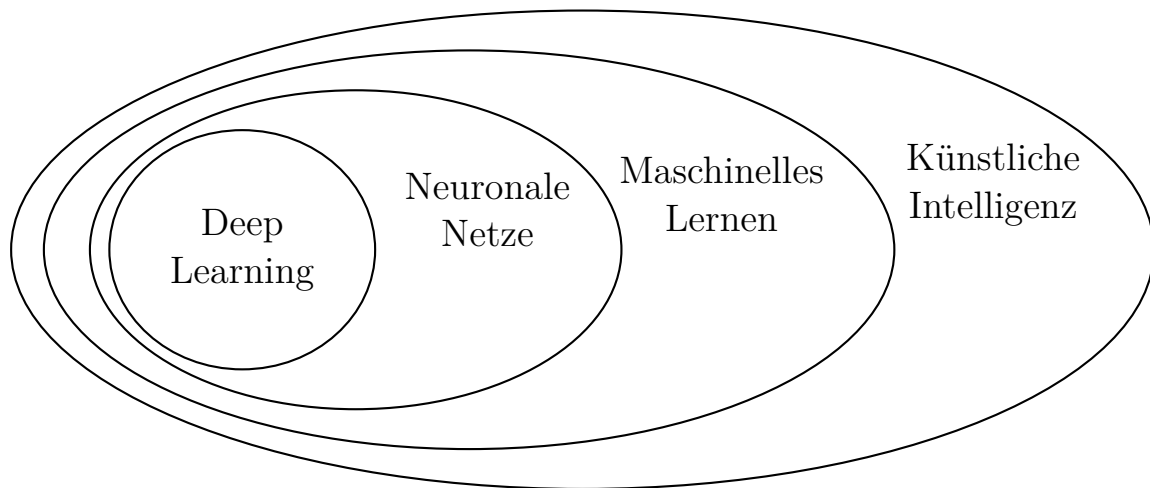


Abbildung 2.1: Venn Diagramm zur Einordnung der KNN

Der Vollständigkeit halber erscheint im Diagramm auch der für die Arbeit relevante Begriff *Deep Learning*. Bei Deep Learning handelt es sich um ein Teilgebiet der KNN, die mit speziellen Formen von KNN arbeiten. Um welche es sich dabei handelt, wird später im Abschnitt **noch ergänzen** beschrieben.

2.4 Bereiche der KNN

Wie am Anfang des Kapitels 1 bereits erläutert, werden KNN in zahlreichen Anwendungsgebieten eingesetzt. Unterteilt werden kann dabei in zwei große Bereiche (Rey & Wender 1982). Zum einen die *Modellierung menschlichen Verhaltens und Erlebens*, in der KNN eingesetzt werden, um gewisse Gehirnprozesse zu simulieren und die Funktionsweise des Gehirns besser nachvollziehen zu können. Zum anderen die *Lösung konkreter Anwendungsprobleme*, in der KNN zur Lösung dieser in Bereichen wie der Statistik, Informatik, Wirtschaftswissenschaften eingesetzt werden. Die Arbeit wird sich vor allem mit den Problemstellungen des zweiten Bereiches beschäftigen.

3 Vorbild der KNN und das Prinzip eines künstlichen Neurons

3.1 Natürliche neuronale Netze

Das Besondere an KNN und was sie von anderen mathematischen Verfahren unterscheidet ist, dass sie ursprünglich dem menschlichen Gehirn und den natürlichen neuronalen Netzen nachempfunden sind. Mit dem natürlichen Vorbild haben sie genau genommen jedoch kaum zu tun, da es nach aktuellem Stand noch unmöglich ist, komplett und korrekt die Eigenschaften der natürlichen neuronalen Netze in künstlicher Form abzubilden. Daher wird die folgende Darstellung der natürlichen neuronalen Netze stark vereinfacht sein, sodass noch deutlicher wird, inwiefern sich die KNN an ihnen orientieren.

Das menschliche Gehirn gilt als das komplexeste und vielschichtigste Organ in der ganzen Neurobiologie. Bekannt zu seinem Aufbau ist, dass es geschätzt 100 Milliarden (10^{11}) Neuronen bzw. Nervenzellen enthält, die alle zusammen ein Netzwerk bilden. Verknüpft sind die Nervenzellen mit ca. 10^{14} - 10^{15} synaptischen Verbindungen (Kramer 2009: S. 119-122). Eine Nervenzelle besteht aus einem Zellkörper, vielen Dendriten und einem Axon (siehe Abb. 3.1). Die Dendriten sind dafür zuständig Signale, die es von anderen Neuronen bekommt, aufzufangen und an den Zellkörper hinzuleiten. Im Zellkörper findet deren Verarbeitung statt. Wenn die Summe der empfangenen Signale, die die Nervenzelle von den anderen Nervenzellen bekommen hat, einen gewissen Schwellenwert der Erregung überschreitet, wird ein elektrischer Impuls, das Aktionspotenzial, gesendet. Es wird dann gesagt, dass die Nervenzelle *feuert* (Corves 2005). Über das Axon wird dieser zu einer Synapse weitergeleitet. Bei einer Synapse handelt es sich um eine Verbindungsstelle zwischen zwei Nervenzellen, wodurch die Übertragung eines elektrischen Impulses von einer Nervenzelle zu einer anderen ermöglicht wird. Der elektrische Impuls sorgt dafür, dass die andere Nervenzelle entweder gehemmt oder erregt wird.

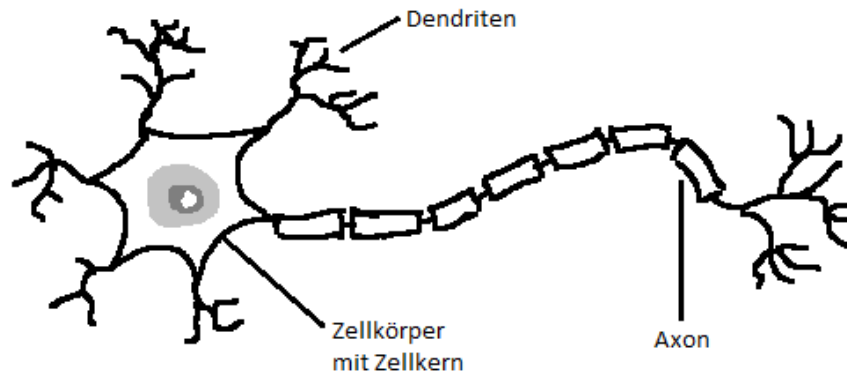


Abbildung 3.1: Aufbau einer Nervenzelle

3.2 Funktionsweise und Aufbau eines künstlichen Neurons

KNN bestehen aus einer technischen Version der Nervenzellen, den *künstlichen Neuronen*. Andere Bezeichnungen für Neuronen sind Units, Einheiten oder Knoten (Rey & Wender 1982: S.14). Ein einzelnes Neuron hat die einfache Aufgabe verschiedene Eingangssignale aufzunehmen, diese zu verarbeiten und daraus ein Ausgangssignal auszugeben. Zur Veranschaulichung der damit verknüpften mathematischen Zusammenhänge, soll die Abbildung 3.2 betrachtet werden. Die Nummerierung soll die Reihenfolge darstellen.

Es soll $i, j, n \in \mathbb{N}$ gelten.

1. Ist n die Anzahl der Eingaben, so bekommt das Neuron j die Eingangssignale beziehungsweise *Eingangswerte* x_1 bis x_n (biologische Analogie: Dendriten). Die Eingangswerte stammen entweder von anderen Neuronen oder Reizen aus der Umwelt.
2. Die *Gewichte* (engl. *weights*) w_{ij} stehen für die Stärke der Verbindungen (biologische Analogie: Synapse) vom Sender Neuron i zum Empfänger Neuron j . Durch sie wird bestimmt, zu welchem Grad die empfangenen Eingangswerte Einfluss auf die spätere Aktivierung des Neurons j nehmen werden. Je nach Vorzeichen werden die Eingangssignale jeweils verstärkt oder geschwächt.
3. Die *Netzeingabe* net_j verarbeitet die erhaltenen Informationen (also die Eingangssignale und die dazu gehörigen Gewichtungen) und berechnet sich mithilfe der Übertragungsfunktion $net(j)$ (auch Propagierungs- oder Netzeingabefunktion genannt).

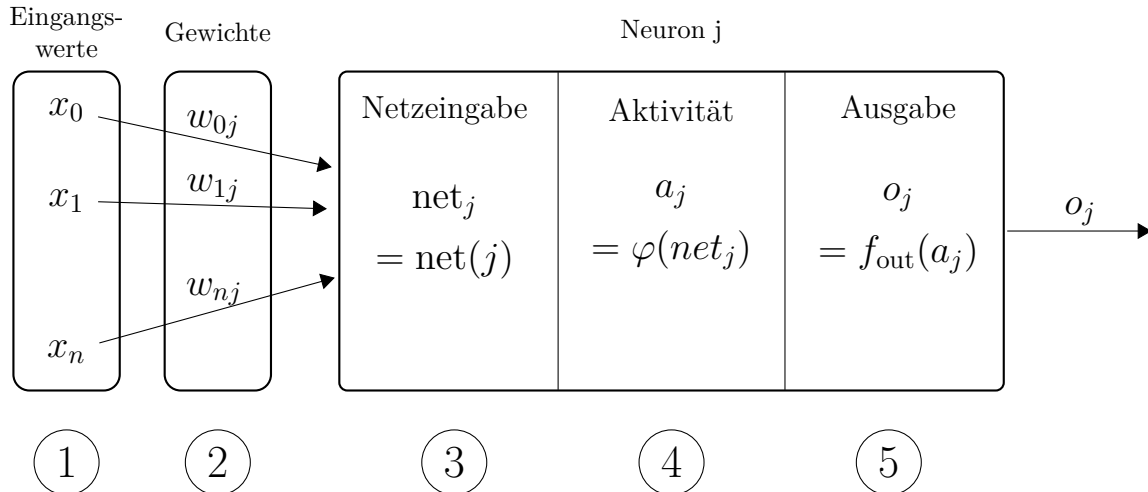


Abbildung 3.2: Funktionsweise eines Neurons, modifizierte Version aus <http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html>

- Der *Aktivitätslevel* a_j beschreibt den aktuellen Zustand des Neurons j und berechnet sich aus der Netzeingabe mithilfe der Aktivierungsfunktion $\varphi(net_j)$ (auch Aktivitätsfunktion genannt).
- Die *Ausgabe* o_j berechnet sich mithilfe der Ausgabefunktion $f_{out}(a_j)$. Sie ist der Wert, der entweder später anderen Neuronen als Eingabe gesendet wird oder einen Ausgabewert des KNN darstellt.

3.2.1 Übertragungsfunktion

Die am häufigsten verwendete Übertragungsfunktion ist die Linearkombination, bei der die Eingangssignale x_1 bis x_n mit den Gewichten w_{1j} bis w_{nj} multipliziert werden und alle Produkte aufsummiert werden (siehe Gl. 3.1). Daher wird für die Übertragungsfunktion häufig als Symbol \sum verwendet.

$$net_j = net(j) = \sum_{i=0}^n x_i w_{ij} \quad (3.1)$$

3.2.2 Aktivierungsfunktion

In der Literatur sind zahlreiche Beispielfunktionen zu finden, die sich als Aktivierungsfunktion nutzen lassen. In den Tabellen 3.1 und 3.2 sind die Aktivierungsfunktionen aufgeführt, die häufig zu finden sind.

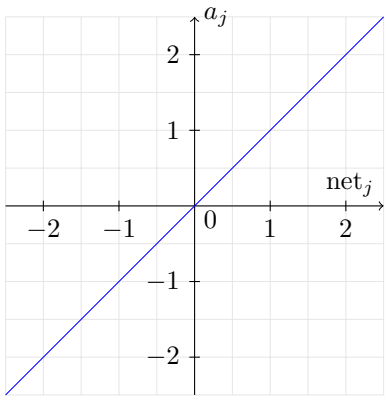
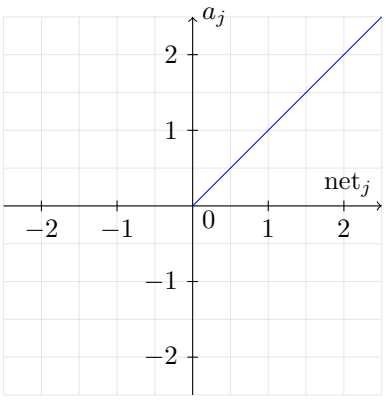
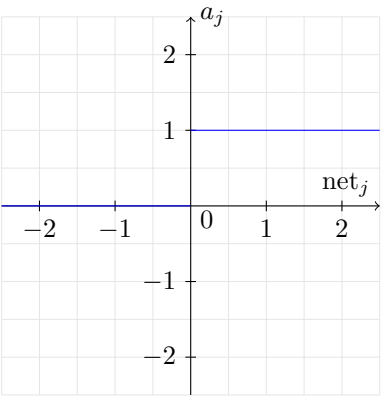
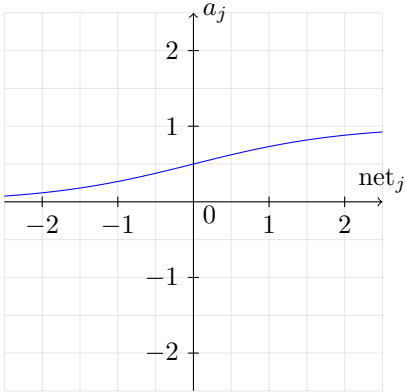
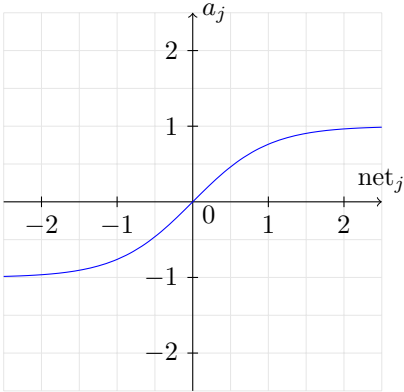
Lineare Aktivierungsfunktion	ReLu (Rectified Linear Unit)	Binäre Aktivierungsfunktion
Linearer Zusammenhang zwischen dem Netinput net_j und dem Aktivitätslevel a_j . Weder nach unten noch nach oben ist der Wertebereich für den Aktivitätslevel beschränkt.	Lineare Aktivierungsfunktion mit Schwelle 0. Es ist erforderlich, dass die festgelegte Schwelle 0 überschritten wird, erst dann ist der Zusammenhang zwischen den beiden Größen linear.	Häufig wird diese Funktion auch Heavyside-Funktion, Schwellenwertfunktion oder Schrittfunktion genannt. Der Aktivitätslevel kann lediglich zwei Zustände annehmen, nämlich 0 (in manchen Fällen auch -1) oder +1.
$a_j = m \cdot net_j + b$	$a_j = \max(0, net_j)$	$a_j = \begin{cases} 1 & \text{falls } net_j \geq 0 \\ 0 & \text{falls } net_j < 0 \end{cases}$
 <p>mit $m = 1$ und $b = 0$</p>		

Tabelle 3.1: lineare Aktivierungsfunktion, ReLu und die binäre Aktivierungsfunktion

Logistische Aktivierungsfunktion	Tangens Hyperbolicus Aktivierungsfunktion
Der Wertebereich ist auf 0 bis +1 begrenzt. Werden große negative Werte (z.B. -100) für net_j eingesetzt, ist der Aktivitätslevel nahe 0. Mit zunehmenden Netzinput steigt der Graph zunächst langsam, wird dann immer steiler (sodass er dabei zwischenzeitlich einer linearen Funktion gleicht) und nähert sich daraufhin wieder asymptotisch dem Wert +1 an.	Diese Funktion wird häufig abgekürzt tanh Aktivierungsfunktion genannt und verläuft ähnlich wie die logistische Aktivierungsfunktion. Der Unterschied besteht jedoch vor allem darin, dass der Wertebereich zwischen -1 und +1 liegt.
$a_j = \frac{1}{1+e^{-k \cdot net_j}}$	$a_j = \frac{e^{net_j} - e^{-net_j}}{e^{net_j} + e^{-net_j}}$
 <p>mit $k = 1$</p>	 <p>mit $\theta_j = 0$</p>

Dabei gilt:

m = Steigung der Geraden

b = Achsenschnittpunkt der Geraden

k = konstanter Faktor

Tabelle 3.2: Logistische und tanh Aktivierungsfunktion

Öfters wird für die Aktivierungsfunktion (vor allem für die binäre Aktivierungsfunktion) ein bestimmter Schwellenwert θ_j für das Neuron j mitberücksichtigt. Bevor das Neuron aktiviert wird, muss noch zusätzlich der Schwellenwert überschritten werden. Bei der Berechnung der Aktivierung kommt als zusätzlicher Schritt, dass der Schwellenwert von der Netzeingabe abgezogen wird (siehe Gl. 3.2).

$$a_j = \varphi(\text{net}_j - \theta_j) \quad (3.2)$$

Welche Aktivierungsfunktion gewählt wird, hängt von der Art des KNN bzw. vom konkreten Anwendungsproblem ab. Im Kapitel 5.5 soll nochmal darauf eingegangen werden.

3.2.3 Ausgangsfunktion

Bei natürlichen Neuronen ist es erforderlich, dass der Aktivitätslevel eine bestimmte Schwelle überschreitet, bevor das Neuron feuert. Diese Bedingung haben künstliche Neuronen nicht, dafür wird aber verlangt, dass der Ausgang mit zunehmender Aktivität nicht kleiner wird, d.h. die Ausgangsfunktion hat die Anforderung *monoton wachsend* zu sein. Für bestimmte Arten von KNN wie den hybriden Netzen ist eine klare Trennung zwischen der Aktivierungsfunktion und der Ausgangsfunktion erforderlich. In zahlreicher Literatur jedoch wird die Ausgangsfunktion von den Autoren als Bestandteil der Aktivierungsfunktion angesehen und daher gar nicht erwähnt. Im Grunde genommen wird dann also für die Ausgangsfunktion die Identitätsfunktion (siehe Gl. 3.3) eingesetzt, auch für diese Arbeit soll sie verwendet werden.

$$o_j = f_{\text{out}(a_j)} = a_j \quad (3.3)$$

4 Struktur der KNN

4.1 Aufbaustruktur

Der Aufbau eines KNN lässt sich als ein Netz künstlicher Neuronen beschreiben, die im Prinzip beliebig über gerichtete und gewichtete Verbindungen miteinander verknüpft sind. Als Netzwerkgraph kann das so aussehen, dass dessen Knoten die Neuronen und die Kanten die Verbindungen repräsentieren sollen. Aufgrund dieser Darstellungsmöglichkeit kann die mathematische Definition von KNN folgendermaßen lauten: Ein KNN „besteht aus einer Menge von Neuronen $N = \{n_1, \dots, n_m\}$ und einer Menge von Kanten $K = \{k_1, \dots, k_p\}$ mit $K \subset N \times N$ “ (Scherer 1997: S.54).

Die Struktur und Anordnung der künstlichen Neuronen innerhalb des KNN wird als *Topologie* definiert. Welche Topologie verwendet werden soll, hängt vom konkreten Anwendungsproblem ab. Es lassen sich zwei Grundtypen von Netzen unterscheiden:

- Zum einen die *rückgekoppelten Netze* (bzw. *rekurrente Netzwerke* oder *Feedback Netzwerke*), bei denen die Ausgabewerte auf die Eingabe zurückgeführt werden können. Die Verbindungen können also in beide Richtungen verlaufen.
- Zum anderen die *vorwärts gerichteten Netze* (bzw. *Feedforward Netzwerke*), dessen berechneten Ausgabewerte keinen Einfluss auf die Eingabe haben. Hier können die Verbindungen also nur in eine Richtung gehen, nämlich von der Eingabe zur Ausgabe.

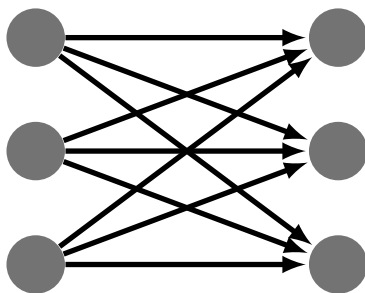


Abbildung 4.1: Schematisches Beispiel
Feedforward Netzwerk

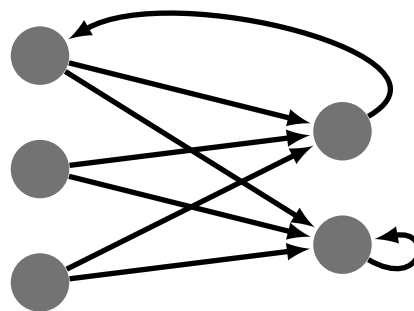


Abbildung 4.2: Schematisches Beispiel
Feedback Netzwerk

Auf erstere soll nicht mehr näher eingegangen werden, zwar sind sie aufgrund ihrer größeren Leistungsfähigkeit im Vergleich zu den vorwärts gerichteten Netzen interessanter, doch dafür deutlich komplexer.

Bei den Feedforward Netzwerken können drei verschiedene Arten von Neuronen klassifiziert werden:

- Zum einen gibt es die *Input-Neuronen* (bzw. *Eingangsneuronen*), die Eingabesignale von der Außenwelt zum Beispiel in Form von Reizen und Mustern bekommen.
- Desweiteren gibt es die *Hidden-Neuronen*, die zwischen den Input- und Output-Neuronen stehen.
- Als letztes sind die *Output-Neuronen* (bzw. *Ausgangsneuronen*) aufzuführen, die die Aufgabe haben, Signale an die Außenwelt auszugeben und auszuwirken.

Häufig sind bei Feedforward Netzwerken die Neuronen in mehrere *Schichten* eingeteilt. Es gibt keine verbindlichen Regeln für die Einteilung eines Netzes in Schichten, für gewöhnlich werden die Neuronen zusammengefasst, die gemeinsam eine bestimmte Aufgabe durchführen. Beispielsweise bei der Abbildung 4.3 ist das Kriterium für die Einteilung die Verbindungen der Neuronen. Dabei lassen sich die Input-Neuronen 1,...,3 zur Eingabeschicht und die Neuronen 11 und 12 zur Ausgabeschicht zusammenfassen. Dazwischen bilden die Neuronen 4,...,7 und die Neuronen 8,...,10 jeweils eine versteckte Schicht.

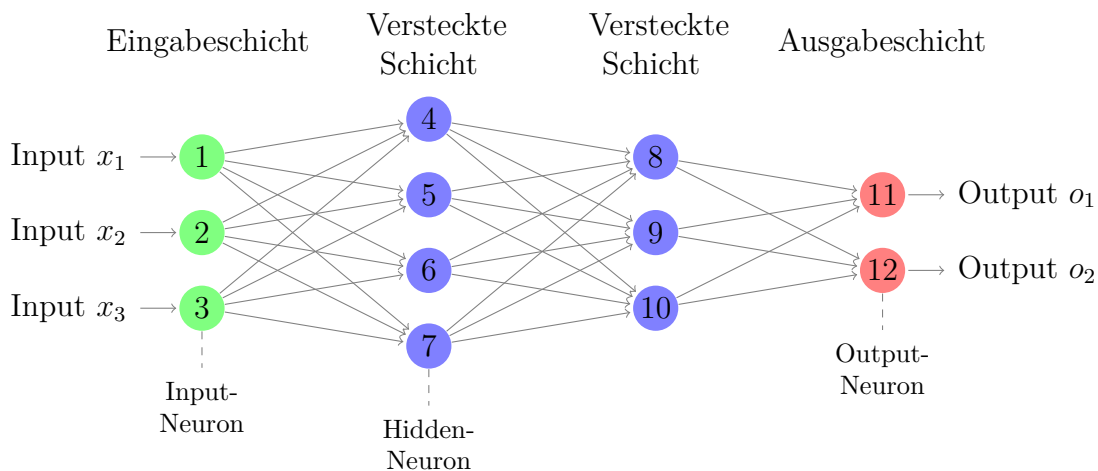


Abbildung 4.3: Beispiel Feedforward Netzwerk, modifizierte Version aus <http://www.texample.net/tikz/examples/neural-network/>

4.2 Einlagiges Perzeptron

Die erste Fassung eines Perzeptrons wurde 1958 von Frank Rosenblatt vorgestellt. Eine vereinfachte Version davon, das *einfache Perzeptron*, das nur aus einem künstlichen Neuron besteht, ist als simpelstes Beispiel für die KNN zu nennen. Es zählt zu den *einlagigen Perzeptronen* (eng. *single-layer perceptron*). Das einlagige Perzeptron stellt ein Feedforward Netzwerk dar, das nur zwei Schichten, die Ein- und Ausgabeschicht enthält, die beide jeweils beliebig viele Neuronen enthalten können. Es gibt keine versteckte Schichten, daher sind alle Neuronen der Eingabeschicht mit denen der Ausgabeschicht direkt verbunden. Im folgenden Unterkapitel soll erläutert werden, was mit einfachen Perzeptronen unter anderem möglich ist.

4.2.1 Darstellung der booleschen Funktionen als einfaches Perzeptron

Mit dem bisher erworbenen Wissen lassen sich einfache boolesche Funktionen wie die Konjunktion, die Disjunktion und die Negation als einfache Perzeptronen darstellen. Es soll davon ausgegangen werden, dass als Übertragungsfunktion die Summenfunktion (siehe Gl. 3.1), als Aktivierungsfunktion die binäre Aktivierungsfunktion (siehe Tabelle 3.1) und als Ausgangsfunktion die Identitätsfunktion (siehe Gl. 3.3) genommen werden sollen. Die AND-Funktion und die OR-Funktion können dann zum Beispiel folgende Werte für die Gewichte und Schwellenwerte zugewiesen bekommen (siehe Abbildungen 4.4 und 4.5).

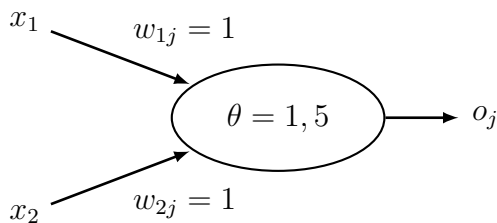


Abbildung 4.4: Repräsentation der Konjunktion $x_1 \wedge x_2$ als einfaches Perzeptron

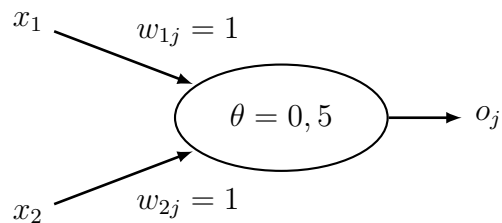


Abbildung 4.5: Repräsentation der Disjunktion $x_1 \vee x_2$ als einfaches Perzeptron

Zum besseren Verständnis soll die Abbildung 4.4 genauer erläutert werden. Da es sich um eine boolesche Funktion handelt, werden für x_1 und x_2 nur Werte aus der Menge $\{0, 1\}$ zugelassen. Um festzustellen, ob das Neuron j zum Beispiel für $x_1 = 1$

und $x_2 = 1$ feuert oder nicht, wird folgendes berechnet:

$$o_j = a_j = \varphi(\text{net}_j - \theta_j) \quad (4.1)$$

$$= \varphi((x_1 \cdot w_{1j} + x_2 \cdot w_{2j}) - \theta_j) \quad (4.2)$$

$$= \varphi((1 \cdot 1 + 1 \cdot 1) - 1,5) = \varphi(0,5) \quad (4.3)$$

Anhand der binären Aktivierungsfunktion sollte für o_j schließlich herauskommen:

$$o_j = \varphi(0,5) = 1 \quad (4.4)$$

Damit wurde festgestellt, dass das Neuron feuert. Bei den anderen möglichen Kombination der binären Eingabeparameter x_1 und x_2 sollte dagegen immer 0 herauskommen.

4.2.2 Klassifizierung und lineare Separierbarkeit

KNN werden häufig zur Klassifizierung genutzt. Dabei werden Klassen definiert, zu denen die verschiedenen Kombinationen der Eingabeparameter eingeordnet werden können. Im vorherigen Unterkapitel wurde also eingeteilt, welche Kombinationen der Eingabeparameter zur Klasse gehören, die das Neuron zum Feuern bringen und welche Kombinationen zu der Klasse, bei dem das Neuron dies nicht tut. Die *lineare Separierbarkeit* spielt für die Klassifizierung eine wesentliche Rolle. Unter linearer Separierbarkeit ist die Eigenschaft zu verstehen, zwei Mengen im n -dimensionalen Vektorraum durch eine *Hyperebene* voneinander trennen zu können. Die Dimension der Hyperebene beträgt $(n - 1)$, d.h. eine Dimension weniger als der Vektorraum, in dem es sich befindet. Beispielsweise ist im eindimensionalen Raum ein Punkt, im zweidimensionalen Raum die Gerade und im dreidimensionalen Raum die Ebene die Hyperebene.

Einlagige Perzeptronen sind in der Lage, separierbare Funktionen darzustellen, wobei die Dimension n sich aus der Anzahl der Eingabeparameter ergibt. Zur Veranschaulichung soll das Perzeptron bzw. Neuron aus Abbildung 4.4 geometrisch in einem zweidimensionalen Koordinatensystem interpretiert werden. Zweidimensional deshalb, weil es zwei verschiedene Eingabeparameter gibt. Dazu wird angenommen, dass für die Eingabeeinheiten nicht nur die Menge $\{0, 1\}$ zugelassen wird, stattdessen soll $x_1, x_2 \in \mathbb{R}$ gelten.

Da für die Funktion 4.2 die binäre Aktivierungsfunktion verwendet wird, muss folgende Ungleichung erfüllt werden, damit $o_j = 1$ ergibt:

$$(x_1 \cdot w_{1j} + x_2 \cdot w_{2j}) - \theta_j \geq 0 \quad (4.5)$$

4 Struktur der KNN

Durch Äquivalenzumformungen ergibt sich daraus:

$$x_1 \geq -\frac{w_{1j}}{w_{2j}} \cdot x_1 + \frac{\theta}{w_{2j}} = \frac{1}{w_{2j}}(\theta - x_1 w_{1j}) \quad (4.6)$$

Unter der Annahme, dass x_1 und x_2 eine Ebene bilden, kann die Gleichung 4.6 als die sich in der Ebene ergebene Gerade bzw. Hyperebene sehen, die die Ebene in zwei Bereiche aufteilt. Oberhalb der Gerade befindet sich die Menge aller Punkte deren Kombination von x_1 und x_2 zu $o_j = 1$ und damit zum Feuern des Neurons führen, sofern w_{2j} positiv ist. Logischerweise befindet sich dann unterhalb der Gerade die Menge aller Punkte, deren Kombinationen von x_1 und x_2 zu $o_j = 0$ führen.

Die Abbildung 4.6 veranschaulicht grafisch die Trennung durch die Hyperebene. Für die Bildung der Geraden wurde die Geradengleichung 4.6 genommen und die aus Abb. 4.4 entnommenen Werte $w_{1j} = 1, w_{2j} = 1$ und $\theta = 1,5$ eingesetzt. Der blaue Bereich stellt den Bereich dar, in dem das Neuron feuert. Bei der Betrachtung der Punkte, die aus den binären Eingabemöglichkeiten resultieren, ist zu sehen, dass der Punkt P_0 (1/1) in dem Bereich liegt, in dem das Neuron feuert und die Punkte P_1 (0/1), P_2 (0/0), P_3 (1/0) in dem Bereich, in dem das Neuron nicht feuert. Bei der Abbildung 4.7 wurde in die Geradengleichung $w_{1j} = 0.4, w_{2j} = 0.25$ und $\theta = 0,5$ eingesetzt.

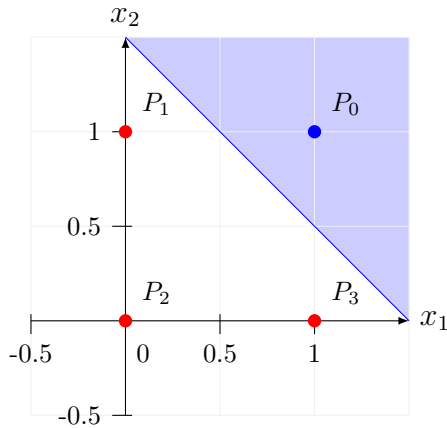


Abbildung 4.6: Beispiel 1 für die lineare Separierbarkeit der AND-Funktion

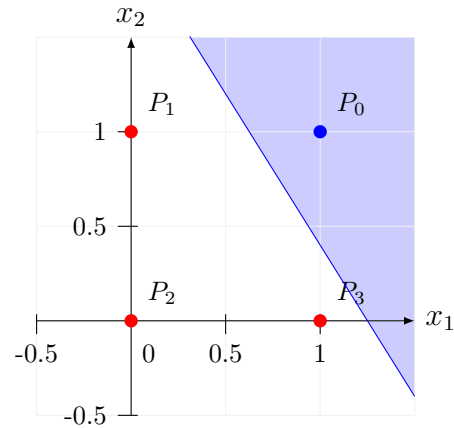


Abbildung 4.7: Beispiel 2 für die lineare Separierbarkeit der AND-Funktion

4.2.3 Das XOR-Problem

Geht es darum, Funktionen darzustellen, die nicht linear separierbar sind, stoßen die einlagigen Perzeptronen an ihre Grenzen.ⁱ So kann zum Beispiel die boolesche XOR-Funktion nicht als einlagiges Perzeptron repräsentiert werden. Wie die Abbildung 4.8 zeigt, reicht für korrekte Klassifizierung der Punkte P_1, \dots, P_4 nur eine Hyperebene nicht aus und es werden mindestens zwei Hyperebenen benötigt. Es ist daher ein komplexeres KNN als das einlagige Perzeptron nötig, um das Problem zu lösen.

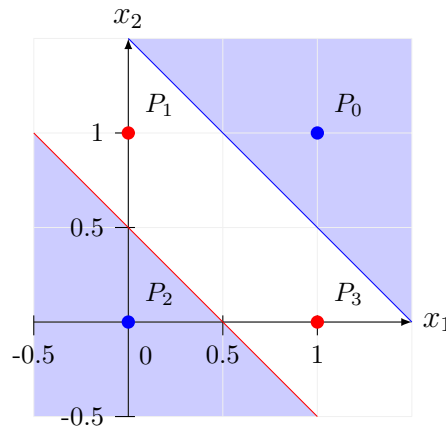


Abbildung 4.8: Die OR-Funktion ist nicht linear separierbar

4.3 Mehrlagiges Perzeptron

Das mehrlagige Perzeptron (MLP) (auch als Multilayer Perzeptron bekannt) enthält neben der Ein- und Ausgabeschicht zusätzlich noch mindestens eine versteckte Schicht, wobei jede versteckte Schicht im Prinzip unendlich viele Neuronen enthalten kann. Das in Abbildung 4.1 dargestellte Feedforward Netzwerk zeigt wie ein MLP aussehen kann. Eine Eigenschaft des MLP ist, dass in der Eingabeschicht und in jeder versteckten Schicht jedes Neuron i jeweils mit einem bestimmten Gewicht w_{ij} zu jedem Neuron j der darauffolgenden Schicht verbunden sein muss. Das MLP ist in der Lage, Funktionen zu repräsentieren, die nicht linear separierbar sind und daher komplexere Problemstellungen zu lösen.

ⁱDies wurde von den Kritikern *Marvin Minsky* und *Seymour Papert* 1969 in deren Buch *Perceptrons* (Minsky & Papert 1969) beschrieben, das eine Reihe an mathematischen Beweisen von Perzeptronen enthält, die unter anderem die Grenzen dieser aufzeigen. Die daraus gewonnenen Erkenntnisse führten dazu, dass das Interesse an KNN deutlich abnahm und daher die Forschungsgelder dafür gestrichen wurden. Erst ab 1985 brachten neue Erkenntnisse wieder das Forscherinteresse an KNN deutlich zum Steigen.

4.3.1 Darstellung der XOR-Funktion als MLP

Um das XOR-Problem zu lösen, gibt es verschiedene Möglichkeiten wie das KNN aufgebaut sein kann, z.B. das XOR-Netzwerk stellt eine Lösungsmöglichkeit dar. Die hier vorgestellte Lösungsmöglichkeit soll durch ein MLP erfolgen, das folgendermaßen gebildet werden kann: Bei der Betrachtung der Abbildung 4.8 kann die blaue Hyperebene durch das Perzeptron aus Abbildung 4.4 und die rote Hyperebene durch das Perzeptron aus 4.5 dargestellt werden. Beide Perzeptronen lassen sich so kombinieren, dass deren Ausgaben von einem dritten Perzeptron als Eingangssignale empfangen werden. Dieser verarbeitet die Eingangssignale und bestimmt anschließend, ob das Ausgangssignal o_j den Wert 1 oder den Wert 0 zugewiesen bekommt. Für o_j soll erst 1 herauskommen, wenn vom Perzeptron der OR-Funktion das Ausgabesignal 1 und vom Perzeptron der AND-Funktion das Ausgabesignal 0 gesendet wird. Die Kombination dieser drei Perzeptronen ist in Abb. 4.9 dargestellt.

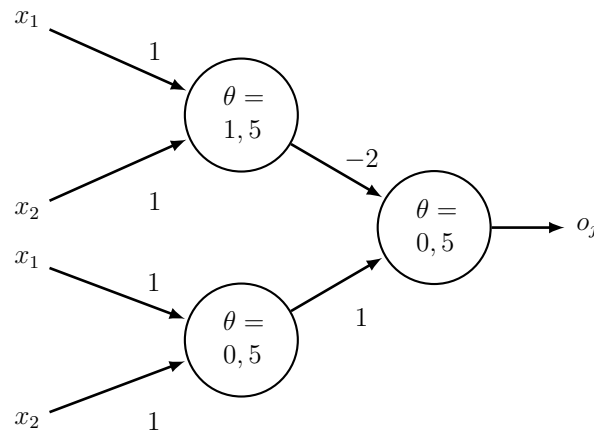


Abbildung 4.9: Durch die Kombination von drei Perzeptronen, kann die XOR-Funktion gelöst werden.

Zur besseren Übersicht werden häufig die Input-Neuronen verwendet, dabei ist für jedes Eingabesignal jeweils ein Input-Neuron zuständig. Da in diesem Beispiel zwei Eingabesignale x_1 und x_2 existieren, gibt es daher zwei Input-Neuronen. Ein Perzeptron, das in der Lage ist, das XOR-Problem zu lösen, kann mit Input-Neuronen aus fünf Neuronen bestehen und wie in Abbildung 4.10 aussehen.

4.3.2 Bias-Neuronen

Das Bias-Neuron, auch on-Neuron genannt, ist eine Alternative, den Schwellenwert eines Neurons festzulegen. Er stellt einen weiteren Eingang x_0 für das Neuron j dar und hat immer den konstanten Wert 1. Der negative Schwellenwert bildet dabei die Gewichtung, also $w_{0j} = -\theta$. Der praktikable Vorteil darin liegt, dass der Schwellenwert

4 Struktur der KNN

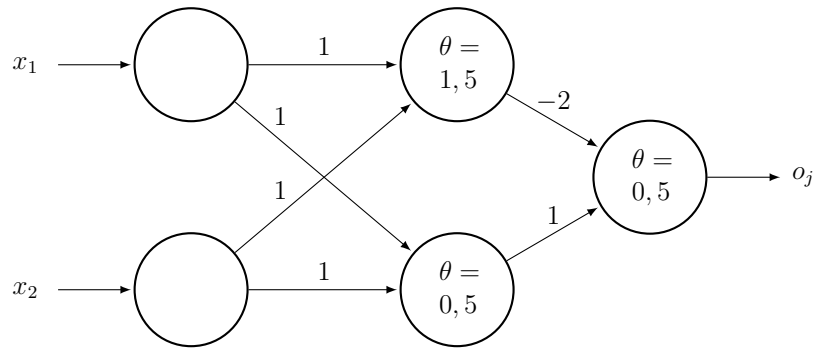


Abbildung 4.10: MLP, der die XOR-Funktion lösen kann ohne Bias-Neuronen

nicht mehr in der Aktivierungsfunktion enthalten ist, daher kann sie die Aktivierungsfunktion nicht mehr verändern. Zur Veranschaulichung der Bias-Neuronen kann die Abbildung 4.11 betrachtet werden.

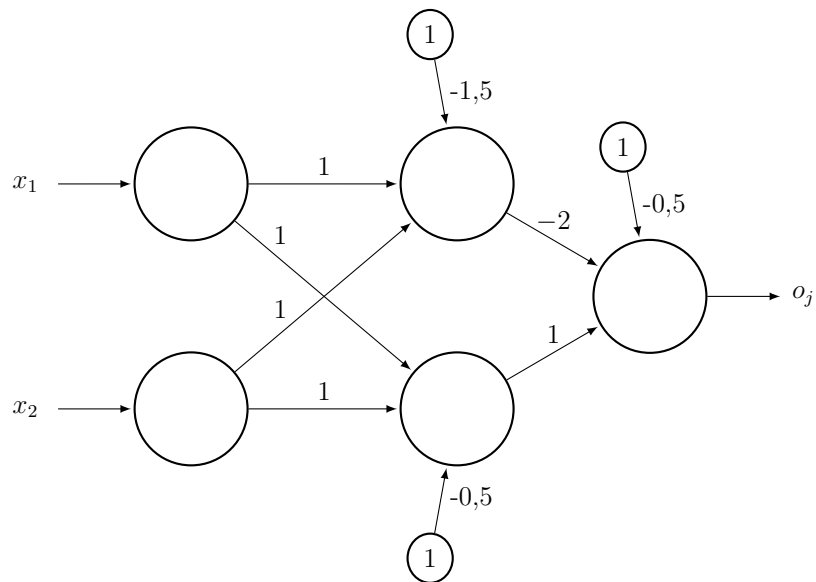


Abbildung 4.11: MLP, der die XOR-Funktion lösen kann mit Bias-Neuronen

5 Lernen in KNN

In den vorherigen Beispielen wie Abbildung 4.4 oder Abbildung 4.10 sind bereits Netze mit den passenden Gewichten vorgegeben worden, die imstande sind, die zugehörige Problemstellung zu lösen. Das Interessante an KNN ist, dass sie meist zu Beginn noch nicht alle erforderlichen Informationen besitzen, um das Problem zu lösen. Stattdessen sind sie in der Lage selbständig aus Beispielen zu *lernen*, ein passendes Netz zur Lösung der Problemstellung zu bilden. In der Praxis wird der Vorgang des Lernens häufig auch als *Training* bezeichnet. Das Prinzip aus Beispielen zu lernen, entspricht dem des maschinellen Lernens. Es gibt viele mögliche Arten des Lernens bei KNN, zum Beispiel das Löschen existierender Verbindungen oder Neuronen oder auch das Entwickeln neuer Verbindungen oder Neuronen.

5.1 Lernen durch Modifikation der Gewichte

Vor allem hat es sich etabliert, das Lernen in KNN durch das Verändern der Werte für die Gewichte w_{ij} erfolgen zu lassen. Anhand einer vorher definierten *Lernregel* werden die Gewichte solange modifiziert, bis die Problemstellung zufriedenstellend gelöst werden kann. Um die neuen Gewichtswerte zu bestimmen, wird durch die Lernregel für jedes Gewicht w_{ij} ein Fehlerkorrekturwert Δw_{ij} bestimmt, der zum alten Gewichtswert hinzuaddiert wird. Als Formel ausgedrückt:

$$w_{ij}(\text{neu}) = w_{ij}(\text{alt}) + \Delta w_{ij} \quad (5.1)$$

5.2 Lernarten

Es lassen sich drei Lernarten von KNN unterscheiden, je nachdem wie die erwartete Ausgabe aussehen soll. Zur einfacheren Formulierung soll im Folgenden bei den verschiedenen Kombinationsmöglichkeiten der verschiedenen Eingabewerte von Eingabemustern und analog dazu bei den Ausgabewerten von Ausgabemustern gesprochen werden.ⁱ

ⁱVeranschaulicht an einem Beispiel führt bei der AND-Funktion das Eingabemuster $\{1, 1\}$ zum Ausgabemuster $\{1\}$ und das Eingabemuster $\{0, 1\}$ zum Ausgabemuster $\{0\}$.

- *Überwachtes Lernen* (engl. *supervised learning*): Die Trainingsdaten bestehen aus einer Menge an Paaren von Eingabe- und Ausgabemustern $\{(E_p, O_p) | p \in \mathbb{N}^+\}$, d.h. zu jedem Eingabemuster ist das erwünschte Ausgabemuster bekannt. Jedes Paar bzw. jeder Trainingsdatensatz kann als Muster (engl. pattern) p bezeichnet werden.
- *Bestärkendes Lernen* (engl. *reinforcement learning*) Hier besteht die Trainingsmenge aus einer Menge an Eingabemustern, zu denen jeweils nur bekannt gegeben wird, ob sie richtig oder falsch klassifiziert worden sind. Heißt also zu jedem aus dem Eingabemuster berechneten Ausgabemuster wird nur angegeben, ob das Ausgabemuster stimmt oder nicht. Wie das korrekte Ausgabemuster aussieht, muss das Netz selbst herausfinden.
- *Unüberwachtes Lernen* (engl. *unsupervised learning*) Bei dieser Art besteht die Trainingsmenge lediglich aus einer Menge von Eingabemustern. Dem Netz ist also unbekannt, welche Ausgabemuster herauskommen sollen und es entscheidet selber, welche Klassen sich aus den ausgerechneten Ausgabemustern ergeben. So können ähnliche Eingabemuster in ähnliche Klassen von Ausgabemustern eingeteilt werden.

Von den vorherigen vorgestellten Lernarten, ist das überwachte Lernen, die schnellste Art, das Netz zu trainieren. Die beiden anderen Lernarten sind dafür biologisch plausibler, da nicht die erwünschten Ausgabemuster (bzw. Aktivierungen) bekannt sein müssen.ⁱⁱ Deren Nachteile sind jedoch, dass sie deutlich langsamer sind als das überwachte Lernen.

5.3 Lernverfahren

Bei dem Lernprozess ist es weiterhin wichtig, das Verfahren zu bestimmen, welches festlegt, wann das KNN die Gewichte verändern soll:

- *batch- oder offline-Trainingsverfahren*: Bei diesem Verfahren muss dem System alle Muster p präsentiert werden, bevor alle Gewichte in einem Schritt modifiziert werden. Die Muster werden also stapelweise verarbeitet.
- *online-Trainingsverfahren*: Bei diesem Verfahren erfolgt die Änderung der Gewichte direkt nach Darbietung eines Musters p aus den Trainingsdaten.

ⁱⁱBeispiel

5.4 Lernregel

Im Folgenden sollen Lernregeln vorgestellt werden, die vor allem für das überwachte Lernen eine Rolle spielen. Auf die genaue Herleitung der Regeln wird jedoch verzichtet, bei Interesse sei das Buch *Simulation neuronaler Netze* von *Andreas Zell* zu empfehlen (Zell 1994: S.105-110).

5.4.1 Hebbsche Lernregel

Die älteste und einfachste Regel, die für jede der drei Lernarten verwendet werden kann, ist die Hebbsche Lernregel. Anzumerken ist, dass sie lediglich auf KNN anwendbar ist, die keine versteckten Schichten besitzen. Dennoch ist sie aus dem Grunde interessant, weil es sich bei den darauf folgenden Lernregeln um Spezialisierungen der Hebbschen Lernregel handelt. Basieren tut sie in einer abgewandelten Form auf einer Vermutung vom Psychologen Donald Olding Hebb über die Veränderung von Synapsen in Nervensystemen (Hebb 1949). Die Regel besagt, dass wenn das Empfänger Neuron j vom Sender Neuron i eine Eingabe erhält und dabei beide gleichermaßen aktiv sind, dann das Gewicht w_{ij} erhöht wird (also die Verbindung von i nach j verstärkt). Die Formel für das offline-Trainingsverfahren soll der Einfachheit halber nicht gezeigt werden, stattdessen soll nur die für das online-Trainingsverfahren aufgeführt werden:

$$\Delta w_{ij} = \eta o_i a_j \quad (5.2)$$

Dabei gilt

- Δw_{ij} : Änderung des Gewichts w_{ij}
- η : Lernrate, ein positiver konstanter Wert, der meist kleiner als 1 ist
- o_i : Ausgabe der Sender-Neurons i
- a_j : Aktivierung des Empfänger-Neurons j

Werden für o_i und a_j nur die binären Werte 0 und 1 zugelassen (wie ursprünglich bei der Hebbschen Regel angedacht), verändert sich w_{ij} nur, wenn beide den Wert 1 haben, bei den anderen Fällen findet keine Veränderung statt. Dann wird das Gewicht erhöht, eine Verringerung ist also nicht mehr möglich. Um dies zu umgehen, bietet es sich an, für o_i und a_j stattdessen die binären Werte -1 und +1 zuzulassen. In diesem Fall wird das Gewicht erhöht, wenn o_i und a_j identisch sind (jeweils +1 oder -1). Sind sie dagegen unterschiedlich, so werden die Gewichte verringert.

5.4.2 Delta-Regel

Die Delta-Regel (auch Widrow-Hoff-Regel genannt) kann für Netze verwendet werden, die das überwachte Lernverfahren nutzen. Die mathematische Formel für das online-Trainingsverfahren lautet:

$$\Delta w_{ij} = \eta o_i \delta_j \quad (5.3)$$

mit

$$\delta_j = t_j - a_j \quad (5.4)$$

Dabei gilt

- t_j : teaching input, erwünschte Aktivierung
- δ_j : Differenz der aktuellen Aktivierung a_j und der erwünschten Aktivierung t_j .
- alle anderen Variablen sind analog zur vorher vorgestellten Hebb-Regel

Anhand der Differenz von erwünschter und berechneten Aktivierung wird ein Deltawert δ_j bestimmt (siehe Gl. 5.4). Die Gewichtsänderung Δw_{ij} soll zur Größe des Fehlers δ_j proportional sein (siehe Gl. 5.3).

Anwendbar ist die Delta-Regel ebenfalls nur bei KNN ohne versteckte Schichten, da nur die Ausgabewerte für die Output-Neuronen bekannt sind, aber nicht die für die Hidden-Neuronen. Dafür kann sie im Gegensatz zur Hebbschen Lernregel reellwertige Ausgaben erlernen.

5.4.3 Backpropagation-Regel

Mithilfe der Backpropagation-Regel ist es möglich, mehrlagige Perzeptronen zu trainieren. Bevor darauf eingegangen wird, wie die Backpropagation-Regel funktioniert, soll das *Gradientenabstiegsverfahren* vorgestellt werden, die für die Regel von Bedeutung ist.

Gradientenabstiegsverfahren

Um zu bestimmen, wie groß die Differenz zwischen erwarteter und tatsächlich berechneter Ausgabe ist, kann eine sogenannte Fehlerfunktion (siehe Gl. 5.5) bestimmt werden.

$$E(W) = E(w_1, \dots, w_n) \quad (5.5)$$

Es gilt

- \mathbf{W} Gewichtsvektor, dessen Einträge aus w_1 bis w_n bestehen

Anhand derer lässt sich bestimmen, wie groß der aufsummierte Fehler des KNN ist, wenn eine bestimmte Kombination der Gewichte w_1, \dots, w_n über alle Muster der Trainingsdaten erfolgt. Graphisch kann mit zwei Gewichten w_1 und w_2 die Funktion wie in Abbildung 5.1 dargestellt werden.

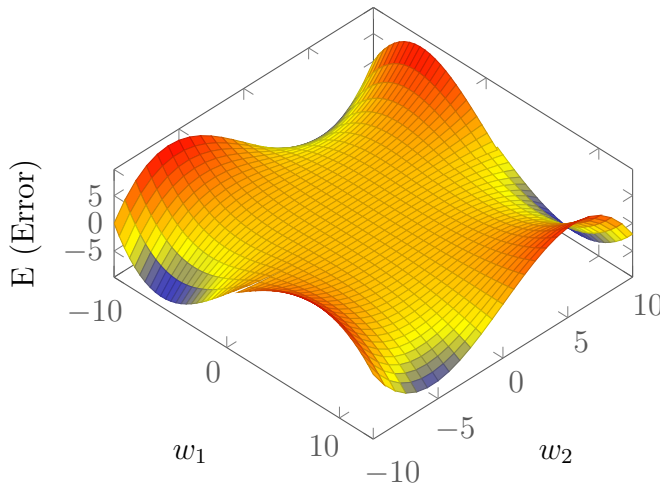


Abbildung 5.1: Bei zwei verschiedenen Gewichten kann die Fehlerfunktion E als Fläche dargestellt werden

Zur Veranschaulichung der Grafik wird oft folgendes Beispiel vorgestellt, um die Aufgabe des Gradientenabstiegsverfahrens klarzustellen: Eine Person befindet sich in einer Hügellandschaft, die so nebelig ist, dass er nur die unmittelbare Umgebung sehen kann. Ihr Ziel ist es, das tiefste Tal in der Landschaft zu finden. Um an ihr Ziel zu kommen, startet sie zunächst an einem beliebigen Startpunkt und prüft dort, in welcher Richtung es am steilsten bergab geht. Da sich die Person in einer 3-dimensionalen Landschaft befindet, muss sie sich daher komplett um die eigene Achse drehen, um den steilsten Abstieg zu finden. Sie folgt einige Meter dem steilsten Abstieg, prüft an der neuen Stelle, ob die Richtung des steilsten Abstiegs sich geändert hat und korrigiert ihre Laufrichtung gegebenenfalls danach. Dies wiederholt sie solange bis sie zu einer Stelle kommt, von dem es nicht mehr bergab geht oder bis sie zu erschöpft ist, um weiterzulaufen.

Das Beispiel zeigt eine ungefähre Vorstellung wie das Gradientenabstiegsverfahren für die KNN funktioniert. Das tiefste Tal stellt für den Fehlerterm das *globale Minimum* dar, um so die Gewichtungskombination zu finden, die zum geringsten Fehler zwischen erwarteter und tatsächlicher Ausgabe führt. Allgemein zeigt der sogenannte *Gradient* einer Funktion f im Punkt x in Richtung des steilsten Anstiegs einer Funktion f . Da aber in Richtung des steilsten Abstiegs gegangen werden soll, ist der negative Gradient der Fehlerfunktion (siehe Gl.5.5) zu bestimmen. Die Modifizierung der Gewichte,

um den Fehler zu minimieren, soll um einen Bruchteil des negativen Gradienten der Fehlerfunktion erfolgen (siehe Gl. 5.6). Übertragen auf das Beispiel soll damit festgelegt werden, wie weit die Person in Richtung des steilsten Abstiegs laufen soll.

$$\Delta W = -\eta \nabla E(W) \quad (5.6)$$

Es gilt

- η Lernrate (bzw. Lernfaktor oder Schrittweite), die vorher festgelegt wurde; meist liegt sie zwischen 0,005 und 0,75
- $\nabla E(W)$: Gradient der Funktion $E(W)$ ⁱⁱⁱ

Der Gradient der Fehlerfunktion ist im n -dimensionalen Raum (n = Anzahl der Gewichte + 1) ein $(n-1)$ -dimensionaler Vektor. Die Einträge dieses Vektors bildet sich aus den ersten partiellen Ableitungen der Funktion. Der erste Eintrag ist die partielle Ableitung des ersten Gewichtes w_1 , der zweite Eintrag der des zweiten Gewichtes w_2 usw. Für die Änderung eines einzelnen Gewichtes w_{ij} gilt daher:^{iv}

$$\Delta w_{ij} = -\eta \frac{\partial E(W)}{\partial w_{ij}} \quad (5.7)$$

Zur Bestimmung des Gesamtfehlers E werden die Fehler über alle Muster p aufsummiert (siehe Gleichung 5.8) und durch P , die Anzahl der Muster, geteilt.

$$E = \frac{1}{P} \sum_p E_p \quad (5.8)$$

Eine wichtige Information, um E zu bestimmen, ist die Wahl der *Kostenfunktion*, die für das gesamte KNN gilt. Für diese Arbeit soll als Kostenfunktion die *mittlere quadratische Abweichung*, abgekürzt *MSE* (aus dem Englischen *mean squared error*), benutzt werden, welche aus Gründen der Übersichtlichkeit, nicht in dieser Arbeit aufgeführt und erläutert werden soll. Die Wahl des MSE als Kostenfunktion führt dazu, dass zur Bestimmung von E_p sich folgende Gleichung ergibt:

$$E_p = \sum_j (t_{pj} - o_{pj})^2 \quad (5.9)$$

Es gilt

- t_{pj} : erwartete Ausgabe vom Neuron j beim Muster p
- o_{pj} : tatsächliche Ausgabe vom Neuron j beim Muster p

ⁱⁱⁱ ∇ wird nabla ausgesprochen

^{iv} ∂ steht für die partielle Ableitung

Der Schritt der Modifizierung der Gewichte und dem Berechnen des Gesamtfehlers erfolgt solange, bis der Gradient $\nabla = 0$ ist (Analogie: für die Person geht es ab der Stelle nicht mehr weiter bergab) oder bis eine bestimmte Anzahl an Iterationen/Wiederholungen dieses Schrittes erreicht wurde (Analogie: die Person ist zu erschöpft zum weitergehen). Beim Gradientenabstiegsverfahren ergeben sich viele Probleme (beispielsweise, dass oft nicht das globale, sondern das lokale Minimum gefunden wird oder dass gute Minima übersprungen werden können), teilweise gibt es schon Lösungsansätze für diese Probleme.^v

Der Backpropagation-Algorithmus

Das Problem der Delta-Regel ist, dass immer die erwünschte Ausgabe t bekannt sein muss, bei den Hidden-Neuronen sind diese aber in der Regel nicht gegeben. Um dieses Problem zu umgehen, wurde der Backpropagation-Algorithmus entwickelt, der die Trainingsphase/Lernphase jeder Gewichtsänderung in mehrere Schritte aufteilt:

1. Festlegen des Startpunktes: Der Startpunkt wird anhand einer zufällig ausgewählten Gewichtungskombination bestimmt.
2. Trainingsmuster auswählen
3. Forward-pass: Dem KNN werden Eingabemuster präsentiert und daraus die Ausgabemuster des Netzwerks berechnet. Von der ersten bis zu letzten versteckten Schicht sowie der Ausgabeschicht werden die Ausgaben zwischengespeichert.
4. Fehlerbestimmung: Es wird der Gesamtfehler E (siehe Gl. 5.8) zwischen erwünschter und tatsächlicher Ausgabe verglichen. Ist der Gesamtfehler gering (also überschreitet er keine gewisse Schwelle) oder ist die vorher festgelegte maximale Anzahl an Iterationen/Wiederholungen erreicht worden, kann die Trainingsphase abgebrochen werden, ansonsten wird zu Schritt 5 übergegangen.
5. Backward-pass: Durch die allgemeine (bzw. erweiterte) Delta-Regel (siehe Gl. 5.11) ist es möglich, sowohl für die Output- als auch für die Hidden-Neuronen ein δ_{ij} zu bestimmen. Anders als beim Forward-pass ist die Reihenfolge umgekehrt, angefangen wird bei der Ausgabeschicht, dann geht es zur letzten versteckten Schicht, dann zur vorletzten usw. bis zur ersten versteckten Schicht. Durch dieses Vorgehen ist es möglich, nach und nach die Gewichte w_{ij} mithilfe des Gradientenabstiegsverfahren zu bestimmen, da die Gewichtsänderung unter anderem von δ_{ij} abhängt. Dies wird durch die Formel für die Backpropagation-Regel deutlich:

$$\Delta_p w_{ij} = \eta o_{pi} \delta_{pj} \quad (5.10)$$

^vBei Interesse kann dies zum Beispiel bei (Rey & Wender 1982: S.43-48) nachgelesen werden.

mit

$$\delta_{pj} = \begin{cases} \varphi'(net_j)(t_{pj} - o_{pj}) & \text{falls } j \text{ ein Output-Neuron ist} \\ \varphi'(net_j) \sum_k \delta_{pk} w_{jk} & \text{falls } j \text{ ein Hidden-Neuron ist} \end{cases} \quad (5.11)$$

Es gilt

- k : Das nachfolgende Neuron k befindet sich eine Schicht über der Schicht des Neurons j

Alle Schritte ab Schritt 3 werden solange wiederholt, bis wie in Schritt 4 beschrieben, das Training unterbrochen werden kann.

Wie bereits erwähnt, verwendet die Backpropagation-Regel unter anderem das Gradientenabstiegsverfahren, deshalb ergeben sich dieselben Probleme und daher zahlreiche Varianten, die Backpropagation-Regel zu erweitern und zu modifizieren.

5.5 Auswahl der Aktivierungsfunktion

Binäre Aktivierungsfunktionen haben den Vorteil, dass sie schnell und einfach zu implementieren sind, aber dafür auch aufgrund ihrer mathematischen Eigenschaften nur für einfache Modelle von KNN in Frage kommen. Was ihnen fehlt, ist die Differenzierbarkeit an allen Stellen, was für die Backpropagation-Lernregel ein Problem ist. Dies ist bei genauerer Betrachtung der allgemeinen Delta-Regel 5.11 zu erkennen, die die Ableitung der Aktivierungsfunktion φ enthält. Differenzierbar und auch häufig eingesetzt sind sigmoide Aktivierungsfunktionen, zu denen die logistische Aktivierungsfunktion und die tanh Aktivierungsfunktion zählen. Der Trend geht jedoch immer mehr dazu über, die ReLu Funktion als Aktivierungsfunktion zu benutzen, die in der Praxis bei MLP mit sehr vielen versteckten Schichten effektiver ist.

6 Das ursprüngliche Programm von Koenecke

6.1 Problemstellung und Netzwerktopologie

Die Anwendung soll in der Lage sein, Klassifizierungsprobleme zu lösen, da diese zum Einstieg greifbarer sind als beispielsweise Regressionsprobleme. Zur Lösung dieser Problemstellung eignet es sich, als Topologie für das KNN das MLP festzusetzen. Bei jeder Initialisierung eines neuen MLP werden die Gewichtswerte zufällig vom Programm festgelegt. Jeder Datensatz besteht aus zwei Eingabewerten, denen eine Klasse zugeordnet wird, wobei es drei verschiedene Klassen gibt. Ziel des Trainings des KNN ist es, zu jeder Eingabemöglichkeit eine Klasse bestimmen zu können, indem ähnliche Eingabekombinationen gleich klassifiziert werden.

6.2 Technologie

Mit der Programmiersprache *Google Go (golang)* hatte Koenecke eine ausführbare Anwendung `mlpmain_windows_amd64.exe` erstellt, dessen Herzstück die Implementierung des KNN als Netzwerk-Bibliothek ist. Beim Ausführen der Anwendung wird ein Server zum Port 8080 gestartet. Der Server ist dafür zuständig, die komplexeren Berechnungen für das KNN im Hintergrund durchführen zu lassen.

Die grafische Oberfläche, im Folgenden soll hierfür der Fachbegriff Grafical User Interface (GUI) benutzt werden, wurde mit HTML, CSS und Javascript realisiert. Zur grafischen Darstellung von Daten wie die Topologie des KNN wurde die Javascript-Bibliothek D3.js verwendet, die extra darauf ausgelegt ist, Daten manipulieren und (interaktiv) visualisieren zu können. Die GUI bietet Konfigurationsmöglichkeiten und eine Visualisierung der Struktur und Ergebnisse des KNN.

Zur Kommunikation zwischen Backend und Frontend wurde ein Webserver als Schnittstelle mit `gopkg.in/igm/sockjs-go.v2/sockjs`, ein Paket zur Websocket Emulation, realisiert. Unter der Adresse <http://localhost:8080> im Webbrowser wird die GUI der Anwendung für die Daten der Netzwerk-Bibliothek aufgerufen.

6.3 Festgelegte Parameter

Um die Bedienung und die Übersichtlichkeit der GUI zu erleichtern, entschied sich Koenecke dafür, bestimmte Parameter festzulegen. Die Beschränkung auf zwei Eingabe- und drei Ausgabewerte ermöglicht eine zweidimensionale Visualisierung der Inhalte, eine dreidimensionale Visualisierung wäre deutlich aufwändiger gewesen. Zudem sind in seiner Anwendung als Aktivierungsfunktion eine sigmoide Funktion und als Lernregel die Backpropagation-Regel festgelegt, für die Lernrate wurde ebenfalls schon ein vordefinierter Wert gesetzt.

6.4 Aufbau der GUI

6.4.1 Netzwerktopologie

Das erste Element der Anwendung (siehe Abb. 6.1) dient zur Konfiguration der Topologie. Es ist möglich, bis zu 5 Hidden-Layer hinzuzufügen und sie auch wieder zu entfernen, zudem gibt es bei jedem Hidden-Layer die Möglichkeit, die Anzahl der Neuronen zwischen 1-9 festzulegen. Beim Drücken auf den Button *apply* wird direkt ein MLP bzw. Netzwerk mit der gewünschten Topologie als Graph visualisiert und kann wie in Abb. 6.2 aussehen. Die Dicke der Linien soll die Größe der Gewichtswerte andeuten, die Farbe der Linien, welches Vorzeichen die Gewichtswerte haben.

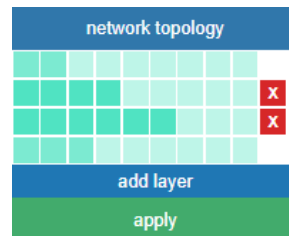


Abbildung 6.1: Konfiguration der Netzwerktopologie

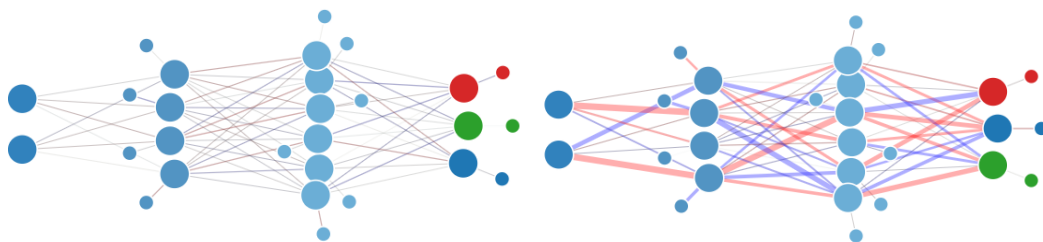


Abbildung 6.2: Darstellung eines Netzwerkgraphs (untrainiert)

Abbildung 6.3: Darstellung eines Netzwerkgraphs (trainiert)

6.4.2 Training

Erzeugung der Trainingsdaten

Zur Erzeugung der Trainingsdaten gibt es eine Fläche von 300×300 px, die als Koordinatensystem gesehen werden kann, auf der rote, grüne und blaue Punkte eingetragen werden können (siehe Abb. 6.4). Die Wahl auf diese Farben als Klassen fiel laut Koenecke aufgrund der dadurch gegebenen Möglichkeit, so optimal den RGB-Farbraum darstellen zu können. Jeder Punkt ist eine Darstellung eines Trainingsdatensatzes, bei dem die x- und die y-Koordinate die Eingabe und die Farbe die erwartete Ausgabe bzw. Klasse darstellt.

Trainieren des Netzwerks

Sind die Trainingsdaten festgelegt, kann über den Button *train network* das Training gestartet werden (siehe Abb. 6.5). Der Slider ermöglicht es, das Training zu beschleunigen oder zu verlangsamen. Beim Trainingsprozess verändern sich die Gewichtswerte, dies ist auch bei der Visualisierung des Netzwerkgraphs zu sehen, bei dem sich die Dicken der Linien ändern (siehe Abb. 6.3).

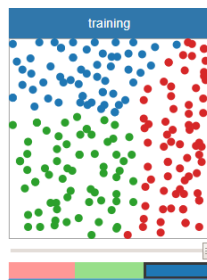


Abbildung 6.4: Erzeugung der Trainingsdaten

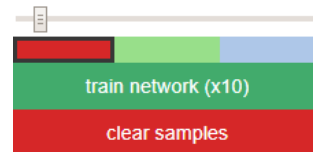


Abbildung 6.5: Start des Trainings mit 10-facher Geschwindigkeit

6.4.3 Präsentation der Netzwerkergebnisse

Vorschau der Ausgabe

Bei jedem Trainingsschritt wird für jeden Punkt des Eingabekoordinatensystems, d.h. für jede mögliche Eingabemöglichkeit der x- und der y-Koordinate,ⁱ drei Ausgabewerte berechnet, die in der Anwendung die RGB-Werte repräsentieren. Dadurch ergibt sich die Möglichkeit, die Trainingsergebnisse als Bild zu visualisieren (siehe z.B. Abb.

ⁱIn diesem Fall gibt es also $300 \cdot 300 = 90000$ Eingabemöglichkeiten.

6.6). Jeder Ausgabewert steht dafür, wie hoch die Wahrscheinlichkeit ist, dass der Punkt zu einer bestimmten Klasse bzw. Farbe gehört. Dadurch kann sich für einen Punkt eine Mischfarbe ergeben, wenn sie zwei ähnlich hohe Ausgabewerte hat, meist befindet sie sich in dem Falle nahe der Hyperebene (siehe z.B. Abb. 6.7, bei dem manche Punkte die Mischfarbe gelb-orange oder lila haben). Je mehr Trainingsschritte durchlaufen werden, desto eindeutiger kann jedem Punkt eine Farbe bzw. eine Klasse zugeordnet werden, was sich in der Vorschau dadurch bemerkbar macht, dass die Farben gesättigter und die Kanten schärfer sind (siehe z.B. Abb. 6.8).



Abbildung 6.6: Vorschau nach einer halben Million Beispieldaten



Abbildung 6.7: Vorschau nach einer Million Beispieldaten

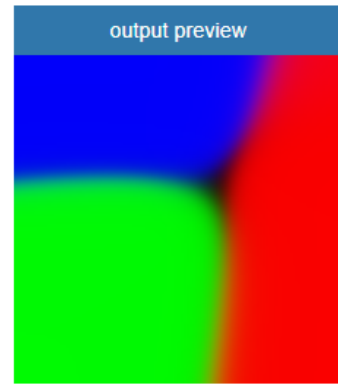


Abbildung 6.8: Vorschau nach drei Millionen Beispieldaten

Netzwerkinfo

Unter der Vorschau werden dem Anwender bestimmte Informationen des Netzes gezeigt (siehe Abb. 6.9). Die Information *samples total* gibt die Anzahl der bisher durchgeführten Trainingsschritte an, *sample coverage* wie viel Prozent der Trainingspunkte durch das KNN korrekt klassifiziert wurden und *mean weight change* wie hoch die durchschnittliche Gewichtsänderung ist.

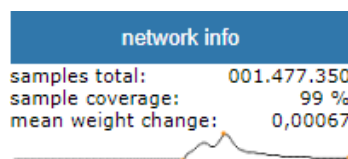


Abbildung 6.9: Netzwerkinfo während des Trainings

7 Eigene Implementierungen

7.1 Implementierung des KNN in Javascript

7.1.1 Wahl der passenden Javascript Bibliothek für das KNN

Bei der Recherche stellte sich heraus, dass es bereits einige Javascript Bibliotheken gibt, mit denen KNN erstellt und lokal im Browser trainiert werden können. In die engere Auswahl fielen *brain.js*, *Mind*, *Neataptic* und *Synaptic*. Für jede Bibliothek wurde ein Testprogramm geschrieben, um sie besser miteinander vergleichen zu können, wobei jedes Programm dieselben Trainingsbedingungen hat. Grob zusammengefasst besteht jeder Trainingsdatensatz aus zwei Punkten, denen eine Klasse zugeordnet wurde. Für das Training wird bei jedem eine bestimmte Topologie festgelegt und eine bestimmte Anzahl an Iterationen durchgeführt. Am Ende des Trainings sollte angegeben werden, wie die berechneten Ausgabewerte der Punkte aussehen und wie lange das Training gedauert hat. Die Testprogramme befinden sich in der CD im Ordner *NNBibTests*.ⁱ

Folgende Punkte waren wichtig für die Wahl der Bibliothek gewesen:

- die Möglichkeit, ein MLP erstellen zu können
- die Handhabung mit der Bibliothek (z.B. wie aufwändig ist es, den Code zur Erstellung des MLP oder des Trainings zu schreiben)
- die Geschwindigkeit der Berechnung
- die Effektivität des Trainings
- die Dokumentation der Bibliothek
- die Übersichtlichkeit und Struktur des Codes der Bibliothek
- zu einem geringen Teil die Aktivität an der Bibliothek, also wie oft die Bibliothek aktualisiert wurde und wann die letzte Aktualisierung stattfand

ⁱEbenfalls wurde aus Interesse zum Vergleich für die Bibliothek *ConvNetJS* ein Testprogramm geschrieben, die bei der Berechnung deutlich am präzisesten und schnellsten war, allerdings werden dort keine MLP zur Berechnung genutzt, sondern sogenannte *Convolutional Neural Networks*, eine besondere Form von Feedforward Netzen.

In der Handhabung konnten `brain.js` und `Mind` ganz gut punkten, so ist es schon mit wenigen Zeilen Code möglich, ein KNN zu erstellen. Allerdings sind die Konfigurationsmöglichkeiten des KNN mit den beiden Bibliotheken eher beschränkt und beide bieten eine eher dürftige Dokumentation. Bei `Neataptic` handelt es sich um eine Bibliothek, die für bestimmte Teile des Codes die `Synaptic` Bibliothek verwendet hat. Von allen betrachteten Bibliotheken bietet sie auch mit einem Wiki aus 30 Seiten die beste Dokumentation und bietet gegenüber `Synaptic` Extras wie die Visualisierung der Topologie des KNN, mehr implementierte Aktivierungsfunktion etc. Allerdings handelt es sich um Extras, die für den eigentlichen Zweck des Programms nicht notwendig sind. Schlussendlich fiel die Wahl auf `Synaptic`, deren ausschlaggebendster Vorteil die Möglichkeit ist, das Training über `Web Worker` durchführen lassen zu können. Was `Web Worker` sind und weshalb sie einen Vorteil für die Anwendung bringen, soll im Kapitel 7.1.2 erläutert werden.

7.1.2 Performancevorteile durch Nutzung von Web Workern

Problem der Synchronität und des Single-Threadings

Allgemein kann Javascript als eine Programmiersprache betrachtet werden, die normalerweise *synchron* und *single-threaded* läuft. Single-threaded bedeutet, dass zur gleichen Zeit nur ein Prozess stattfinden kann ([Brij 2015](#)). Von synchroner Programmierung spricht man, wenn der Start von einem Prozess das ganze Programm solange zum Stoppen bringt, bis der Prozess zu Ende ausgeführt wurde. Das bedeutet also, dass bei Javascript die Prozesse sequenziell bzw. nacheinander abgearbeitet werden.

Bezogen auf die Anwendung für die Bachelorarbeit stellt diese Eigenschaft ein ungünstiges Problem da. Für das Training werden komplexe Berechnungen durchgeführt, besonders bei der Backpropagation finden viele Berechnungsschritte statt. Das Training des MLP mit der Methode `train()` der `Synaptic` Bibliothek führte dazu, dass die Bedienung der GUI während des Trainings sehr träge wirkte. Bis auf Interaktionen vom Anwender wie ein Mausklick auf einen Button reagiert wurde, vergingen teilweise mehrere Sekunden.

Multithreading mithilfe von Web Workern

Es musste eine passende Technik gefunden werden, die Nebenläufigkeit bzw. Multithreading in Javascript zu ermöglichen, d.h. mehrere Prozesse gleichzeitig ausführen lassen zu können. Denn um dem Anwender eine reaktionsschnelle GUI bieten zu können, müssen die Berechnungen für das Trainings nebenläufig laufen. Es gibt verschiedene Wege die Nebenläufigkeit in Javascript nachzuahmen, wie die Verwendung von `setTimeout()`, `setInterval()`, `XMLHttpRequest` oder Ereignis-Handlern ([Bidelman](#)

2010). Allerdings findet bei diesen Techniken alles immer noch im selben Hauptthreadⁱⁱ im Browser statt, so wechseln sich lediglich die Prozesse der Skriptaussführungen mit den Aktivitäten der GUI ab. Web Worker sind eine Javascript APIⁱⁱⁱ für HTML5, mit der tatsächlich im Hintergrund Skripte über mehrere Threads ausgeführt werden können.

Zum Testen der Geschwindigkeit gibt es für die Synaptic Bibliothek im Ordner *synaptic-withoutWW* ein Testprogramm, welches das Training normal ohne einen Web Worker ausführt und im Ordner *synaptic-withWW* ein Testprogramm, welches das Training mit einem Web Worker ausführt. Auf dem getesteten Rechner fiel beim Vergleichen der Zeiten auf, dass sogar mit dem Web Worker eine bessere Zeit erzielt wurde (ca. 750 ms gegenüber ca. 1000 ms ohne Web Worker). Bei der Implementierung des MLP mit Synaptic ist auch deutlich aufgefallen, dass durch die Nutzung eines Web Workers die GUI während des Trainings deutlich schneller auf die Interaktionen des Anwenders reagiert hat.

7.1.3 Trainieren eines MLP mit Synaptic

In der Javascript Datei *neural-network.js* ist der Codeteil zu finden, in dem das MLP erstellt und trainiert wird. Es soll hier in dieser Arbeit nicht der ganze Code erläutert werden, sondern nur kurz auf den interessanten Codeteil, der für das Training eingegangen werden (siehe Listing 7.1). Bei *trainAsync()* handelt es sich um eine Methode für das Perzeptron, das Training in einem Webworker stattfinden lassen zu können. Der Parameter *rate* legt die Lernrate und *iterations* legt die maximale Anzahl an Iterationen fest, diese wurden vom Anwender zuvor in der GUI festgelegt. Der *error* gibt den Wert an, der unterschritten werden muss, bis das Training beendet werden kann. Durch *cost* wird die Fehlerfunktion festgelegt, in diesem Fall wird der MSE genommen, der im Kapitel 5.4.3 erwähnt wurde. Mit *myTrainer.trainAsync()* wird ein Promise-Objekt *promiseTrain* erstellt, der bei erfolgreicher Operation mit *promiseTrain.then()*, die Trainingsergebnisse an die Funktion *updateAndSendMessageForApp* übergibt.^{iv} Dieser führt wiederum die Operationen durchführt, damit die Ergebnisse später in der GUI visualisiert werden können.

```
1 var promiseTrain = myTrainer.trainAsync(trainingSet, {
2   rate: messageForApp.nnConfigInfo.learningRate,
3   iterations: iterations,
```

ⁱⁱBei einem Thread handelt es sich um eine Aktivität innerhalb eines Prozesses. Jeder Prozess besteht aus mindestens einem Thread, dem Hauptthread. Neben dem Hauptthread können im Prozess noch weitere Threads ausgeführt werden (Wolf 2009).

ⁱⁱⁱSchnittstelle zur Anwendungsprogrammierung (engl.: application programming interface)

^{iv}Das Promise-Objekt wird dazu verwendet, um asynchrone Berechnungen durchführen zu können. Genauerer zu der Technologie kann auf der Seite https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise nachgelesen werden.

[illegible]

Listing 7.1: Codeausschnitt des Trainings eines MLP mit Synaptic

Aufbau eines Perzeptrons in Synaptic

Vereinfacht sieht der Code eines Perzeptrons in Synaptic als JSON wie in Listing 7.2 aus. Die Attribute „*input*“ und „*output*“ bekommen beide jeweils als Wert ein Layer-Objekt (siehe Listing 7.3) zugewiesen, sie stellen die Eingabeschicht und die Ausgabeschicht des MLP dar. Das Attribut „*hidden*“ bekommt als Wert ein Array zugewiesen, welches aus mehreren Layer-Objekten besteht, welches die versteckten Schichten darstellen.

```
1 {"layers" = {
2   "hidden": [...],
3   "input": {...},
4   "ouput": {...}
5 }}
```

Listing 7.2: JSON eines Perzeptron-Objekts in Synaptic

Das Listing der JSON vom Layer-Objekt ist stark vereinfacht, indem nur die Attribute und Werte angezeigt werden, die für die Visualisierung des MLP in der GUI von Bedeutung sind.

```
1 {"Layer":{:
2   "list":[
3     "Neuron":{:
4       "ID": 5,
5       "activation": 0,
6       "bias": 0.46,
7       "connections":{:
8         "projected":{:
9           "Connection":{:
10            "from":{:
11              "ID": 5,
12              ...
13            },
```

```

14         "to":{
15             "ID": 7,
16             ...
17         },
18         "weight": 6.553,
19         ...
20     },
21     ...
22 },
23 ...
24 },
25 ...
26 },
27 ...
28 ],
29 "size": 5,
30 ...
31 }}

```

Listing 7.3: JSON eines Layer-Objekts

7.1.4 Schnittstelle zwischen der Netzworkebibliothek und der GUI

Dem Vorteil von Web Workern, dass rechenintensive Skripte nicht mehr im Hauptthread ausgeführt werden müssen, stehen einige Nachteile gegenüber. Im Bezug auf die Anwendung für die Bachelorarbeit sind die wichtigsten Nachteile, dass Web Worker nicht in der Lage sind, das DOM^v zu ändern und dass sie nicht auf globale Variablen und Funktionen zugreifen können (Buckler 2013). Für die Anwendung bedeutet dies, dass während des Trainingsvorgangs, indem die intensiven Berechnungen stattfinden, die GUI nicht verändert werden kann.

In seiner Bachelorthesis hat Koenecke die Vermutung aufgestellt, dass eine Implementierung des KNN in Javascript wohl keine Schnittstelle zur Kommunikation zwischen der Netzworkebibliothek und der GUI erfordert hätte (Koenecke 2016: S.42). Im Grunde genommen hat er damit Recht, dies setzt jedoch voraus, dass die Berechnungen des Trainings im Hauptthread stattfinden, um währenddessen gleichzeitig die GUI verändern zu können. Wie bereits vorher erläutert, würde dies aber zu sehr die Performance verschlechtern und daher ist die Verwendung von Web Workern zu bevorzugen. Um einen Austausch der Trainingsergebnisse mit den Interaktionen der GUI zu ermöglichen, hat es sich also weiterhin angeboten die Schnittstelle zur Kommunikation beizubehalten. Zudem gewährleistet eine Trennung der Netzworkebibliothek von der GUI eine bessere Übersichtlichkeit vom Code der Anwendung. Wie auch in Koeneckes Anwendung stellt die Javascript-Datei *app.js* die Schnittstelle dar, mit

^vDOM: Document Object Model; Programmierschnittstelle für HTML und XML Dokumente

dem Unterschied, dass in dieser Anwendung kein Server und daher keine emulierten Websockets mehr benötigt werden.

Aufbau eines message-Objektes für die GUI

In der Schnittstelle wird zur Erstellung des MLP mit der Methode *newNetwork()* und zum Updaten des MLP mit der Methode *updateNetwork()* in der GUI ein message-Objekt mit einer bestimmten Struktur benötigt (siehe Listing 7.4). Sie entspricht zum größten Teil dem message-Objekt von Koenecke, aufgrund der Erweiterungen in der Anwendung wurde das message-Objekt etwas erweitert. In der *neural-network.js* wurden zur Erstellung der message-Objekte in diesem Format die Funktionen *getMessageForApp()* und *getUpdatedMessageForApp()* geschrieben.

```

1 {"message" = {
2   "bMaxIterationsReached": false,
3   "nnConfigInfo": {
4     "activationFunction": "relu",
5     "learningRate": 0.01,
6     "maxIterations": 100000
7   },
8   "id": 1,
9   "graph":{
10    "layers": [
11      {
12        "numberOfNeurons": 3,
13        "weights":{
14          "data":[
15            {0.54, 0.945, 0.435},
16            ...
17          ]
18        }
19      },
20      ...
21    ],
22    "sampleCoverage": 0,
23    "samplesTrained":0,
24    "weightChange":0
25  }
26  "output":{
27    "data": [
28      [253, 0, 0],
29      ...
30    ]
31  }
32 }}

```

Listing 7.4: JSON eines Perzeptron-Objekts in Synaptic

7.1.5 Erweiterungen an der GUI

8 Konfiguration eines KNN passend zur Problemstellung

Abkürzungsverzeichnis

MLP mehrlagige Perzeptron

KI Künstliche Intelligenz

KNN künstliche neuronale Netze

GUI Grafical User Interface

Abbildungsverzeichnis

1.1	Anwendungsgebiete für den Einsatz von KNN, entnommen aus http://vieta.math.tu-cottbus.de/~kolb/ml-nn/node10.html	6
2.1	Venn Diagramm zur Einordnung der KNN	12
3.1	Aufbau einer Nervenzelle	14
3.2	Funktionsweise eines Neurons, modifizierte Version aus http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html	15
4.1	Schematisches Beispiel Feedforward Netzwerk	19
4.2	Schematisches Beispiel Feedback Netzwerk	19
4.3	Beispiel Feedforward Netzwerk, modifizierte Version aus http://www.texample.net/tikz/examples/neural-network/	20
4.4	Repräsentation der Konjunktion $x_1 \wedge x_2$ als einfaches Perzeptron . . .	21
4.5	Repräsentation der Disjunktion $x_1 \vee x_2$ als einfaches Perzeptron . . .	21
4.6	Beispiel 1 für die lineare Separierbarkeit der AND-Funktion	23
4.7	Beispiel 2 für die lineare Separierbarkeit der AND-Funktion	23
4.8	Die OR-Funktion ist nicht linear separierbar	24
4.9	Durch die Kombination von drei Perzeptronen, kann die XOR-Funktion gelöst werden.	25
4.10	MLP, der die XOR-Funktion lösen kann ohne Bias-Neuronen	26
4.11	MLP, der die XOR-Funktion lösen kann mit Bias-Neuronen	26
5.1	Bei zwei verschiedenen Gewichten kann die Fehlerfunktion E als Fläche dargestellt werden	31
6.1	Konfiguration der Netzwerktopologie	36
6.2	Darstellung eines Netzwerkgraphs (untrainiert)	36
6.3	Darstellung eines Netzwerkgraphs (trainiert)	36
6.4	Erzeugung der Trainingsdaten	37
6.5	Start des Trainings mit 10-facher Geschwindigkeit	37
6.6	Vorschau nach einer halben Million Beispieldaten	38
6.7	Vorschau nach einer Million Beispieldaten	38
6.8	Vorschau nach drei Millionen Beispieldaten	38
6.9	Netzwerkinfo während des Trainings	38

Tabellenverzeichnis

3.1	lineare Aktivierungsfunktion, ReLu und die binäre Aktivierungsfunktion	16
3.2	Logistische und tanh Aktivierungsfunktion	17

Literaturverzeichnis

- Bidelman, Eric: *Web Worker-Grundlagen*, <https://www.html5rocks.com/de/tutorials/workers/basics/>, 26.07.2010, letzter Zugriff: 29. 08. 2017
- Brij: *Concurrency vs Multi-threading vs Asynchronous Programming : Explained*, <https://codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/>, 29.07.2015, letzter Zugriff: 29. 08. 2017
- Buckler, Craig: *Implementing JavaScript Threading with Web Workers*, <https://www.safaribooksonline.com/blog/2013/10/24/implementing-javascript-threading-with-web-workers/>, 24.10.2013, letzter Zugriff: 29. 08. 2017
- Corves, Anna: *Nervenzellen im Gespräch*, <https://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/nervenzellen-im-gespraech?gclid=C0iWg7TdzNQCFU4W0wodK74IwA>, 2012, letzter Zugriff: 1. 07. 2017
- Ertel, Wolfgang: *Grundkurs Künstliche Intelligenz : Eine praxisorientierte Einführung*, 4. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2016
- Hebb, Donald Olding: *The Organization Of Behavior: A Neuropsychological Theory*, Psychology Press edition 2002, 1949
- Hillmann, Leonard: *Intelligentes Leben*, <http://www.tagesspiegel.de/themen/gehirn-und-nerven/gesund-leben-intelligentes-leben/13410564.html>, Veröffentlichungsdatum unbekannt, letzter Zugriff: 01.07.2017
- Hoffmann, Norbert *Kleines Handbuch Neuronale Netze : Anwendungsorientiertes Wissen zum Lernen und Nachschlagen*, Berlin Heidelberg New York: Springer-Verlag, 1993
- Koenecke, Finn Ole: *Realisierung eines interaktiven künstlichen neuronalen Netzwerks*, 2016
- Kramer, Oliver: *Computational Intelligence : Eine Einführung*, 1. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2009.
- Manhart, Klaus: *Was Sie über Maschinelles Lernen wissen müssen*, <https://www.computerwoche.de/a/was-sie-ueber-maschinelles-lernen-wissen-muessen,3329560>, 04.05.2017, letzter Zugriff: 02.07.2017

- Minsky, Marvin; Papert, Seymour: *Perceptrons: An Introduction to Computational Geometry*, 2nd edition with corrections, first edition 1969 , The MIT Press, Cambridge MA, 1972.
- Rey, Günter Daniel ; Wender, Karl F.: *Neuronale Netze : eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*, 2. vollst. überarb. und erw. Aufl., Bern: Huber, 2011.
- Riley, Tonya: *Artificial intelligence goes deep to beat humans at poker*, <http://www.sciencemag.org/news/2017/03/artificial-intelligence-goes-deep-beat-humans-poker>, 03.03.2017, letzter Zugriff: 07.07.2017
- Rimscha, Markus: *Algorithmen kompakt und verständlich : Lösungsstrategien am Computer*, 3. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2014.
- Russell, Stuart ; Norvig, Peter: *Artificial Intelligence : A Modern Approach*, 3. Aufl(2010), London: Prentice Hall, 2010.
- Scherer, Andreas: *Neuronale Netze : Grundlagen und Anwendungen*. Berlin Heidelberg New York: Springer-Verlag, 1997.
- SethBling: *MarI/O - Machine Learning for Video Games*, <https://www.youtube.com/watch?v=qv6UVOQOF44>, 13.06.2017, letzter Zugriff: 07.07.2017
- Turing, Alan M.: *Computing Machinery and Intelligence*, Mind, 59, 1950.
- Wolf, Jürgen: *C von A bis Z : das umfassende Handbuch*, http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/026_c_paralleles_rechnen_003.htm Bonn: Rheinwerk Verlag GmbH, 2009, letzter Zugriff: 29.08.2017
- Wunderlich-Pfeiffer, Frank : *Alpha Go geht in Rente*, <https://www.golem.de/news/kuenstliche-intelligenz-alpha-go-geht-in-rente-1705-128059.html>, 29.05.2017, letzter Zugriff: 04.07.2017
- Zell, Andreas: *Simulation neuronaler Netze*, 4.Auflage(2003) Deutschland: Oldenbourg Wissenschaftsverlag GmbH, 1994.

Ich versichere, die vorliegende Arbeit selbstständig 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

Xuan Linh Do