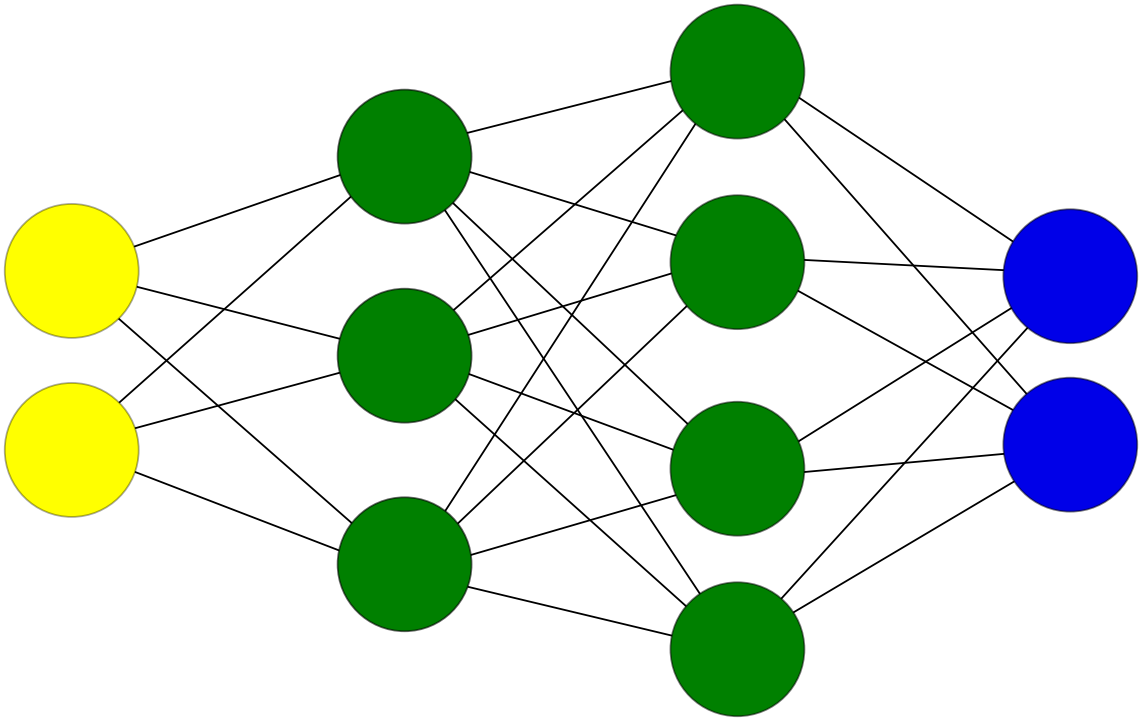
Künstliche neuronale Netze

Einführung und Anwendung im Bereich der Mustererkennung



Michael Pucher

BG/BRG Weiz, Offenburgergasse 23

Künstliche neuronale Netze

Einführung und Anwendung im Bereich der Mustererkennung

Fachbereichsarbeit aus dem Fach Informatik

Vorgelegt bei OStR Mag. Helmuth Peer
von Michael Pucher, 8B

Weiz, am 1. März 2013

Kurzzusammenfassung

Diese Arbeit behandelt den Aufbau und die Funktionsweise einfacher künstlicher neuronaler Netze, sowie einen Einblick in den damit umsetzbaren Bereich der Mustererkennung. Dabei wird prägnant auf biologische neuronale Netze als Vorlage eingegangen und das Problem Mustererkennung mithilfe des Computers dargestellt. Grundlegend soll eine kurze Einführung in die Geschichte, sowie ein Überblick über Struktur, Funktion einzelner Elemente, Typen und Lernalgorithmen künstlicher neuronaler Netze gegeben werden. Zur Untermalung der vorgestellten, aus Literaturrecherchen gewonnenen Erkenntnisse, werden auch die Ergebnisse zweier Versuche, die mithilfe der Programmiersprache Python und dem Modul PyBrain durchgeführt wurden, angeführt. Bei den Versuchen handelte es sich um die Erkennung von handgeschriebenen Ziffern und Gesichtserkennung.

Abstract

This paper is dealing with the structure and the functionality of simple artificial neural networks and providing insight into the field of pattern recognition using those networks. At the same time a short introduction into biological neural networks, as reference for artifical ones is given, and the problem of a computer not being able to recognize patterns is explained. This paper should basically give a brief insight of history, structure, functions of each elements,learning algorithms and types of artificial neural networks. To accentuate the presented information gathered through literature research, two experiments were realized by using programming language Python and the module PyBrain. The experiments dealt with recognizing handwritten digits and face recognition.

Vorwort

Aus Liebe zur Informatik war mir bereits lange zuvor klar, dass ich mich dafür entscheiden würde in diesem Fach zur Matura anzutreten und eine Fachbereichsarbeit zu verfassen. Allerdings fehlte beim Treffen dieser Entscheidung noch ein brauchbares Thema für diese Arbeit. Ausschlaggebend für das Thema war die Woche der Modellierung mit Mathematik in Pöllau. Dort wurde ich vertraut mit der grundlegenden Funktionsweise neuronaler Netze, sowie mit dem Python-Modul PyBrain, welches anschaulich und einfach künstliche neuronale Netze erzeugen und trainieren kann. Da in mir schon immer ein großes Interesse an künstlicher Intelligenz vorlag und künstliche neuronale Netze einen Teilbereich künstlicher Intelligenz abdecken, entschied ich mich dafür dieses Thema in meiner Arbeit näher zu beleuchten. Mit meinem langjährigen Informatikprofessor Helmuth Peer war schnell ein ausgezeichneter Betreuungslehrer gefunden, der immer wieder mit Verbesserungsvorschlägen zur Seite stand. Besonderer Dank gilt meinem Betreuer auf der Modellierungswoche, Stefan Fürtinger, der mit Vorschlägen bezüglich Fachliteratur aufwarten konnte.

Weiz, am 1. März 2013 Michael Pucher

Inhaltsverzeichnis

1 Einleitung

Während Menschen spielend leicht Gesichter erkennen können, versagen Computer auf diesem Gebiet. Gebaut, um den Menschen Rechenarbeit zu erleichtern, hätte niemand daran gedacht, dass es weitaus mehr Rechenaufwand erfordert ein einfaches Bild wiederzuerkennen.

Die vorliegende Arbeit befasst sich mit künstlichen neuronalen Netzen. Über die Jahre wurden einige mathematisch aufwendige Algorithmen für z.B. Gesichtserkennung entwickelt, künstliche neuronale Netze verwenden hingegen eine Großzahl an einfachen Recheneinheiten. Die Idee dahinter kommt direkt aus der Biologie: Wenn das Gehirn Bilder erkennen kann, so sollte auch eine Simulation des Gehirns dazu in der Lage sein. Das erste Kapitel des Hauptteils gibt einen Vergleich Mensch-Maschine an und erläutert prägnant den Aufbau und die Funktionsweise menschlicher neuronaler Netze, bzw. der eigentlichen Nervenzellen (Neuronen). Diese Neuronen sind im Grunde nichts anderes, als elektrische und chemische Schaltungen, weshalb es für die ersten Informatiker, welche sich damit befassten, einfach war ein mathematisches Modell aufzustellen. Von dort an wurde die erste künstliche mathematische Nervenzelle weiterentwickelt. Dazu kam die rasante Entwicklung immer schnellerer Computer, welche auch immer größere neuronale Netze berechnen konnten. Heutzutage können einfache künstliche neuronale Netze auf dem Heimrechner mithilfe der Programmiersprache Python und dem Modul PyBrain simuliert werden.

Der vorliegende Text versucht einen groben Einblick in die Welt künstlicher neuronaler Netze, deren Funktionsweise und Ähnlichkeit zu biologischen neuronalen Netzen zu gewähren. Die Einsatzmöglichkeiten künstlicher neuronaler Netze sind vielfältig: So können diese benutzt werden, um beispielsweise menschliches Verhalten zu simulieren. Allerdings würde eine Behandlung aller Verwendungsmöglichkeiten den Rahmen dieser Arbeit sprengen, weshalb sich dieser Text ausschließlich mit Mustererkennung, auf Grundlage von Literaturrecherchen und Software-Beispielen, befasst.

2 Die Natur als Vorbild

2.1 Problemstellung Mustererkennung

Während ein Mensch seine tägliche Umwelt, Bilder und Muster ohne Aufwand wiedererkennen kann, hat er Schwierigkeiten, große Zahlenmengen ohne Hilfsmittel zu verarbeiten. Im Vergleich dazu kann ein Computer große Zahlenwerte innerhalb weniger Sekunden berechnen, braucht aber zum Verarbeiten oder Erkennen von Bildern viel Zeit und verzeichnet dabei einen großen Anstieg der Rechenleistung. In der heutigen Zeit wird es allerdings immer wichtiger Bilddaten mithilfe von Computern zu analysieren.

2.1.1 Vergleich Mensch-Maschine

Während es im menschlichen Gehirn ca. 10^{11} Schalteinheiten, die Neuronen, gibt, besitzt ein durchschnittlicher Prozessor nur etwa 10^9 Transistoren. Dabei beträgt allerdings die Schaltzeit von Neuronen etwa 10^{-3}s, die Schaltzeit von Transistoren etwa 10^{-9}s, das heißt, ein Computer sollte theoretisch in einer Sekunde mehr Schaltvorgänge durchführen können als ein Gehirn. Die Praxis zeigt allerdings das Gegenteil: der Unterschied zwischen den in der Praxis durchgeführten und den möglichen Schaltvorgängen eines Computers beträgt einige Zehnerpotenzen, während dieser Unterschied im menschlichen Gehirn erheblich kleiner ausfällt. Der größte Teil des Gehirns arbeitet durchgehend, während ein Computer einen Großteil der Zeit nur passiv Daten speichert. Zusätzlich arbeitet das Gehirn im Gegensatz zum Computer parallel und verändert ständig seine Struktur, um Fehler zu kompensieren und Ergebnisse zu optimieren. Der aber wohl wichtigste Vorteil ist die Fähigkeit des Gehirns, zu lernen.[1]

[1] vgl. David Kriesel. *Ein kleiner Überblick über Neuronale Netze*. http://www.dkriesel.com. 2007, S. 4.

2.1.2 Vorteile von neuronalen Netzen

Anstelle weniger leistungsaufwendiger Recheneinheiten wird in neuronalen Netzen eine große Anzahl an einfach gebauten Recheneinheiten verwendet. Diese arbeiten parallel und besitzen Lernfähigkeit. Bei künstlichen neuronalen Netzen geschieht das Lernen durch Algorithmen, welche das Netz durch Veränderung der eigenen Neuronen und deren Verbindungen anhand von immer wieder präsentierten Daten trainieren. Da neuronale Netze nicht einfach Daten speichern, sondern durch Lernen Daten assoziieren, besitzen sie die Fähigkeit zur Generalisierung, was bedeutet, dass nach erfolgreichem Lernvorgang auch *ähnliche* Daten assoziiert werden können. Zusätzlich ergibt sich durch die Verteilung der Daten auf alle im Netz vorhandenen Neuronen eine gewisse Fehlertoleranz. Beispielsweise sterben bei einem Vollrausch 10^5 Neuronen ab und das Gehirn funktioniert am nächsten Tag trotzdem einwandfrei. Allerdings ergeben sich durch die Verteilung der Daten auch Probleme, denn man kann auf den ersten Blick nicht erkennen, was ein neuronales Netz weiß, was es kann oder wo seine Fehler liegen.[2]

Grundgedanke am Anfang der Erforschung künstlicher neuronaler Netze war nun, anhand der biologischen Vorlage von Neuronen und ihren Verbindungen ein mathematisches Modell für die Verwendung am Computer zu entwickeln.

2.2 Biologische neuronale Netze & Aufbau eines Neurons

2.2.1 Nervensysteme beim Menschen

Die Nervensysteme im Körper des Menschen lassen sich grundlegend in zwei Gruppen einteilen: das **zentrale Nervensystem** und das **periphere Nervensystem**. Zum zentralen Nervensystem gehören Gehirn und Rückenmark, während sich das periphere Nervensystem durch den ganzen Körper zieht und verzweigt. Die neuronalen Net-

[2] vgl. Kriesel, *Ein kleiner Überblick über Neuronale Netze*, S. 5.

ze des peripheren Nervensystems beinhalten eine große Zahl von Recurrent Networks (rückgekoppelten Netzwerken), bei denen Neuronen mit sich selbst oder vorhergehenden Neuronen im Netz verbunden sind. Dadurch ergibt sich der Vorteil, die Signale der Neuronen zu verstärken, was vor allem für die Motorik wichtig ist. Für Mustererkennung ist aber hauptsächlich das Gehirn verantwortlich.

In Kapitel 3 werden hauptsächlich Begriffe wie Input-, Hidden- oder Output-Schichten bzw. Layer verwendet. Um dies zu erläutern, ein Beispiel von biologischen neuronalen Netzen zum Vergleich: Ein Mensch erkennt einen Apfel und assoziiert diesen mit dem dazugehörigen Wort.

Der Apfel reflektiert Licht, welches von den Augen eingefangen wird, wo das Licht die Sinneszellen im Auge dazu bringt Signale in das Gehirn weiterzuleiten. Diese Signale sind der **Input** des neuronalen Netzes, die Schicht der Sinneszellen im Auge ist hier die Input-Schicht, welche Daten, in diesem Fall ein Bild des Apfels, in das Netzwerk sendet. Als zweiter Schritt durchlaufen diese Daten eine Unmenge an Neuronen im Gehirn, welche man in diesem Beispiel als **Hidden**-Neuronen bezeichnen kann. Diese Neuronen sind lediglich dafür zuständig die Daten, in diesem Fall den Apfel, zu assoziieren und durch Lernvorgänge eine Assoziation zu erzeugen oder eine vorhandene Assoziation weiter auszubauen. Die Hidden-Neuronen enden dort, wo die Verbindung mit dem Wort "Apfel" erreicht ist. Im menschlichen Gehirn entsteht nun der Gedanke "Apfel", oder das Wort wird ausgesprochen. Die Neuronen, die nun für diesen Gedanken bzw. für das Aussprechen des Wortes verantwortlich sind, werden als **Output**-Neuronen bezeichnet. Je öfter ein Apfel beobachtet wird, desto stärker wird die Assoziation mit dem Wort. Ohne Sprache fehlt die Verbindung mit dem Wort, allerdings bildet das Gehirn trotzdem die nötigen neuronalen Strukturen aus, z.B. bei Kleinkindern, die noch nicht sprechen können. Diese erkennen Gegenstände, Personen etc. wieder, aber zu diesem Zeitpunkt fehlt noch die Assoziation mit den entsprechenden Wörtern. Ähnliches tritt beispielsweise beim Competitive Learning (Abschnitt 3.4.3) auf, wo der Input zwar kategorisiert wird, aber keine Verbindung mit einem konkreten Begriff hergestellt wird.

2.2.2 Aufbau eines biologischen Neurons

Neuronen sind im Grunde mit elektrischen Schalteinheiten vergleichbar. Am Kopf des Neurons befinden sich **Dendriten**, Rezeptoren für chemische Botenstoffe, die im **Soma**, dem Kern des Neurons, münden. Am Soma schließt das **Axon** an, ein Leiter für elektrische Signale, welches sich am Ende in weitere Dendriten aufspaltet, welche sich wiederum mit den Dendriten anderer Neuronen verbinden. Diese Verbindungen werden **Synapsen** genannt.[3]

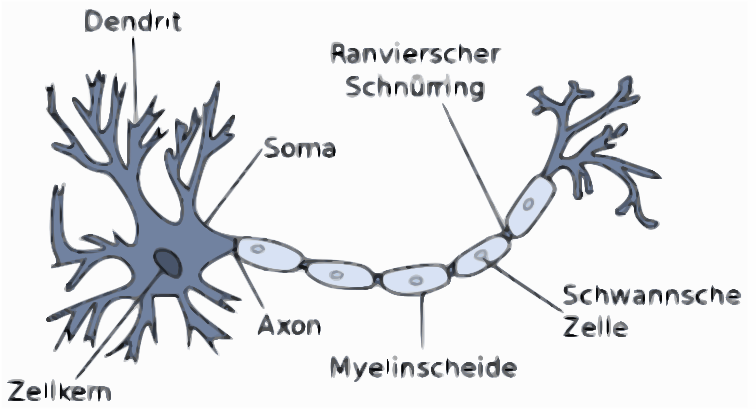


Abbildung 1: Aufbau eines Neurons. (*Abbildung nach* Kriesel, *Ein kleiner Überblick über Neuronale Netze*. S.20)

Wenn es in Kapitel 3 um den Aufbau von mathematisch nachgeahmten Neuronen geht, werden abermals die Begriffe Input und Output verwendet. Im übertragenen Sinne existieren diese Begriffe auch bei biologischen Neuronen. Der **Input** eines Neurons kann durch zwei verschiedene Arten von Synapsen empfangen werden: **elektrische** und **chemische Synapsen**. Während elektrische Synapsen ihre Signale direkt an den Soma übertragen, senden chemische Synapsen ihre Signale an die Dendriten, wo es zwischen Dendrit und Synapse einen **synaptischen Spalt** gibt. Dort wird das elektrische Signal des sendenden Neurons in chemische Botenstoffe, die sogenannten Neurotransmitter, umgewandelt und übertragen. Im Soma werden nun die eingehenden Signale aufsummiert,

[3]vgl. Kriesel, *Ein kleiner Überblick über Neuronale Netze*, S. 19/20.

dabei können die eingehenden Signale entweder einen **inhibitorischen**, also hemmenden, oder **exzitatorischen**, steigernden Einfluss besitzen. Überschreiten diese einen bestimmten Schwellwert, sendet das Neuron selbst ein Signal, welches über das Axon und schlussendlich über Dendriten und Synapsen an weitere Neuronen weitergeleitet wird.[4]

[4]vgl. Kriesel, *Ein kleiner Überblick über Neuronale Netze*, S. 20/21.

3 Künstliche neuronale Netze

3.1 Geschichtliche Entwicklung

3.1.1 Das McCulloch-Pitts Neuron & das Perzeptron

Das McCulloch-Pitts Neuron

Den Grundstein für die später aufblühende Entwicklung künstlicher neuronaler Netze legten der Logiker Walter Pitts und der Neurophysiologe Warren McCulloch, welche nach einer mathematischen Darstellung der Informationsverarbeitung im Gehirn suchten und 1943 ein formales mathematisches Modell eines Neurons aufstellten.[5]

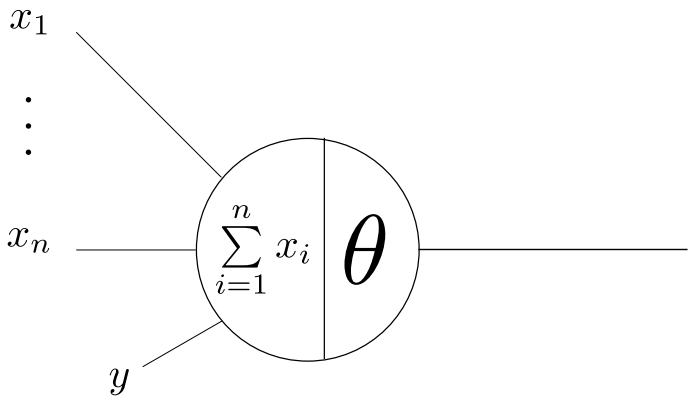


Abbildung 2: Darstellung des McCulloch-Pitts Neurons. Links befinden sich die eingehenden Verbindungen $x_1 ... x_n$ und die eingehende hemmende Verbindung y. Der Kern des Neurons wird durch den Schwellenwert θ repräsentiert. Wird dieser überschritten, senden alle ausgehenden Verbindungen 1.

McCulloch hatte bereits zuvor Vorstellungen eines neuronalen Netzes auf Basis künstlicher Neuronen. Allerdings konnte er seine Vorstellung nicht umsetzen, da inhibitorische

[5] vgl. Günter Daniel Rey und Karl F. Wender. *Neuronale Netze. Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung.* 2. Auflage. Verlag Hans Huber, Hogrefe AG, 2011, S. 14.

Einflüsse (also hemmende Einflüsse von verschiedenen Neuronen in einem Netz) biologisch nicht belegbar waren und er Probleme mit der Darstellung rekurrenter Netzverbindungen (ausgehende Verbindungen, die an Neuronen weiter vorne in der Netzstruktur senden) hatte. Das McCulloch-Pitts Neuron setzt sich aus einer beliebigen Anzahl an eingehenden und ausgehenden Netzverbindungen, sowie einer simplen mathematischen Funktion zusammen, welche Folgendes besagt: Ist die Summe der Werte aller eingehenden Netzverbindungen größer als ein bestimmter Schwellenwert θ, wird allen ausgehenden Verbindungen der Wert 1 zugewiesen, andernfalls der Wert 0. Diese Darstellung ist auch biologisch plausibel, vgl. Abschnitt 2.2. Die McCulloch-Pitts Darstellung ist das simpelste Modell eines Neurons. Es funktioniert binär, d.h. als Eingabe- und Ausgabewerte werden nur Nullen und Einsen verwendet. Außerdem besitzen Verbindungen des ursprünglichen McCulloch-Pitts Neurons keine Gewichte, die Erregung/Hemmung einer Verbindung bewirken, sondern erregende und hemmende Verbindungen. Da das Neuron nur binär arbeitet, kann man daraus schließen, dass eine einzelne hemmende Verbindung mit dem Wert 1 das gesamte Neuron deaktiviert.[6]

$$f(x) = \begin{cases} 0, \text{ wenn } m \geq 1 \text{ und mindestens eine Verbindung } y_1, y_2, ..., y_n \text{ gleich 1 ist} \\ 1, \text{ wenn } \sum_{i=1}^{n} x_i \geq \theta \end{cases}$$

Die Funktion f(x) prüft, ob die Anzahl der eingehenden hemmenden Verbindungen m größer oder gleich 1 ist und ob eine dieser Verbindungen den Wert 1 besitzt. Ist dies der Fall, sendet das Neuron den Wert 0. Gibt es keine hemmende Verbindung mit Wert 1, so werden die eingehenden Verbindungen summiert und mit dem Schwellenwert θ verglichen. Erreicht oder überschreitet die Summe den Schwellenwert, sendet das Neuron den Wert 1.

[6] vgl. Raúl Rojas. *Neural networks: a systematic introduction.* Springer-Verlag New York, Inc., 1996, S. 52,31,32.

Dieses ursprüngliche Neuronen-Modell ist in Funktionsweise und Handhabung sehr beschränkt. Wollte man ein neuronales Netz mit diesen Neuronen entwickeln, musste man zuvor die Netzstruktur sorgfältig planen, was bei Aufgabenstellungen, welche viele Neuronen benötigen, schwer bis unmöglich war. Ein verbessertes Modell mit dem Namen Perzeptron sollte das ändern.

Das Perzeptron

Das Perzeptron als generalisiertes Modell des McCulloch-Pitts Neurons wurde 1958 vom amerikanischen Psychologen Frank Rosenblatt vorgeschlagen. Die wichtigste Änderung war das Einführen von Gewichten.[7] Gewichte bestimmen, ob sich eine Verbindung inhibitorisch oder exzitatorisch, also hemmend oder verstärkend auf den Eingabewert der Verbindung auswirkt. Auch ist das Perzeptron nicht auf binäre Werte begrenzt, sondern kann jeden beliebigen reellen Zahlenwert annehmen. Allerdings wurde das Perzeptron in den 1960er Jahren von Marvin Minsky und Seymour Papert neu definiert und perfektioniert.[8]

Ein Perzeptron besteht aus einer Neuronenschicht, an welche der Input gesendet wird. Die Neuronen dieser Schicht verarbeiten die Input-Werte mit Berücksichtigung der Gewichte und senden die berechneten Werte, wie bei der einfachen McCulloch-Pitts Zelle, als Output. Dadurch, dass das Perzeptron nicht auf binäre Werte beschränkt ist, ändert sich der Aufbau. Der Schwellenwert verschwindet und an seine Stelle treten Aktivitätsfunktionen.

[7] vgl. Rojas, *Neural networks: a systematic introduction*, S. 55.

[8] Marvin Minsky und Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, 1969.

$$a = f(\sum_{i=1}^{n} x_1 w_1)$$

Abbildung 3: Mathematische Funktionsweise eines Perzeptrons. Eingehende Werte $x_1, ..., x_n$ werden mit ihren Gewichten $w_1, ..., w_n$ multipliziert und danach addiert. Die Aktivitätsfunktion $f(x)$ berechnet dann den Output-Wert a, welcher an alle ausgehenden Verbindungen gesendet wird.

Das Perzeptron verbessert zwar das Prinzip der McCulloch-Pitts Zelle, hat aber ein entscheidendes Problem. Mit einem Perzeptron können nur linear separable Probleme gelöst werden, wie in Abbildung 4 ersichtlich. Außerdem kann das boolsche XOR nicht mit einem Perzeptron dargestellt werden.

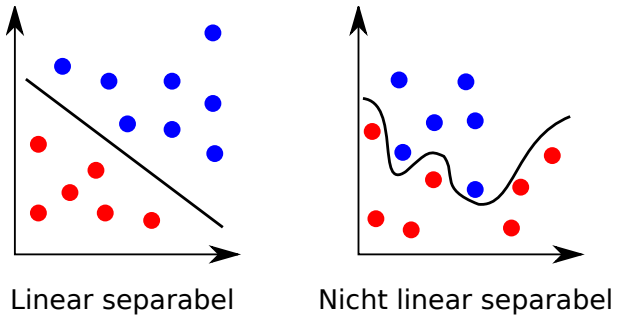


Abbildung 4: Das rechte Problem ist nicht linear separabel, d.h. man würde eine Kurve benötigen, um die Punkte zu trennen. Das linke Problem ist linear separabel und daher auch von einem Perzeptron lösbar.

3.1.2 Weitere Entwicklung bis heute

Entwicklung von Lernregeln

Hatte man vor dem Perzeptron nur die anhand des McCulloch-Pitts Modells aufgebauten statischen Netzwerke, welche man vor der Benutzung mit vorberechneten Werten versah, wurde es nun notwendig bzw. möglich, die Gewichte des Netzwerkes automatisiert

anzupassen. So konnte man Lernregeln für eine breite Auswahl an Anwendungsgebieten finden, sei es für die Simulation von biologischen neuronalen Netzen oder für die spezifische Anwendung, z.B. in der Mustererkennung. Wie der Lernprozess bei neuronalen Netzen definiert wird, findet sich in Kapitel 3.2.2, und in Kapitel 3.4 werden einige der häufigsten und bekanntesten Lernalgorithmen vorgestellt.

Ausbreitung der Anwendungsmöglichkeiten

War der Zweck hinter künstlichen neuronalen Netzen anfangs noch die Simulation des biologischen Aspekts, so haben sich konkrete Anwendungsbereiche herauskristallisiert. Grundsätzlich lassen sich die Anwendungen für künstliche neuronale Netze in zwei große Themenbereiche teilen: die Simulation menschlichen Verhaltens und das Lösen konkreter Probleme, wie im Bereich der Informatik, der Ingenieurswissenschaften, der Wirtschaft, etc.[9] Vor allem die statistische Anwendung bzw. das Benutzen von statistischen Methoden zum Trainieren und Auswerten von künstlichen neuronalen Netzen wurde zum momentan aktuellsten Forschungsgebiet in diesem Bereich.[10]

[9] vgl. Rey und Wender, *Neuronale Netze*, S. 14.

[10] vgl. Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006, S. 226.

3.2 Aufbau künstlicher neuronaler Netze

3.2.1 Units und Layer

Units

Wie auch biologische Netze setzen sich künstliche neuronale Netzen aus Einzelbausteinen zusammen. Zwei mögliche Arten dieser Bausteine, die McCulloch-Pitts Zelle und die Perzeptron Zelle, wurden bereits in Abschnitt 3.1.1 erwähnt. Die einzelnen Neuronen werden „Units“ genannt.

Für die weitere Verwendung für neuronale Netze wurde die Perzeptron-Zelle weiter generalisiert. Aus den eingehenden Verbindungen mit ihren Gewichten w wird mithilfe einer sogenannten Propagierungsfunktion, die meistens wie beim Perzeptron eine Summenfunktion ist, ein Gesamtinput berechnet. Die Aktivitätsfunktion der Zelle berechnet aus dem Gesamtinput den sogenannten Aktivitätslevel. Danach wird aus dem Aktivitätslevel der Output durch die Output-Funktion berechnet, allerdings gilt hier meistens Aktivitätslevel = Output. Alle Gewichte eines Netzwerkes zusammen können durch eine Gewichtsmatrix/einen Gewichtsvektor repräsentiert werden.

Als Aktivitätsfunktion kann jede beliebige reelle Funktion verwendet werden, allerdings unter Abhängigkeit der jeweiligen Anwendung. Bei statistischen Anwendungen ist diese Funktion stark abhängig von der jeweiligen Problemstellung, für biologische Simulationen wird eine Aktivitätsfunktion mit biologischer Plausibilität benötigt. Bei der Wahl der Aktivitätsfunktion in Bezug auf den später gewünschten Lernalgorithmus für das Netz ist die Differenzierbarkeit zu beachten, mehr dazu in Abschnitt 3.4.

In einem Netzwerk kann es auch sogenannte Bias-Units geben, welche keine Berechnungen durchführen, sondern einfach Output-Werte in das Netz schicken. Der Wert dieser Units bleibt konstant, während des Lernvorganges werden nur die Gewichte angepasst.

Layer

Die verschiedenen Schichten, also aufeinanderfolgende Gruppierungen von Units, eines künstlichen neuronalen Netzes werden als „Layer“ bezeichnet. Als Layer werden nur die Schichten gezählt, in welchen auch Aktivitätsberechnungen von den Units durchgeführt werden, weshalb die erste Schicht, der Input-Layer, meist nicht als Layer gezählt wird.

Ein Netzwerk bestehend aus Input- und Output-Layer, die letzte Schicht eines Netzwerks wird einschichtig genannt. Minsky zeigte, dass mit diesem Netztyp nur linear separable Probleme gelöst werden können. Diese Beschränkung fällt weg, wenn man mehrschichtige Netzwerke verwendet. Bei mehrschichtigen Netztypen werden die Layer zwischen Input- und Output-Schicht als Hidden-Layer bezeichnet. Hornik, Stinchcombe und White zeigten 1989, dass Problemlösungen mit mehreren Hidden-Layern auf einen einzigen Hidden-Layer mit ausreichender Neuronenanzahl reduziert werden können.[11]

Die Funktionsweise und der Aufbau des Netzwerkes mit verschiedenen Layern bleibt dem Netzarchitekten überlassen, denn es gibt auch Netze, welche die vorgestellte Netzstruktur mit Input-/Hidden- und Output-Layern gänzlich oder vollkommen auflösen können, z.B. haben sogenannte Kohonennetze einen zweidimensionalen Outputlayer.

3.2.2 Definition des Lernbegriffs & Durchführungsphasen

Trainieren von künstlichen neuronalen Netzen

Man kann künstliche neuronale Netze auf verschiedenste Art und Weisen konstruieren und trainieren. Zum einen gibt es die Möglichkeit, wie bei dem ursprünglichen McCulloch-Pitts Modell, die Gewichte und die gesamte Struktur des Netzes von Hand zu erstellen. Dabei kommt es allerdings zu keinem Lernprozess. Geschickter ist es, den geeigneten Netzwerktyp und den geeigneten Lernalgorithmus für die jeweilige Anwen-

[11] vgl. Rey und Wender, *Neuronale Netze*, S. 15.

dung zu wählen. Um überhaupt von einem Lernalgorithmus sprechen zu können, muss zuerst definiert werden, was man unter „Lernen“ im Sinn künstlicher neuronaler Netze versteht.

Zum Großteil wird das Lernen bei künstlichen neuronalen Netzen durch eine Anpassung der Gewichte ausgeführt, weshalb auch im weiteren Verlauf dieser Arbeit das Lernen immer als Anpassen der Gewichte verstanden werden kann. Durch spezifische Algorithmen können die Gewichte eines Netzwerks so angepasst werden, dass z.B die Fehlerrate bei einer Mustererkennungsanwendung minimal wird. Allerdings gibt es noch einige andere Arten, wie ein künstliches neuronales Netz lernen kann (vgl. Rey und Wender, *Neuronale Netze*, S. 33):

- Entwicklung von neuen oder Beseitigung von bereits vorhandenen Verbindungen zwischen Units
- Modifikation der Input-Schwelle, ab der eine Unit den Input weiterverarbeitet
- Veränderung der Propagierungs-, Aktivierungs- oder Ausgabefunktion
- Hinzufügen und Entfernen von einzelnen Units

Von den Möglichkeiten der Veränderung des Netzwerks abzugrenzen sind die Möglichkeiten, in welcher Art die Lernalgorithmen das Netzwerk verändern. Grundsätzlich kann man dabei drei Arten unterscheiden: Supervised learning, Reinforcement learning und Unsupervised learning. Beim Supervised learning kennt das neuronale Netz den zu erzielenden Output-Wert und das Netz wird darauf hintrainiert. Das Reinforcement learning, als teilweise Supervised learning System, gibt dem neuronalen Netz die zu erzielenden Werte nicht genau vor, sondern teilt dem Netz nur Richtigkeit der Ergebnisse mit. Dagegen gibt das Unsupervised learning keinerlei Informationen über die Richtigkeit der Ausgabewerte vor und das Netz wird nach Ähnlichkeit der Eingabewerte verändert.[12]

[12]vgl. Rojas, *Neural networks: a systematic introduction*, S. 78.

Durchführungsphasen künstlicher neuronaler Netze

Mit Ausnahme von Netzwerken, deren Struktur und Gewichte der Verbindungen bereits von Hand erstellt wurden, wie Netzwerke, die nur auf McCulloch-Pitts Neuronen basieren, kann man den Entwicklungszyklus eines künstlichen neuronalen Netzes in drei Phasen aufteilen: die Trainingsphase, die Testphase und die Anwendungsphase (vgl. Rey und Wender, *Neuronale Netze*, S. 26/27):

- Dem Netzwerk wird zuerst in der **Trainingsphase** auf bereits vorhandenes Lernmaterial, ein sogenanntes Trainingsset, eintrainiert. Im Falle der Mustererkennung von beispielsweise handgeschriebenen Ziffern werden dem Netzwerk eine Reihe von Bildern der Ziffern zur Verfügung gestellt und mithilfe von Lernalgorithmen erlernt (Anpassung der Gewichte/Struktur etc. siehe Abschnitt Lernalgorithmen). Das Netzwerk trainiert so lange mit dem Trainingsset, bis eine akzeptable Erkennungsquote erreicht wird. Allerdings muss darauf geachtet werden, dass das Netzwerk nicht zu sehr auf das Trainingsset trainiert wird, denn dann kann es zwar das bereits dargebotene Material einwandfrei erkennen, aber bei der Erkennung von neuem Material oder neuen Bildern wird es scheitern. Dieses Phänomen wird *overfitting* genannt.

- In der **Testphase** wird die Erkennungsrate des neuronalen Netzes überprüft. Während meistens bereits in der Trainingsphase immer wieder kontinuierlich die Erkennung des bereits vorhandenen Lernmaterials durchgeführt wird, gibt es auch die Möglichkeit in dieser Phase nochmals eine Überprüfung durchzuführen. Der Hauptzweck dieser Phase ist allerdings, das Netzwerk mit neuen, unbekannten Inhalten zu prüfen, denn nur dadurch lässt sich feststellen, ob es auch für die Anwendung geeignet ist. Das Netzwerk wird in dieser Phase nicht verändert.

- Als finale Entwicklungsstufe wird in der **Anwendungsphase** das fertig trainierte und geprüfte künstliche neuronale Netzwerk in einer konkreten Anwendung eingesetzt, z.B. zur Erkennung von handschriftlichen Ziffern.

Künstliche neuronale Netze müssen sich aber nicht strikt an diese Durchführungsphasen halten, denn es gibt auch verschiedene Netztypen, die beispielsweise laufend trainiert werden. Dies ist z.B. bei der Verwendung von neuronalen Netzen zur Gesichtserkennung der Fall. Außerdem ist ein anhaltender Lernprozess biologisch plausibler.

3.3 Beispiele für Netztypen

3.3.1 Pattern Associator

Pattern Associator steht für ein muster-assoziatives künstliches neuronales Netz, zur Zuordnung von Mustern zu einer Beschreibung und Wiedererkennung dieser Muster (z.B. handgeschriebene Ziffern, Gesichter, etc...). Der Vorteil eines Pattern Associator Netzes liegt im Fehlen der Hidden-Layer. Dadurch können einfache Lernalgorithmen wie die Hebb- oder die Delta-Regel verwendet werden.[13]

Der Input-Layer wird so erzeugt, dass die Musterinformationen (hauptsächlich Bildinformationen) eingelesen werden können. Danach wird der Output-Layer angelegt, wobei die Anzahl der Output-Units die Anzahl der verschiedenen zu erkennenden Muster darstellt. Durch Lernalgorithmen werden die Input-Werte mit den Output-Units assoziiert, das Netzwerk lernt die Muster zu erkennen.

Ein Pattern Associator Netz besitzt einige der für neuronale Netze wünschenswerten Eigenschaften (vgl. Rey und Wender, *Neuronale Netze*. S. 64):

[13] vgl. Rey und Wender, *Neuronale Netze*, S. 61/62.

- **Generalisierung und Erkennen von Prototypen der Kategorie**: Ähnliche Muster werden zu einer Gruppe kategorisiert. Diese Kategorisierung ermöglicht das korrekte Assoziieren von neuen, ähnlichen Mustern, ohne dabei das Netzwerk mit Lernalgorithmen weiter trainieren zu müssen. Das Netz bildet einen Prototypen für jede Muster-Kategorie aus, z.B. beinhaltet der Prototyp "Hund" die Eigenschaft "vier Beine".

- **Toleranz gegenüber internen und externen Fehlern**: Das Netzwerk kann Muster auch erkennen, wenn diese teilweise unkennbar gemacht werden oder wenn ein Unit des Netzwerks fehlerhafte Verbindungen aufweist.

3.3.2 Mehrschichtige Perceptron-Netze

Ein anderer Begriff für ein Perceptron-Netz mit mehreren Layern ist Multilayer **Feed-Forward** Netz. Hierbei handelt es sich um eine Modernisierung des in Abschnitt 3.1.1 vorgestellten Perceptron-Modells. Feed-Forward Netze besitzen, im Gegensatz zu Perceptron-Netzen, auch Hidden-Layer, mithilfe derer auch nicht linear separable Problemstellungen gelöst werden können. Die wichtigste Eigenschaft von Feed-Forward Netzen ist auch namensgebend: Die Informationen der Input-Schicht werden nur in eine Richtung nach vorne zum Output verarbeitet, keine Unit sendet Werte an sich selbst oder an eine Unit in einem vorhergehenden Layer. Dieser Netztyp wird in Abschnitt 4 zur Mustererkennung verwendet.

Dieser Netztyp besitzt wiederum die bereits beim Pattern Associator genannten Eigenschaften und wird z.B. mithilfe des Backpropagation-Algorithmus und Supervised learning trainiert, mehr dazu in Abschnitt 3.4.4.

3.3.3 Rekurrente Netze

Rekurrente Netze oder Recurrent Networks besitzen im Gegensatz zu Feed-Forward Netzen **Rückkopplungen**, also Verbindungen von einzelnen Units zu sich selbst, oder zu Units der vorangehenden Schicht. Dies ermöglicht das Erkennen von zeitlich codierten Informationen (z.B. bei der Spracherkennung) oder die Simulationen von menschlichen Bewegungen.[14] Auch ermöglichen Recurrent-Netze das Verarbeiten von Input mit variabler Länge, indem durch rückgekoppelte Verbindungen die Daten der letzten Berechnung gespeichert werden.[15] Trainiert werden rekurrente Netze mithilfe von Backpropagation.

Verbindungen, welche eine Rückkopplung im Netzwerk anzeigen, lassen sich nach Rey und Wender, *Neuronale Netze.* in folgende Arten unterteilen:

- **Direct feedback**: Eine Unit sendet an sich selbst.
- **Indirect feedback**: Eine Unit sendet an eine andere Unit in der vorhergehenden Schicht.
- **Lateral feedback**: Eine Unit sendet an eine andere Unit der selben Schicht.
- **Vollständige Verbindungen**: Jede Unit ist mit jeder Unit verbunden, ohne direct feedback.

Simple Recurrent Networks

Jeffrey L. Elman führte die sogenannten "Simple Recurrent Networks"(SRNs) als eine spezielle Variante von Recurrent networks ein. Er erweiterte ein Feed-Forward Netz um eine Reihe von Kontext-Units, die in der Netzstruktur im Input-Layer platziert waren.

[14] vgl. Rey und Wender, *Neuronale Netze*, S. 64.
[15] vgl. Rojas, *Neural networks: a systematic introduction*, S. 42.

Diese Kontext-Units sind wie gewöhnliche Input-Units mit allen Units des Hidden-Layers verbunden. Ihren Input erhalten sie allerdings durch die Hidden-Units: Jede Hidden-Unit sendet an lediglich eine Kontext-Unit. Es existieren gleich viele Kontext- wie Hidden-Units. Die jeweiligen Gewichte von Hidden- zu Kontext-Units behalten immer den Wert 1, nur die Gewichte von Kontext- zu Hidden-Layer werden verändert. Dadurch kann das Gedächtnis simuliert werden: ein SRN speichert die Information vorangegangener Durchläufe.[16]

Ähnlich aufgebaut wie SRNs, werden bei Jordan-Netzen allerdings nicht die Hidden-, sondern die Output-Units mit der Kontext-Schicht verbunden, d.h. die Zahl der Kontext-Units hängt von der Zahl der Output-Units ab. Die Kontext-Units senden an sich selbst und auch, wie bei SRNs, an alle Hidden-Units. Eine Erweiterung der SRNs stellen auch Elman-Netze dar, welche mehrere Hidden-Schichten zulassen. Pro Hidden- und Output-Layer gibt es einen Kontext-Layer. Eine Recurrent-Erweiterung für Pattern Associator Netze ist der Autoassociator. Beim Autoassociator sind alle Units der Output-Schicht mit allen anderen Units dieser Schicht, ausgenommen sich selbst, verbunden. Dies ermöglicht das Identifizieren und gleichzeitig das Wiederherstellen von unvollständigen Mustern.[17]

3.4 Beispiele für Lernalgorithmen

Wie in Abschnitt 3.2.2 erwähnt, werden künstliche neuronale Netze anhand von Lernalgorithmen trainiert. Einige bekannte Lernalgorithmen sollen hier nun erläutert werden. Es wird dabei immer davon ausgegangen, dass wir unter Lernen ein Anpassen der Gewichte verstehen.

[16] vgl. Rey und Wender, *Neuronale Netze*, S. 66/67.
[17] vgl. Rey und Wender, *Neuronale Netze*, S. 68.

3.4.1 Die Hebbsche Regel

1949 entwickelte der Psychologe Donald Hebb die nach ihm benannte Hebbsche Regel, die nicht nur eine der einfachsten Lernregeln für künstliche neuronale Netze darstellt, sondern auch biologische Plausibilität besitzt.

Seine Idee war, die Gewichte von Units, die gleichzeitig aktiv wären, zu verstärken, bzw. den Gewichten von gleichzeitig aktiven Units einen höheren Stellenwert zuzuschreiben als Units, die keine gemeinsame Aktivität aufweisen.[18] Er drückte seine Überlegung als mathematische Formel aus:

$$\Delta w_{ij} = \varepsilon \cdot a_i \cdot a_j$$

In Worten erklärt bedeutet dies, dass der Aktivitätslevel der sendenden Unit a_j multipliziert mit dem Aktivitätslevel der empfangenden Unit a_i und einer frei wählbaren Lernkonstante ε, in der Gewichtsveränderung Δw resultiert. In der klassischen Ausführung der Lernregel können die Aktivitätslevel jedoch nur Null oder einen positiven Wert annehmen, weshalb es zu Problemen kommen kann. Zum einen ist die Möglichkeit der Zustände, die ein künstliches neuronales Netz mithilfe der Hebbschen Lernregel erlangen kann, sehr gering, zum anderen werden die Gewichte mit großer Wahrscheinlichkeit überlaufen und gewaltig große Werte annehmen. Um diesen beiden Problemen vorzubeugen, kann man den Raum der möglichen Aktivitätslevel von 0 bis 1 auf -1 und $+1$ beschränken. Dadurch kann die Lernregel Gewichte nicht nur verstärken, sondern auch schwächen. Durch geringe Anpassungen kann diese Lernregel sowohl für Supervised-, Reinforcement- oder Unsupervised-Probleme angwendet werden.[19] Mit der Zeit wurden Modifikationen der Hebbschen Lernregel veröffentlicht, die ein breites Anwendungsgebiet ermöglichen.

[18] vgl. Rojas, *Neural networks: a systematic introduction*, S. 314.
[19] vgl. Rey und Wender, *Neuronale Netze*, S. 36/37.

3.4.2 Die Delta-Regel

Bei der Delta-Regel wird der Output von Output-Units mit dem Zielwert verglichen. Der Zielwert muss bereits vorhanden sein, es handelt sich also um Supervised learning. Die Berechnung des beobachteten Wertes erfolgt über die Propagierungs-, die Aktivitäts-, und die Output-Funktionen der jeweiligen Units. Als Formel kann der sogenannte Delta-Wert wie folgt dargestellt werden:

$$\delta = a_{Zielwert} - a_{Beobachtungswert}$$

Aus dieser Formel können nun drei mögliche Ausgänge hergeleitet werden. Ist der Delta-Wert größer als Null, dann ist die beobachtete Aktivität zu niedrig und die Werte der Gewichte müssen erhöht werden. Im umgekehrten Fall, also wenn Delta kleiner als Null ist, müssen die Gewichte der betroffenen Verbindungen abgeschwächt werden. Ist Delta exakt Null, muss keine Änderung vorgenommen werden. Diese drei Fälle werden wie folgt als mathematische Formel zusammengefasst(vgl. Rey und Wender, *Neuronale Netze*, S. 38):

$$\Delta w_{ij} = \varepsilon \cdot \delta_i \cdot a_j$$

Die Veränderung des Gewichts zwischen der sendenden Unit j und der empfangenden Unit i wird als Δw_{ij} bezeichnet und ist das Produkt des Delta-Werts der Output-Unit i, dem Aktivitätslevel der sendenden Unit a_j und einem festgelegten Lernparameter ε. Die Formel sorgt dafür, dass die Gewichte der Units, welche einen größeren Einfluss auf den Delta-Wert haben, stärker verändert werden und dass die Veränderung proportional zu δ erfolgt.

Diese Regel wird so lange auf ein neuronales Netz angewandt, bis eine festgelegte Anzahl an Durchläufen erreicht wurde oder bis der Delta-Welt unrelevant klein wird. Da die Lernregel lediglich für supervised learning verwendet werden kann, d.h. die gewünschten Zielwerte müssen vorliegen, kann die Delta-Regel nur für neuronale Netze ohne Hidden-Layer verwendet werden. Befinden sich Hidden-Units im Netz, ist der Zielwert für diese Units unbekannt, bekannt sind anfangs nur die Zielwerte der Output-Units. Diesem Problem kann mithilfe des in Kapitel 3.4.4 vorgestellten Backpropagation-Algorithmus entgegengewirkt werden.[20]

3.4.3 Competitive Learning

Bei vielen Anwendungsmöglichkeiten künstlicher neuronaler Netze steht man vor dem Problem, die korrekten Zielwerte nicht zu kennen. Daher reicht hier z.B. die Delta-Regel, wo bekannte Zielwerte eine Vorraussetzung sind, es sich also um *supervised learning* handelt, nicht aus. Es wird eine Methode für *unsupervised learning* benötigt, das Netzwerk muss in der Lage sein selbst entscheiden zu können, welcher Output am besten zu welchem Input passt.

Realisiert werden kann dies unter anderem durch *Competitive Learning*, also das Lernen durch Wettbewerb. Die einzelnen Units "wetteifern" um das Recht für das Weiterleiten eines Input-Reizes. Nur eine Unit sendet weiter, die Aktivität aller anderen Units des selben Layers wird blockiert.[21] Während des Lernprozesses kann zwischen drei Phasen unterschieden werden (vgl. Rey und Wender, *Neuronale Netze*. S. 54):

- In der **Excitation**-Phase wird für jede Output-Unit der sogenannte Netzinput, der Gesamt-Input, den eine Unit erhält, berechnet. Zuvor werden die Gewichte des Netzwerks mit Zufallswerten initialisiert.

[20] vgl. Rey und Wender, *Neuronale Netze*, S. 39.
[21] vgl. Rojas, *Neural networks: a systematic introduction*, S. 99.

- Während der **Competition**-Phase werden die in der ersten Phase berechneten Netzinput-Werte der verschiedenen Units verglichen. Die Unit mit dem größten Wert gewinnt und darf weitersenden.

- Im letzten Schritt findet das **weight adjustment**, das Anpassen der Gewichte, statt. Es werden nur Gewichte verändert, die auch in Verbindung zur Gewinner-Unit stehen, alle anderen bleiben unverändert("the winner takes it all"). Die Gewichte werden dem Inputmuster ähnlicher gemacht.

$$\Delta w_{ij} = \varepsilon \cdot (a_j - w_{ij})$$

Abbildung 5: Formel zur Gewichtsanpassung

Die Gewichtsveränderung wird mithilfe der in Abbildung 5 dargestellten Formel durchgeführt. Dabei ist a_j der Aktivitätslevel bzw. Output der jeweiligen, zur Gewinner-Unit sendenden Unit, w_{ij} das Gewicht zwischen diesen beiden Neuronen und ε ein frei wählbarer Lernparameter (wird $\varepsilon = 1$ gewählt, vereinfacht sich die Formel zu $\Delta w_{ij} = a_j - w_{ij}$). Wie bei der Delta-Regel ergeben sich drei Möglichkeiten: Ist das Gewicht niedriger als die beobachtete Aktivität, wird das Gewicht erhöht, ist es größer, so wird es gesenkt und sind Gewicht und Aktivität gleich groß, erfolgt keine Veränderung.[22]

Ein relativ einfach lösbares Problem des Algorithmus stellt das "the winner takes it all"-Prinzip dar. Denn es kann vorkommen, dass eine einzelne Unit alle Inputreize bekommt, oder dass vereinzelte Units überhaupt keine Inputmuster gewinnen ("Dead Units"), was aber durch Fixieren einer absoluten Größe der Gewichte, mithilfe von Umverteilungen der Gewichtsmatrix, gelöst werden kann. Im Vergleich zu anderen Lernalgorithmen ist das Competitive Learning auch eher biologisch plausibel, denn wie beim Gehirn können auch durch diesen Algorithmus Redundanzen innerhalb der Inputmuster beseitigt werden und inhibitorische Neuronen simuliert werden. [22]

[22] vgl. Rey und Wender, *Neuronale Netze*, S. 55.

3.4.4 Der Backpropagation Algorithmus

Lernen durch Backpropagation ist wesentlich komplizierter als die zuvor genannten Algorithmen. Backpropagation greift auf das sogenannte Gradientenabstiegsverfahren zurück und ermöglicht *supervised learning* bei Netzwerken mit Hidden-Layern. Die Delta-Regel ist ein Spezialfall des Gradientenverfahrens.

Die Netz-Funktion

Bei Netzwerken mit Hidden-Layern werden Berechnungen schnell komplexer als bei Netzwerken, die nur aus Input- und Output-Layer bestehen. Zur Vereinfachung wird hier der Aktivitätslevel gleich dem Output als a gewählt. Der Netzinput einer Unit, die addierten Werte aller eingehenden Verbindungen, wird berechnet, indem die Aktivitätslevel a_j jeder sendenden Unit mit den Gewichten ihrer Verbindung zur Zielunit w_{ij} multipliziert wird und alle Werte summiert werden. Hier ist N die Anzahl der Units, welche an die Zielunit senden:

$$a_i = \sum_{j=1}^{N} w_{ij} \cdot a_j$$

Im Folgenden wird angenommen, dass das zu berechnende Netzwerk aus einer Input-/Hidden-/Output-Schicht besteht, jede Unit mit jeder Unit des nächsthöheren Layers verbunden ist, die Units des Input-Layers mit $a_1, ..., a_D$, die des Hidden-Layers mit $z_1, ..., z_M$ und die des Output-Layers mit $y_1, ..., y_k$ bezeichnet werden. So kann mithilfe folgender Formeln für jede Unit des Hidden-Layers z_k und für jede Output-Unit y_k der Aktivitätslevel berechnet werden, wobei $h(k)$ und $g(k)$ Aktivitätsfunktionen sind:

$$z_k = h(\sum_{j=1}^{D} w_{z_k j} \cdot a_j)$$

$$y_k = g(\sum_{j=1}^{M} w_{y_k j} \cdot h(z_j))$$

Daraus kann für jede Output-Unit y_k ein Term abgeleitet werden:

$$y_k = g(\sum_{j=1}^{M} w_{kj} \cdot h(\sum_{n=1}^{D} w_{jn} \cdot a_n))$$

Dieser Term ist lediglich von den Gewichten w und den Inputwerten $a_1, ..., a_n$ abhängig. Im Weiteren werden dafür ein Gewichtsvektor w und ein Vektor mit Inputwerten x verwendet, welche durch die an Vektoren angepasste Funktion $y(x, w)$ einen Vektor an Output-Werten zurückgeben.

Die Error-Funktion

Um ein künstliches neuronales Netz mithilfe von supervised learning zu trainieren, muss zuerst die Gesamtfehlerquote (Abweichungen der beobachteten Werte von den korrekten Werten) ermittelt werden. Bei Backpropagation wird eine Funktion verwendet, welche die Summe der Quadrate der Fehlerabweichungen berechnet, kurz *min-squared-error* (vgl. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. S. 233):

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} |y(x_n, w) - t_n|^2$$

Hier wird für jeden Input-Vektor $x_1, ..., x_n$ und den Gewichtsvektor w der jeweilige Output-Vektor des Netzwerks berechnet. Danach wird von diesem Output-Vektor der Vektor mit den gewünschten Werten t_n abgezogen, der Betrag gebildet und quadriert. Die Werte der einzelnen Input-Vektoren werden summiert und ergeben so einen Gesamtfehler $E(w)$. Die Input-Vektoren bleiben konstant während des Trainingsprozesses, weshalb die Funktion $E(w)$ nur vom Gewichtsvektor w abhängt.

Ziel ist es nun den Gesamtfehler zu minimieren. Bei der Error-Funktion handelt es sich allerdings um eine n-dimensionale Funktion (bei $n-1$ Gewichten), daher muss zum Finden eines Minimums das *Gradientenabstiegsverfahren* verwendet werden.

Das Gradientenabstiegsverfahren

Um das Minimum einer differenzierbaren Funktion im $\mathbb{R}^2$ zu finden, muss lediglich die erste Ableitung der Funktion gleich Null gesetzt werden. Bei einer Funktion im $\mathbb{R}^n$ gilt dies nicht mehr. Anstatt das Minimum direkt zu berechnen, was bei Funktionen in $\mathbb{R}^n$ entweder gar nicht oder nur mithilfe von großem Rechenaufwand möglich ist, nähert man sich langsam an das Minimum an:

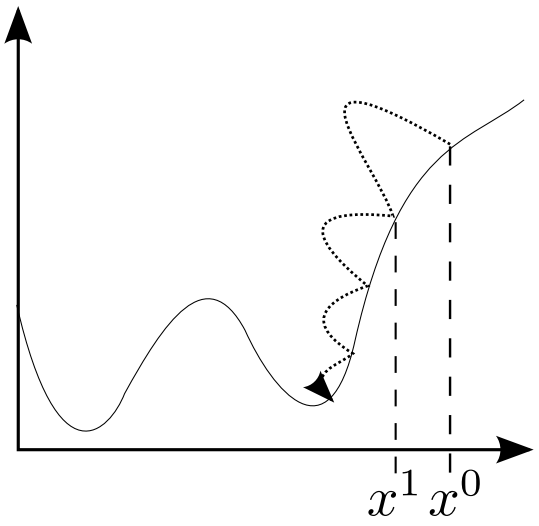


Abbildung 6: Das Minimum wird angenähert.

Diese Annäherung wird durch folgende Formel ausgedrückt:

$$x^{(T+1)} = x^{(T)} - \eta f'(x^{(T)})$$

Der nächste Punkt auf dem Weg zum Minimum ist also ein bereits bekannter Punkt minus der Ableitung an diesem Punkt multipliziert mit einem Lernparameter η, wo-

bei hier T den jeweiligen Iterationsschritt darstellt. Da es sich hier allerdings um eine Funktion im $\mathbb{R}^n$ handelt, kann nicht einfach die Ableitung der Funktion gebildet werden. Stattdessen wird der für dieses Verfahren namensgebende **Gradient** gebildet. Der Gradient ist ein Vektor, der alle partiellen Ableitungen einer Funktion enthält. Bei diesem Vektor handelt es sich nun bei der Error-Funktion um den Gradienten ∇E. Gesucht ist die optimale Gewichtskombination, also wird in jeder Iteration der Gewichtsvektor w aktualisiert. Ändert man nun die vorher genannte Formel auf die Vektoren ab, erhält man (vgl. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. S. 240):

$$w^{(T+1)} = w^{(T)} - \eta \nabla E$$

Das Gradientenverfahren wirft allerdings einige Probleme auf, die während der Benutzung zum Aufsuchen eines Minimums auftreten, denn anders als im $\mathbb{R}^2$ ist im $\mathbb{R}^n$ nur die **lokale Umgebung** bekannt. Mithilfe des Gradienten tastet sich das Verfahren so langsam nach vor und da nur ein Teil der "Funktionslandschaft" bekannt ist, ist es mitunter eher unwahrscheinlich ein globales Minimum zu finden. Außerdem gibt es mehrere Probleme, die vom Lernparameter η abhängen: Wird er zu groß gewählt, werden gute Minima übersprungen oder es kann zu einer Oszillation kommen, also zu einem endlosen "Schwingen" um einen bestimmten Punkt der Funktion. Wird der Parameter zu klein gewählt, kann man kaum Fortschritte beobachten. Um diesen Problemen entgegenzuwirken, kann durch mehrere Versuche ein geeigneter Lernparameter bestimmt werden. Außerdem kann es im Falle der neuronalen Netze helfen, die Gewichte des Netzwerks neu zu initialisieren.[23]

[23] vgl. Rey und Wender, *Neuronale Netze*, S. 44/45.

Backpropagation

Die in diesem Kapitel genannten Erkenntnisse bilden die drei Schritte des Backpropagation-Algorithmus. Man kann grundsätzlich unterscheiden, ob jedes Gewicht des Netzwerks einzeln angepasst wird, oder ob ein Gewichtsvektor verwendet wird. Bei Ersterem fällt die Error-Funktion weg, was mit einer Anpassung des Gradientenverfahrens verbunden ist. Die drei Phasen von Backpropagation werden hier mithilfe eines Gewichtsvektors beschrieben (vgl. Rey und Wender, *Neuronale Netze*. S. 52):

1. **Forward-pass:** Mithilfe der Netzfunktion wird für alle Inputreize der Output berechnet.

2. **Fehlerbestimmung:** Die Error-Funktion berechnet den Gesamtfehler des Netzwerks. Da diese Error-Funktion bereits die Netzfunktion enthält, fällt bei Verwendung eines Gewichtsvektors der Forward-pass in diesen Schritt hinein.

3. **Backward-pass:** Die Gewichte werden von der Output- zur Input-Schicht hin angepasst, also rückwärts. Bei Verwendung eines Gewichtsvektors wird hier lediglich das Gradientenverfahren angewendet und die Gewichte nahezu zeitgleich angepasst.

4 Anwendung zur Mustererkennung

4.1 Planung und Umsetzung

Dieses Kapitel behandelt die Umsetzungsmöglichkeiten der in Abschnitt 3 behandelten Themen mithilfe der Interpretersprache Python und des dazugehörigen Modules PyBrain.

4.1.1 Planen der Netzstruktur

Vor dem Trainieren und der schlussendlichen Benutzung von künstlichen neuronalen Netzen steht an erster Stelle das Wählen eines Netztyps und das Planen der Netzstruktur, hier vor allem bezüglich Fähigkeit zur Mustererkennung. Das einfachste Netz zum Erkennen von Mustern ist der in Abschnitt 3.3.1 vorgestellte Pattern-Associator. Dieser sollte mithilfe einer Input- und einer verknüpften Output-Schicht die Input-Bilder einem Ergebnis zuordnen, allerdings erwiesen sich die Ergebnisse dieses Modells bei den in Abschnitt 4.2 vorgestellten Versuchen als nicht brauchbar. Die Erkennungsrate eines trainierten Pattern-Associator Netzes lag hier bei 40% und darunter.

Abhilfe schafft das Einfügen eines Hidden-Layers in die Netzstruktur, das Verwenden eines Multilayer-Perceptrons (Abschnitt 3.3.2). Bei einer kleinen Anzahl an Hidden-Neuronen verbessern sich die Ergebnisse im Vergleich zum Pattern-Associator nur gering, teilweise fällt die Erkennungsrate auch niedriger aus. Die Anzahl an nötigen Hidden-Layern hängt von der Komplexität der Input-Daten ab und die perfekte Anzahl kann nur empirisch bestimmt werden.

Anzahl der Neuronen

Während die Anzahl der Hidden-Neuronen variabel ist sind Anzahl von Input- und Output-Neuronen bereits festgelegt. In Abschnitt 4.2 werden Bilder als Input für das Netzwerk verwendet. Ein Bild kann in seine Pixel zerlegt werden, von denen jedes drei Werte für Rot-, Grün- und Blau-Anteil besitzt. Bei einem Bild mit der Auflösung 8x8 Pixel, also 64 Pixel, sind das 192 Werte. Für jeden dieser Werte wird eine Input-Unit benötigt, um die Informationen ins Netz zu schicken, d.h. die Anzahl der Neuronen im Input-Layer wird gebildet aus der Anzahl der Werte, die benötigt werden um, in diesem Fall, ein Bild zu laden. Die Anzahl der Output-Units ist nun gleich der Anzahl der Gruppen, in die das Netzwerk alle Input-Daten kategorisieren soll. Werden einem neuronalen Netz beispielsweise Bilder von handgeschriebenen Ziffern (0-9) präsentiert, so gibt es 10 Output-Neuronen, für jede Ziffer eine Unit. Die Output-Unit, die für ein Input-Bild den höchsten Wert errechnet, repräsentiert seine Ziffer. Bei einer anderen Variante kommt ein Netzwerk mit nur einem Output-Neuron aus, egal in wie viele Gruppen der Input klassifiziert werden soll: Die einzelne Unit errechnet einen Wert, und die einzelnen Gruppen können anhand dieser Werte unterschieden werden.

Zusätzlich zu Input-, Hidden- und Output-Units wird auch meistens eine einzelne Bias-Unit verwendet, wie in Abschnitt 3.2.1 beschrieben. Durch diese einzelne Unit, die einen festen Wert besitzt und deren Gewichte mit allen Hidden-Neuronen verbunden sind (jedoch nicht umgekehrt, da eine Verbindung von Hidden- zu Bias-Units eine rückgekoppelte Verbindung darstellen würde und es sich hier um ein Feedforward-Netz handelt), können die Ergebnisse von mehrschichtigen Feedforward-Netzen mit wenig zusätzlichem Aufwand verbessert werden.

Wahl der Übertragungsfunktionen

Wie aus 3.2.1 bekannt, besitzt eine Unit drei Funktionen: die Propagierungsfunktion, die Aktivitäts- oder Übertragungsfunktion und die Output-Funktion. In fast allen

Anwendungen ist die Propagierungsfunktion eine Summenfunktion, welche alle Input-Verbindungen addiert, und die Output-Funktion eine Funktion, die lediglich den durch die Aktivitätsfunktion berechneten Wert weitergibt. Was bleibt, ist die wählbare Aktivitätsfunktion. Beispiele für brauchbare Aktivitätsfunktionen sind z.B. eine logistische Sigmoidfunktion, eine Tangens Hyperbolicus Aktivitätsfunktion oder eine normalverteilte Aktivitätsfunktion.[24]

$$\text{Logistisch-Sigmoid: } a_i = \frac{1}{1 + e^{-input_i}} \qquad \text{Tangens Hyperbolicus: } a_i = \frac{e^{input_i} - e^{-input_i}}{e^{input_i} + e^{-input_i}}$$

$$\text{Normalverteilt: } a_i = \frac{1}{\sqrt{2 \cdot \pi}} \cdot e^{\frac{-netinput_i^2}{2}} \qquad \text{Softmax: } a_i = \frac{e^{input_i}}{\sum_{j=1}^{n} e^{a_j}}$$

Abbildung 7: Mathematische Darstellung von Aktivitätsfunktionen. Zur Softmax Funktion: a_j stellt den Output der Neuronen dar, die an die Unit, welche die Berechnung durchführt, senden.

Die Wahl der Übertragungsfunktionen ist stark problemabhängig. Zur Musterklassifizierung eignet sich am besten eine logistische Sigmoidfunktion oder die Softmax-Funktion. In der Regel wird nicht für jede Unit eine andere Übertragungsfunktion gewählt, die Übertragungsfunktionen werden nach Layern vergeben. Da für den Input-Layer üblicherweise Input = Aktivitätslevel = Output gewählt wird, wird hier keine Aktivitätsfunktion benötigt und dieser Layer somit nicht zur Anzahl der Layer gezählt.

[24] vgl. Rey und Wender, *Neuronale Netze*, S. 21.

4.1.2 Python & Pybrain

"Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs."[25]

Python ist eine heutzutage weit verbreitete Programmiersprache, welche zu der Gruppe der Interpretersprachen gehört. Dies bedeutet, dass der eigentliche Code nicht in maschinenlesbaren Code übersetzt wird, sondern von einem Interpreterprogramm zur Laufzeit ausgeführt wird. Dadurch ergibt sich der Vorteil der Systemunabhängigkeit: Anstatt den Code für jedes System zu übersetzen, reicht es, nur das Interpreterprogramm zu portieren. So können Programme, die in Python oder einer anderen Interpretersprache verfasst sind, ein breites Spektrum an Systemen erreichen, ohne gravierende Änderungen vornehmen zu müssen.

Eine weitere Besonderheit von Python ist das **Modulsystem**. Anstatt für jedes Problem eine neue Anwendung von Grund auf zu entwickeln, gibt es eine große Anzahl von Modulen, die einen großen Anwendungsbereich abdecken und für eigene Anwendungen verwendet werden können. Solche Module sind z.B. NumPy für mathematische Berechnungen oder Matplotlib zum Plotten von Informationen.

Das relevanteste Modul für neuronale Netze ist allerdings **PyBrain**:

"PyBrain is a modular Machine Learning Library for Python. Its goal is to offer flexible, easy-to-use yet still powerful algorithms for Machine Learning Tasks and a variety of predefined environments to test and compare your algorithms. "[26]

[25] o.V.: *The Python Language Reference.* online unter http://docs.python.org/release/2.6.8 /reference/ index.html (zugegriffen am 12. September 2012, 16:00 Uhr).

[26] o.V.: *PyBrain v0.3 Documentation.* online unter http://pybrain.org/docs/index.html (zugegriffen am 24. September 2012, 15:50 Uhr).

PyBrain ist eine Bibliothek an Modulen für maschinelles Lernen in Python. Es enthält Algorithmen und Funktionen für den Einsatz von neuronalen Netzen, supervised, unsupervised und reinforcement learning. Ziel von PyBrain ist es, eine einfache Python-Schnittstelle für künstliche neuronale Netze mit Studenten oder Vortragenden als Zielgruppe bereitzustellen.

Im Folgenden werden Python und die Module *PyBrain*, *Numpy*(wissenschaftliche Berechnungen) und *Matplotlib* für das Erzeugen, Trainieren und Analysieren von künstlichen neuronalen Netzen benützt.

4.1.3 Erstellen einfacher Netze mit Python und PyBrain

Python und PyBrain warten nicht nur mit einer unkomplizierten Syntax, sondern auch mit einfachen Befehlen zum Erstellen einfacher neuronaler Netze auf. Bevor künstliche neuronale Netze erstellt werden, wird ein Datenset aus vorhandenen Daten erstellt, in diesem Fall Bilder. Da hier nur Supervised-Learning Netze verwendet werden, enthält ein Datensatz die Input-Information sowie den gewünschten Output. Ein Bild der Ziffer '0' enthält also sowohl die Bildinformationen der Ziffer als auch die Beschreibung, die dem neuronalen Netz vermittelt, dass auf dem Bild eine '0' ist. Wurden alle Daten in ein Datenset verwandelt, wird dieses aufgespalten in ein Trainingsset und ein Testset. Das Trainingsset wird zum Trainieren des Netzes verwendet, mit dem Testset wird die Generalisierungsfähigkeit, also ob das Netzwerk auch neue Bilder richtig assoziiert, kontrolliert. Es folgen die Schritte zum Erstellen eines künstlichen neuronalen Netzes mit PyBrain(ohne das Erstellen der Datensets)

1. Erstellen eines "leeren" Feedforward-Netzes

```
# Anlegen eines neuen FeedForward-Netzwerks
net = FeedForwardNetwork()
```

2. Hinzufügen der Input-Schicht. Der Input-Layer enthält genau so viele Units, wie die Anzahl von Input-Werten eines Trainingsset-Datensatzes

```
# Erzeugt eine Input-Schicht, die so viele Neuronen besitzt,
# wie vom Trainingsset benötigt wird.
# Die Übertragungsfunktion ist linear.
net.addInputModule(LinearLayer(trainingset.indim, name="input"))
```

3. Hinzufügen der Hidden-Schicht. Die Anzahl der Units im Hidden-Layer wird in einer Variable namens "hidden_neurons" definiert. Alle Neuronen erhalten eine logistische Sigmoidaktivitätsfunktion.

```
# Erzeugt eine Hidden-Schicht mit n-Neuronen, die eine sigmoide
# Übertragungsfunktion besitzen
net.addModule(SigmoidLayer(hidden_neurons, name="hidden"))
```

4. Hinzufügen der Output-Schicht. Die im Trainingsset definierte Anzahl von zu unterscheidenden Fällen gibt die Anzahl der Output-Units vor. Alle Neuronen erhalten eine logistische Sigmoidaktivitätsfunktion.

```
# Erzeugt eine Output-Schicht, die so viele Neuronen besitzt,
# wie vom Trainingsset benötigt wird.
# Die Übertragungsfunktion ist sigmoid.
net.addOutputModule(SigmoidLayer(trainingset.outdim, name="
    output"))
```

5. Dem Netzwerk wird eine Bias-Unit hinzugefügt.

```
# Fügt eine Bias-Schicht hinzu
net.addModule(BiasUnit(name="bias"))
```

6. Die einzelnen Schichten werden verknüpft. Dabei werden alle Units mit allen Units der nächsten Schicht verbunden.

```
# Erzeuge Verbindungen zwischen den einzelnen Neuronen
# Verbindungen: Von Input zu Hidden, von Hidden zu Output,
# Hidden zu Bias
net.addConnection(FullConnection(net['input'],net['hidden'],
    name="input_to_hidden"))
net.addConnection(FullConnection(net['hidden'],net['output'],
    name="hidden_to_output"))
net.addConnection(FullConnection(net['bias'],net['hidden'], name
    ="bias_to_hidden"))
```

7. Abschließend müssen die einzelnen Module sortiert und dadurch das Netz benutzbar gemacht werden.

```
# Das konstruierte Netzwerk sortieren und dadurch benutzbar
    machen
net.sortModules()
```

Sind diese Schritte erledigt, so kann PyBrain mithilfe eines **Trainers** trainiert werden. Mit jedem Trainigsdurchlauf, auch Epoche genannt, wird das Netzwerk besser und besser trainiert, bis ein Minimum an Fehlern erreicht wird. Danach kann das Netzwerk benutzt werden, um die Art von Bildern, auf die es trainiert wurde, zu erkennen.

4.2 Empirische Versuche

4.2.1 Erkennung von handgeschriebenen Ziffern

Bei diesem Versuch ging es darum, ein künstliches neuronales Netz zu konstruieren, welches in der Lage sein sollte, Bilder von handgeschriebenen Ziffern von 0-9 zu erkennen. Dabei wurden insgesamt 5620 Bilder von Ziffern verwendet, davon 3823 zum Training und 1797 zum Testen der Generalisierungsfähigkeit. Die Auflösung der Bilder betrug 16x16 Pixel und sie lagen in Graustufen vor, woraus eine geringere Anzahl an Input-Units des Netzes resultierte.

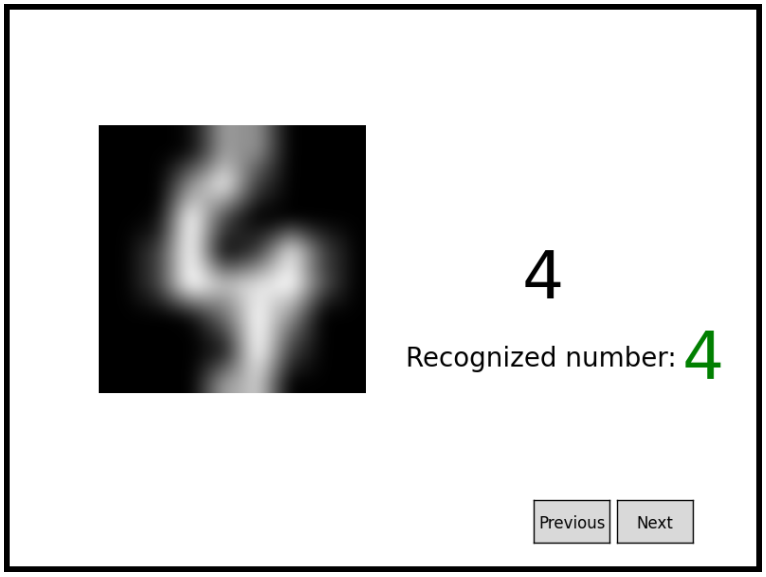


Abbildung 8: Ein Ergebnis des trainierten Netzes visuell ausgegeben.

Verwendet wurde ein 2-Layer Feedforward-Network. Der Input-Layer bestand aus 256 Units für 256 Graustufenwerte (0.0-1.0) der Bilder, der Output-Layer umfasste 10 Neuronen, jeweils für die Ziffern von 0-9. Der Versuch wurde mehrmals durchgeführt, wobei die Anzahl der Hidden-Units variierte. Trainiert wurden Netze mit 7, 8 und 9 Hidden-Neuronen, alle über mehr als 27000 Epochen (Trainingsdurchläufe). Input-Neuronen besaßen keine Aktivitätsfunktion, d.h. der Input wurde direkt über die Verbindungen

an die Hidden-Schicht weitergegeben. Dort und im Output-Layer wurden logistische Sigmoidfunktionen als Übertragungsfunktion verwendet.

Trainiert wurde das Netzwerk mithilfe von Backpropagation, wobei das Output-Neuron mit dem höchsten Ausgabewert die erkannte Zahl repräsentierte. Während des Trainingsvorgangs wurde überprüft, wie viele Zahlen aus dem Trainingsset und wie viele Zahlen des Testsets das Netz erkennen konnte, und diese anschließend aufgezeichnet. Hatte das Netz nach einer bestimmten Epoche die höchste Anzahl an korrekt erkannten Zahlen aus dem Testset, ergo die beste Generalisierungsfähigkeit, so wurde es gespeichert.

Das Netz mit 7 Hidden-Units konnte 56,6% der Testdaten erkennen, jenes mit 8 Hidden-Units bereits 86,2% des Testsets und jenes mit 9 Hidden-Units konnte 90,4% der Testdaten korrekt klassifizieren. Das Netz mit 7 Hidden-Units erreichte seine Bestleistung im Testset bei 96 Epochen und seine Bestleistung im Trainingsset bei 2197 Epochen. Auffallend im Verlauf der Lernkurve ist, dass dieses Netzwerk niemals eine konstante Minimumstelle erreichte, sondern immer um ein Minimum oszillierte. Das 8 Hidden-Neuronen Netz erreichte seine Bestleistung bei 876 Epochen im Testset und bei beinahe 10000 Epochen im Trainingsset. Bei 5000 Epochen erreichte das Netz ein stabiles Minimum, ein weiteres stabiles Minimum erreichte das Netzwerk erst ab ca. 12000 Epochen, dieses blieb aber stabil bis zum Ende.

Das Netzwerk mit 9 Hidden-Units erkannte bereits bei 403 Epochen die meisten Zahlen aus dem Testset und bei 3355 Epochen erkannte es 94% aller Zahlen im Trainingsset. Nach ca 4000 Epochen erreichte das Netz ein stabiles Minimum, welches bis auf kleinere Unterbrechungen bis zum Ende des Trainingsvorgangs andauerte. Die Werte der Gewichte variierten von -19 bis +20, mit einem Mittelwert von 0,1. Die Standardabweichung betrug 2,9. Das Verwenden einer Softmax-Aktivitätsfunktion hätte ein Überlaufen der Gewichte verursacht.

4.2.2 Gesichtserkennung

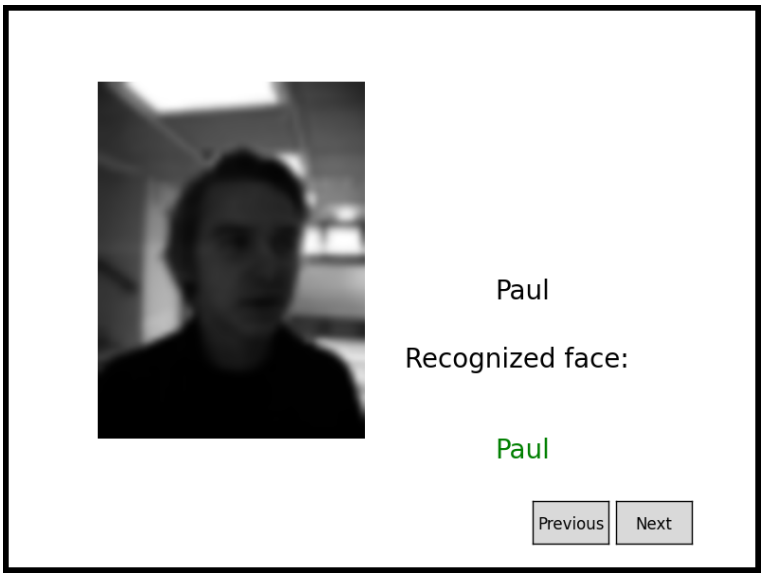


Abbildung 9: Auch bei groben Helligkeitsunterschieden erkennt das trainierte Netz Gesichter.

Der zweite Versuch sollte mithilfe einer annähernd gleichen Netzstruktur wie beim Ziffernversuch die Verwendungsmöglichkeit von neuronalen Netzen zur Gesichtserkennung zeigen. Für den Versuch wurden sechs Schüler jeweils ca. vier Minuten lang gefilmt. Diese Filme wurden in Einzelbilder aufgeteilt und in Graufstufenbilder mit einer Größe von 48x64 Pixeln konvertiert. Danach folgte die Erstellung der Datensets, d.h. das Verarbeiten der Bilder zu einem von PyBrain lesbaren Format, welche mithilfe eines Python-Programms erfolgte. Die Datenmenge war hierbei wesentlich größer als bei dem Ziffernversuch: Insgesamt wurde das neuronale Netz mit insgesamt 16382 Bildern trainiert und die Erkennungsrate des Netzes mit 8190 Bildern überprüft.

Wie auch im ersten Versuch wurde hier ein 2-Layer Feedforward-Network verwendet. Die Anzahl der Input-Neuronen betrug 3072 und repräsentierte damit die Anzahl von Pixeln und deren Graustufenwerte eines Einzelbildes. Output-Neuronen gab es sechs, für jeden der sechs gefilmten Schüler. Anfangs war aufgrund der hohen erforderlichen Rechenleistung nur ein Versuch mit 8 und 9 Hidden-Neuronen geplant, nach den Ergebnissen dieser beiden Netze wurde der Versuch jedoch mit 6, 7 und 10 Hidden-Neuronen wiederholt.

Die Werte der Input-Schicht wurden wieder direkt an die Hidden-Neuronen weitergegeben, von dort durch eine logistische Sigmoidfunktion an die Output-Neuronen geschickt, welche die Daten nochmals mit derselben Funktion bearbeiteten, bevor sie ausgegeben wurden.

Der Versuch lief gleich ab wie der erste Versuch: Das Netzwerk wurde mit Backpropagation trainiert, Statistiken und beste Verteilung der Gewichte wurden aufgezeichnet. Aufgrund der Resultate des Ziffernversuches und wegen großen Rechenaufwands wurden die Netzwerke für die Gesichtserkennung weniger oft trainiert, dennoch sprachen die Ergebnisse für sich.

Zunächst wurden Netzwerke mit 8 und 9 Hidden-Neuronen getestet, da diese im ersten Versuch die besten Resultate lieferten. Das Netz mit 8 Hidden-Neuronen wurde beinahe 6000 Epochen lang trainiert, erreichte seine Bestleistung allerdings bereits bei 3300 Epochen, wo das Netz 93,3% der Testbilder erkannte. Erstaunlicherweise konnte diese Leistung vom neuronalen Netz mit 9 Neuronen im Hidden-Layer nicht erreicht werden, welches seine beste Leistung von 92,6% bereits bei 2800 Epochen zeigte. Damit hatte ein Netz, welches weniger Hidden-Neuronen besaß, wie das beste Netz des Ziffernversuchs im Falle der Gesichtserkennung die besten Werte erzielt.

Daraufhin wurde der Versuch mit 6 (89,9%), 7 (91,7%) und 10 (92,7%) Hidden-Neuronen durchgeführt. Daran zeigte sich, dass sich bei höherer Anzahl von Hidden-Neuronen die Ergebnisse nur geringfügig veränderten bzw. eher verschlechterten. Es geht also bei Mustererkennung nicht darum, möglichst viele Hidden-Neuronen zu verwenden, sondern das beste Maß an Hidden-Neuronen für die Problemstellung zu finden. Außerdem zeigte sich, dass bei Gesichtserkennung ein Netz mit nur 6 Hidden-Neuronen noch immer eine höhere Erkennungsrate besaß, als ein Netz mit 7 Hidden-Neuronen im Ziffern-Versuch, was mit der Qualität der Bilder in Zusammenhang steht. (Vgl.: Im Ziffernversuch besaß ein Bild 256 Pixel, zur Gesichtserkennung wurden 3072 Pixel verwendet.)

Die Gewichte des Netzes mit 8 Hidden-Neuronen lagen in einem Intervall von -28.1

bis +20.1. Der Mittelwert der Gewichte lag bei -0.003, die Standardabweichung betrug 1.495. Keines der Netze zeigte während des Trainingsvorgangs eine instabile Lernkurve auf, die Netze lernten allesamt stabil, die Lernkurve oszillierte geringfügig.

5 Zusammenfassung und Ausblick

Die Rechenleistung von biologischen neuronalen Netzen, welche eine Unmenge an Nervenzellen besitzen, deren Verbindungen sich ständig verändern, kann ein künstliches neuronales Netz, welches sich lediglich mathematischer Recheneinheiten, den Units, bedient, auch heutzutage noch nicht mithalten. Dennoch konnte anhand der vorgestellten Experimente gezeigt werden, dass auch künstliche neuronale Netze bemerkenswerte Leistungen erbringen können. Hierbei steht mit PyBrain ein mächtiges Werkzeug zum Experimentieren mit neuronalen Netzen zur Verfügung und mithilfe der vorgestellten Informationen über Struktur, Netztypen und Lernalgorithmen können einfach künstliche Netze erzeugt werden.

Der in dieser Arbeit vorgestellte Teilaspekt von künstlichen neuronalen Netzen im Anwendungsgebiet der Mustererkennung ist bei weitem kein vollständiger Überblick. Es gibt eine Vielzahl von Netztypen, welche für die Mustererkennung eingesetzt werden können, beispielsweise Kunihiko Fukushimas Cognitron und Neocognitron, welche komplexere Netzstrukturen aufweisen. Auch bei den genannten Netztypen gibt es Weiterentwicklungen. So wurden im Laufe der Zeit eine Vielzahl rekurrenter Netztypen entwickelt oder die Anzahl der benötigten Trainingsdurchläufe mithilfe der Cascade-Correlation Technik beim Backpropagation-Algorithmus erheblich vermindert.

Künstliche neuronale Netze entwickeln sich in der heutigen Zeit zusammen mit Anwendung statistischer Methoden zu mächtigen Werkzeugen der Gesichts-, Bild- oder Mustererkennung im Allgemeinen. Man darf also gespannt sein, inwiefern künstliche neuronale Netze unseren zukünftigen Alltag beeinflussen werden.

Literaturverzeichnis

Bishop, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

Kriesel, David. *Ein kleiner Überblick über Neuronale Netze*. http://www.dkriesel.com. 2007.

Minsky, Marvin und Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

o.V.: *PyBrain v0.3 Documentation.* online unter http://pybrain.org/docs/index.html (zugegriffen am 24. September 2012, 15:50 Uhr).

— *The Python Language Reference.* online unter http://docs.python.org/release/2.6.8 /reference/ index.html (zugegriffen am 12. September 2012, 16:00 Uhr).

Rey, Günter Daniel und Karl F. Wender. *Neuronale Netze. Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. 2. Auflage. Verlag Hans Huber, Hogrefe AG, 2011.

Rojas, Raúl. *Neural networks: a systematic introduction*. Springer-Verlag New York, Inc., 1996.

Eidesstattliche Erklärung zur Fachbereichsarbeit

Ich, Michael Pucher (Klasse: 8B), erkläre hiermit, dass ich die Fachbereichsarbeit ohne fremde Hilfe verfasst und nur die angegebene Literatur verwendet habe.

20. Mai 2013 .....
Datum *Unterschrift*

Schülerprotokoll

Juni 2012	Festlegen des Themas und Absprache mit Betreuer Helmuth Peer.
Sommer 2012	Suche und Auswahl von Fachliteratur. Nach der Auswahl der Literatur wurde auszugsweise gelesen und exzerpiert.
September 2012	Festlegen der ungefähren Gliederung und Einreichung des FBA-Themas.
Oktober 2012	Literaturrecherche und Schreiben des Kapitels "Künstliche neuronale Netze".
November 2012	Verfassen des Kapitels "Die Natur als Vorbild".
Dezember 2012	Beginn des Kapitels "Anwendung zur Mustererkennung" mit anschließender Vorlage der bisherigen Arbeiten an Mag. Peer. Durchführung des ersten Versuches zur Ziffernerkennung.
Jänner 2012	Betreuungsgespräch über bisherige Arbeiten. Durchführung des Versuches zur Gesichtserkennung. Sechs Schüler galten als Versuchspersonen.
Februar 2012	Abschließende Korrekturen.
März 2013	Abgabe der Arbeit.