Back-Propagation Algorithmus und genetische Algorithmen zum Training neuronaler Netze

Bachelorarbeit
zur Erlangung des Grades *Bachelor of Science*

an der
Hochschule Niederrhein
Fachbereich Elektrotechnik und Informatik
Studiengang *Informatik*

vorgelegt von
Evgenij Helm
838231

Datum: 11. September 2014

Prüfer: Prof. Dr. Peer Ueberholz
Zweitprüfer: Prof. Dr. Christoph Dalitz

Erklärung

Ich versichere durch meine Unterschrift, dass die vorliegende Abschlussarbeit ausschließlich von mir verfasst beziehungsweise angefertigt wurde. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Krefeld, den _____ _____

Evgenij Helm

Zusammenfassung

Neuronale Netze finden in jüngster Zeit immer mehr Verwendung, da sie bei herausfordernden Problemen oft bessere Ergebnisse als konkurrierende Lernverfahren liefern. In der vorliegenden Arbeit wird das Training neuronaler Netze zur Klassifizierung von Daten mit dem Back-Propagation Algorithmus und genetischen Algorithmen untersucht. Zu diesem Zweck werden neuronale Netze, der Back-Propagation Algorithmus und genetische Algorithmen näher betrachtet und Erkenntnisse anderer Arbeiten zu diesem Thema zusammengetragen.
Anschließend wird eine Auswahl freier populärer neuronaler Netzwerk Bibliotheken verglichen und eine eigene zum Training mittels Back-Propagation und genetischem Algorithmus entwickelt. Mit der entwickelten Bibliothek und zwei Datenbanken werden die Erkenntnisse anderer Arbeiten überprüft und andere Ansätze durch Kombination verschiedener Heuristiken untersucht.
Abschließende Versuche deuten darauf hin, dass genetisches Training kein generell besseres Lernverhalten als der Back-Propagation Algorithmus besitzt.

Abstract

Neural networks are recently more and more used, as they often provide better results than competitive learning methods for challenging problems. In the present work, the training of neural networks for the classification of data is investigated with the back-propagation algorithm and genetic algorithms. For this purpose, neural networks, the back-propagation algorithm and genetic algorithms are considered in detail and insights were gathered from other studies on this topic.
Subsequently, a selection of free popular neural network libraries are compared and an own developed for training using back-propagation and genetic algorithm. With the developed library and two databases, the insights from other studies are reviewed and other approaches examined by combining different heuristics.
Final experiments indicate that genetic training has generally no better learning performance than the back-propagation algorithm.

Inhaltsverzeichnis

1 Einführung

Ein moderner Computer ist in der Abarbeitung von vielen Aufgaben deutlich schneller als ein Mensch, jedoch sind einige Aufgaben bisher kaum lösbar. Ein einfaches und oft gebrachtes Beispiel ist die Unterscheidung von Objekten. Für den Menschen ist es leicht einen Hund von einer Katze zu unterscheiden, wohingegen diese einfach erscheinende Aufgabe für den Computer eine Herausforderung darstellt. Viele Fähigkeiten des Menschen sind antrainiert, die ohne viel Überlegung angewendet werden können und auch auf neue Aufgaben übertragen werden können. So kann ein Mensch ein Objekt auch dann noch als Hund identifizieren, selbst wenn die Rasse für diesen unbekannt ist.
Eine Möglichkeit solche Probleme zu lösen, sind die künstlichen neuronalen Netze, welche ein Forschungsgebiet der künstlichen Intelligenz sind und versuchen, die Arbeitsweise des menschlichen Gehirns nachzubilden. Ähnlich wie beim Menschen sollen künstliche neuronale Netze, mit Hilfe von Training, Lösungen zu bestimmten Problemen erlernen und diese auch auf neue Probleme anwenden können. Probleme, die dafür in Betracht kommen, sind solche, für die es aufgrund ihrer Komplexität oder ihres Umfangs keine algorithmischen Lösungen gibt. Übliche in Betracht kommende Probleme sind zum Beispiel: Erkennung von Anomalien in Sequenzen, Prognosen oder Erkennung von Mustern. Das letzte Beispiel wird in dieser Arbeit näher betrachtet.

2 Künstliche neuronale Netze

2.1 Biologisches Vorbild

Das Gehirn ist ein sehr komplexes Gebilde, dessen Aufbau und Funktionsweise noch nicht ganz erfasst wurde und somit auch noch nicht annähernd auf einem Computer modellierbar ist. Deshalb beschränkt man sich auf den wesentlichen Baustein, die biologischen Nervenzellen oder auch Neuronen.

2.2 Idealisiertes Neuronen Modell

Oft ist es einfacher die komplexe Realität der wirklichen Welt zu vernachlässigen und sich an einem idealisierten Model zu orientieren, so auch bei den Neuronen. Im Folgenden wird eine vereinfachte Arbeitsweise eines Neurons beschrieben, welche jedoch viele Aspekte, zum Beispiel die zeitliche Komponente innerhalb des Neurons, nicht betrachtet.
Beim idealisierten Model eines Neurons (siehe Abbildung 1) sind drei Hauptstrukturen zu unterscheiden, und zwar Dendriten, Zellkörper und Axon, denen man folgende Aufgaben zuordnen kann: Eingabe, Verarbeitung und Ausgabe. Die Dendriten summieren die Ausgabesignale der eingehenden Neuronen in Form von elektrischen Signalen auf und leiten diese an den Zellkörper weiter.
Überschreitet dieses Signal einen für jeden Zellkörper individuellen Schwellwert, erzeugt der Zellkörper einen elektrischen Impuls, welcher vom Axon an andere Neuronen weitergeleitet wird. Die Kontakte zwischen zwei Neuronen werden als Synapsen bezeichnet, diese können das Signal erhöhen oder auch verringern. Womit die Synapsen als Gewichtung des Signals betrachtet werden können.
Damit ergibt der Schwellwert in Kombination mit den Gewichtungen, neben den Verbindungen zu den umliegenden Neuronen, die charakteristische Eigenschaft eines einzelnen Neurons.

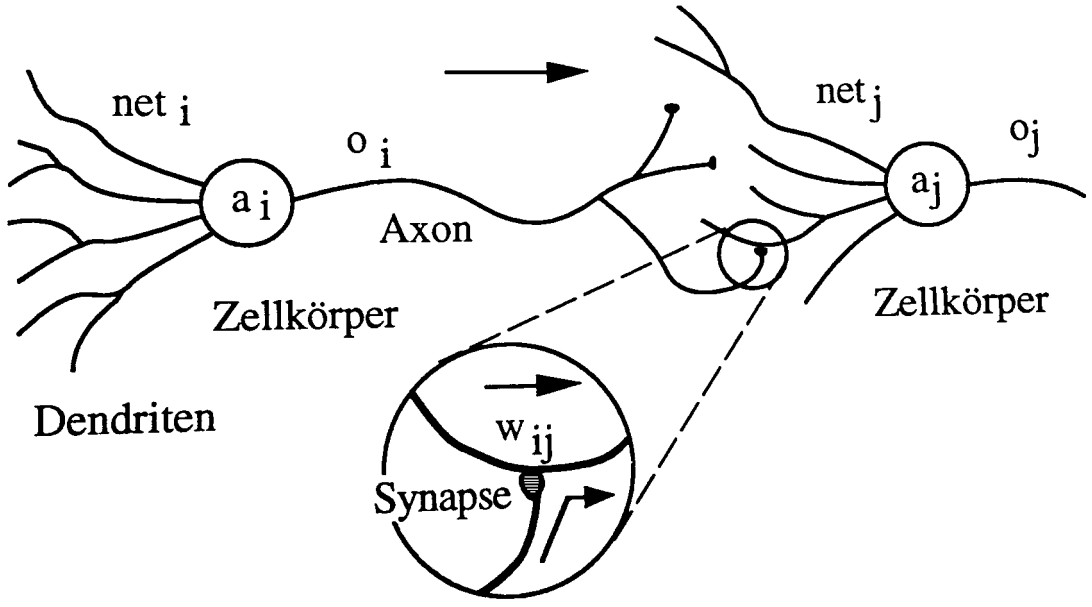


Abbildung 1: Idealisiertes Modell eines Neurons (aus [Zell94]/modifiziert)

2.3 Künstliches Neuronen Modell

So wie Neuronen den zentralen Baustein des Nervensystems bilden, stellen künstliche Neuronen den Grundbaustein eines künstlichen neuronalen Netzes dar.
Wie in Abbildung 2 dargestellt, besteht ein künstliches Neuron aus mehreren Werten und Funktionen. Eingehende Werte $(x_1, \ldots, x_n)$ werden gewichtet $(w_{1,j}, \ldots, w_{n,j})$, was den Synapsen entspricht und durch eine Propagierungsfunktion f_{prop} zur Netzeingabe net_j verarbeitet. Dies stellt das Äquivalent zum Dendriten dar.
Der Zellkörper wird mithilfe einer Aktivierungsfunktion f_{act} und eines Schwellenwertes θ simuliert, welcher aus der Netzeingabe net_j ein Aktivierungszustand a_j ermittelt. Aus der Aktivierung wird über eine Ausgabefunktion f_{out} die Ausgabe o_j des Neurons berechnet, was beim biologischen Vorbild dem Axon entspricht.

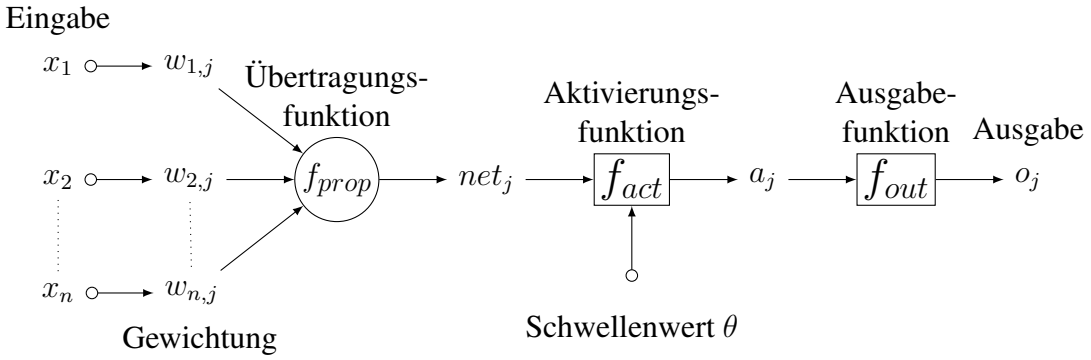


Abbildung 2: Modell eines künstlichen Neurons

2.3.1 Eingabeinformationen

Der Eingabevektor $(x_1, ..., x_n)$ wird entweder durch vorherige Neuronen bestimmt und übergeben oder entstammt der Umgebung des künstlichen neuronalen Netzwerks. Dies können zum Beispiel die einzelnen Pixel Grauwerte eines Bildes sein.

2.3.2 Propagierungsfunktion

Den nächsten Schritt bildet die Propagierungsfunktion f_{prop}, diese berechnet aus dem Eingabevektor $(x_1, \ldots, x_n)$ und dem Gewichtungsvektor $(w_{1,j}, \ldots, w_{n,j})$ für das Neuron j die Netzeingabe net_j:

$$net_j = f_{prop}(x_1, \ldots, x_n, w_{1,j}, \ldots, w_{n,j}).$$

Für gewöhnlich wird als Propagierungsfunktion das aufsummierte Produkt aus den einzelnen Gewichten und den entsprechenden Eingabewerten verwendet:

$$net_j = \sum_{i=1}^{n} x_i w_{i,j}.$$

2.3.3 Aktivierungsfunktion

Die Aktivierungsfunktion kann allgemein definiert werden als

$$a_j(t) = f_{act}(net_j(t), a_j(t-1), \theta_j)$$

und verarbeitet damit die Netzeingabe net_j, den vorherigen Aktivierungszustand $a_j(t-1)$ und den Schwellenwert θ_j. Der vorherige Aktivierungszustand $a_j(t-1)$ wird jedoch nur bei Netzen mit Rückkoppelungen benötigt z. B. Hopfield-Netzen, welche hier nicht weiter betrachtet werden.
Das biologische Neuron erzeugt beim Überschreiten des Schwellwerts eine konstante Ausgabe was einer Stufenfunktion als Aktivierungsfunktion entsprechen würde (siehe Abbildung 3). Einige Lernverfahren erwarten jedoch stetige differenzierbare Funktionen, weshalb oft Sigmoide Funktionen (siehe Abbildung 4) verwendet werden. Dazu zählen unter anderem logistische Funktionen und hyperbolische Tangensfunktionen. Alle Funktionen haben jedoch gemeinsam, dass der Schwellwert θ die Stelle der größten Steigung markiert. Zudem kann man z.B. die Tangens Hyperbolicus Funktion durch Stauchung beliebig nah an die Stufenfunktion annähern. Eine der gängigsten Aktivierungsfunktionen ist die logistische Funktion:

$$f_{act_{log}}(net_j) = \frac{1}{1 + exp(-net_j + \theta_j)}$$

oder die Tangens Hyperbolicus Funktion:

$$f_{act_{tanh}}(net_j) = tanh(net_j - \theta_j).$$

Da der Schwellenwert einer Verschiebung der Funktion entlang der x-Achse entspricht, ist es praktischer diesen einfach vom Netzeingang, zu subtrahieren:

$$net_j = w_{0,j} + \sum_{i=1}^{n} (x_i w_{i,j}) \text{ mit } w_{0,j} = -\theta_j.$$

Oft wird zum Eingabevektor ein weiteres Element x_0 mit der Konstanten 1 hinzugefügt, um net_j wie folgt definieren zu können.

$$net_j = \sum_{i=0}^{n} (x_i w_{i,j}) \text{ mit } w_{0,j} = -\theta_j \text{ und } x_0 = 1$$

Die Abbildung 6 visualisiert unter anderem diese Änderung. Als Quelle der Konstanten 1 werden normalerweise weitere Neuronen, welche als Bias Einheit bezeichnet werden, dem Netzwerk hinzugefügt. Diese erhalten selber keine Eingaben und dienen lediglich zum Propagieren der Konstanten. Dadurch ist es nicht mehr nötig den Schwellwert in der Aktivierungsfunktion, zu berücksichtigen:

$$a_j(t) = f_{act}(net_j(t), a_j(t-1)).$$

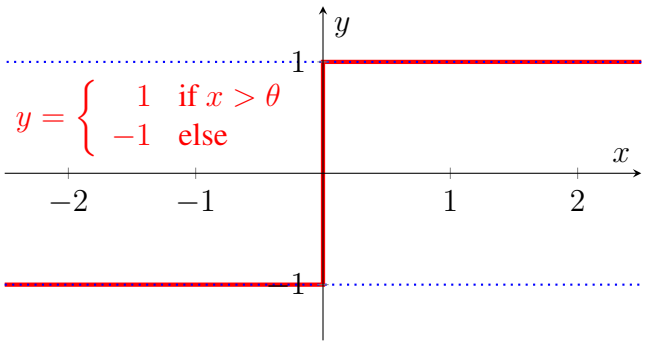


Abbildung 3: Stufenfunktion mit $\theta = 0$

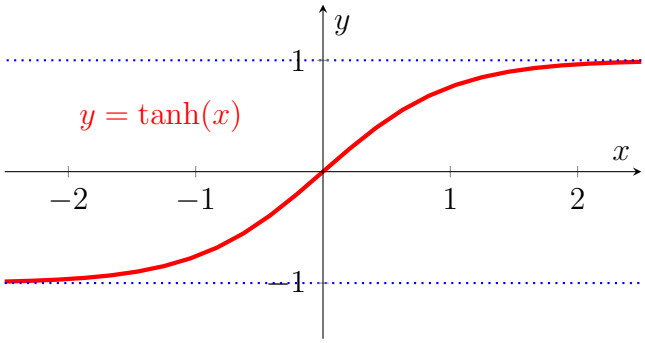


Abbildung 4: Beispiel für eine Sigmoide Funktion: Tangens Hyperbolicus Funktion

2.3.4 Ausgabefunktion

In einigen Fällen wird die Ausgabefunktion zur Skalierung der Ausgabe verwendet. Normalerweise entspricht diese jedoch der Identitätsfunktion:

$$f_{out}(a_j) = a_j \text{ , womit } o_j = a_j.$$

2.4 Netzwerkarchitektur

Ein künstliches neuronales Netz besteht aus einer Menge von Neuronen, welche durch gerichtete und gewichtete Verbindungen miteinander verbunden sind. Im biologischen Vorbild ist jedes Neuron mit einer sehr hohen Anzahl benachbarten Neuronen verbunden. In der Praxis mit künstlichen neuronalen Netzen hat sich jedoch gezeigt, dass dies zu keinen guten Lösungen führt. Für gewöhnlich werden die neuronalen Netze in Schichten angeordnet (siehe Abbildung 5).

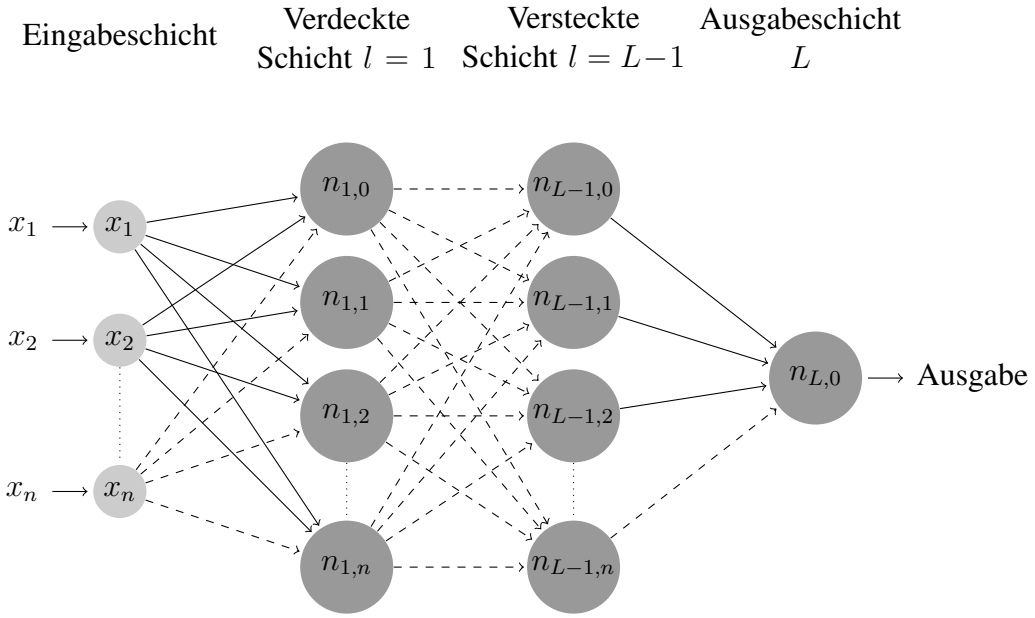


Abbildung 5: Grundlegende Netzwerkarchitektur eines schichtenweise verbundenen feedforward Netzes. $n_{l,j}$ bezeichnet das j-te Neuron auf der l-ten Schicht, die Eingabeschicht dient in dieser Arbeit lediglich zur Weitergabe der Eingabe und besitzt somit keine Neuronen. Diese Definition kann je nach Literatur abweichen. Alle Schichten zwischen der Ausgabeschicht und der Eingabeschicht werden als verdeckte Schicht *(engl. hidden layer)* bezeichnet. Ab $L = 3$ spricht man von einem *deep neural network*.

Durch die unterschiedlichen Verbindungen zwischen und innerhalb der Schichten werden verschiedene Architekturen definiert.
In dieser Arbeit werden ausschließlich vorwärts gerichtete Netzwerke *(engl. feedforward networks)* verwendet. Diese definieren sich dadurch, dass die Neuronen innerhalb einer Schicht *(engl. layer)* l_i nur mit Neuronen der nachfolgenden Schichten l_{i+k} mit $k > 0$ verbunden sind und die Ausgaben eines Neurons nur in die nachfolgende Schichten propagiert werden. Eine weitere Einschränkung bilden die schichtenweise verbundenen feedforward Netze. Hier ist ein Neuron nur mit Neuronen der nachfolgenden

Schicht l_{i+1} verbunden. Diese Architektur wird üblicherweise für Klassifizierungsprobleme verwendet.
Durch verschiedene Anzahl an Schichten und Verbindungen gibt es eine hohe Anzahl unterschiedlicher Architekturen, auf welche hier jedoch nicht weiter eingegangen wird.

3 Training neuronaler Netze

Wie in der Einführung bereits beschrieben, kann man künstliche neuronale Netze trainieren, wodurch diese auch neue Probleme derselben Klasse lösen können. Dies wird als Generalisierung bezeichnet. Unter "Lernen" versteht man bei künstlichen neuronalen Netzen die Veränderung des Netzes. Beispiele für solche Veränderungen sind:

- Erzeugung oder Löschung von Verbinden zwischen Neuronen
- Veränderungen von Verbindungsgewichten
- Veränderungen von Schwellwerten
- Veränderungen der Funktionen innerhalb des künstlichen Neurons
- Neuronen hinzufügen oder entfernen

Die gängigsten Verfahren verändern die Verbindungsgewichte, was den Schwellwert, der als Gewicht mit einer konstanten Eingabe von 1 behandelt wird, einschließt. Zusätzlich ist das Löschen oder Erzeugen von Verbindungen durch das Setzen von Gewichten gleich oder ungleich 0 möglich.
Lernverfahren werden im Wesentlichen in zwei Kategorien aufgeteilt, nicht überwachtes Lernen und überwachtes Lernen.
Beim nicht überwachten Lernen werden lediglich Eingabemuster zum Lernen verwendet, welche das Netz versucht zu identifizieren und in ähnliche Kategorien zu klassifizieren.
Beim überwachten Lernen ist sowohl das Eingabemuster als auch das Ausgabemuster bekannt. Anhand der Differenz zwischen der tatsächlichen Ausgabe des Netzes und des bekannten Ausgabemusters wird das Netz dann trainiert. In dieser Arbeit werden ausschließlich überwachte Lernalgorithmen betrachtet.

3.1 Perzeptron

Den ersten Ansatz für ein künstliches Neuron stellte McCulloch und Pitts (1943) vorgestelltes Perzeptron dar. Das Perzeptron ist in der ursprünglichen Version ein künstliches Neuron mit einer Schwellwertfunktion die eine 1 oder -1 ausgibt und eine lineare Ausgabefunktion besitzt, welche heute jedoch allgemeiner definiert wird. Eine visuelle Darstellung kann man der Abbildung 6 entnehmen.

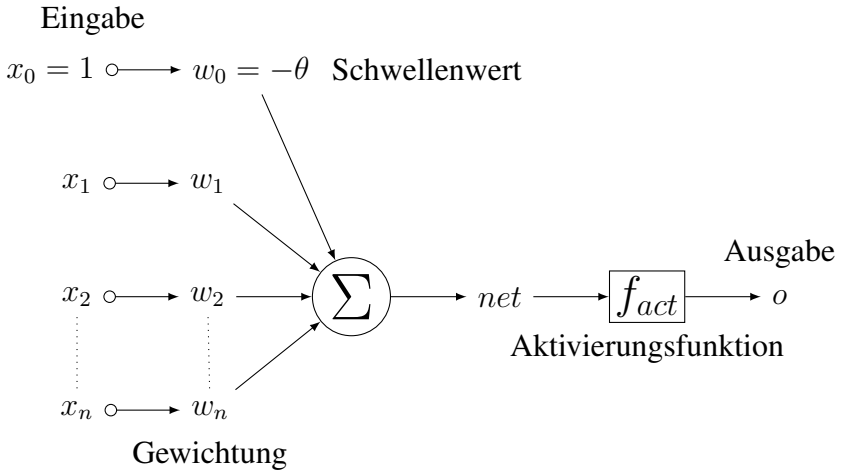


Abbildung 6: Darstellung eines Perzeptrons

Das Perzeptron kann man als Hyperebene sehen, welche den n-dimensionalen Raum teilt und damit eine Klassifizierung vornimmt, indem alle Elemente auf der einen Seite eine Ausgabe von -1 und auf der anderen Seite eine Ausgabe von 1 erzeugen.
Ein Beispiel für so eine Klassifizierung wäre die Teilung der Ausgabe einer logischen UND-Verknüpfung, welches nur bei der Eingabe $(1, 1)$ eine 1 ausgibt. Veranschaulicht man die möglichen Ausgabewerte einer logischen UND-Verknüpfung, so erkennt man leicht, dass diese mit einer einfachen linearen Funktion z. B. $x_2 = -x_1 + 1.5$ separierbar sind (siehe Abbildung 7).

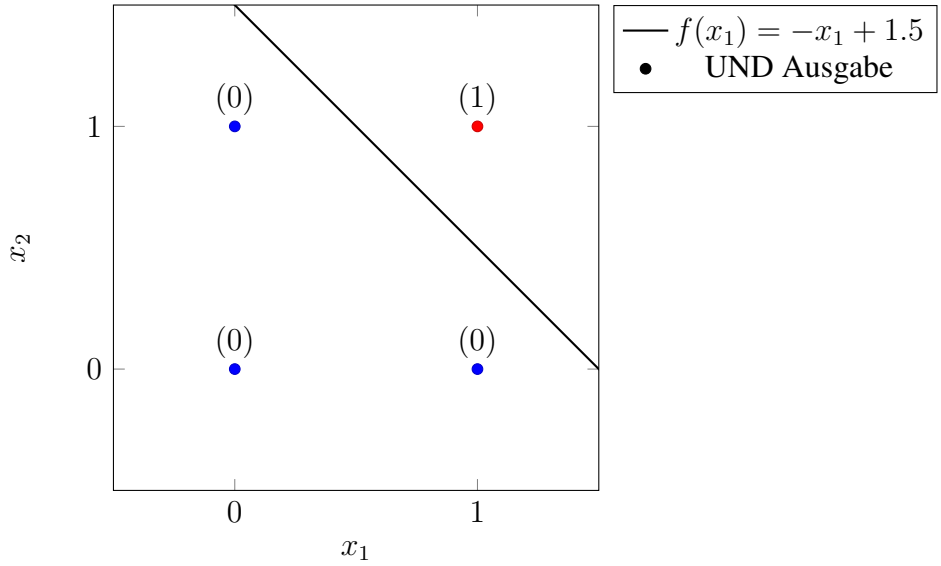


Abbildung 7: Ausgabe einer logischen UND-Verknüpfung

Die lineare Funktion $x_2 = -x_1 + 1.5$ kann auch als Entscheidungsfunktion geschrieben werden $f(x_1, x_2) = x_2 + x_1 - 1.5$, welches sich wiederum als Perzeptron darstellen lässt (siehe Abbildung 8).

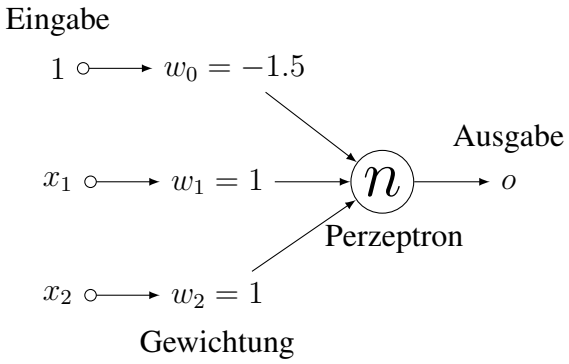


Abbildung 8: Perzeptron zur Klassifizierung der Ausgabe einer logischen UND-Verknüpfung

Für dieses einfache Beispiel konnten die Gewichte durch Betrachtung bestimmt werden, jedoch ist dies für gewöhnlich nicht möglich. Die Lösung, beziehungsweise die Werte der Gewichte, können mithilfe einer der folgenden Lernregeln gefunden werden. Wobei die Gewichte mit zufälligen Werten initialisiert werden.

3.1.1 Perzeptron-Regel

Eine Möglichkeit die unbekannten Gewichte zu finden ist die Perzeptron-Regel. Dabei werden Trainingsdaten mit D Datensätzen, bestehend aus den Eingaben $\vec{x}_d$ und den Ausgaben t_d, in das Perzeptron gegeben und, solange das Ergebnis o nicht der gewünschten Ausgabe t entspricht, die Gewichte modifiziert. Als Initialwerte für die Gewichte w_i werden normalerweise Zufallszahlen genommen, dazu mehr im Abschnitt 3.4.5. Die Änderung, für ein einzelnes Gewicht w_i und einen einzelnen Datensatz kann formal ausgedrückt werden als:

$$w_i = w_i + \Delta w_i \text{ mit } \Delta w_i = \eta(t - o)x_i.$$

Die Ausgabe o ist bei einem Perzeptron von der Eingabe $\vec{x}$ abhängig.

$$o(\vec{x}) = \begin{cases} 1 & \text{wenn } f_{act}(\sum_{i=0}^{n} x_i w_i) > \theta \\ \text{-1} & \text{sonst} \end{cases}$$

Die Idee dahinter ist, dass bei korrekter Ausgabe $t = o$ das Gewicht unverändert bleibt, bei zu niedrigem Wert $o < t$ das Gewicht w_i erhöht wird und bei einem zu hohem Wert $o > t$ das entsprechende Gewicht erniedrigt wird.
Mit der Lernrate η kann die Gewichtsänderung angepasst werden, im Abschnitt 3.3 wird diese genauer betrachtet.

3.2 Delta-Regel

Die folgenden Gleichungen sind teilweise aus dem Buch [Mitchell97] entnommen.

Eine der bekanntesten Lernregeln für überwachtes Lernen, welche auch Grundlage für den Back-Propagation Algorithmus ist, ist die Delta-Regel. Es handelt sich, wie bei der Perzeptron-Regel, um ein Verfahren zur Anpassung der Gewichte innerhalb eines feed-forward Netzes mit nur einer Schicht.
Die grundlegende Idee des Verfahrens besteht darin, den Fehler zwischen der tatsächlichen Ausgabe und der Zielausgabe mit einer Fehlerfunktion zu messen und mithilfe des Gradientenabstiegs diesen zu minimieren. Der Gradient einer Funktion f gibt dabei die Richtung des steilsten Anstiegs an einem Punkt $\vec{x}$ an und wird durch die partielle Ableitung der Funktion f nach allen Elementen des Punktes $\vec{x}$ beschrieben, beim Gradientenabstieg wird dem negativen Gradienten gefolgt. Eine übliche Fehlerfunktion ist der quadratische Fehler:

$$E(w_i) = \frac{1}{2}(t - o)^2 \text{ mit } o = f_{act}(\sum_{i=0}^{n} x_i w_i).$$

Wobei der Faktor $\frac{1}{2}$ nur dazu dient den Faktor 2, der bei der Ableitung hinzukommt, auszugleichen.
Der Fehler E hängt von den Gewichten w_i ab, da die Ausgabe des künstlichen Neurons von diesen abhängt. Natürlich hängt dieser auch von den Trainingsdaten ab, die jedoch für gewöhnlich fest definiert sind.
Zur Bestimmung des Gradienten ist eine partielle Ableitung der Funktion E nach allen Gewichten w_i nötig, was jedoch eine differenzierbare Aktivierungsfunktion $f_{act}(net)$ voraussetzt. Zu beachten ist noch, dass sich der korrekte Gradient über den Gesamtfehler:

$$E_{ges} = \sum_{d=0}^{D} E_d(w_i)$$

bestimmen lässt, jedoch wird im Folgenden nur $E(w_i)$ betrachtet, da sich der Gesamtfehler aus der Summe dieser über alle Trainingsdaten ergibt.
Formal kann der korrekte Gradient beschrieben werden als:

$$\Delta E_{ges}(w_0, \ldots, w_n) = \sum_{d=0}^{D} \left[\frac{\partial E_d}{\partial w_0}, \frac{\partial E_d}{\partial w_1}, \ldots, \frac{\partial E_d}{\partial w_n} \right],$$

beziehungsweise für ein einzelnes Gewicht und nur einen Trainingsdatensatz, was der Projektion des Gradienten auf diese Variable entspricht.

$$\Delta E(w_i) = \left[\frac{\partial E}{\partial w_i} \right]$$

Mithilfe des negativen Gradienten $-\Delta E(w_i)$ kann die Delta-Regel für ein einzelnes Gewicht w_i formal ausgedrückt werden als:

$$w_i = w_i + \Delta w_i \text{ mit } \Delta w_i = -\eta \Delta E(w_i).$$

Wobei η die Schrittweite angibt, die man dem Gradienten folgt. Im Zusammenhang mit Lernregeln wird diese als Lernrate bezeichnet.
$\Delta(w_i)$ kann mithilfe der Kettenregel auch wie folgt ausgedrückt werden:

$$\Delta(w_i) = -\eta \frac{\partial E}{\partial o} \frac{\partial o}{\partial net} \frac{\partial net}{\partial w_i}.$$

Der Fehler E ist von der Ausgabe o abhängig, die Ausgabe ist abhängig von der Netzeingabe net des Neurons und dieser ist wiederum von den Gewichten w_i abhängig.

Die erste partielle Ableitung $\frac{\partial E}{\partial o}$ ergibt:

$$\begin{aligned}\frac{\partial E}{\partial o} &= \frac{\partial}{\partial o}\left[\frac{1}{2}(t-o)^2\right] \\ &= \left[\frac{1}{2}2(t-o)\frac{\partial}{\partial o}(t-o)\right] \\ &= \left[\frac{1}{2}2(t-o)(-1)\right] \\ &= -(t-o)\end{aligned} \tag{1}$$

Die Ausgabe o hängt von der Aktivierungsfunktion $f_{act}(net)$ ab, daher ergibt die zweite partielle Ableitung $\frac{\partial o}{\partial net}$:

$$\frac{\partial o}{\partial net} = \frac{\partial}{\partial net} f_{act}(net) = f'_{act}(net). \tag{2}$$

Die Gleichung 1 und 2 ergeben zusammengesetzt:

$$\frac{\partial E}{\partial net} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial net} = -\left(t-o\right) f'_{act}(net) \tag{3}$$

Dies wird im Folgenden als Fehlersignal $\delta = -\frac{\partial E}{\partial net}$ abgekürzt. Die Netzeingabe net ergibt sich wiederum aus den Gewichten w_i, was partiell nach nur einer Komponente abgeleitet Folgendes ergibt:

$$\begin{aligned}\frac{\partial net}{\partial w_i} &= \frac{\partial}{\partial w_i}\sum_{m=0}^{n}(x_m w_m) \\ &= x_i\end{aligned} \tag{4}$$

Damit ergeben die Gleichungen 3 und 4 zusammengesetzt:

$$\begin{aligned}\Delta w_i &= -\eta\frac{\partial E}{\partial o}\frac{\partial o}{\partial net}\frac{\partial net}{\partial w_i} \\ &= \eta\left(t-o\right) f'_{act}(net)x_i \\ &= \eta\delta x_i\end{aligned}$$

Daraus folgt für die Delta-Regel:

$$w_i = w_i + \Delta w_i \text{ mit } \Delta w_i = \eta\left(t-o\right) f'_{act}(net)x_i.$$

Bei Verwendung einer linearen Aktivierungsfunktion mit $f'_{act}(net) = 1$ ergibt sich die vorgestellte Perzeptron-Regel:

$$\Delta w_i = \eta \delta x_i = \eta \, (t - o) \, x_i.$$

Die mit der Delta-Regel lösbaren Probleme sind jedoch eingeschränkt, so lassen sich z. B. die Ausgaben einer logischen XOR-Verknüpfung (siehe Abbildung 9) mit nur einer Schicht nicht klassifizieren. Durch Verbindung mehrerer Perzeptronen lassen sich das XOR-Problem und andere komplexe Probleme lösen, jedoch war lange kein Algorithmus zum Training mehrschichtiger Netze bekannt, was das Gebiet der künstlichen neuronalen Netze zur Lösung von Problemen uninteressant machte.

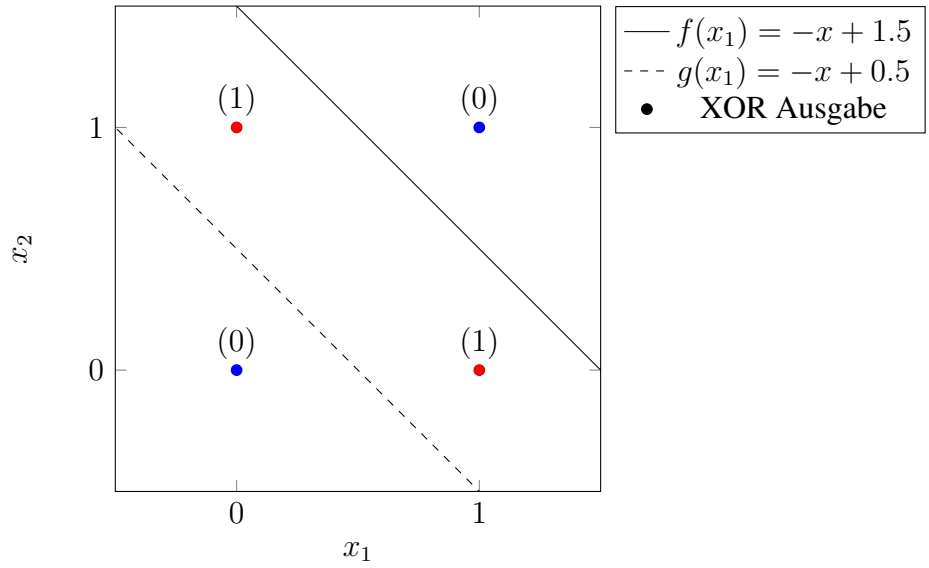


Abbildung 9: Ausgabe einer logischen XOR-Verknüpfung. Das Problem ist aufgrund der nicht linear separierbaren Daten nicht mit nur einem Perzeptron lösbar, da mindestens zwei Gleichungen benötigt werden.

3.3 Back-Propagation Algorithmus

1986 wurde von Rumelhart, Hinton und Williams der Back-Propagation Algorithmus vorgestellt, mit dessen Hilfe sich auch mehrschichtige Netze trainieren lassen, was das Gebiet der neuronalen Netze für die Forschung wieder interessanter machte.
Die Berechnung der Gewichte für die Ausgabeschicht deckt sich mit der Delta-Regel. Da es sich um ein mehrschichtiges Netz handelt, müssen die einzelnen Neuronen auf jeder Schicht berücksichtigt werden, womit sich für die Delta-Regel ergibt:

$$w_{i,j} = w_{i,j} + \Delta w_{i,j} \;\text{ mit }\; \Delta w_{i,j} = \eta \, (t_j - o_j) \, f'_{act}(net_j) x_i.$$

Wobei die Variable $w_{i,j}$ das Gewicht zwischen dem Neuron j und den eingehenden Neuronen i bezeichnet. Äquivalent dazu bezeichnet eine Variable mit dem Index j die Zugehörigkeit zum jeweiligen Neuron j. Das gleiche gilt für den Index i für eingehende Neuronen und Index k für nachfolgende Neuronen. Zu beachten ist noch, dass die Eingabe x_i, je nach Schicht, der Trainingseingabe oder der Ausgabe des eingehenden Neurons o_i entspricht.
Für die inneren Schichten lässt sich der Fehler E nicht direkt über die Ausgabe o_j berechnen, da im Gegensatz zur Ausgabeschicht der Soll Wert t_j unbekannt ist. Die Idee beim Back-Propagation ist die Berechnung des Fehlers für ein verdecktes Neuron über die gewichteten Fehlersignale der nachfolgenden Neuronen zu berechnen. Dieses Zurückpropagieren der Fehler der Nachfolger gibt dem Verfahren auch seinen Namen.
Der Fehler für ein Neuron, das nicht in der Ausgabe Schicht liegt, lässt sich über die gewichtete Summe aller nachfolgenden Neuronen k berechnen.
Basierend auf dieser Idee lässt sich $\frac{\partial E}{\partial o_j}$ durch Zuhilfenahme von Bekannten Ausdrücken formal beschreiben mit:

$$\frac{\partial E}{\partial o_j} = \sum_{k=0}^{K} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial o_j} \text{ mit } K = \text{Anzahl der nachfolgenden Neuronen.}$$

$\frac{\partial E}{\partial net_k}$ ist aus Gleichung 1 bekannt und wurde als Fehlersignal $\delta_k = -\frac{\partial E}{\partial net_k}$ definiert.
Da die nachfolgenden Neuronen k, ihre Eingabe nicht mehr aus dem Datensatz D erhalten, sondern aus der Ausgabe der Neuronen j ergibt sich für net_k:

$$net_k = \sum_{j=0}^{n} (o_j w_{j,k}) \ .$$

Die partielle Ableitung $\frac{\partial net_k}{\partial o_j}$ nach nur einer Komponente der Summe ergibt:

$$\begin{aligned} \frac{\partial net_k}{\partial o_j} &= \frac{\partial}{\partial o_j} \sum_{m=0}^{n} (o_m w_{m,k}) \\ &= w_{j,k} \ . \end{aligned} \tag{5}$$

Für $\frac{\partial E}{\partial o_j}$ ergibt sich aus δ_k und der Gleichung 5

$$\frac{\partial E}{\partial o_j} = \sum_{k=0}^{K} -\delta_k w_{j,k} \ . \tag{6}$$

Die Gewichtsänderung $\Delta(w_{i,j})$ für ein verdecktes Neuron berechnet sich damit aus Gleichung 2, 3 und 6. Wobei die Eingabe x_i, je nach Schicht, der Trainingseingabe oder der Ausgabe des eingehenden Neurons o_i entspricht.

$$\begin{aligned}\Delta(w_{i,j}) &= -\eta \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}} \\ &= \eta f'_{act}(net_j) x_i \sum_{k=0}^{K} \delta_k w_{j,k}\end{aligned}$$

Zusammengesetzt ergibt dies für den Back-Propagation Algorithmus:

$$w_{i,j} = w_{i,j} + \Delta w_{i,j} \quad \text{mit}$$

$$\Delta w_{i,j} = \begin{cases} \eta f'_{act}(net_j) x_i \, (t_j - o_j) & \text{wenn } j \text{ Neuron der Ausgabe Schicht} \\ \eta f'_{act}(net_j) x_i \sum\limits_{k=0}^{K} \delta_k w_{j,k} & \text{sonst} \end{cases}$$

Eine Pseudocode Implementierung kann dem Algorithmus 1 entnommen werden. Die Delta- und Back-Propagation-Regel wurde anhand des sogenannten Inkrementellen-Trainings hergeleitet. Dabei wird nach jedem Trainingsdatensatz die Gewichtung $w_{i,j}$ angepasst. Eine andere häufige Trainingsart ist das Batch-Training. Hierbei werden erst alle Trainingsdaten durchiteriert und die Gewichtsänderungen $\Delta w_{i,j}$ für jedes einzelne Gewicht aufsummiert. Anschließend werden diese auf die Gewichte $w_{i,j}$ angewendet.
Eine weitere Trainingsart, welche als Verallgemeinerung der anderen zwei Trainingsarten betrachtet werden kann, ist das Mini-Batch-Lernen. Hierbei wird ein Stapel von Trainingsdaten der Größe n iteriert, bevor die Neuronen-Gewichte angepasst werden, was bei einer Stapelgröße von $n = D$ dem Batch-Training und $n = 1$ dem Inkrementellen-Training entspricht.

Algorithmus 1 Pseudocode einer üblichen Back-Propagation Implementierung mit Inkrementellem-Training

Inputs

N: Netz mit n Schichten, N_i entspricht der Anzahl Neuronen auf Schicht i. Die Eingabeschicht entspricht Index 1 und das letzte Neuron auf jeder Schicht entspricht dem Bias Neuron.

w: Gewichte, $w_{l,i,j}$ entspricht i-tes Gewicht des j-ten Neuron auf Schicht l

o: Ausgaben, $o_{l,i}$ entspricht Ausgabe des i-ten Neuron auf Layer l (letzte Ausgabe ist konstant 1 für den Schwellenwert)

net: Netzeingaben, $net_{l,i}$ entspricht Netzeingabe des i-ten Neuron auf Layer l

t: angestrebte Lösungswerte

Steps

procedure BACKPROPAGATIONSTEP(N, w, o, net, t)
  for $j \leftarrow 1$ **to** $N_n - 1$ **do** ▷ berechne Fehlersignale für Ausgabeschicht
    $\delta_{n,j} \leftarrow f'_{act}(net_{n,j}) \cdot (t_j - o_{n,j})$
  end for
  for $l \leftarrow n - 1$ to 2 **do** ▷ berechne Fehlersignale für verdeckte Schichten
    $sum\delta w \leftarrow 0$
    for $j \leftarrow 1$ **to** N_l **do** ▷ j bis N_l um Schwellenwert einzubeziehen
      for $k \leftarrow 1$ **to** $N_{l+1} - 1$ **do**
        $sum\delta w \leftarrow sum\delta w + \delta_{l+1,k} \cdot w_{l+1,j,k}$
      end for
      $\delta_{l,j} \leftarrow f'_{act}(net_{l,j}) \cdot sum\delta w$
    end for
  end for
  for $l \leftarrow 2$ **to** n **do** ▷ alle nicht Eingabeschicht Gewichte modifizieren
    for $j \leftarrow 1$ **to** $N_l - 1$ **do**
      for $i \leftarrow 1$ **to** N_{l-1} **do** ▷ i bis N_{l-1} um Schwellenwert einzubeziehen
        $w_{l,i,j} \leftarrow w_{l,i,j} + \eta \cdot o_{l-1,i} \cdot \delta_{l,j}$
      end for
    end for
  end for
end procedure

3.3.1 Grenzen des Back-Propagation

Gradientenorientierte Verfahren haben den Vorteil, dass sie immer konvergieren, jedoch hat das Back-Propagation Verfahren aufgrund der Fehleroberfläche, welche von der Anzahl der Verbindung zwischen den Neuronen abhängt, auch einige Probleme:

- Aufgrund der zufälligen Wahl der initialen Gewichte kann nicht eindeutig festgelegt werden, ob das Verfahren in einem lokalen Minimum oder globalem Minimum endet.
- Bei flachen Plateaus wird der Gradient sehr klein und die Überwindung kann eine hohe Anzahl von Schritten benötigen.
- Bei Tälern können Oszillationen auftreten, im unglücklichsten Fall springt das Verfahren zwischen mehreren Punkten hin und her.

Es gibt diverse Ansätze und Heuristiken um einige Probleme zu umgehen und das Back-Propagation Verfahren zu verbessern. Auf einige wird in den nachfolgenden Abschnitten eingegangen.

3.4 Back-Propagation Heuristiken

In der Literatur findet man eine Vielzahl von Heuristiken, einige sollen hier dargestellt werden. Eine der meistzitierten Arbeiten zu diesem Thema ist [LeCun98] von Yann LeCun. In der Arbeit von LeCun geht es um die Verbesserung der Performance, was nicht direkt mit der Verbesserung der Lösung gleichzustellen ist, trotzdem sollen einige seiner Heuristiken hier betrachtet werden.

3.4.1 Inkrementelles- oder Batch-Training

Eine grundlegende Frage, die auch je nach Autor anders beantwortet wird, ist, ob Inkrementelles- oder Batch-Training verwendet werden sollte. Auf Grundlage der Arbeit [Wilson03] sollte man Inkrementelles-Lernen in Betracht ziehen. Wilson begründet dies damit, dass das Batch-Training zwar dem korrekten Gradienten folgt, jedoch diesem nur eine kurze Strecke gefolgt werden kann, da unklar ist ab welchem Punkt sich die Richtung ändert. Dieses Problem wächst bei ansteigender Anzahl von Trainingsdaten und Redundanz der Daten. Da sich die Vektorlänge erhöht, muss die Schrittweite verringert werden, um der Fehlerfunktion zu folgen. Bei Verwendung einer zu hohen Schrittweite kann ein einzelner Schritt sonst zu weit von der Fehlerfunktion abweichen und damit nicht zum Minimum konvergieren (siehe Abbildung 10a).

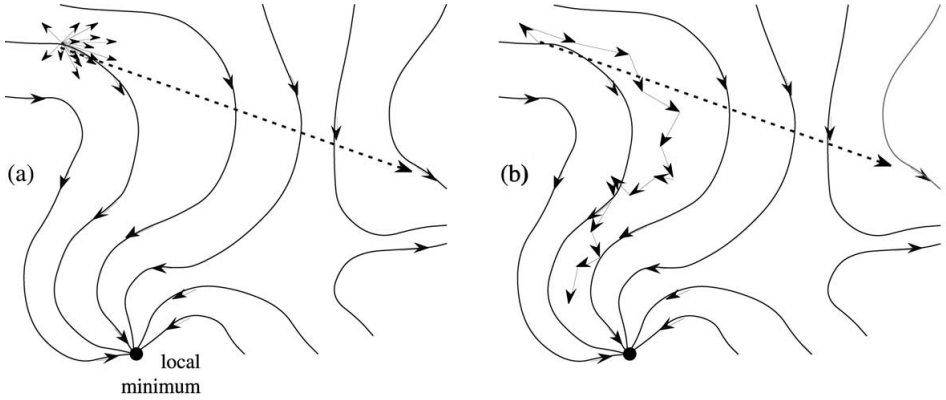


Abbildung 10: Grafische Darstellung des Batch-Trainings (a) und Inkrementellen-Trainings (b) (aus [Wilson03]/modifiziert)

Wohingegen beim Inkrementellen-Training man bei steigender Anzahl von Trainingsdaten, die Schrittweite nicht anpassen muss, da man sich nach jedem Schritt einen neuen 'lokalen' Gradienten berechnet und somit insgesamt weniger Iterationen durch die Trainingsdaten benötigt. Wie Abbildung 10b schematisch darstellt kann bei gleicher Schrittweite das Inkremententelle-Training mit weniger Operationen in die Nähe des Minimums kommen, während beim Batch-Training eine geringere Schrittweite und somit mehr Iterationen durch die Daten benötigt werden würden. Zudem können durch die 'Unruhe' der 'lokalen' Gradienten lokale Minima verlassen werden und zu besseren Minimums führen.
In einigen Fällen kann ein Mini-Batch-Training mit einer Stapelgröße zwischen 1 und D zu guten Lösungen führen, da diese zum einen mit einer höheren Schrittweite als Batch-Training trainieren kann und zum anderen eine bessere Annäherung an den korrekten Gradienten als Inkrementelles-Training berechnet.

3.4.2 Anpassung der Eingabedaten

Eingabedaten sollten bei Verwendung der Tangens Hyperbolicus Funktion nach LeCun im Durchschnitt 0 ergeben, die Idee dabei ist, dass wenn alle Eingabedaten positiv sind, sich die Gewichte bei einem Lernschritt je nach Fehlersignal δ immer nur in eine Richtung gemeinsam bewegen können. Dies führt bei Richtungswechsel eines Vektors zu Zickzack Bewegungen und das Verfahren verlangsamt. Für die Logistische-Funktion lässt sich diese Überlegung übertragen weshalb die Daten in diesem Fall im Durchschnitt 0.5 ergeben sollten.
Außerdem sollten die Daten skaliert werden, aus dem einfachen Grund, dass bei zu großen Eingaben die Aktivierungsfunktion aufgrund der beschränken Gleitkommazahl

Genauigkeit schnell den Maximalen Ausgabewert für die Aktivierungsfunktion ausgibt was bei Ableitung zur 0 führt und somit keine Änderungen der Gewichte vorgenommen werden. Natürlich kann die Genauigkeit erhöht werden was jedoch einen höheren Aufwand bedeutet und damit eine höhere Ausführungszeit.

3.4.3 Anpassung der Ausgabedaten

Die Ausgabedaten sollten auf den Ausgabebereich der Aktivierungsfunktion abgebildet werden, aus dem einfachen Grund, dass der Fehler sonst nicht 0 annehmen kann. Zusätzlich sollte die Ausgabe, aufgrund der Asymptote z. B. bei der $tanh(x)$ Funktion, nicht auf -1 und 1 gelegt werden, da diese sonst nie erreicht werden und es allein um in ihre Nähe zu kommen sehr hohe Gewichte braucht. Aus diesem Grund sollte die $tanh(x)$ Funktion gestreckt werden um die Ausgabe auf das Intervall $[-1, 1]$ skalieren zu können. In [LeCun98] wird als Sigmoidfunktion die gestreckte Tangensfunktion $f(x) = 1.7159 \cdot tanh(\frac{2}{3} \cdot x)$ vorgeschlagen.

3.4.4 Anzahl Layer und Neuronen

Wie in Abschnitt 3.1 beschrieben kann ein einzelnes Perzeptron, welches einer einzigen Schicht entspricht, den Raum durch eine Hyperebene teilen, was bei zwei Dimensionen wie im Beispiel der logischen UND-Verknüpfung einer Geraden entspricht.
Mit einer verdeckten Schicht kann ein konvexes Polygon klassifiziert werden. Dies wird durch eine Verknüpfung der Hyperebenen erreicht. Ein Beispiel für den zweidimensionalen Raum wäre: Klassifiziere Daten als 1, wenn sie unter Gerade 1 und über Gerade 2 liegen, was der Lösung des XOR-Problems entspricht.
Ein Netzwerk mit zwei verdeckten Schichten kann durch die Verknüpfung konvexer Polygone, jede beliebige Menge klassifizieren. Die Einführung einer weiteren Schicht bringt damit keinen weiteren Vorteil, kann sich jedoch positiv auf das Lernverhalten auswirken. Für gewöhnlich genügt für praktische Probleme damit ein Netz mit einer einzigen verdeckten Schicht.
Eine andere Frage, ist die Anzahl der Neuronen pro Schicht. Für die Eingabe und Ausgabe Schicht wird diese Frage durch die Codierung des Problems festgelegt. Für die Wahl der Anzahl der Neuronen auf einer verdeckten Schicht gibt es keine allgemeingültige Regel und wird deshalb normalerweise über Versuch und Irrtum gesucht.
Bei der Wahl von zu wenigen Neuronen können die Daten nicht separiert werden, wodurch sich die Erkennungsrate ab einem bestimmten Punkt nicht weiter verbessern lässt. Dies wird auch als *underfitting* bezeichnet (siehe Abbildung 11 c).
Bei der Wahl von zu vielen Neuronen können die Trainingsdaten exakt separiert werden, was die Erkennungsrate für die Trainingsdaten, je nachdem wie viele Neuronen zu

viel gewählt wurden, auf 100% bringt. Jedoch verschlechtert sich damit die Generalisierung des Netzes und somit die Erkennungsrate für neue unbekannte Datensätze. Dies wird auch als *overfitting* bezeichnet (siehe Abbildung 11 a).

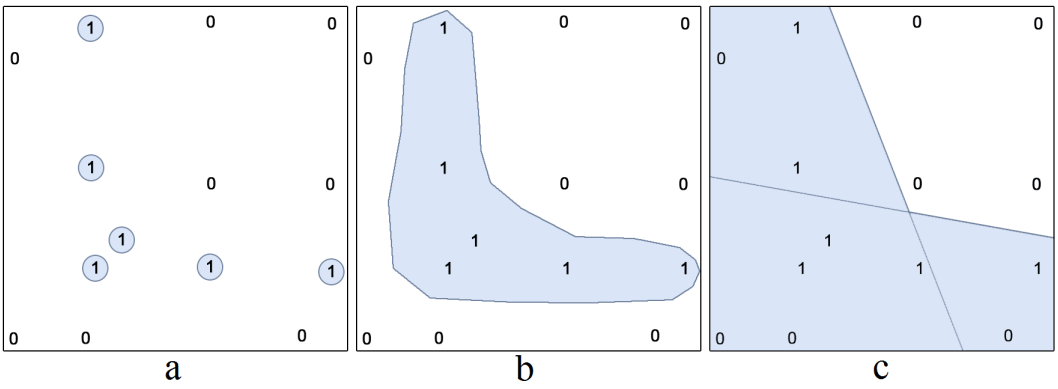


Abbildung 11: Grafische Darstellung von zu vielen Neuronen (a) optimale Anzahl (b) und zu wenigen Neuronen (c) (aus [Wilson03]/modifiziert).

Zur Wahl der Anzahl stellt [Heaton08] einige Heuristiken auf, die jedoch nur einen groben Richtwert für die Anzahl ergeben.

- Die Anzahl sollte zwischen der Eingabeneuronenanzahl und Ausgabeneuronenanzahl liegen.
- Die Anzahl sollte $\frac{2}{3}$ der Summe aus Eingabeneuronenanzahl und Ausgabeneuronenanzahl sein.
- Die Anzahl sollte kleiner als 2-mal die Anzahl der Eingabeneuronen betragen.

3.4.5 Initialisierung der Gewichte

Gewichte sollten nicht mit den gleichen Werten initialisiert werden, da dies zur Folge hat, dass sich alle Neuronen gleich verändern und es somit zu keiner Lösung kommt.
Stattdessen sollte eine zufällige Initialisierung erfolgen, jedoch haben zu große Werte den Nachteil, dass diese, wie bei zu großen Eingabedaten, zu sehr kleinen Gradienten führen, was das Verfahren verlangsamt und durch die beschränkte Fließkomma-Genauigkeit zu Fehlern führt.
Die übliche Initialisierung ist eine zufällige Gleichverteilung zwischen -1 und 1. LeCun empfiehlt Werte mit einem mittleren Wert von 0 und einer Standardabweichung σ_w mit $\sigma_w = m^{-1/2}$, wobei m die Anzahl der eingehenden Gewichte des jeweiligen Neurons entspricht.

3.4.6 Lernrate

Eine Unbekannte, auf die bis jetzt nicht weiter eingegangen wurde, ist die Lernrate beziehungsweise Schrittweite η. Eine schlechte Wahl der Lernrate kann zu verschiedenen Problemen führen. Bei einer zu großen Wahl könnte das gesuchte Minimum übersprungen werden und bei einer zu kleinen Wahl könnte das Verfahren zu lange dauern. Da die Wahl der optimalen Lernrate von vielen Faktoren abhängt, bleibt meistens nur die Bestimmung durch Experimente. In den meisten Fällen wird eine Lernrate zwischen 0.01 und 0.5 verwendet, wobei beim Inkrementellem-Training eher höhere und beim Batch-Training eher niedrigere Werte verwendet werden.
Zur Anpassung der Lernrate gibt es auch eine Vielzahl von Methoden, welche in einigen Fällen vom Gradienten abhängen. Jedoch hängen diese stark vom Problem ab und können sowohl zu einer Verbesserung als auch zu einer Verschlechterung des Verfahrens führen.
Ein Kernproblem bei den gradientenabhängigen Methoden ist, dass in einigen Regionen des Lösungsraums ein kleiner Gradient berechnet wird, aber große Schritte benötigt werden z. B. am Anfang durch die kleinen Initialisierungswerte. In anderen Regionen ist der Gradient ebenfalls klein, jedoch wird an diesen Stellen auch ein kleiner Gradient benötigt z. B. in der Nähe eines Minimums. Aus diesem Grund sollte man gradientenabhängige Anpassungsverfahren vermeiden.
Eine allgemeine gradientenunabhängige Methode ist der progressive Verfall, die Idee dabei ist mit einer großen Lernrate anzufangen, wodurch der Suchraum anfangs mit größeren Schritten durchquert wird und diese im Laufe der Iterationen zu verringern, um das Ergebnis näher an das Minimum zu führen. Eine einfache Berechnungsvorschrift dafür ist z. B.:

$$\eta(i) = \frac{\eta(0)}{1 + i \cdot r}$$

wobei i dem Iterationsschritt entspricht und r der Zerfallsrate zur Verringerung der Lernrate. Jedoch kann bei der Wahl einer unpassenden Zerfallsrate das Ergebnis auch verschlechtert werden.
In den betrachteten Bibliotheken wurde kein Verfahren zur Anpassung der Lernrate verwendet, was darauf schließen lässt, dass die Anpassung in praktischen Problemen zu keinen guten Ergebnissen führt, zudem muss bedacht werden, dass eine weitere Variable r hinzukommt.

3.4.7 Abwandlungen des Standard-Back-Propagation

Es gibt eine große Anzahl unterschiedlicher Abwandlungen des Back-Propagation Algorithmus. Hier soll nur der Back-Propagation mit Momentum vorgestellt werden, welches eines der Bekanntesten Abwandlungen darstellt. Der Hintergrund des Verfahrens

ist, dass zwei hintereinander folgende Gradienten in flachen Plateaus das gleiche Vorzeichen besitzen und sich die Vorzeichen in Tälern oft ändern, wodurch es zu Oszillationen kommt. Um diesem Verhalten entgegen zu wirken, ist die Idee des Momentum-Verfahrens, den vorherigen Gradienten miteinzubeziehen. Dies hat zur Folge, dass zum einen bei flachen Plateaus größere Schritte gemacht werden und zum anderen sich die Oszillationen in Tälern verringern.
Aus dieser Überlegung ergibt sich die Formel:

$$w_{i,j}(t+1) = w_{i,j}(t) + \Delta w_{i,j}(t) + \alpha \Delta w_{i,j}(t-1).$$

Wobei α den vorherigen Gradienten gewichtet. Für die Wahl von α gibt es ähnlich wie bei der Lernrate η keine allgemeine Regel zur Bestimmung eines optimalen Wertes. Für gewöhnlich wird $0 < \alpha < 1$ gewählt.

3.5 Genetischer Algorithmus

Ein anderes Verfahren zum Trainieren von neuronalen Netzen sind genetische Algorithmen. Diese gehen auf die Arbeit von John Holland zurück und sind durch Charles Darwins Evolutionstheorie inspiriert.
Genetische Algorithmen gehören zu den evolutionären Algorithmen, welche der Klasse der heuristischen Suchverfahren zugeordnet werden, die insbesondere bei der Suche einer Lösung innerhalb eines Suchraums angewendet werden. Damit eignen sich genetische Algorithmen für eine Vielzahl von Problemen. Obwohl es auch genetische Ansätze zur Erstellung von Netzen und zur Optimierung von anderen Parametern für den Back-Propagation Algorithmus gibt, soll hier das Training eines künstlichen neuronalen Netzwerks selbst betrachtet werden.

3.5.1 Biologische Begriffe

Stark vereinfacht gesehen, bestehen Organismen aus Zellen, welche aus dem gleichen Satz von einem oder mehreren Chromosomen bestehen und als Bauplan für den Organismus dienen. Chromosome kann man wiederum in Gene unterteilen, wobei ein Gen als Merkmal gesehen werden kann, z. B. als Merkmal Augenfarbe. Die möglichen Ausprägungen der Merkmale werden als Allele bezeichnet. In diesem Fall: (Braun, Grün, Blau, Grau, ...).
In genetischen Algorithmen wird ein Lösungskandidat als Chromosom bezeichnet und besteht aus einer Liste von Genen, ein einzelnes Gen wird in der Regel durch einen einzelnen Datentyp repräsentiert.

3.5.2 Verfahren

Bei den genetischen Algorithmen wird zu Beginn eine Gruppe von Chromosomen mit zufälligen Genen erstellt, welche die Anfangspopulation bildet. Für jedes Chromosom wird mit einer Problem spezifischen Funktion die Güte berechnet. Anschließend werden Chromosome zufällig selektiert, wobei die Wahrscheinlichkeit für jedes Chromosom von seiner Güte abhängt. Die selektierten Chromosome werden zur Bildung einer neuen Population herangezogen, dies geschieht mit Hilfe von Rekombination und Mutation
Die neue Population durchgeht nun die gleiche Prozedur wie die Anfangspopulation. Das Verfahren wiederholt sich, bis ein gegebenes Abbruchkriterium erfüllt wird, z. B. eine bestimmte Güte erreicht wurde. Einen Pseudocode kann man Algorithmus 2 entnehmen.
Bei der Verwendung von genetischen Algorithmen muss insbesondere das Problem als Chromosom encodiert werden und eine Funktion zur Bewertung der Güte dieser Chromosome gefunden werden.

Algorithmus 2 Pseudocode eines einfachen genetischen Algorithmus

Inputs

n: Größe der Population
c_p: Kreuzungswahrscheinlichkeit
m_p: Mutataionswahrscheinlichkeit

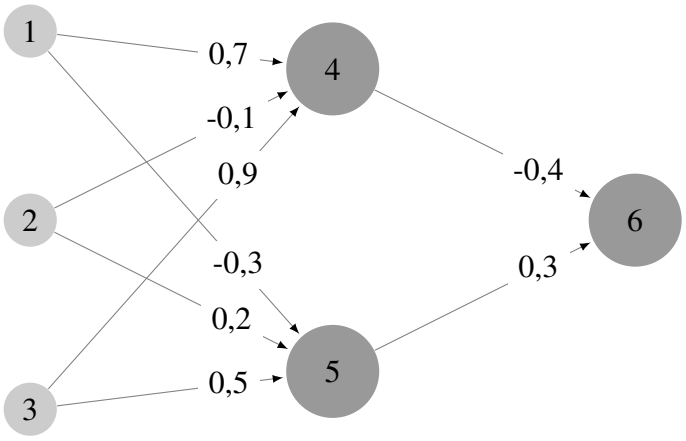
Steps

procedure GENETICALGORITHM(n, c_p, m_p)
    $P \leftarrow CreateRandomPopulation(n)$
    $N \leftarrow CreateEmptyPopulation(n)$
    $F \leftarrow CalculateFitness(P)$ ▷ Berechne Fitness für jedes Chromosom
    repeat
        for $j \leftarrow 1$ **to** n **do**
            ▷ Wähle, mit fitnessabhängiger Wahrscheinlichkeit, zufällig zwei aus
            $x_1 \leftarrow SelectRandomChromosom(F)$
            $x_2 \leftarrow SelectRandomChromosom(F)$
            $N_j \leftarrow CrossoverChromosoms(P_{x_1}, P_{x_2}, c_p)$
            $N_j \leftarrow MutateChromosom(N_j, m_p)$
        end for
        $P \leftarrow N$ ▷ Übernehme neue Population
        $F \leftarrow CalculateFitness(P)$ ▷ Berechne Fitness für jedes Chromosom neu
    until $SomeStopCriteria(F)$
    return P
end procedure

3.5.3 Neuronales Netz als Chromosom

Zur Encodierung des neuronalen Netzes als Chromosom wird, wie in der Arbeit [Montana89] beschrieben jedes Gewicht als Gen betrachtet. Die Auflistung aller Gewichte ergibt damit das Chromosom (siehe Abbildung 12). Im Kontext der genetischen Algorithmen wird eine konkrete Lösung, in diesem Fall ein neuronales Netz, also Phänotyp und die encodierte Lösung als Genotyp bezeichnet.

Netzwerk:



Chromosom: (-0,4 0,3 0,7 -0,1 0,9 -0,3 0,2 0,5)

Abbildung 12: Encodierung eines neuronalen Netzes als Chromosom

3.5.4 Operationen

Das oben beschriebene Verfahren besteht im Wesentlichen aus vier Operationen, einer Funktion zur Bewertung der Güte, welche als Fitness Funktion bezeichnet wird, ein Selektion Verfahren, ein Verfahren zur Rekombination und ein Verfahren zur Mutation.

Fitness-Funktion

Die Fitness-Funktion bestimmt die Güte eines Chromosoms. Bei neuronalen Netzen wäre das Propagieren von Testdaten, durch das Netz und der daraus berechnete mittlere quadratische Fehler MSE *(engl. mean square error)* eine naheliegende Funktion dafür, welcher den Mittelwert der vorgestellten Fehlerfunktion über alle Ausgabe-Neuronen und ohne den Faktor $\frac{1}{2}$ in Abschnitt 3.2 darstellt.

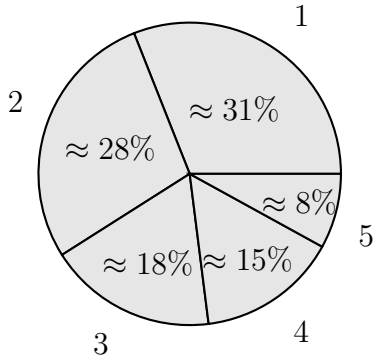
$$MSE = \frac{1}{D} \sum_{d=0}^{D} \sum_{j=0}^{N} (t_{d,j} - o_j)^2$$

Wobei D der Anzahl der Datensätze und N der Anzahl Neuronen auf der Ausgabe Schicht entspricht.

Selektion

Auf Basis der Ergebnisse der Fitness-Funktion werden Chromosome für die Erzeugung der nächsten Population ausgewählt. Die üblichen Selektionsverfahren werden im Folgenden aufgelistet.

Roulette Wheel Selection Bei der *Roulette Wheel Selection* wird jedem Chromosom, abhängig von seiner Fitness, ein Abschnitt eines Roulette Rades zugeteilt (siehe Abbildung 13) und anschließend eine zufällige Position ausgewählt und damit ein Chromosom zufällig bestimmt. Dabei gilt je höher die Fitness, desto wahrscheinlicher die Auswahl. Da der MSE bei Verbesserung sinkt, muss entweder der inverse MSE verwendet werden oder bei sortierter Reihenfolge der inverse Index.



Chromosm	Inverser MSE
1	35
2	32
3	20
4	17
5	9

Abbildung 13: Beispiel für ein Roulette Wheel mit 5 Elementen

Die Wahrscheinlichkeit für ein Chromosom lässt sich bei einer Populationsgröße n berechnen mit:

$$P(i) = \frac{MSE_i}{\sum_{j=1}^{n} MSE_j}$$

wobei i den Index des Chromosoms angibt. Der Pseudocode 3 beschreibt eine Simulation des *Roulette Wheel Selection* Verfahren.

Algorithmus 3 Pseudocode einer Roulette Auswahl

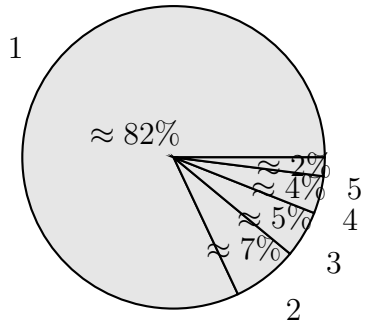
Inputs

F: Fitness mit n Elementen, F_j entspricht der Fitness des j-ten Chromosoms.

Steps

```
procedure TURNWHEEL(F)
    sum ← 0
    offset ← 0
    for j ← 1 to n do
        sum ← sum + F_j
    end for
    r ← GenerateRandomNumberBetween(0, sum)
    for pick ← 1 to n do
        offset ← offset + F_pick
        if r < offset then
            return pick                ▷ pick entspricht Index des Chromosoms
        end if
    end for
end procedure
```

Rank Selection Die fitnessbasierte Roulette Wheel Selection Methode hat Probleme, wenn ein Chromosom eine viel bessere Fitness als die restlichen Individuen in der Population aufweist. In diesem Fall haben die anderen Chromosome eine sehr niedrige Wahrscheinlichkeit gewählt zu werden (siehe Abbildung 14).



Chromosm	Inverser MSE
1	350
2	32
3	20
4	17
5	9

Abbildung 14: Beispiel für ein unausgeglichenes Roulette Wheel mit 5 Elementen

Bei Rank Selektion werden die Wahrscheinlichkeiten nicht direkt über die Fitness berechnet, sondern über das Ranking, welches sich aus der Fitness ergibt. Bei 5 Individuen ergibt sich für das schlechteste Chromosom eine Wahrscheinlichkeit von $\frac{1}{1+2+3+4+5}$ und für das Beste $\frac{5}{1+2+3+4+5}$ beziehungsweise für eine Populationsgröße von n, allgemein:

$$P(i) = \frac{n+1-i}{\sum_{j=1}^{n} j}.$$

Steady-State Selection Die Idee bei *Steady-State* ist es, anstatt eine neue Generation zu erstellen, bei der aktuellen die Schlechtesten zu ersetzen.

Elitism Bei *Elitism* werden die besten Lösungen in die nächste Generation übernommen um das Verlieren dieser zu vermeiden, anschließend werden die restlichen Lösungen über die klassischen Methoden ermittelt.

Rekombination

Bei der Rekombination werden zwei Chromosome gewählt und zu einem neuen Chromosom vereint. Es gibt eine Vielzahl unterschiedlicher Verfahren. Für das Training von neuronalen Netzen mit genetischen Algorithmen stellt David Montana in seiner Arbeit [Montana89] einige Rekombinationsoperatoren vor, von denen einige ausgewählte hier aufgelistet werden sollen:

- CROSSOVER-WEIGHTS: Das Kind Chromosom wird durch zufällige Auswahl der Eltern Chromosomen zusammengesetzt.

- CROSSOVER-NODES: Für jedes Neuron im Kind Chromosom wird das entsprechende Neuron von einem der zwei Eltern Chromosome ausgewählt und alle Gene für die eingehenden Gewichte übernommen (siehe Abbildung 15).

Eltern Chromosom 1:	(-0,4 0,3	0,7 -0,1 0,9	-0,3 0,2 0,5)
Eltern Chromosom 2:	(0,6 0,1	-0,7 1,0 -0,3	-0,2 0,1 0,5)
Kind Chromosom 1:	(0,6 0,1	0,7 -0,1 0,9	-0,2 0,1 0,5)
Kind Chromosom 2:	(-0,4 0,3	-0,7 1,0 -0,3	-0,3 0,2 0,5)

Abbildung 15: Beispiel für CROSSOVER-NODES ausgehend von einem schichtenweise voll vernetzten 3-2-1 feedforward Netz

Weitere übliche Rekombinationsoperatoren sind:

- Single-Point-Crossover: Bis zu einem Punkt p_1 werden alle Gene vom ersten Eltern Chromosom übernommen und die restlichen Gene vom zweiten Eltern Chromosom.
- Two-Point-Crossover: Zwei Punkte p_1 und p_2 werden ausgewählt wobei $p_1 < p_2$ und alle Gene vor dem ersten Punkt und nach dem zweiten Punkt werden vom ersten Eltern Chromosom übernommen, die restlichen werden vom zweiten Eltern Chromosom übernommen.

Die Rekombination kann als Durchqueren des Suchraums zwischen zwei Punkten verstanden werden, wobei die Punkte durch die gewählten Chromosome zur Rekombination repräsentiert werden.

Mutation

Bei der Mutation wird ein Chromosom gewählt und Gene zufällig verändert. Wie bei der Rekombination existiert auch hier eine große Anzahl von möglichen Operationen. Es folgt eine Auswahl von Mutationsoperationen aus der Arbeit [Montana89]:

- UNBIASED-MUTATE-WEIGHTS: Mit einer Wahrscheinlichkeit $p = 0.1$ soll jedes Gen in einem Chromosom mit einem Wert aus dem Anfangs-Initialisierungsbereich ersetzt werden.
- BIAS ED-MUTATE-WEIGHTS: Mit einer Wahrscheinlichkeit $p = 0.1$ soll zu jedem Gen in einem Chromosom ein Wert aus dem Anfangs-Initialisierungsbereich hinzuaddiert werden.

- MUTATE-NODES: Es werden n Neuronen ausgewählt und zu jedem Gewicht dieser Neuronen wird ein Wert aus dem Anfangs-Initialisierungsbereich hinzuaddiert. In der Arbeit wurde für n gleich 2 gesetzt. Die Abbildung 16 veranschaulicht einen Mutationsschritt.

Kind Chromosom: (0,6 0,1	-0,7 1,0 -0,3	-0,2 0,1 0,5)
Mutiertes Kind Chromosom: (0,6 0,1	0,7 -0,1 0,9	-0,2 0,1 0,5)

Abbildung 16: Beispiel für MUTATE-NODES mit $n = 1$, ausgehend von einem schichtenweise voll vernetzten 3-2-1 feedforward Netz

Die Mutation kann als Abweichung der Richtung bei der Durchquerung des Suchraums interpretiert werden.

3.5.5 Heuristiken

Aus den praktischen Erfahrungen von [Kitano90] und [Obitko98] sollen hier einige Heuristiken bezüglich der Parameter Wahl bei genetischen Algorithmen aufgelistet werden.

- Als Rekombinationswahrscheinlichkeit wird ein hoher Wert empfohlen zwischen $0,6$ und $0,95$.
- Die Mutationsrate hingegen sollte niedrig sein, empfohlen werden Werte zwischen $0,05$ und $0,1$.
- Für die Wahl der Populationsgröße gibt es unterschiedliche Empfehlung, diese Schwanken zwischen 20 bis 100 je nach Problem.
- Für die Wahl der Operationen- und Selektion-Verfahren kann ebenfalls keine allgemeine Aussage gegeben werden, jedoch wird die Verwendung von Elitism empfohlen.

3.5.6 Grenzen des genetischen Algorithmus

Da es sich bei genetischen Algorithmen um heuristische Verfahren handelt, bieten diese keine Garantie, dass sie das globale Optimum finden. Zudem gelten genetische Algorithmen als rechenintensiv, wobei dies von den gewählten Operationen, der Fitnessfunktion und der Populationsgröße abhängt.

4 Neuronale Netzwerk Bibliotheken

Es gibt eine Vielzahl verschiedener Programme und Bibliotheken im Bereich der neuronalen Netze, daher fällt die Entscheidung, welche man verwenden sollte nicht leicht. Ein Überblick bietet die Liste der University of Colorado Boulder[1], welche jedoch nur einen ersten Überblick bietet. Nach einiger Recherche scheinen die folgenden Bibliotheken besonders häufige Verwendung zu finden, wobei ausschließlich nach freier Software gesucht wurde.

Fast Artificial Neural Network Library[2] (*kurz FANN*) ist die mit Abstand am meisten erwähnte Bibliotheken bei der Recherche. Der Kern ist in C geschrieben, jedoch gibt es eine große Anzahl von Wrappern für andere Sprachen z.B C++, C#, Java, Python und R. Die Bibliothek wird oft mit anderen Bibliotheken verglichen, insbesondere wenn es um Performance geht. Neben dem Standard-Back-Propagation sind auch Back-Propagation mit Momentum, Quickpropagtion und Resillent-Propagation implementiert, welche eine Modifikation des Standard-Back-Propagation Algorithmus darstellen. Zum Testen der Funktionalität wurde ein der Bibliothek beiliegendes Beispiel in C++ modifiziert.

Encog[3] ist eine *Machine Learning Library* und implementiert eine große Anzahl von Netzwerk Architekturen und Trainingsmethoden, und ist vom Umfang der Funktionalität eine der mächtigsten freien Bibliotheken. Encog ist für Java, C# und C++ verfügbar, jedoch befindet sich die C++ Version noch in einer Beta Phase, wohingegen die Java- und C#-Bibliotheken bereits in der Version 3.2 erhältlich sind. Zum Testen der Funktionalität habe ich mich deshalb gegen die C++ Version entschieden und die C# Bibliothek verwendet.

Stuttgart Neural Network Simulator[4] (*kurz SNNS*) wird, im Gegensatz zu den anderen vorgestellten Bibliotheken, nicht mehr aktiv weiterentwickelt. Sie wird oft als Einsteig in neuronale Netze verwendet, insbesondere die Software JavaNNS, welche eine grafische Benutzeroberfläche bereitstellt, wobei der Kern in C geschrieben ist. Der Umfang der Funktionalität ist zwischen FANN und Encog anzusiedeln. Aufgrund einer Inkompatibilität zwischen dem verwendeten Betriebssystem und der JavaNNS Software, wurde *RSNNS* verwendet, welche eine Wrapper Version für R ist.

[1] `http://grey.colorado.edu/emergent/index.php/Comparison_of_Neural_Network_Simulators`

[2] `http://leenissen.dk/fann/wp/`

[3] `http://www.heatonresearch.com/encog`

[4] `http://www.ra.cs.uni-tuebingen.de/SNNS/`

4.1 Performance Vergleich

Ein wichtiges Kriterium ist neben der Funktionalität auch die Performance, da das Trainieren je nach Anzahl der Verbindungen und des angestrebtem Ergebnisses eine lange Zeit in Anspruch nehmen kann. Für einen kurzen Vergleich der Performance wurde das XOR-Problem ausgewählt, da dieses einfach zu realisieren ist und keine zusätzliche Konvertierung von Trainingsdaten, für die jeweilige Bibliothek, vorgenommen werden musste. Zum Testen wurde eine Konstellation verwendet, die alle Bibliotheken unterstützten:

- Back-Propagation Algorithmus ohne Momentum
- Batch-Learning
- Schichten verbundene Feedforward Architektur: 2 Eingabe Neuronen, 10 verdeckte Neuronen und 1 Ausgabe Neuron $(2 - 10 - 1)$
- Aufgrund der geringen Datensatzanzahl von 4 beim XOR-Problem wurden 100.000 Iterationen durchgeführt
- Zusätzliche Ausgaben, Loggen oder andere nicht mit dem Lernen verbundene Operationen wurden, so weit wie möglich, deaktiviert

Wie der Tabelle 1 zu entnehmen ist, hat die FANN Bibliothek für die gewählte Konstellation eine geringere Ausführungszeit als die anderen zwei Bibliotheken, jedoch gibt das Ergebnis nur einen ersten Eindruck über die Performance, da diese für andere Konstellationen abweichen kann. Andere ausführlichere Performance Vergleiche legen jedoch nahe, dass FANN eine der Performanteren freien Bibliotheken ist.
Eine wichtigere Rolle als die Performance, ist jedoch die Wahl des Verfahrens und der Parameter. Hier bietet Encog mehr Auswahl, zudem ist eine Multithreading Version verfügbar, welche jedoch nicht getestet wurde.

Bibliothek	*Ausführungszeit in Sekunden*
FANN	0,37
Encog	2,91
RSNNS	26,24

Tabelle 1: Bibiotheken Performance Vergleich

5 Eigene neuronale Netz Bibliothek

Zum Testen verschiedener Konstellationen und Heuristiken, sowohl bei genetischen Algorithmen, als auch beim Back-Propagation Algorithmus, sollte eine Bibliothek erstellt werden. Im Folgenden wird der Entwurf der Bibliothek, die Implementierung der Lernalgorithmen, eine Anwendung zur Verwendung dieser und ein Vergleich mit einer ausgewählten Bibliothek beschrieben.

5.1 Bibliothek Entwurf

Bei der Modellierung des neuronalen Netzes, welche die Grundlage zum Trainieren bildet, habe ich mich für eine objektorientierte Architektur entschieden. Die Hauptgründe für diese Entscheidung sind: die größere Flexibilität, die einfachere Erweiterbarkeit, die bessere Wiederverwendbarkeit und die für gewöhnlich bessere Lesbarkeit durch Kapselung, die eine objektorientierte Architektur mit sich bringt.

Die Architektur des neuronalen Netzes ist dem Klassendiagramm in Abbildung 17 zu entnehmen. Diese stellt die grundlegende Netzwerkarchitektur, wie in Abschnitt 2.4 beschrieben, dar. Da die Eingabeschicht keine Neuronen besitzt, sondern lediglich den Eingabevektor weiterreicht, wurde dieser als *InputValues* des jeweiligen *Layers* modelliert. Dieser beinhaltet in der ersten Schicht mit Neuronen den Eingabevektor, der in das Netz eingegeben wird und in den nachfolgenden Schichten die jeweiligen Ausgabewerte der vorherigen Schicht.
Dies ermöglicht eine einfache Erweiterbarkeit für andere Architekturen, indem für eine Schicht die Nachfolgeschicht und die zu übergebenen Neuronen definiert werden. Zusätzlich zum Gewichtsvektor (*Weights*) besitzt jedes Neuron einen Vektor zur Speicherung der Gewichtsänderung (*DeltaWeights*), welche für das Mini-Batch-Lernen mit einer Stapelgröße > 1 benötigt wird. Der Vektor (*LastDeltaWeights*) dient zur Speicherung der letzten Gewichtsänderung, welche für verschiedene Modifikationen des Back-Propagation Algorithmus benötigt werden, unter anderem für das vorgestellte Momentum-Verfahren.

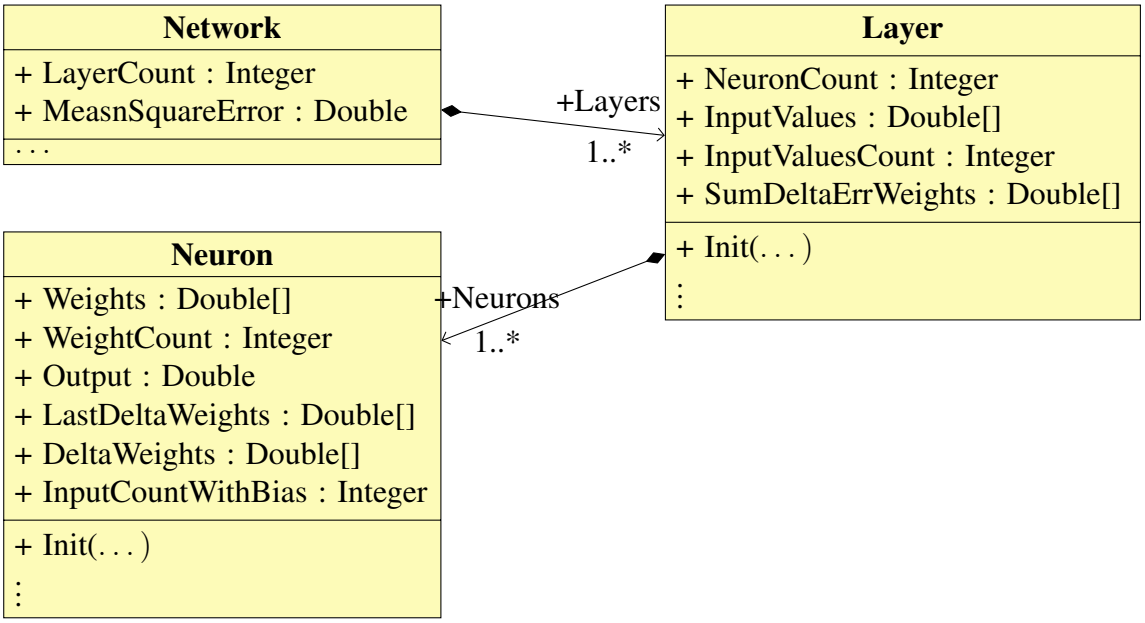


Abbildung 17: Klassendiagramm des neuronalen Netzes (aus Übersichtsgründen nicht vollständig, ein vollständiges Klassendiagramm ist der Programm Dokumentation zu entnehmen)

Die Trainingsalgorithmen wurden in eigene Klassen gekapselt (*Backpropagation*, *GeneticAlgorithm*) und erben von einer abstrakten Basis Klasse (*TrainerBase*), welche die grundlegenden Aufgaben von Trainingsalgorithmen implementiert. Unter anderem die Erfassung und die Speicherung von Messergebnissen (siehe Klassendiagramm 18).

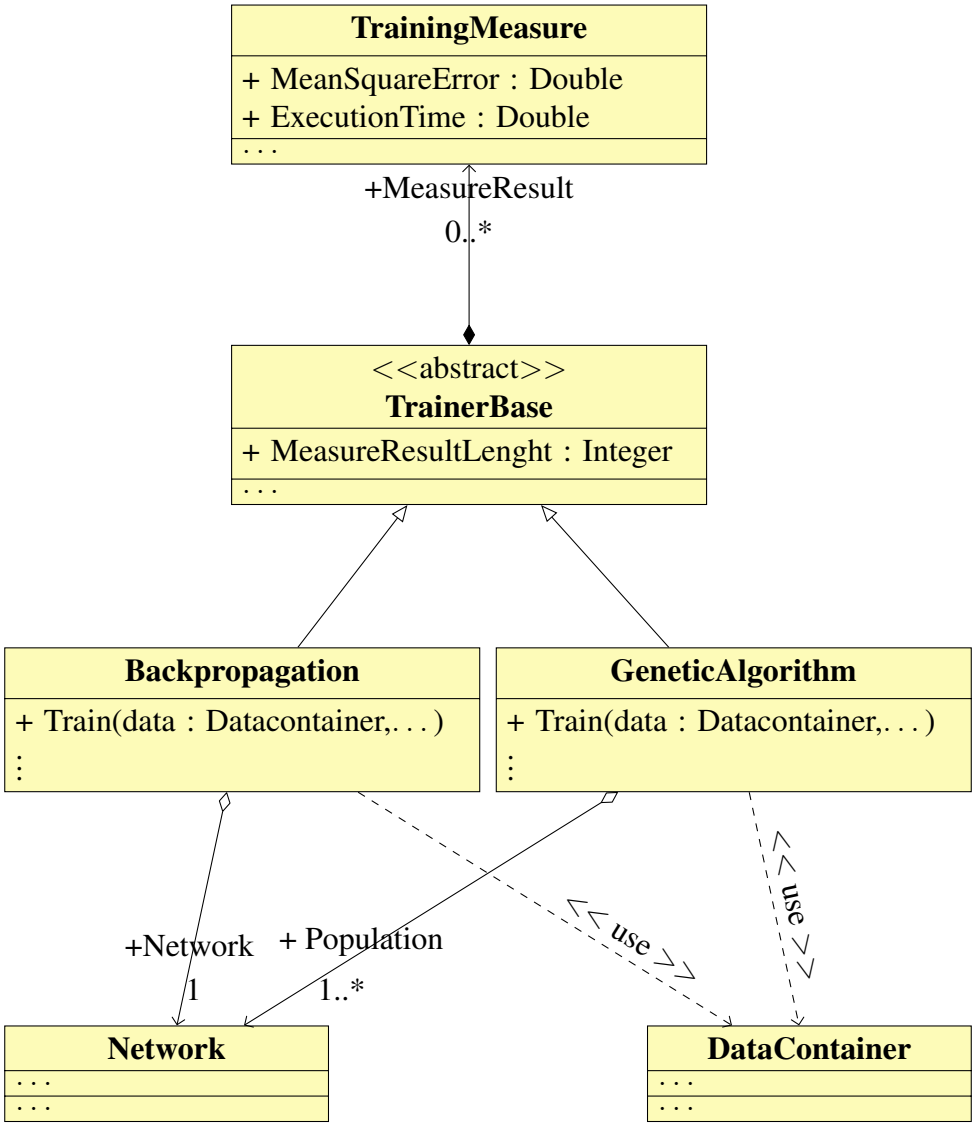


Abbildung 18: Klassendiagramm des neuronalen Netzes (aus Übersichtsgründen nicht vollständig)

5.2 Bibliothek Implementierung

In diesem Abschnitt soll kurz die Implementierungen der Trainingsklassen *Backpropagation* und *GeneticAlgorithm* näher betrachtet werden, welche neben der *Network* Implementierung den Kern der Bibliothek darstellen.

5.2.1 Back-Propagation Implementierung

Die übliche Implementierung, wie sie der Pseudocode in Algorithmus 1 darstellt, wurde von mir als umständlich angesehen. Deshalb fand eine Umstrukturierung, wie der Pseudocode im Algorithmus 4 zeigt, statt. Diese Umstrukturierung brachte eine Performance-Verbesserung sowohl für kleine, als auch große Netze, welche hier jedoch nicht im Detail betrachtet wird. Zum Testen der Implementierung wurden zwei Referenz Programme mit den gleichen Werten initialisiert. Anschließend erfolgte ein Vergleich der Ergebnisse. Die Ergebnisse weisen darauf hin, dass die Implementierung korrekt funktioniert.
Zum Vergleich der Performance wurde die eigene Implementierung unter anderem mit der freien Bibliothek FANN verglichen. Verglichen wurde der Standard Back-Propagation Algorithmus, alle anderen Optionen wurden deaktiviert. Zudem wurden beide Bibliotheken mit den gleichen Compiler Optionen kompiliert und die gleiche Aktivierungsfunktion verwendet. Dabei konnte, je nach Konstellation, eine Performance Verbesserung um bis zu Faktor 2 erzielt werden. Die Ergebnisse bezüglich der Performance sind Tabelle 2 entnehmbar.

Versuchs Beschreibung			Performance in Sek.	
Topologie	Iterationen Insgesamt	Trainingsart	FANN	Eigene Impl.
2-3-1	40.000.000	Inkrementell	15	7
4-6-1	134.600.000	Inkrementell	85	41
784-300-10	600.000	Inkrementell	549	360
784-300-10	600.000	Batch	371	240

Tabelle 2: Performance Vergleich Back-Propagation, die Gesamtanzahl der Iterationen berechnet sich aus: (Anzahl der Datensätze) $\cdot$ (Anzahl Iterationen).

Zusätzlich wurden alle beschriebenen Verbesserungsmöglichkeiten, wie im Abschnitt 3.4 beschrieben, implementiert.

Algorithmus 4 Pseudocode des implementierten Back-Propagation Algorithmus

Inputs

N: Netz mit n Schichten, N_i entspricht der Anzahl Neuronen auf Schicht i. Die Eingabeschicht entspricht Index 1 und das letzte Neuron auf jeder Schicht entspricht dem Bias Neuron.

w: Gewichte, $w_{l,i,j}$ entspricht i-tes Gewicht des j-ten Neuron auf Layer l. Das letzte Gewicht entspricht dem Schwellenwert-Gewicht.

o: Ausgaben, $o_{l,j}$ entspricht Ausgabe des j-ten Neuron auf Layer l

net: Netzeingaben,$net_{l,j}$ entspricht Netzeingabe des j-ten Neuron auf Layer l

t: angestrebte Lösungswerte

Steps

procedure BACKPROPAGATIONSTEP(N, w, o, net, t)
  for $l \leftarrow n$ **to** 3 **do** ▷ Ausgabeschicht bis zweite verdeckte Schicht
    for $j \leftarrow 1$ **to** $N_l - 1$ **do**
      if $l == n$ **then**
        $\delta \leftarrow f'_{act}(net_{n,j}) \cdot (t_j - o_{n,j})$ ▷ berechne Fehlersignal
      else
        $\delta \leftarrow f'_{act}(net_{l,j}) \cdot sum\delta w_{l,j}$
      end if
      $\delta\eta \leftarrow \delta \cdot \eta$
      for $i \leftarrow 1$ **to** $N_{l-1} - 1$ **do**
        $sum\delta w_{l-1,i} \leftarrow sum\delta w_{l-1,i} + w_{l,i,j} \cdot \delta$ ▷ Fehler für Neuron$_{l-1,i}$
        $w_{l,i,j} \leftarrow w_{l,i,j} + o_{l-1,i} \cdot \delta\eta$ ▷ Gewichte anpassen
      end for
      $w_{l,N_{l-1},j} \leftarrow w_{l,N_{l-1},j} + 1 \cdot \delta\eta$ ▷ Schwellenwert-Gewicht anpassen
    end for
  end for
  for $j \leftarrow 1$ **to** $N_2 - 1$ **do** ▷ Berechnungen für erste verdeckte Neuronen-Schicht
    $\delta\eta \leftarrow f'_{act}(net_{2,j}) \cdot sum\delta w_{2,j} \cdot \eta$
    for $i \leftarrow 1$ **to** $N_1 - 1$ **do**
      $w_{2,i,j} \leftarrow w_{2,i,j} + o_{1,i} \cdot \delta\eta$ ▷ $o_{1,i}$ entspricht Eingabe x_i
    end for
    $w_{2,N_1,j} \leftarrow w_{2,N_1,j} + 1 \cdot \delta\eta$
  end for
end procedure

5.2.2 Genetischer Algorithmus Implementierung

Da beim genetischen Algorithmus die Berechnung des MSE die meiste Rechenzeit in Anspruch nimmt, bringt eine Optimierung nur eine geringe Verbesserung der Performance. Deshalb wurde ein einfacher genetischer Algorithmus, ähnlich dem im Pseudocode 2 beschrieben, implementiert. Alle erwähnten Operationen wie in Abschnitt 3.5 beschrieben wurden implementiert.

5.2.3 C++ spezifische Optimierungen

Im Wesentlichen wurden nur drei Optimierungen vorgenommen die eine Verbesserung sowohl beim *Microsoft (R) C/C++ Optimizing Compiler Version 17*, als auch beim *GNU GCC Compiler Version 4.91* brachten, welche hier kurz aufgelistet werden sollen:

Zeiger-Ketten Vermeidung

Durch Verwendung von lokalen Variablen sollen Zeiger-Ketten vermieden werden. Ein einfaches Beispiel dafür wäre der Zugriff auf den Gewichtsvektor des ersten Neurons in der ersten Schicht:

```
network->Layers[0].Neurons[0].Weights[0] = 0;
network->Layers[0].Neurons[0].Weights[1] = 0;
network->Layers[0].Neurons[0].Weights[2] = 0;
```

Dieser Code muss `network->Layers[0].Neurons[0]` nach jeder Zuweisung neu laden. Eine einfache Möglichkeit dies zu umgehen, ist die Verwendung einer Cache Variable.

```
double * weights = network->Layers[0].Neurons[0].Weights;
weights[0] = 0;
weights[1] = 0;
weights[2] = 0;
```

Durch diese Optimierung sollte der Nachteil der objektorientierten Architektur und der damit verbundenen tiefen Struktur kompensiert werden. Ein genauerer Performance-Test wurde nicht dokumentiert, jedoch konnte eine Verbesserung festgestellt werden.

Partielles Schleifen entrollen

Die Idee der Verwendung entstammt dem Quellcode von FANN, wo diese Technik an diversen Stellen Verwendung findet. In der eigenen Implementierung wurde dies der Übersichtlichkeit halber nur in der Propagierungs-Funktion (*Propagate*) in der Klasse (*NeuralNetwork*) angewendet, was eine Performance Steigerung in der Propagierens-Funktion um bis zu Faktor 2 brachte. Aus Zeitgründen wurde dies jedoch nicht näher analysiert.

Auszug aus dem Quellcode: Die erste Version stellt eine übliche Implementierung dar.

```
for(j=0;j != Layers[i].InputValuesCount;++j)
{
  net+=neuron->Weights[j] * Layers[i].InputValues[j];
}
```

Die gleiche Implementierung mit partiell entrollter Schleife:

```
int j = Layers[i].InputValuesCount & 3; //mod 4

//behandle Neuronen die nicht in der nachfolgenden
//4er Schritt loop behandelt werden können
switch (j)
{
  case 3:
    net+=neuron->Weights[2] * Layers[i].InputValues[2];
  case 2:
    net+=neuron->Weights[1] * Layers[i].InputValues[1]
  case 1:
    net+=neuron->Weights[0] * Layers[i].InputValues[0];
  case 0:
    break;
}

for(; j != Layers[i].InputValuesCount; j += 4)
{
  net +=neuron->Weights[j] * Layers[i].InputValues[j]+
        neuron->Weights[j+1] * Layers[i].InputValues[j+1]+
        neuron->Weights[j+2] * Layers[i].InputValues[j+2]+
        neuron->Weights[j+3] * Layers[i].InputValues[j+3];
}
```

Bedingte Anweisungen in Schleifen vermeiden

Die Vermeidung von bedingten Anweisungen in Schleifen ist eine einfache und Performance bringende Modifikation, welche jedoch oft nur auf Kosten von redundantem Code realisierbar ist, und damit die Pflege des Codes erschwert. Aus diesem Grund wurde diese nur an den kritischsten Stellen im Code durchgeführt.

Auszug aus dem Quellcode: Die erste Version stellt eine übliche Implementierung dar.

```
for(k=0;k<inputVectorCount;++k)
{
  if(i!=0)
  {
    sumDeltaErrWeightsNext[k] += weights[k] * error;
  }
  lastDeltaWeights[k]= ... ;
  weights[k]+=lastDeltaWeights[k];
}
```

Die gleiche Implementierung mit vorgezogener if-Bedingung:

```
if(i!=0)
{
  for(k=0;k<inputVectorCount;++k)
  {
    sumDeltaErrWeightsNext[k] += weights[k] * error;
    lastDeltaWeights[k]= ... ;
    weights[k]+=lastDeltaWeights[k];
  }
}
else
{
  for(k=0;k!=inputVectorCount;++k)
  {
    lastDeltaWeights[k]= ... ;
    weights[k]+=lastDeltaWeights[k];
  }
}
```

5.3 Anwendung

Zur Verwendung der Bibliothek wurde eine kleine Konsolen-Anwendung geschrieben, die neben dem Laden der Daten zum Training und Test des neuronalen Netzes zusätzlich Konfigurationsdateien für die Klassen *NeuronalNetwork*, *Backpropagation* und *GeneticAlgorithm* einliest, welche nach dem einfachen Schema `Varibale=Wert` aufgebaut sind. Da alle Konfigurationsklassen die gleiche Funktionalität, jedoch mit anderen Variablennamen erfüllen, besitzen sie eine gemeinsame Basisklasse *ConfigBase* (siehe Klassendiagramm 19).

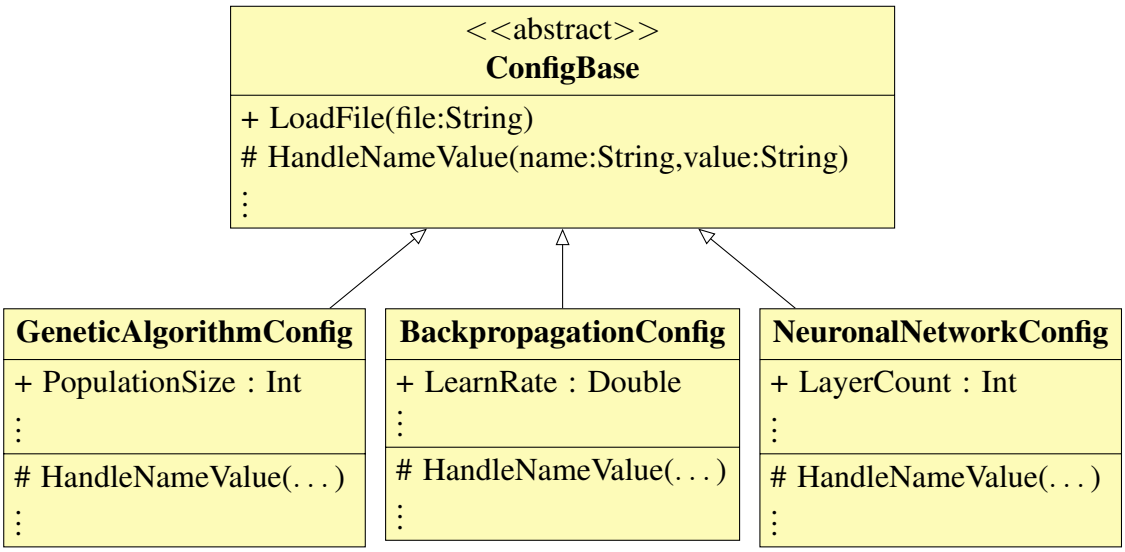


Abbildung 19: Klassendiagramm der Konsolenanwendung (aus Übersichtsgründen nicht vollständig)

Zum einfacheren Test der Ergebnisse wurde eine einfache K-Fache Kreuzvalidierung implementiert, diese versucht die Daten in K gleiche Teile aufzuteilen, wobei $K - 1$ Teile zum Training verwendet werden und der nicht verwendete Teil zum Testen. Dies wird K-Mal wiederholt, bis jeder Teil einmal zum Testen verwendet wurde. Bei einer nicht restfreien Aufteilung wird der Testteil um den Rest verlängert.

6 Versuchsaufbau

6.1 Daten

Im ersten Schritt wurde ein Tool in C# geschrieben, welches eine einfache Skalierung nach $[a, b]$ vornimmt mit $f(x) = a + \frac{(x-x_{min})*(b-a)}{x_{max}-x_{min}}$. Zusätzlich können die Datensätze zufällig gemischt werden und in ein einheitliches Format konvertiert werden. Als Format wurde dasselbe gewählt, das auch *FANN* verwendet, welches sich wie folgt darstellt:

```
1 n k l
2 x_{0,0} x_{0,1} ... x_{0,k} //Eingabewerte des ersten Datensatzes
3 y_{0,0} y_{0,1} ... y_{0,l} //Ausgabewerte des ersten Datensatzes
4 ⋮
5 x_{n,0} x_{0,1} ... x_{n,k} //Eingabewerte des letzten Datensatzes
6 y_{n,0} y_{0,1} ... y_{n,l} //Ausgabewerte des letzten Datensatzes
```

Abbildung 20: Schema des verwendeten Datenformats mit n=Anzahl Datensätze, k=Anzahl Eingabewerte und l=Anzahl Ausgabewerte

Für das Training von neuronalen Netzen standen zwei Datenbanken zur Verfügung:

MNIST Datenbank

Die Mixed National Institute of Standards and Technology[5] Datenbank besteht aus 70.000 Sätzen, wobei jeder Satz eine handgeschriebene Ziffer darstellt. Ein Satz besteht aus $28 \cdot 28$ Bytes, wobei jedes Byte einem Pixel Grauwert entspricht. Die Datenbank wird häufig zum Training auf neuronalen Netzen verwendet. Dementsprechend gibt es viele Vergleichswerte. Mit einem 2-Schichten-Netz, mit 300 Neuronen auf der ersten Neuronen Schicht, wurde eine Testfehlerrate von $4,7\%$ erzielt. Andere Konstellationen kommen auf niedrigere Werte, so erreicht ein 6-Schichten-Netz mit einer 784-2500-2000-1500-1000-500-10 Topologie eine Testfehlerrate von $0,39\%$.
Die Daten wurden einmal nach $[0, 1]$ und einmal nach $[-1, 1]$ skaliert, an der ursprünglichen Aufteilung von Trainingsdaten (60.000) und Testdaten (10.000) wurden keine Änderungen vorgenommen.
Als Topologie wurde 784-300-10 gewählt, die Anzahl der Eingabe-Neuronen ergibt sich

[5] `http://yann.lecun.com/exdb/mnist/`

aus der Anzahl der Eingabe-Daten ($28 \cdot 28$), die Anzahl der Ausgabe-Neuronen entspricht der Anzahl der Kategorien ($0, 1, 2, 3, 4, 5, 6, 7, 8, 9$), wobei der Ausgabevektor so encodiert wird, dass dieser nur am Index der entsprechenden Kategorie eine 1 aufweist und an den restlichen Stellen eine 0 beziehungsweise -1, womit zum Beispiel für die Kategorie 4 der Ausgabevektor $(0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$, bei einer Verwendung eines $[0, 1]$ Intervalls, entsteht. Die Anzahl der verdeckten Neuronen wurde von den Versuchen durch LeCun übernommen, unter anderem, um somit einen Referenzwert für die Fehlerrate zu haben.

Achsen/Rotations-Datenbank

Diese Datenbank wurde vom Zweitprüfer Prof. Dr. Christoph Dalitz bereitgestellt und besteht aus 1346 Datensätzen. Jeder Datensatz beinhaltet 4 Merkmale zu je einen Punkt, welcher entweder der Kategorie achsensymmetrisch oder rotationssymmetrisch zu geordnet wird. Die Merkmale werden durch Fließkommazahlen mit 4-stelliger Genauigkeit dargestellt und sind auf $[0, 1]$ skaliert. Die Zuordnung, ob ein Punkt eine Rotationssymmetrie oder Achsensymmetrie aufweist, ist durch die Werte ‘r’ oder ‘a’ dargestellt, welche mit $a = 1$ und $r = 0$ encodiert wurden.
Die Anzahl der Eingabe-Neuronen und Ausgabe-Neuronen wurde auf 4 und 1 festgelegt, da zwei Kategorien ähnlich wie beim XOR-Problem mit nur einem Neuron dargestellt werden können.
Um einen verlässlicheren Wert für die Güte des Trainings zu erhalten, wurde eine einfache Kreuzvalidierung mit $K = 5$ durchgeführt. Womit sich je eine Trainingsgröße von 1076 Datensätzen und eine Testgröße von 270 Datensätzen ergibt.

7 Versuchsdurchführung

Im Folgenden soll das Training der beiden vorgestellten Datenbanken dokumentiert werden. Da der Back-Propagation Algorithmus den üblichen Trainingsalgorithmus darstellt, wird zunächst dieser betrachtet und durch verschiedene Konstellationen versucht diesen zu verbessern. Anschließend wird das Verhalten des neuronalen Netzes nach dem Training, mithilfe eines genetischen Algorithmus, betrachtet. Zusätzlich wird das XOR-Problem betrachtet um die Ergebnisse aus der Arbeit [Kitano90] nachzuvollziehen.

7.1 Achsen/Rotations-Datenbank

Back-Propagation Lernen

Da es im Gegensatz zur MNIST Datenbank keine Referenz für die Anzahl der verdeckten Neuronen gibt, wurde eine möglichst optimale Anzahl mit einem einfachen Versuchsaufbau gesucht. Getestet wurde eine Neuronen-Anzahl zwischen 2 und 9. Basierend auf den gleichen initialen Gewichtswerten wurden 1.000, 10.000 und 100.000 Iterationen durchgeführt bis es entweder zu einem underfitting oder einem overfitting kam. Verwendet wurde das Inkrementelle-Training mit einer Lernrate von 0.5.

Wie der Tabelle 3 zu entnehmen ist, erweist sich die Topologie Nr. 3 (4-4-1) als die Vielversprechendste. Der Fehler entspricht dem relativen Fehler der falschen Klassifizierungen der Testdaten. Ein Ergebnis wird als Fehler gezählt, wenn die absolute Differenz zwischen Ist- und Soll-Ausgabe $\geq x$ ist, mit $x = \frac{(b-a)}{2}$, wobei a, b den Intervallgrenzen $[a, b]$ der Skalierung der Daten entspricht.

		1.000 Iterationen		10.000 Iterationen		100.000 Iterationen	
Nr.	Topologie	MSE	Fehler	MSE	Fehler	MSE	Fehler
1	4-2-1	0.0524	7.4%0	0.0522	7.33%	0.0520	7.18%
2	4-3-1	0.0517	7.48%	0.0516	7.25%	0.0510	7.33%
3	4-4-1	0.0505	7.25%	0.0489	7.18%	0.0482	6.96%
4	4-5-1	0.0502	7.25%	0.0485	7.03%	0.0478	7.48%
5	4-6-1	0.0501	7.25%	0.0483	7.11%	-	-
6	4-7-1	0.0497	7.03%	0.0459	7.48%	-	-
7	4-8-1	0.0499	7.11%	0.0482	7.18%	-	-
8	4-9-1	0.0496	7.18%	0.0444	7.77%	-	-

Tabelle 3: Topologievergleich mit der Achsen/Rotations-Datenbank

Da eine Kombination aller möglichen Parameterwerte nicht möglich ist, wurde je ein Parameter variiert, während die restlichen Parameter unverändert blieben. Angefangen wurde mit dem Vergleich der verschiedenen Datenskalierungen, wobei die logistische Funktion die besten Ergebnisse lieferte. Um die Idee von LeCun zur Vermeidung der Asymptote aufzugreifen, wurden die Daten zusätzlich noch auf das Intervall [0.1,0.9] skaliert, wodurch ein besseres Ergebnis erzielt werden konnte (siehe Tabelle 4).

Nr.	Skalierung	Aktivierungsfunktion	MSE	Fehler(Testdaten)
1	[0,1]	logistic	0.0520	7.48%
2	[-1,1]	tanh	0.4156	10.44%
3	[-1,1]	tanh (LeCun)	2.2769	26.00%
4	[0.1,0.9]	logistic	0.0332	6.81%

Tabelle 4: Daten-Skalierungsvergleich mit der Achsen/Rotations-Datenbank. Die Aktivierungsfunktion tanh (LeCun) entspricht, wie in Abschnitt 3.4.3, beschrieben der Funktion $f(x) = 1.7159 \cdot tanh(\frac{2}{3} \cdot x)$.

Basierend auf den Ergebnissen der Tabelle 4 wurde mit der logistischen Funktion und den auf $[0.1, 0.9]$ skalierten Daten weiter getestet. Wie den Ergebnissen Nr. 1 bis 6 in Tabelle 5 zu entnehmen ist, wurde zunächst Inkrementelles- und Batch-Training in Kombination mit verschiedenen Lernraten getestet. Diese Kombination wurde gewählt, da bei Kombination einer hohen Lernrate und Batch-Training schlechte Ergebnisse erwartet wurden, was die Ergebnisse im Nachhinein auch bestätigten. Insgesamt konnten mit Inkrementellem-Training in Kombination mit einer hohen Lernrate bessere Ergebnisse erzielt werden, der Abbildung 21 kann ein Vergleich des besten inkrementellen Versuchs und des besten Batch Versuchs entnommen werden.

Nr.	Lernrate	Stapel Größe	Momentum	Zerfallsrate	MSE	Fehler
1	0.5	1	0	0	0.0332	6.81%
2	0.5	1346	0	0	0.0932	10.37%
3	0.1	1	0	0	0.0332	6.66%
4	0.1	1346	0	0	0.0932	10.37%
5	0.01	1	0	0	0.0419	9.70%
6	0.01	1346	0	0	0.0339	6.96%
7	0.5	1	0.9	0	0.0369	7.11%
8	0.1	1	0.9	0	0.0341	7.11%
9	0.01	1	0.9	0	0.0332	6.66%
10	0.1	1	0	0.1	0.0456	10.37%
11	0.1	1	0	0.01	0.0351	7.40%
12	0.1	1	0	0.001	0.0334	7.03%

Tabelle 5: Back-Propagation Konstellationsvergleich mit der Achsen/Rotations-Datenbank

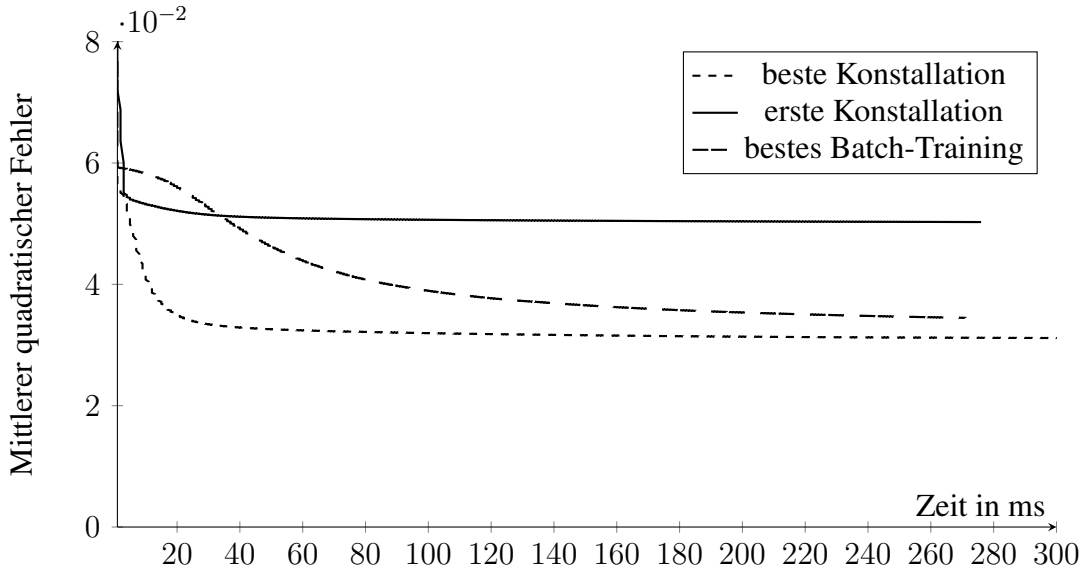


Abbildung 21: Darstellung des ersten Topologie Versuchs (4-4-1) (Nr. 3 aus Tabelle 3), der besten Konstellation insgesamt (Nr. 3 aus Tabelle 5), und der besten Batch-Training Konstellation (Nr. 6 aus Tabelle 5)

Anschließend wurden zwei weitere Versuche unternommen, um die Fehlerrate zu senken, einmal mit verschiedenen Momentum-Raten und anschließend mit verschiedenen Zerfallsraten, jedoch konnten keine Verbesserungen erreicht werden. Um auszuschließen, dass die initialen Gewichtswerte der Grund sind, wurden weitere Versuche mit anderen zufälligen initialen Werten versucht, wodurch nur eine geringe Verbesserung der Ergebnisse erreicht wurde. Das beste Ergebnis ist in Tabelle 6 aufgelistet.

		Fehler (Trainingsdaten)		Fehler (Testdaten)	
	MSE	Relativ	Absolut	Relativ	Absolut
Durchlauf 1/5	0.0331	6.87%	74	6.29%	17
Durchlauf 2/5	0.0347	6.87%	74	4.44%	12
Durchlauf 3/5	0.0323	6.31%	68	7.40%	20
Durchlauf 4/5	0.0317	6.22%	67	7.77%	21
Durchlauf 5/5	0.0311	6.50%	70	7.03%	19
Mittelwert	0.0326	6.56%	70.6	6.59%	17.8

Tabelle 6: Ergebnisse des besten Kreuzvalidierungs-Versuchs

In weiteren Konstellationen wurde eine weitere Schicht hinzugefügt mit 2 bis 4 Neuronen, jedoch konnten keine besseren Ergebnisse erzielt werden. Da unausgeglichene Daten ein generelles Problem für neuronale Netze darstellen, wurden diese in einem Versuch durch Reduzierung der achsensymmetrischen Datensätze und in einem anderen Versuch durch Duplizierung der rotationssymmetrischen Datensätze ausgeglichen. Durch diese Anpassung konnte die Fehlerrate für einzelne Validierungsdurchläufe verbessert werden, jedoch konnte die durchschnittliche Fehlerrate nicht verringert werden.

Genetisches Lernen

Aufgrund der Verwendung eines Zufallsgenerators wurde jeder Versuch jeweils 20-mal durchgeführt und über alle Iterationen der Mittelwert und die Streuung berechnet. Da ein besonderes Interesse an der Konvergenzgeschwindigkeit im Vergleich zum Back-Propagation Verfahren besteht, wurden die Konvergenzgeschwindigkeiten der genetischen Versuche mit dem besten Back-Propagation Versuch verglichen. Basierend auf den Versuchen in den Arbeiten [Kitano90] und [Montana89] wurde eine Population von 50 gewählt und ein Etilism von 2. In beiden Arbeiten konnte mithilfe genetischer Algorithmen eine bessere Konvergenzgeschwindigkeit als mit dem Back-Propagation Algorithmus erzielt werden. In den ersten Konstellationen wurden die verschiedenen Operationen verglichen, wobei neben den aufgelisteten Mittelwerten und Streuungen, das Konvergenzverhalten betrachtet wurde. Als Mutationsrate wurde zunächst 0.1 und

als Rekombinationsrate 0.8 gewählt, weitere Versuche mit anderen Raten brachten keine ersichtliche Verbesserung. In den Versuchen wurde immer nur eine Methode variiert und die restlichen beibehalten, die beste Methode aus jedem Versuch wurde in die nachfolgenden Versuche übernommen. Da die Ergebnisse bis auf eine Ausnahme nah beieinanderlagen (siehe Tabelle 7), wurde insbesondere anhand der Konvergenzgeschwingkeit entschieden. Die Operationen mit dem besten Konvergenzverhalten in den jeweiligen Versuchen waren Nr. 2, 5, 8, 10. Einen Vergleich mit der besten Back-Propagation Konstellation ist in der Abbildung 22 dargestellt.

Beschreibung		MSE		Fehler (Testdaten)	
Nr.	Mutationsoperation	Mittelwert	Streuung	Mittelwert	Streuung
1	UNBIASED-MUTATE-WEIGHTS	0.0499	0.0001	10.37%	0.00
2	BIAS ED-MUTATE-WEIGHTS	0.0338	0.0001	6.81%	0.16
3	MUTATE-NODES	0.0329	0.0002	6.90%	0.20
	Kreuzungsoperation				
4	CROSSOVER-NODES	0.0338	0.0002	6.90%	0.18
5	CROSSOVER-WEIGHTS	0.0337	0.0002	6.87%	0.18
6	CROSSOVER-ONEPOINT	0.0341	0.0002	6.96%	0.19
7	CROSSOVER-TWOPOINT	0.0340	0.0002	6.95%	0.20
	Rouletteverfahren				
8	FITTNESSBASED	0.0339	0.0002	6.88%	0.15
9	INDEXBASED	0.0338	0.0001	6.96%	0.17
	Trainingsverfahren				
10	Steady-State	0.0338	0.0002	6.97%	0.16

Tabelle 7: Genetische Operationen Vergleich mit der Achsen/Rotations-Datenbank

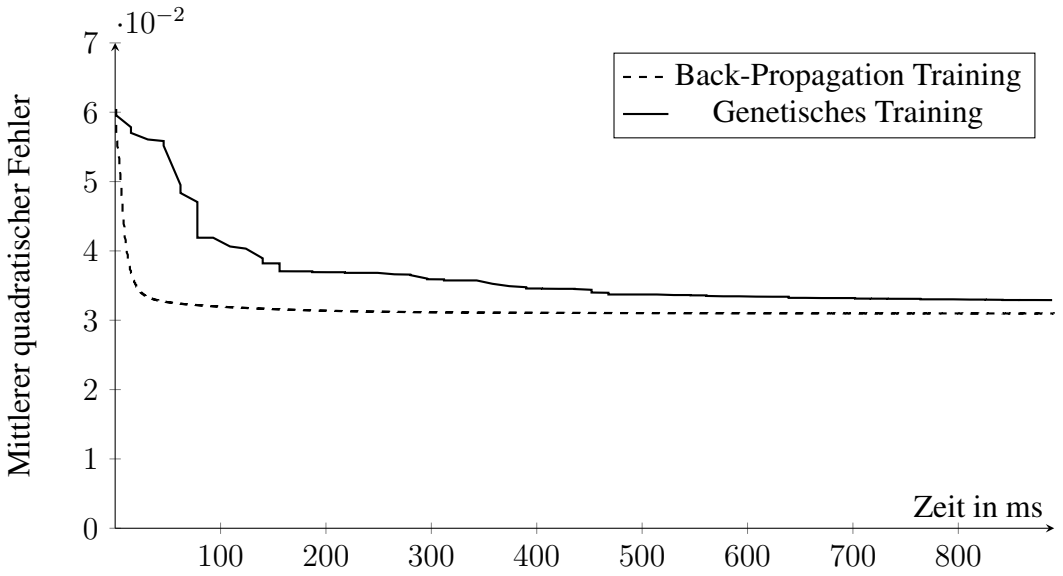


Abbildung 22: Darstellung der besten Back-Propagation Konstellation (Nr. 3 aus Tabelle 5), und des besten genetischen Versuchs (Nr. 10 aus Tabelle 7)

Durch eine anschließende Back-Propagation Ausführung auf dem genetisch trainierten Netz konnte der MSE mit wenigen Iterationen auf die gleichen Ergebnisse gebracht werden, wie mit Back-Propagation alleine. Insgesamt konnte jedoch kein besseres Lernverhalten, als mit Back-Propagation, festgestellt werden. Jedoch erscheint die Klassifizierung insgesamt als schwer trainierbar mit neuronalen Netzen, zumindest im Vergleich zu anderen Klassifizierungsproblemen z. B. der Klassifizierung von Ziffern mithilfe der MNIST Datenbank.

7.2 MNIST- Datenbank

Back-Propagation Lernen

Das Vorgehen zur Auswahl der einzelnen Parameter ist bei der MNIST Datenbank, mit Ausnahme der Iterations-Anzahl und der vorgegebenen Aufteilung in Trainings und Testdaten, mit dem Vorgehen bei der Achsen/Rotations-Datenbank identisch. Bei Klassifizierungsproblemen mit mehreren Klassen gilt eine Klasse als erkannt, wenn das entsprechende Neuron, welches diese Klasse repräsentiert, den höchsten Wert aufweist. Entsprechend gibt der Fehler die relative Anzahl der nicht erkannten Muster an. Wie der Tabelle 8 entnommen werden kann, konnte die Konstellationen mit Tangens Hyperbolicus Aktivierungsfunktion nicht konvergieren. Dieses Problem ist auf die hohen Netzeingaben zurückzuführen und der verwendeten $tanh$ Methode in der Bibliothek die ab einem Eingabewert > 9 eine 1 zurückliefert, welche bei der Ableitung 0 ergibt und somit keine Änderungen an den Gewichten vornimmt. Aus diesem Grund wurden die Konstellationen mit logistischen Funktionen näher untersucht.

Beschreibung				Fehler	
Nr.	Skalierung	Aktivierungsfunktion	MSE	Trainingsdaten	Testdaten
1	[0,1]	logistic	0.0061	0.49%	1.92%
2	[-1,1]	tanh	7.2056	-	89.68%
3	[-1,1]	tanh (LeCun)	17.6121	-	91.08%
4	[0.1,0.9]	logistic	0.0112	0.12%	2.46%

Tabelle 8: Daten-Skalierungsvergleich mit der MNIST-Datenbank 100 Iterationen

Durch zusätzliche 900 Iterationen konnte für beide Versuche, Nr. 1 und 2 der Tabelle 9, ein overfitting festgestellt werden, wobei die Trainingsdaten zu fast 100% erkannt wurden und sich die Erkennungsrate für die Testdaten verringerte. Dies lässt darauf schließen, dass eine verdeckte Schicht mit 300 Neuronen nicht das optimale Netz darstellt.

Beschreibung				Fehler	
Nr.	Skalierung	Aktivierungsfunktion	MSE	Trainingsdaten	Testdaten
1	[0,1]	logistic	0.0047	0.41%	1.97%
2	[0.1,0.9]	logistic	0.0025	0.02%	3.89%

Tabelle 9: Daten-Skalierungsvergleich mit der MNIST-Datenbank 1.000 Iterationen

Eine oft verwendete Technik besteht darin die Daten in Trainings-, Validierungs- und Testdaten aufzuteilen, wobei die Validierungsdaten dazu verwendet werden, um zu überprüfen, ob ein overfitting aufgetreten ist. In diesem Fall wird die vorherige Lösung verwendet. Diese Technik wurde jedoch hier nicht eingesetzt, da dies eine andere Aufteilung der Daten zur Folge hätte und die Ergebnisse somit nicht mehr mit anderen Arbeiten vergleichbar wären.
Um auszuschließen, dass die zufällige Initialisierung nahe an einer guten Lösung lag, wurden 5 weitere zufällig initialisierte Trainingsdurchläufe durchgeführt. Da die Ergebnisse zwischen 1.72% und 1.97% lagen, kann davon ausgegangen werden, dass dies nicht der Fall war.

Aufgrund der Fehlerrate von ca. 2%, welche unter der erwarteten Fehlerrate von $4,7\%$ lag wurde keine weitere Verbesserung der Fehlerrate erwartet und stattdessen das Konvergenzverhalten im Vergleich zum genetischen Training untersucht. Der Abbildung 23 können die zwei besten Versuche entnommen werden, hierbei ist jedoch zu beachten das aufgrund der unterschiedlichen Skalierung der MSE nicht direkt verglichen werden kann, so kann ein Ausgabe Neuron bei einer Ausgabe von 0 bei einer Skalierung von $[0, 1]$ zu einem MSE von $(0 - 0)^2 = 0$ und bei einer Skalierung von $[0.1, 0.9]$ zu einem MSE von $(0.1 - 0)^2 = 0.01$ führen. Dieses Problem ist Unteranderem beim Vergleich der Nr. 1 und Nr. 4 in der Tabelle 8 ersichtlich, wobei die logistische Funktion bei größerem MSE einen kleineren Trainingsdatenfehler erzielen kann.
Da die Skalierung $[0, 1]$ in diesem Fall ein besseres Ergebnis nach 100 Iterationen erzielen konnte und eine Abbruchbedingung basierend auf einer weiteren Aufteilung der Daten nicht vorgenommen wurde, wurde diese zum Vergleich mit genetischem Training verwendet.

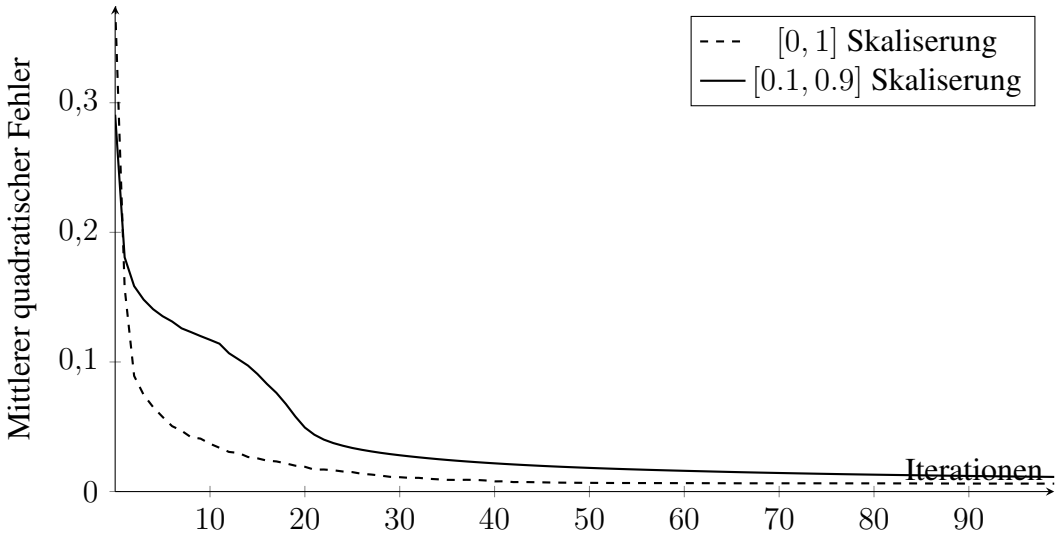


Abbildung 23: Darstellung der Versuche Nr. 1 und 4 aus Tabelle 8

Genetisches Lernen

Basierend auf den Ergebnissen bei der Achsen/Rotations-Datenbank wurde die Mutationsoperation BIAS-ED-MUTATE-WEIGHTS, die Kreuzungsoperation CROSSOVER-WEIGHTS und das Rouletteverfahren FITTNESSBASED gewählt. Diese Konstellation wurde einmal mit Steady-State und einmal ohne durchgeführt. In beiden Fällen konnte nur ein relativer Fehler von ca. 90% erreicht werden.

Weitere Versuche wurden mit unterschiedlichen Mutationsraten 0.01, 0.05, 0.10 und 0.5 durchgeführt. Außerdem wurden weitere einfache Initialisierungsvarianten getestet, wobei dabei lediglich die Intervalle variiert wurden, mit Ausnahme der von LeCun vorgestellten Initialisierungsmethode, welche im Abschnitt 3.4.5 beschrieben wurde. Abschließend wurden mit den Populationsgrößen 20 und 100 trainiert.
Mit keiner getesteten Variante konnten die anfänglichen Ergebnisse verbessert werden. Der beste Versuch kann Abbildung 24 entnommen werden.

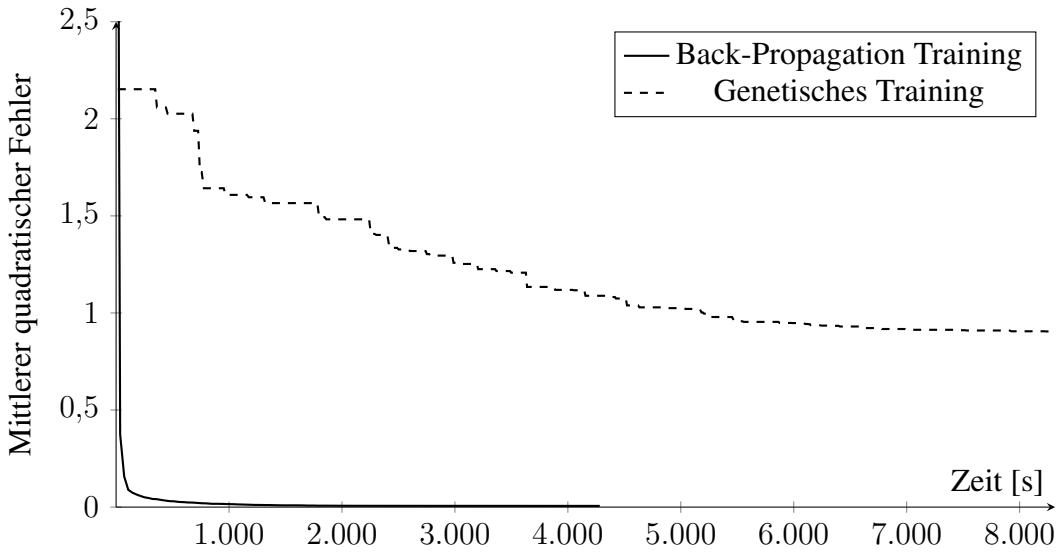


Abbildung 24: Darstellung des besten genetischen Versuchs im Vergleich zum Versuch Nr. 1 aus Tabelle 8 mit der MNIST-Datenbank

7.3 XOR- Problem

Im Folgenden bezeichnet eine Epoche die vollständige Iteration durch einen Trainingsdatensatz mit dem Back-Propagation Algorithmus.
Da mit den genetischen Versuchen anders als in den Arbeiten [Kitano90] und [Montana89] kein besseres Konvergenzverhalten, als mit Back-Propagation festgestellt werden konnte, wurde der XOR-Problem-Versuch in der Arbeit [Kitano90] untersucht. Dabei wurde aufgrund der sehr geringen Ausführungszeiten, die gleiche Annahme gemacht wie in der Arbeit, dass die Berechnung einer Population von 50 mit dem genetischen Ansatz, 25 Epochen mit dem Back-Propagation entspricht, wobei dieses Verhältnis aus verschiedenen Gründen nicht der Implementierung der Bibliothek entspricht. Mit einer Konstellation, ähnlich der Nr. 8 in Tabelle 7 und einem Back-Propagation Batch-Training mit einer Lernrate von 0.5, konnten ähnliche Ergebnisse erzielt werden wie in [Kitano90] (siehe Darstellung 25). Jedoch konnte ein inkrementelles Training mit einer Lernrate von 0.5 und einer Momentumrate von 0.9 bessere Ergebnisse erzielen. Weitere Untersuchungen der Ergebnisse in den Arbeiten konnten aufgrund fehlender Datenbanken, welche in den Arbeiten verwendet wurden, nicht vorgenommen werden.
Die Ergebnisse deuten darauf hin das die Vergleiche zwischen den genetischen Versuchen und dem Back-Propagation Versuchen in den Arbeiten basierend auf Batch-Training durchgeführt wurden und dadurch nicht ausgeschlossen werden kann das andere Back-Propagation Konstellationen ebenfalls besser sind als die restlichen genetischen Versuche, welche hier jedoch nicht rekonstruiert werden können.

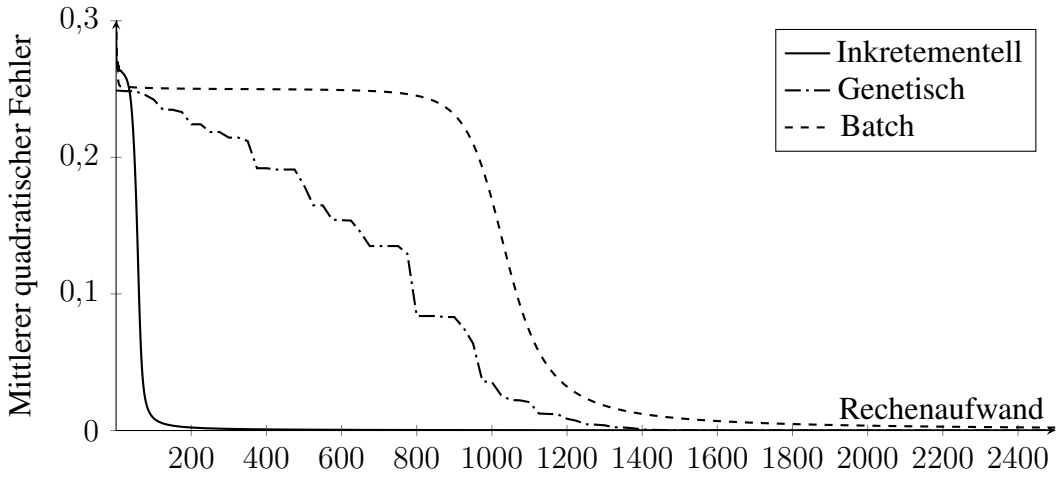


Abbildung 25: XOR-Problem Vergleich

8 Fazit und Ausblick

Gegenstand dieser Arbeit war unter anderem das Training neuronaler Netze mittels Back-Propagation und genetischen Algorithmen, dabei wurde ein besonderer Fokus auf den Vergleich der beiden Verfahren gelegt. Das Training mittels genetischem Ansatz konnte insbesondere bei kleineren Netzen gute Ergebnisse erzielen, jedoch konnte mithilfe des Back-Propagation, sowohl beim XOR-Problem, als auch bei der Achsen/Rotations-Datenbank bessere Ergebnisse mit inkrementellem Training erzielt werden.
Bei der Klassifizierung der MNIST Daten hingegen, konnten keine akzeptablen Ergebnisse mittels genetischem Training erzielt werden, wohingegen mit Back-Propagation bessere Ergebnisse erzielt werden konnten, als erwartet. Nach wenigen Iterationen konnte ein Testdaten Fehler von 5% erreicht werden, nach 100 Iterationen ein Fehler von $1 - 2\%$, weiteres Training führte aufgrund der Topologie zu overfitting, wobei ein früheres overfitting nicht auszuschließen ist.
Die besten Ergebnisse für die betrachteten Klassifizierungsprobleme konnten mit inkrementellem Back-Propagation Training, einer hohen Lernrate und der Vermeidung von Asymptote erzielt werden.
Insgesamt stellten sich genetische Algorithmen zum optimieren von Gewichten bei neuronalen Netzen als weniger geeignet, als das Back-Propagation Verfahren heraus, zumindest für die betrachteten Probleme.

Zwei wichtige Aspekte wurden in dieser Arbeit nicht näher betrachtet, welche sich jedoch als besonders kritische Faktoren beim Training herausstellten. Zum einen die Wahl einer geeigneten Topologie und zum anderen die Initialisierung der Gewichte. In beiden Fällen könnten genetische Ansätze eine gute Wahl zur Optimierung darstellen, wie unter anderem in der Arbeit [Yeremia13] beschrieben.

Literatur

[Callan03] R. Callan: *Neuronale Netze im Klartext.* Pearson Studium, 1. Aufl. (2003)

[Darken92] C. Darken, J. Moody: *Towards faster stochastic gradient search.* (1992)

[Denker90] J. Denker, Y. LeCun: *Transforming Neural-Net Output Levels to Probability Distributions.* (1990)

[Haykin99] S. Haykin: *NEURAL NETWORKS: A Comprehensive Foundation.* Pearson Education, 2. Aufl. (1999)

[Heaton08] J. Heaton: *Introduction to Neural Netwroks with Java.* Heaton Research Inc., 2. Aufl. (2008)

[Kitano90] H. Kitano: *Empirical Studies on the Speed of Convergence of Neural Network Training using Genetic Algorithms.* (1990)

[Kriesel05] D. Kriesel: *D. Kriesel.* `http://www.dkriesel.com/science/neural_networks` (2005)

[LeCun98] Y. LeCun, L. Bottou, G. Orr, K. Müller: *Effizient BackProp.* (1998)

[Lippe02] W. Lippe: *Einführung in Neuronale Netze.* `http://cs.uni-muenster.de/Professoren/Lippe/lehre/skripte/wwwnnscript/` (2002)

[Luger97] G. Luger,W. Subblefield: *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Addison Wesley Longman, Inc., 1. Aufl. (1997)

[Mitchell97] T. Mitchell: *Machine Learning.* McGraw-Hill Science/Engineering/Math, 1. Aufl. (1997)

[Mitchell99] M. Mitchell: *An Introduction to Genetic Algorithms.* MIT Press, 5. Aufl. (1999)

[Montana89] D. Montana, L. Davis: *Training Feedforward Neural Networks Using Genetic Algorithms.* (1989)

[Nissen03] . Nissen: *Implementation of a Fast Artificial Neural NetworkLibrary.* `ftp://ftp.jaist.ac.jp/pub/sourceforge/f/fa/fann/fann_doc/1.0/fann_doc_complete_1.0.pdf` (2003)

[Obitko98] M. Obitko: *GENETIC ALGORITHMS.* `http://www.obitko.com/tutorials/genetic-algorithms/about.php` (1998)

[Petersohn05] H. Petersohn: *Data Mining: Verfahren, Prozesse, Anwendungsarchitektur.* Oldenbourg Wissenschaftsverlag, 1. Aufl. (2005)

[Rojas93] R. Rojas: *Theorie der neuronalen Netze: Eine systematische Einführung.* Springer-Verlag, 1. Aufl. (1993)

[Sarle02] W. Sarle: *Neural Network FAQ.* `ftp://ftp.sas.com/pub/neural/FAQ.html` (2002)

[Wilson03] R. Wilson, T. Martinez: *The general inefficiency of batch training for gradient descent learning.* (2003)

[Yeremia13] H. Yeremia, N. Yuwono, P. Raymond, W. Budiharto: *GENETIC ALGORITHM AND NEURAL NETWORK FOR OPTICAL CHARACTER RECOGNITION.* (2013)

[Zell94] A. Zell: *Simulation neuronaler Netze.* Addison Wesley, 1. Aufl. (1994)

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis