FAKULTÄT FÜR !NFORMATIK

Genetische Algorithmen und Neuronale Netze

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Radu Ion

Matrikelnummer 9727184

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao. Univ. Prof. DI Dr. techn. Karl Grill

Wien, 13.10.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Für Olimpia,
Persis und
Christian

Kurzfassung

Genetische Algorithmen versuchen, Prinzipien der Evolution (Mutation, Rekombination und Selektion) in Computerprogrammen nachzubilden. Dazu gibt es verschiedene Ansätze, deren Vor- und Nachteile kurz besprochen werden.

Eines der Wunderwerke ist das menschliche Gehirn. Obwohl die Neuronen, aus denen es besteht, relativ einfach aufgebaut sind, ist das Gehirn zu Leistungen fähig, die einem Computer nur schwer einprogrammiert werden können, wie etwa Sprach- oder Gesichtserkennung. Neuronale Netze sollen den Lernvorgang, der im Gehirn stattfindet, nachbilden, um ähnliche Fähigkeiten zu erlangen. Auch hierfür werden einige Modelle vorgestellt.

In der Natur arbeiten diese beiden Konzepte hervorragend zusammen. Daher liegt es nahe, die Bestandteile und Parameter von neuronalen Netzen durch genetische Algorithmen optimieren zu lassen. Leider treten dabei Schwierigkeiten auf, wie lange Laufzeiten oder mangelnde genetische Vielfalt. Es werden einige Lösungsvorschläge für diese Probleme erläutert.

Eine mögliche Anwendung für genetische Algorithmen und neuronale Netze ist die Zeitreihenanalyse. Dafür gibt es zwar auch statistische Verfahren wie die lineare und nichtlineare Regression. Diesen ist aber jeweils ein bestimmtes Modell zugrunde gelegt, von dem man nicht weiß, ob es die Zeitreihe adäquat beschreibt. Mithilfe eines Programms von der Hochschule für Technik und

Wirtschaft Dresden kann man Zeitreihen auch durch genetische Algorithmen und neuronale Netze analysieren lassen. Als Beispiel wurden die Tagesschlusskurse der Aktie der Deutschen Bank verwendet. Im Rahmen mehrerer Tests mit diesem Programm zeigt sich, dass nicht nur zu kleine, sondern auch zu große neuronale Netze sich negativ auswirken können. Schließlich wird jedoch ein neuronales Netz gefunden, das die Änderung der Aktienkurse hervorragend prognostiziert und damit die Leistungsfähigkeit von genetischen Algorithmen und neuronalen Netzen eindrucksvoll unter Beweis stellt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

John Holland untersuchte genetische Algorithmen bereits in den sechziger Jahren, Warren McCulloch und Walter Pitts experimentierten mit neuronalen Netzen sogar schon in den vierziger Jahren des letzten Jahrhunderts. Es handelt sich also beileibe nicht um neue Konzepte. Sie führten jedoch einige Jahrzehnte lang ein Schattendasein und waren lediglich Spezialisten bekannt. Liest man nämlich ältere Autoren wie etwa Heistermann [3], findet man immer wieder die Kritik, dass einfache Modelle den Problemen nicht gerecht werden, aber komplizierte die Rechenleistung des Computers überfordern und deshalb nicht praktikabel sind.

Wie wir alle wissen, vervielfacht sich die Leistungsfähigkeit moderner Computer in atemberaubendem Tempo. Berechnungen, die vor nicht allzu langer Zeit Tage erforderten, sind mittlerweile innerhalb von Sekunden erledigt. Das bedeutet, dass immer größere Populationen immer komplexerer neuronaler Netze verwendet werden können, ohne dass die Laufzeit ins Unermessliche steigt.

Grund genug, sich im Rahmen dieser Arbeit einen Überblick über die in den letzten Jahrzehnten entwickelten Konzepte zu verschaffen, sie miteinander zu vergleichen und schließlich ihre Leistungsfähigkeit in praktischen Tests zu überprüfen.

Kapitel 2

Grundlagen: Genetische Algorithmen

2.1 Motivation

Wir leben im Zeitalter der Optimierung. Längst genügt es nicht mehr, dass Vorgänge funktionieren, ständig werden Kosten und Zeitaufwand minimiert bzw. Produktivität und Gewinn maximiert. Man mag diesem „Optimierungswahn“ kritisch gegenüberstehen, aber letztlich ist es ihm zu verdanken, dass Computer heutzutage nicht mehr 27 Tonnen wiegen.

Die Mathematik liefert zahlreiche Algorithmen zur Optimierung, die bei einfachen Modellen hervorragend funktionieren. Leider ist die Wirklichkeit immer um einiges komplizierter. Häufig gibt es unübersichtlich viele Einflussgrößen, die nicht alle einzeln optimiert werden können, sondern nur in bestimmten Kombinationen das Ergebnis verbessern.

Man stelle sich ein Optimierungsproblem als eine Gebirgslandschaft im Nebel vor, wo man den höchsten Gipfel finden soll. Viele mathematische Algorithmen erinnern an einen Bergsteiger, der mit viel Mühe einen Gipfel erklimmt, nur, um oben festzustellen, dass andere Gipfel noch höher sind. Genetische Algorithmen

hingegen kann man mit einem ganzen Team von Bergsteigern vergleichen, die nicht nur gleichzeitig klettern, sondern auch noch untereinander kommunizieren und einander wertvolle Tipps für den Aufstieg geben.

Das Vorbild genetischer Algorithmen ist die Natur, eine Meisterin der Optimierung: Flöhe springen 200 mal weiter, als ihre Körperlänge beträgt.
Jesus-Echsen laufen problemlos übers Wasser. Spinnen produzieren Fäden, die 10 mal dünner als ein Haar, aber im Verhältnis zur Fadendicke 5 mal reißfester als Stahlseile sind. Wie konnten sich diese unglaublichen Fähigkeiten entwickeln?

2.2 Die Mechanismen der Evolution

2.2.1 Mutation

Das Erbgut wird nicht immer exakt an die Nachkommen weitergegeben. Ab und zu passieren „Kopierfehler“. Es ist purer Zufall, ob die Mutation eine Verbesserung oder Verschlechterung bewirkt, aber auf jeden Fall ist dafür gesorgt, dass die Individuen einer Population einander nur ähnlich, aber nicht völlig identisch sind.

2.2.2 Rekombination

Die Nachkommen zweier solcher Individuen mit kleinen Unterschieden haben Erbanlagen, die eine Mischung aus denen ihrer Eltern sind. So entsteht unter

Umständen eine Kombination von Eigenschaften, die es in den Generationen davor nur einzeln, aber noch nicht zusammen gab. Auch hier ist es wieder Zufall, ob diese neue Kombination sich vorteilhaft oder nachteilig auswirkt.

2.2.3 Selektion

Lebewesen haben immer wieder mit Krankheiten, Fressfeinden und Nahrungsknappheit zu kämpfen. Wenn eine durch Mutation und/oder Rekombination entstandene Erbanlage einen Vorteil im Überlebenskampf bringt, haben deren Träger eine höhere Überlebens- und somit auch Fortpflanzungswahrscheinlichkeit. Selbst, wenn der Vorteil nur gering ist, wird im Verlauf tausender Generationen die Häufigkeit dieser Erbanlage langsam zunehmen, bis sie schließlich die ganze Population hat. Umgekehrt sterben nachteilige Kombinationen von Erbanlagen rasch wieder aus.

Es gibt noch weitere Mechanismen der Evolution, aber diese drei sind die wichtigsten. Ohne Mutation werden nur die vorhandenen Erbanlagen durchmischt, aber es kommen keine neuen hinzu. Ohne Rekombination ist Evolution zwar möglich, aber die Lebewesen ohne sexuelle Fortpflanzung und damit ohne Rekombination sind alle recht primitiv geblieben (Bakterien, Algen, Pilze), weil sich bei komplizierten Organismen Mutationen meist negativ auswirken und die Rekombination an Bedeutung gewinnt. Ohne Selektion bleiben die schlechteren Erbanlagen erhalten und hemmen die Weiterentwicklung.

2.3 Das Modell

Wie kann man diese offenbar äußerst erfolgreichen Mechanismen für Optimierungsalgorithmen nutzbar machen? Würde man den Code von Programmen der Mutation und Rekombination aussetzen, würden wohl in den seltensten Fällen wieder lauffähige Programme herauskommen. Erfolgversprechender ist es, das Programm möglichst allgemein zu codieren, möglichst viele Details darin Steuerungsvariablen zu überlassen, die Steuerungsvariablen in Vektoren zusammenzufassen und die Vektoren dann der Mutation, Rekombination und Selektion auszusetzen.

Eine Population bestehe aus n Individuen:

$$P = \{\vec{i}_1, \vec{i}_2, ..., \vec{i}_n\}$$

Jedes dieser Individuen werde durch einen Vektor mit seinen Erbanlagen charakterisiert. Die Positionen innerhalb dieses Vektors heißen Gene, die Werte an diesen Positionen heißen Allele. Da diese beiden Begriffe oft verwechselt werden: Ein Beispiel für ein Gen wäre „Haarfarbe“, ein Beispiel für ein Allel wäre „blond“. Sei m die Anzahl der Gene und A die Menge aller möglichen Allele, dann sehen die Individuen folgendermaßen aus:

$$\vec{i} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \text{ mit } a_j \in A$$

Beispiele: $A = \{0,1\}$: Individuen werden durch einen Binärcode dargestellt.

$A = ¡$: Individuen sind Punkte im Raum, Gene ihre Koordinaten.

Die Menge I aller möglichen Individuen ist folglich die Menge aller m-dimensionalen Vektoren aus Elementen von A:

$$I = A^m$$

Manche dieser Individuen sind in der (aktuellen) Population enthalten, manche nicht:

$$P \subseteq I$$

Schließlich soll es noch eine so genannte Fitnessfunktion geben, die jedem Individuum aus I eine reelle Zahl zuordnet, von der später die Überlebens- bzw. Fortpflanzungswahrscheinlichkeit abhängig sein wird:

$$F : I \to ¡$$

Geht es um ein Optimierungsproblem, bei dem eine Zielfunktion minimiert oder maximiert werden soll, wird üblicherweise die Zielfunktion als F gewählt. Insgesamt lässt sich das Modell folgendermaßen zusammenfassen:

$$I = A^m$$
$$P \subseteq I$$
$$F : I \to ¡$$

2.4 Die Simulation der Mutation

Sei p die Wahrscheinlichkeit, mit der ein Allel mutieren soll. Im Fall $A = \{0,1\}$ wird einfach 0 zu 1 oder 1 zu 0. Im Fall $A = ¡$ hingegen muss man außerdem festlegen, wie stark das Allel a sich verändern soll. Eine Möglichkeit besteht darin, dass das mutierte Allel gemäß einer Normalverteilung mit Erwartungswert a verteilt ist. Im Rahmen eines genetischen Algorithmus müssen üblicherweise tausende Individuen generiert werden. Um Rechenzeit zu sparen, wird als Verteilung für das mutierte Allel auch gerne einfach die Gleichverteilung auf $[a - \varepsilon, a + \varepsilon]$ gewählt.

Die Wahl der Mutationsparameter p und ε will gut überlegt sein. Wählt man sie zu groß, werden gute Individuen immer wieder vom rechten Weg abgebracht, was die Laufzeit unnötig verlängert. Wählt man die Mutationsparameter zu klein, können die Mutationen nur langsame Fortschritte bringen. Schlimmer noch, die Individuen werden leichter in einem lokalen Optimum „gefangen“, aus dem sie durch kleine Mutationen nicht mehr „entkommen“ können, um doch noch das globale Optimum zu erreichen.

Um das zu vermeiden, beginnen Algorithmen in solchen Fällen üblicherweise mit großen Werten für p und ε, die dann nach und nach reduziert werden. So können die Individuen zu Beginn noch lokalen Optima entkommen. Wenn es dann am Ende bereits gute Individuen gibt, lassen die Mutationen nach. Es bleibt das Problem, vernünftige Startwerte und Reduktionsgeschwindigkeiten zu finden. In der Praxis gibt man dem System zunächst willkürliche Werte vor. Wenn die Laufzeit zu lange oder die Lösungen unbefriedigend sind, werden die Werte verändert und der Vorgang wiederholt.

Eine interessante Lösung für dieses Problem haben Rechenberg und Schwefel vorgeschlagen, die so genannte Metamutation. Dabei werden für die Mutationsparameter einfach zusätzliche Gene reserviert, d. h. es findet gleichzeitig und ohne viel zusätzlichen Aufwand auch eine Evolution der Mutationsparameter statt: Unsinnig niedrige oder hohe Mutationsparameter sterben aus.

Heistermann [3] hat das in einem genetischen Algorithmus getestet. Zunächst erhöhen sich die Mutationsparameter, weil die Individuen noch weit vom Optimum entfernt sind und starke Mutationen gelegentlich massive Verbesserungen ermöglichen. Irgendwann nähern sich die Individuen dem gesuchten Optimum und die Mutationsparameter beginnen zu sinken, weil starke Mutationen nun eher zerstörerische Wirkung haben.

Es liegt auf der Hand, dass dieses System, bei dem sich die Mutationsparameter automatisch selbst regulieren, bessere Ergebnisse bringt als jedes System, wo die Mutationsparameter und ihre Veränderung von außen starr vorgegeben werden.

2.5 Die Simulation der Rekombination

Seien $\vec{a}, \vec{b} \in A^m$ zwei beliebige Individuen. Nun soll ein Nachkomme $\vec{c}$ entstehen, der die Gene der „Eltern“ kombiniert. Am naheliegendsten ist die zufällige Vermischung:

$$P(c_j = a_j) = P(c_j = b_j) = \frac{1}{2} \qquad j = 1,\ldots,m$$

Das hat den Vorteil, dass man sich keine Gedanken über die Reihenfolge der Gene zu machen braucht, weil sie völlig unerheblich ist. Eine andere Idee ist das so genannte Crossing Over. Hier werden nach einer zufällig gewählten Stelle s die Gene vertauscht, wodurch zwei Nachkommen entstehen:

$$c_j = \begin{cases} a_j \text{ für } j = 1,...,s \\ b_j \text{ für } j = s+1,...,m \end{cases} \qquad d_j = \begin{cases} b_j \text{ für } j = 1,...,s \\ a_j \text{ für } j = s+1,...,m \end{cases}$$

Gibt man jeder der $m-1$ Stellen zwischen den Genen dieselbe Wahrscheinlichkeit $\frac{1}{m-1}$, als Bruchstelle zu dienen, dann stammen Nachbargene auch nur mit Wahrscheinlichkeit $\frac{1}{m-1}$ von verschiedenen Elternteilen. Je weiter Gene auseinander liegen, desto größer wird diese Wahrscheinlichkeit. Daher muss man sich gut überlegen, welche Gene nebeneinander liegen und dann im Rekombinationsprozess seltener getrennt werden sollen.

Auch hier hat Heistermann [3] Tests durchgeführt, welche Rekombinations-methode häufiger bzw. schneller das globale Optimum findet. Das Crossing Over erreicht nicht nur mit größerer Wahrscheinlichkeit das globale Optimum, sondern schafft das auch noch mit weniger Generationen, also in kürzerer Zeit. Woran liegt das?

Ein wichtiges Anwendungsgebiet für genetische Algorithmen ist die Konstruktion von Fahrzeugen, die einen möglichst geringen Luftwiderstand aufweisen. Angenommen, die Gene 90-95 sind für die Form der Rückspiegel verantwortlich, und im Verlauf des Rekombinationsprozesses erhält ein

Individuum zufällig genau die optimale Konfiguration dafür. Dann haben viele seiner Nachkommen dieselben optimalen Rückspiegel. Auch, wenn die anderen Parameter noch nicht optimal eingestellt sind, haben diese Nachkommen einen kleinen Vorteil, der sich innerhalb der Population immer mehr ausbreiten wird. Später findet ein anderes Individuum eine optimale Konfiguration für den Kotflügel. Seine Nachkommen müssen nun nicht mehr den Rückspiegel optimieren, sondern erhalten früher oder später den optimalen Rückspiegel durch Crossing Over.

Wie bereits bei dem Beispiel mit den Bergsteigern angedeutet: Genetische Algorithmen verfolgen nicht nur gleichzeitig mehrere verschiedene Lösungsansätze, sondern „kommunizieren“ auch untereinander, indem sie sich via Crossing Over vielversprechende Teillösungen zukommen lassen.

Erhalten Nachkommen hingegen immer eine willkürliche Mischung der Gene ihrer Eltern, werden in den seltensten Fällen die Gene 90-95 unverändert erhalten bleiben, die zufällig gefundene Teillösung geht immer wieder verloren.

2.6 Die Simulation der Selektion

Das eigentliche Ziel ist ja, Individuen $\vec{i}$ mit möglichst großer Fitness $F(\vec{i})$ zu finden. Die Selektion ist dafür zuständig, dass „gute“ Gene von Generation zu Generation häufiger und „schlechte“ Gene seltener werden. Es muss also abhängig von $F(\vec{i})$ eine gewisse Bevorzugung bzw. Benachteiligung bei der Weitergabe der Gene geben. Dafür gibt es mehrere Möglichkeiten:

2.6.1 Selektion durch Tod

Das ist die einfachste Variante. Durch Rekombination bzw. Mutation entstehen aus n Individuen $\lambda \cdot n$ Nachkommen. Sie werden gemäß ihrer Fitness sortiert und nur die n besten von ihnen bilden die nächste Generation. Die übrigen sterben und können ihre Gene nicht mehr weitergeben.

2.6.2 Selektion durch Fortpflanzung

Aus der Fitness $F(\vec{i})$ eines Individuums wird seine Wahrscheinlichkeit $p(F(\vec{i}))$ berechnet, mit der es zur Fortpflanzung herangezogen wird. Diese Funktion p muss monoton wachsend sein, d. h. größere Fitness bedeutet mehr Nachkommen.

2.6.3 Selektion durch Lebensdauer

Aus der Fitness $F(\vec{i})$ eines Individuums wird seine Lebensdauer $L(F(\vec{i}))$ berechnet. Auch hier muss L monoton wachsend sein, d. h. größere Fitness bedeutet längere Lebensdauer und somit auch mehr Zeit, Nachkommen zu zeugen.

2.6.4 Vergleich der Selektionsmethoden

Alle drei Varianten sowie Kombinationen von ihnen wurden schon für genetische Algorithmen implementiert. Alle drei finden auch in der Natur statt. Einerseits gibt es Tierarten, die in Rudeln leben, wo das dominante Männchen die höchste Fortpflanzungswahrscheinlichkeit hat. Die anderen überleben, können ihre Gene aber selten weitergeben. Andererseits gibt es auch Tierarten, wo sich nur jeweils ein Männchen und ein Weibchen zusammentun, und wo daher jedes Tier, das überlebt, dieselbe Fortpflanzungswahrscheinlichkeit hat.

Auch hier hat Heistermann [3] die drei Varianten und Kombinationen von ihnen getestet. Die eindeutig besten Ergebnisse wurden mit ausschließlicher Selektion durch Tod erzielt.

Der Grund lässt sich am besten wieder anhand des Beispiels mit dem Luftwiderstand erklären: Ein Individuum hat den optimalen Rückspiegel, ein anderes den optimalen Kotflügel gefunden. Letzteres hat die höhere Fitness, da sich der Kotflügel stärker auf den Luftwiderstand auswirkt. Bei der Selektion durch Fortpflanzung oder Lebensdauer werden daher die Rückspiegel benachteiligt, können sich schlechter in der Population verbreiten und sterben eventuell sogar aus. Mit anderen Worten: Alle überlebenden Individuen sollten sich mit der gleichen Wahrscheinlichkeit fortpflanzen können, damit sich Fortschritte, egal ob groß oder klein, in der Population verbreiten können.

Aus demselben Grund sollte λ nicht zu klein gewählt werden. Bei $\lambda = 10$ (also zehnmal so viele Nachkommen wie Eltern und nur jedes zehnte Individuum überlebt) ist sichergestellt, dass sich eine vielversprechende Teillösung rasch ausbreitet und via Crossing Over den anderen Individuen zur Verfügung steht.

2.7 Gray-Code

Im Fall $A = \{0,1\}$ geschieht die Mutation wie gesagt, indem 0 zu 1 wird und umgekehrt. Wie verändert das binär gespeicherte Zahlen? Sehen wir uns dazu die Binärcodes der Zahlen 0 bis 8 an:

Zahl	Binärcode
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

Der Binärcode von 3 kann durch Mutation des letzten Bits problemlos in den Binärcode von 2 übergehen. Hingegen kann 3 nur zu 4 werden, wenn drei Bits mutieren, was sehr unwahrscheinlich ist. Angenommen, das Allel ist momentan 3, sollte aber optimalerweise 4 sein, dann stehen die Chancen des genetischen Algorithmus schlecht, das Optimum zu finden.

Abhilfe schafft der so genannte Gray-Code (siehe z. B. auch Forster [4]). Sei $(b_1, b_2, ..., b_m)$ der Binärcode einer Zahl, dann ist der Gray-Code $(g_1, g_2, ..., g_m)$ folgendermaßen definiert:

$$g_1 := b_1$$
$$g_i := \begin{cases} 0 & \text{bei } b_i = b_{i-1} \\ 1 & \text{bei } b_i \neq b_{i-1} \end{cases} \quad \text{für } i = 2, 3, ..., m$$

Diese Zuordnung kann zugleich als Abbildungsvorschrift jener Abbildung f verstanden werden, die den Binärcode auf den Gray-Code abbildet:

$$f : \{0,1\}^m \rightarrow \{0,1\}^m$$
$$(b_1, b_2, ..., b_m) \mapsto (g_1, g_2, ..., g_m)$$

Einfach ausgedrückt: Der Gray-Code liefert die erste Stelle des Binärcodes und dann jeweils die Information, ob sich die nächste Stelle von der vorherigen unterscheidet. Da Binärcodes nur aus 0 und 1 bestehen, ist der zugehörige Binärcode dadurch eindeutig bestimmt. Die Abbildung f ist also injektiv. Da ihre Definitions- und Wertemenge identisch sind, ist f folglich auch surjektiv und bijektiv. Es ist demnach zulässig, die Erbanlagen eines Individuums in Form eines Gray-Codes statt eines Binärcodes anzugeben. Doch was bringt das außer einem erhöhten Programmieraufwand?

Eine Binärzahl werde um 1 erhöht. Wie verändert sie sich bzw. ihr Gray-Code?

1. Fall: Die letzte Stelle war bisher 0, dann wird sie 1. Es ändert sich nur die letzte Stelle des Binärcodes und somit auch des Gray-Codes.

2. Fall: Die letzte Stelle war bisher 1. Davor stehen eventuell weitere 1. Allgemeine Form …01…1 Wird jetzt um 1 erhöht, ergibt sich … 10…0 Vor der Erhöhung stehen am Schluss lauter 1, nach der Erhöhung lauter 0. In beiden Fällen unterscheiden sich die Stellen nicht voneinander, was im Gray-Code lauter 0 ergibt. Es hat sich im Gray-Code also nur eine einzige Stelle verändert!

Hier die Gray-Codes der Zahlen 0 bis 8:

Zahl	Binärcode	Gray-Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100

In der Tat unterscheiden sich die Gray-Codes zweier benachbarter Zahlen immer nur an einer Stelle. Es genügt also immer eine einzige Mutation, um eine Erhöhung bzw. Senkung um 1 zu bewirken.

2.8 Konkrete Modelle

Rechenberg und Schwefel entwickelten das so genannte Evolutionsverfahren. Hierbei wird $A = ¡$ verwendet, und alle (!) Gene mutieren gemäß einer Normalverteilung $N(a,\sigma)$, wobei a das Allel vor der Mutation ist und σ der Metamutation unterworfen wird, d. h. als Gen mutiert und weitervererbt wird. In diesem Modell gab es ursprünglich keine Rekombination, sie wurde erst in späteren Varianten hinzugefügt. Die Selektion erfolgt durch Tod: In der ersten Version noch mit $n = 1$ und $\lambda = 1$, d. h. es gibt nur ein Individuum mit einem Nachkommen, dasjenige der beiden mit der größeren Fitness überlebt. Im Laufe der Zeit wurden die Modelle durch Erhöhung von n und λ aufwendiger.

Ein ganz anderer Ansatz sind die Genetic Algorithms von Holland und Goldberg. Hier ist $A = \{0,1\}$. Mutationen finden sehr selten statt, dafür ist die Rekombination durch Crossing Over umso wichtiger. Die Selektion erfolgt durch Fortpflanzung anstatt Tod: Je größer die Fitness eines Individuums, desto häufiger wird es zur Fortpflanzung herangezogen. Die Elterngeneration stirbt auf jeden Fall, unabhängig von deren Fitness. In der folgenden Tabelle sind die Unterschiede kurz zusammengefasst:

	Evolutionsverfahren	Genetic Algorithms
A	¡	{0,1}
Mutation	immer	selten
Rekombination	nein	ja
Selektion	durch Tod	durch Fortpflanzung

Das Evolutionsverfahren selektiert aggressiv, weil immer nur ein kleiner Teil überlebt. Die Individuen werden einander rasch ähnlich, Rekombination bringt nicht viel. Die häufigen Mutationen sind nötig, um ein Mindestmaß an Vielfalt zu gewährleisten. Die Genetic Algorithms selektieren hingegen eher sanft, weil suboptimale Gene nicht sofort aussterben. Die Vielfalt ist größer, wodurch Rekombination aussichtsreicher und Mutationen verzichtbar werden.

Kapitel 3

Grundlagen: Neuronale Netze

3.1 Motivation

Herkömmliche Computerprogramme treffen Entscheidungen aufgrund starrer Kriterien, die einmal programmiert und dann normalerweise nicht mehr verändert werden. Viele Probleme können Computer damit hervorragend lösen, von den Grundrechnungsarten bis zur Mondlandung.

Soll ein Computer jedoch z. B. anhand der Bildpunkte eines Fotos entscheiden, ob es sich um eine weibliche oder männliche Person handelt, wird es schwierig sein, starre Kriterien zu programmieren, die für alle Menschen auf der Welt unabhängig vom Gesichtsaudruck funktionieren. Warum aber ist dieselbe Entscheidung für einen Menschen meist einfach?

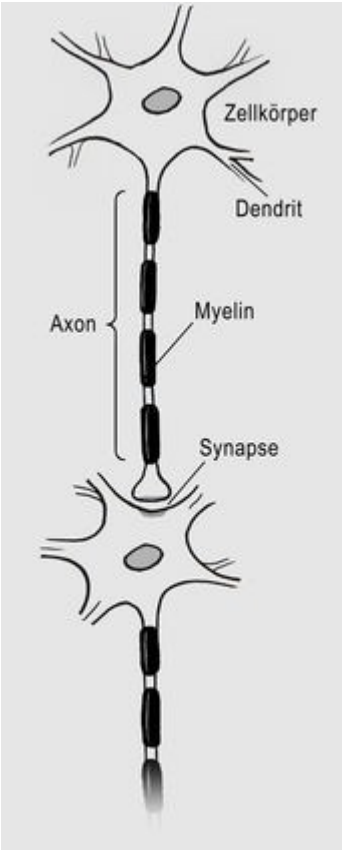
Das menschliche Gehirn wird nicht programmiert, sondern es lernt. Es bekommt immer wieder die unterschiedlichsten Gesichter zu sehen und assoziiert im Laufe der Zeit diverse Gesichtsmerkmale mit den Geschlechtern. Dabei ist es sehr flexibel: Wenn ein Gesichtsmerkmal verdeckt ist, werden andere herangezogen.

Wenn sich Gesichtsmerkmale verändern, lernt das Gehirn einfach um, z. B. dass Ohrringe kein frauenspezifisches Merkmal mehr sind.

Es liegt auf der Hand, dass bei dieser und zahllosen anderen Aufgabenstellungen die Vorgangsweise des Gehirns einem starren Computerprogramm weit überlegen ist. Neuronale Netze versuchen daher, den Lernprozess zu imitieren, der im Gehirn stattfindet. Dabei steht nicht im Vordergrund, die biologischen Vorgänge möglichst exakt nachzubilden, was wohl auch zu schwierig wäre, sondern eine einfache Struktur zu schaffen, die quasi dazu trainiert werden kann, Leistungen zu erbringen, die auf anderem Wege schwer oder gar nicht möglich wären.

3.2 Die Vorgänge im Gehirn

Das menschliche Gehirn enthält über 10^{11} Nervenzellen (Neuronen). Sie bestehen aus einem Zellkörper, von dem viele kleine Äste (Dendriten) sowie ein einziger großer Ast (Axon) ausgehen. Die Verbindungsstellen zwischen dem Axon eines Neurons und den Dendriten anderer Neuronen heißen Synapsen.



Quelle: IFP [5]

Im Ruhezustand hat das Neuron eine Oberflächenspannung von -70mV. Kommen nun Spannungsimpulse von einem oder mehreren Dendriten, steigt die elektrische Spannung vorübergehend. Wenn sie eine bestimmte Schwelle überschreitet, wird ein Spannungsimpuls an das Axon abgegeben. Sobald dieser an der Synapse eintrifft, werden chemische Botenstoffe freigesetzt, die in die Ionenkanäle der benachbarten Dendriten gelangen und einen Spannungsimpuls in der nächsten Nervenzelle erzeugen können. Manche Synapsen geben den Spannungsimpuls zur Gänze, zum Teil oder gar nicht weiter. Ja, es gibt sogar Synapsen, die die Polarität umkehren und damit einen weiteren Impuls erschweren, anstatt ihn auszulösen (siehe auch Zöller-Greer [1]).

Wie lernt nun dieses System? Das folgende Beispiel soll das veranschaulichen, es ist aber zugegebenermaßen stark vereinfacht:

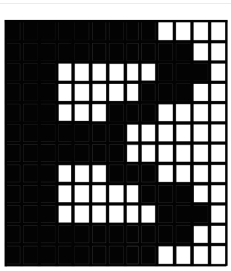
Ein Kind wächst in einer Gegend auf, in der die meisten Blumen rot sind. Wenn jemand über Blumen spricht, wird das Kind an rote Blumen denken. Die Synapse zwischen dem Neuron „Blume“ und dem Neuron „rot“ gibt den Impuls in voller Stärke weiter. Ein zweites Kind hat erst selten rote Blumen gesehen, die Synapse gibt den Impuls abgeschwächt weiter. Ein drittes Kind hat noch nie eine rote Blume gesehen, die Synapse gibt den Impuls gar nicht weiter. Die Hirnforschung hat festgestellt, dass Synapsen ihr Verhalten ändern, wenn sie aktiviert werden. Vereinfacht gesagt: Jedes Mal, wenn das Kind eine rote Blume sieht, wird die entsprechende Synapse „leitfähiger“. Das Kind „lernt“, dass Blumen rot sein können.

Das ist übrigens auch der Grund, warum man den Lernstoff nicht einmal kurz vor der Prüfung, sondern mehrmals durchgehen sollte. Mit jeder Wiederholung werden die betroffenen Synapsen aktiviert und leitfähiger. Wenn Synapsen andererseits lange nicht aktiviert werden, verlieren sie ihre Leitfähigkeit mit der Zeit oder sterben gar ab. Man vergisst den Lerninhalt.

In den letzten Jahrzehnten wurden unzählige Modelle entwickelt, die auf ähnlichen Prinzipien aufbauen. Von ganz einfachen, die sich an den biologischen Vorgängen orientieren, bis zu komplizierten Algorithmen, die es so im Gehirn nicht gibt. Die wichtigsten dieser Modelle sollen in den nächsten Kapiteln kurz vorgestellt werden, für detailliertere Informationen zu diesen Modellen siehe auch Ertel [2].

3.3 Hopfield-Netze

Wenn ein Mensch einen Buchstaben sieht, erkennt er ihn auch dann noch, wenn er geringfügig verfälscht ist. Zerlegen wir den Platz, an dem der Buchstabe steht, in n Bildpunkte. Wenn ein Bildpunkt schwarz ist, also zum Buchstaben gehört, erhält er den Wert 1, wenn er weiß ist, den Wert -1:



Quelle: ACM [6]

Somit kann man jeden Buchstaben als Vektor darstellen:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \text{ mit } x_j \in \{-1, 1\}$$

Für jeden Bildpunkt konstruieren wir ein Neuron, das diesen Bildpunkt korrigieren kann, wenn er verfälscht sein sollte. Von außen erhalte das Neuron i den Gesamtspannungsimpuls

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

Die Gewichte w_{ij} repräsentieren hierbei den Einfluss der Synapsen. Bei $w_{ij} > 0$ leitet die Synapse den Impuls weiter, bei $w_{ij} = 0$ blockiert sie ihn und bei $w_{ij} < 0$ dreht sie dessen Polarität um. Wie beim wirklichen Axon hängt das Ergebnis davon ab, ob eine bestimmte Schwelle θ erreicht wird:

$$x_i = \begin{cases} -1 & \text{bei } \sum_{j=1}^{n} w_{ij} x_j < \theta \\ 1 & \text{sonst} \end{cases}$$

Wie kann dieses System lernen, Muster zu erkennen? Genau wie das Gehirn lebenslang kein einziges Neuron hinzubekommt, sondern nur durch die Veränderung der Synapsen lernt, sollen zum Erkennen der Muster die Gewichte passend eingestellt werden und zwar mit der Hebb-Regel:

$$w_{ij}^{neu} = w_{ij}^{alt} + \eta x_i x_j$$

Wenn die Neuronen i und j gleichzeitig aktiv sind und damit auch die dazwischen liegende Synapse, wird deren Leitfähigkeit erhöht. Die Lernrate η bestimmt, wie stark sich das Lernen auf die Gewichte auswirkt.

Bezeichnen wir die m Referenzmuster, die erkannt werden sollen, mit

$$\vec{q}^{1} = \begin{pmatrix} q_1^1 \\ q_2^1 \\ \vdots \\ q_n^1 \end{pmatrix} \quad \vec{q}^{2} = \begin{pmatrix} q_1^2 \\ q_2^2 \\ \vdots \\ q_n^2 \end{pmatrix} \quad \cdots \quad \vec{q}^{m} = \begin{pmatrix} q_1^m \\ q_2^m \\ \vdots \\ q_n^m \end{pmatrix}$$

und initialisieren wir die Gewichte mit $w_{ij}^{alt} = 0$, dann erhalten wir nach dem Lernen aller Muster $k=1,2,...,m$ mit der Hebb-Regel insgesamt

$$w_{ij} = \eta \sum_{k=1}^{m} q_i^k \cdot q_j^k \qquad (i \neq j)$$

Die Gewichte für $i = j$ werden auf 0 gesetzt, da kein Neuron mit sich selbst verbunden sein soll. Ein verfälschtes Muster $\vec{x}$ wird durch die Neuronen mit diesen Gewichten verändert. Das muss solange wiederholt werden, bis alle Bildpunkte gleich bleiben. Das Verfahren konvergiert immer. Bei m<<n, wenn es also viel mehr Neuronen als Muster gibt, auch meist gegen das richtige Muster. Wenn aber zu viele Muster gelernt werden müssen, häufen sich die Fehler.

3.4 Assoziativspeicher

Oft genügt es nicht, zu einem Muster „ähnliche“ Muster zu finden, sondern das neuronale Netz soll bei den Inputs $\vec{x}^k$ die Outputs $\vec{y}^k$ generieren (k=1,2,…m). Für jeden Input-Bildpunkt j und jeden Output-Bildpunkt i gibt es ein Neuron und dazwischen Synapsen mit den Gewichten w_{ij}. Das neuronale Netz funktioniert perfekt, wenn sich die Input-Impulse genau auf das aufsummieren, was als Output-Impuls erwartet wird:

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

Schreibt man die Vektoren $\vec{x}^k$ spaltenweise zur Matrix X zusammen, sowie die $\vec{y}^k$ zu Y, dann bedeutet das nichts anderes als

$$Y = W \cdot X$$

Wenn die Neuronen wieder nach der Hebb-Regel lernen, ergibt sich bei $\eta = 1$

$$w_{ij} = \sum_{k=1}^{m} y_i^k \cdot x_j^k$$

Das ist eine Skalarmultiplikation einer Zeile von Y mit einer Zeile von X. Um es als Matrixmultiplikation schreiben zu können, bräuchten wir aber eine Zeile von Y und eine Spalte von X. Die Transposition von X behebt das:

$$W = Y \cdot X^T$$

Wir setzen das in die obige Matrixgleichung ein:

$$Y = W \cdot X = (Y \cdot X^T) \cdot X = Y \cdot (X^T \cdot X)$$

Wenn $X^T \cdot X$ die Einheitsmatrix ist, d. h. X eine orthogonale Matrix ist und die Input-Muster in X zueinander orthonormal sind, dann ist die obige Matrix-gleichung erfüllt. In diesem Fall kann also garantiert werden, dass das neuronale Netz perfekt arbeitet. Da hinter den Formeln die Hebb-Regel steckt, die Gewichte zwischen erwünschten Input-Output-Kombinationen verstärkt, funktioniert das System häufig auch bei nicht orthonormalen Input-Mustern, allerdings wie bei Hopfield-Netzen nur für $m<<n$.

3.5 Methode der kleinsten Quadrate

Wenn man hingegen zu viele Muster vorgegeben hat, wird eine perfekte Übereinstimmung nicht mehr möglich sein. Stattdessen versucht man, den Fehler zwischen den ausgegebenen und den erwünschten Mustern zu minimieren, und zwar mit der Methode der kleinsten Quadrate.

Das Neuron i gibt als Muster k aus: $\sum_{j=1}^{n} w_{ij} \cdot x_j^k$

Das Neuron i sollte als Muster k ausgeben: y_i^k

Zu minimieren ist also: $E = \sum_{k=1}^{m} \left(\sum_{j=1}^{n} w_{ij} \cdot x_j^k - y_i^k \right)^2$

Die x- und y-Werte sind vorgegeben, nur die Gewichte können eingestellt werden. Daher ist nach ihnen partiell abzuleiten. Da j schon für den Summationsindex vergeben ist, bezeichnen wir das Gewicht, nach dem abgeleitet wird, mit w_{ir}:

$$\frac{\partial E}{\partial w_{ir}} = \sum_{k=1}^{m} 2 \left(\sum_{j=1}^{n} w_{ij} \cdot x_j^k - y_i^k \right) \cdot x_r^k = 0$$

$$\sum_{k=1}^{m} \sum_{j=1}^{n} w_{ij} \cdot x_j^k \cdot x_r^k = \sum_{k=1}^{m} y_i^k \cdot x_r^k$$

$$\sum_{j=1}^{n} \left(w_{ij} \sum_{k=1}^{m} x_j^k \cdot x_r^k \right) = \sum_{k=1}^{m} y_i^k \cdot x_r^k$$

Wie im letzten Kapitel kann man auch das als Matrixgleichung schreiben:

$$W \cdot (X \cdot X^T) = Y \cdot X^T$$

Es lässt sich zeigen, dass es immer mindestens eine Lösung W gibt und dass jede Lösung stets ein lokales Minimum von E darstellt.

3.6 Delta-Regel

Bekanntlich liefert der Gradient (bestehend aus obigen partiellen Ableitungen) die Richtung des stärksten Anstiegs, wechselt man das Vorzeichen, die Richtung des stärksten Abstiegs. Es liegt nahe, die Gewichte in diese Richtung zu verändern, um zum Minimum von E zu gelangen (Gradientenabstiegsverfahren). Die Lernrate η bestimmt wieder, wie stark sich das Lernen auf die Gewichte auswirkt:

$$\begin{aligned} w_{ir}^{neu} &= w_{ir}^{alt} - \eta \frac{\partial E}{\partial w_{ir}} \\ &= w_{ir}^{alt} - \eta \sum_{k=1}^{m} 2 \left(\sum_{j=1}^{n} w_{ij} \cdot x_j^k - y_i^k \right) \cdot x_r^k \\ &= w_{ir}^{alt} + 2\eta \sum_{k=1}^{m} \left(y_i^k - \sum_{j=1}^{n} w_{ij} \cdot x_j^k \right) \cdot x_r^k \end{aligned}$$

Definiert man $z_i^k := \sum_{j=1}^{n} w_{ij} \cdot x_j^k$ als den tatsächlichen Output im Gegensatz zum angestrebten Output y_i^k, sowie $\tilde{\eta} := 2\eta$, dann ergibt sich

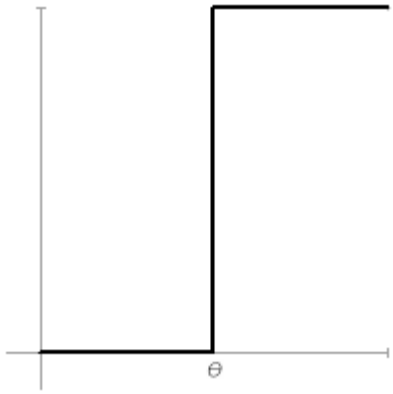
$$w_{ir}^{neu} = w_{ir}^{alt} + \tilde{\eta} \sum_{k=1}^{m} \left(y_i^k - z_i^k \right) \cdot x_r^k$$

Diese so genannte Delta-Regel hat den unschätzbaren Vorteil, dass man beim Hinzukommen neuer Muster nicht wie bei der Methode der kleinsten Quadrate wieder ganz von vorne beginnen muss, sondern das System quasi „dazulernen“ kann.

3.7 Backpropagation

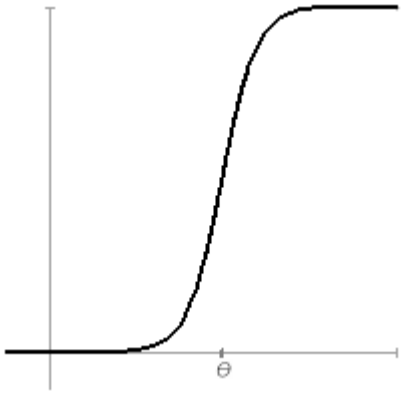
Alle bisherigen Modelle haben die Input-Signale nur linear verarbeitet. Damit neuronale Netze nichtlineare Probleme lösen können, müssen sie zwischen Ein- und Ausgabeneuronen weitere Schichten von Neuronen haben. Das alleine genügt jedoch nicht, denn Verknüpfungen von beliebig vielen linearen Abbildungen sind gleichfalls linear. Wieder dient die Natur als Vorbild: Das Axon gibt den Impuls erst weiter, wenn eine bestimmte Schwelle θ erreicht wird, d. h. sie wendet auf die eingehenden Signale eine nichtlineare so genannte Aktivierungsfunktion ähnlich der folgenden an:

$$f(x) = \begin{cases} 0 & x < \theta \\ 1 & x \geq \theta \end{cases}$$



Diese Aktivierungsfunktion ist jedoch bei $x = \theta$ unstetig und daher nicht differenzierbar. Diesen Nachteil hat die Fermi-Funktion nicht:

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



Statt $\sum_{j=1}^{n} w_{ij} x_j$ soll das Neuron i also $x_i = f\left(\sum_{j=1}^{n} w_{ij} x_j \right)$ ausgeben.

Wie im letzten Kapitel werden mit dem Gradientenabstiegsverfahren die Gewichtsveränderungen hergeleitet:

$$w_{ij}^{neu} = w_{ij}^{alt} + \eta \sum_{k=1}^{m} \delta_i^k \cdot x_j^k$$

Die δ_i^k werden für die Neuronen der „letzten“ Schicht, die für die Ausgabe zuständig ist, analog zur Delta-Regel aus den Differenzen zwischen dem angestrebten Output y_i^k und dem tatsächlichen Output x_i^k berechnet:

$$\delta_i^k = x_i^k(1 - x_i^k)(y_i^k - x_i^k)$$

Bei den Neuronen davor ist das nicht möglich. Hier werden die δ rekursiv aus den nachfolgenden δ berechnet. Man könnte auch sagen: Der Fehler pflanzt sich beginnend bei der Ausgabeschicht von hinten nach vorne durch alle Neuronen fort und stellt überall die Gewichte ein. Daher auch der Name Backpropagation.

$$\delta_i^k = x_i^k(1 - x_i^k)\sum_r \delta_r^k \cdot w_{ri}$$

Kapitel 4

Problembehandlung: Genetische Algorithmen und neuronale Netze

4.1 Motivation

In den letzten beiden Kapiteln haben wir zwei mächtige Hilfsmittel kennengelernt: Die genetischen Algorithmen, die der Evolution nachempfunden sind, und die neuronalen Netze, die das Gehirn zum Vorbild haben. Welches dieser beiden Prinzipien ist leistungsfähiger? Ist es so leistungsfähig, dass man auf das andere verzichten kann?

In der Natur sind beide aufeinander angewiesen. Einen Goldfisch kann man noch so lange mit Lerninhalten konfrontieren, er wird sie nie verarbeiten und intelligent darauf reagieren können. Ihm hat die Evolution nicht die genetische Ausstattung verschafft, komplizierte Dinge lernen zu können. Andererseits: Was würde es einem Baby nützen, mit dem leistungsfähigen menschlichen Gehirn zur Welt zu kommen, wenn es keine Eltern, Spielkameraden, Lehrer etc. hätte, von denen es lernen könnte?

Für die menschliche Intelligenz ist beides nötig: Ein Tausendejahren dauernder Evolutionsprozess, der ein extrem lernfähiges Gehirn hervorbringt, und ein Jahre dauernder Lernprozess, der dieses Gehirn mit Wissen, Ideen und Fertigkeiten füllt, sodass es sein volles Potential entfalten kann. Es ist ein wenig so wie mit Hardware und Software: Das eine ohne das andere ist völlig nutzlos, beide zusammen sind aber zu erstaunlichen Leistungen fähig.

Selbstverständlich muss man dem Vorbild der Natur nicht folgen und kann genetische Algorithmen und neuronale Netze auch unabhängig voneinander verwenden. Das funktioniert bei vielen Anwendungen auch zufriedenstellend, aber gerade bei komplexeren Aufgaben kann es Probleme geben:

Lässt man neuronale Netze nur durch Lernverfahren wie z. B. Backpropagation trainieren, dann spielt die Startkonfiguration eine große Rolle, also welche Neuronen mit welchen Gewichten untereinander verbunden sind. Es ist nicht auszuschließen, dass das Lernverfahren ein lokales Optimum ansteuert, dort hängenbleibt und das globale Optimum niemals findet.

Wenn anderseits das tatsächliche Optimum z. B. bei $\pi = 3{,}14159265...$ liegt, wie soll ein rein genetischer Algorithmus mit seinen Mutationen und Rekombinationen genau auf diesen Wert kommen? Typischerweise liefern genetische Algorithmen Ergebnisse in der Nähe des Optimums, aber nie oder nur mit unvertretbar großem Aufwand das genaue Optimum.

Kombiniert man genetische Algorithmen mit neuronalen Netzen, profitiert man von den Stärken beider Systeme und eliminiert zugleich deren Schwächen.

4.2 Anwendungsgebiete für genetische Algorithmen in neuronalen Netzen

4.2.1 Bestimmung der Gewichte

Man kann die Gewichte lediglich durch Lernverfahren wie Backpropagation trainieren lassen, dann besteht jedoch wie gesagt die große Gefahr, nur ein lokales Optimum zu finden. Genetische Algorithmen mit großen Populationen durchsuchen den Raum möglicher Lösungen sehr gründlich und können das globale Optimum meist ungefähr lokalisieren. Idealerweise folgt dann noch ein Lernverfahren, um das exakte Optimum zu bestimmen.

Eine Möglichkeit ist, die Gewichte mit $A = \{0,1\}$ binär zu kodieren und den Gray-Code aus Kapitel 2.7 zu verwenden. Nicht leicht zu beantworten ist die Frage, wie viele Bits man jedem Gewicht zur Verfügung stellen soll. Nimmt man zu wenige, haben die Gewichte nicht viel Spielraum und können die ihnen gestellte Aufgabe möglicherweise nicht erfüllen. Nimmt man zu viele, erhält man eine sehr große Anzahl von Genen.

Bei der Alternative $A = ¡$ beansprucht jedes Gewicht nur ein einziges Gen für sich. Die deutlich reduzierte Anzahl von Genen verkürzt nicht nur die Rechenzeit pro Generation, sondern braucht außerdem weniger Generationen bis zur Lösung. Folglich können auf diese Art viel größere neuronale Netze verwendet werden.

4.2.2 Festlegen der Netzwerkarchitektur

Ein Parameter, der die Leistungsfähigkeit des neuronalen Netzes massiv beeinflusst, ist die Anzahl der Neuronen. Stellt man zu wenige zur Verfügung, können die Informationen nicht adäquat verarbeitet werden und die Leistung lässt zu Wünschen übrig (siehe z. B. Ertel [2]). Überraschenderweise wirken sich aber auch zu viele Neuronen negativ aus. Wie kann man das erklären?

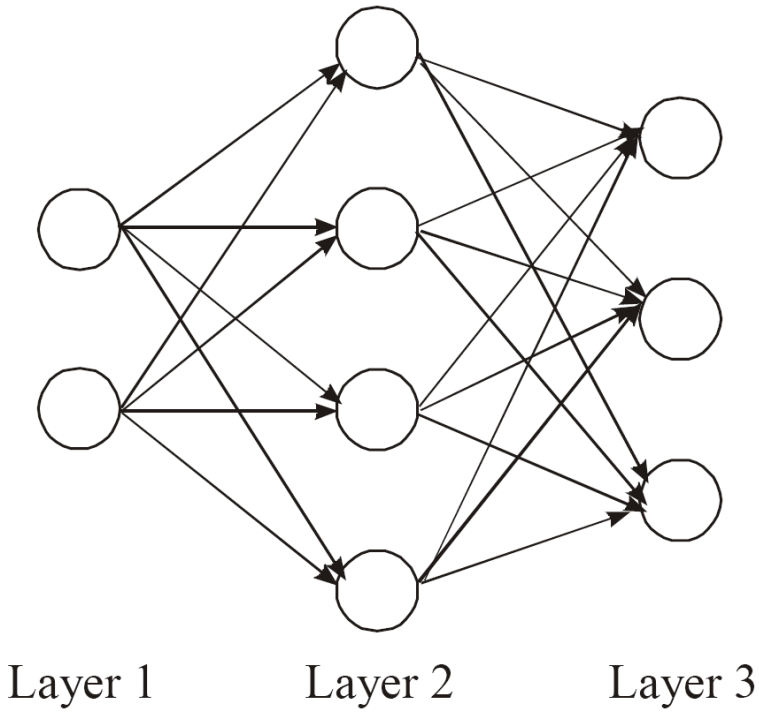
Es gibt zwei Arten von Studenten: Die einen lernen den Stoff auswendig, die anderen versuchen, die Zusammenhänge zu verstehen. Klarerweise ist letzteres vorzuziehen, weil man sich dann in einer leicht veränderten Situation immer noch zurechtfindet. Ähnlich kann man sich das auch bei neuronalen Netzen vorstellen: Bei einer quasi unbegrenzten Anzahl von Neuronen wird das neuronale Netz die Inputmuster exakt auswendig lernen, weil es ja bestrebt ist, den Fehler zu minimieren. Nur, wenn es dafür nicht ausreichend Neuronen zur Verfügung hat, ist es gleichsam gezwungen, in den Inputmustern Zusammenhänge aufzuspüren.

Es geht jedoch nicht allein um die Anzahl der Neuronen, sondern auch darum, welche mit welchen verbunden sind. Der naheliegendste Ansatz, einfach alle mit allen zu verbinden, entspricht einerseits nicht den natürlichen Vorgängen und führt bei n Neuronen zu

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

Gewichten. Eine Verdopplung der Neuronen bedeutet also in etwa eine Vervierfachung der Gewichte und damit der Gene.

Ein anderer Ansatz sind die so genannten Feed-Forward-Netze, die natürlichen Vorbildern entsprechen. Hierbei sind die Neuronen in Schichten (englisch Layer) angeordnet und die Neuronen einer Schicht jeweils ausschließlich mit den Neuronen der nächsten Schicht verbunden:



Quelle: Forster [4]

Dadurch kann ein Großteil der Gewichte eingespart werden. Das wirft jedoch neue Fragen auf: Wie viele Schichten soll es geben? Wie viele Neuronen sollen in den Schichten liegen? Sollen es in allen Schichten gleich viele sein oder nicht?

Traditionell wurden diese Fragen durch Ausprobieren, also nach der Methode „trial and error“ beantwortet. Es liegt auf der Hand, dass dabei nur ein kleiner Teil der möglichen Netzwerkarchitekturen getestet wird und dass fast zwangsläufig die für das jeweilige Problem optimale Netzwerkarchitektur

unentdeckt bleibt. Genetische Algorithmen haben auch hier weitaus bessere Chancen, das Optimum zu finden.

4.2.3 Auswahl des Lernverfahrens

In Kapitel 3 haben wir einige Methoden kennengelernt, mit denen neuronale Netze trainiert werden können. Teilweise sind sie natürlichen Vorgängen nachempfunden wie die Hopfield-Netze, teilweise sind es mathematische Optimierungsalgorithmen wie die Methode der kleinsten Quadrate. Es sind jedoch abseits von Natur und Mathematik noch viele andere Lernverfahren denkbar, die für ein bestimmtes Problem vielleicht besser geeignet sein könnten. Um diese zu finden, kann man sein Lernverfahren in größtmöglicher Allgemeinheit formulieren und alle Details durch genetische Algorithmen bestimmen lassen.

4.2.4 Einstellen der Lernparameter

Egal, ob man sich von Anfang an für ein bestimmtes Lernverfahren entscheidet oder es durch genetische Algorithmen suchen lässt, in jedem Fall bleibt die Frage, wie die Lernparameter zu wählen sind. Ein sehr wichtiger Lernparameter ist beispielsweise die Lernrate η, die bestimmt, wie stark ein einzelner Lernvorgang die Gewichte verändern kann. Eine zu kleine Lernrate sorgt für langsame Konvergenz und ein hohes Risiko, in einem lokalen Optimum hängenzubleiben, entspricht also einem Individuum, das sehr langsam lernt bzw. manche Dinge überhaupt nicht lernen kann. Eine zu große Lernrate tendiert dazu, erfolgversprechende Lösungswege immer wieder zu verlassen, und entspricht

einem Individuum, das schnell lernt, aber beim Auftauchen neuer Lerninhalte die alten rasch wieder vergisst.

Es ist ein wenig so wie bei der Mutation (siehe Kapitel 2.4): Auch dort war es empfehlenswert, zunächst starke Mutationen zuzulassen, um lokalen Optima entkommen zu können, und später die Mutationen abzuschwächen, um die Konvergenz zu erleichtern. Erreicht wurde das durch Metamutation, also das Vererben der Mutationsparameter, die dadurch ebenfalls der Selektion ausgesetzt sind (siehe Heistermann [3]).

Analog kann man auch die Lernparameter als Gene kodieren, sodass die Selektion zunächst automatisch starke Lernfortschritte bevorzugt, aber am Schluss jene Individuen aussterben lässt, deren Lernparameter zum Vergessen des bereits Gelernten führen.

4.3 Probleme

4.3.1 Laufzeitproblem

Im letzten Kapitel wurden viele Dinge aufgezählt, die genetische Algorithmen leisten können. Je mehr Unbekannte man aber der Mutation, Rekombination und Selektion unterzieht, desto aufwendiger werden die Berechnungen in jeder Generation und desto mehr Generationen sind auch nötig, weil der Suchraum eine höhere Dimension erhält. Insbesondere das Optimieren der Netzwerkarchitektur kann leicht den Rahmen des Machbaren sprengen, weil Netze die Tendenz haben, eher komplizierter zu werden als einfacher, wenn man

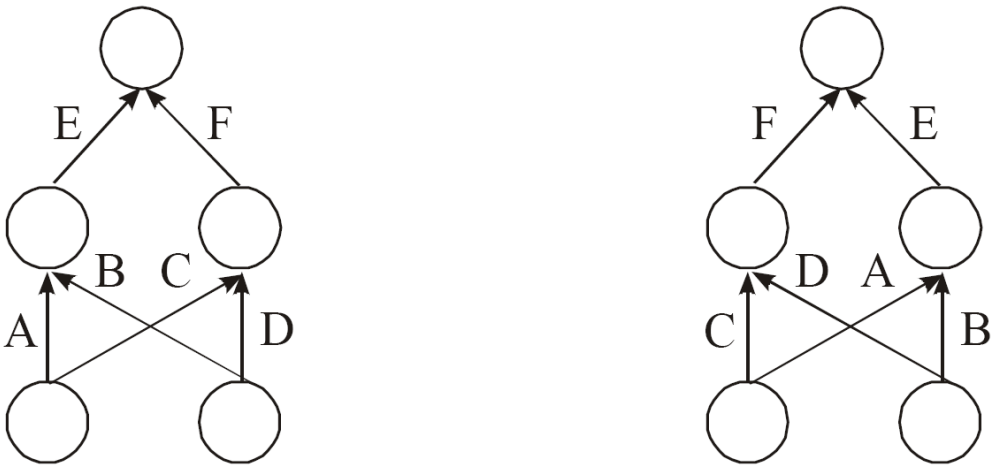
sie sich frei verändern lässt. Auch moderne Hardware stößt dabei an ihre Grenzen: Forster [4] schreibt von 9000 Netzwerkarchitekturen, die pro Jahr (!) getestet werden können.

4.3.2 Konvergenzproblem

Der entscheidende Vorteil genetischer Algorithmen, der den erhöhten Aufwand rechtfertigt, ist die Vielfalt an Lösungsstrategien, die gleichzeitig entwickelt und miteinander kombiniert werden können. Wenn die Individuen einer Population nach wenigen Generationen sehr ähnlich oder gar identisch sind, ist dieser Vorteil dahin. Dazu kann es kommen, wenn sich Individuen mit hoher aber keineswegs optimaler Fitness zu schnell verbreiten, die Population dominieren und andere vielversprechende Individuen aussterben. Das System sollte auch Individuen mit geringerer Fitness genug Zeit zur Weiterentwicklung lassen.

4.3.3 Permutationsproblem

Man vertausche zwei Neuronen inklusive aller Gewichte der Verbindungen, die zu ihnen hin oder von ihnen weg führen. Es ist klar, dass das neuronale Netz genau dasselbe Verhalten zeigt wie vor der Vertauschung, obwohl die Gene verschieden sind. Im folgenden Beispiel haben die beiden Neuronen in der Mitte die Plätze getauscht:



Quelle: Forster [4]

Bedenkt man die normalerweise sehr große Anzahl von Neuronen, erkennt man, dass es zu jedem einzigartigen Netz immer sehr viele verschiedene Individuen gibt, die bis auf Umnummerierungen alle dieses eine Netz repräsentieren.

Das klingt zunächst nicht besorgniserregend, verschärft jedoch beide bereits erwähnten Probleme: Die ohnehin schon große Komplexität wird noch um ein Vielfaches gesteigert, weil man es bei angenommen 1000 verschiedenen Netzen z. B. mit einer Million Individuen zu tun bekommt. Auch die vorzeitige Konvergenz wird begünstigt, weil es zu jedem Netz nicht nur ein Individuum gibt, das im Alleingang die Konkurrenzindividuen verdrängen muss, sondern es kann in der Population mehrere verschiedene Individuen zu diesem Netz geben, die miteinander durch Rekombination ständig Nachkommen produzieren, die wiederum dasselbe Netz repräsentieren. Man erhält eine scheinbar vielfältige Population, aber in Wirklichkeit nur ein Netz, in dem die Neuronen mit viel Aufwand ständig umnummeriert werden.

4.4 Lösungen

4.4.1 Möglichkeiten zur Verringerung der Laufzeit

Selbst, wenn ein neuronales Netz zur Erfüllung seiner Aufgabe ein bestimmtes Neuron oder sogar eine Gruppe von Neuronen nicht benötigt, ist es extrem unwahrscheinlich, dass zufällige Mutationen die Gewichte aller Verbindungen, die zu diesen überflüssigen Neuronen hin oder von ihnen weg führen, auf 0 setzt, was de facto einer Löschung gleichkommt. Daher kann man zusätzlich zur Mutation eine so genannte Deletion durchführen, die ab und zu ein zufällig ausgewähltes Neuron und alle seine Verbindungen löscht. Häufig wird dieses „verstümmelte“ neuronale Netz dann aussterben, aber zumindest ermöglicht man dem System, auszuprobieren, ob die gestellte Aufgabe nicht vielleicht auch mit weniger Neuronen bewältigt werden kann. Die Deletion spielt also quasi die Rolle von Ockhams Rasiermesser (siehe Ertel [2]). Durch die Deletion kann die Anzahl der Neuronen und damit die Laufzeit des Algorithmus unter Umständen erheblich reduziert werden.

Eine ganz andere Möglichkeit, die Komplexität der neuronalen Netze in Grenzen zu halten, sind Penalties. Das sind Terme, die der Fitnessfunktion hinzugefügt werden und übertrieben komplizierte Netze benachteiligen sollen. Denkbar sind Penalties, die von der Anzahl der Neuronen, der Anzahl der Schichten bei Feed-Forward-Netzen oder einfach der verbrauchten Rechenzeit abhängen. Dadurch wird erreicht, dass das System automatisch nur dann erhöhte Komplexität zulässt, wenn eine entsprechende Steigerung der Fitness dies rechtfertigt.

Besonders reizvoll sind Strategien, mit denen sich die Laufzeit verringern lässt, ohne das neuronale Netz verkleinern zu müssen. Man kann z. B. zusätzlich zur Netzwerkarchitektur auch die Gewichte der Eltern an die Nachkommen vererben. Diese durchlaufen zwar dann ihren eigenen Lernprozess, der aber dank der guten Startwerte von den Eltern erheblich kürzer dauert. Im Grunde bedeutet das, dass Nachkommen nicht alles selbst neu lernen müssen, sondern auch Wissen von den Eltern erben. Dass etwas Derartiges möglich ist, hat Lamarck im 19. Jahrhundert behauptet, weshalb diese Vorgangsweise Lamarckismus genannt wird. In der Naturwissenschaft wurde der Lamarckismus widerlegt, aber in den neuronalen Netzen findet er sich bis heute.

4.4.2 Möglichkeiten zur Verlangsamung der Konvergenz

Der Lamarckismus hat einen großen Nachteil: Die Population konvergiert oft rasch gegen ein lokales Optimum, weil durch die von den Eltern übernommenen Startwerte die Nachkommen eher im lokalen Optimum hängenbleiben. Im Gegensatz dazu kann man die Startwerte der Gewichte bei den Nachkommen zufällig generieren anstatt sie von den Eltern zu übernehmen. Im Grunde bedeutet das, dass nur die Lernfähigkeit vererbt wird, die Lerninhalte muss sich jedes Individuum selbst erarbeiten. Da das einst die Theorie von Baldwin war, wird diese Vorgangsweise nach ihm benannt. Sie entspricht erstens den natürlichen Vorgängen und hat zweitens eine weitaus größere Wahrscheinlichkeit, das globale Optimum zu finden, als der Lamarckismus.

Eine andere Möglichkeit, die die Laufzeit des Algorithmus nicht so stark verlängert, ist die Einstellung der Evolutionsparameter. Beispielsweise kann man häufiger Mutationen durchführen, um mehr Vielfalt in die Population zu bekommen. Oder man kann die Selektion abschwächen, indem man weniger Nachkommen produziert, von denen dann ein größerer Anteil überlebt. Beides sollte man jedoch keinesfalls übertreiben, da der genetische Algorithmus sonst immer mehr zu einem banalen Suchalgorithmus wird, der wahllos und zufällig Konfigurationen durchprobiert, was ja nicht Sinn der Sache ist.

Man kann die Individuen auch „altern“ lassen, d. h. bei Individuen, die bereits sehr lange existieren, die Fitness automatisch verringern. Das ist ein Weg, um hartnäckige lokale Optima wieder loszuwerden. Wichtig ist, dass das Altern wie im richtigen Leben nicht sofort einsetzt, sondern erst nach einer gewissen Zeit, damit vorteilhafte Individuen ausreichend Gelegenheit haben, sich fortzupflanzen.

Zusammenfassend lässt sich sagen, dass zwar zahlreiche Probleme bei der Zusammenarbeit von genetischen Algorithmen und neuronalen Netzen auftreten, dass es aber auch bereits viele erfolgversprechende Lösungsmethoden für diese Probleme gibt. Weitere Probleme und Lösungsmethoden sind bei Forster [4] zu finden.

Kapitel 5

Anwendung: Zeitreihenanalyse

5.1 Was sind Zeitreihen?

Eine Zeitreihe ist eine Folge von Zufallsvariablen $X_1, X_2, ..., X_T$. Jede dieser Zufallsvariablen repräsentiert den momentanen Wert einer Größe zu einem der Zeitpunkte $t = 1, 2, ..., T$.

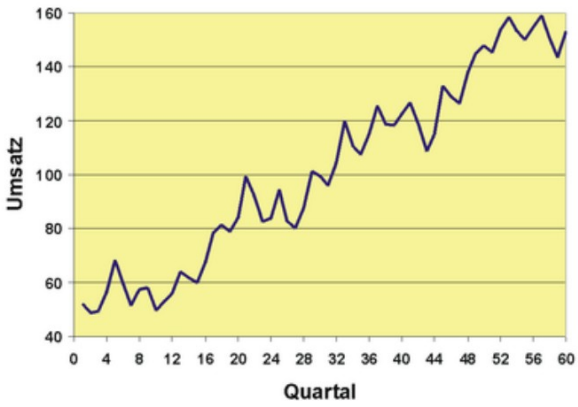
Es gibt viele Beispiele für Größen, die nicht nur einmal, sondern immer wieder gemessen werden und daher üblicherweise in Form von Zeitreihen vorliegen. In der Meteorologie z. B. Temperatur, Windgeschwindigkeit oder Niederschlagsmenge. In der Wirtschaft Börsenkurse, Bruttosozialprodukt oder Arbeitslosenzahlen. In der Medizin Blutdruck, Herz- oder Atemfrequenz etc.

Alle diese Beispiele haben eines gemeinsam: Die Zeitreihen entstehen durch so genannte Abtastung. Das bedeutet, dass die betroffenen Größen zu meist äquidistanten Zeitpunkten gemessen werden. Zu welchen, ist im Grunde eine willkürliche Entscheidung. Die Größen könnten prinzipiell auch zu Zeitpunkten dazwischen gemessen werden.

Das erste Ziel der Zeitreihenanalyse besteht darin, die verschiedenen Ursachen für Veränderungen in der Zeitreihe zu identifizieren und zu quantifizieren. Danach können störende Einflüsse aus der Zeitreihe „herausgerechnet“ werden, wie etwa die saisonale Schwankung bei den Arbeitslosenzahlen, weil es im Winter generell mehr Arbeitslose als im Sommer gibt. Die weitaus spannendere Anwendung der Zeitreihenanalyse ist jedoch die Prognose, also der Versuch, aus dem bisherigen Verlauf der Zeitreihe auf den zukünftigen zu schließen. Es geht um nichts weniger, als quasi in die Zukunft sehen zu können, zumindest bis zu einem gewissen Grad.

5.2 Statistische Zeitreihenanalyse

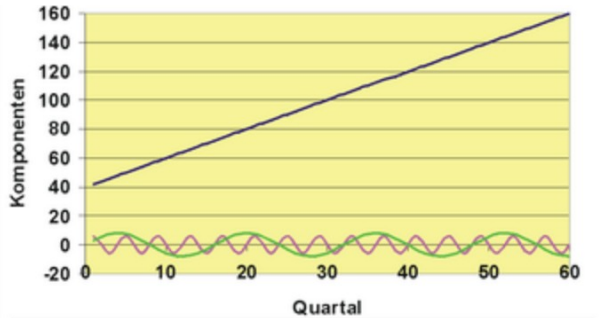
Gehen wir davon aus, dass jene Zeitreihe analysiert werden soll, die der folgende Graph veranschaulicht:



Quelle: Wikipedia [8]

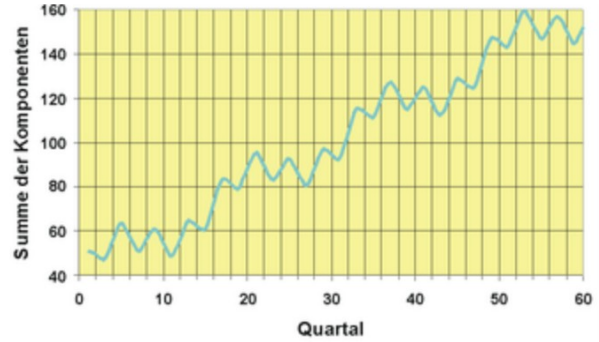
Man kann gut einen allgemeinen annähernd linearen Aufwärtstrend erkennen. Es gibt relativ gleichmäßige Oszillationen um diesen Aufwärtstrend herum.

Hier bietet sich das so genannte additive Modell an. Das bedeutet, dass man sich die Zeitreihen als Summe mehrerer unabhängiger Zeitreihen vorstellt. Mithilfe statistischer Verfahren werden diese einzelnen Zeitreihen geschätzt, in unserem Fall der Aufwärtstrend als Gerade und die Oszillationen als Sinus- bzw. Kosinusfunktionen:



Quelle: Wikipedia [8]

Addiert man die einzelnen Komponenten wieder, sollte im Idealfall eine Zeitreihe herauskommen, die der ursprünglichen recht ähnlich ist:



Quelle: Wikipedia [8]

Ein sehr einfaches additives Modell ist folgendes (siehe Wikipedia [9]):

$$X_t = Q_t + S_t + R_t$$

t: Zeitpunkt innerhalb der Zeitreihen

X_t: Originalzeitreihe

Q_t: Trendkomponente

S_t: Saisonkomponente

R_t: Restschwankung

Wenn man z. B. von einem linearen Trend ausgeht, dann soll Q_t linear in t sein und kann durch die wohlbekannte lineare Regression geschätzt werden:

$$Q_t = \hat{a} + \hat{b} \cdot t$$

$$\hat{b} := \frac{\sum_{t=1}^{T} (t - \bar{t})(X_t - \bar{X})}{\sum_{t=1}^{T} (t - \bar{t})^2} \qquad \hat{a} := \bar{X} - b \cdot \bar{t}$$

$$\bar{X} := \frac{1}{T} \cdot \sum_{t=1}^{T} X_t \qquad \bar{t} := \frac{1}{T} \cdot \sum_{t=1}^{T} t = \frac{1}{T} \cdot \frac{T(T+1)}{2} = \frac{T+1}{2}$$

Selbstverständlich sind auch nichtlineare Trends möglich. Steigt die Größe G pro Zeiteinheit um einen bestimmten Prozentsatz p, dann gehorcht sie der Formel:

$$G_t = G_0 \cdot \left(1 + \frac{p}{100}\right)^t = G_0 \cdot e^{t \cdot \ln\left(1 + \frac{p}{100}\right)}$$

Dann hat man also einen Trend in Form einer Exponentialfunktion. Angenehmerweise muss man zur Schätzung dieses Trends nicht völlig neue Regressionsformeln herleiten, sondern betrachtet einfach die logarithmierte Zeitreihe

$$\ln G_t = \ln G_0 + t \cdot \ln\left(1 + \frac{p}{100}\right)$$

Wir haben also einen Ausdruck erhalten, der linear in t ist und daher wiederum mit der linearen Regression geschätzt werden kann.

Sobald man den Trend geschätzt hat, wird er von der Originalzeitreihe abgezogen, um die so genannte trendbereinigte Zeitreihe zu erhalten:

$$X_t - Q_t = S_t + R_t$$

Wie schätzt man nun die Saisonkomponente, also z. B. den Einfluss der Jahreszeit auf die Arbeitslosenzahlen? Eine Möglichkeit ist, sie als Sinussschwingung anzusetzen und möglichst gut an die trendbereinigte Zeitreihe anzupassen. Es steht jedoch nirgends geschrieben, dass sich der saisonale Einfluss wie eine Sinusfunktion verhalten muss. Daher wird häufig einfach das arithmetische Mittel aller Jänner-, Februar-, …, Dezember-Werte als Saisonkomponente genommen. Das, was übrig bleibt, also weder durch Trend noch durch Saison erklärt werden kann, wird als Restschwankung bezeichnet.

Selbstverständlich kann man so nicht nur den Einfluss von Saisonen im Jahreszyklus, sondern auch im Monats-, Wochen- oder gar Tageszyklus berücksichtigen, indem man dem additiven Modell weitere Summanden hinzufügt.

Obwohl es in der Statistik noch zahlreiche weitere viel raffiniertere Methoden zur Zeitreihenanalyse gibt, ein Problem haben sie alle gemeinsam: Man muss sich für ein bestimmtes Modell entscheiden, das nur die Einflüsse korrekt berücksichtigt, die im Modell vorgesehen sind. Man kann sich jedoch nie sicher sein, ob nicht andere wichtige Einflüsse übersehen werden. Beispielsweise ein Konjunkturzyklus, der sich über mehrere Jahre erstreckt, oder Termine von Wahlen, Notenbanksitzungen und dergleichen.

Genetische Algorithmen und neuronale Netze hingegen verarbeiten die Daten quasi unvoreingenommen. Sie versuchen, Muster darin zu finden und zu verallgemeinern, ohne dass man sich vorher festlegen muss, welcher Art diese Muster sein könnten. Es besteht die Möglichkeit, dass das optimale neuronale Netz einen sehr ungewöhnlichen Zusammenhang entdeckt, auf den ein Statistiker, der sich ein Modell überlegt, nie gekommen wäre.

5.3 Das Programm

5.3.1 Wahl des Programms

Es wäre ein großer Aufwand, für die Zeitreihenanalyse alle bereits erwähnten Prinzipien wie Mutation, Rekombination, Selektion und vor allem die Backpropagation eigenhändig zu programmieren. Das würde den Rahmen dieser Arbeit zweifellos sprengen. Glücklicherweise muss das Rad nicht ständig neu erfunden werden, im Zeitalter des Internets sind unzählige mehr oder weniger leistungsfähige Programme dafür verfügbar. Verlangt man, dass ein und dasselbe Programm sowohl neuronale Netze als auch genetische Algorithmen einsetzt, wird die Auswahl bereits merklich geringer. Das Programm, das den bisherigen Ausführungen dieser Arbeit am nächsten kommt, wurde von Kluge und Förster an der Hochschule für Technik und Wirtschaft Dresden entwickelt [7]. Es soll im Folgenden kurz vorgestellt werden.

5.3.2 Schritt 1: Netzwerkarchitektur

Implementiert ist ein Feed-Forward-Netz (siehe Kapitel 4.2.2), bei dem die Anzahl der Neuronen in jeder Schicht durch einen genetischen Algorithmus optimiert wird. Dazu ist jedem Neuron einfach ein binäres Gen zugeordnet, das das entsprechende Neuron je nach Bedarf hinzufügen oder entfernen kann.

Die Population besteht aus mehreren neuronalen Netzen mit unterschiedlicher Architektur. Um zu jedem von ihnen die zugehörige Fitness zu ermitteln, wird das neuronale Netz mit den Trainingsdaten trainiert und anschließend mit den Validierungsdaten getestet. Trainingsdaten und Validierungsdaten müssen strikt auseinander gehalten werden, denn es wäre ja keine Kunst, wenn das neuronale Netz jene Daten richtig verarbeitet, mit denen es trainiert wurde. Nein, ein gutes neuronales Netz soll auch mit ihm bislang unbekannten Daten zurechtkommen. Je schneller (d. h. mit weniger Trainingszyklen) ein neuronales Netz den Validierungsfehler reduzieren kann, desto größer soll dessen Fitness sein.

Man beachte, dass nur die Netzwerkarchitektur vererbt wird, nicht aber die Gewichte. Somit handelt es sich nicht um einen Lamarckismus, sondern um ein Baldwin-System (siehe Forster [4]).

Die Selektion erfolgt nach dem so genannten Roulette-Verfahren (siehe auch Zöller-Greer [1]). Hierbei ist die Wahrscheinlichkeit, mit der ein Individuum zur Fortpflanzung herangezogen wird, proportional zu dessen Fitness. Es handelt sich also um Selektion durch Fortpflanzung (siehe Kapitel 2.6.2).

5.3.3 Schritt 2: Initialisierung der Gewichte

Dies erfolgt in einem separaten genetischen Algorithmus. Hierzu wird das optimale Netz aus Schritt 1 mit zufälligen Gewichten initialisiert. Das geschieht mehrmals, um eine Population solcher Netze zu erhalten.

Anschließend werden wie in Schritt 1 nach dem Roulette-Verfahren jene Netze selektiert, die einen möglichst geringen Validierungsfehler haben. Der Unterschied zu Schritt 1 besteht darin, dass diesmal kein Training erfolgt. Denn wie in Kapitel 4.4.2 besprochen führt das Trainieren mit vererbten Startwerten für die Gewichte häufig dazu, dass die Individuen in lokalen Optima hängenbleiben. Deshalb werden in diesem Schritt nur die optimalen Startwerte bestimmt, das Training erfolgt separat im nächsten Schritt.

5.3.4 Schritt 3: Training

Das neuronale Netz aus Schritt 1 mit den Startwerten für die Gewichte aus Schritt 2 wird mittels Backpropagation trainiert. Wie üblich ist die Aktivierungsfunktion nichtlinear, um dem Netz nichtlineares Verhalten zu ermöglichen. Verwendet wird die so genannte logistische Funktion:

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

Das ist nichts anderes als die Fermi-Funktion aus Kapitel 3.7, wenn man $\theta = 0$ wählt.

5.4 Die Bedienung des Programms

5.4.1 Vorbereitungen

Das Programm sowie speziell dafür aufbereitete Trainings- und Validierungsdaten können unter folgenden Links heruntergeladen werden:

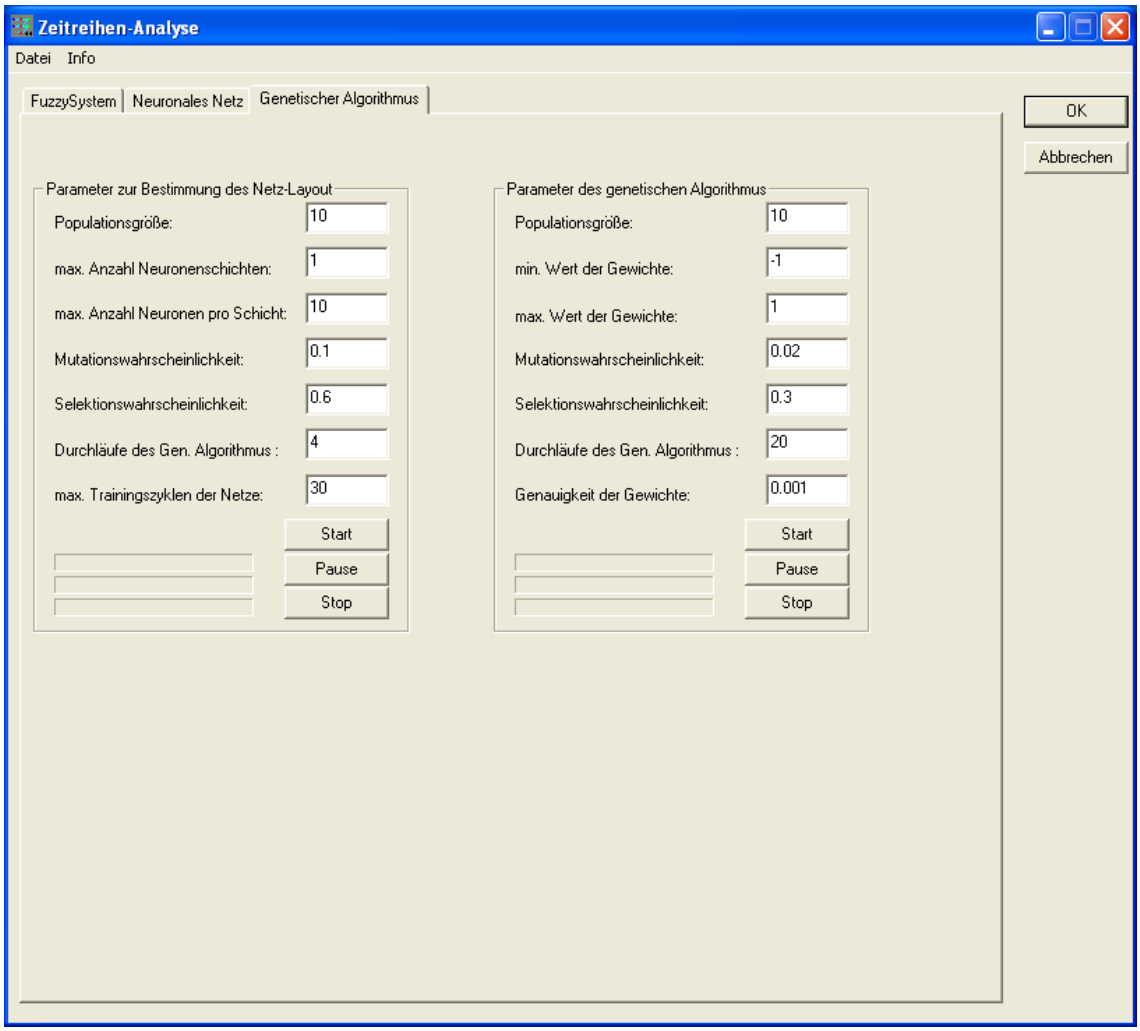
http://www.htw-dresden.de/%7Eiwe/Belege/Foerster/download/NeuroGui.zip
http://www.htw-dresden.de/%7Eiwe/Belege/Foerster/download/Projects.zip

Die Benutzung des Programms und der Daten ist gestattet, solange sie für wissenschaftliche Zwecke verwendet werden und die Autoren Kluge und Förster von der Hochschule für Technik und Wirtschaft Dresden erwähnt werden. Für kommerzielle Zwecke hingegen ist gemäß Urheberrecht das Einverständnis der Autoren erforderlich.

Nachdem Programm und Daten entpackt wurden, kann das Programm ohne Installation sofort gestartet werden. Zunächst ist im Menü „Datei“ das Projekt zu öffnen. Hier sind anfänglich alle Felder leer. Drückt man den Knopf rechts oben mit den Punkten, kann man die mitgelieferten Daten importieren. Nun werden stets drei Fehlermeldungen angezeigt. Das ist jedoch kein Grund zur Besorgnis, es wird nur das Fehlen von Dateien gemeldet, die erst zu einem späteren Zeitpunkt angelegt werden. Nun ist das Programm einsatzbereit.

5.4.2 Schritt 1: Netzwerkarchitektur

Man wählt den Reiter „Genetischer Algorithmus“ aus und muss zunächst die Parameter in der linken Hälfte wählen:



Populationsgröße

Voreingestellt sind $n = 10$ Individuen. Laut Heistermann [3] sind das allerdings zu wenige. Tests werden zeigen, ob größere n das Resultat verbessern können.

Maximale Anzahl von Neuronenschichten

Voreingestellt ist lediglich eine Schicht. Die Autoren begründen das damit, dass die Anzahl der Schichten den Trainingsaufwand rapide erhöht. Tests werden auch das nachprüfen.

Maximale Anzahl Neuronen pro Schicht

Hier ist ebenfalls 10 voreingestellt. Die Autoren begründen das damit, dass das Optimum häufig bei 8 Neuronen lag und das Maximum ein wenig höher sein sollte, da es sehr unwahrscheinlich ist, dass ein genetischer Algorithmus das Maximum genau erreicht.

Mutationswahrscheinlichkeit

Voreingestellt ist, dass es mit Wahrscheinlichkeit 0.1 zu einer Mutation kommt.

Selektionswahrscheinlichkeit

Der Wert von 0.6 bedeutet, dass von den Individuen, die zur Fortpflanzung verwendet werden, 60 % mit Selektion durch das Rouletteverfahren und 40 % ohne Selektion (also zufällig) ausgewählt werden.

Durchläufe des genetischen Algorithmus

Hiermit ist nichts anderes als die Anzahl der Generationen gemeint. Wieder ist der voreingestellte Wert 4 gemäß Heistermann [3] viel zu niedrig und wird im Rahmen von Tests erhöht werden.

Maximale Anzahl von Trainingszyklen der Netze

Voreingestellt ist, dass die Trainingsdaten dem Netz höchstens 30 mal vorgelegt werden, bevor die Fitness des Netzes bewertet wird.

Hat man alle Parameter nach Wunsch eingestellt, drückt man auf den Knopf „Start“ und wartet, bis die Anzeigen sich nicht mehr ändern. Dann hat der genetische Algorithmus sich für eine Netzwerkarchitektur entschieden.

5.4.3 Schritt 2: Initialisierung der Gewichte

Nun sind die Parameter in der rechten Hälfte an der Reihe. Populationsgröße, Mutations- und Selektionswahrscheinlichkeit sowie Durchläufe haben dieselbe Bedeutung wie in der linken Hälfte.

Minimaler und maximaler Wert der Gewichte

Laut Voreinstellung sollen die Gewichte aus dem Intervall $[-1,1]$ stammen. Negative Werte sind nötig, um Neuronen zu simulieren, die Impulse anderer Neuronen hemmen (siehe Zöller-Greer [1]). Unter diesem Gesichtspunkt scheint ein anderes Intervall als $[-1,1]$ wenig sinnvoll.

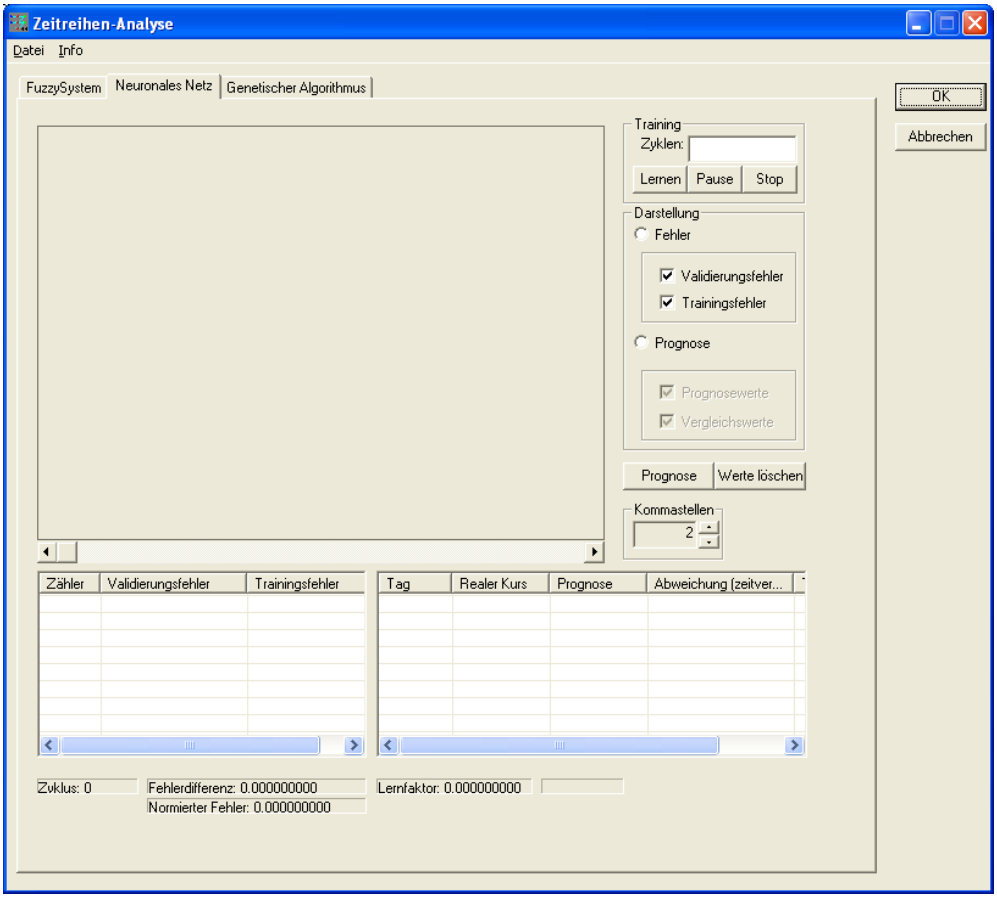
Genauigkeit der Gewichte

Die Gewichte können nur Vielfache dieses Parameters sein, die Voreinstellung beträgt 0.001. Laut den Autoren sollte dieser Wert nicht gesenkt werden. Das ist

nachvollziehbar, da der Qualitätsunterschied zu Netzen mit einer feineren Genauigkeit kaum von Bedeutung sein dürfte.

Nach dem Einstellen all dieser Parameter ist erneut der Knopf „Start“ zu drücken, die Frage, ob das Netz aus Schritt 1 verwendet werden soll, mit Ja zu beantworten und abzuwarten, bis die Anzeigen sich nicht mehr verändern.

5.4.4 Schritt 3: Training



Wählt man den Reiter „Neuronales Netz“, sieht man obigen Dialog. In das Feld „Zyklen“ ist einzugeben, wie oft dem Netz die Trainingsdaten vorgelegt werden sollen. Hier ist kein Wert vorgegeben. Für größere Netze empfehlen die Autoren

100 Zyklen. Wie wir noch sehen werden, ist diese Eingabe nicht kritisch, weil der Lernfortschritt graphisch dargestellt wird und man deshalb gut abschätzen kann, ob eine Erhöhung der Zyklen noch etwas bringt oder nicht. Um diese graphische Darstellung auch sehen zu können, ist das Kästchen neben „Fehler“ zu aktivieren. Das Training beginnt, wenn der Knopf „Lernen“ gedrückt wird. Wieder ist abzuwarten, bis sich die Anzeigen nicht mehr ändern. Der letzte Schritt wäre die Interpretation der Ergebnisse, doch dazu später mehr, wenn uns bei den Tests konkrete Ergebnisse vorliegen.

5.5 Tests

5.5.1 Was wird getestet?

In Kapitel 5.1 wurden einige Beispiele für Zeitreihen erwähnt, die alle mit diesem Programm analysiert werden könnten. Gehen wir einmal optimistischerweise davon aus, dass die verwendeten Algorithmen erfolgreich sind und tatsächlich zu aussagekräftigen Prognosen führen. Bei Zeitreihen meteorologischer Daten verschafft uns das nicht mehr als eine Wetterprognose, die wohl kaum besser sein dürfte als die erfahrener Meteorologen.

Reizvoller ist die Analyse von Zeitreihen, die den Ruf haben, schwer bis gar nicht prognostizierbar zu sein, wie etwa Börsenkurse. Dort gibt es zwar auch eine nicht enden wollende Fülle von Prognoseinstrumenten. Einerseits so genannte fundamentale, die die Wirtschaftsdaten der Unternehmen betrachten, andererseits technische, die die Charts der Aktien unter die Lupe nehmen. Wer sich schon einmal hoffnungsvoll mit diesen so genannten Indikatoren beschäftigt hat, wird

bald die Erfahrung gemacht haben, dass es für jeden von ihnen zahlreiche historische Beispiele gibt, wo sie den weiteren Verlauf richtig prognostizierten, aber ausgerechnet dann, wenn man selbst auf sie vertraut, irren sie sich. Die klassische Antwort darauf lautet, mehrere dieser Indikatoren zu kombinieren. Aber auch da hält die Gegenwart meist nicht, was die Vergangenheit verspricht. Der Grund ist einfach: Da die handelnden Menschen ständig ihr Marktverhalten ändern, wandelt sich das „Verhalten“ der Börse auch ununterbrochen. Eine einfache Strategie, nennen wir sie Strategie A, die ständig Gewinne erzeugt, kann nicht lange Bestand haben, weil es nur eine Frage der Zeit ist, bis andere Marktteilnehmer die Strategie A ebenfalls entdecken, mit denen der Gewinn dann geteilt werden muss. Wenn schließlich zu viele Marktteilnehmer die Strategie A anwenden, werden früher oder später andere eine Strategie B entwickeln, die von dem vorhersehbaren Verhalten der Anhänger der Strategie A profitiert. Da an der Börse kein Geld gedruckt wird, sondern es nur den Besitzer wechselt, sind die Gewinne von B zwangsläufig die Verluste von A. Unglücklicherweise gilt für B genau dasselbe: Irgendwann wird es wohl eine Strategie C geben, die auf Kosten von B Gewinne lukriert usw.

Ein mithilfe genetischer Algorithmen optimiertes neuronales Netz, das die Zukunft eines Aktienkurses prognostizieren kann, wäre wohl der Traum jedes Börsianers, selbst wenn die Prognose nur kurzfristig ist und nicht immer, sondern nur mit einer bestimmten Wahrscheinlichkeit zutrifft. Die nun folgenden Tests sollen überprüfen, ob dieses ehrgeizige Ziel durch das Programm von Kluge und Förster zumindest ansatzweise erreicht werden kann.

Die Programmautoren empfehlen, nur Aktienkurse von Unternehmen zu verwenden, von denen ausreichend Datenmaterial zur Verfügung steht. Außerdem ist es wohl eine gute Idee, eher große Unternehmen zu wählen, da die Liquidität in den zugehörigen Aktienmärkten größer ist, weshalb man die Aktien

jederzeit problemlos kaufen und verkaufen kann. Bei Aktien kleiner Unternehmen ist das nicht selbstverständlich. Daher werden im Folgenden die Aktienkurse der Deutschen Bank analysiert.

Das Programm verwendet jeweils fünf aufeinander folgende Tagesschlusskurse als Eingangsdaten für das neuronale Netz und strebt als Ausgabe eine Prognose für den sechsten Tagesschlusskurs an. Die Differenz zwischen Prognose und tatsächlichem Kurs ist der zu minimierende Fehler.

Die zur Verfügung stehenden Daten werden in 90 % Trainingsdaten und 10 % Validierungsdaten aufgeteilt (siehe Kapitel 5.3.2). Diese Aufteilung erfolgt zufällig.

5.5.2 Der erste Test

Beim ersten Test lassen wir alle von den Programmautoren vorgegebenen Werte unverändert. Jene Parameter, die über die Laufzeit entscheiden, wurden hierbei vorsichtig niedrig eingestellt (siehe Kapitel 5.4.2). Es sind also schnelle aber suboptimale Ergebnisse zu erwarten. Das ist für den ersten Test nicht unklug, denn für den Anwender eines Programms ist es frustrierend, wenn das Programm lange läuft und er sich ständig fragt, ob er es abbrechen oder noch länger auf das Ergebnis warten soll.

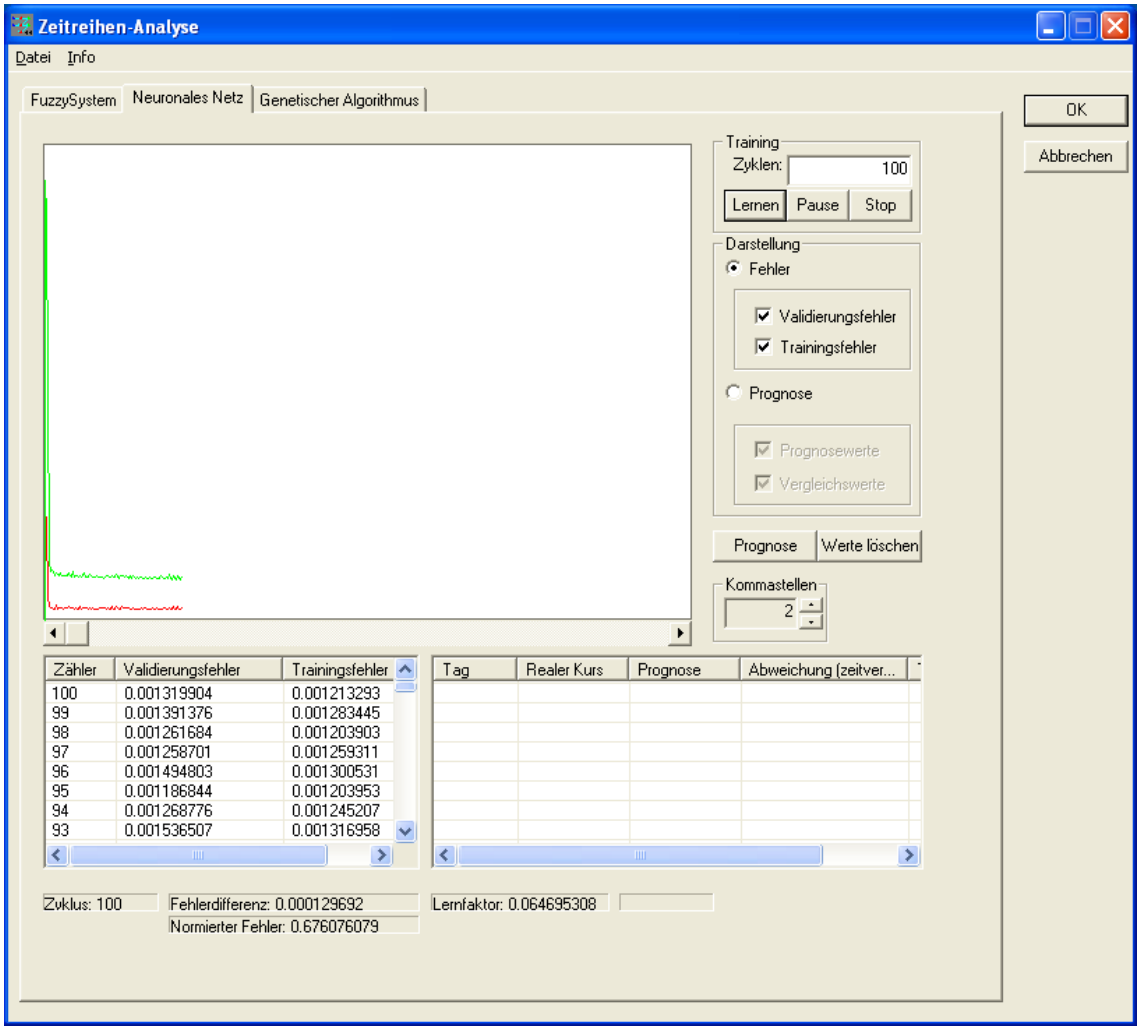
Der einzige Wert, für den es keine Voreinstellung gibt, ist die Anzahl der Trainingszyklen. Hier wird der Wert 100 verwendet (siehe Kapitel 5.4.4).

Hier die Laufzeiten der drei Schritte des Programms in Sekunden auf einem PC mit einer Taktfrequenz von 2,67 GHz:

Schritt	Programmteil	Laufzeit
1	Genetischer Algorithmus für die Netzwerkarchitektur	2 s
2	Genetischer Algorithmus zur Initialisierung der Gewichte	2 s
3	Training des neuronalen Netzes	2 s
Gesamt		6 s

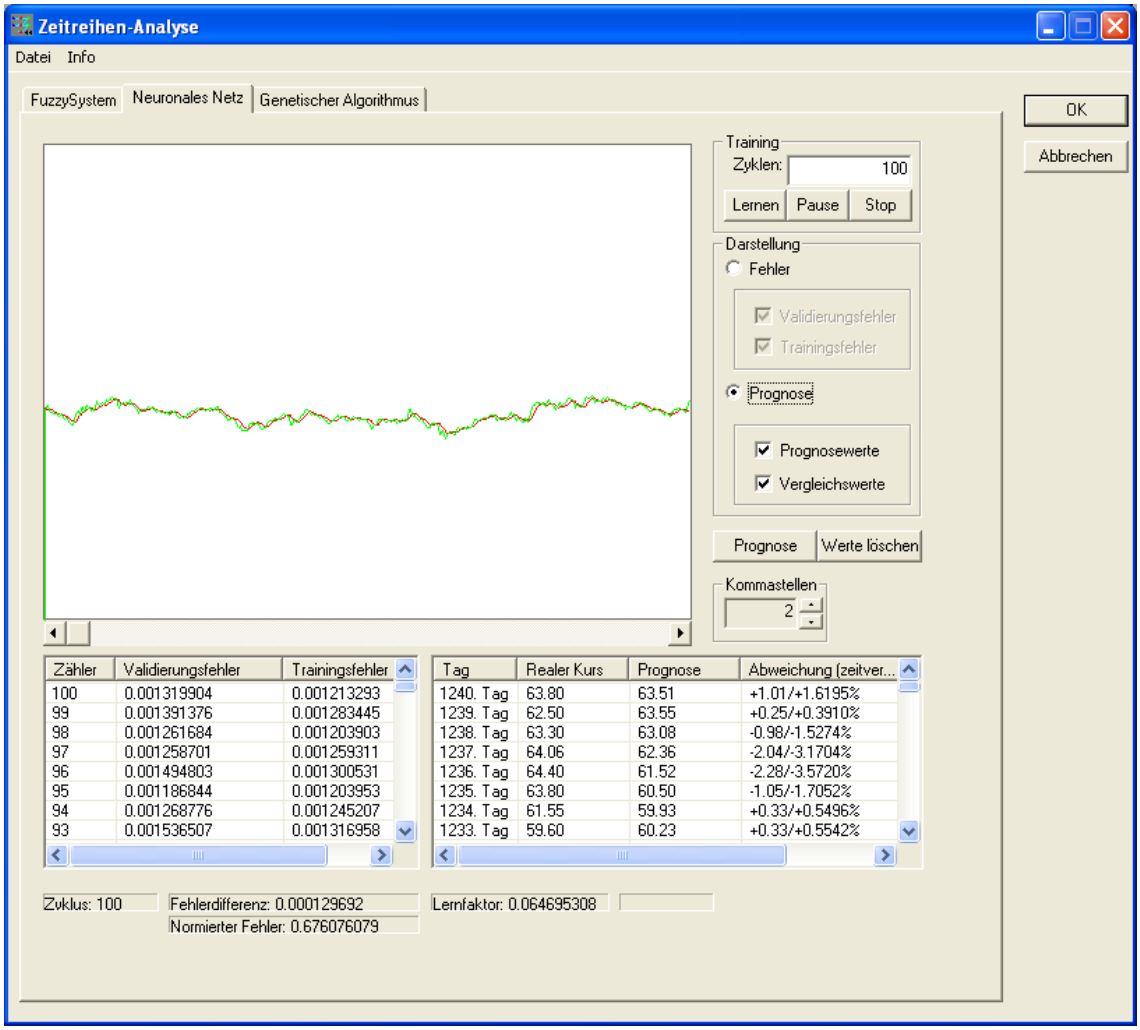
Hier waren die Autoren offenbar übervorsichtig, wenn sie keine zu lange Laufzeit riskieren wollten. Zu ihrer Ehrenrettung muss man jedoch hinzufügen, dass sich heutzutage die Leistungsfähigkeit der Computer mit unglaublichem Tempo vervielfacht. So wird ein Problem, das den Rechner an seine Grenzen führt, innerhalb kürzester Zeit zu einem Problemchen, das schnell nebenbei erledigt werden kann.

Wenn auf dem Reiter „Neuronales Netz“ das Kästchen „Fehler“ ausgewählt war, wird am Ende folgende Grafik angezeigt:



Die beiden Kurven stellen den Validierungsfehler und den Trainingsfehler dar. Sie sinken in den ersten Trainingszyklen sehr rasch und bleiben dann in etwa konstant. Wir wären also auch mit weit weniger als 100 Zyklen ausgekommen.

Doch nun zu dem, was uns wirklich interessiert, der Prognose. Dazu ist zunächst der gleichnamige Knopf zu drücken, was die Tabelle rechts unten mit Werten füllt. Wenn man dann das Kästchen neben „Prognose“ aktiviert, erscheint folgende Grafik:



Es sollten eigentlich zwei Kurven dargestellt sein: Die realen Kurse und deren Prognose durch das neuronale Netz. Da die Prognose hervorragend ist, muss man schon sehr genau hinsehen, um zu bemerken, dass es sich um zwei Kurven und nicht um eine Kurve handelt. Grafische Darstellungen sind jedoch nur bedingt dazu geeignet, den Erfolg eines Tests zu beurteilen bzw. ihn mit dem anderer Tests zu vergleichen.

Wenn man bei der rechten Tabelle ganz nach rechts scrollt, kann man in jeder Zeile das Wort „ja“ oder „nein“ finden. Was hat es damit auf sich? Das Programm stellt fest, ob der reale Kurs an dem entsprechenden Tag gestiegen oder gefallen ist, und, ob die Prognose gestiegen oder gefallen ist. Bei einer Übereinstimmung gibt es ein „ja“, ansonsten ein „nein“. Somit bedeutet jedes „ja“, dass das neuronale Netz am Vortag die Richtung des Kurses korrekt vorhergesehen hat.

Das ist jene Information, die die Börsianer am meisten interessiert. Wenn die Prognose nämlich sagt, dass der Kurs steigen wird, dann sollte man die Aktie kaufen. Wenn die Prognose sagt, dass der Kurs fallen wird, dann sollte man die Aktie verkaufen. Eine Prognose ist also umso wertvoller, je seltener sie sich bei der Richtung irrt.

Die Zahlen am unteren Ende des Fensters sind hingegen für die Praxis weniger relevant. Sie beschreiben nämlich, kurz gesagt, wie groß die Abstände zwischen der Prognose und dem realen Kurs sind. Ein neuronales Netz, das einen Kursanstieg von 1 vorhersagt, obwohl er in Wirklichkeit 3 beträgt, würde wegen der großen Differenz schlecht beurteilt werden. Den Börsianer würde es aber wohl kaum stören, dass er mehr Gewinn macht als erwartet. Umgekehrt hätte er keine Freude, wenn das neuronale Netz einen Kursanstieg von 1 prognostiziert, der Börsianer die Aktie kauft und dann fällt der Kurs um 1. In beiden Beispielen weicht die Prognose um 2 ab, aber das zweite Beispiel ist bei weitem unangenehmer als das erste, weil die Richtung der Kursbewegung falsch prognostiziert wurde.

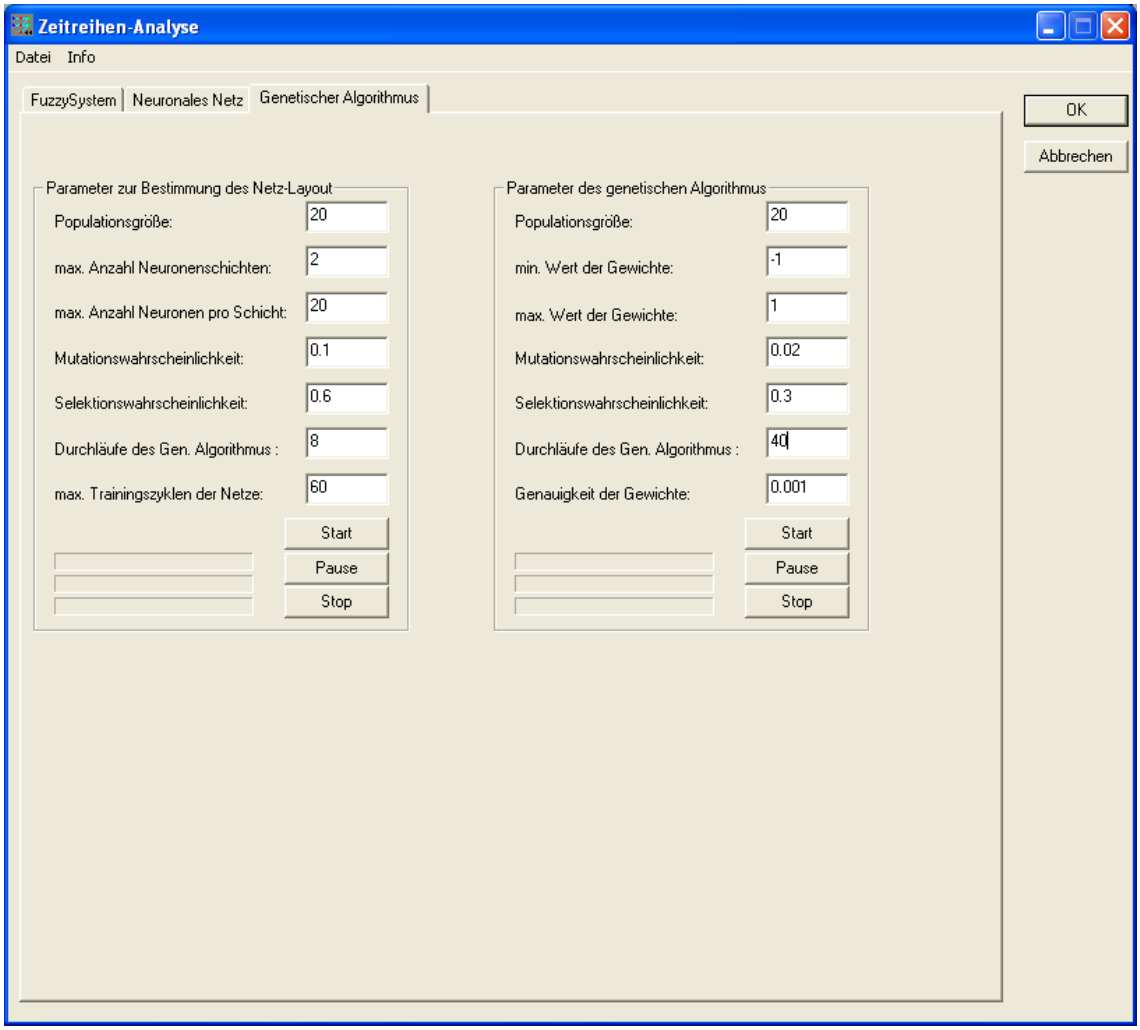
Daher werden wir den Erfolg eines Tests mit dem Prozentsatz an „ja“ also an korrekt vorhergesagten Richtungen der Kursbewegung messen. Für die Aktie der Deutschen Bank liegen die Kurse von 1240 Handelstagen vor. Es gibt jedoch nur 1238 Prognosen, da das Programm an den ersten beiden Tagen mangels Eingangsdaten keine Prognose erstellen kann:

Übereinstimmung	absolute Häufigkeit	relative Häufigkeit
ja	874	71 %
nein	364	29 %
gesamt	1238	100 %

71 % Erfolgswahrscheinlichkeit sind nicht schlecht, wenn man bedenkt, dass man durch „Raten“ nur auf 50 % käme und einige der technischen Indikatoren sich damit rühmen, zwei Drittel also gerundet 67 % zu erreichen.

5.5.3 Der zweite Test

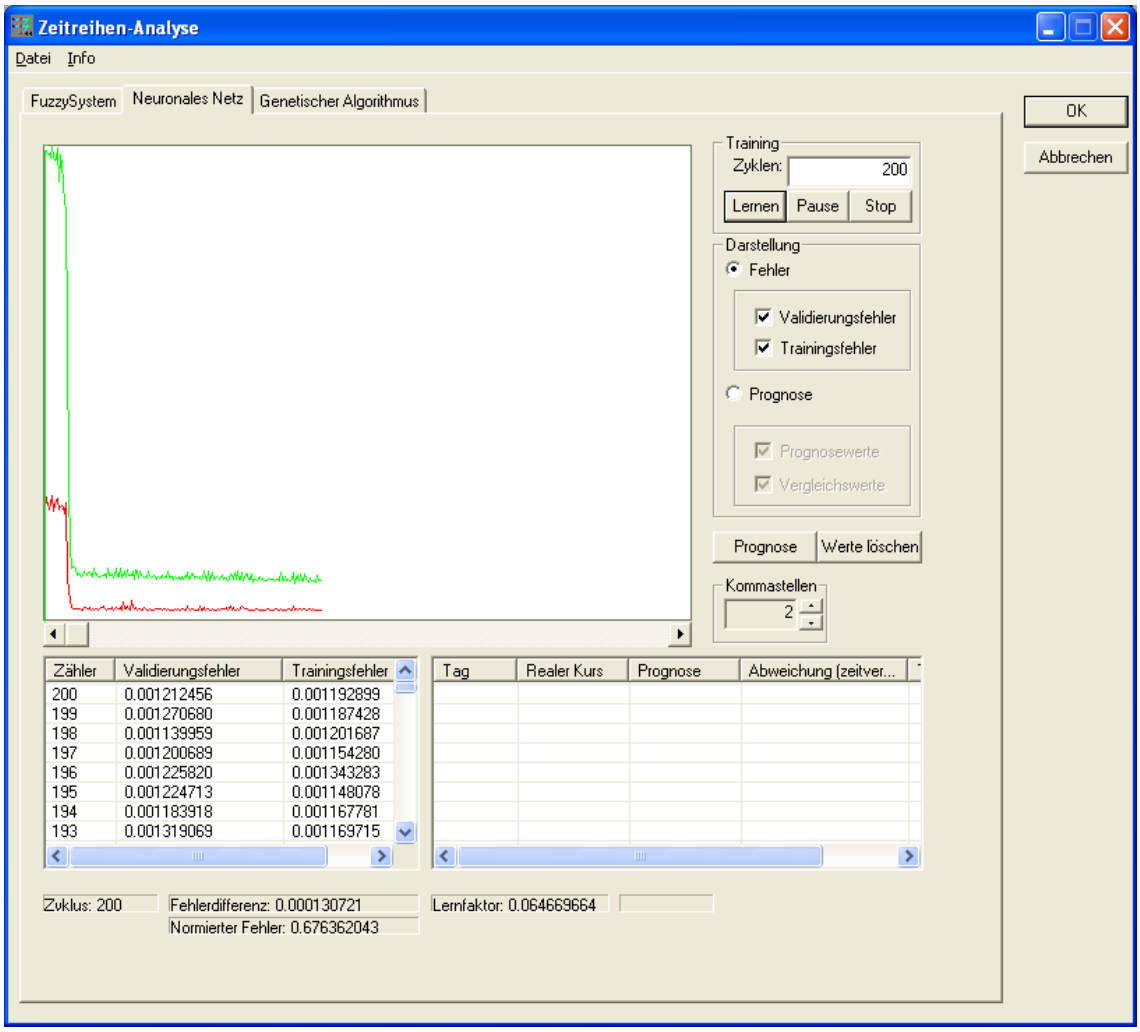
Ein Ergebnis des ersten Tests ist, dass man bei allen Parametern, die die Laufzeit verlängern, viel großzügiger sein kann. Wir können uns also mehr Individuen, Schichten, Neuronen, Generationen und Trainingszyklen leisten. Normalerweise wäre es angebracht, jeden dieser Parameter einzeln zu erhöhen und die Auswirkung auf das Ergebnis und die Laufzeit zu beobachten. Da die Laufzeit des ersten Tests allerdings so extrem kurz war, können wir uns den Luxus leisten, alle diese Parameter einfach zu verdoppeln. Somit geben wir beim zweiten Test folgende Parameter ein:



Die Laufzeiten verlängern sich wie erwartet, sind aber immer noch sehr gering:

Schritt	Programmteil	Laufzeit
1	Genetischer Algorithmus für die Netzwerkarchitektur	22 s
2	Genetischer Algorithmus zur Initialisierung der Gewichte	9 s
3	Training des neuronalen Netzes	4 s
Gesamt		35 s

Während die Grafik, die reale Kurse und Prognose miteinander vergleicht, genau gleich aussieht wie vorhin, gibt es bei der Grafik, die Validierungs- und Trainingsfehler darstellt, eine Veränderung:



Man erkennt, dass die Fehler erst nach etwas mehr Zyklen sinken als vorhin. Das deckt sich mit den Ausführungen der Programmautoren [7], die dringend anraten, bei größeren neuronalen Netzen mehr Trainingszyklen durchzuführen. Gleichzeitig sieht man aber auch, dass eine weitere Erhöhung der Trainingszyklen wohl kaum noch eine Verbesserung bringen würde.

Hat dieser erhöhte Aufwand aber auch ein besseres Ergebnis gebracht?

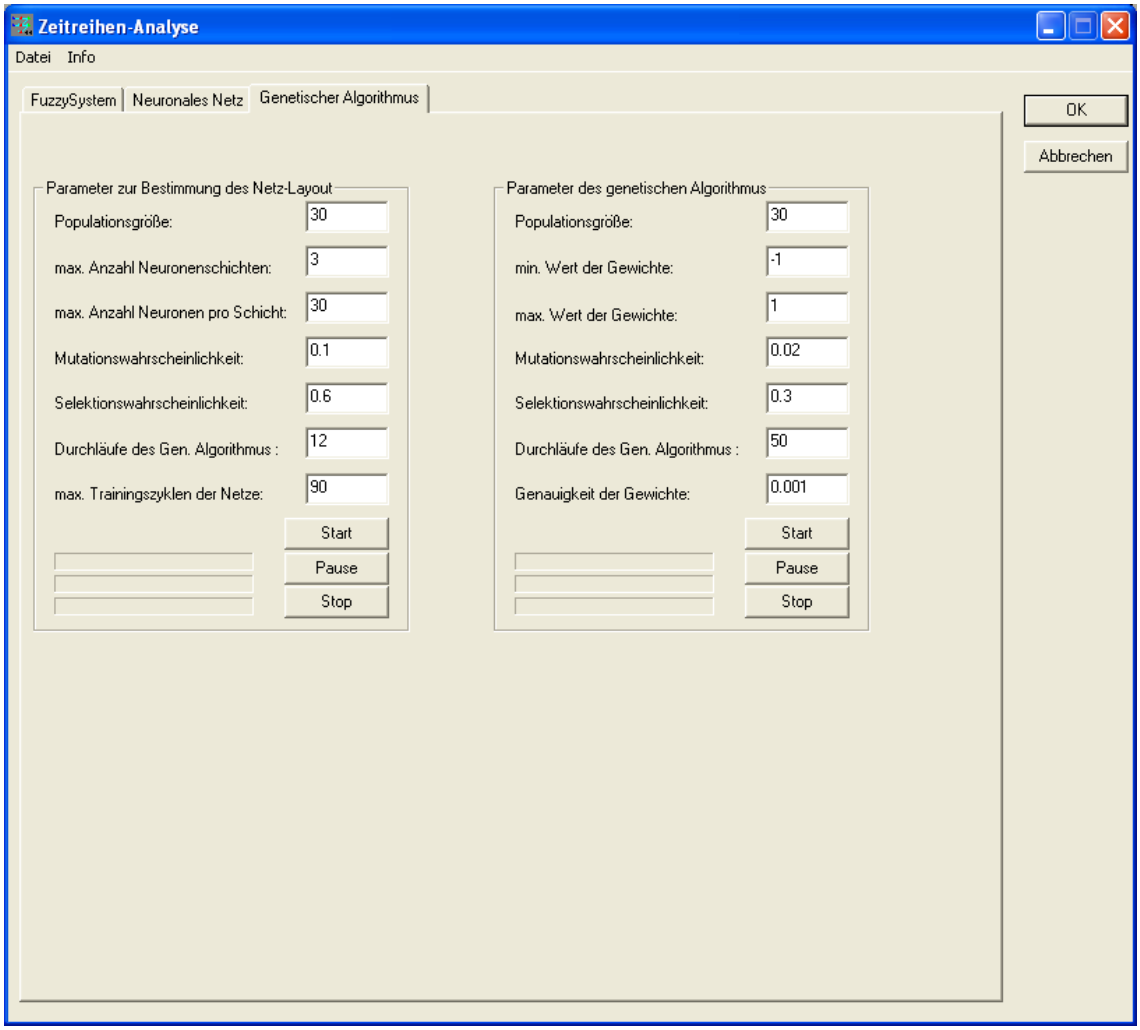
Übereinstimmung	absolute Häufigkeit	relative Häufigkeit
ja	911	74 %
nein	327	26 %
gesamt	1238	100 %

Ja, es gibt eine Steigerung von 71 % auf 74 %, aber berauschend ist das nicht.

5.5.4 Der dritte Test

Trotzdem rechtfertigen selbst diese bescheidenen 3 % eine weitere Erhöhung der kritischen Parameter, weil wir uns eine Verlängerung der Laufzeit problemlos leisten können. Jetzt werden die voreingestellten Parameter verdreifacht anstatt verdoppelt.

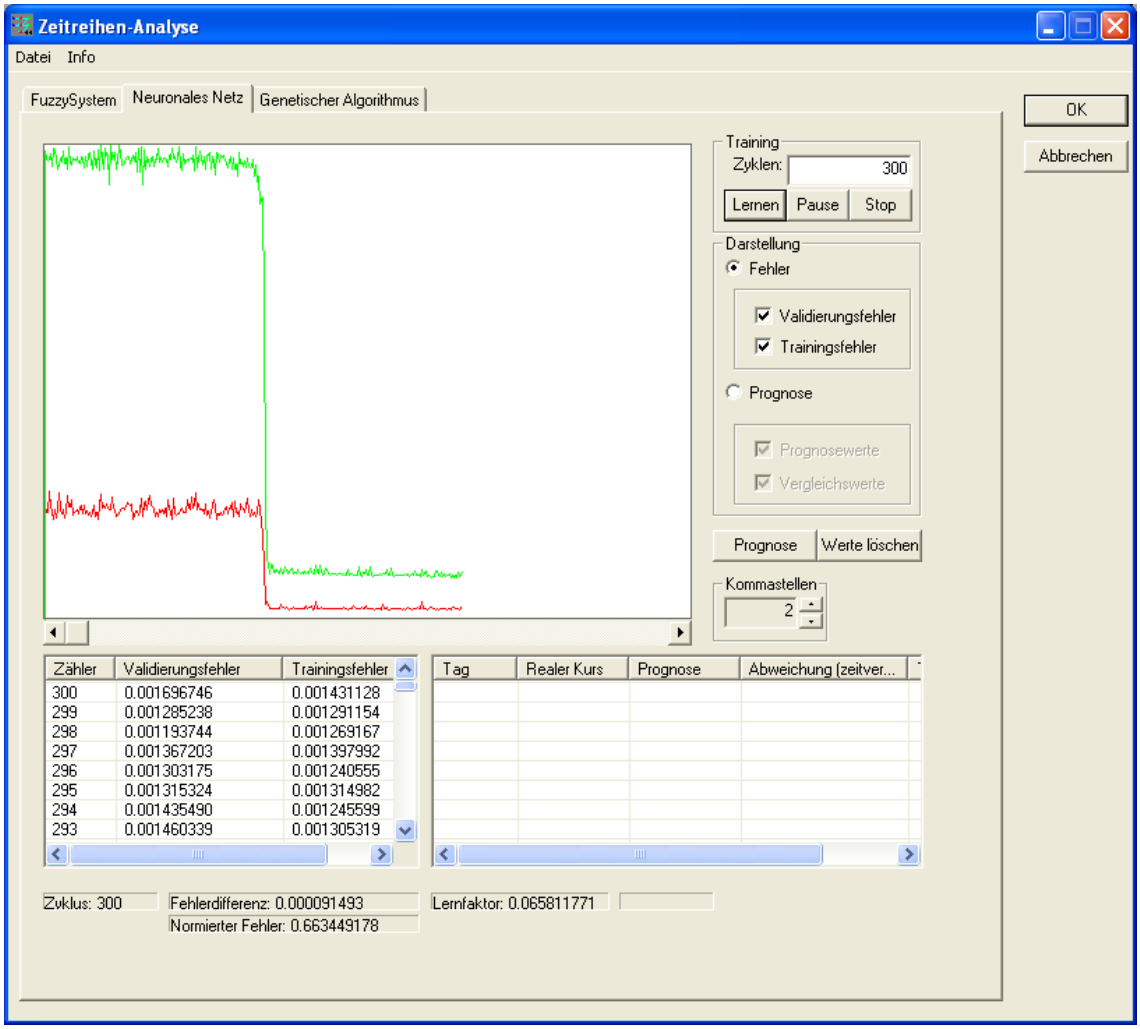
Damit stoßen wir noch nicht an die Grenzen moderner Hardware, jedoch leider an die Grenzen der verwendeten Software. Sie erlaubt bei Schritt 2 auf der rechten Seite des Eingabedialogs maximal 50 Durchläufe (Generationen). Daher werden im dritten Test 50 Generationen benutzt anstatt 60, was das Dreifache des voreingestellten Werts 20 wäre:



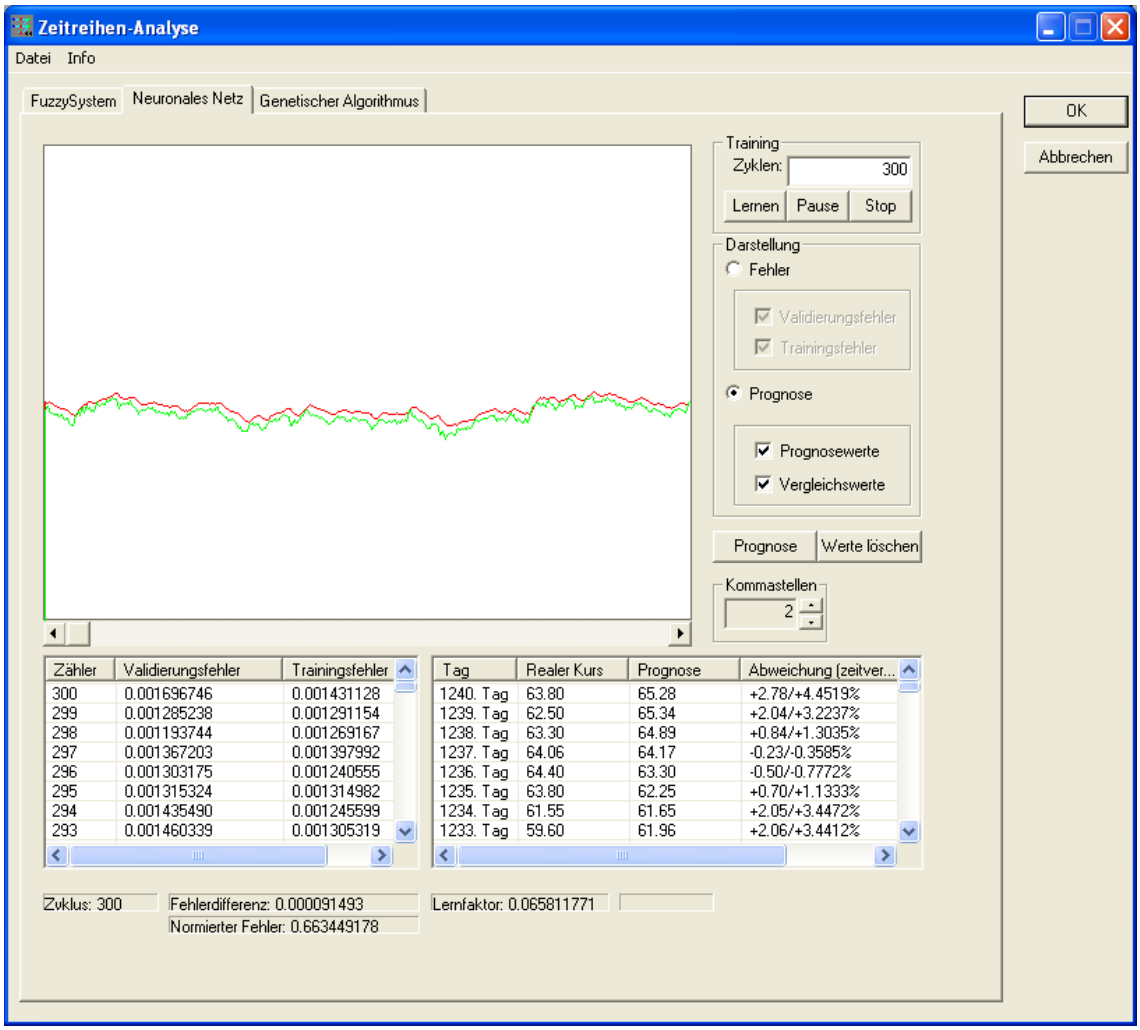
Nun wird zum ersten Mal die Geduld auf die Probe gestellt, während man auf die Ergebnisse wartet:

Schritt	Programmteil	Laufzeit
1	Genetischer Algorithmus für die Netzwerkarchitektur	257 s
2	Genetischer Algorithmus zur Initialisierung der Gewichte	24 s
3	Training des neuronalen Netzes	6 s
Gesamt		287 s

Nicht einmal fünf Minuten Gesamtlaufzeit sind aber immer noch sehr moderat. Der Fehler sinkt jetzt erst nach deutlich mehr Trainingszyklen als bisher:



Auch bei der Prognose ergibt sich ein verändertes Bild:



Erstmals ist mit freiem Auge zu erkennen, dass tatsächlich zwei verschiedene Kurven eingezeichnet sind. Laut den Zahlenwerten in der Tabelle rechts unten überschätzt die Prognose den realen Kurswert meistens, ab und zu unterschätzt sie ihn auch. Das ist definitiv kein gutes Zeichen, was die Prognosequalität angeht, und tatsächlich:

Übereinstimmung	absolute Häufigkeit	relative Häufigkeit
ja	879	71 %
nein	359	29 %
gesamt	1238	100 %

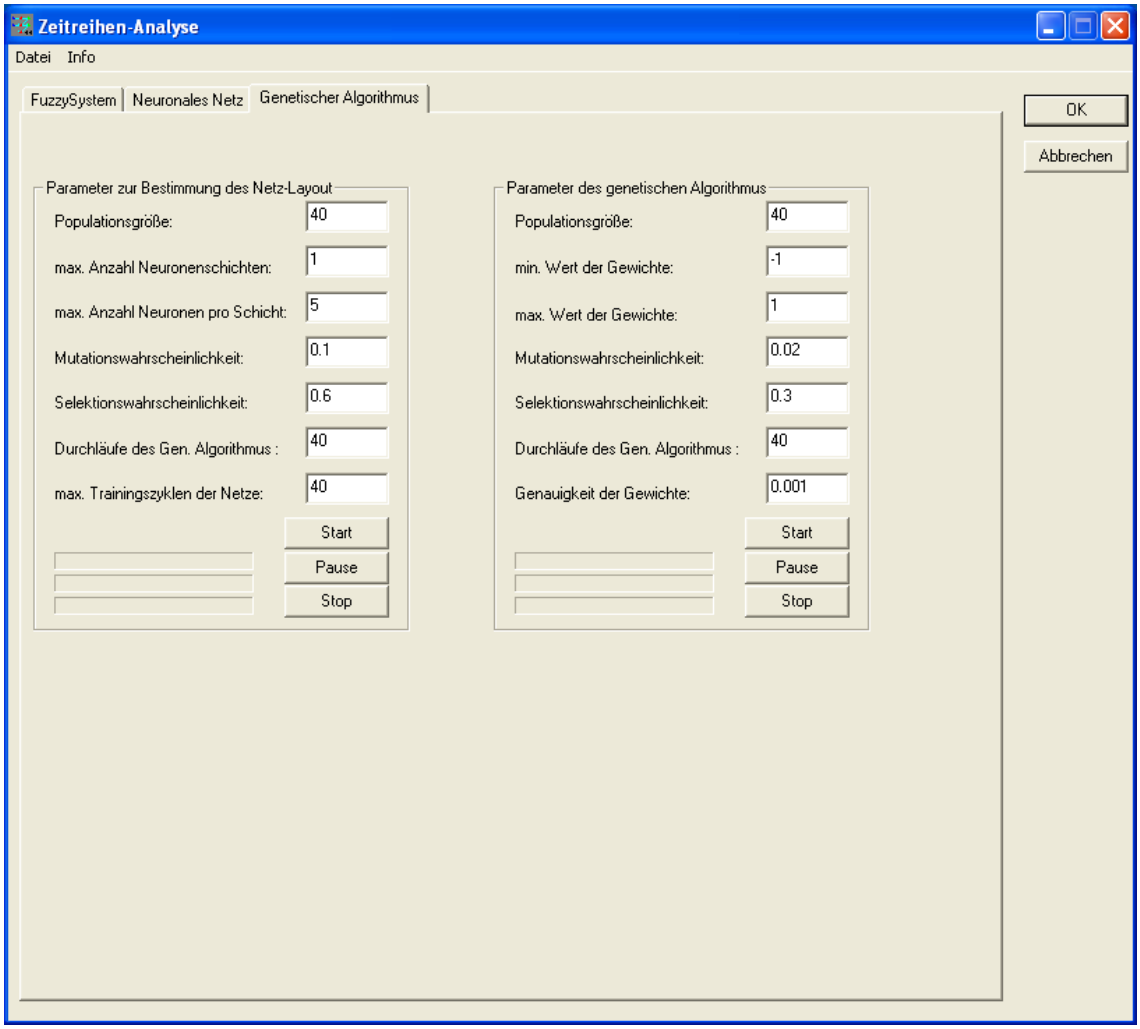
Obwohl wir dem neuronalen Netz mehr Individuen, Schichten, Neuronen, Generationen und Trainingszyklen zur Verfügung gestellt haben, ist die Erfolgswahrscheinlichkeit dennoch nicht gestiegen, sondern sogar gefallen. Daher scheint es nicht ratsam, die für die Laufzeit kritischen Parameter noch weiter zu erhöhen.

Eine Erklärung für dieses scheinbar paradoxe Ergebnis liefert Ertel [2]: Stellt man dem neuronalen Netz zu viele Neuronen zur Verfügung, wird es zur Minimierung des Fehlers die Muster auswendig lernen. Nur, wenn die Neuronen knapp sind, ist das neuronale Netz gezwungen, Gemeinsamkeiten in den Mustern zu finden und zu verallgemeinern. Weniger ist mehr. Ein kleines Netz kann unter Umständen bessere Ergebnisse liefern als ein großes.

5.5.5 Der vierte Test

Nach dieser interessanten Erkenntnis aus dem dritten Test lohnt es sich, darüber nachzudenken, wie die Parameter vernünftig eingestellt werden sollten, damit das neuronale Netz weder durch zu wenige noch durch zu viele Ressourcen an einer guten Performance gehindert wird.

Im letzten Test bestand das neuronale Netz aus 3 Schichten mit je maximal 30 Neuronen, also bis zu 90 Neuronen insgesamt. Alles, was dieses neuronale Netz als Datenmaterial zur Verfügung hatte, waren die 5 letzten Tagesschlusskurse. Da stellt sich die Frage, was quasi 90 kleine Taschenrechner sinnvoll berechnen können, wenn alle Berechnungen auf nur 5 Zahlen aufgebaut sind. Aus dieser Perspektive erscheint es unsinnig, mehr Neuronen als Daten zu verwenden. In diesem Test werden wir uns daher mit 5 Neuronen in einer einzigen Schicht begnügen. Es gibt jedoch keinen Grund, bei Individuen, Trainingszyklen oder Generationen zu sparen. Da manche Parameter 40 oder 50 als obere Schranke haben, nehmen wir einfach überall 40:



Bei Schritt 3, wenn das neuronale Netz trainiert wird, läge 40 unter dem in [7] vorgeschlagenen Wert von 100 Trainingszyklen. Um auf jeden Fall ausreichend Trainingszeit zur Verfügung zu stellen, nehmen wir hier 1000. Das produziert folgende Laufzeiten und Ergebnisse:

Schritt	Programmteil	Laufzeit
1	Genetischer Algorithmus für die Netzwerkarchitektur	44 s
2	Genetischer Algorithmus zur Initialisierung der Gewichte	8 s
3	Training des neuronalen Netzes	19 s
Gesamt		71 s

Übereinstimmung	absolute Häufigkeit	relative Häufigkeit
ja	1096	89 %
nein	142	11 %
gesamt	1238	100 %

Damit hat der vierte Test das mit Abstand beste Ergebnis gebracht. Weniger ist tatsächlich mehr.

5.5.6 Ausblick

Natürlich könnte man noch weitere Tests durchführen, Mutations- und Selektionswahrscheinlichkeit variieren, mehr als 5 Tagesschlusskurse verarbeiten usw. So ließe sich das Ergebnis eventuell noch weiter verbessern. Aber ehrlich gesagt ist bereits das Ergebnis des vierten Tests viel besser, als man das im Vorhinein erwarten durfte. Mir ist kein technischer Indikator für Aktienkurse bekannt, der behauptet, in 89 % der Fälle richtig zu liegen. Davon, dass er es tatsächlich schafft, ganz zu schweigen.

Bevor jetzt jemand seinen Job kündigt, um ausschließlich mit dem Programm von Kluge und Förster Aktien zu handeln, noch ein paar warnende Worte: Das neuronale Netz hat gelernt, wie sich der Aktienkurs der Deutschen Bank in jenem Zeitraum von 1240 Handelstagen verhalten hat. Das heißt nicht, dass sich dieselbe Aktie für alle Zeit weiter so verhält, und schon gar nicht, dass sich andere Aktien so verhalten müssen. Wie in Kapitel 5.5.1 dargelegt ändern die Märkte ihr Verhalten zwangsläufig ununterbrochen. Man kann also nicht damit rechnen, mit dem im vierten Test gefundenen neuronalen Netz in alle Ewigkeit Gewinne zu machen. Vielmehr müssen für jede Aktie immer wieder neue Netze gefunden werden. Das ist aber gerade die Stärke von neuronalen Netzen und auch von genetischen Algorithmen, dass sie ein ständiges flexibles Anpassen an geänderte Umweltbedingungen ermöglichen.

Insgesamt haben die genetischen Algorithmen und neuronalen Netze eindrucksvoll bewiesen, dass sie zumindest zur Zeitreihenanalyse hervorragend geeignet sind. Aber auch in anderen Bereichen wie der Erkennung von Texten, Bildern, Gesichtern oder Sprache liefern sie zum Teil hervorragende Ergebnisse, die auf anderem Wege unmöglich wären.

Kapitel 6

Literaturverzeichnis

[1] Zöller-Greer P., Künstliche Intelligenz – Grundlagen und Anwendungen, Composia Verlag

[2] Ertel W., Grundkurs Künstliche Intelligenz – Eine praxisorientierte Einführung, Vieweg

[3] Heistermann J., Genetische Algorithmen, B. G. Teubner Verlagsgesellschaft Stuttgart Leipzig

[4] Forster M., Optimierung Künstlicher Neuronaler Netze mit genetischen Algorithmen, Diplomica GmbH Hamburg

[5] Internationale Stiftung für Forschung in Paraplegie,
URL: http://www.ifp-zh.ch

[6] Association for Computing Machinery, URL: http://www.acm.org

[7] Hochschule für Technik und Wirtschaft Dresden,
URL: http://www.htw-dresden.de/~iwe/Belege/Foerster/Zeitreihe.html#problem

[8] Wikipedia, URL:
http://de.wikibooks.org/wiki/Mathematik:_Statistik:_Zeitreihenanalyse

[9] Wikipedia, URL:
http://de.wikibooks.org/wiki/Mathematik:_Statistik:_Trend_und_Saisonkomponente