// UNIVERSITÄT DORTMUND

REIHE COMPUTATIONAL INTELLIGENCE

SONDERFORSCHUNGSBEREICH 531

Design und Management komplexer technischer Prozesse und Systeme mit Methoden der Computational Intelligence

Simulationstool für Dynamische Neuronale Netze

Markus Rossmann, Christian Keim,
und Karl Goser

Nr. CI-36/98

Interner Bericht ISSN 1433-3325 April 1998

Sekretariat des SFB 531 · Universität Dortmund · Fachbereich Informatik/XI
44221 Dortmund · Germany

Diese Arbeit ist im Sonderforschungsbereich 531, „Computational Intelligence", der Universität Dortmund entstanden und wurde auf seine Veranlassung unter Verwendung der ihm von der Deutschen Forschungsgemeinschaft zur Verfügung gestellten Mittel gedruckt.

Inhaltsverzeichnis

1 Einleitung

Die Analyse künstlicher neuronaler Netzwerke erfordert häufig die Abbildung der entsprechenden Algorithmen in Hardware, da der Aufwand zur Generierung entsprechender Testsignale aufwendig ist und die Leistung der zur Verfügung stehenden Simulationssystemen insbesondere bei Echtzeitanforderungen häufig nicht ausreicht. Im Gegensatz dazu zeichnen sich Simulatoren insbesondere im Entwurfsstadium durch ihre hohe Flexibilität aus, wodurch sie sich für den Einsatz in zeitunkritischen Offline-Umgebungen eignen.

Die Informationsverarbeitung in neuronalen Netzwerken wird auch als *parallel distributed processing* (PDP) oder *neuronale Verbände* bezeichnet. Für die Simulation künstlicher Netzstrukturen mit dynamischen Neuronen werden die wesentlichen Eigenschaften biologischer Netze abstrahiert und als C++-Klassenbibliothek dem Anwender zur Verfügung gestellt.

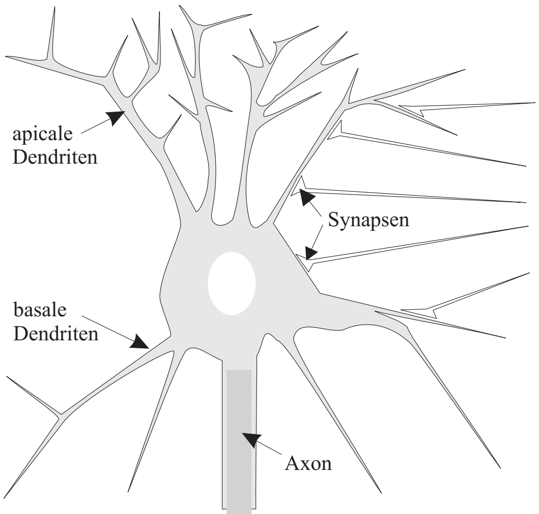
Dieser Forschungsbericht erläutert die Abstraktion *biologischer Neuronen* mit dem Ergebnis der Integration des modifizierten Hebbschen Lernverfahrens in Software, wodurch dessen Funktionalität zunächst durch diese simuliert werden kann und später in Silizium integrierbar ist. Das eigentliche Wissen der neuronalen Netzwerke ist hier sowohl in dessen Netzstruktur als auch in den Synapsengewichten kodiert. Durch seine Modularität und Objektorientiertheit schränkt die hier beschriebene Simulationssoftware die Vernetzung der zu analysierenden neuronalen Netze nicht ein, sondern richtet sich ausschließlich nach den Vorgaben des Anwenders. Die Stärke der Software liegt in der Tatsache, daß *beliebig viele* Neuronen *beliebig* miteinander verknüpfbar sind. Dem Anwender wird damit ermöglicht, ohne großes programmiertechnisches Vorwissen komplexe künstliche neuronale Netze zu erzeugen und effizient zu simulieren.

2 Theorie

Das Wissen über die Abläufe und Details in biologischen Neuronen wird stetig ausgebaut. Die Flut von wesentlichen und unwesentlichen Informationen erschwert es hierbei jedoch, die Natur präzise und perfekt zu beschreiben. Einen Kompromiß stellt die Modellbildung dar, die es uns ermöglicht, eine einfache Beschreibung unserer Vorstellungen zu formulieren und diese nach der Modellbildung experimentell zu verifizieren. Die Komplexität empirisch gewonnener Daten wird auf dem Wege der Abstraktion auf wesentliche Zusammenhänge reduziert, und daraus wird das Neuronenmodell entworfen, das möglichst universelle Eigenschaften der biologischen Neuronen widerspiegelt. Das hier analysierte Neuronenmodell beschreibt das biologisch motivierte *modifizierte Hebbsche Lernverfahren* und geht auf verschiedene Eigenschaften der biologischen Synapsen und Neuronen ein. Dieses Modell eines dynamischen Neurons beschäftigt sich mit der Übertragung von Signalen und deren Weiterverarbeitung im Neuron.

2.1 Aufbau der Neuronen

Künstliche neuronale Netze sind informationsverarbeitende Systeme, die aus einer großen Anzahl einzelner Elemente zusammengesetzt sind. Die Verarbeitung von Information in ihnen ist im Gegensatz zu den in der künstlichen Intelligenz verwendeten Verfahren nicht-symbolisch. Die einzelnen Elemente des künstlichen neuronalen Netzes sind biologischen Neuronen mehr oder weniger ähnlich und werden daher auch als Neuronen[1] bezeichnet. Um die starke Abstraktion der künstlichen Neuronen von natürlichen Neuronen deutlich zu machen, werden in der Literatur auch häufig die Bezeichnungen *Knoten* und *Element* statt Neuron verwendet. In neuronalen Netzen werden Informationen durch Erregung und Hemmung der Neuronen verarbeitet. Alle biologischen Nervenzellen, so unterschiedlich sie auch in Größe und Gestalt sind, haben den gleichen prinzipiellen Aufbau, der sich auch in dem hier beschriebenen künstlichen Neuronenmodell widerspiegelt. Er weist drei anatomisch unterschiedliche Regionen auf: Von den *Synapsen* auf den apicalen und basalen *Dendriten* gelangen Signale direkt zum *Zellkörper* (Soma), der in der Regel einen fadenförmigen Fortsatz besitzt, der als *Axon* bezeichnet wird. Während die Dendriten und dessen Synapsen für den Empfang von Signalen zuständig sind, hat das Axon die Aufgabe, Signale weiterzuleiten, die vom Zellkörper kommen.



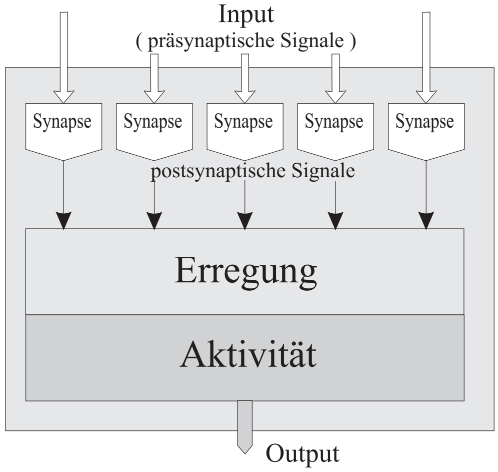


Abbildung 1: Aufbau der Neuronen

[1] Bei der Abstraktion biologischer Neuronen werden Faktoren wie mikroskopische Struktur, Form, Zellphysiologie, Neurochemie etc. vernachlässigt. Neuronen lassen sich als Informationsverarbeitungselemente verstehen.

Der Aufbau der verwendeten künstlichen Neuronen ist prinzipiell identisch. Jedes Neuron besteht aus beliebig vielen Synapsen, einer Einheit, die aus den postsynaptischen Signalen der Synapsen die Erregung des Neurons berechnet, und aus einer weiteren Einheit, welche aus der Erregung des Neurons die Aktivität berechnet und diese an nachfolgende Neuronen weiterleitet. In informationstheoretischer Hinsicht besteht die Funktion der Neuronen in der Bildung von mathematischen Produkten[2], der Summation dieser Produkte und der Weiterverarbeitung[3] zu einem Ausgabewert.

2.2 Synapse

Axone biologischer Nervenzellen besitzen präsynaptische Endungen, welche jeweils über einen synaptischen Spalt mit der postsynaptischen Membran verbunden sind. Alle drei Elemente zusammen repräsentieren eine biologische Synapse. Die Informationsübertragung zwischen biologischen Nervenzellen geschieht auf biochemischem Weg. Bei einer Umsetzung des beschriebenen Neuronenmodells in eine elektronische Schaltung basiert der Informationsfluß hingegen auf dem Fluß von Elektronen oder wird durch entsprechende elektrische Potentiale repräsentiert. Die Eigenschaft, daß eine Synapse das präsynaptische auf das postsynaptische Signal abbildet, bleibt jedoch erhalten. Es existieren verschiedene Synapsenvarianten, die sich in ihren Eigenschaften stark unterscheiden und jeweils mehr oder weniger dem biologischen Vorbild entsprechen.

2.2.1 Aufbau der Synapsen

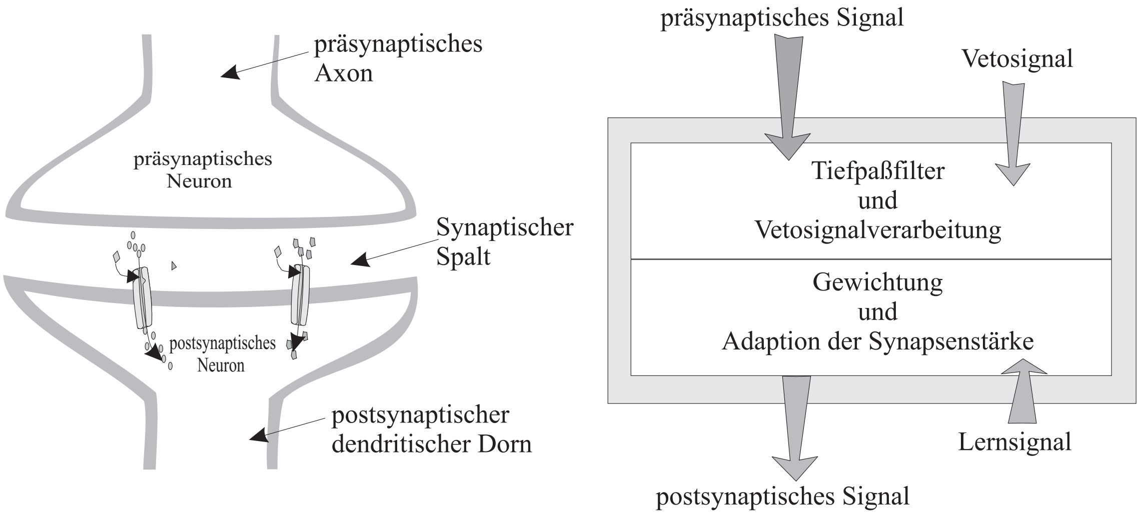


Abbildung 2: Aufbau der Synapsen

Synapsen, die erregend auf das Neuron wirken, werden als *exzitatorische Synapsen* bezeichnet und unterscheiden sich von den Synapsen, die hemmenden Einfluß auf das Neuron ausüben und als *inhibitorische Synapsen* bezeichnet werden. *Statische Synapsen* besitzen ein konstantes Synapsengewicht und unterscheiden sich von den plastischen Synapsen, deren Synapsenstärke aufgrund des *modifizierten Hebbschen Lernverfahrens* verändert wird. Aufgrund des Eingangssignalverhaltens unterscheiden sich tiefpaßgefilterte von nicht tiefpaßgefilterten Synapsen. In diesem Modell existieren Synapsen, die Vetosignale entweder beachten oder nicht beachten. Synapsen, die Vetosignale beachten, unterscheiden sich aufgrund ihrer Vetosignalverarbeitung. Es existieren zwei Arten der Vetosignalverarbeitung, die später näher beschrieben werden, die Eingangs- und Ausgangsvetosignalverarbeitung. Dieser Abschnitt beschäftigt sich mit den verschiedenen Eigenschaften der Synapsen, geht aber nicht auf jede spezielle Variante der Synapsen ein.

[2] Produkt aus den Eingangswerten und den Synapsengewichten.
[3] Z. B. dem Vergleich der Summe mit einem Schwellwert.

Die Implementierung einzelner Synapsenvarianten wird in anderen Kapiteln und im Handbuch für die Simulationssoftware vorgestellt. In diesem Abschnitt werden die Grundlagen für das allgemeine Verständnis von Synapsen dargestellt.

2.2.2 Die Tiefpaßeigenschaft der Synapse

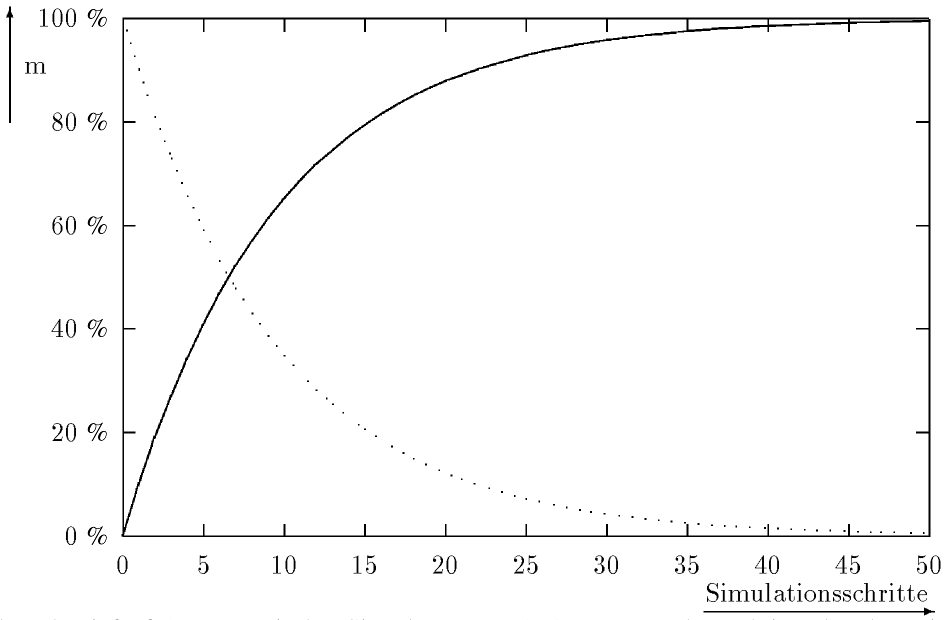


Abbildung 3: Tiefpaß (präsynaptisches Signal $x = const.$): Ausgang m als Funktion über der Zeit t ($\tau = 10$). Die steigende Funktion gilt für $x = 1$ und $m(0) = 0$ und die fallende (gepunktete Funktion) gilt für $x = 0$ und $m(0) = 1$.

Aufgrund des Eingangssignalverhaltens unterscheiden sich tiefpaßgefilterte von nicht tiefpaßgefilterten Synapsen. Die Tiefpaßeigenschaft wurde in dieses Neuronenmodell eingeführt, um dem biologischen Vorbild Rechnung zu tragen. Das Membranpotential bei biologischen Synapsen, welches durch die Öffnung von ionischen Kanälen entsteht, breitet sich mittels Diffusion in andere Regionen der Membran aus. Die Verschiebung des Membranpotentials erreicht per Diffusion das Soma. Der Dendrit wirkt dabei als *raumzeitliches Tiefpaßfilter* mit einer durchschnittlichen Zeitkonstante von etwa 5 *ms* in biologischen Systemen. Bei künstlichen Synapsen mit einer Tiefpaßfiltereigenschaft wird das Eingangssignal zunächst durch einen Tiefpaß modifiziert. Das Eingangssignal x ist hierbei eine beliebige (normierte) Funktion, deren Funktionswerte zwischen Null und Eins liegen.

$$x \in [0, 1]$$

Das modifizierte Eingangssignal ergibt sich zu:

$$x'(t) = \begin{cases} x(t) & \forall \text{ Synapsen ohne Tiefpaß} \\ \\ m(x,t) & \forall \text{ Synapsen mit Tiefpaß} \end{cases} \tag{1}$$

Durch die folgende Iterationsgleichung wird eine Funktion beschrieben, welche die Tiefpaßfiltereigenschaft beschreibt. Die auf- und die absteigende Flanke dieses Tiefpaßfilters haben unterschiedliche Zeitkonstanten. Die Zeitkonstante der aufsteigenden Flanke wird mit τ_r, und die Zeitkonstante der fallenden Flanke wird mit τ_f bezeichnet.

$$m(t+1) = \begin{cases} m(t) + \frac{1}{\tau_r} \cdot (x - m(t)) & \forall \ \ x \geq m(t) \\ \\ m(t) + \frac{1}{\tau_f} \cdot (x - m(t)) & \forall \ \ x < m(t) \end{cases}$$

2.2.3 Vetosignalverarbeitung

Es existieren Synapsen, bei denen für die Berechnung des postsynaptischen Signals das Vetosignal zu beachten ist. Vetosignale wurden eingeführt, damit Synapsen deaktiviert werden können, die zu einem späteren Zeitpunkt wieder aktiviert[4] werden sollen. Folglich wird die Plastizität eines Netzes erhöht. Biologisch motiviert sind diese Vetoeigenschaften aufgrund der Möglichkeit, daß in einem biologischen Netz Verbindungen sowohl ab- als auch aufgebaut werden können. Bei einem Eingangsveto wird das Eingangssignal in den Tiefpaßfilter bzw. das Eingangssignal der Synapse kurzgeschlossen. Das präsynaptische Signal, welches unter Umständen noch auf weitere Synapsen wirkt, bleibt jedoch unverändert.

2.2.4 Lernverfahren

Statische Synapsen besitzen ein zeitinvariantes Synapsengewicht und unterscheiden sich von den *plastischen Synapsen*, also den hier verwendeten *Hebbschen Synapsen*, deren Synapsenstärke aufgrund des *modifizierten Hebbschen Lernverfahrens* verändert wird. Diese *plastischen Synapsen* verändern ihre Eigenschaften, d. h. ihre Synapsenstärke, sofern es zu einer zeitkorrelierten Aktivität der prä- und postsynaptischen Erregung kommt. Eine solche langfristige Verstärkung, *long term potentiation* (LTP), oder Abschwächung, *long term depression* (LTD), der Synapsenstärke bildet nach dem heutigen Verständnis einen wesentlichen Beitrag für komplexe Gedächtnisleistungen in biologischen Netzstrukturen. Das Lernen nach dem modifizierten Hebbschen Lernverfahren ist ein Prozeß der räumlich *und* zeitlich lokalisiert stattfindet. Veränderungen der synaptischen Aktivität hängen nur von der Aktivität des präsynaptischen Neurons und der erzeugten postsynaptischen Erregung ab. Zunächst werden im folgenden die Variablen eingeführt, welche die Lerneigenschaft bestimmen. Die Variable x repräsentiert das präsynaptische Signal, d. h. das Eingangssignal der Synapse. Der Schwellwert des präsynaptischen Signals wird als θ_x bezeichnet. Die Variable y stellt die Differenz aus der Erregung des Neurons und dem postsynaptischen Signal der Hebb-Synapse dar. Folglich ist das Lernsignal y die Summe aller *anderen* postsynaptischen Signale des Neurons. Der Schwellwert wird als θ_y bezeichnet. Dieses Verfahren ermöglicht die ausschließliche Anwendung heterosynaptischen Lernens unter Vernachlässigung homosynaptischer Effekte.

$$\begin{aligned} y_i(t,\vec{h},\vec{x},\vec{v}) &= -s_i(t,h,x,v) + E(t,\vec{h},\vec{x},\vec{v}) \\ &= -s_i(t,h,x,v) + \sum_{j=1}^{n} s_j(t,h,x,v) \\ &= \sum_{j=1, j \neq i}^{n} s_j(t,h,x,v) \end{aligned} \tag{2}$$

Es werden drei mögliche Vorgänge in einer dynamischen Synapse unterschieden: Das *Lernen*, das *Verlernen* und das *Langzeitverlernen*. Eine Hebb-Synapse lernt, wenn die Differenz aus Erregung und postsynaptischem Signal den entsprechenden Schwellwert erreicht ($y \geq \theta_y$) und gleichzeitig das präsynaptische Signal über seinem Schwellwert liegt ($x \geq \theta_x$). Eine Hebb-Synapse vergißt, wenn die Differenz y aus Erregung und postsynaptischen Signal unter den entsprechenden Schwellwert θ_y fällt ($y < \theta_y$) und gleichzeitig das präsynaptische Signal über seinem Schwellwert liegt ($x \geq \theta_x$). In einer Hebb-Synapse findet Langzeitvergessen statt, wenn ebenfalls das präsynaptische Signal seinen Schwellwert nicht erreicht ($x \geq \theta_x$).

Die Änderung der Synapsenstärke, d. h. die Gewichtsveränderung der Hebb-Synapsen ergibt sich aus der modifizierten Hebbschen Lernbedingung wie folgt:

[4] In Hardware implementierte Verbindungen zwischen Neuronen können z. B. durch Hochstrom deaktiviert werden, sind dann aber irreversibel zerstört und können nicht wieder aktiviert werden.

$$h(t+1)-h(t)=\begin{cases}\frac{1}{\tau_l}\left(h_{max}-h(t)\right) & \forall x\geq\theta_x, y\geq\theta_y\\ \frac{1}{\tau_v}\left(h_{min}-h(t)\right) & \forall x\geq\theta_x, y<\theta_y\\ \frac{1}{\tau_{lv}}\left(h_{min}-h(t)\right) & \forall x<\theta_x\end{cases}\tag{3}$$

2.3 Erregung

2.3.1 Somatische Summation

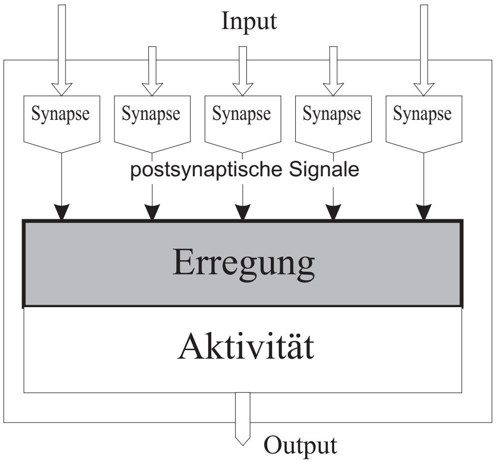


Abbildung 4: Somatische Summation

Die Erregung des jeweiligen künstlichen Neurons entspricht dem postsynaptischen Membranenpotential bei biologischen Neuronen. Der Dendritenbaum und der Zellkörper von biologischen Nervenzellen wirken hierbei wie ein *raumzeitlicher Integrator*. Bei diesem Neuronenmodell bildet sich die Erregung aus der algebraischen Summe aller postsynaptischen Potentiale. Dabei unterscheiden sich erregend von hemmend wirkenden postsynaptischen Signalen. Dieser Ansatz gestattet damit die Repräsentation sowohl exzitatorischer als auch inhibitorischer Synapsen, die durch das Vorzeichen der postsynaptischen Signale gekennzeichnet sind. Exzitatorische Synapsen wirken grundsätzlich erregend auf das Neuron, während inhibitorische Synapsen grundsätzlich hemmend auf das Neuron wirken. Die Erregung eines Neurons berechnet sich folglich aus der Subtraktion der Summe aller inhibitorischen postsynaptischen Signale von der Summe aller exzitatorischen postsynaptischen Potentiale.

$$E(t,\vec{h},\vec{x})=\sum_{i=1}^{n}s_i(t,h_e,x_i)-\sum_{j=1}^{m}s_j(t,h_i,x_i)\tag{4}$$

Hierbei bezeichnet $E(t)$ die Erregung des jeweiligen Neurons, n die Anzahl der exzitatorischen und m die Zahl der inhibitorischen Synapsen und $s(t)$ das postsynaptische Signal. Es wird hier außer acht gelassen, daß inhibitorische Potentiale beim biologischen Vorbild flacher verlaufen und länger anhalten als exzitatorische Anregungen. Für kleine synaptische Übertragungszeiten wird dieser Unterschied jedoch so klein, daß Gleichung 4 eine gute Näherung ergibt. Da jede Synapse maximal ein Signal zwischen Null und Eins zur Erregung des Neurons beitragen kann, liegt der Wertebereich der Erregung zwischen der negierten Anzahl der inhibitorischen Synapsen und der Anzahl der exzitatorischen Synapsen.

$$-m\leq E\leq n$$

2.4 Aktivität

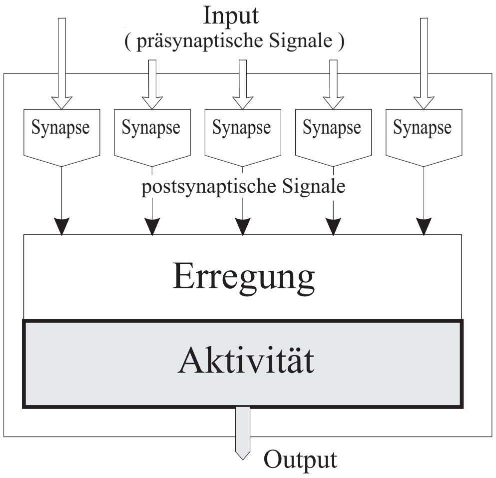


Abbildung 5: Berechnungseinheit der Aktivität des Neurons

Das Ausgangssignal des künstlichen Neurons wird durch seine Aktivität beschrieben. Diese berechnet sich mit Hilfe einer Übertragungsfunktion direkt aus der Erregung. Im Prinzip kann jede sigmoidale Funktion als Übertragungsfunktion verwendet werden.

$$\lim_{\varepsilon \to \infty} A(\varepsilon E) = \begin{cases} 1 & \forall\ E > 0 \\ \\ 0 & \forall\ E \leq 0 \end{cases}$$

Der Wertebereich der Aktivität ist zwischen Null und 100 Prozent festgelegt. Da die Aktivität des Neurons an die nachfolgenden Synapsen weitergeleitet wird, stellt sie den Output, d. h. das Ausgangssignal des Neurons dar. Damit bildet die Aktivität eines Neurons in einem Zeitschritt das einlaufende Signal einer Synapse im darauffolgenden Zeitschritt ab. Es gibt verschiedene Möglichkeiten, die Aktivität des Neurons aus der Erregung des Neurons zu berechnen. Soll die biologische Äquivalenz des Neuronenmodells dargestellt werden, so wird in der Regel eine Übertragungsfunktion ausgewählt, die dem biologischen Vorbild am nächsten ist. Für theoretische Untersuchungen eignen sich in der Regel Verteilungsfunktionen als Übertragungsfunktionen, und für die technische Anwendung des Neuronenmodelles eignen sich Übertragungsfunktionen, die einfach in Hardware zu realisieren sind.

2.4.1 Lineare Übertragungsfunktionen

Unter die linearen Übertragungsfunktionen fallen die Stufenfunktionen und beschränkten linearen Abbildungen. Bei der hier favorisierten linearen Übertragungsfunktion wird die Erregung linear auf die Aktivität abgebildet, wobei der Vorteil der linearen Übertragungsfunktion in der effizienten Hardwareimplementierung liegt. Es findet eine Begrenzung bei der maximalen und der minimalen Aktivität statt. Die maximale Aktivität wird dabei auf 100 Prozent beschränkt.

$$A(t) = \begin{cases} 1 & \forall\ 1 \leq E(t) \\ E(t) & \forall\ 0 \leq E(t) < 1 \\ 0 & \forall\ E(t) \leq 0 \end{cases} \tag{5}$$

3 Aufbau des Simulationstools

Zur Untersuchung der Qualität von Netzarchitekturen künstlicher neuronaler Netze werden in der Regel Computersimulatoren eingesetzt, da eine analytische Lösung für komplexe Netze zumeist nicht gefunden werden kann oder nur in engen Grenzen möglich ist. Dabei sind diese meist auf reguläre Netzarchitekturen beschränkt und daher nicht universell anwendbar.

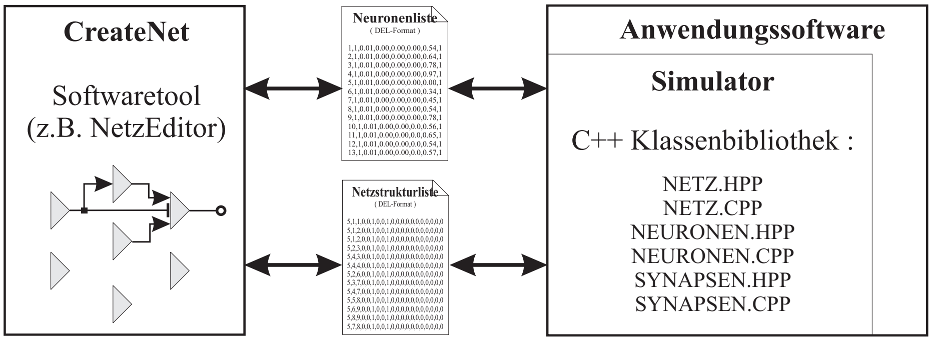


Abbildung 6: Verwendung der Simulationssoftware

Zur Übersicht und einfachen Anpaßbarkeit sind die Simulatoren der Anwendungen modular aufgebaut. Die Simulationssoftware gliedert sich in drei wichtige Teile:

- **CreateNet**
  Programme vom Typ *CreateNet* konstruieren aus abstrakten Vorgaben konkrete künstliche neuronale Netze und geben diese Netze jeweils in Form einer Netz- und einer Neuronenliste im DEL-Datenformat[5] aus. Diese Listen stellen den Bauplan des zu simulierenden Netzes dar. Mit dem sogenannten Netzeditor kann dieser Bauplan übersichtlich bearbeitet werden.

- **Simulator**
  Der Simulator ist durch eine C++ Klassenbibliothek definiert und basiert auf einem dynamischen Neuronenmodell (11). Diese Klassenbibliothek stellt Funktionen zur Verfügung, die es ermöglichen, das jeweilige Netz im Speicher aufzubauen, mit Eingabewerten zu beschicken, zu simulieren und Ausgabewerte abzurufen.

- **Anwendungsprogramme**
  Die jeweilige Anwendung importiert die C++ Klassenbibliothek und ist damit in der Lage, die eigentliche Anwendungssimulation zu steuern. In der eigentlichen Simulation werden die Konstruktionspläne des jeweiligen Netzes eingelesen und die entsprechende Netzstruktur wird im Speicher des Simulationsrechners aufgebaut. Die jeweilige Anwendung steuert das Einlesen der Eingabewerte, die Simulationsschritte und die Ausgabe.

3.1 Leistungsmerkmale der Simulationssoftware

Um beliebig strukturierte Netze untersuchen zu können, muß die Simulationssoftware universell einsetzbar sein. Es wird beispielsweise ein neuronales Netz benötigt, welches aus zwei verschiedenen Neuronentypen besteht, und folgende Funktionalität aufweist: Jedes Neuron des ersten Neuronentyps soll nur eine Hebb-Synapse besitzen. Jedes Neuron des zweiten Neuronentyps soll aus zwei Hebb-Synapsen, einer inhibitorischen und einer exzitatorischen Synapse bestehen. Um diese beiden Neuronentypen in einem Netz zu simulieren, ließe sich ein kleines Anwendungsprogramm schreiben, welches die Definitionen der Neuronentypen und die Anzahl der jeweiligen Neuronen schon im Quellcode[6] aufweist. Um diesen enormen

[5] Das DEL-Format wird in der Literatur auch als CSV-Format bezeichnet.
[6] Mit Quellcode ist z. B. die Klassendefinitionen der Neuronen gemeint.

Aufwand zu reduzieren, der für die Erstellung der jeweiligen Software notwendig ist, sind die hier vorgestellten Softwaretools entstanden.

Jedes neuronale Netz besteht aus künstlichen Neuronen, interneuronalen Verbindungen und Eingangsleitungen aus der Umgebung auf die Neuronen. Die Beschreibung dieser Elemente wird durch eine offene Schnittstelle vorgenommen, mit der sich demnach beliebige Netze auf der Grundlage der zur Verfügung gestellten dynamischen Neuronen aufbauen lassen. Die offene Schnittstelle soll aus Text-Dateien bestehen, die jeweils ein konkretes künstliches neuronales Netz definieren. Diese offene Schnittstelle erfüllt folgende Funktionalität:

1. **Beliebige Anzahl von Neuronen**
   Jedes Netz, das mit der hier vorgestellten Software simuliert werden kann, darf aus beliebig vielen unterschiedlichen Neuronen bestehen.

2. **Beliebige Neuronen**
   Jedes Neuron in einem künstlichen neuronalen Netz soll individuell konfigurierbar sein. Jedes Neuron kann damit über eine individuelle Initialisierungserregung, einen Offset und einen Initialisierungsoutput verfügen.

3. **Beliebige Anzahl von Synapsen**
   Jedes Neuron soll beliebig viele Synapsen jedes Synapsentyps auf dessen Dendritenbaum und Axon besitzen können.

4. **Beliebige Synapsen**
   Jede Synapse, d. h. jede Eingangsleitung und jede Verbindung zwischen zwei Neuronen soll individuell konfigurierbar sein. Konkret bedeutet diese Forderung, daß jede Synapse individuelle Zeitkonstanten[7] und eine individuelle Initialisierung erhalten kann.

5. **Veto Synapsen**
   Auf jede Synapse sollte eine Vetoleitung führen, so daß mit Hilfe von Veto-Signalen einzelne Synapsen beeinflußt werden können.

Mit Hilfe einer solchen offenen Schnittstelle können im allgemeinen Fall beliebig viele Neuronen beliebig miteinander verbunden werden. In der Simulation einer speziellen Anwendung werden ein oder mehrere konkrete Netze verwendet. Folglich ist ein Ziel des *Softwaresimulators*, eine Datenstruktur zur Verfügung zu stellen, welche alle erdenklichen, anwendungsspezifischen Netze mit Hilfe der offenen Schnittstelle aufbauen kann. Die verschiedenen Simulatoren der speziellen Anwendungen werden die verallgemeinerte Datenstruktur als Basis verwenden.

[7] Mit Zeitkonstanten sind z. B. für Hebb Synapsen die Zeitkonstanten für das Lernen, das Vergessen und das Langzeitvergessen gemeint. Bei Synapsen, die eine Tiefpaßfiltercharakteristik aufweisen, sind die Zeitkonstanten für den Anstieg und Abfall der Flanken gemeint.

3.2 Allgemeines zu Simulatoren

3.2.1 Vor- und Nachteile eines Software-Simulators

Der Vorteil eines Software-Simulators gegenüber einem in Hardware implementierten Netzes liegt in der großen Flexibilität und dem schnellen Aufbau der neuronalen Verbindungen. Durch einen Softwaresimulator kann a priori getestet werden, ob ein spezieller Hardwareaufbau grundsätzlich funktionsfähig ist. Es ist möglich, mit dessen Hilfe die unterschiedlichen Netzarchitekturen für ein spezielles Problem zu analysieren und zu bewerten. Ferner kann bereits realisierte Hardware mit einer Simulation verglichen werden, um die ordnungsgemäße Funktion der Hardware zu verifizieren.

Der größte Nachteil eines Simulators zur Analyse paralleler System besteht darin, daß er auf einem sequentiellen Rechensystem umgesetzt wird. Die Leistungsfähigkeit eines neuronalen Netzes beruht jedoch auf seiner massiven Parallelität, was im Simulator zwar nachgebildet wird, aber aufgrund der sequentiellen Architektur schnell an seine Grenzen stößt.

3.2.2 Geschwindigkeit der Simulation

Um den Geschwindigkeitstest durchführen zu können, werden 500 Neuronen- und Netzlisten, d. h. 500 künstliche neuronale Netze mit einem C++-Steuerprogramm vom Typ *CreateNet* erzeugt. Jedes der dadurch beschriebenen Netze besitzt eine Eingangsleitung auf das erste Neuron. Das i-te Neuron ist mit dem $i+1$-ten Neuron über sechs Synapsen verbunden. Diese teilen sich zu gleichen Teilen auf zwei Hebbsche, zwei inhibitorische und zwei exzitatorische Synapsen auf. Jedes Paar besteht aus einer Synapse mit und einer ohne Tiefpaßeigenschaft. Das erste Netz besteht lediglich aus einem Neuron und einer Eingangsleitung. Zu jedem weiterem Netz werden ein Neuron und sechs verschiedene Synapsen vom letzten Neuron zu diesem neuen Neuron hinzugefügt. Die folgende Grafik stellt die Zeit dar, die auf einer herkömmlichen Intel-basierten PC Plattform mit Pentium-Prozessor (133 MHz) für einen vollständigen Simulationsschritt für das entsprechende Netz benötigt wird. Die dargestellte Zeit wird dabei über 10^3 Simulationsschritte gemittelt.

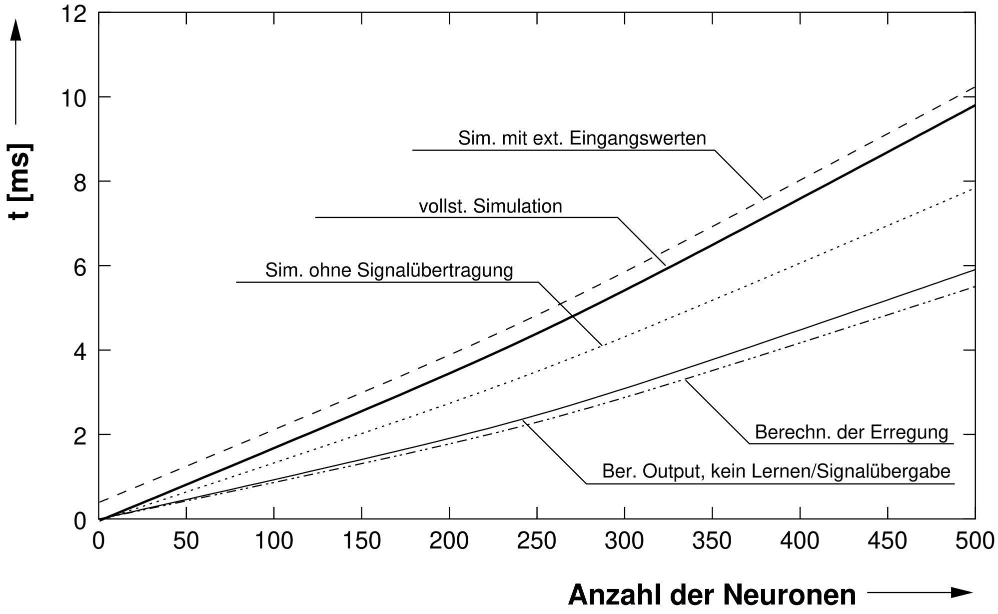


Abbildung 7: Simulationsdauer über der Größe des Netzes

Der mit „vollst. Simulation“gekennzeichnete Verlauf stellt die Zeitdauer des eigentlichen Simulationsschrittes dar. In einem weiteren Durchlauf wird zudem eine Eingangsleitung sequentiell mit 500 Werten beschickt, woraus sich der mit „Sim. mit ext. Eingangswerten“markierte Verlauf ergibt. Der Offset dieser beiden Verläufe stellt demnach die Dauer für die Beschickung einer Eingangsleitung mit 500 externen Werten dar. Zur Übergabe eines Eingangssignals an eine Eingangssynapse benötigt dieser Durchlauf ca. $t = 0.88\mu s$.

Jeder Simulationsschritt setzt sich aus der Berechnung der Erregung, der Berechnung des Outputs, dem Lernen aller Hebb-Synapsen und der Übergabe des Outputs an die folgenden Synapsen zusammen. Für die Bestimmung der Dauer, den diese Teile des Simulationsschrittes benötigen, werden weitere Durchläufe durchgeführt, bei denen jeweils einzelne Funktionalitäten der Simulation herausgenommen werden.

In Abbildung 7 spiegelt der mit „Sim. ohne Signalübertragung“die Zeitdauer für einen Simulationsschritt ohne Übergabe der Ausgangssignale der Neuronen an die jeweils nächsten Synapsen wieder. Wird das Lernen beider Hebbschen Synapsen pro Neuron abgeschaltet, ergibt sich der mit „Ber. Output, kein Lernen/Signalübergabe“bezeichnete Verlauf. Der unterste Verlauf berücksichtigt die Berechnung der Erregung unter Vernachlässigung der Übertragungsfunktion. Die Differenz zwischen der untersten und der darüberliegenden Linie stellt demnach die Berechnung der Variablen *Output* aus der Variablen *Erregung* dar. Der Offset zwischen dem zweiten und dritten Kurvenverlauf von unten stellt das Lernen der Hebbschen Synapsen dar.

Art	**Simulationsdauer**
Statische Synapse	$1.9\,\mu s$
Hebb-Synapse	$2.9\,\mu s$

Tabelle 1: Simulationsdauer einzelner Funktionalitäten

Bei den Geschwindigkeitstests wird abgeschätzt, wieviel Zeit ein Simulationsschritt auf einer Intel Pentium 133 MHz Plattform benötigt. Diese Daten sind in Tab. 1 dargestellt. Die Zeit für jeden Simulationsschritt hängt dabei nicht nur von der Anzahl der Neuronen ab, sondern im wesentlichen von der Art und der Anzahl der Synapsen[8].

[8] Bei Hebbschen Synapsen wird zusätzlich in jedem Simulationsschritt ein Lernschritt durchgeführt.

4 Aufbau neuronaler Netze (CreateNet)

Ein anwendungsspezifisches neuronales Netz wird durch zwei Dateien definiert, die sogenannte Neuronenliste und die Netzstruktur. Jede dieser Listen kann mit einem Texteditor oder mit Hilfe des CreateNet-Tools erstellt und bearbeitet werden. Im CreateNet-Tool wird das Netz definiert und als Neuronenliste und Netzstruktur ausgegeben. Zum weiteren Verständnis wird nun eine einfaches neuronales Netz definiert, das später auch simuliert werden wird.

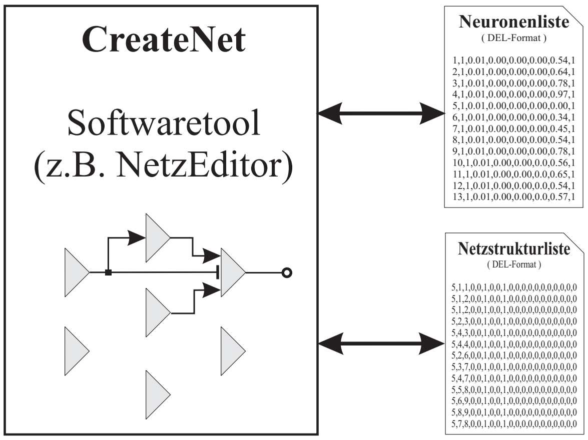


Abbildung 8: Funktion von Tools des Typs CreateNet

4.1 Die offene Schnittstelle

In diesem Abschnitt wird erläutert, wie die Definition des jeweiligen Netzes an die Simulationssoftware übergeben wird. Um beliebige neuronale Netze auf Basis eines Lernverfahrens aufzubauen, wird eine offene Schnittstelle benötigt. Diese Schnittstelle soll sowohl einfach und lesbar, als auch möglichst funktionell aufgebaut sein. Um dies zu verwirklichen, besteht sie aus zwei Listen, der *N*euronenliste und der *N*etzstrukturliste.

1. **Neuronenliste**
   In der Neuronenliste stehen alle Neuronen des jeweiligen Netzes. Für jedes Neuron existiert in der Neuronenliste eine eigene Initialisierungszeile.

2. **Netzstrukturliste**
   Die Netzstruktur ist die Verbindungsliste des Netzes. Die Netzstruktur besteht aus einer Liste von Synapsen, wobei jede Zeile dieser Liste eine Synapse zwischen zwei Neuronen oder einer Eingabeleitung darstellt.

Ein Vorteil dieser Listen ist, daß die Simulation in jedem beliebigen Simulationsschritt unterbrochen werden kann und die Werte auf einem Datenträger abgelegt werden können. Dies ermöglicht nicht nur, beliebige Simulationsschritte zwischenzuspeichern, sondern auch einen Simulationsschritt erneut einzuladen, um die Simulation auf einem anderen System oder zu einem späteren Zeitpunkt fortzusetzen. Der wesentliche Vorteil dieser Schnittstelle besteht jedoch vor allem darin, daß die künstlichen neuronalen Netze jederzeit umstrukturiert werden können, ohne einen neuen Anwendungssimulator zu erzeugen.

4.1.1 Datenformat

Um die Schnittstelle lesbar zu halten, benutzen wir kein Standard-Datenbankformat wie PC/IXF[9] oder WSF[10], sondern ein gebräuchliches ASCII-Format, das sogenannte DEL-Format[11], welches erweitert auch als CSV-Format[12] bekannt ist. Der Aufbau der Dateien im DEL-Format folgt einem einfachen Schema:

- Die Datei wird als ASCII-String zeilenweise aufgebaut.
- Die einzelnen Werte werden durch Kommas separiert.
- Texte werden zusätzlich in Anführungszeichen eingerahmt.

4.2 Neuronenliste

4.2.1 Definition der Neuronenliste

Die Neuronenliste soll sowohl als Ein- als auch als Ausgabeschnittstelle[13] benutzt werden. Jede Zeile der Neuronenliste stellt ein Neuron dar. Jedes Neuron erhält eine fortlaufende Nummer, um es später zu referenzieren. Zusätzlich steht in der Liste der Neuronentyp, die Erregung, der Offset, der Output und der Simulationsschritt eines jeden Neurons. Die Bezeichnungen der einzelnen Neuroneneigenschaften wurden so gewählt, daß sie später auch in einer Datenbanktabelle weiterverwendet werden können. Jede Zeile besteht aus einer Reihe von Neuroneneigenschaften, die folgende Reihenfolge aufweist:

1. **NeuronIndex**
   Der Index des Neurons ist der numerische Bezeichner des Neurons. Der Index des Neurons ist eine ganze positive Zahl. Die Verwendung des Neuronenindex ist dem Netzdesigner überlassen.

2. **Typ**
   Der Typ des Neurons legt das Clipping fest, d. h. ob der Output des Neurons linear oder nichtlinear aus der Erregung des Neurons berechnet wird. Der Typ des Neurons wird über eine ganze positive Zahl beschrieben. Welche Typen zur Verfügung stehen, wird weiter unten in diesem Abschnitt erläutert.

3. **Erregung**
   Die Erregung gibt bei der Neuronendefinition die Initialisierungserregung an. In einer Ausgabeliste kann die Erregung, d. h. der innere Neuronzustand abgelesen werden. Die Erregung ist eine reelle Zahl mit einem Wertebereich, der zwischen Null und Eins liegen sollte. Höhere positive und negative Erregungen sind auch möglich, haben aber zur Folge, daß bei der Berechnung der Aktivität des Neurons diese Information verloren geht, da sie ohnehin entsprechend begrenzt werden.

4. **Offset**
   Der Offset beschreibt denjenigen Wert, um den der Output des Neurons verringert wird. Da der Offset in der Regel konstant bleibt, ist er nur als Initialisierungsinformation, d. h. als Eingabeinformation von Bedeutung. Der Offset ist eine reelle Zahl.

5. **Verschiebung**
   Zur Berechnung des Outputs des Neurons wird bei einigen Neuronentypen eine Verteilungsfunktion verwendet. Diese Größe gibt die Verschiebung der Verteilungsfunktion an, d. h. ab wann die Erregung den Output des Neurons besonders stark beeinflußt. Die Verschiebung ist eine reelle Zahl.

---

[9] IXF ist ein IBM-DB2-Format.

[10] WSF ist die Abkürzung für Work-Sheet Format (Arbeitsblattformat), welches z. B. bei Lotus Symphony und 1-2-3 Programmen verwendet wird.

[11] Die Abkürzung DEL steht für delimited ASCII. Das DEL-Format wird unter anderem in einer Reihe von Datenbankmanager- und Dateimanagerprogrammen verwendet. Die bekanntesten sind dBASE II, dBASE III, BASIC, IBM DB2 und IBM Personal Decision Series (IBM PDS).

[12] CSV-Format = Comma Separated Values

[13] Die Neuronenliste kann damit nicht nur zur Erzeugung des Netzes genutzt werden, sondern dient gleichzeitig dazu, die Erregung zu jedem Simulationsschritt abzulesen.

6. **Steilheit**
   Diese Größe gibt die Steilheit der nichtlinearen Übertragungsfunktion an, d. h. in welchem Wertebereich die Erregung den Output des Neurons besonders stark beeinflußt. Die Steilheit ist eine positive reelle Zahl. Je größer die Steilheit gewählt wird, desto steiler wird die Übertragungsfunktion, und umso kleiner wird der Wertebereich, in dem die Erregung den Output besonders stark beeinflußt.

7. **Output**
   Der Output ist das Ausgabesignal des Neurons. Der Output des Neurons ist eine reelle positive Zahl mit einem normierten Wertebereich zwischen Null und Eins. Dieser Wert ist für die Zwischenspeicherung einer Neuronenliste interessant.

8. **Simulationsschritt**
   Diese Angabe ist nur für eine Ausgabe interessant. Man kann dann den Zustand des Neurons anhand der Erregung mit Hilfe dieses Index und des Neuronenindex zuordnen. In jeder Liste sollte nur ein Neuron mit dem gleichen Index und Zeitschritt auftreten.

4.2.2 Neuronentypen

Entsprechend verschiedener Eigenschaften der Neuronen unterscheiden wir die in Tab. 2 zusammengefaßten Neuronentypen.

Typ	Neuron	Eigenschaft
Typ 1	Lineare Neuronen	Die linearen Neuronen beschreiben Neuronen mit linearer Übertragungsfunktion und modifizierter Hebbscher Lernregel.
Typ 2	Interneuronen	Die Interneuronen beschreiben Neuronen mit nichtlinearer Übertragungsfunktion und modifizierter Hebbscher Lernregel.
Typ 3	nichtbinäre Neuronen	Diese Neuronen beschreiben Neuronen mit nichtlinearer Übertragungsfunktion und einer anderen Lernregel.

Tabelle 2: Typen verschiedener Neuronen

4.2.3 Ausschnitt aus einer typischen Neuronenliste

Index	Typ	Erregung	Offset	Verschiebung	Steilheit	Output	Simulationsschritt
1	1	0.00	0.0	0.0	0.0	0.00	0
2	1	0.00	0.0	0.0	0.0	0.00	0

Tabelle 3: Neuronenliste

Anhand dieses Beispiels wird demonstriert, wie einfach eine Neuronenliste definiert wird. Jede beliebige Anzahl von Neuronen kann über eine textbasierte Liste definiert werden. Sie entspricht im DEL-File-Format der in Tab. 4 gezeigten Form.

Neuronenliste im DEL-Format
1,1,0.00,0.0,0.0,0.0,0.00, 0
2,1,0.00,0.0,0.0,0.0,0.00, 0

Tabelle 4: Neuronenliste im DEL-Format

4.3 Netzstrukturliste

4.3.1 Definition der Netzstrukturliste

Die Netzstrukturliste beschreibt sowohl die Verbindungen zwischen den Neuronen als auch die Eingangsleitungen. Da jede Verbindung auf einer Synapse endet, wird die Verbindungsstruktur des jeweiligen neuronalen Netzes durch die einzelnen Synapsen beschrieben. Jede Zeile dieser Netzstrukturliste stellt dabei eine Synapse dar. Ob es sich um eine Synapse zwischen zwei Neuronen oder um eine Eingangsleitung handelt, wird durch das Vorzeichen der Variablen *PreSynaptischerIndex* festgelegt. Ist der Wert negativ, so handelt es sich um eine Eingangsleitung, ist er positiv, so handelt es sich um eine Synapse zwischen zwei Neuronen. Der *NeuronIndex* bezeichnet das Neuron, auf das die Ausgabe der Synapse geleitet werden soll.

Die Netzstruktur besteht aus den folgenden Spalten.

1. **Typ**
   Der Synapsentyp unterscheidet die Eigenschaft der Synapse, z. B. bedeutet eine Eins, daß es sich um eine Hebb-Synapse handelt. Der Synapsentyp ist eine natürliche positive Zahl zwischen Eins und Sechs. Welche Typen zur Verfügung stehen, wird weiter unten in diesem Abschnitt erläutert.

2. **PreSynaptischerIndex**
   Dieser Index beschreibt den Index des Neurons, welches das präsynaptische Signal liefert. Der Wertebereich dieser Indizes ist durch die Neuronenindizes festgelegt und erweitert sich lediglich um die Kennziffern der Eingangsleitungen. Die Kennziffern der Eingangsleitungen sind negative ganze Zahlen.

3. **NeuronIndex**
   Dieser Index bezeichnet das Neuron, zu dem die Synapse gehört. Der Wertebereich ist durch die Indizes der Neuronen beschränkt.

4. **PreSynaptischesSignal**
   Dieses Signal ist das Eingangssignal der Synapse. Das Signal wird durch eine reelle Zahl codiert, dessen Wertebereich zwischen Null und Eins liegt.

5. **PostSynaptischeSignal**
   Dieses Signal ist das Ausgangssignal der Synapse. Das Signal wird durch eine reelle Zahl codiert, dessen Wertebereich zwischen Null und Eins liegt.

6. **Gewicht**
   Das Gewicht der Synapse beschreibt den Einfluß der Synapse auf das Neuron. Die Gewichte sind normiert, d. h. ein Synapsengewicht ist eine reelle Zahl zwischen Null und Eins.

7. **VetoIndex**
   Dieser Index bezeichnet das Neuron, welches auf die Synapse ein Veto-Signal abgibt. Der Veto-Index ist eine natürliche Zahl. Der Wertebereich ist durch die Indizes der Neuronen beschränkt. Wenn der Veto-Index Null ist, bedeutet dies, daß kein Neuron auf ihn ein Veto ausüben kann. Jede Synapse kann höchstens von einem Neuron ein Vetosignal erhalten.

8. **VetoSignal**
   Dieser Wert beschreibt das Veto-Signal zum jeweiligen Simulationsschritt. Dieses Veto-Signal ist in der laufenden Simulation durch das Ausgangssignal des Neurons mit dem VetoIndex bestimmt. Das Signal ist eine positive reelle Zahl mit einem Wertebereich zwischen Null und Eins.

9. **VetoSchwelle**
   Der Wert bestimmt die Schwelle, ab der die Synapse das Veto erfährt. Die Vetoschwelle ist eine reelle Zahl zwischen Null und Eins. Ist das Veto-Signal größer als diese Schwelle, so gibt die Synapse kein Ausgangssignal zurück. Durch Festlegung dieser Vetoschwelle auf den Wert größer Eins wird erreicht, daß die Vetoschwelle nicht überschritten wird.

10. **STM**
Das *Short Term Memory* wird für die Zwischenspeicherung des Tiefpaßausgangs benötigt. Bei dem gespeicherten Wert des STM handelt es sich stets um den letzten gespeicherten Wert des Tiefpaßfilters in der Synapse, was durch eine zeitliche Verzögerung bewirkt wird. Bei Synapsen ohne Tiefpaßfilter findet dieser Wert keine Verwendung. Zur Initialisierung der Synapse kann der Wert auf Null gesetzt werden. Das STM ist eine reelle Zahl zwischen Null und Eins.

11. **TauRisingSTM**
Die Zeitkonstante des Tiefpaßfilters auf der aufsteigenden Flanke in Sekunden. Diese Zeitkonstante ist eine reelle positive Zahl. Eine Größe von beispielsweise 0.015 legt die Zeitkonstante auf $15\,ms$ fest.

12. **TauFallingSTM**
Die Zeitkonstante des Tiefpaßfilters auf der absteigenden Flanke in Sekunden. Diese Zeitkonstante ist eine reelle positive Zahl. Eine Größe von beispielsweise 0.12 legt die Zeitkonstante auf $120\,ms$ fest.

13. **SchwellwertLPF/INP**
Diese reelle Zahl zwischen Null und Eins beschreibt den Schwellwert des Eingangs. Diese Information wird nur bei Hebb-Synapsen benötigt.

14. **SchwellwertOTH**
Diese reelle Zahl zwischen Null und Eins beschreibt den Schwellwert für die Entscheidung, ob die Erregung des Neurons durch die übrigen Synapsen ausreicht, um Hebbsches Lernen zu initiieren.

15. **TauLernen**
Dieser Wert beschreibt die Zeitkonstante für das Hebbsche Lernen in Sekunden. Die Zeitkonstante ist eine positive reelle Zahl. Ein möglicher Wert von 300 beschreibt eine Zeitkonstante von $300\,s$. Diese Information wird nur bei Hebbschen Synapsen benötigt.

16. **TauVergessen**
Dieser Wert beschreibt die Zeitkonstante für das aktive Vergessen nach dem modifizierten Hebbschen Lernverfahren in Sekunden. Diese Zeitkonstante ist eine positive reelle Zahl. Einen üblicher Wert ist z. B. das achtfache der Zeitkonstante des Hebb-Lernens. Diese Information wird nur bei Hebbschen Synapsen benötigt.

17. **TauLangzeitVergessen**
Dieser Wert beschreibt die Zeitkonstante für das Langzeitvergessen nach dem modifizierten Hebbschen Lernverfahren in Sekunden. Die Zeitkonstante ist eine positive reelle Zahl. Ein üblicher Wert ist ein Vielfaches der Zeitkonstante für das Lernen (TauLernen). Diese Information wird nur bei Hebbschen Synapsen benötigt.

18. **SchrittAnzahl**
Die Anzahl der ausgeführten Simulationsschritte ist interessant, wenn das entsprechende File als Ausgabeprotokoll verwendet wird. Die Anzahl der Simulationsschritte ist immer eine ganze positive Zahl.

4.3.2 Die verschiedenen Synapsenklassen

Jeder Synapsentyp besitzt allgemeine Synapseneigenschaften. Es werden sechs Synapsenarten unterschieden, die sich aufgrund ihres Eingangs- und Adaptionsverhaltens unterscheiden.

4.3.3 Ausschnitt aus einer typischen Netzliste

Die Netzliste wird der Schnittstelle des Programmes im DEL-Format zur Verfügung gestellt. Einen Ausschnitt aus einer Netzstrukturliste zeigt Tab. 6. Die Reihenfolge der Einträge richtet sich dabei nach der Auflistung in Abschnitt 4.3.1.

Typ	Synapse	Eigenschaft
Typ 1	Hebbsche Synapse	Dieser Hebb-Synapsentyp besitzt neben den allgemeinen Synapseneigenschaften die Lerneigenschaft nach dem modifizierten Hebbschen Lernverfahren. Eine Berücksichtigung einer zeitlichen Verzögerung am Synapseneingang durch ein Tiefpaßfilter besteht hierbei nicht.
Typ 2	Hebbsche Synapse mit Tiefpaßfilter	Dieser Hebb-Synapsentyp besitzt zusätzlich zu den Eigenschaften der einfachen Hebbschen Synapse noch die Tiefpaßeigenschaft am Eingang.
Typ 3	Statische inhibitorische Synapse	Dieser inhibitorische Synapsentyp besitzt die allgemeinen Synapseneigenschaften. Der Synapsenausgang wirkt stets hemmend auf das Zielneuron, das Synapsengewicht ist nicht adaptiv.
Typ 4	Statische inhibitorische Synapse mit Tiefpaßfilter	Dieser inhibitorische Synapsentyp besitzt neben den Eigenschaften des einfachen inhibitorischen Synapsentyps noch eine Tiefpaßeigenschaft am Eingang.
Typ 5	Statische exzitatorische Synapse	Dieser exzitatorische Synapsentyp besitzt die allgemeinen Synapseneigenschaften. Der Synapsenausgang wirkt stets erregend auf das Zielneuron, das Gewicht ist nicht adaptiv.
Typ 6	Statische exzitatorische Synapse mit Tiefpaßfilter	Dieser exzitatorische Synapsentyp besitzt neben den Eigenschaften des einfachen exzitatorischen Synapsentyps noch eine Tiefpaßeigenschaft am Eingang.

Tabelle 5: Typen der verschiedenen Synapsen

Netzstrukturliste im DEL-Format
...
5,-1,1,0,0,1.0,0,0,1,0,0,0,0.0,0.0, 0 , 0 , 0 ,0
5, 1 ,2,0,0,0.2,0,0,1,0,0,0,0.0,0.0, 0 , 0 , 0 ,0
1, 1 ,2,0,0,0.0,0,0,1,0,0,0,0.1,0.1,100,1000,30000,0
3, 2 ,1,0,0,0.5,0,0,1,0,0,0,0.0,0.0, 0 , 0 , 0 ,0
...

Tabelle 6: Beispiel einer Netzstrukturliste

4.3.4 Die Eigenschaften der sechs Synapsenklassen

Jede Synapsenklasse hat individuelle Eigenschaften. Die Netzliste ist so aufgebaut, daß in jeder Zeile alle Spalten eingetragen werden, auch wenn sie nicht berücksichtigt werden. In der folgenden Liste ist beschrieben, welche Spalten eingelesen werden. Die Reihenfolge der entsprechende Einträge richtet sich nach der Auflistung in Abschnitt 4.3.2.

Eigenschaft	**Variable**	**1 Hebb**	**2 Hebb LPF**	**3 inhi.**	**4 inhi. LPF**	**5 exzi.**	**6 exzi. LPF**
Allgemein	PreSynaptischerIndex	ja	ja	ja	ja	ja	ja
	NeuronIndex	ja	ja	ja	ja	ja	ja
	VetoIndex	ja	ja	ja	ja	ja	ja
	VetoSignal	ja	ja	ja	ja	ja	ja
	VetoSchwelle	ja	ja	ja	ja	ja	ja
	PreSynaptischesSignal	ja	ja	ja	ja	ja	ja
	PostSynaptischeSignal	ja	ja	ja	ja	ja	ja
	Gewicht	ja	ja	ja	ja	ja	ja
Tiefpaß	STM	nein	ja	nein	ja	nein	ja
	TauRisingSTM	nein	ja	nein	ja	nein	ja
	TauFallingSTM	nein	ja	nein	ja	nein	ja
Lernen	SchwellwertLPF / INP	ja	ja	nein	nein	nein	nein
	SchwellwertOTH	ja	ja	nein	nein	nein	nein
	TauLernen	ja	ja	nein	nein	nein	nein
	TauVergessen	ja	ja	nein	nein	nein	nein
	TauLangzeitVergessen	ja	ja	nein	nein	nein	nein

Tabelle 7: Relevanz der Eigenschaften der jeweiligen Synapsenklassen

4.4 NetzEditor - Eine Software zur Bearbeitung der Listen

Zur Erstellung und Bearbeitung der Neuronen- und der Netzstrukturliste reicht ein normaler Texteditor aus, der standardmäßig in allen Betriebssystemen vorzufinden ist. Bei großen künstlichen neuronalen Netzen wächst die Anzahl der Zeilen in der Neuronenliste mit der Anzahl der Neuronen im zu simulierenden neuronalen Netz. Entsprechend wächst die Anzahl der Zeilen der Netzstrukturliste mit der Anzahl der Synapsen. Obwohl die Reihenfolge der Zeilen in den einzelnen Listen keine Rolle spielt, geht bei größeren Netzen schnell die Übersicht über die genaue Netzstruktur verloren. Für die schnelle und effiziente Erstellung und Bearbeitung der Neuronen- und Netzstrukturliste dient der im folgenden beschriebene *NetzEditor*. Der *NetzEditor* ist eine Software, die wie das Simulationstool auch, in der Programmiersprache C++ realisiert ist und für die Betriebssysteme IBM OS/2, Microsoft Windows 95 und Microsoft Windows NT programmiert ist. Eine Portierung auf andere Plattformen hängt zum Zeitpunkt der Drucklegung von der Verfügbarkeit der entsprechenden Klassenbibliotheken für die entsprechenden Systeme ab. Da die Software auf der benutzerfreundlichen grafischen Oberfläche basiert, ist sie für die drei Betriebssysteme jeweils angepaßt.

4.4.1 Hauptfenster des Netzeditors

Der NetzEditor wird durch den Start des Programms „netzedit.exe"gestartet.

Das Hauptfenster besteht aus der Menüleiste, der Neuronen- und der Synapsenliste. Diese Listen können jeweils durch die drei Buttons *Hinzufügen*, *Bearbeiten* und *Entfernen* verändert werden. Jede Liste besitzt eine Anzeige, auf der die aktuelle Anzahl der Einträge in der Liste abgelesen werden kann. Mit

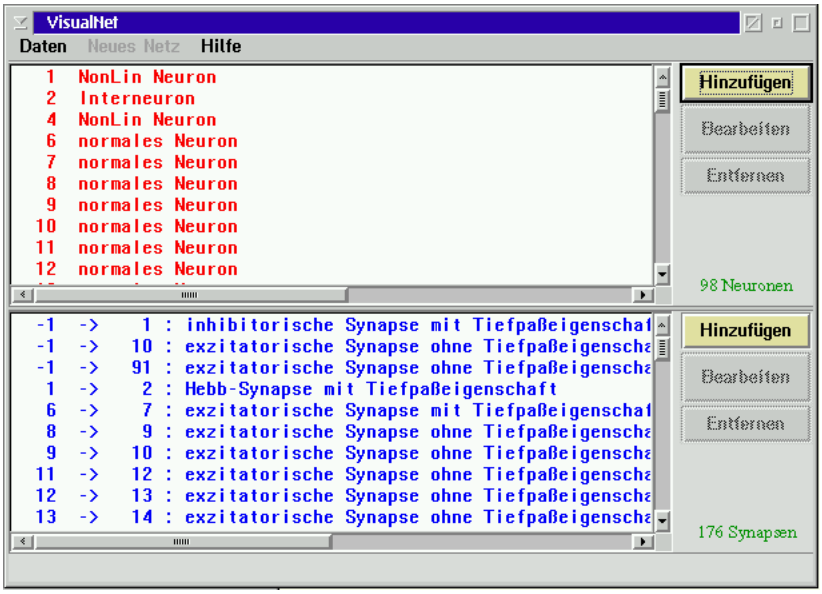


Abbildung 9: Eingabemaske Hauptfenster

Hilfe des Pushbuttons *Hinzufügen* kann jeweils ein Eintrag in der Liste hinzugefügt werden. Durch Betätigen dieses Bedienelementes erscheint ein neues Arbeitsfenster, mit dessen Hilfe das neue Neuron bzw. die neue Synapse bearbeitet wird. Die einzelnen Einträge in die Listen können durch einen Mausklick auf die entsprechende Zeile markiert werden. Durch Betätigen des Pushbuttons *Bearbeiten* kann jeweils der aktuelle Eintrag in der Liste verändert werden. Die gleiche Funktionalität steht dem Benutzer durch einen Doppelklick auf den entsprechenden Eintrag der Listen zur Verfügung. Der *Entfernen*-Pushbutton löscht eine markierten Eintrag aus der Liste.

4.4.2 Öffnen einer Neuronen- und einer Netzstrukturliste

Die Neuronenliste und die Netzstrukturliste können mit Hilfe des Menüs *Daten-Öffnen* in die Arbeitsfenster eingeladen werden. Nach Betätigung des Menüeintrags erscheint zunächst das Auswahlfenster *Öffnen einer Neuronenliste*. In diesem Arbeitsgebiet wird eine Neuronenliste ausgewählt. Unter *Dateiname:* wird der Name der Neuronenliste eingetragen. Die darunterstehenden Datenbereiche dienen der Auswahl des Dateityps, des Laufwerks, des Verzeichnisses und der Datei selbst. Die Betätigung des *Abbruch*-Pushbuttons führt zu einem vollständigen Abbruch des Einlesevorgangs. Der *OK*-Pushbutton dient zur Bestätigung der Auswahl und führt zum Auswahlfenster *Öffnen einer Netzstrukturliste*. Dieses Menü ist genauso aufgebaut wie das vorherige Auswahlfenster auch und dient der Auswahl der Synapsenliste. Nach bestätigter Auswahl wird die Neuronen- und die Netzstrukturliste im Hauptfenster des *NetzEditors* angezeigt und kann bearbeitet werden. Die Bearbeitung erfolgt, wie schon eingangs erwähnt, durch die Pushbuttons *Hinzufügen*, *Bearbeiten* und *Entfernen*.

4.4.3 Speichern einer Neuronen- und einer Netzstrukturliste

Die Neuronenliste und die Netzstrukturliste können mit Hilfe des Menüs *Daten-Speichern unter* abgespeichert werden. Nach Betätigung des Menüeintrags erscheint zunächst das Auswahlfenster *Speichern der Neuronenliste*. Hier wird die Bezeichnung der Neuronenliste festgelegt, unter der sie endgültig gespeichert bzw. zwischengespeichert werden soll. Unter *Dateiname:* wird der Name der Neuronenliste eingetragen. Die darunterstehenden Datenbereiche dienen der Auswahl des Dateityps, des Laufwerks, des Verzeichnisses und der Datei selbst. Die Betätigung des *Abbruch*-Pushbuttons führt zu einem vollständigen Abbruch des Ausgabevorgangs. Der *OK*-Pushbutton dient zur Bestätigung des Speichervorgangs und führt zum Fen-

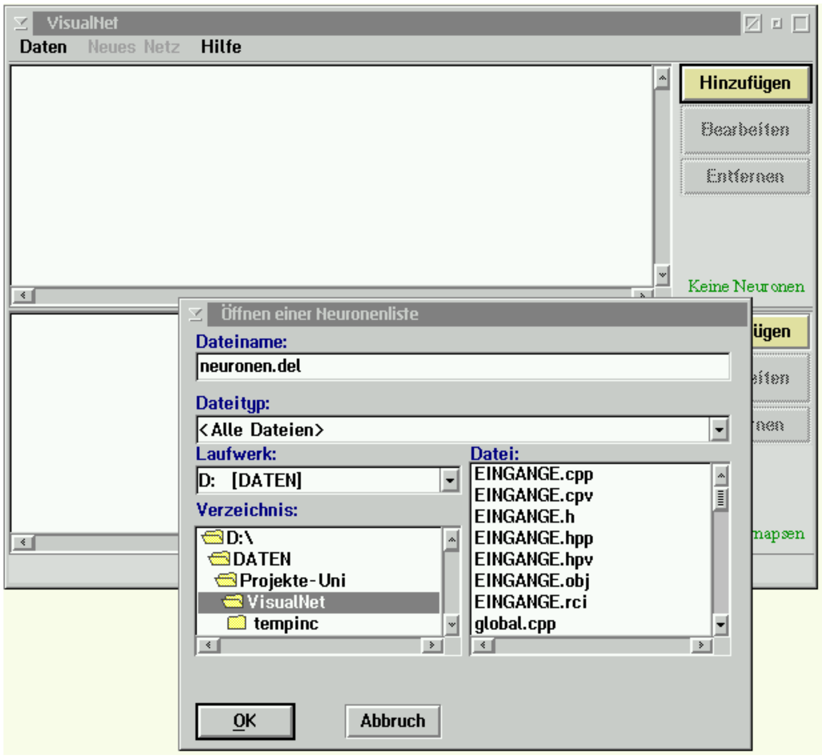


Abbildung 10: Eingabemaske Öffnen einer Datei

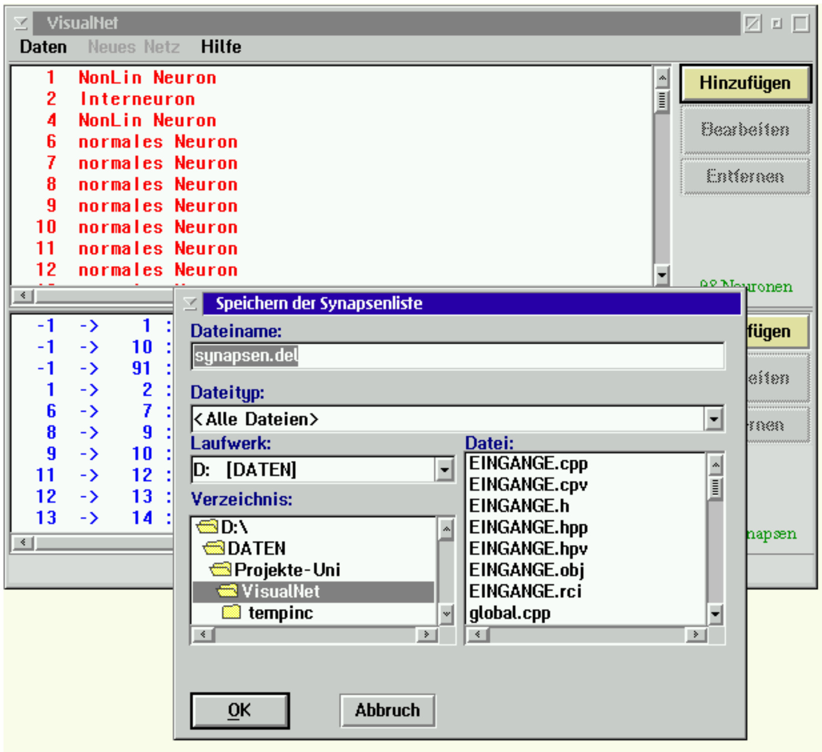


Abbildung 11: Eingabemaske Speichern der Daten

ster *Speichern der Netzstrukturliste*. Dieses Arbeitsfenster ist genauso aufgebaut wie das vorherige und dient der Bezeichnung der Synapsenliste. Nach dem Speichervorgang wird die Neuronen- und die Netzstrukturliste im Hauptfenster des *NetzEditors* weiterhin angezeigt und kann weiterhin bearbeitet werden. Wird jedoch der Menüeintrag *Daten-Speichern und Exit* aufgerufen, so führt dies nach dem Speichervorgang zur Beendigung der Anwendung.

4.4.4 Bearbeiten eines Neurons

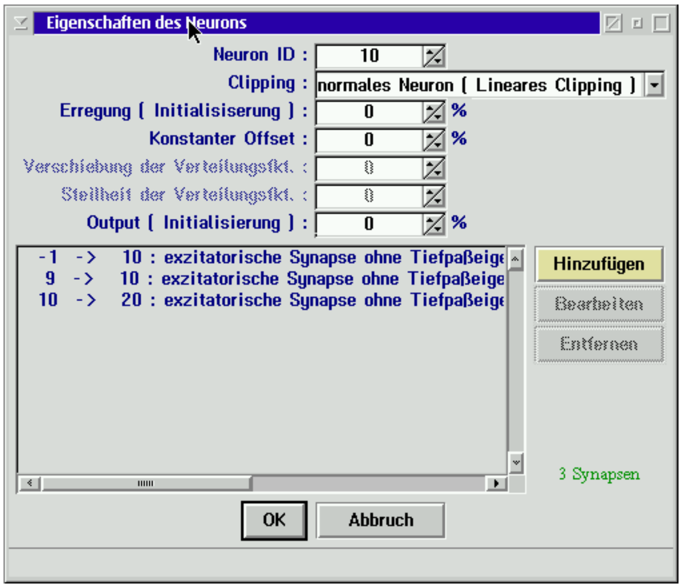


Abbildung 12: Dialogfenster zur Bearbeitung der Neuroneigenschaften

Im Arbeitsfenster *Eigenschaften des Neurons* wird das jeweilige Neuron konfiguriert. Der obere Teil des Arbeitsfensters beinhaltet die Definition des entsprechenden Neurons und ermöglicht die Konfiguration der Eigenschaften, die unter Abschnitt 4.2.1 beschrieben sind. Der untere Teil des Arbeitsfensters beinhaltet eine Liste aller Synapsen des Neurons und aller Synapsen, die von diesem Neuron zu folgenden Neuronen führen. Die Synapsenliste dieses Arbeitsfensters besteht aus einer Listbox, in der die einzelnen Synapsen aufgeführt werden, drei Pushbuttons und einem Synapsenzähler. Mit Hilfe des Pushbuttons *Hinzufügen* kann jeweils eine neue Synapse der Liste hinzugefügt werden. Die einzelnen Einträge in die Synapsenlisten können durch einen Mausklick auf die entsprechende Zeile markiert werden. Durch Betätigen des Pushbuttons *Bearbeiten* kann die aktuelle Synapse verändert werden. Die gleiche Funktionalität steht dem Benutzer durch einen Doppelklick auf die entsprechende Synapse zur Verfügung. Der *Entfernen*-Pushbutton löscht die markierte Synapse aus der Liste.

4.4.5 Bearbeiten einer Synapse

Im Arbeitsfenster *Eigenschaften der Synapse* wird die jeweilige Synapse konfiguriert. Die Einträge dieses Fensters beinhalteten die Eigenschaften der Synapse, die in Abschnitt 4.3.1 beschrieben sind.

4.4.6 Der Quellcode

Der Quellcode des Netzeditors befindet sich im Anhang. Die einzelnen Dateien haben die in Tab. 8 zusammengefaßten Inhalte.

Bezeichnung	Funktion der Datei	Betriebssystem
GLOBAL.cpp	Code globaler Oberflächenfunktionen	Unabhängig
GLOBAL.hpp	Def. globaler Oberflächenfunktionen	Unabhängig
NEURON.cpp	Klassenimplementation von NEURON	IBM OS/2
NEURON.cpp	Klassenimplementation von NEURON	Windows 95/NT
NEURON.cpv	Benutzer definierter Code der Klasse NEURON	Unabhängig
NEURON.h	Konstanten Deklaration der Klasse NEURON	IBM OS/2
NEURON.h	Konstanten Deklaration der Klasse NEURON	Windows 95/NT
NEURON.hpp	Deklaration der Klasse NEURON	IBM OS/2
NEURON.hpp	Deklaration der Klasse NEURON	Windows 95/NT
NEURON.hpv	Deklaration der Klasse NEURON	Unabhängig
SYNAPSE.cpp	Klassenimplementation von SYNAPSE	IBM OS/2
SYNAPSE.cpp	Klassenimplementation von SYNAPSE	Windows 95
SYNAPSE.cpp	Klassenimplementation von SYNAPSE	Windows NT
SYNAPSE.cpv	Benutzer definierter Code der Klasse SYNAPSE	Unabhängig
SYNAPSE.h	Konstanten Deklaration der Klasse SYNAPSE	IBM OS/2
SYNAPSE.h	Konstanten Deklaration der Klasse SYNAPSE	Windows 95/NT
SYNAPSE.hpp	Deklaration der Klasse SYNAPSE	IBM OS/2
SYNAPSE.hpp	Deklaration der Klasse SYNAPSE	Windows 95/NT
SYNAPSE.hpv	Deklaration der Klasse SYNAPSE	Unabhängig
VBMAIN.cpp	Hauptprogramm der Klasse VISUAL	IBM OS/2
VBMAIN.cpp	Hauptprogramm der Klasse VISUAL	Windows 95/NT
VISUAL.cpp	Klassenimplementation von VISUAL	IBM OS/2
VISUAL.cpp	Klassenimplementation von VISUAL	Windows 95/NT
VISUAL.cpv	Benutzer definierter Code der Klasse VISUAL	Unabhängig
VISUAL.h	Konstanten Deklaration der Klasse VISUAL	IBM OS/2
VISUAL.h	Konstanten Deklaration der Klasse VISUAL	Windows 95/NT
VISUAL.hpp	Deklaration der Klasse VISUAL	IBM OS/2
VISUAL.hpp	Deklaration der Klasse VISUAL	Windows 95/NT
VISUAL.hpv	Deklaration der Klasse VISUAL	Unabhängig

Tabelle 8: Partitionierung des Quellcodes

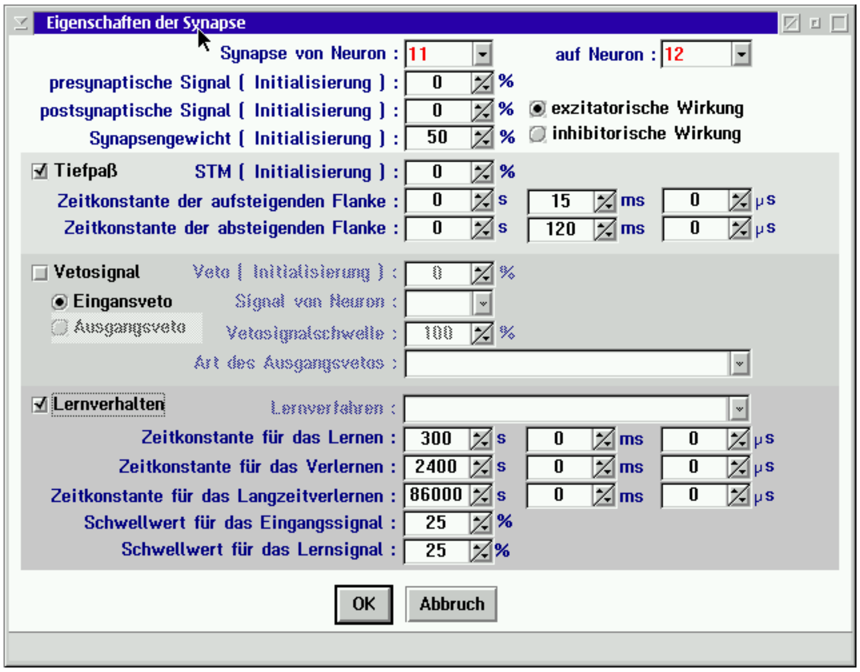


Abbildung 13: Dialogfenster zur Bearbeitung der Synapseneigenschaften

5 Simulator

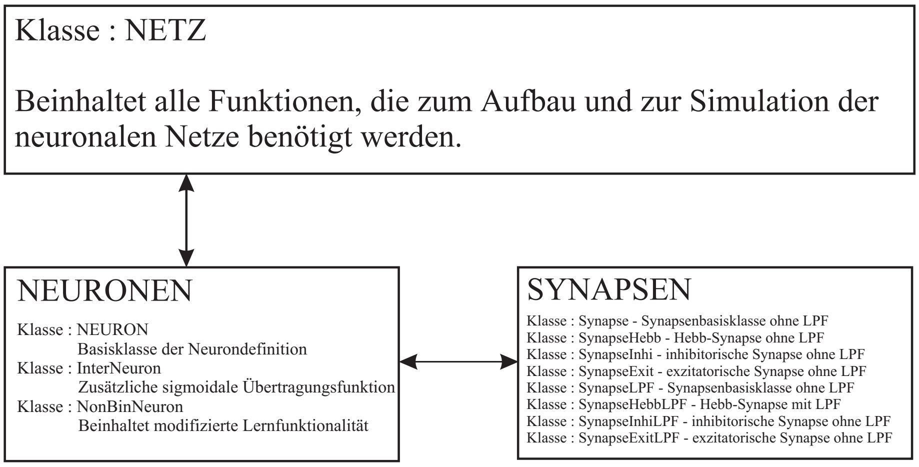


Abbildung 14: Struktur der Klassen

Dieses Kapitel beschäftigt sich mit der Klassenbibliothek des Simulators. Das Modul ist in der Programmiersprache C++ realisiert und stellt alle Funktionen zur Verfügung, die zur Softwaresimulation von künstlichen neuronalen Netzen auf Basis des modifizierten Hebbschen Lernverfahrens benötigt werden. Eines der Ziele ist es, ein universelles Simulationswerkzeug zur Verfügung zu stellen, mit Hilfe dessen komplexe Netzstrukturen auf Basis des modifizierten Hebbschen Lernverfahrens simuliert werden können, ohne Einschränkungen in Bezug auf die Zahl der Verbindungen, die Struktur der Verbindungen, die Zahl der Synapsen pro Neuron, den Parametern einzelner Synapsen etc. zu unterliegen. Darüberhinaus zeichnet sich der Simulator durch seine gute Verständlichkeit sowohl in der Handhabung als auch in der Programmierung der Simulation aus. Die Handhabung der Simulationsklassen ist so übersichtlich gestaltet worden, daß jeder Bediener auch ohne Vorkenntnisse in der Lage sein sollte, eine Simulation zu entwerfen, durchzuführen und entsprechende Änderungen vorzunehmen. Bei neuen Ideen oder anderen Gesichtspunkten sollte eine Simulation ferner leicht erweiterbar sein, ohne über die Spezialkenntnisse des Programmierers zu verfügen.

5.1 Klassen

Ein neuronales Netz besteht aus Neuronen, die durch Synapsen miteinander verbunden sind. Analog zu dieser biologischen Inspiration sind folgende C++-Klassen entstanden.

- **Netz**
  Die C++-Klasse Netz stellt alle Funktionen zur Verfügung, um Netze in einem Anwendungsprogramm aufzubauen und zu simulieren. Unter anderem existieren Funktionen, welche die Neuronenliste und die Netzstruktur aus der offenen Schnittstelle auslesen und im Speicher des Simulators aufbauen.

- **Neuronen**
  Die verschiedenen Neuronen-Klassen stellt künstliche Neuronen und deren Eigenschaften zur Verfügung. Da diese Neuronen-Klassen in der Klasse Netz benutzt werden, sind sie nur softwareintern und zum allgemeinen Verständnis von Bedeutung. Alle Eigenschaften und Funktionen, die einem künstlichen Neuron zugeschrieben werden, sind in dieser Klasse verallgemeinert zusammengefaßt.

- **Synapsen**
  In den Synapsen-Klassen werden alle künstlichen Synapsen zur Verfügung gestellt. Da diese Klassen in der Klasse Neuronen verwendet werden, sind sie ebenfalls nur softwareintern und zum allgemeinen Verständnis von Bedeutung. Alle Eigenschaften[14] und Funktionalitäten, die die künstlichen Synapsen betreffen, sind in diesen Klassen verallgemeinert.

Die Klassen Neuronen und Synapsen werden von der Klasse Netz verwendet. Um eine konkrete Anwendung zu programmieren, ist es lediglich erforderlich, auf der Klasse Netz aufzubauen. In diesem Kapitel wird folglich nur die Klasse Netz erläutert.

5.2 Kon- und Destruktoren der Klasse Netz

Die Simulationsbibliothek stellt zwei Konstruktoren und einen Destruktor der Klasse Netz zur Verfügung.

Rückgabewert	Konstruktor	Übergabeparameter
void	Netz	(void)
void	Netz	(int Schritt, int Debug, double Zeit, const char* Neuronen, const char* Synapsen)
void	Netz	(void)

Tabelle 9: Konstruktoren und Destruktor der Klasse Netz

Der erste der beiden Konstruktoren erzeugt eine Instanz der Klasse Netz. Der zweite Konstruktor setzt zusätzlich die internen Variablen SchrittAnzahl auf Schritt, Info auf Debug und ZeitSkala auf Zeit. Die Angabe des Parameters Zeit ist hier zwingend erforderlich, da er zur Überprüfung der Einhaltung des Abtasttheorems dient und darüberhinaus zu Normierung der Zeitskala verwendet wird. Zusätzlich zu den oben genannten Variablen werden die Funktionen getNeuronen mit dem Übergabeparameter Neuronen und getVerbindungen mit dem Parameter Synapsen aufgerufen. Dieser Konstruktor erlaubt folglich einen vollständigen Aufbau des gesamten Netzes im Simulator. Der Destruktor ruft die Funktionen delVerbindungen und delNeuronen auf und löscht somit das Netz aus dem Speicher des Simulationsrechners.

5.3 Ein- und Ausgabe der Neuronenliste

Jede Simulation beginnt in der Regel mit dem Einlesen der Neuronenliste. Die C++-Klasse Netz stellt die in Tab. 10 dargestellten Funktionen zum Bearbeiten der Neuronenliste zur Verfügung.

Rückgabewert	Funktionsname	Übergabeparameter
void	getNeuronen	(char *DateiName)
void	delNeuronen	(void)
void	putNeuronen	(char *DateiName, int Nachkommastellen)

Tabelle 10: Funktionen zur Manipulation der Neuronenliste

Die Neuronenliste wird mit der Funktion getNeuronen aus einer Datei eingelesen, im Speicher aufgebaut und initialisiert. Mit der Funktion putNeuronen wird die Neuronenliste in der Datei abgespeichert, und mit Hilfe der Funktion delNeuronen kann die Neuronenliste aus dem Speicher gelöscht werden.

[14] So z. B. die Tiefpaßeigenschaft einiger Synapsentypen.

5.4 Ein- und Ausgabe der Netzliste

Nachdem die Neuronenliste im Speicher aufgebaut ist, muß die Netzstruktur eingelesen werden. Damit werden die verschiedenen Synapsen initialisiert und können anschließend entsprechend den Vorgaben aus der Synapsensteuerdatei verbunden werden. Die C++-Klasse Netz stellt die folgende Funktionen zur Bearbeitung der Netzstruktur zur Verfügung

Rückgabewert	**Funktionsname**	**Übergabeparameter**
void	getVerbindungen	(char *DateiName)
void	delVerbindungen	(void)
void	putVerbindungen	(char *DateiName, int Nachkommastellen)

Tabelle 11: Funktionen zur Manipulation der Netzstruktur

Die Netzliste, d. h. die Verbindungen zwischen den Neuronen und den Eingangsleitungen werden durch die Funktion getVerbindungen aus einer Datei ausgelesen, im Speicher aufgebaut und initialisiert. Die Liste wird durch putVerbindungen in eine Datei ausgegeben und kann mit Hilfe von delVerbindungen aus dem Speicher wieder gelöscht werden.

5.5 Simulationsschritt

Die eigentliche Simulation besteht aus einer großen Anzahl einzelner Simulationsschritte. In jedem Simulationsschritt werden dabei zunächst die Eingangsleitungen des neuronalen Netzes mit Daten beschickt. Zur Übergabe eines externen Eingangswertes auf das Netz und somit auf die jeweilige Eingabeleitung steht die Funktion ExternerInput zur Verfügung. Mit der Funktion Simulationsschritt wird das gesamte Netz einmal iteriert. Zum Auslesen des Ausgabewertes eines bestimmten Neurons steht die Funktion ExternerOutput zur Verfügung.

Rückgabewert	**Funktionsname**	**Übergabeparameter**
void	ExternerInput	(int Eingang, double Wert)
void	Simulationsschritt	(void)
double	ExternerOutput	(int NeuronenIndex)
double	putGewicht	(int PreIndex , int NeuronenIndex)

Tabelle 12: Funktionen zur Beeinflussung der Simulation

5.5.1 Funktion ExternerInput

MyNet→ExternerInput(1, 1.0);
MyNet→ExternerInput(2, 1.0);
MyNet→ExternerInput(3, 1.0);
MyNet→ExternerInput(4, 1.0);
MyNet→ExternerInput(5, 1.0);

Tabelle 13: Beispiel externer Eingaben

Bei einem Simulationsschritt müssen zunächst die sensorischen Daten, d. h. die des externen Inputs auf das Netz gegeben werden. Sollen beispielsweise die ersten fünf Eingänge mit dem Signal 1.0 beschickt werden, so müssen die nebenstehenden Anweisungen durchgeführt werden.

5.5.2 Funktion Simulationsschritt

Der Simulationsschritt durchläuft zweimal die Neuronenliste. Die Aufteilung auf zwei Durchläufe ist erforderlich, da die Updatereihenfolge der Neuronen in einem Simulationsschritt keine Rolle spielen darf. Würde die Neuronenliste nur einmal durchlaufen werden, so würde die Reihenfolge der Neuronen eine Rolle spielen, da die Berechnungen auf einem normalen Computer sequentiell ablaufen. Um ein einzelnes Neuron zu berechnen, müssen alle Synapsen des Neurons berechnet werden. Anschließend muß die Erregung des Neurons aus der Summe der Synapsenausgänge gebildet werden. Der Output des jeweiligen Neurons wird danach aus der Erregung berechnet. Anschließend muß der Output auf alle Synapsen, die auf dem Axon des Neurons liegen, weitergeleitet werden. Fände hierbei die Übergabe des Outputs im gleichen Durchlauf durch die Neuronenliste wie die Berechnung der Erregung statt, so könnten Eingabedaten in diesem Simulationsschritt berücksichtigt werden, die erst für den nächsten Schritt vorgesehen waren[15]. Dem Problem der Updatereihenfolge kann begegnet werden, indem in einem Simulationsschritt zunächst die Erregung und der Output aller Neuronen berechnet wird. Im gleichen Durchlauf lernen auch alle Hebb-Synapsen. Nach diesem Durchlauf durch die Neuronenliste wird erneut die Neuronenliste durchlaufen, wobei lediglich die entsprechenden Outputdaten der einzelnen Neuronen auf die folgenden Synapsen übertragen werden.

5.5.3 Funktion ExternerOutput

Die Funktion ExternerOutput liest aus dem angegebenen Neuron den Output aus. Dieser kann dann zur Weiterverarbeitung genutzt werden, wie z. B. zur Ansteuerung von Aktoren oder als Eingangssignale für ein System zur Emulation einer Umgebung.

5.5.4 Funktion putGewicht

Die Funktion putGewicht liest aus der Hebb-Synapse des entsprechenden Neurons das Gewicht aus. Die Übergabeparameter PreIndex und NeuronenIndex bezeichnen hierbei die Hebb-Synapse. Existieren mehrere Hebb-Synapsen mit den gleichen Indizes, so kann die Funktion zwischen diesen nicht unterscheiden.

5.6 Netzvariablen

Ein wichtiger Netzparameter ist die SchrittAnzahl. Die Variable gibt an, wieviele Simulationsschritte durchgeführt wurden. Für die Fehleranalyse steht die Variable Info zur Verfügung. Wenn die Variable Info auf Eins (verbose mode) gesetzt ist, gibt das Programm jeden wichtigen Schritt auf der Standardausgabe aus. Die Simulation wird in diesem Betriebsmodus durch die Ausgabe der Informationen stark verlangsamt, gibt jedoch Aufschluß über mögliche Fehlerquellen.

Rückgabewert	Variablenname
int	SchrittAnzahl
int	Info

Tabelle 14: Erweiterte Simulationsparameter

[15] Als Beispiel soll hier angenommen werden, daß der Output des ersten Neurons auf das zweite Neuron geschaltet wird. Bei einem Simulationsschritt würde zunächst das erste Neuron berechnet werden und der Output des ersten Neurons würde das zweite Neuron erreichen, bevor das zweite Neuron berechnet wird und so dessen eigentlich notwendigen Berechnungsschritt übergehen. Dies wäre nicht relevant, wenn das zweite Neuron vor dem ersten Neuron in dem Simulationsschritt berechnet würde. Die Updatereihenfolge spielt folglich eine bedeutende Rolle.

Für die Echtzeitsimulation ist die Variable ZeitSkala erforderlich. Die Variable ZeitSkala wird in Sekunden angegeben und sollte der durchschnittlichen Simulationszeit pro Simulationsschritt entsprechen. Um die private Variable ZeitSkala zuzuweisen, steht eine Funktion setZeitSkala zur Verfügung. Die Funktion setZeitSkala überprüft zudem die Einhaltung des Abtasttheorems. Die ZeitSkala darf nie größer als TauMin[16] werden. Da die Variable TauMin erst beim Einlesen der Netzstruktur, d. h. durch die Funktion getVerbindungen zugewiesen wird, macht der Aufruf der Funktion setZeitSkala erst nach dem Einlesen des Netzes Sinn.

Rückgabewert	**Funktionsname**	**Übergabeparameter**
void	setZeitSkala	(double Skala)

Tabelle 15: Manipulation für Echtzeitsimulationen

[16] Die private Variable TauMin wird beim Einlesen der Netz- bzw. Neuronenstruktur ermittelt. Sie ist die kleinste Zeitkonstante aus den beiden Dateien.

6 Anwendungssoftware

Die Simulation einer speziellen Anwendung ist sehr einfach gehalten. Dazu ist lediglich die Erstellung eines kurzen C++-Programmes nötig, das die Klassenbibliotheken des Simulators verwendet. Die Datenstruktur der dynamischen Neuronen ist für den Anwender hierbei unerheblich, da das Anwendungsprogramm nur einzelne Funktionen der Simulationsklassenbibliotheken verwendet. Die Simulation selbst, die Eingaben und die Ausgaben werden durch Funktionen der Simulationsbibliothek bereitsgestellt und gesteuert. Nach der Erstellung des entsprechend angepaßten Anwendungssimulators können anschließend verschiedenste Netze mit dem fertigen Programm simuliert werden. Die jeweilige Netzstruktur wird dabei mit Hilfe der offenen Schnittstelle definiert.

6.1 Aufbau eines konkreten Netzes

Anhand eines Beispiels soll demonstriert werden, wie einfach die Simulation eines dynamischen neuronalen Netzes mit Hilfe des hier erläuterten Simulationstools ist. Als Beispiel soll an dieser Stelle das Netz aus Abb. 15 als Basis dienen.

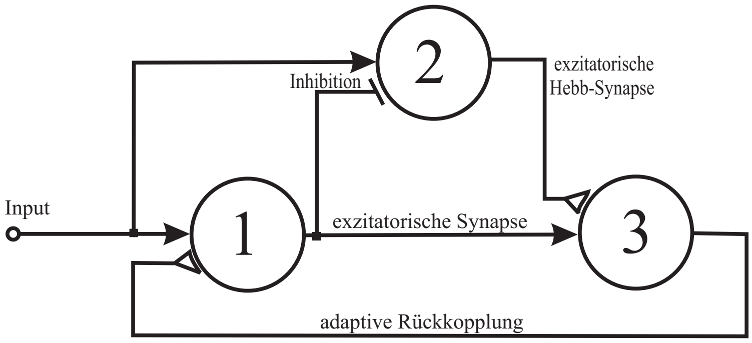


Abbildung 15: Ein einfaches Beispielnetz

Anhand der grafischen Darstellung dieses Netzes wird nun mit Hilfe des *NetzEditors* das Netz in eine konkrete Neuronenliste und eine Netzstrukturliste transformiert. Jede dieser Listen kann hierbei abweichend auch mit einem beliebigen ASCII-Texteditor erstellt werden.

Die Neuronenliste entspricht dann dem in Tab. 16 dargestellten DEL-Format, während die Netzstrukturliste in Tab. 17 dargestellt ist.

Index,	**Typ**,	**Erregung**,	**Offset**,	**Verschiebung**,	**Steilheit**,	**Output**,	**Simulationsschritt**
1 ,	1 ,	0.0 ,	0.0 ,	0.0 ,	0.0 ,	0.0 ,	0
2 ,	1 ,	0.0 ,	0.0 ,	0.0 ,	0.0 ,	0.0 ,	0
3 ,	1 ,	0.0 ,	0.0 ,	0.0 ,	0.0 ,	0.0 ,	0

Tabelle 16: Neuronenliste im DEL-Format

Zur endgültigen Simulation wird nur noch eine kleines Anwendungsprogramm benötigt, das den eigentlichen Simulationsablauf steuert und entsprechende Ein- und Ausgaben bewirkt, wie es im folgenden Abschnitt erläutert wird.

6.2 Typischer Quellcode des Anwendungsprogramms

Der Grund für die zusätzliche Implementierung eines Anwendungsprogrammes zu den vorhandenen C++-Klassen liegt in der Vielfalt der Einsatzgebiete. Jedes Programm paßt die C++-Klassen an die entsprechenden Simulationsrandbedingungen an, so daß sie für viele verschiedene Versuche weiterverwendet werden

Typ	PreSynaptischerIndex	NeuronIndex	PreSynaptischesSignal	PostSynaptischeSignal	Gewicht	VetoIndex	VetoSignal	VetoSchwelle	STM	TauRisingSTM	TauFallingSTM	SchwellwertLPF/INP	SchwellwertOTH	TauLernen	TauVergessen	TauLangzeitVergessen	SchrittAnzahl
5	-1	1	0	0	0.5	0	0	1	0	0	0	0.0	0.0	0	0	0	0
5	-1	2	0	0	0.5	0	0	1	0	0	0	0.0	0.0	0	0	0	0
3	1	2	0	0	0.5	0	0	1	0	0	0	0.0	0.0	0	0	0	0
1	2	3	0	0	0.0	0	0	1	0	0	0	0.1	0.1	100	10000	1e+06	0
5	1	3	0	0	0.5	0	0	1	0	0	0	0.0	0.0	0	0	0	0
1	3	1	0	0	0.0	0	0	1	0	0	0	0.1	0.1	100	10000	1e+06	0

Tabelle 17: Netzstruktur im DEL-Format

können. Beispielsweise wurde ein Anwendungsprogramm dazu erstellt, Sensordaten eines Fahrzeuges auf einer Carrera-Spielzeugbahn aufzunehmen, an verschiedene neuronale Netze weiterzuleiten, diese zu simulieren und die Simulationsergebnisse in Form einer Stellgröße wieder an das Fahrzeug zurückzuübertragen. Diese Aufgabe unterscheidet sich erheblich von einer weiteren Simulation, die z. B. Zugriffsdaten von Benutzern in einem Rechnerverbund eines Netzwerkes auswerten soll. Ein anderes Anwendungsgebiet dieses Simulationstools ist die Steuerung eines Rollstuhls (4; 5; 6).

In diesem Kapitel soll aber nicht auf die komplexe Problematik einer solchen Anwendung eingegangen, sondern lediglich das Netz aus Abb. 15 mit einem konstanten Eingangssignal beschickt werden. Anschließend soll der zeitliche Verlauf des Hebb-Synapsengewichtes zwischen Neuron 2 und Neuron 3 ermittelt und dargestellt werden. Zur grafischen Ausgabe soll dabei auf das unter dem GNU-Projekt entstandene Tool „gnuplot“(7) zurückgegriffen werden, wobei auch jedes andere Datenvisualisierungstool geeignet ist.

Zunächst wird ein Beispielquellcode angegeben, der auf das neuronale Netz einen konstanten Eingangswert aufschaltet und das Verhalten des zu analysierenden Gewichtes in eine Datei ausgibt.

```
#include <iostream.h>
#include <fstream.h>
#include "netz.hpp"

int main ( void )
{
 int    MaxSimulationsschritte  = 1000;
 double KonstanterEingabeWert   = 0.70 ;

 Netz *NET = new Netz( 0, 0, 1.0, NeuronenListe.DEL, NetzListe.DEL );
 ofstream Datei ( 'output.del', ios::out );
 for ( int i = 0 ; i < MaxSimulationsschritte ; i++ )
  {
  NET->setExternerInput ( -1, KonstanterEingabeWert );
  NET->Simulationsschritt ();
  Datei << i                     << " "
        << NET->putGewicht( 2 , 3 ) << " "
        << endl;
  }
 Datei.close();
 NET->delNeuronen     ( );
 NET->delVerbindungen ( );
 return 0;
}
```

In diesem Beispiel wird zunächst ein Netz definiert und dessen `SchrittAnzahl` und `Info` auf Null gesetzt. Danach werden die Neuronen und die Synapsen eingelesen. In der Schleife werden jeweils die einzelnen Simulationsschritte durchgeführt. In jedem Schritt wird dazu zunächst der erste Eingang mit einem Eingangswert beschickt, darauffolgend ein Simulationsschritt iteriert und die Ausgabewerte in eine Datei ausgegeben. Wird es erforderlich, nicht nur den Zustand eines Gewichtes, sondern zusätzlich den Zustand der ersten beiden Neuronen zu protokollieren, so ist der Ausgabeteil

```
Datei << i                        << " "
      << NET->putGewicht( 2 , 3 ) << " "
      << endl;
```

durch den entsprechend angepaßten Ausgabeteil zu ersetzen:

```
Datei << i                        << " "
      << NET->putGewicht( 2 , 3 ) << " "
      << NET->ExternerOutput( 1 ) << " "
      << NET->ExternerOutput( 2 ) << " "
      << endl;
```

Dieses Quellcodebeispiel erlaubt nur eine Simulation mit einem festen Eingabewert. Ist das Grenzverhalten des Gewichts in Abhängigkeit vom Eingangswert von Interesse, so ist es erforderlich, mit einem Quellcode mehrere Simulationen durchzuführen. Der Programmcode muß folglich um eine Schleife erweitert werden, bei der jeweils der Eingabewert inkrementiert wird.

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "netz.hpp"

int main ( void )
{
 double a,n;
 char   * quotation = "\"";
 char   * Komma     = ",";
 char   * pfad;

 cout << "Schrittanzahl : ";
 cin  >> n ;

 ofstream Datei ( "output.del", ios::out );
 a = 1.0 / n;
 double i = 0.0;
 while ( i <= 1.0 + a )
   {
   Netz *NET = new Netz (0,0,1.0,"neuronen.del","synapsen.del");
   for ( int j = 0; j < 100000 ; j++ )
     {
     NET->ExternerInput ( - 1 , i );
     NET->Simulationsschritt ();
     }
   i += a;
   Datei4.precision( 30 );
   Datei4 << i << " " << NET->putGewicht ( 2, 3 ) << " " << endl;
   NET->delNeuronen     ( );
   NET->delVerbindungen ( );
   }
 Datei << endl;
 Datei.close () ;

 ofstream GnuPlotDatei ( "test.gnu" , ios::out );
 GnuPlotDatei.precision( 30 );
 GnuPlotDatei << "set parametric" << endl << "set yrange [0:1]" << endl;
 pfad = "D:\\daten\\projekte-uni\\tests\\irregular\\test0003\\output.del";
 GnuPlotDatei << "plot " << quotation << pfad << quotation <<
                 " u 1:1 with lines "
              << ", "    << quotation << pfad << quotation << "
                   u 1:2 with lines " << endl;
 GnuPlotDatei << "pause -1" << endl;
 GnuPlotDatei.close ();

 return 0;
}
```

Von der Anwendung wird zusätzlich zur Simulation eine Ausgabe-Datei erzeugt, die die direkte grafische Visualisierung des Simulationsergebnisses mit Hilfe von „gnuplot“erlaubt (siehe Abb. 16 und 17). In Abb. 16 ist das transiente Verhalten des Netzes aus dem einfacheren Quellcode aufgezeigt.

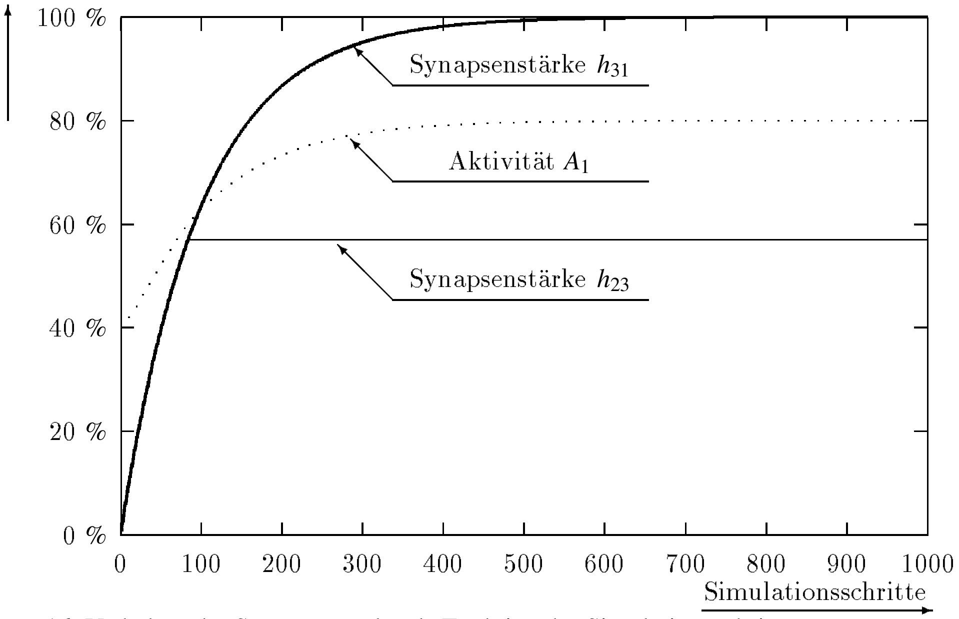


Abbildung 16: Verhalten der Synapsenstärke als Funktion der Simulationsschritte
– Synapsenstärke h_{23} als Funktion der Simulationsschritte, - Synapsenstärke h_{31} als Funktion der Simulationsschritte
··· Aktivität A_1 als Funktion der Simulationsschritte

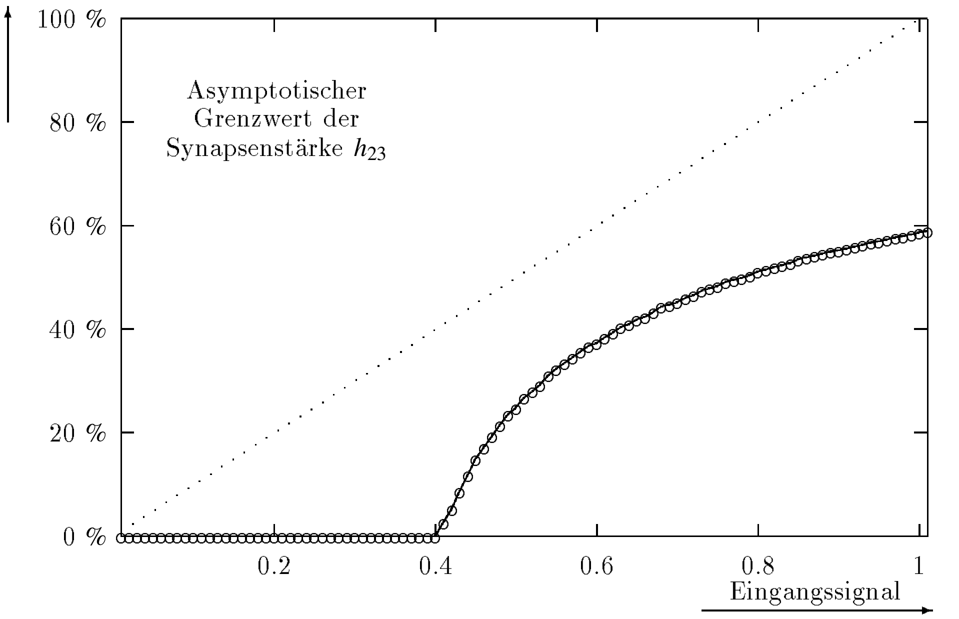


Abbildung 17: Verhalten der Grenzwerte als Funktion des Inputsignals
··· Konstanter Input auf die ersten zwei Neuron, ∘ Grenzwert der Synapsenstärke h_{23}

Mit der hier analysierten Netzstruktur (gem. Abb. 15) wird demonstriert, wie durch dynamische Synapsen nichtlineare Gleichgewichtszustände erreicht werden können. Für dieses Beispiel werden die dynamischen Synapsengewichte mit einer Lernkonstanten $\tau_L = 10^2$ Simulationsschritten belegt. Die Zeitkonstante des Verlernens τ_V wird um einen Faktor 10^2 größer gewählt als die Lernkonstante. Die Zeitkonstante des Langzeitverlernens τ_{Lv} wird nochmals um den gleichen Faktor vergrößert. Die Hebb-Synapse von Neuron 3 zum Neuron 1 wird hierbei so initialisiert, daß sie weder eine Eingangsschwelle ($\Theta_x = 0$) noch eine Lernschwelle ($\Theta_y = 0$) besitzt. Das Verhalten der zweiten Hebb-Synapse wird durch einen Eingangsschwellwert $\Theta_x = 10$ Prozent und eine Lernschwelle $\Theta_y = 10$ Prozent beschrieben. Bei der ersten Simulation mit dieser Schaltung 15 wird ein konstanter Eingang von $I = 0.80$ verwendet. Die Abb. 16 zeigt, daß nach 85 Simulationsschritten das Synapsengewicht h_{23} einen konstanten Wert von $h_{23} = 0.57$ annimmt. Diese Simulation wird für Eingabewerte zwischen Null und 100 Prozent mit einer Schrittweite von einem Prozent durchgeführt und gegen die jeweils konstante Synapsenstärke h_{23} aufgetragen dargestellt (Abb. 17).

7 Zusammenfassung

Im vorliegenden Bericht wurde ein Softwaretool zur Simulation komplexer neuronaler Netze mit dynamischen Neuronen beschrieben und dessen Funktion erläutert. Das Simulationstool verwendet ein On-Line-Lernverfahren zur elementbasierten autonomen Adaption, das auf dem Hebbschen Lernverfahren beruht. Das zugrundeliegende Modell basiert dabei auf experimentell nachgewiesene Vorgängen in biologischen Nervensystemen.

Sowohl die hohe Dimensionalität möglicher Netzstrukturen, als auch die große Anzahl der Freiheitsgrade für jedes einzelne Neuron erlauben keine analytische Lösung und erfordern so den Einsatz eines numerischen Simulators. Mit Hilfe des hier vorgestellte Software können universell sowohl vorwärtsgekoppelte als auch rückgekoppelte Netze mit individuellen Netzstrukturen simuliert werden.

Das Programm wurde so konzipiert, daß es die größtmögliche Flexibilität aller Neuronen gewährleistet, so daß sowohl Anfangsbedingungen als auch die unterschiedlichen Parameter der verschiedenen Neuronen individuell beeinflußt werden können, ohne dabei Einschränkungen in Hinblick auf Übersichtlichkeit und Funktionalität des Simulators zu unterliegen. Zu diesem Zweck sind textbasierte ASCII-Schnittstellen eingeführt und erläutert worden, die die leichte Bedienbarkeit und Analyse des Systems ermöglichen. Die modulare, objektorientierte Umsetzung des Konzeptes erlaubt zusätzlich die leichte Wartbarkeit für spätere Erweiterungen des Systems. Für den Einsatz in realen Umgebungen ist das System darüberhinaus mit einer Option zur Echtzeitverarbeitung versehen, durch die das System bereits in sensomotorischen Aufbauten erfolgreich verwendet werden konnte.

Zusätzlich zu den C++-Klassen, die die Simulation jedes beliebigen neuronalen Netzes ermöglichen, ist ein *NetzEditor* entstanden, mit dessen grafischer Benutzeroberfläche einzelne Netzstrukturen erstellt und bearbeitet werden können. Die Programmierung eines Frontends zur Generation verschiedener Inputsignale als auch ein visueller Netzeditor für die Eingabe von Netzstrukturen wurden hier nicht vorgestellt, sind jedoch durch die offene Schnittstelle der Software leicht zu implementieren und als Option für spätere Simulationssysteme denkbar. Ferner könnte die Implementierung verschiedener Werkzeuge auf den bisherigen Arbeiten aufbauen wie z. B. für den automatisierten Entwurf von Netzstrukturen auf Basis evolutionären Strategien.

Danksagung

Diese Arbeit wurde im Rahmen des Projektes „Mikroneuron“durch die *Deutsche Forschungsgemeinschaft* (Förderungs-Nr. 379/12) und des *Sonderforschungsbereichs 531* „Design und Management komplexer technischer Prozesse und Systeme mit Methoden der Computational Intelligence“an der Universität Dortmund gefördert.

Literatur

[1] R. BLANKENAGEL, *Entwicklung eines Simulators zur Analyse neuronaler Modelle*, studienarbeit, Universität Dortmund S385, Fakultät für Elektrotechnik, Lehrstuhl Bauelemente der Elektrotechnik, Prof. Dr. Ing. Karl Goser, Januar 1996.

[2] G. BORN, *Noch mehr Dateiformate*, Addison-Wesley, 1995.

[3] ———, *Referenzhandbuch Dateiformate*, Addison-Wesley, 1996.

[4] A. BÜHLMEIER, M. ROSSMANN, K. GOSER, AND G. MANTEUFFEL, *Robot learning in analog neural hardware*, in Artificial Neural Networks: 6th international conference; proceedings / ICANN 96, Bochum, Germany, July 16-19, 1996, vol. 1112 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg New York, ISSN 0302-9743, ISBN 3-540-61510-5, 1996, pp. 311–316.

[5] ———, *Application of a local learning rule in a wheelchair robot*, in Neurap'97, Third International Conference on Neural Networks and their Applications, Marseille, France, March 12-13, 1997, 1997, pp. 177–182.

[6] A. BÜHLMEIER, P. STEINER, M. ROSSMANN, K. GOSER, AND G. MANTEUFFEL, *Hebbian multilayer network in a wheelchair robot*, in Artificial Neural Networks: 7th Int. Conf.; Proc. (ICANN 97), vol. 1327 of Lecture Notes in Computer Science, Springer-Verlag Berlin, Oct 18-10 1997, pp. 727–732.

[7] FREE SOFTWARE FOUNDATION, *GNU's Not Unix! - the GNU Project and the Free Software Foundation (FSF)*. http://www.gnu.org.

[8] IBM, *DATABASE 2 Command Reference for common servers - Version 2 - Appendix C*, IBM, 1995.

[9] ———, *VisualAge C++ Language Reference*, IBM, 1995.

[10] ———, *VisualAge C++ Open Class Library*, IBM, 1995.

[11] C. KEIM, *Biomorphe Netzstrukturen mit dynamischen Neuronen*, Diplomarbeit, Universität Dortmund, Fakultät für Elektrotechnik, Lehrstuhl Bauelemente der Elektrotechnik, Prof. Dr. Ing. Karl Goser, November 1997.

[12] S. B. LIPPMANN, *C++ Einführung und Leitfaden*, Addison-Wesley, 1992.

[13] M. ROSSMANN, A. BÜHLMEIER, G. MANTEUFFEL, AND K. GOSER, *Short- and Long-Term-Dynamics in a Stochastic Pulse Stream Neuron Implemented in FPGA*, in Artificial Neural Networks: 7th Int. Conf.; Proc. (ICANN 97), vol. 1327 of Lecture Notes in Computer Science, Springer-Verlag Berlin, Oct 18-10 1997, pp. 1241–1246.

[14] M. ROSSMANN, C. BURWICK, A. BÜHLMEIER, G. MANTEUFFEL, AND K. GOSER, *Dynamic Hebbian Learning in System Architecture based on High Speed Chip*, in Proc. of the 6th Int. Conf. on Microelectronics for Neural Networks, Evolutionary and Fuzzy Systems, MicroNeuro'97, University of Technology Dresden, ISBN 3-86005-190-3, September 24-26 1997, pp. 251–256.

[15] M. ROSSMANN AND K. GOSER, *Dynamic Artificial Neural Networks for adaptive Control of Velocity of a Track Vehicle*, tech. rep., Department of Microelectronic, Faculty of Electrical Engineering, University of Dortmund, Germany, 1997. in preperation.

[16] M. ROSSMANN, B. HESSE, K. GOSER, A. BÜHLMEIER, AND G. MANTEUFFEL, *Implementation of a biologically inspired neuron model in FPGA*, in Proceedings of the Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, IEEE Computer Society Press, ISBN 0-8186-7373-7, ISSN 1086-1947, February 1996, pp. 322–329.

[17] M. ROSSMANN, T. JOST, K. GOSER, A. BÜHLMEIER, AND G. MANTEUFFEL, *Exponential Hebbian on-line learning implemented in FPGAs*, in Artificial Neural Networks: 6th international conference; proceedings / ICANN 96, Bochum, Germany, July 16-19, 1996, vol. 1112 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg New York, ISSN 0302-9743, ISBN 3-540-61510-5, 1996, pp. 767–772.

[18] M. ROSSMANN, C. KEIM, AND K. GOSER, *Simulationstool für Dynamische Neuronale Netze*, Tech. Rep. 0497, Department of Microelectronic, Faculty of Electrical Engineering, University of Dortmund, Germany, 1997. in German.

[19] B. STROUSTRUP, *Die C++ Programmiersprache*, Addison-Wesley, 1992.