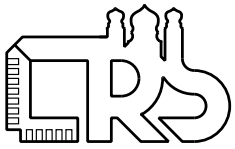
Neuronale Netze in ICONNECT – Theoretische Grundlagen und Modulbeschreibung

K. Ehrnböck, B. Sick

Version 1.00



Lehrstuhl für Rechnerstrukturen

Prof. Dr.-Ing. W. Grass

UNIVERSITÄT PASSAU

Inhaltsverzeichnis

1 Einführung

Spätestens seit der Beschreibung des als Backpropagation bekannten Lernverfahrens 1986 steigt die Bedeutung Neuronaler Netze in industriellen Anwendungen. Dieses Lernverfahren wird zum Training der Gewichte in Mehrlagigen Perzeptren (MLP) mit kontinuierlichen, differenzierbaren Aktivierungsfunktionen verwendet. Bis heute ist MLP das mit großem Abstand am häufigsten Netzparadigma. Es eignet sich für regelungstechnische Anwendungen genauso wie für Klassifikationsaufgaben usw.

MLPs zeigen ein statisches Systemverhalten, in vielen Anwendungsbereichen werden jedoch nichtlineare Modelle mit einem dynamischen Systemverhalten benötigt. D.h., nicht nur einzelne Werte (Muster), sondern sogar Zeitreihen müssen verarbeitet oder generiert werden.

Zwei Arten von Mechanismen ermöglichen eine Verarbeitung von Zeitreihen [UDC+92]:

- *Rekursion* und
- *Speicherfähigkeit*.

Rekursion meint hier eine Rückkopplung mit verzögerter Weitergabe des Wertes um einen Zeitschritt, d.h., Auswirkungen der Rückkopplung sind erst im folgenden Zeitschritt zu beobachten. Bei *voll rekurrenten* Netzen kann ein Neuron Eingangswerte von allen Neuronen eines Netzes (auch von sich selbst) erhalten [Hay94, Zel94]. Werden Einschränkungen irgendeiner Art gemacht, so spricht man von *partiell rekurrenten* Netzparadigmen. Hierzu zählen beispielsweise *Jordan-* und *Elman-Netze* [Elm90, Zel94]. Einen Überblick über verschiedene bekannte rekurrente Paradigmen gibt beispielsweise [Tso98].

Rekursion bedeutet, daß Neuronen unter anderem Informationen über **eigene**, vergangene Aktivierungszustände erhalten. Bei der Verwendung eines Speichermechanismus in *nicht-rekurrenten* Netzen stehen einem Neuron nur Informationen über vergangene Aktivierungszustände **anderer** Neuronen (Vorgängerneuronen) oder über vergangene Eingangsmerkmale zur Verfügung. Im ersten Fall geben Verzögerungselemente (Speicherelemente) die Aktivierung eines Vorgängerneurons um einen oder mehrere Zeitschritte verzögert an einen Nachfolger weiter. Zeitliche Information wird somit intern im Netz repräsentiert (*implizite* zeitliche Information). Im zweiten Fall wird ein gleitender Ausschnitt der Zeitreihe (*sliding window*) simultan an mehrere Eingangsneuronen eines Netzes angelegt. Durch die Kombination von solchermaßen extern repräsentierter zeitlicher Information (*expliziter* zeitlicher Information) und einem statischen Netzparadigma kann ein dynamisches Verhalten des Gesamtsystems erzielt werden. Interne Zeitverzögerungen werden

beispielsweise in Time-Delay-Netzen (auch als FIR-Netze bezeichnet; *FIR: finite impulse response*), adaptiven Time-Delay-Netzen oder Gamma-Netzen verwendet [Hay94, LDL92a, Tso98, Wan93a]. Gleitende Eingabefenster kommen sehr häufig in Kombination mit Mehrlagigen Perzeptren zum Einsatz [UDC+92, Zel94].

Prinzipiell kann fast jedes statische Netzparadigma um einen oder mehrere der genannten Mechanismen erweitert werden, jedoch werden die dadurch entstehenden Netztypen im allgemeinen als eigene Netzparadigmen angesehen (u.a. da sie meist auch spezielle Lernalgorithmen benötigen).

Es gibt auch Netzparadigmen, die Speichermechanismen mit Rekursion kombinieren. Hierzu zählen beispielsweise NARX-Netze (gleitendes Eingabefenster und Rückkopplungen; *NARX: nonlinear autoregressive model with exogenous inputs*) oder IIR-Netze mit IIR-Filtern (*IIR: infinite impulse response*) zwischen aufeinanderfolgenden Neuronen [LGHK97, LHTG96, Wan93a].

Natürlich sind die verschiedenen Mechanismen je nach Art der zu modellierenden zeitlichen Zusammenhänge unterschiedlich gut geeignet: Spielt nur die relative Position eines Musters in einem lokalen Ausschnitt der Zeitreihe eine Rolle (soll also eine kurze Musterfolge translationsinvariant erkannt werden), so sind Netze mit Speichermechanismen vorteilhafter. Kommt es dagegen auf die gesamte Historie einer Zeitreihe an, so sind rekurrente Netze eventuell besser geeignet. Bei der Verschleißüberwachung ist davon auszugehen, daß insbesondere lokale Trends innerhalb eines kurzen Zeitabschnitts bedeutsam sind. Da es zudem bei rekurrenten Netzen, wie bei allen rückgekoppelten Systemen, leicht zu Stabilitätsproblemen kommen kann (insbesondere bei verrauschten Eingangsdaten, wie sie hier vorliegen), werden Netze mit Speichermechanismen zur Verschleißüberwachung gewählt (siehe dazu auch das erste Zitat am Anfang des Kapitels). Dabei sind wiederum Netze mit Verzögerungselementen vorzuziehen, da sie mit einer einfacheren Netzstruktur komplexere zeitliche Zusammenhänge als Netze mit gleitenden Eingabefenstern erfassen können. Mehrlagige Perzeptren mit Eingabefenstern und auch statische Mehrlagige Perzeptren können als Sonderfall von *Time-Delay-Netzen* angesehen werden, die hier zur Verschleißüberwachung verwendet werden. Im Gegensatz zu rekurrenten Netzen sind Time-Delay-Netze sogar zur Rauschunterdrückung gut geeignet [Hay94, LLD93b].

Der vorliegende Bericht beschreibt die im Signalverarbeitungstool ICONNECT [SBF+98a, SBF+98b] vorhandenen Möglichkeiten zur Modellbildung mit Neuronalen Netzen. Zunächst werden in Abschnitt 2 die Netzparadigmen TDNN, MLP-sw und MLP ausführlich beschrieben. Dann werden in Abschnitt 3 verschiedene Lernalgorithmen zum Training dieser Netze vorgestellt. Abschnitt 4 macht einige Anmerkungen zur Strukturoptimierung Neuronaler Netze. Hinweise zur Durchführung von Versuchen mit Neuronalen Netzen und zu möglichen Bewertungskriterien der Trainingsresultate werden in Abschnitt 5 gegeben. Abschnitt 6 stellt verschiedene

Möglichkeiten des Einsatzes Neuronaler Netze als nichtlineare Regler vor. Die in ICONNECT zur Verwendung Neuronaler Netze implementierten Module werden in Abschnitt 7 beschrieben. Die Ausführungen in den Abschnitten 2 bis 5 sind stark an die Erläuterungen in [Sic00] angelehnt.

2 Dynamische, nicht-rekurrente Netzparadigmen

In diesem Abschnitt werden verschiedene vorwärtsgerichtete (d.h. nicht-rekurrente) Netzparadigmen vorgestellt. Neben Time-Delay-Netzen werden auch Mehrlagige Perzeptren und Mehrlagige Perzeptren mit gleitendem Eingabefenster als Sonderfall von Time-Delay-Netzen definiert.

2.1 Time-Delay-Netze

Time-Delay-Netze sind vorwärtsgerichtete Neuronale Netze mit Verzögerungselementen, die erstmals durch Waibel et al. (siehe beispielsweise [WHH+89]) beschrieben wurden. Die Verzögerung von Aktivierungen ist durchaus auch biologisch motiviert, wie das Zitat am Kapitelanfang zeigt (siehe auch [Wan93a]).

Definition 2.1 (*Time-Delay-Netz*)
Ein *Time-Delay-Netz* (*time-delay neural network, TDNN*) oder *FIR-Netz* (*finite impulse response network*) ist definiert durch:

(1) $\mathcal{U} = \mathcal{U}_1 \cup \ldots \cup \mathcal{U}_L$ mit $L \in \mathbb{N}$ und $L \geq 3$ ist eine Menge von *Neuronen* (*Verarbeitungseinheiten*, *Knoten*). Es gelte $\mathcal{U}_h \neq \emptyset$ für alle $h \in \{1, \ldots, L\}$ und $\mathcal{U}_h \cap \mathcal{U}_l = \emptyset$ für alle $h, l \in \{1, \ldots, L\}$ mit $h \neq l$. $\mathcal{U}_1$ heißt *Eingabeschicht*, $\mathcal{U}_L$ heißt *Ausgabeschicht* und alle $\mathcal{U}_l$ mit $l \in \{2, \ldots, L-1\}$ heißen *verdeckte*, *innere* oder *verborgene* Schichten; die entsprechenden Neuronen heißen *Eingabeneuronen*, *Ausgabeneuronen* bzw. *verdeckte*, *innere* oder *verborgene* Neuronen.

(2) Jedem $\mathcal{U}_l$ (mit $l \in \{2, \ldots, L\}$) wird eine Zahl $d^{(l)} \in \mathbb{N}$ zugewiesen; dabei ist $d^{(l)} - 1$ die *maximale Verzögerung* zwischen den Neuronen in den Schichten $l-1$ und l. Jedem Paar von Neuronen $(i, j) \in \mathcal{U}_{l-1} \times \mathcal{U}_l$ (mit $l \in \{2, \ldots, L\}$) wird ein Vektor $\mathbf{w}_{i,j}^{(l)} \stackrel{def}{=} \left(w_{i,j}^{(l)}(0), w_{i,j}^{(l)}(1), \ldots, w_{i,j}^{(l)}(d^{(l)} - 1)\right) \in \mathbb{R}^{d^{(l)}}$ zugeteilt. Dieser Vektor heißt *Gewichtsvektor* der *Verbindung* zwischen Neuron i in Schicht $\mathcal{U}_{l-1}$ und Neuron j in Schicht $\mathcal{U}_l$; die einzelnen Elemente des Vektors heißen *Gewichte*. Durch die Neuronen und die Verbindungen wird die *Netzstruktur* festgelegt.

(3) Jedem Neuron $j \in \mathcal{U}_l$ mit $l \in \{2, \ldots, L\}$ wird eine *Netzeingabe-* oder *Propagierungsfunktion* zur Berechnung der *Netzeingabe* $s_j^{(l)}(k)$ definiert durch

$$s_j^{(l)}(k) \stackrel{def}{=} \sum_{i \in \mathcal{U}_{l-1}} \sum_{m=0}^{d^{(l)}-1} w_{i,j}^{(l)}(m) a_i^{(l-1)}(k-m) + w_{b,j}^{(l)},$$

zugeordnet. $w_{b,j}^{(l)} \in \mathbb{R}$ heißt *Bias* oder *Schwellwert*; $k \in \mathbb{N}$ ist eine Zeitvariable. Mit $\mathbf{a}_i^{(l)}(k) \stackrel{def}{=} \left(a_i^{(l)}(k), a_i^{(l)}(k-1), \ldots, a_i^{(l)}(k-d^{(l+1)}+1)\right)$ und dem Standardskalarprodukt $\langle \cdot | \cdot \rangle$ für Vektoren aus $\mathbb{R}^{d^{(l)}}$ gilt

$$s_j^{(l)}(k) = \sum_{i \in \mathcal{U}_{l-1}} \left\langle \mathbf{w}_{i,j}^{(l)} \middle| \mathbf{a}_i^{(l-1)}(k) \right\rangle + w_{b,j}^{(l)}.$$

(4) Jedem Neuron $j \in \mathcal{U}_l$ mit $l \in \{2, \ldots, L\}$ wird eine *Aktivierungsfunktion* $\sigma : \mathbb{R} \to]a, b[$ mit $a, b \in \mathbb{R}$ zugeordnet, so daß

$$a_j^{(l)}(k) \stackrel{def}{=} \sigma(s_j^{(l)}(k)).$$

Dabei ist σ eine nichtlineare, sigmoide Funktion, die für alle Neuronen gleich gewählt wird. $a_j^{(l)}(k)$ wird als *Aktivierung* des Neurons j zum Zeitpunkt k bezeichnet. Für die Neuronen der Eingabeschicht ($j \in \mathcal{U}_1$) gelte $a_j^{(1)}(k) = s_j^{(1)}(k)$ (identische Abbildung).

(5) Für jedes Eingabeneuron $j \in \mathcal{U}_1$ gilt $s_j^{(1)}(k) \stackrel{def}{=} x_j(k)$ mit der *j-ten externen Eingabe* des Netzes $x_j(k) \in \mathbb{R}$. Analog ist für jedes Ausgabeneuron $j \in \mathcal{U}_L$ die *j-te externe Ausgabe* des Netzes $y_j(k)$ definiert durch $y_j(k) \stackrel{def}{=} a_j^{(L)}(k)$. Der Vektor $\mathbf{x}(k) \stackrel{def}{=} \left(x_1(k), \ldots, x_{|\mathcal{U}_1|}(k)\right)$ heißt *Eingabemuster* und der Vektor $\mathbf{y}(k) \stackrel{def}{=} \left(y_1(k), \ldots, y_{|\mathcal{U}_L|}(k)\right)$ heißt *Ausgabemuster* des Netzes zum Zeitpunkt k.

Neuronen:

Die Verarbeitungseinheiten oder Neuronen eines TDNN berechnen aus einer Eingabe einen Aktivierungszustand, der an nachfolgende Neuronen weitergegeben wird. Neuronen arbeiten unabhängig voneinander und können ihre Eingaben (quasi-)parallel verarbeiten. Nur Eingabe- oder Ausgabeneuronen kommunizieren mit der Umgebung des Netzes.

Bei den verwandten Mehrlagigen Perzeptren (siehe Abschnitt 2.2) wird häufig jedes Neuron um eine zusätzliche sogenannte *(Netz-)Ausgabefunktion* erweitert (siehe z.B. [NKK96]). Die Aktivierung dient als Argument dieser Ausgabefunktion und der

Funktionswert wird dann statt der Aktivierung an nachfolgende Neuronen weitergegeben. Da bei TDNN die Ausgabefunktion im allgemeinen die identische Abbildung ist, wird sie nicht in der Definition angegeben.

Die Zahl der Schichten eines TDNN ist wegen der besseren Approximationseigenschaften meist (wie auch in der Definition vereinbart) $L \geq 3$. Prinzipiell wäre auch $L = 2$ möglich; die in Abschnitt 3 vorgestellten Lernalgorithmen wären dann genauso anwendbar.

Netzstruktur:

Die Netzstruktur beschreibt die Kommunikationsverbindungen zwischen den Neuronen. Diese Verbindungen werden häufig auch als *Synapsen*, die Gewichte daher als *synaptische Gewichte* bezeichnet. Die Gewichte dienen als Bewertung für die auf den Verbindungen übertragenen Informationen. Ein Gewicht $w_{i,j}^{(l)}(m) < 0$ heißt *inhibitorisch* oder *hemmend*, bei $w_{i,j}^{(l)}(m) > 0$ spricht man von einem *excitatorischen* oder *anregenden* Gewicht. Die Gewichte werden auch als *Informationsspeicher* eines Neuronalen Netzes bezeichnet.

Bei einem TDNN ist eine Verbindung jeweils mit einem Gewichtsvektor annotiert, der die Koeffizienten eines *FIR-Filters* (*FIR: finite impulse response*) zwischen den beiden Neuronen angibt. Eigenschaften von FIR-Filtern sind beispielsweise in [Ham87] beschrieben. Abbildung 1 zeigt die Struktur dieses Filters; dabei ist q^s ($s \in \mathbb{Z}$) ein *Shiftoperator*, der durch $q^s(x(k)) = x(k+s)$ definiert ist. In diesem Fall ist q^{-1} also ein *Verzögerungsoperator* für eine Zeiteinheit. Der FIR-Filter hat die *Ordnung* $d^{(l)} - 1$. Die Berechnung der Filterausgabe $\left\langle \mathbf{w}_{i,j}^{(l)} \middle| \mathbf{a}_i^{(l-1)}(k) \right\rangle$ erfolgt im Rahmen der Berechnung der Netzeingabe des Nachfolgeneurons.

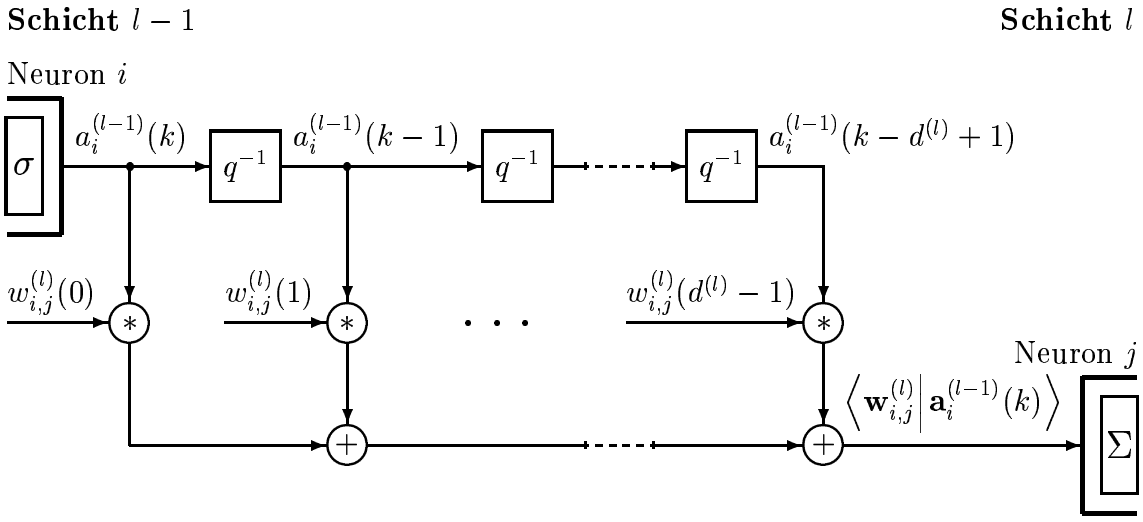


Abbildung 1: Synapse mit FIR-Filter im TDNN bei der Vorwärtspropagierung von Aktivierungszuständen

Bei der Definition des TDNN sind keine rückwärtsgerichteten Verbindungen (*Rückkopplungen*) zugelassen, nur *vorwärtsgerichtete* Verbindungen in die folgende Schicht. Man spricht daher auch von einem *schichtweisen* oder *geschichteten* Aufbau eines Netzes. Um die im folgenden hergeleiteten Lernalgorithmen (Abschnitt 3) einfacher darstellen zu können, sind bei dieser Definition auch keine *schichtübergreifenden* Verbindungen (sog. *shortcut*-Verbindungen) zugelassen. Diese Einschränkung ist prinzipiell nicht notwendig, wird aber häufig bei TDNN und verwandten Netzen durchgeführt. Nur auf den ersten Blick fordert die Definition ein *vollständig verbundenes* Netz, d.h. Verbindungen zwischen **jedem** Paar $(i,j) \in \mathcal{U}_{l-1} \times \mathcal{U}_l$ mit $l \in \{2, \ldots, L\}$, und die **gleiche** Filterordnung für alle Verbindungen zwischen zwei bestimmten Schichten. Einer nicht-existierenden Verbindung entspricht das Setzen aller Gewichte der Verbindung auf Null. Da $d^{(l)} - 1$ die maximale Verzögerung zwischen zwei Schichten (d.h. die maximale Filterordnung) angibt, sind auch hier nur entsprechende Gewichte auf Null zu setzen.

Propagierung:

In vielen statischen Netzparadigmen wird die Netzeingabe als gewichtete Summe der aktuellen Aktivierungen der Vorgängerneuronen berechnet. Beim TDNN ist die Netzeingabe die gewichtete Summe von aktuellen und vergangenen Aktivierungen der Vorgängerneuronen. Dadurch zeigt ein TDNN ein dynamisches Systemverhalten, das allerdings nicht – wie bei vielen anderen dynamischen Netzparadigmen – auf Rückkopplungen im Netz zurückzuführen ist. Die angegebene Propagierungsfunktion enthält die Berechnung der Filterausgabe durch eine Faltung des Gewichtsvektors mit dem Vektor der verzögerten Aktivierungen.

Aktivierung:

Die Aktivierung eines Neurons hängt nur von der aktuellen Netzeingabe ab. Zur nichtlinearen Modellierung werden Netze benötigt, die als Aktivierungsfunktionen für die Neuronen verdeckter Schichten und der Ausgabeschicht nichtlineare Funktionen verwenden. Im allgemeinen werden sogenannte *sigmoide* Aktivierungsfunktionen gewählt, die kontinuierliche Aktivierungszustände aufweisen, monoton steigend und differenzierbar sind. Für die Neuronen der Eingabeschicht wird üblicherweise die identische Abbildung gewählt.

Tabelle 1 gibt drei geeignete Aktivierungsfunktionen (siehe auch Abbildung 2) an: die *logistische Aktivierungsfunktion* oder *Fermi-Funktion* [Hof93, Zel94], der *Tangens hyperbolicus* [Hof93, Zel94] und die *„schnelle " Aktivierungsfunktion* [Ell93]. Wichtig für ein effizientes Training eines Neuronalen Netzes (siehe Abschnitt 3) ist, daß sich die Ableitung $\sigma'(x)$ einer Aktivierungsfunktion $\sigma(x)$ (siehe auch Abbildung 3) an einer bestimmten Stelle x mit möglichst geringem Rechenaufwand bestimmen läßt (siehe dazu die als Differentialgleichung interpretierbare vorletzte Spalte in Tabelle 1).

Tabelle 1: Verschiedene sigmoide Aktivierungsfunktionen und ihre Ableitungen

Bezeichnung	**Funktion** σ	**Wertebereich** $]a, b[$	**Ableitung** σ'	**Steigung für** $x = 0$
logistische Aktivierungsfunktion $\sigma_{log}(x)$	$\frac{1}{1+e^{-x}}$	$]0, 1[$	$\sigma_{log}(x)(1 - \sigma_{log}(x))$	0.25
Tangens hyperbolicus $\sigma_{tanh}(x)$	$\tanh(x)$	$]-1, 1[$	$1 - \sigma^2_{tanh}(x)$	1.0
„schnelle“ Aktivierungsfunktion $\sigma_{fast}(x)$	$\frac{x}{1+\|x\|}$	$]-1, 1[$	$(1 - \|\sigma_{fast}(x)\|)^2$	1.0

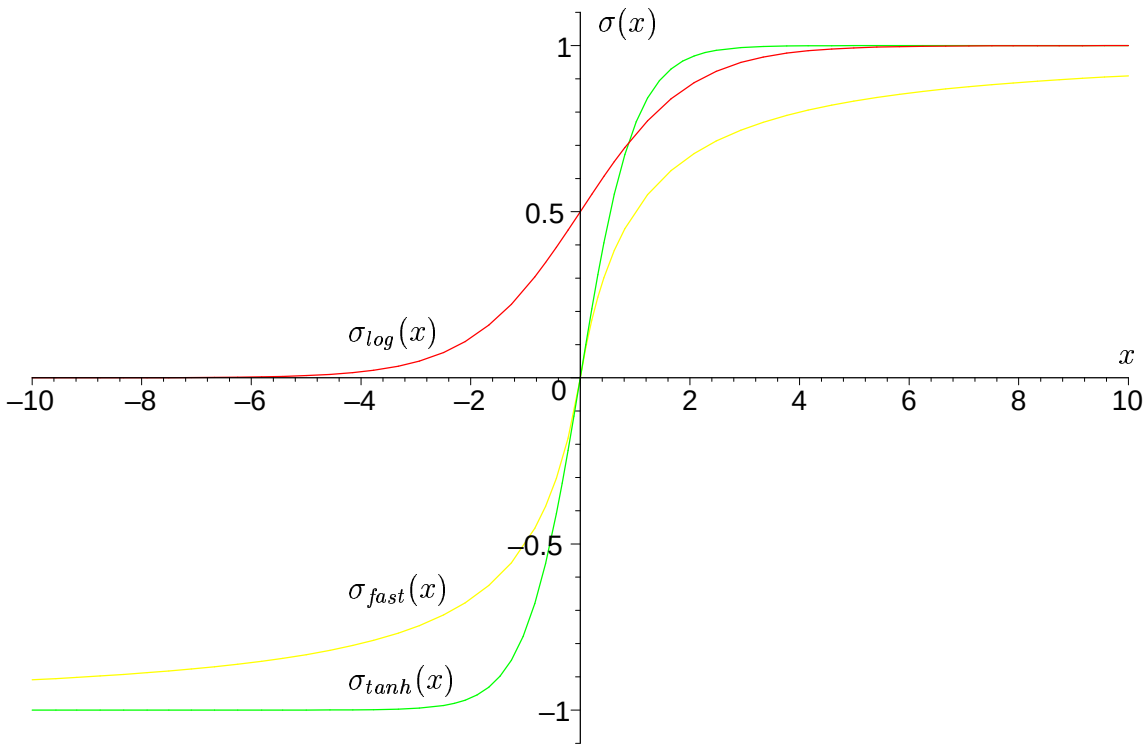


Abbildung 2: Graphen verschiedener Aktivierungsfunktionen

Für die Ableitung der logistischen Aktivierungsfunktion gilt:

$$\sigma_{log}(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

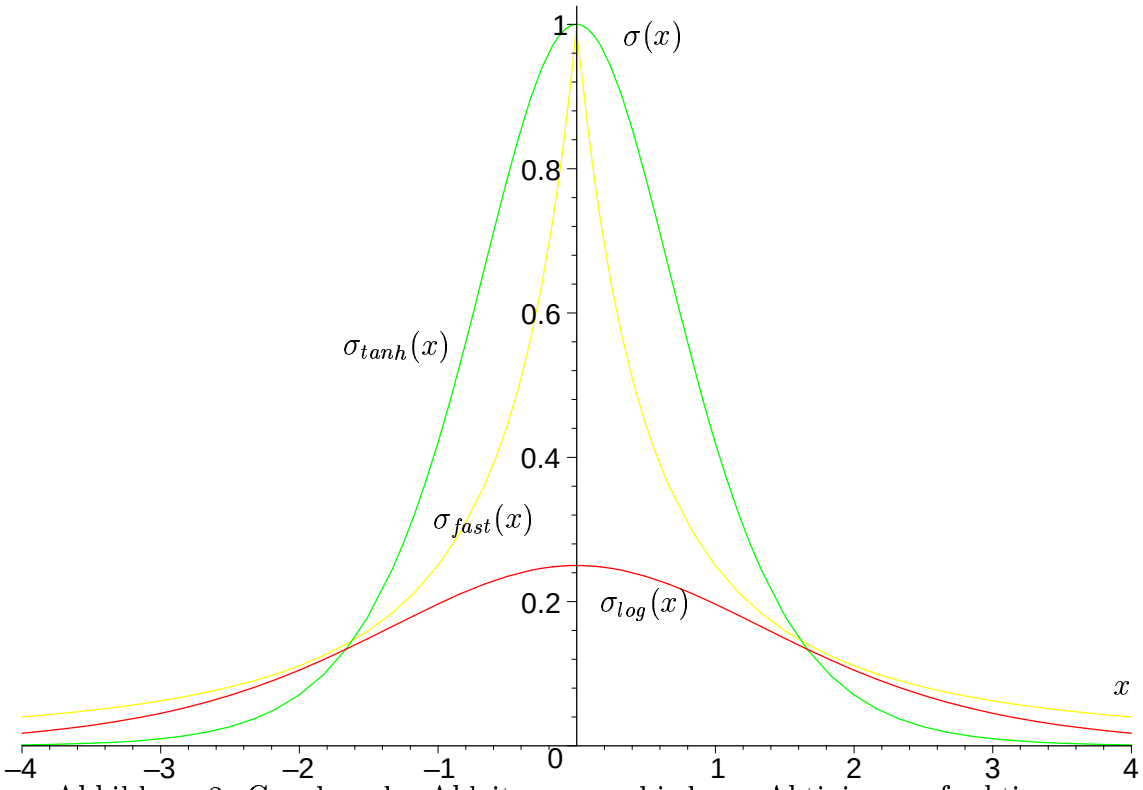


Abbildung 3: Graphen der Ableitung verschiedener Aktivierungsfunktionen

$$\sigma'_{log}(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{2}$$

$$= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \tag{3}$$

$$= \sigma_{log}(x)\,\frac{1+e^{-x}-1}{1+e^{-x}} \tag{4}$$

$$= \sigma_{log}(x)\,(1-\sigma_{log}(x)) \tag{5}$$

Analog kann die Ableitung des Tangens hyperbolicus angegeben werden:

$$\sigma_{tanh}(x) = \tanh(x) \tag{6}$$

$$= \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

$$\sigma'_{tanh}(x) = \frac{(e^x+e^{-x})(e^x-e^{-x}(-1)) - (e^x-e^{-x})(e^x+e^{-x}(-1))}{(e^x+e^{-x})^2} \tag{8}$$

$$= \frac{(e^x+e^{-x})^2 - (e^x-e^{-x})^2}{(e^x+e^{-x})^2} \tag{9}$$

$$= 1-\tanh^2(x) \tag{10}$$

$$= 1-\sigma^2_{tanh}(x) \tag{11}$$

Zwischen dem Tangens hyperbolicus und der logistischen Funktion besteht übrigens folgender Zusammenhang:

$$\sigma_{tanh}(x) = \tanh(x) \tag{12}$$

$$= \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{13}$$

$$= \frac{e^x(1 - e^{-2x})}{e^x(1 + e^{-2x})} \tag{14}$$

$$= \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{15}$$

$$= \frac{2 - (1 + e^{-2x})}{1 + e^{-2x}} \tag{16}$$

$$= 2\left(\frac{1}{1 + e^{-2x}}\right) - 1 \tag{17}$$

$$= 2\,\sigma_{log}(2x) - 1 \tag{18}$$

Für die Ableitung der „schnellen“ Aktivierungsfunktion gilt:

$$\sigma_{fast}(x) = \frac{x}{1 + |x|} \tag{19}$$

$$\sigma'_{fast}(x) = \frac{1}{(1 + |x|)^2} \tag{20}$$

$$= \left(\frac{1}{1 + |x|}\right)^2 \tag{21}$$

$$= \left(\frac{1 + |x| - |x|}{1 + |x|}\right)^2 \tag{22}$$

$$= \left(1 - \frac{|x|}{1 + |x|}\right)^2 \tag{23}$$

$$= (1 - |\sigma_{fast}(x)|)^2 \tag{24}$$

Die Güte eines neuronalen Modells auf der Basis eines TDNN ist durch die Wahl der Aktivierungsfunktion beeinflußt. Es hängt aber stark von der jeweiligen Problemstellung ab, welche Aktivierungsfunktion bei einer bestimmten Netzstruktur besser geeignet ist. Die für die Modellbildung ebenfalls wichtige *Steilheit* (Steigung) der Aktivierungsfunktion an der Stelle $x = 0$ kann bei den meisten Aktivierungsfunktionen über zusätzliche Faktoren eingestellt werden, z.B. bei der *logistischen*

Aktivierungsfunktion mit Temperaturkoeffizient T ($T \in \mathbb{R}^+$) [Zel94]:

$$\sigma_{logT}(x) = \frac{1}{1 + e^{-x\frac{1}{T}}}, \qquad \sigma'_{logT}(x) = \frac{1}{T}\,\sigma_{logT}(x)\,(1 - \sigma_{logT}(x))\,.$$

Je kleiner T ist, desto steiler ist die Aktivierungsfunktion an der Stelle $x = 0$ (siehe Abbildung 4).

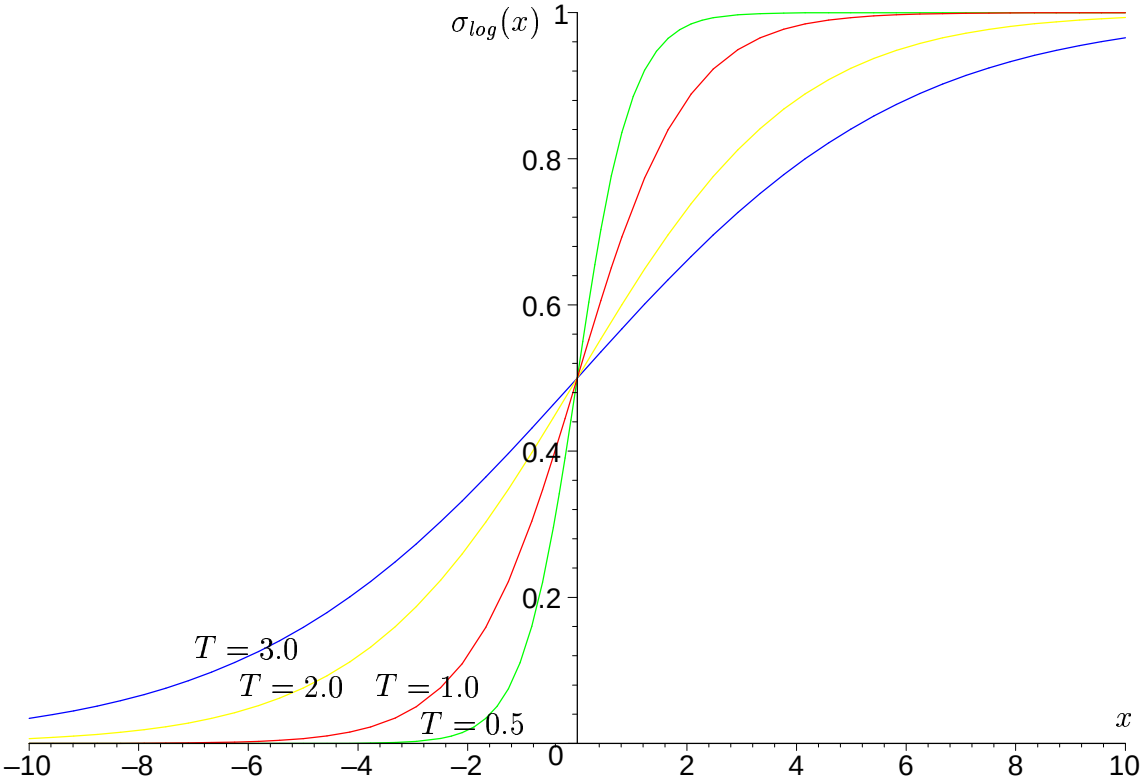


Abbildung 4: Graphen der logistischen Aktivierungsfunktion mit Temperaturkoeffizient (verschiedene Werte von T)

Für rechenintensive Anwendungen ist jedenfalls interessant, daß sich die angegebene „schnelle“ Aktivierungsfunktion tatsächlich deutlich schneller auswerten läßt. So wurde in Untersuchungen (Simulationen auf einer SparcStation20 von SUN) festgestellt, daß der Zeitaufwand für die Berechnung der Aktivierung plus der Ableitung der Aktivierung an einer bestimmten Stelle bei der logistischen Funktion etwa 2.3 Mal so hoch ist, beim Tangens hyperbolicus sogar etwa 4.3 Mal so hoch [Wei98].

Externe Eingabe und externe Ausgabe:

Die externe Eingabe und die externe Ausgabe stellen die Verbindung des Netzes mit seiner Umgebung dar.

Entsprechend dem Wertebereich der Aktivierungsfunktion $]a, b[$ und der Steigung an der Stelle $x = 0$ werden häufig die Eingabewerte und Ausgabewerte eines Netzes

jeweils (d.h. für jedes Neuron) auf ein Intervall $[c, d]$ skaliert, wobei $c = a + \varepsilon$, $d = b - \varepsilon$ und $0 \leq \varepsilon \ll \frac{b-a}{2}$ mit $c, d \in \mathbb{R}$ und $\varepsilon \in \mathbb{R}^+$ [Sar96]. D.h., der größte in den Trainingsdaten enthaltene Wert wird auf d, der kleinste auf c abgebildet. Dies kann sich als sehr vorteilhaft für das Training des Netzes erweisen (siehe Abschnitt 3).

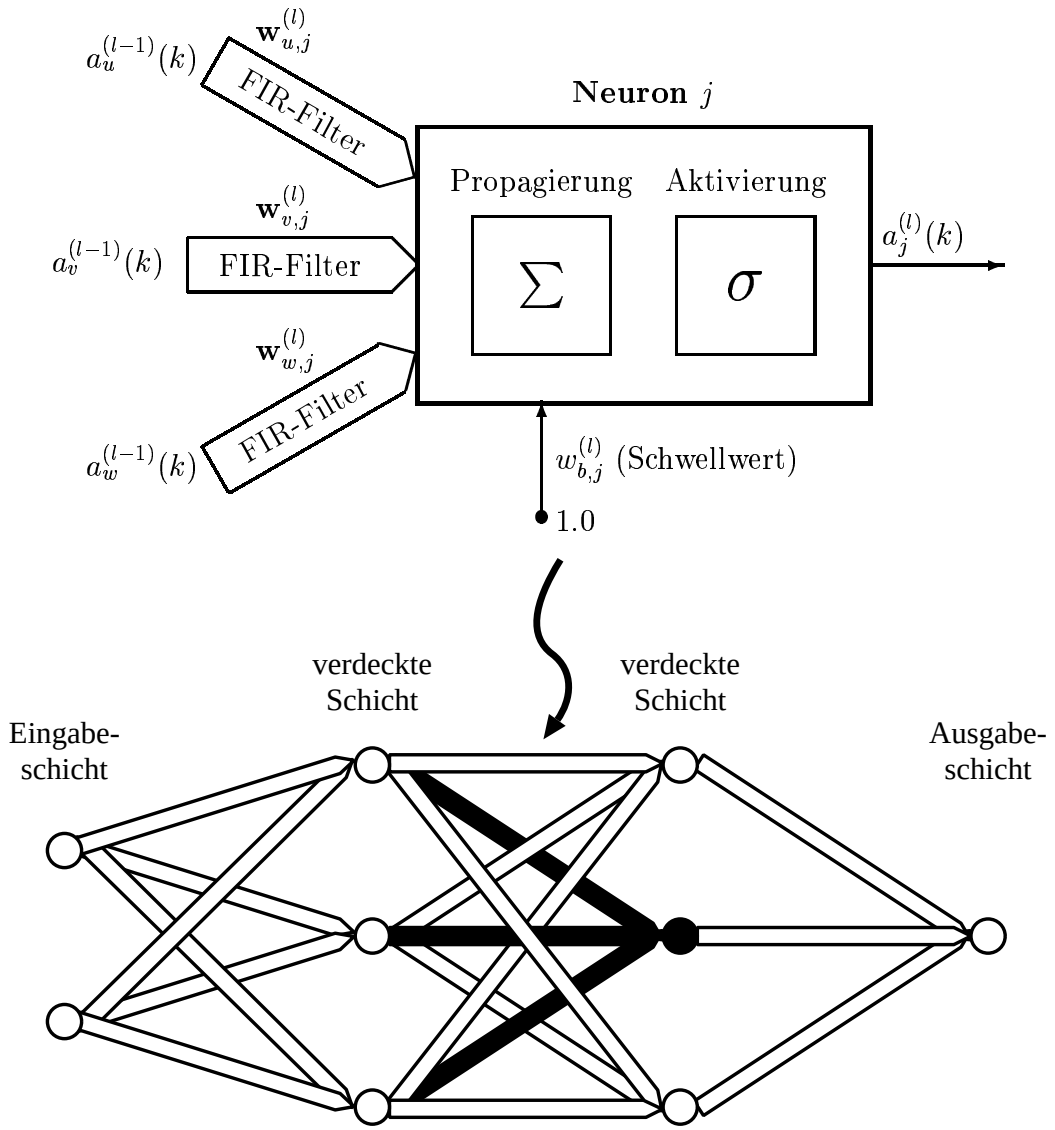


Abbildung 5: Aufbau eines TDNN aus Neuronen und FIR-Filtern

Ein TDNN läßt sich nun zusammenfassend so darstellen, wie in Abbildung 5 gezeigt. Alle Verbindungen sind dabei als FIR-Filter modelliert. Der Schwellwert ist hier als Gewicht einer Verbindung modelliert, deren Aktivierungswert konstant 1.0 ist. Diese Form der Beschreibung erleichtert die Darstellung von Lernalgorithmen (siehe Abschnitt 3) für TDNN und verwandte Netze.

Für die Struktur eines schichtweise aufgebauten, vollständig verbundenen TDNN wird folgende Notation verwendet: $|\mathcal{U}_1| \overset{d(2)}{\rightarrow} |\mathcal{U}_2| \overset{d(3)}{\rightarrow} \cdots \overset{d(L)}{\rightarrow} |\mathcal{U}_L|$. Beispielsweise ist $3 \overset{2}{\rightarrow} 12 \overset{3}{\rightarrow} 1$ ein dreischichtiges Netz mit drei Eingabeneuronen, zwölf verdeckten Neuronen und einem Ausgabeneuron sowie Verzögerungen null und eins zwischen Eingabe- und verdeckter Schicht und null, eins und zwei zwischen verdeckter und Ausgabeschicht.

TDNN wurden ursprünglich Ende der achtziger Jahre für die Spracherkennung (Phonemklassifikation) entwickelt [WHH+89]. Erst deutlich später wurden sie auch in anderen Anwendungsgebieten eingesetzt; einige Beispiele für Anwendungen dieser Netze sind:

- Sprachanalyse und Spracherkennung [BW93a, HLW93, WG94], z.B. auch als Bestandteil eines Systems zur modellbasierten Synthese von Lippenbewegungen [LLBC97],
- verschiedene Aufgabenstellungen aus dem Bereich der Zeitreihenvorhersage (z.B. chaotisches Pulsieren der Intensität eines NH_3-Lasers [Wan93b, Wan93c], Schlachtpreise für Schweine [KWL94] oder Wechselkursvorhersage Schweizer Franken in US-Dollar [Wan93a]),
- Ermittlung der Größe und der Geschwindigkeit von Regentropfen (mit Anwendungen in der Meteorologie und Telekommunikation) [DGT98],
- Erkennung von Handschrift (z.B. automatische Auswertung des Überweisungsbetrags bei Bankschecks [MKB97], Erkennung von Postleitzahlen [BPR97] oder als Komponente eines Multimediasystems [WSVY97]),
- Klassifikation seismischer Signale [KDMM98],
- Schädlingsbekämpfung (Bestimmung von Insekten in Getreidesilos mit Hilfe von akustischen Signalen) [CP98],
- Identifikation nichtlinearer Systeme (mit unterschiedlichsten Anwendungen) [YK98],
- Navigation einer Kugel durch ein Labyrinth auf einem kippbaren Tisch (TDNN als nichtlinearer, dynamischer Regler) [May99] und
- Werkzeugzustandsüberwachung (Verschleißbestimmung) in CNC-Drehmaschinen [Sic00].

Die angegebene Form der Definition eines TDNN ordnet die Verzögerung der Aktivierungszustände eines Vorgängerneurons den Verbindungen zu, die in Form eines

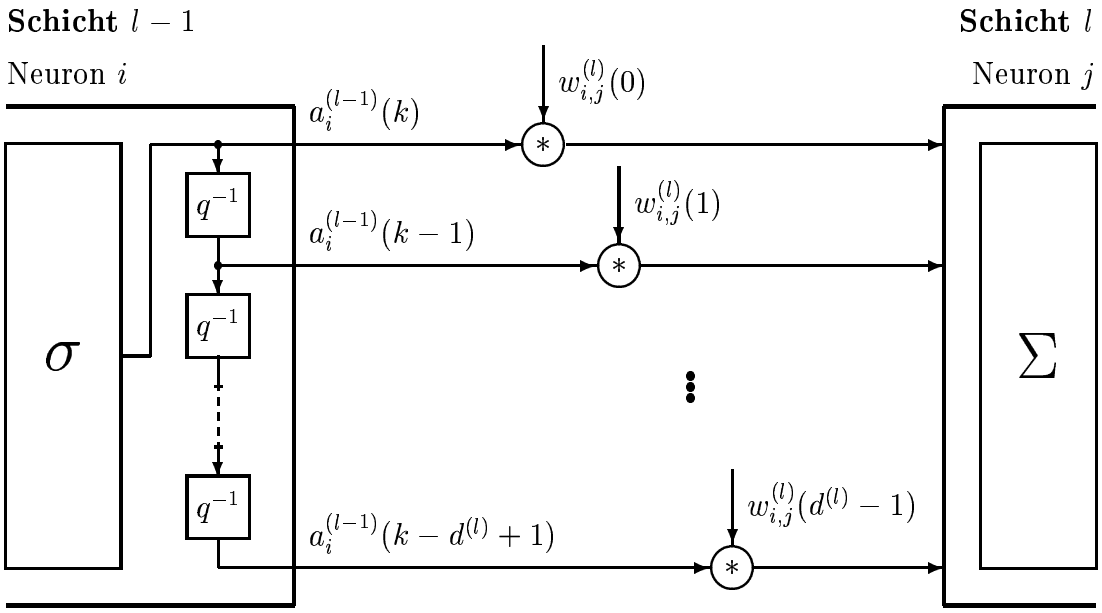


Abbildung 6: Ursprüngliche Beschreibung einer Synapse im TDNN

FIR-Filters modelliert sind. Daher werden diese Netze auch als FIR-Netze bezeichnet [Wan93a, Wan93c].

Bei der ursprünglichen Definition von TDNN wird die Verzögerung von Aktivierungszuständen den Neuronen zugeteilt (siehe Abbildung 6). Beide Netze (TDNN und FIR) sind funktional äquivalent. Unterschiede sind bei der Implementierung erkennbar: Bei FIR-Netzen sind Verzögerungselemente den Synapsen zugeordnet, also für jedes Neuron mehrfach vorhanden. In dieser Arbeit wurde die Beschreibungsform des FIR-Netzes gewählt, da sie eine einfachere Darstellung der Lernalgorithmen ermöglicht. Trotzdem wird die Bezeichnung TDNN verwendet, da sie die ältere und in der Literatur weitaus bekanntere ist. Interessant ist auch die Analogie zwischen TDNN und bestimmten endlichen Automaten (*DMM: definite memory machines*, siehe [CGHC97]).

2.2 Mehrlagige Perzeptren mit gleitendem Eingabefenster

In diesem Abschnitt werden Mehrlagige Perzeptren und Mehrlagige Perzeptren mit gleitendem Eingabefenster definiert und ihr Bezug zu TDNN wird erläutert.

Definition 2.2 (*Mehrlagiges Perzeptron, MLP*)
Ein *mehrlagiges Perzeptron* (oder *Backpropagation-Netz*) ist ein TDNN, bei dem für alle Schichten $l \in \{2, \ldots, L\}$ gilt: $d^{(l)} = 1$.

Diese Form der Definition beschreibt ein MLP als Sonderfall eines TDNN mit statischem Systemverhalten. Eine bekannte Methode, um einem MLP ein dynamisches Verhalten zu geben, ist die Verwendung eines gleitenden Eingabefensters, wie in der folgenden Definition angegeben.

Definition 2.3 (*MLP mit gleitendem Eingabefenster, MLP-sw*)
Ein *mehrlagiges Perzeptron mit gleitendem Eingabefenster* (*sw: sliding window*) ist ein MLP mit einer Menge von Eingabeneuronen $\mathcal{U}_1 = \mathcal{U}_{1,0} \cup \ldots \cup \mathcal{U}_{1,N-1}$ mit $N \in \mathbb{N}$. Es gelte $\mathcal{U}_{1,h} \neq \emptyset$ und $\mathcal{U}_{1,h} \cap \mathcal{U}_{1,l} = \emptyset$ für alle $h, l \in \{0, \ldots, N-1\}$ mit $h \neq l$. Außerdem sei $|\mathcal{U}_{1,0}| = \ldots = |\mathcal{U}_{1,N-1}| = d$ mit $d \in \mathbb{N}$. Die jeweils d externen Eingaben einer Teilmenge von Eingabeneuronen $\mathcal{U}_{1,h}$ hängen wie folgt zusammen:

$$\begin{aligned}&\big(x_{h\cdot d+1}(k), x_{h\cdot d+2}(k), \ldots, x_{(h+1)\cdot d}(k)\big)\\ &\quad = \quad (x_{h\cdot d+1}(k), x_{h\cdot d+1}(k-1), \ldots, x_{h\cdot d+1}(k-d+1))\,;\end{aligned}$$

d heißt die *Länge* des gleitenden Eingabefensters.

Abbildung 7 zeigt den Aufbau eines MLP-sw; MLP-sw wird hier als eigenes Netzparadigma aufgefaßt. Die Vereinbarung von N Eingabefenstern der jeweils gleichen Länge d bedeutet wiederum keine Einschränkung: Man wähle d als das Maximum der jeweils gewünschten Längen der N Eingabefenster und setze die Gewichte entsprechender, von der Eingangsschicht des MLP ausgehender Verbindungen gleich Null.

Definition 2.4 (*Rezeptives Fenster*)
Das *rezeptive Fenster* ist das Zeitintervall, aus dem einem Neuronalen Netz Informationen irgendeiner Form zur Berechnung der externen Ausgabe zur Verfügung stehen.

Im Gegensatz zu Netzen mit Rückkopplungen hat das rezeptive Fenster bei TDNN, MLP und MLP-sw eine endliche, feste Länge. Die Länge dieses rezeptiven Fensters ist

(1) beim TDNN $\sum_{l=2}^{L}(d^{(l)} - 1) + 1$,

(2) beim MLP 1 und

(3) beim MLP-sw d.

Sinnvollerweise sollte das rezeptive Fenster eines Netzes deutlich kürzer als jede der verarbeiteten Zeitreihen sein.

Die Struktur eines MLP wird im folgenden analog zur Notation für TDNN mit $|\mathcal{U}_1| \overset{1}{\rightarrow} |\mathcal{U}_2| \overset{1}{\rightarrow} \cdots \overset{1}{\rightarrow} |\mathcal{U}_L|$ (oder kurz mit $|\mathcal{U}_1| \rightarrow |\mathcal{U}_2| \rightarrow \cdots \rightarrow |\mathcal{U}_L|$); die Struktur eines

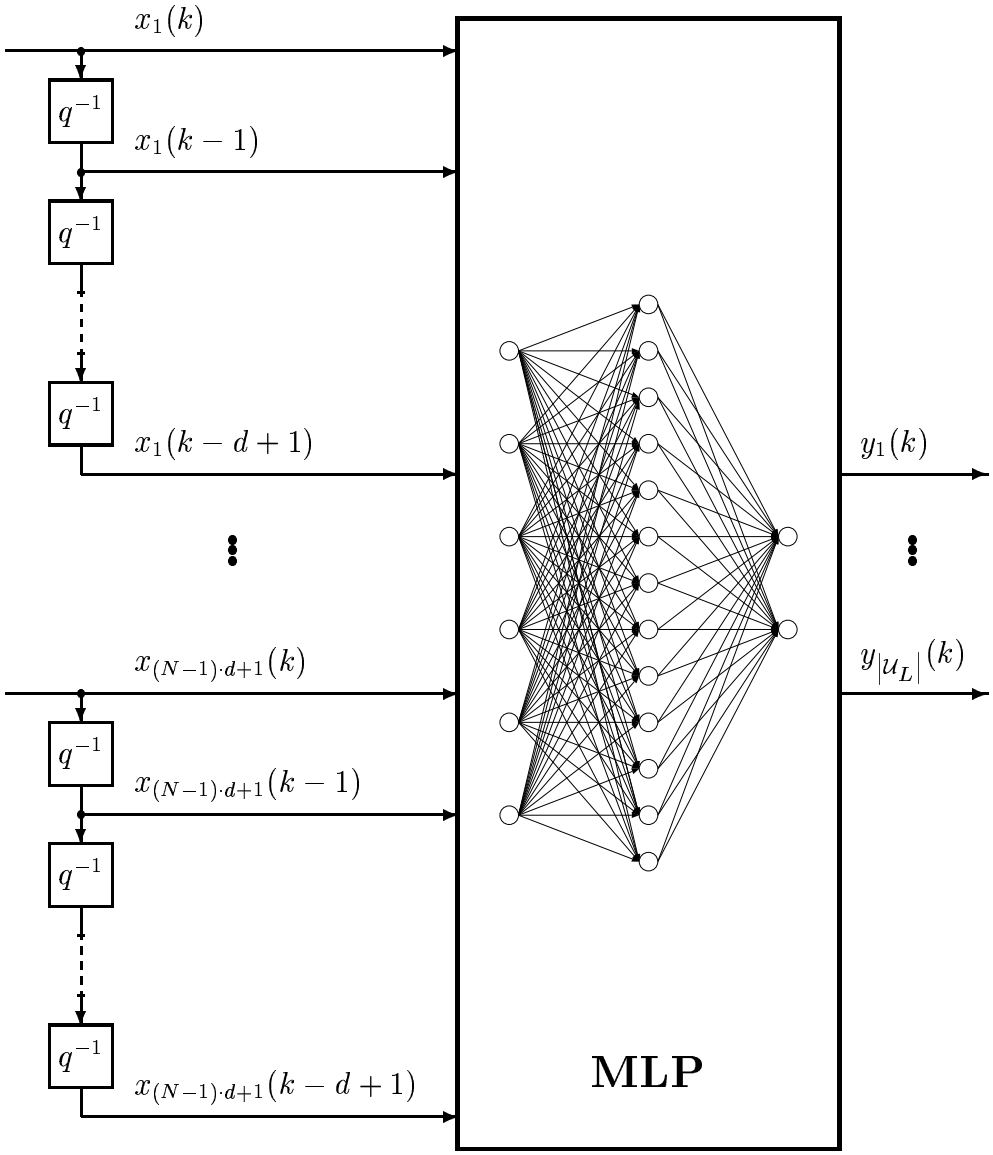


Abbildung 7: Aufbau eines Mehrlagigen Perzeptrons mit gleitendem Eingabefenster

MLP-sw mit $|\mathcal{U}_1| \cdot d \xrightarrow{1} |\mathcal{U}_2| \xrightarrow{1} \cdots \xrightarrow{1} |\mathcal{U}_L|$ (oder kurz mit $|\mathcal{U}_1| \cdot d \to |\mathcal{U}_2| \to \cdots \to |\mathcal{U}_L|$) angegeben.

Wie sehr einfach zu sehen ist, gibt es zu jedem TDNN, das nur Aktivierungen der Eingabeschicht verzögert weitergibt (d.h. $d^{(2)} > 1$ und $d^{(l)} = 1$ für $l \in \{3, \ldots, L\}$) ein äquivalentes MLP-sw mit gleitendem Eingabefenster der Länge $d = d^{(2)}$. Darüberhinaus gibt es aber sogar zu **jedem beliebigen** TDNN ein äquivalentes MLP-sw. Die Konstruktion dieses MLP-sw aus einem TDNN erfolgt mit Hilfe eines Algorithmus, der als *zeitliche Entfaltung* bekannt ist [Wan93a, Wan93c]. Der Algorithmus startet bei der Ausgabeschicht und interpretiert jede verzögerte Aktivierung als die Ausgabe eines virtuellen Neurons, dessen Eingabe durch eine geeignete Anzahl von Zeitschritten verzögert wird. Dementsprechend wird jeder FIR-Filter

eliminiert, indem der vorausgehende Teil des TDNN vervielfacht wird und entsprechend verzögerte Eingaben an diese Kopien angelegt werden (siehe Abbildung 8). Der Algorithmus wird von der Ausgabe- hin zur Eingabeschicht nacheinander auf alle Schichten angewandt, bis alle Verzögerungen eliminiert sind. Das Verfahren erzeugt ein äquivalentes MLP-sw, das jedoch eine deutlich höhere Zahl von Neuronen und von Gewichten besitzt, obwohl es nicht vollständig verbunden ist. Information, die im TDNN in **einem** Gewicht repräsentiert ist, ist im MLP-sw auf **mehrere** Gewichte verteilt. Man spricht daher auch von *gemeinsam verwendeten Gewichten*. Außer einer höheren Zahl von Gewichten weist dieses Netz, dessen Schichten nicht vollständig miteinander verbunden sind, auch eine höhere Neuronenzahl auf. Weitere große Unterschiede zwischen beiden Netzen werden bei der Suche nach geeigneten Lernverfahren zur Einstellung der Gewichte deutlich. Daher wird im folgenden statt von *äquivalenten* (Begriff eingeführt in [Wan93a, Wan93c]) von *funktional äquivalenten* Netzen gesprochen.

Analog zu dem hier beschriebenen Algorithmus der zeitlichen Entfaltung von der Ausgabe- hin zur Eingabeschicht läßt sich auch ein Algorithmus angeben, der ein MLP-sw in umgekehrter Richtung von der Eingabe- hin zur Ausgabeschicht erzeugt [Hay94].

Vergleicht man ein TDNN und ein vollständig verbundenes MLP-sw mit gleicher Länge des rezeptiven Fensters, gleicher Schichtenzahl und gleicher Neuronenzahl in den verdeckten Schichten und der Ausgabeschicht (d.h. ein TDNN $|\mathcal{U}_1| \overset{d^{(2)}}{\rightarrow} |\mathcal{U}_2| \overset{d^{(3)}}{\rightarrow} \cdots \overset{d^{(L)}}{\rightarrow} |\mathcal{U}_L|$ mit einem MLP-sw $|\mathcal{U}_1| \cdot (\sum_{l=2}^{L}(d^{(l)} - 1) + 1) \overset{1}{\rightarrow} |\mathcal{U}_2| \overset{1}{\rightarrow} \cdots \overset{1}{\rightarrow} |\mathcal{U}_L|$), so stellt man fest, daß auch dieses MLP-sw noch mehr Gewichte als das TDNN hat. Das MLP-sw ist nur funktional äquivalent zum TDNN, wenn für das TDNN $d^{(l)} = 1$ für $l \in \{3, \ldots, L\}$ gilt. Beiden Netzen steht zwar Information aus dem gleichen Zeitraum zur Berechnung der externen Ausgabe zur Verfügung, das TDNN ermöglicht aber eine kompaktere Repräsentation zeitlicher Information, da es auch komplexer strukturierte Zeitinformation erfassen kann [LDL94]: In der ersten verdeckten Schicht werden zeitliche Zusammenhänge zwischen Eingangsmustern (Merkmalen) beschrieben; die Ausgänge der Neuronen dieser Schicht könnte man auch als Meta-Merkmale bezeichnen. In den folgenden Schichten werden dann zeitliche Zusammenhänge zwischen solchen Meta-Merkmalen modelliert. Daher hat nicht nur die Länge des rezeptiven Fensters Einfluß auf das Trainingsergebnis; entscheidend ist auch, in welchen Schichten Aktivierungszustände verzögert werden. Eine experimentelle Untersuchung dieser Fragestellung ist beispielsweise in [CGHC97] zu finden.

Tabelle 2 gibt für verschiedene Netzparadigmen bzw. -strukturen mit gleichem rezeptiven Fenster die Gewichtszahl an. Formeln zur Berechnung der Gewichtszahl lassen sich leicht herleiten bzw. sind in [Wan93a] zu finden. Eine höhere Gewichtszahl bedeutet, daß das neuronale Modell mehr Freiheitsgrade besitzt. Dies kann

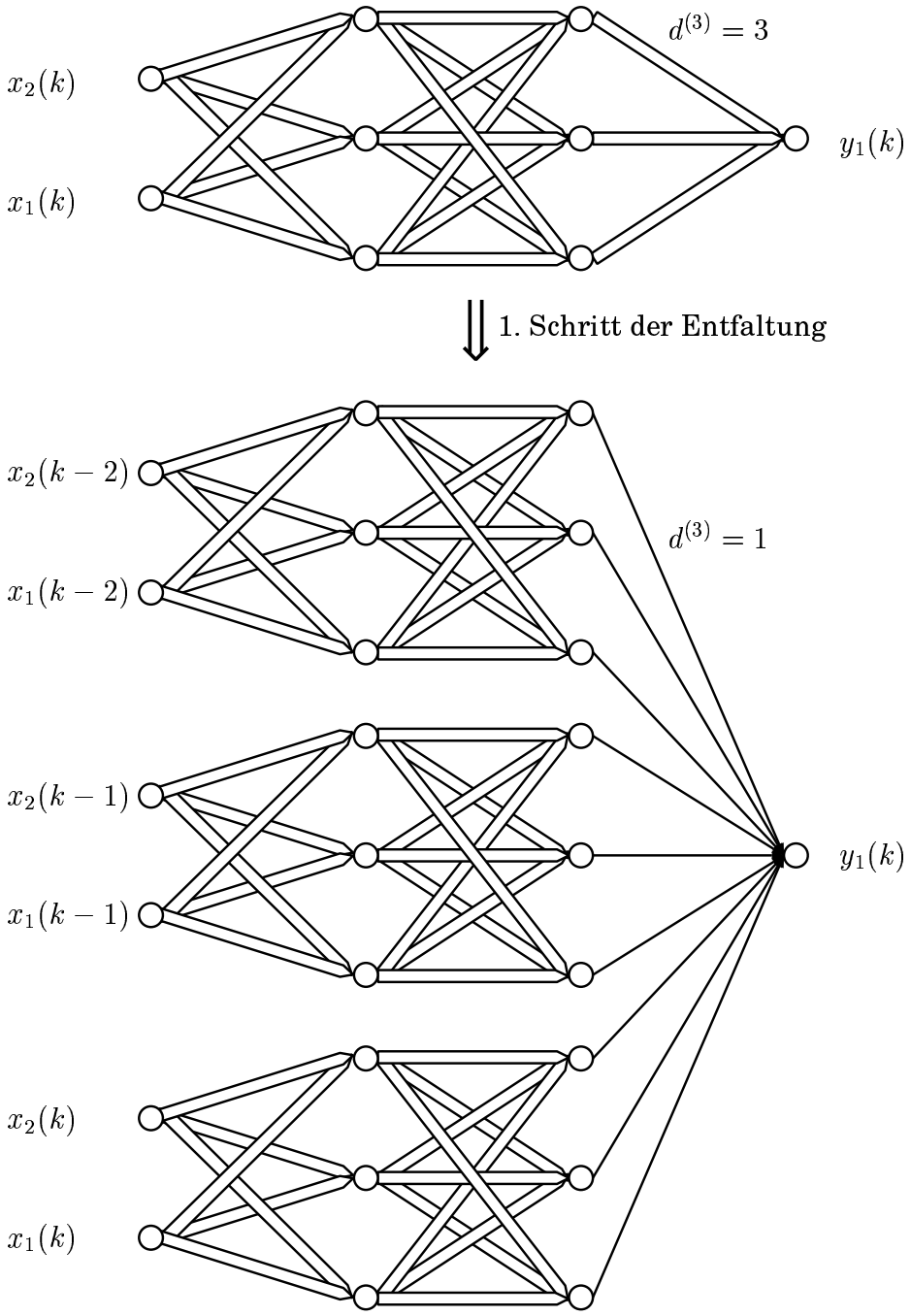


Abbildung 8: Konstruktion eines funktional äquivalenten MLP-sw aus einem TDNN durch zeitliche Entfaltung

verschiedene Nachteile haben, beispielsweise

(1) dauert das Training des Netzes länger,

(2) es werden mehr Trainingsmuster benötigt und

(3) die Generalisierungsfähigkeit des Netzes ist schlechter.

Müssen (z.B. bei der Strukturoptimierung des Netzes) sehr viele Trainingsdurchgänge durchgeführt werden, dann ist die in Punkt (1) angesprochene kurze Trainingsdauer wünschenswert, höhere Schätz- oder Klassifikationsfehler sollten aber deswegen nicht in Kauf genommen werden. Punkt (2) spielt eine besondere Rolle in Anwendungen, in denen Trainingsmuster nur mit großem Aufwand an Kosten (z.B. Personal- und Maschineneinsatz) bereitgestellt werden können. Punkt (3) bedeutet, daß das Netz schlechtere Ergebnisse liefern kann. Speziell auf diesen Punkt wird in Abschnitt 3.1 noch näher eingegangen.

Tabelle 2: Beispiele für die Anzahl von Gewichten in verschiedenen Netzparadigmen bzw. -strukturen mit gleichem rezeptiven Fenster

Struktur des TDNN	**Länge des rez. Fensters**	**Zahl der Gewichte**		
		TDNN	funktional äquivalentes MLP-sw (mit dem Entfaltungsalgorithmus erzeugt)	vollständig verbundenes MLP-sw mit identischer Schichten- und Neuronenanzahl in den verdeckten Schichten und der Ausgabeschicht wie das TDNN
$2 \overset{3}{\rightarrow} 2 \overset{3}{\rightarrow} 2 \overset{3}{\rightarrow} 1$	7	30	150	34
$3 \overset{2}{\rightarrow} 12 \overset{3}{\rightarrow} 1$	4	108	252	156
$5 \overset{10}{\rightarrow} 5 \overset{10}{\rightarrow} 5 \overset{10}{\rightarrow} 1$	28	550	27 550	930
$10 \overset{5}{\rightarrow} 10 \overset{5}{\rightarrow} 3$	9	650	2 650	730

2.3 Verwandte Netzparadigmen

Die Idee, Aktivierungszustände von Vorgängerneuronen zeitverzögert an Nachfolger weiterzugeben, läßt sich auch auf andere, ursprünglich statische Netzparadigmen übertragen. *Radiale-Basisfunktionen-Netze* (*RBF*, siehe u.a. [Bis95, Zel94]) lassen sich mit Zeitverzögerungen zu *TDRBF* kombinieren. Solche Netze werden beispielsweise in [HB98] für die Bildverarbeitung (Erkennung einfacher menschlicher Gesten in Bildsequenzen) oder in [NMSG93] zur Modellierung eines Antriebssystems eingesetzt.

Das Netzparadigma TDNN geht von einer diskreten Zeitachse mit äquidistanten Zeitpunkten aus. Beim *TDNN mit adaptiven Zeitverzögerungen* (*adaptive time-delay neural network*, *ATNN*) wird dagegen eine kontinuierliche Zeitachse angenommen [DD93]. Die Werte der Zeitverzögerungen werden durch ein geeignetes

Lernverfahren eingestellt (siehe beispielsweise [LDL92a, LDL95, LLD93a]). Auch für ATNN wurde ein Algorithmus zur zeitlichen Entfaltung entwickelt [LD95]. Ein praktisches Anwendungsbeispiel im militärischen Bereich ist die in [Lin94] vorgestellte automatische Erkennung fliegender Ziele (Raketen). Selten wird jedoch in einer praktischen Anwendung eine Adaption der Zeitverzögerungen möglich oder sinnvoll sein, so daß der Einsatz von ATNN im Vergleich zu TDNN vorteilhaft wäre. Meist liegen die Merkmale zu äquidistanten Zeitpunkten vor und der lokale Ausschnitt der Mustersequenz, der Einfluß auf den Funktionswert der durch das Netz zu modellierenden Funktion hat (der also idealerweise durch das rezeptive Fenster erfaßt werden sollte), umfaßt häufig relativ wenige Muster, die alle die Netzausgabe mehr oder weniger stark beeinflussen.

Interessant sind auch noch diejenigen Netztypen, bei denen Rekursion und Speichermechanismen kombiniert werden. Bei *IIR-Netzen* werden die FIR-Filter des TDNN durch rekursive Filter mit unendlicher Impulsantwort (*IIR: infinite impulse response*) ersetzt [WB96, WB98, Wan93a]. Bei IIR-Netzen sind allerdings aufgrund der rekursiven Filterstrukturen auch Stabilitätsbetrachtungen notwendig [WB96]. *NARX-Netze* (*NARX: nonlinear autoregressive model with exogenous inputs*) verwenden gleitende Eingabefenster und Rückkopplungen aktueller und verzögerter Netzausgaben auf die Eingabeschicht [LGHK97, LHTG96]. Vorwärts- und rückwärtsgerichtete Verbindungen werden beispielsweise auch in den Netzparadigmen Tempo und Tempo2 eingesetzt [Bod90, BW91].

3 Training von Time-Delay-Netzen

Zur Modellbildung mit Neuronalen Netzen werden Algorithmen benötigt, die eine Bestimmung geeigneter Werte für die Gewichte ermöglichen. Ziel dieses sogenannten *Lernvorganges* (oder *Trainings*) ist es, die Gewichte so zu bestimmen, daß das Netz auf eine angelegte externe Eingabe mit einer gewünschten Ausgabe reagiert. Lernphase und Anwendungsphase sind üblicherweise (so auch hier) zeitlich getrennt, nur in manchen Anwendungen wird das Training während der Anwendungsphase durchgeführt oder fortgesetzt. Man spricht von *offline-* und *online-Training*, was insofern problematisch ist, als diese Begriffe auch mit anderer Bedeutung verwendet werden (siehe Abschnitt 3.1).

In diesem Abschnitt werden zunächst einige Grundbegriffe des Lernens eingeführt, bevor zwei speziell zum Training von TDNN geeignete Lernverfahren vorgestellt werden. Anschließend folgt ein Ausblick auf weitere mögliche Lernverfahren.

3.1 Grundbegriffe des Lernens

In diesem Abschnitt werden die Begriffe Lernaufgabe, Lernalgorithmus und Lernziel sowie das hier zum Training verwendete Fehlermaß definiert.

Definition 3.1 (*Feste Lernaufgabe*)
Eine *feste Lernaufgabe* $\mathcal{L}$ eines TDNN ist eine Sequenz von Eingabe- und Ausgabemusterpaaren $\mathcal{L} \stackrel{def}{=} (\mathbf{x}, \mathbf{t})$, wobei $\mathbf{x}$ nacheinander die Werte $\mathbf{x}(1), \ldots, \mathbf{x}(k), \ldots, \mathbf{x}(K)$ und $\mathbf{t}$ die Werte $\mathbf{t}(1), \ldots, \mathbf{t}(k), \ldots, \mathbf{t}(K)$ mit $K \in \mathbb{N}$ annimmt. Das TDNN soll auf eine Sequenz von Eingabemustern $\mathbf{x}$ mit einer Sequenz von Ausgabemustern $\mathbf{t}$ reagieren (*gewünschte Ausgabe*). Das TDNN *erfüllt* die Lernaufgabe, wenn bei einer *Eingabe* von $\mathbf{x}$ die *tatsächliche Ausgabe* $\mathbf{y}$, d.h. die Sequenz von Ausgabemustern $\mathbf{y}(1), \ldots, \mathbf{y}(k), \ldots, \mathbf{y}(K)$, der *gewünschten Ausgabe* $\mathbf{t}$ entspricht oder mit einer vorgegebenen maximalen Abweichung entspricht.

Prinzipiell kann natürlich auch eine Lernaufgabe aus mehreren Mustersequenzen bestehen.

Andere Netzparadigmen lernen auch mit *freien Lernaufgaben*; dabei werden dem Netz nur Eingabemuster präsentiert. Das Netz erfüllt dann eine freie Lernaufgabe, wenn es zu jeder Eingabe eine Ausgabe so bestimmt, daß im Sinne eines geeigneten Abstandsmaßes ähnliche Eingaben auch ähnliche Ausgaben erzeugen.

Ein Netz kann nur bezüglich der Bewältigung einer festen Lernaufgabe bewertet werden, wenn ein geeignetes Fehlermaß verwendet wird.

Definition 3.2 (*Fehlermaß*)
Sei $\mathcal{L}$ eine feste Lernaufgabe eines TDNN. Der *Fehler* $\mathcal{E}(k)$ *des Netzes zu einem Zeitpunkt* k ist definiert durch die Differenz zwischen tatsächlicher und gewünschter Ausgabe zum Zeitpunkt k:

$$\mathcal{E}(k) \stackrel{def}{=} \langle \mathbf{e}(k) | \mathbf{e}(k) \rangle ;$$

dabei ist $\mathbf{e}(k) \stackrel{def}{=} \left(e_1(k), \ldots, e_{|\mathcal{U}_L|}(k) \right)$ definiert durch $\mathbf{e}(k) \stackrel{def}{=} (\mathbf{t}(k) - \mathbf{y}(k))$ mit dem tatsächlichen Ausgabemuster $\mathbf{y}(k) \stackrel{def}{=} \left(y_1(k), \ldots, y_{|\mathcal{U}_L|}(k) \right)$ und dem gewünschten Ausgabemuster $\mathbf{t}(k) \stackrel{def}{=} \left(t_1(k), \ldots, t_{|\mathcal{U}_L|}(k) \right)$. $\langle \cdot | \cdot \rangle$ ist das Standardskalarprodukt für Vektoren aus $\mathbb{R}^{|\mathcal{U}_L|}$.

Der *gesamte quadrierte Fehler* (kurz: *Gesamtfehler*) *für eine Lernaufgabe* $\mathcal{L}$ (*least-squares-error (LSE), sum-of-squares-error*) ist dann bestimmt durch

$$\mathcal{E}_{LSE} \stackrel{def}{=} \frac{1}{2} \cdot \sum_{k=1}^{K} \mathcal{E}(k).$$

Sehr häufig wird auch der *mittlere quadratische Fehler* (*mean-squared-error, MSE* oder *least-mean-squares, LMS*) verwendet, der folgendermaßen definiert ist:

$$\mathcal{E}_{MSE} \stackrel{def}{=} \frac{1}{2 \cdot K \cdot |\mathcal{U}_L|} \cdot \sum_{k=1}^{K} \mathcal{E}(k).$$

Die angegebenen Fehlermaße sind über den euklidischen Abstand von gewünschter und tatsächlicher Ausgabe definiert und „bestrafen“ somit größere Abweichungen an einzelnen Neuronen deutlich stärker als kleinere. Andere Fehlermaße, wie beispielsweise

- ein auf beliebigen anderen Normen (z.B. der Betragsbildung statt der Quadrierung) basierendes Fehlermaß (Minkowski-Fehler),
- der Cross-Entropy Fehler oder
- der exponentielle Fehler

könnten prinzipiell auch verwendet werden (siehe beispielsweise [Bis95, NKK96, She97]).

Mit einem geeigneten Fehlermaß kann der Fortschritt beim *Training* des Netzes (*Lernen*) mit Hilfe eines *Lernalgorithmus* bewertet werden.

Definition 3.3 (*Überwachter Lernalgorithmus*)
Ein *Lernalgorithmus* ist ein Verfahren, das anhand einer Lernaufgabe die Gewichte eines Neuronalen Netzes verändert. Erfüllt das Netz diese Lernaufgabe nach Anwendung des Verfahrens, d.h., wird z.B. eine vorher definierte Fehlerschranke unterschritten, so war die Anwendung des Lernalgorithmus *erfolgreich*, anderenfalls *erfolglos*. Ein Lernalgorithmus, der eine feste Lernaufgabe verwendet, heißt *überwachter Lernalgorithmus*.

Im Gegensatz zu überwachten Lernalgorithmen arbeiten *unüberwachte* oder *selbstorganisierende* Lernalgorithmen mit freien Lernaufgaben.

Bei überwachten Lernalgorithmen kann man weiter zwei prinzipielle Vorgehensweisen unterscheiden. Beim *musterweisen Lernen* (*online*-Lernen) werden die Gewichte des Netzes jeweils nach der Präsentation eines Eingangsmusters aus der Lernaufgabe verändert. Beim *epochenweisen Lernen* (*offline*-Lernen) werden die Gewichte erst nach der Präsentation aller Muster einer Lernaufgabe modifiziert. Während bei der ersten Vorgehensweise der Fehler $\mathcal{E}(k)$ verringert wird (was nicht notwendigerweise eine unmittelbare Verringerung von $\mathcal{E}$ ($\mathcal{E}_{LSE}$ oder $\mathcal{E}_{MSE}$) bedeutet, aber oft die Konvergenzeigenschaften des Verfahrens verbessert), hat die zweite Vorgehensweise

das Ziel, den Gesamtfehler $\mathcal{E}$ zu verringern. Die Bearbeitung aller Muster einer Lernaufgabe nennt man auch *Epoche*.

Sind wie beim TDNN zeitliche Zusammenhänge zu modellieren (wirkt sich also eine Eingabe nicht nur auf den aktuellen Fehler aus), so ist im allgemeinen epochenweises Lernen vorzuziehen. Eine Zwischenform beider Lernprinzipien wird in [LDL92a, LDL92b] für TDNN vorgeschlagen: Besteht eine Lernaufgabe aus mehreren Mustersequenzen, so kann auch eine Modifikation der Gewichte (Gewichtsupdate) jeweils nach der Präsentation aller Muster einer Sequenz erfolgen. Diese Form des Lernens, die in der Vorgehensweise äquivalent zum epochenweisen Lernen ist, wird als *inkrementelles Lernen* bezeichnet. Dabei ist allerdings zu beachten, daß bei verschieden langen Mustersequenzen der Einfluß der einzelnen Muster auf das Trainingsergebnis unterschiedlich sein kann.

Die gängigen überwachten Lernalgorithmen nähern einen Gradientenabstieg in der Fehlerfläche an und versuchen, auf diese Weise den Gesamtfehler zu minimieren. Der Startpunkt in der Fehlerfläche wird dabei zufällig gewählt, d.h., er wird bestimmt durch zufällig initialisierte Gewichte zu Beginn des Lernvorgangs und durch die Lernaufgabe $\mathcal{L}$. Die Konvergenz derartiger nichtlinearer Optimierungsverfahren ist allerdings keineswegs garantiert, da die Fehlerfunktion im allgemeinen nicht konvex ist.

Algorithmus 3.4 (*Musterweises Lernen*)

(1) Initialisiere alle Gewichte $w_{i,j}^{(l)}(m)$ des Neuronalen Netzes mit zufällig gewählten, kleinen Werten.

(2) Für alle Muster (d.h. $k = 1, \ldots, K$) einer festen Lernaufgabe $\mathcal{L} = (\mathbf{x}, \mathbf{t})$ führe die folgenden Schritte aus (der Zeitraum für die Ausführung dieser Schritte wird als *Epoche* p bezeichnet):

   (a) Lege $\mathbf{x}(k)$ als externe Eingabe an die Neuronen der Eingabeschicht an.

   (b) Propagiere diese Eingabe durch alle Schichten des Netzes bis zur Ausgabeschicht und bestimme die tatsächliche Ausgabe des Netzes $\mathbf{y}(k)$.

   (c) Berechne den Fehler $\mathcal{E}(k)$ und addiere $\mathcal{E}(k)$ zum Gesamtfehler $\mathcal{E}$.

   (d) Wenn $\mathcal{E}(k) > 0$ ist, so berechne für jedes Gewicht $w_{i,j}^{(l)}(m)$ eine *Gewichtsänderung für das k-te Muster der Lernaufgabe* $\nabla_k w_{i,j}^{(l)}(m)$ und ändere jedes Gewicht um diesen Wert.

(3) Prüfe, ob $\mathcal{E} < \theta$ (wobei $\theta \in \mathbb{R}^+$ eine zuvor geeignet gewählte Fehlerschranke ist). Wenn dies der Fall ist, gilt die Lernaufgabe als erfüllt. Anderenfalls setze das Lernen bei Schritt (2) fort (nächste Epoche).

Die Gewichtsänderung für das k-te Muster der Lernaufgabe (Gewichtswert nach der Änderung abzüglich des Gewichtswertes vor der Änderung) wird dabei so bestimmt,

daß bei einer sofortigen erneuten Präsentation des Musters $\mathbf{x}(k)$ eine Verringerung des Fehlers zum Zeitpunkt k zu erwarten wäre. Insbesondere der Schritt (2d) dieser bisher noch recht allgemeinen Beschreibung eines Lernalgorithmus wird in den folgenden Abschnitten präzisiert; dasselbe gilt für den folgenden Algorithmus.

Algorithmus 3.5 (*Epochenweises Lernen*)

(1) Initialisiere alle Gewichte $w_{i,j}^{(l)}(m)$ des Neuronalen Netzes mit zufällig gewählten, kleinen Werten.

(2) Für alle Muster (d.h. $k = 1, \ldots, K$) einer festen Lernaufgabe $\mathcal{L} = (\mathbf{x}, \mathbf{t})$ führe die folgenden Schritte aus (der Zeitraum für die Ausführung dieser Schritte wird als *Epoche* p bezeichnet):

   (a) Lege $\mathbf{x}(k)$ als externe Eingabe an die Neuronen der Eingabeschicht.

   (b) Propagiere diese Eingabe durch alle Schichten des Netzes bis zur Ausgabeschicht und bestimme die tatsächliche Ausgabe des Netzes $\mathbf{y}(k)$.

   (c) Berechne den Fehler $\mathcal{E}(k)$ und addiere $\mathcal{E}(k)$ zum Gesamtfehler $\mathcal{E}$.

   (d) Wenn $\mathcal{E}(k) > 0$ ist, so berechne die Gewichtsänderungen für das k-te Muster der Lernaufgabe $\nabla_k w_{i,j}^{(l)}(m)$ und akkumuliere diese Änderungen über den Zeitraum einer Epoche, ohne die Gewichte tatsächlich zu modifizieren.

(3) Modifiziere jedes Gewicht um eine *Gewichtsänderung für die Epoche* p $\nabla^p w_{i,j}^{(l)}(m)$. Der Wert dieser Gewichtsänderung entspricht dem akkumulierten Wert der einzelnen Gewichtsänderungen $\nabla_k w_{i,j}^{(l)}(m)$ direkt oder ist aus diesem abgeleitet.

(4) Prüfe, ob $\mathcal{E} < \theta$ (wobei $\theta \in \mathbb{R}^+$ eine zuvor geeignet gewählte Fehlerschranke ist). Wenn dies der Fall ist, gilt die Lernaufgabe als erfüllt. Anderenfalls setze das Lernen bei Schritt (2) fort (nächste Epoche).

Hier sollen insbesondere epochenweise lernende Algorithmen vorgestellt werden. Die Wahl einer geeigneten Fehlerschranke θ hängt von der betrachteten Anwendung ab. Daher wird anstelle der Verwendung einer Fehlerschranke häufig eine zuvor festgelegte Zahl von Epochen trainiert. In anderen Anwendungen wird manchmal auch kein Endekriterium bestimmt und die Gewichte werden auch während der Anwendungsphase des Netzes adaptiert. Auf diese Weise kann man eine kontinuierliche Anpassung des Neuronalen Netzes an eine sich laufend ändernde Umgebung erreichen.

Ist nun nach der erfolgreichen Erfüllung einer Lernaufgabe (d.h., eine Fehlerschranke wurde unterschritten) das trainierte Netz in einer konkreten Anwendung (aus der die Muster der Lernaufgabe ermittelt wurden) nutzbringend einsetzbar? Dies hängt von dem verfolgten Lernziel ab!

Definition 3.6 (*Lernziele*)
Beim Lernziel der *Approximation* soll die durch die Musterpaare der Lernaufgabe $\mathcal{L} = (\mathbf{x}, \mathbf{t})$ beschriebene nichtlineare Funktion durch ein TDNN so angenähert werden, daß nach dem Training des Netzes bei einer erneuten Eingabe der Mustersequenz $\mathbf{x}$ die Differenz zwischen tatsächlicher Ausgabe $\mathbf{y}$ und erwünschter Ausgabe $\mathbf{t}$ möglichst klein wird.

Beim Lernziel der *Generalisierung* soll die durch die Musterpaare der Lernaufgabe $\mathcal{L} = (\mathbf{x}, \mathbf{t})$ beschriebene nichtlineare Funktion durch ein TDNN so angenähert werden, daß nach dem Training des Netzes für beliebige (andere) Sequenzen von Musterpaaren $(\mathbf{x}', \mathbf{t}')$ der nichtlinearen Funktion bei einer Eingabe der Mustersequenz $\mathbf{x}'$ die Differenz zwischen tatsächlicher Ausgabe $\mathbf{y}'$ und erwünschter Ausgabe $\mathbf{t}'$ möglichst klein wird.

Die zum Einstellen der Gewichte verwendeten Muster der Lernaufgabe werden als *Trainings-* oder *Lernmuster* (bzw. *Trainings-* oder *Lerndaten*), die zur Überprüfung der Generalisierungsleistung benutzten als *Testmuster* (bzw. *Testdaten*) bezeichnet. Trainings- und Testmustermengen sind üblicherweise disjunkt; man spricht daher auch von *unbekannten Testmustern* und *extrapolierenden Tests*.

Das Lernziel der Approximation ist im allgemeinen erreicht, wenn eine Lernaufgabe erfüllt ist. Verwendet man ausreichend viele verdeckte Neuronen, so sind die hier definierten MLP mit nur einer Schicht verdeckter Neuronen in der Lage, Funktionen beliebig genau zu approximieren [HSW89]; eine ähnliche Aussage wurde übrigens auch für ATNN bewiesen [LD95].

Eine gute Generalisierungsfähigkeit zu erzielen, ist deutlich schwieriger. Zur Veranschaulichung kann eine ähnliche Aufgabenstellung dienen: Eine Funktion $f : \mathbb{R} \to \mathbb{R}$ sei durch eine endliche Menge von Punkten $\mathcal{P} = \{(x, f(x)) | x \in [a, b] \text{ mit } a, b \in \mathbb{R}\}$ gegeben. Diese Menge von Punkten soll durch ein Polynom $p : \mathbb{R} \to \mathbb{R}$ so approximiert werden (Fehlermaß sei der gesamte quadrierte Fehler), daß die Auswertung von p auch für Werte zwischen den Stützstellen (d.h. für beliebige $y \in [a, b]$) „sinnvolle“ Werte ergibt, also die Differenz $|f(y) - p(y)|$ klein wird. Das Ziel wird sicher nicht erreicht, wenn der Polynomgrad zu niedrig ist; in diesem Fall kann p das Verhalten von f im allgemeinen nur unzureichend nachbilden. Ist der Polynomgrad zu hoch (insbesondere im Interpolationsfall), so treten im allgemeinen zwischen den Stützstellen Oszillationen auf und das Verhalten von p zwischen den Stützstellen wird somit unkontrollierbar. Bei der Wahl des Polynomgrades muß also ein geeigneter Kompromiß gefunden werden. Bei Neuronalen Netzen ist es ganz ähnlich; anstelle eines Polynomgrades hat man Freiheitsgrade des neuronalen Modells zu wählen: die Anzahl verdeckter Schichten bzw. Neuronen und insbesondere die Gewichtsanzahl (vgl. dazu auch die Bemerkungen zum Vorteil von TDNN gegenüber MLP-sw am Ende von Abschnitt 2.2). Zu beachten ist auch, daß in bestimmten Anwendungen die Trainingsmuster durch Rauschen und unbekannte systematische Störungen stark beeinflußt sein können.

Zurück zum Beispiel der Polynomapproximation: Üblicherweise wird nicht verlangt, daß der Approximationsfehler (d.h. die Differenz $|f(y) - p(y)|$) auch für beliebige Werte $y \ll a$ oder $y \gg b$ klein ist. Genau diese Form der Generalisierung wird jedoch in vielen Anwendungen Neuronaler Netze untersucht (siehe z.B. [Sic00]). In diesem Fall wäre es wohl besser, die Stützstellenanzahl zu erhöhen, d.h., geeignete zusätzliche Trainingsmuster bereitzustellen.

In den meisten technischen Anwendungen von TDNN wird das Lernziel der Generalisierung verfolgt. Dazu muß der Bereich prinzipiell möglicher Sequenzen von Eingabe- und Ausgabemusterpaaren möglichst gut durch Lernmuster abgedeckt sein. Durch den Lernvorgang soll erreicht werden, daß das TDNN die Lernaufgabe verallgemeinert, d.h. generalisiert. Natürlich genügt es beim Lernziel der Generalisierung nicht, nach dem Training die Approximationsleistung des trainierten Netzes zu testen. Genau dies ist jedoch leider in vielen Anwendungen der Fall (siehe z.B. [Sic00]).

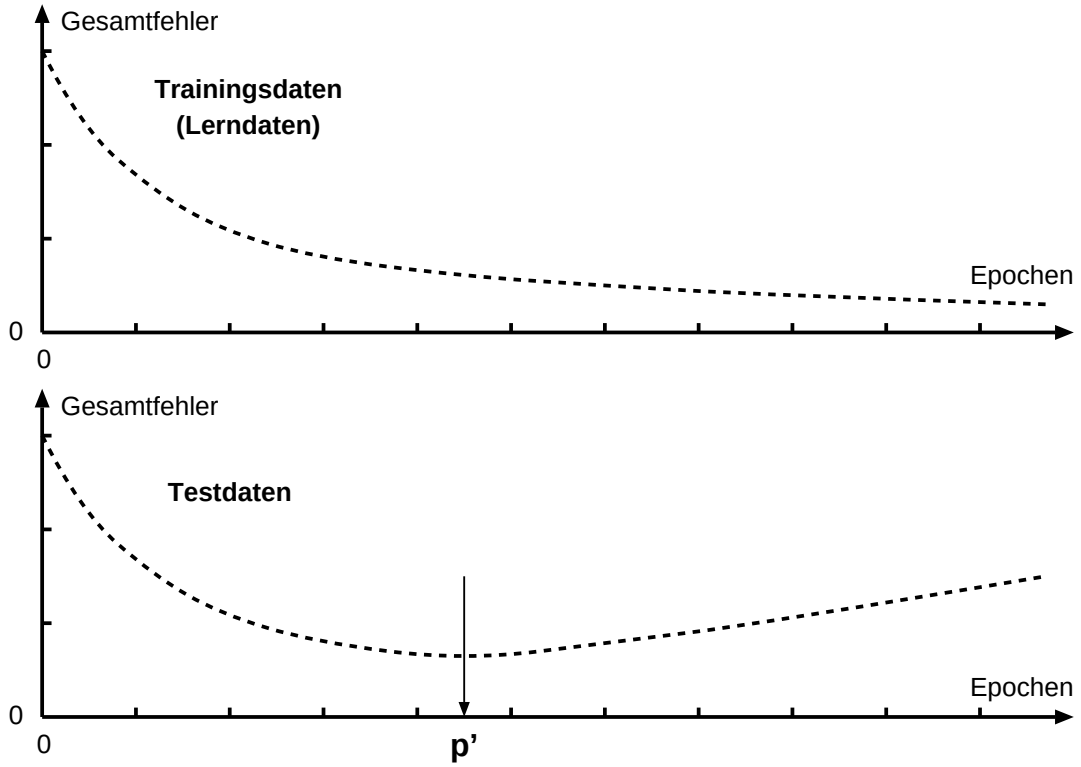


Abbildung 9: Überanpassung Neuronaler Netze an Trainingsdaten

Untersucht man die Entwicklung der Gesamtfehler für Lern- und unbekannte Testmuster während der Anwendung des Lernalgorithmus genauer, so kann der in Abbildung 9 (schematische Darstellung; keine realen Daten) gezeigte Effekt zu beobachten sein, wenn die Zahl der Freiheitsgrade des neuronalen Modells zu hoch ist. Während der Gesamtfehler für Lernmuster kontinuierlich sinkt, steigt der Gesamtfehler für unbekannte Testmuster nach einer Reihe von Epochen wieder an. Dieser Effekt wird als *Überanpassung* (*overfitting*) des Neuro-

nalen Netzes an die Trainingsdaten bezeichnet. Die Vermeidung einer solchen Überanpassung ist eine der wichtigsten Aufgaben bei der Entwicklung des Verschleißüberwachungssystems auf der Basis von TDNN. Die bekanntesten Maßnahmen dazu sind [Bis95, CU93, Hay94, Roj96, Sar96, SDB97, Zel94]:

- **Regularisierungs-Verfahren**: Dabei wird ein zusätzlicher „Strafterm“ zur Fehlerfunktion addiert. Ein Beispiel ist das Verfahren *weight decay*, bei dem versucht wird, die Gewichte eines Netzes betragsmäßig klein zu halten.
- **Bereitstellung ausreichend vieler Trainingsmuster**: Wieviele Muster ausreichend sind, hängt natürlich von der Problemstellung ab. Heuristische Formeln setzen die Musteranzahl in Beziehung zur Anzahl verdeckter Neuronen und / oder zur Gewichtsanzahl. Beispielsweise wird gefordert, 30 Mal mehr Trainingsmuster als Gewichte zu verwenden. In der Praxis ist jedoch die Trainingsmustermenge oft vorgegeben oder es sind Trainingsmuster nur mit erheblichem Kostenaufwand (Personal, Maschinenzeiten usw.) zu beschaffen.
- **Früher Abbruch des Trainings** (*early stopping*): Das Training wird abgebrochen, wenn der Gesamtfehler für Testdaten ein Minimum erreicht hat (Epoche p' in Abbildung 9). Dieses Verfahren ist äußerst problematisch, da die Gewichte im Hinblick auf optimale Testergebnisse eingestellt werden. D.h., eine dritte, unabhängige Datenmenge zur Validierung ist nötig.
- **Training mit verrauschten Daten**: Trainingsdaten werden mit einem *weißen Rauschen* mit Mittelwert Null und fester Varianz überlagert. Der Wert der Varianz könnte sich beispielsweise an der Höhe systematischer Störungen der Daten orientieren, z.B. an der maximalen Abweichung einer nichtlinearen Sensorkennlinie von der Idealkennlinie (wenn nur dieser Wert, aber nicht der genaue Verlauf der Nichtlinearität bekannt ist).
- **Reduktion der Dimension des Eingaberaums**: Obwohl diese Maßnahme in der Literatur kaum genannt wird, ist sie die mit Abstand wichtigste von allen. Insbesondere die Kombination von vielen Freiheitsgraden (d.h. Gewichten) und einem Übermaß an Eingangsinformation oder intern gespeicherter Information in einem TDNN macht eine Überanpassung leicht möglich (siehe z.B. [Sic00]). Die geeignete Vorverarbeitung der zu verarbeitenden Signale bzw. Werte dient somit der Reduktion der Zahl signifikanter Merkmale (Informationsverdichtung) und auch der Reduktion der zum Training erforderlichen Musterzahl. Die in der Literatur gelegentlich empfohlene Verwendung eines Vektors von Hauptkomponenten des Eingaberaums (*principal components*) anstelle eines höherdimensionalen Merkmalsvektors kann sich auch als ungeeignet erweisen (siehe z.B. [Kei98]).
- **Optimierung der Netzstruktur**: Neben der Adaption der Gewichte, einer kontinuierlichen Optimierungsaufgabe, sind bei der Anwendung von TDNN auch noch verschiedene diskrete Optimierungsaufgaben zu lösen:

- die Wahl einer geeigneten Zahl verdeckter Schichten,
- die Bestimmung einer optimalen Anzahl von Neuronen in den verdeckten Schichten,
- die Festlegung einer Verbindungsstruktur und
- die Suche nach den besten Werten für die Anzahl von Zeitschritten, um die die Aktivierungszustände in Speicherelementen zu verzögern sind.

Die genannten Fragestellungen betreffen die Netztopologie (Netzstruktur) und sie sind von essentieller Bedeutung für die Modellbildung mit TDNN (bzw. anderen Netzen). Die für die Optimierung der Netzstuktur bekannten Verfahren (im allgemeinen angewandt für Mehrlagige Perzeptren) lassen sich in drei Klassen einteilen: destruktive, konstruktive und hybride Verfahren. Mögliche Lösungsansätze werden in Abschnitt 4 noch angesprochen.

- **Kombination mehrerer Netzausgaben**: Bei dieser Gruppe von Verfahren werden Ausgaben mehrerer Neuronaler Netze zu einer Gesamtaussage kombiniert. Die Netze haben dabei entweder alle die gleiche Aufgabe oder sie bearbeiten verschiedene Teilprobleme. Im ersten Fall spricht man von *Netzensembles* oder *„Committee "-Verfahren*; im zweiten Fall von *„Mixture of Experts "-Verfahren*.

Weitere Maßnahmen zur Vermeidung einer Überanpassung werden in [Bis95, SDB97] angesprochen; auf die spezielle Problematik bei Klassifikationsaufgaben wird in [Chi95] eingegangen.

3.2 Temporal Backpropagation

Wie wird eine Gewichtsänderung für das k-te Muster einer Lernaufgabe $\nabla_k w_{i,j}^{(l)}(m)$ (Schritt (2d) in Algorithmus 3.4 und Algorithmus 3.5) bzw. eine Gewichtsänderung für die p-te Epoche $\nabla^p w_{i,j}^{(l)}(m)$ (Schritt (3) in Algorithmus 3.5) bestimmt? Der bekannteste Lernalgorithmus für MLP ist *Backpropagation* (oder *verallgemeinerte Delta-Regel*; siehe z.B. [NKK96, Zel94]), der sowohl für musterweises als auch für epochenweises Lernen verwendet werden kann. Das Verfahren nähert einen Gradientenabstieg an. Bei TDNN sind jedoch die Auswirkungen aktueller Eingaben auf mehrere (aktuelle und zukünftige) Ausgaben und Fehler zu berücksichtigen, weshalb auch kein musterweises Training durchgeführt werden kann. Ein geeigneter Lernalgorithmus mit dem Namen *Temporal Backpropagation* (siehe z.B. [Wan93a, Wan93c]) wurde erst einige Jahre nach den ersten Arbeiten über TDNN vorgestellt. Anfänglich wurden TDNN in ihr zeitlich entfaltetes Äquivalent überführt und mit Standard-Backpropagation trainiert (siehe z.B. [WHH+89]). Nachteile dieser Methode sind der deutlich höhere Rechenaufwand (mehr Neuronen und Gewichte) und die Verletzung des *Lokalitätsprinzips*. D.h., gemeinsame

Gewichte erfordern eine globale Betrachtung, da die Gewichtsänderungen nicht unabhängig voneinander erfolgen dürfen.

Der Algorithmus Temporal Backpropagation wird für epochenweises oder inkrementelles Lernen verwendet. Im folgenden werden alle Gewichte betrachtet, die keine Schwellwerte modellieren; Gewichte für Schwellwerte können auf analoge Weise adaptiert werden.

Ziel des Algorithmus ist die Minimierung des Fehlers $\mathcal{E}$ (hier $\mathcal{E}_{LSE}$) durch Veränderung des Gewichte des Netzes. Für die Änderungen eines Gewichtsvektors (Koeffizienten eines FIR-Filters) gilt für das k-te Muster einer Lernaufgabe

$$\nabla_k \mathbf{w}_{i,j}^{(l)} \propto -\frac{\partial \mathcal{E}}{\partial \mathbf{w}_{i,j}^{(l)}}. \tag{25}$$

Dabei sei $\nabla_k \mathbf{w}_{i,j}^{(l)}$ ein Vektor von einzelnen Gewichtsänderungen $\nabla_k w_{i,j}^{(l)}(m)$. Der Gradient der Fehlerfunktion ist (unter Verwendung von Definition 3.2) gegeben durch

$$\frac{\partial \mathcal{E}}{\partial \mathbf{w}_{i,j}^{(l)}} = \frac{\partial \left(\frac{1}{2} \cdot \sum_{k=1}^{K} \langle \mathbf{e}(k) | \mathbf{e}(k) \rangle \right)}{\partial \mathbf{w}_{i,j}^{(l)}} \tag{26}$$

$$= \sum_{k=1}^{K} \frac{1}{2} \cdot \frac{\partial \langle \mathbf{e}(k) | \mathbf{e}(k) \rangle}{\partial \mathbf{w}_{i,j}^{(l)}}. \tag{27}$$

Jeder Summand auf der rechten Seite von Gleichung 27 entspricht einer momentanen (aktuellen) Schätzung des tatsächlichen Gradienten. In dieser Gleichung wird der Gewichtsvektor als konstant angenommen. Dies wäre eine grobe Näherung, wenn die Gewichte nach jedem Schritt angepaßt würden. Da aber der Lernalgorithmus für epochenweises Lernen verwendet wird, approximiert die über K Muster akkumulierte Gewichtsänderung den tatsächlichen Gradienten recht gut (siehe z.B. [Wan93a]).

Der Gradient läßt sich unter Verwendung der Kettenregel auch darstellen als

$$\frac{\partial \mathcal{E}}{\partial \mathbf{w}_{i,j}^{(l)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{E}}{\partial s_j^{(l)}(k)} \cdot \frac{\partial s_j^{(l)}(k)}{\partial \mathbf{w}_{i,j}^{(l)}}. \tag{28}$$

Der erste Faktor jedes Summanden beschreibt die Änderung des Fehlers bezüglich einer Änderung der Netzeingabe für ein Neuron, der zweite Faktor repräsentiert den Effekt, den die Änderung eines bestimmten Gewichts auf diese Netzeingabe hat. Im allgemeinen gilt

$$\frac{\partial \mathcal{E}}{\partial s_j^{(l)}(k)} \cdot \frac{\partial s_j^{(l)}(k)}{\partial \mathbf{w}_{i,j}^{(l)}} \neq \frac{1}{2} \cdot \frac{\partial \langle \mathbf{e}(k) | \mathbf{e}(k) \rangle}{\partial \mathbf{w}_{i,j}^{(l)}}, \tag{29}$$

nur die Summe über alle k ist gleich [Wan93a].

Temporal Backpropagation bestimmt nun die Gewichtsänderungen für das k-te Muster der Lernaufgabe nach der Vorschrift

$$\nabla_k \mathbf{w}_{i,j}^{(l)} = - \left(\frac{\partial \mathcal{E}}{\partial s_j^{(l)}(k)} \cdot \frac{\partial s_j^{(l)}(k)}{\partial \mathbf{w}_{i,j}^{(l)}} \right). \tag{30}$$

Beim epochenweisen Lernen werden die Gewichtsänderungen für die einzelnen Muster der Lernaufgabe $\nabla_k \mathbf{w}_{i,j}^{(l)}$ akkumuliert und erst am Ende einer Epoche wird tatsächlich eine Gewichtsänderung $\nabla^p \mathbf{w}_{i,j}^{(l)}$ durchgeführt.

Der Faktor $\dfrac{\partial s_j^{(l)}(k)}{\partial \mathbf{w}_{i,j}^{(l)}}$ aus Gleichung 30 läßt sich nun mit Hilfe von Definition 2.1(3) durch

$$\frac{\partial s_j^{(l)}(k)}{\partial \mathbf{w}_{i,j}^{(l)}} = \frac{\partial \left(\sum\limits_{v \in \mathcal{U}_{l-1}} \left\langle \mathbf{w}_{v,j}^{(l)} \middle| \mathbf{a}_v^{(l-1)}(k) \right\rangle + w_{b,j}^{(l)} \right)}{\partial \mathbf{w}_{i,j}^{(l)}} \tag{31}$$

$$= \frac{\partial \left\langle \mathbf{w}_{i,j}^{(l)} \middle| \mathbf{a}_i^{(l-1)}(k) \right\rangle}{\partial \mathbf{w}_{i,j}^{(l)}} \tag{32}$$

$$= \mathbf{a}_i^{(l-1)}(k) \tag{33}$$

bestimmen, da $\mathbf{w}_{i,j}^{(l)}$ nur über die Verbindung zwischen den Neuronen i und j Einfluß auf die Netzeingabe $s_j^{(l)}(k)$ hat.

Mit der Abkürzung

$$\delta_j^{(l)}(k) \stackrel{def}{=} -\frac{\partial \mathcal{E}}{\partial s_j^{(l)}(k)} \tag{34}$$

und Gleichung 33 kann man nun Gleichung 30 neu schreiben als

$$\nabla_k \mathbf{w}_{i,j}^{(l)} = \delta_j^{(l)}(k) \cdot \mathbf{a}_i^{(l-1)}(k). \tag{35}$$

Nun muß eine Berechnungsvorschrift zur Bestimmung des sogenannten *Fehlersignals* $\delta_j^{(l)}(k)$ gefunden werden. Dabei wird unterschieden, ob dieses Fehlersignal für die Ausgabeschicht oder eine der verdeckten Schichten zu bestimmen ist.

Im ersten Fall ist $l = L$ und es gilt (da $\partial s_j^{(L)}(k)$ nur die aktuellen Ausgabefehler des Netzes zum Zeitpunkt k beeinflußt):

$$\delta_j^{(L)}(k) = -\frac{\partial \mathcal{E}}{\partial s_j^{(L)}(k)} \tag{36}$$

$$
\begin{aligned}
(37) \qquad &= -\frac{\partial\left(\frac{1}{2}\sum_{k=1}^{K}\langle \mathbf{e}(k)|\mathbf{e}(k)\rangle\right)}{\partial s_j^{(L)}(k)} \\
(38) \qquad &= -\frac{1}{2}\frac{\partial e_j^2(k)}{\partial s_j^{(L)}(k)} \\
(39) \qquad &= -\frac{1}{2}\frac{\partial e_j^2(k)}{\partial a_j^{(L)}(k)}\cdot\frac{\partial a_j^{(L)}(k)}{\partial s_j^{(L)}(k)} \\
(40) \qquad &= -\frac{1}{2}\frac{\partial\left(t_j(k)-y_j(k)\right)^2}{\partial a_j^{(L)}(k)}\cdot\sigma'\left(s_j^{(L)}(k)\right) \\
(41) \qquad &= -\frac{1}{2}\frac{\partial\left(t_j(k)-a_j^{(L)}(k)\right)^2}{\partial a_j^{(L)}(k)}\cdot\sigma'\left(s_j^{(L)}(k)\right) \\
(42) \qquad &= \left(t_j(k)-a_j^{(L)}(k)\right)\cdot\sigma'\left(s_j^{(L)}(k)\right).
\end{aligned}
$$

Dabei wurden die Definitionen 2.1(4), 2.1(5) und 3.2 verwendet. An dieser Stelle wird auch klar, warum der Gesamtfehler (Definition 3.2) üblicherweise mit dem Faktor 0.5 definiert wird.

In den verdeckten Schichten ($l \in \{2,\ldots,L-1\}$) beeinflußt $\partial s_j^{(l)}(k)$ den Gesamtfehler indirekt über alle Netzeingaben der folgenden Schicht $\partial s_j^{(l+1)}(k)$. Wegen der Verzögerungen der Aktivierungen hat eine Änderung von $\partial s_j^{(l)}(k)$ aber nicht nur Auswirkungen auf den Fehler zum Zeitpunkt k, sondern auch auf Fehler zu anderen Zeitpunkten. Genauer gesagt, wird der Fehler indirekt über alle Netzeingaben der folgenden Schicht zwischen den Zeitpunkten k und $k+d^{(l+1)}-1$ entsprechend der maximalen Verzögerung zwischen den Schichten l und $l+1$ beeinflußt. Daher folgt wiederum mit Hilfe der Kettenregel:

$$
\begin{aligned}
(43) \qquad \delta_j^{(l)}(k) &= -\frac{\partial\mathcal{E}}{\partial s_j^{(l)}(k)} \\
(44) \qquad &= -\sum_{u\in\mathcal{U}_{l+1}}\sum_{m=0}^{d^{(l+1)}-1}\frac{\partial\mathcal{E}}{\partial s_u^{(l+1)}(k+m)}\cdot\frac{\partial s_u^{(l+1)}(k+m)}{\partial s_j^{(l)}(k)} \\
(45) \qquad &= \sum_{u\in\mathcal{U}_{l+1}}\sum_{m=0}^{d^{(l+1)}-1}\delta_u^{(l+1)}(k+m)\cdot\frac{\partial s_u^{(l+1)}(k+m)}{\partial s_j^{(l)}(k)}.
\end{aligned}
$$

Für den zweiten Faktor in den Summanden aus Gleichung 45 gilt nach Definition

2.1(3) und 2.1(4)

$$\frac{\partial s_u^{(l+1)}(k+m)}{\partial s_j^{(l)}(k)} = \frac{\partial s_u^{(l+1)}(k+m)}{\partial a_j^{(l)}(k)} \cdot \frac{\partial a_j^{(l)}(k)}{\partial s_j^{(l)}(k)} \tag{46}$$

$$= \frac{\partial\left(\sum_{v\in\mathcal{U}_l}\left\langle \mathbf{w}_{v,u}^{(l+1)} \middle| \mathbf{a}_v^{(l)}(k+m)\right\rangle + w_{b,u}^{(l+1)}\right)}{\partial a_j^{(l)}(k)} \cdot \sigma'\left(s_j^{(l)}(k)\right) \tag{47}$$

$$= \frac{\partial\left\langle \mathbf{w}_{j,u}^{(l+1)} \middle| \mathbf{a}_j^{(l)}(k+m)\right\rangle}{\partial a_j^{(l)}(k)} \cdot \sigma'\left(s_j^{(l)}(k)\right), \tag{48}$$

da $a_j^{(l)}(k)$ nur über die Verbindung mit dem Gewichtsvektor $\mathbf{w}_{j,u}^{(l+1)}$ auf die Netzeingaben $s_u^{(l+1)}(k+m)$ der Folgeschicht Einfluß hat.

Der erste Faktor von Gleichung 48 ist

$$\frac{\partial\left\langle \mathbf{w}_{j,u}^{(l+1)} \middle| \mathbf{a}_j^{(l)}(k+m)\right\rangle}{\partial a_j^{(l)}(k)} = \begin{cases} w_{j,u}^{(l+1)}(m) & \text{für } 0 \le m \le d^{(l+1)}-1 \\ 0 & \text{sonst} \end{cases}. \tag{49}$$

Daher können die Gleichungen 43 bis 45 wie folgt fortgesetzt werden:

$$\delta_j^{(l)}(k) = \sum_{u\in\mathcal{U}_{l+1}} \sum_{m=0}^{d^{(l+1)}-1} \delta_u^{(l+1)}(k+m) \cdot w_{j,u}^{(l+1)}(m) \cdot \sigma'\left(s_j^{(l)}(k)\right) \tag{50}$$

$$= \sigma'\left(s_j^{(l)}(k)\right) \cdot \sum_{u\in\mathcal{U}_{l+1}} \left\langle \mathbf{w}_{j,u}^{(l+1)} \middle| \boldsymbol{\delta}_u^{(l+1)}(k)\right\rangle. \tag{51}$$

Dabei ist $\boldsymbol{\delta}_u^{(l+1)}(k)$, definiert durch

$$\boldsymbol{\delta}_u^{(l)}(k) \stackrel{def}{=} \left(\delta_u^{(l)}(k), \delta_u^{(l)}(k+1), \ldots, \delta_u^{(l)}(k+d^{(l)}-1)\right),$$

der sogenannte *Fehlervektor*.

Anhand dieser Formel wird auch der Namensbestandteil „Backpropagation“ dieses Algorithmus deutlich. Nachdem in einem ersten Schritt zur Berechnung des Fehlers an den Ausgabeneuronen externe Eingaben und Aktivierungen in Vorwärtsrichtung durch das TDNN propagiert werden, werden nun zur Bestimmung der notwendigen Gewichtsänderungen Fehlersignale rückwärts durch das Netz propagiert. Interessant ist, daß auch jeder Faktor $\left\langle \mathbf{w}_{j,u}^{(l+1)} \middle| \boldsymbol{\delta}_u^{(l+1)}(k)\right\rangle$ der Berechnung einer Filterausgabe entspricht (vergleiche Abbildung 10 mit Abbildung 1). Der Namensbestandteil

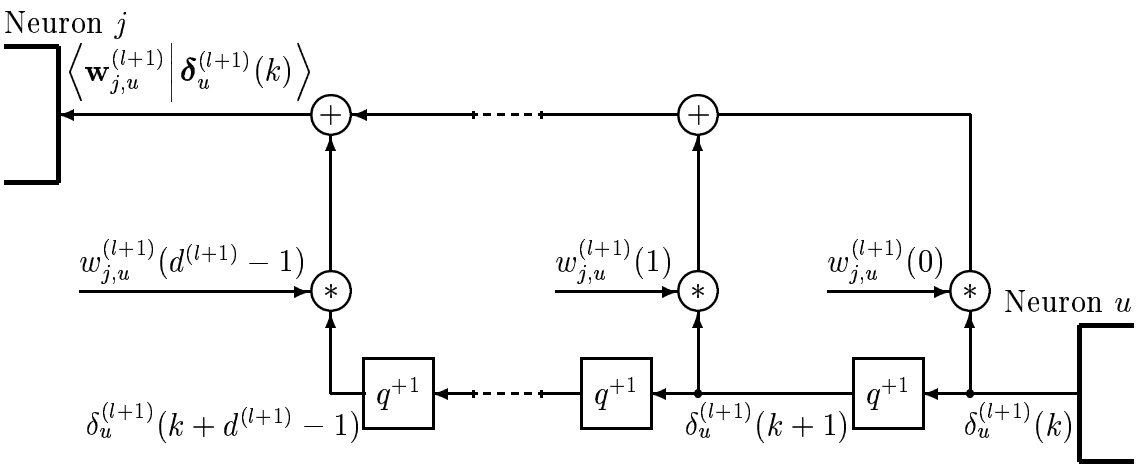


Abbildung 10: Synapse mit FIR-Filter im TDNN bei der Rückwärtspropagierung des Fehlersignals

„Temporal“ ist darauf zurückzuführen, daß bei der Berechnung des Fehlersignals nicht nur Einflüsse auf den aktuellen Fehler zum Zeitpunkt k zu berücksichtigen sind, wie es beim Standard-Backpropagation für MLP der Fall ist.

Zusammenfassend kann nun der gesamte Algorithmus Temporal Backpropagation formuliert werden.

Algorithmus 3.7 (*Temporal Backpropagation*)
Der Lernalgorithmus *Temporal Backpropagation* wird für epochenweises Lernen (Algorithmus 3.5) verwendet. Die Gewichtsänderungen für das k-te Muster der Lernaufgabe werden bestimmt durch

$$\nabla_k \mathbf{w}_{i,j}^{(l)} \stackrel{def}{=} \delta_j^{(l)}(k) \cdot \mathbf{a}_i^{(l-1)}(k).$$

Dabei ist das *Fehlersignal* $\delta_j^{(l)}(k)$ wie folgt gegeben:

$$\delta_j^{(l)}(k) \stackrel{def}{=} \begin{cases} \sigma'\left(s_j^{(l)}(k)\right) \cdot \left(t_j(k) - a_j^{(l)}(k)\right) & \text{für } l = L \\ \sigma'\left(s_j^{(l)}(k)\right) \cdot \sum\limits_{u \in \mathcal{U}_{l+1}} \left\langle \mathbf{w}_{j,u}^{(l+1)} \middle| \boldsymbol{\delta}_u^{(l+1)}(k) \right\rangle & \text{für } l \in \{2, \ldots, L-1\} \end{cases},$$

wobei $\boldsymbol{\delta}_u^{(l)}(k) \stackrel{def}{=} \left(\delta_u^{(l)}(k), \delta_u^{(l)}(k+1), \ldots, \delta_u^{(l)}(k+d^{(l)}-1)\right)$ der *Fehlervektor* ist.

Die Gewichtsänderung für die Epoche p ist dann für jede Verbindung

$$\nabla^p \mathbf{w}_{i,j}^{(l)} \stackrel{def}{=} \mu \cdot \sum_{k=1}^{K} \nabla_k \mathbf{w}_{i,j}^{(l)};$$

dabei ist $\mu \in \mathbb{R}^+$ die *Lernrate*. Der Wert der Lernrate kontrolliert die Schrittweite bei der Verfolgung des Gradienten.

Eine graphische Herleitung von verschiedenen Lernalgorithmen (u.a. auch Temporal Backpropagation für FIR-Netze) findet sich in [Wan93a, WB96, WB98].

Der angegebene Algorithmus entspricht für $d^{(l)} = 1$ ($l \in \{2, \ldots, L\}$) dem bekannten Backpropagation für MLP (und MLP-sw). Dazu sind nur die Vektoren $\mathbf{a}$, $\mathbf{w}$ und $\boldsymbol{\delta}$ durch Skalare zu ersetzen.

In der angegebenen Form ist der Algorithmus Temporal Backpropagation *nicht-kausal*, da zukünftige Fehlersignale in die Berechnung eines aktuellen Fehlersignals einfließen. Der Algorithmus läßt sich durch Modifikation von Indizes leicht in einer kausalen (aber etwas schwerer lesbaren Form) schreiben (siehe beispielsweise [Wan93a]). Bei einer Implementierung ist eine zusätzliche Pufferung von Aktivierungen entsprechend der Länge des Pfades mit der größten Verzögerung zu einem Ausgangsneuron vorzunehmen.

Von großer Bedeutung für das Trainingsergebnis ist die Initialisierung der internen Speicherelemente eines TDNN mit Anfangszuständen. Das TDNN sollte mit Werten initialisiert werden, die auch durch eine zulässige Sequenz von Eingabemustern entstehen könnten [CGHC97]. Gleichzeitig möchte man natürlich die ersten Trainingsmuster einer Sequenz nicht für die Initialisierung, sondern bereits für das Training verwenden. Beispielsweise kann man entsprechend der Länge r des rezeptiven Fensters das erste Muster einer Eingabesequenz r Mal wiederholen. Bei der letzten dieser Wiederholungen werden dann die ersten Gewichtsänderungen $\nabla_1 \mathbf{w}_{i,j}^{(l)}$ berechnet. Besteht eine Lernaufgabe aus mehreren Mustersequenzen, so muß diese Initialisierung natürlich jedesmal neu durchgeführt werden.

3.3 Weitere Lernalgorithmen

MLP ist allgemein das wohl am häufigsten eingesetzte Netzparadigma und Backpropagation der am häufigsten verwendete Lernalgorithmus. Gleichwohl weist Backpropagation eine Reihe von Nachteilen auf (siehe z.B. [Rie93, Rie94a]):

(1) Die Wahl der optimalen Lernrate ist schwierig (siehe Abbildung 11). Die optimale Lernrate hängt von der Form der Fehlerfunktion ab, die sich im Verlauf des Trainings verändert. Eine kleine Lernrate kann zu sehr langen Trainingszeiten führen, bei einer großen Lernrate kann manchmal ein Oszillieren beobachtet werden.

(2) Die Abhängigkeit der Gewichtsänderungen von Werten der partiellen Ableitungen ist nicht unbedingt intuitiv verständlich: In langen, relativ flachen Plateaus der Fehlerfunktion sind die Schritte sehr klein; ein enges, steiles Tal kann andererseits leicht übersprungen werden (siehe Abbildungen 12 und 13). Daher ist die Größe der partiellen Ableitung ein eher schlechtes Maß für die

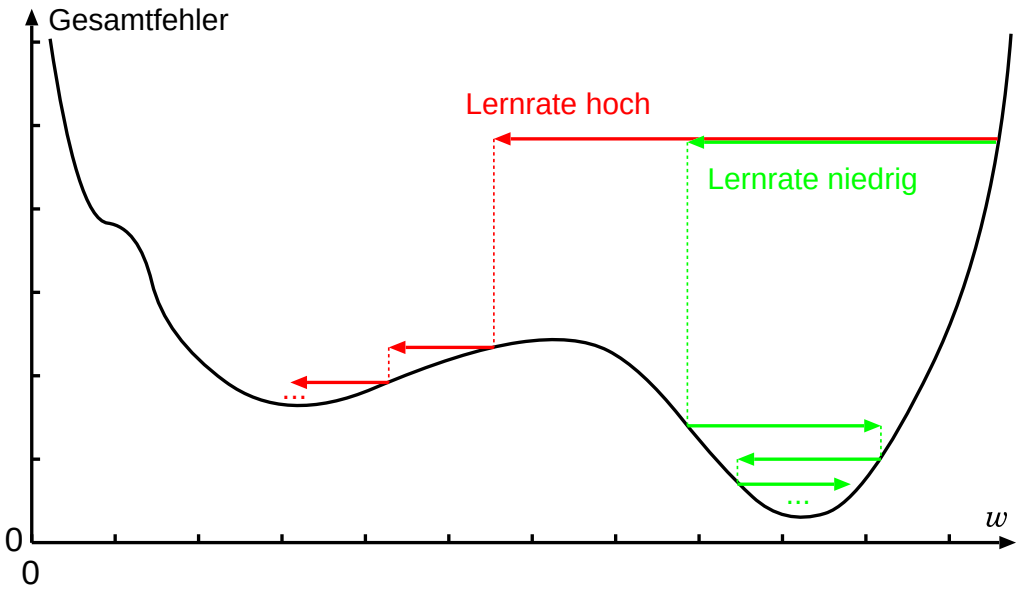


Abbildung 11: Konvergenz in verschiedenen Minima bei unterschiedlicher Wahl der Lernrate

Gewichtsänderung, da sie nur unzureichende Information über die Topologie der zu minimierenden Funktion enthält.

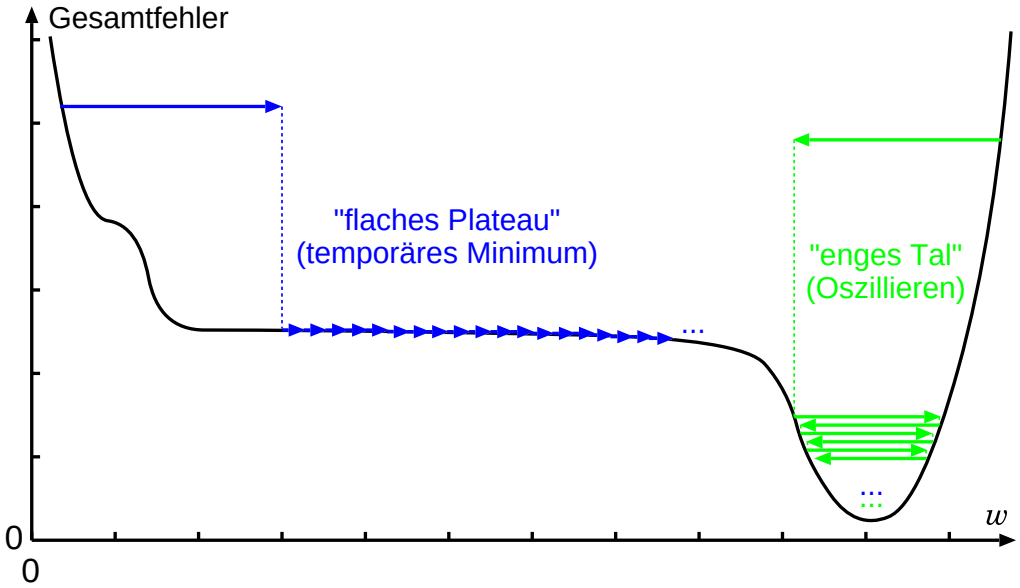


Abbildung 12: Lernfortschritt in „flachen Plateaus“ und „steilen Tälern“

(3) Gewichte, die weit von der Ausgabeschicht entfernt sind, werden deutlich langsamer adaptiert als Gewichte, die näher an der Ausgabeschicht sind (Grund ist der begrenzte Einfluß der Steigung der Aktivierungsfunktion).

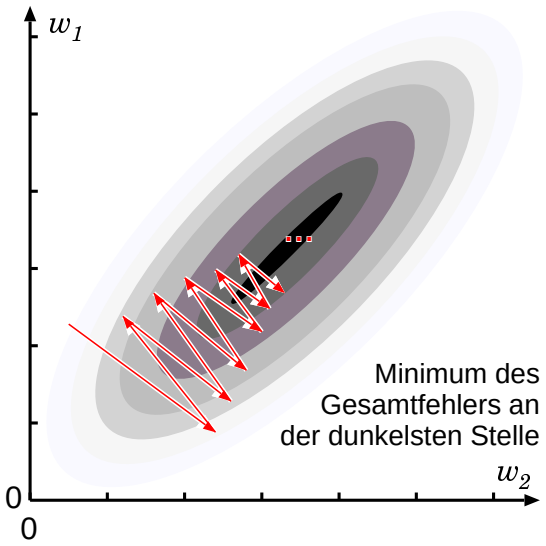


Abbildung 13: Oszillieren in „engen Tälern“

(4) Nicht immer werden Lernaufgaben erfüllt, manchmal konvergiert Backpropagation nicht oder auch nur für bestimmte Startpositionen (Gewichtsinitialisierungen). Bei unterschiedlichen Startpositionen kann das Verfahren auch in verschiedenen Minima konvergieren (siehe Abbildung 14).

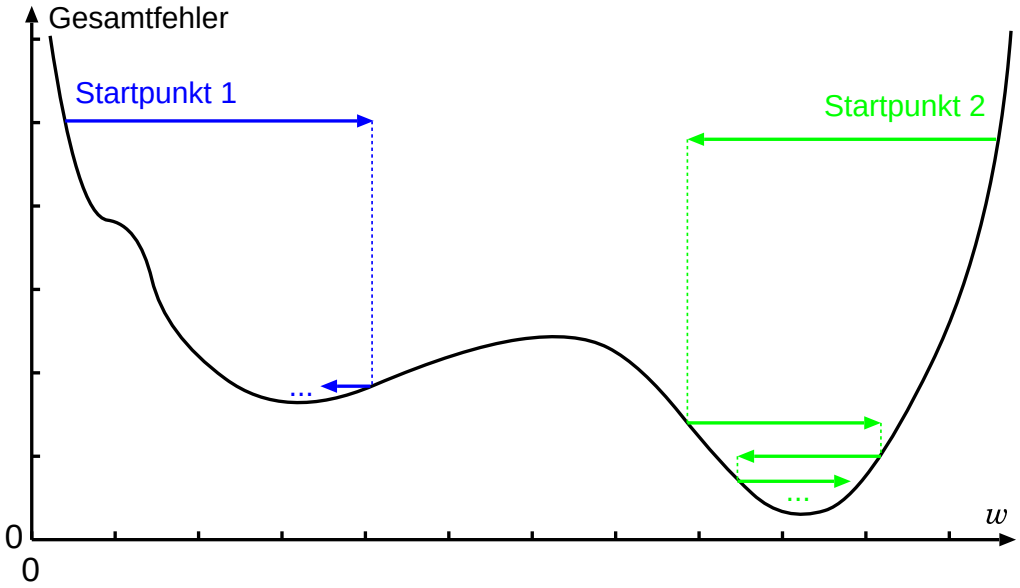


Abbildung 14: Konvergenz in verschiedenen Minima bei unterschiedlicher Wahl der Startpunkte (Gewichtsinitialisierung)

Diese Aussagen gelten auch für TDNN mit Temporal Backpropagation: Beispielsweise wird in [Wan93a] eine starke Abhängigkeit der Konvergenzgeschwindigkeit und des Trainingsergebnisses von der Lernrate festgestellt.

Zur Lösung dieser Probleme gibt es eine Reihe von Lösungsansätzen. Zwei davon, ursprünglich als Weiterentwicklungen das Lernalgorithmus Backpropagation für MLP vorgestellt, werden im folgenden näher beschrieben. Da sie hier zur Erweiterung des Lernalgorithmus Temporal Backpropagation für TDNN eingesetzt werden, sind sie mit dem Zusatz „Temporal“ versehen: Temporal Quickprop und Temporal Resilient Propagation.

3.3.1 Temporal Quickprop

Temporal Quickpropagation (oder kurz Quickprop) wurde mit dem Ziel einer deutlich verkürzten Trainingsdauer entwickelt. Das Verfahren basiert auf der Vorstellung, daß sich die Fehlerfunktion in der Nähe eines Minimums (d.h. lokal) durch eine nach oben geöffnete Parabel beschreiben läßt. Mit Hilfe des Wertes der Fehlerfunktion bzw. der Ableitung der Fehlerfunktion in der aktuellen bzw. der vorausgehenden Epoche wird der Scheitelpunkt der Parabel, d.h. die erwartete Position des Minimums der Fehlerfunktion bestimmt, um dann in einem Schritt dorthin zu springen.

Die gemäß dem Algorithmus Temporal Backpropagation (siehe Algorithmus 3.7) in einer Epoche p akkumulierte Summe der Gewichtsänderungen für ein bestimmtes Gewicht wird nun folgendermaßen abgekürzt:

$$S^{(p)}(\nabla_k w_{i,j}^{(l)}(m)) \stackrel{def}{=} \sum_{k=1}^{K} \nabla_k w_{i,j}^{(l)}(m). \tag{52}$$

Der Wert dieser Größe entspricht dem negativen Wert der partiellen Ableitung des Gesamtfehlers $\mathcal{E}$ nach dem Gewicht $w_{i,j}^{(l)}(m)$ in der Epoche p (siehe Gleichungen 28 und 30). Daher kann die Gewichtsänderung für eine Epoche $\nabla^p w_{i,j}^{(l)}(m)$ wie folgt bestimmt werden (siehe Abbildung 15):

$$\frac{\nabla^p w_{i,j}^{(l)}(m)}{\nabla^{p-1} w_{i,j}^{(l)}(m)} = \frac{S^{(p)}(\nabla_k w_{i,j}^{(l)}(m)) - S^{(p+1)}(\nabla_k w_{i,j}^{(l)}(m))}{S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) - S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}. \tag{53}$$

Mit der Annahme $S^{(p+1)}(\nabla_k w_{i,j}^{(l)}(m)) = 0$ (Scheitelpunkt der Parabel) gilt also

$$\nabla^p w_{i,j}^{(l)}(m) = \frac{S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}{S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) - S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))} \cdot \nabla^{p-1} w_{i,j}^{(l)}(m). \tag{54}$$

Bei der Anwendung dieser Berechnungsvorschrift für ein Gewicht $w_{i,j}^{(l)}(m)$ können die folgenden vier Situationen entstehen:

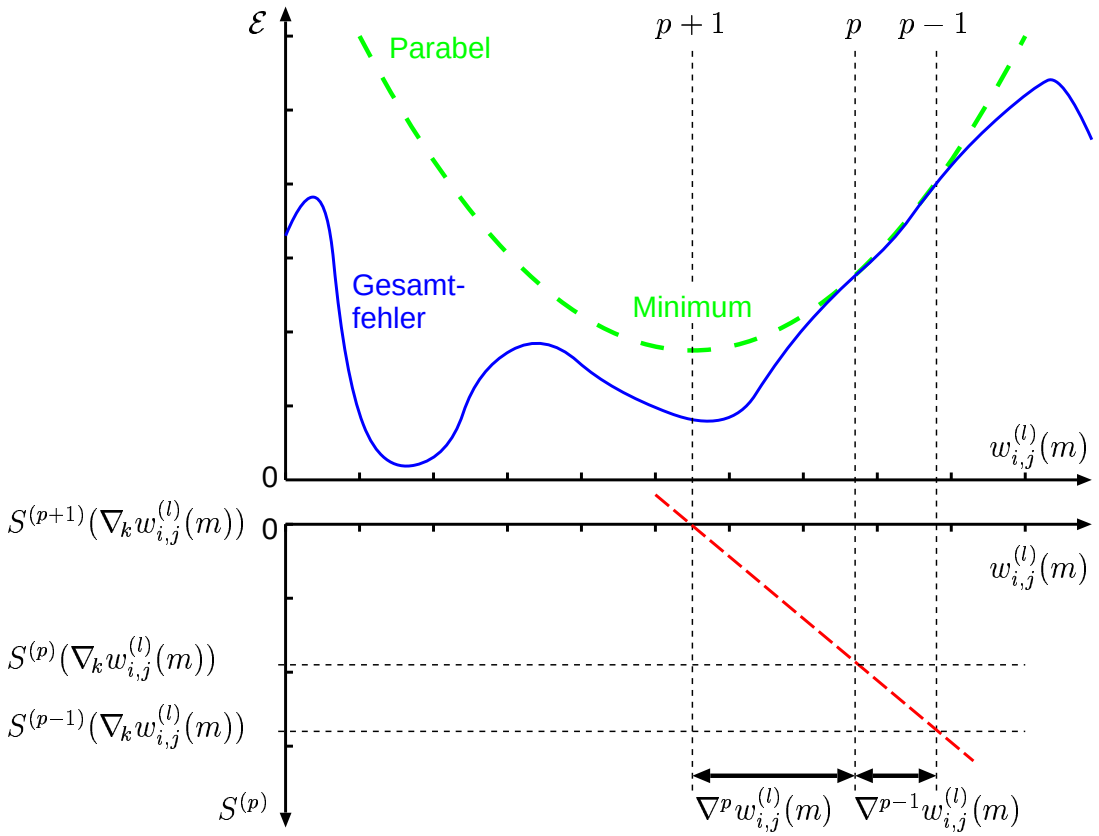


Abbildung 15: Bestimmung des Scheitelpunkts der die Fehlerfunktion approximierenden Parabel im Lernalgorithmus Quickprop

(1) Der Wert von $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ ist betragsmäßig kleiner als der Wert von $S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m))$, hat aber das gleiche Vorzeichen: Dann erfolgt der nächste Schritt in die gleiche Richtung wie der vorhergehende.

(2) Die Werte von $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ und $S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m))$ haben verschiedene Vorzeichen: Dann wurde ein Minimum übersprungen und der nächste Schritt ergibt eine Position zwischen den beiden vorausgehenden.

(3) $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ hat genau den gleichen Wert wie $S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m))$: Diese Situation ist problematisch, da das Verfahren wegen einer Division durch Null mit einem Fehler abbrechen würde. Auch der Fall $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m)) \approx S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m))$ ist nicht unkritisch, wenn das Verfahren einen sehr großen Schritt macht.

(4) Der Wert von $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ ist betragsmäßig größer als der Wert von $S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m))$ und hat auch das gleiche Vorzeichen: Diese Situation ist ebenfalls problematisch, weil das Verfahren dann nach dem Scheitelpunkt einer nach unten offenen Parabel sucht.

Zur Lösung des in (3) beschriebenen Problems wird ein *maximaler Wachstumsfaktor* γ eingeführt, der die Zu- bzw. Abnahme der Gewichtsänderung in aufeinanderfolgenden Epochen begrenzt:

$$\left|\nabla^p w_{i,j}^{(l)}(m)\right| = \gamma \cdot \left|\nabla^{p-1} w_{i,j}^{(l)}(m)\right|. \tag{55}$$

Dabei wird üblicherweise $\gamma \in [1.75, 2.25]$ gewählt [Zel94].

Beim Start des Verfahrens oder wenn die Gewichtsänderung in der vorausgegenden Epoche gleich Null ist, kann eine Gewichtsänderung auf der Basis des aktuellen Wertes des Gradienten erfolgen.

Im folgenden wird die in [Zel94] beschriebene Form von Quickprop vorgestellt, bei der die Gewichtsänderung mit Hilfe eines *Gradiententerms* und eines *Parabelterms* bestimmt wird. Der Gradiententerm enthält einen zusätzlichen Term, der die Gewichte auf betragsmäßig kleinen Werten halten soll (*weight decay*).

Algorithmus 3.8 (*Temporal Quickprop*)
Der Algorithmus *Temporal Quickprop* arbeitet nach dem Prinzip des epochenweisen Lernens (Algorithmus 3.5) und basiert auf dem Lernalgorithmus Temporal Backpropagation (Algorithmus 3.7).

(1) Berechne für jedes Gewicht $w_{i,j}^{(l)}(m)$ eine Gewichtsänderung für die erste Epoche $\nabla^1 w_{i,j}^{(l)}(m) = \mu \cdot S^{(1)}(\nabla_k w_{i,j}^{(l)}(m))$, wobei $S^{(1)}(\nabla_k w_{i,j}^{(l)}(m))$ mit Hilfe des Algorithmus Temporal Backpropagation bestimmt wird, und ändere jedes Gewicht entsprechend.

(2) Wiederhole die folgenden Schritte (a) bis (e) für alle weiteren Epochen $p = 2, 3, \ldots$ bis eine maximale Epochenzahl erreicht oder eine zuvor gewählte Fehlerschranke unterschritten ist:

   (a) Berechne $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ für jedes Gewicht nach dem Algorithmus Temporal Backpropagation.

   (b) Initialisiere für jedes Gewicht einen *Gradiententerm* $G^{(p)}(w_{i,j}^{(l)}(m)) \stackrel{def}{=} 0$ und einen *Parabelterm* $P^{(p)}(w_{i,j}^{(l)}(m)) \stackrel{def}{=} 0$.

   (c) Wenn $\operatorname{sgn}(S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))) = \operatorname{sgn}(S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)))$ oder $\nabla^{p-1} w_{i,j}^{(l)}(m) = 0$, dann ändere den Wert des Gradiententerms nach folgender Regel:

$$G^{(p)}(w_{i,j}^{(l)}(m)) = \mu \cdot S^{(p)}(\nabla_k w_{i,j}^{(l)}(m)) - d \cdot w_{i,j}^{(l)}(m))$$

   (d) Wenn $\nabla^{p-1} w_{i,j}^{(l)}(m) \neq 0$ dann ändere den Wert des Parabelterms nach folgender Regel:

- **Fall A**: Wenn $\left|\frac{S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}{S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) - S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}\right| \leq \gamma$, dann setze

$$P^{(p)}(w_{i,j}^{(l)}(m)) \stackrel{def}{=} \frac{S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}{S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) - S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))} \cdot \nabla^{p-1} w_{i,j}^{(l)}(m).$$

- **Fall B**: Sonst, also wenn $\left|\frac{S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}{S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) - S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))}\right| > \gamma$, setze

$$P^{(p)}(w_{i,j}^{(l)}(m)) \stackrel{def}{=} \gamma \cdot \nabla^{p-1} w_{i,j}^{(l)}(m).$$

(e) Ändere jedes Gewicht um $\nabla^p w_{i,j}^{(l)}(m) = G^{(p)}(w_{i,j}^{(l)}(m)) + P^{(p)}(w_{i,j}^{(l)}(m))$.

Dabei sei $\mu \in]0, 2[$ die *Lernrate*, $d \approx 0.0001$ der *Weight-Decay-Faktor* und $\gamma \in [1.75, 2.25]$ der *maximale Wachstumsfaktor*.

Quickprop basiert auf zwei Annahmen, die allerdings nicht in jeder Anwendung erfüllt sind:

(1) Die Fehlerfunktion kann lokal durch eine nach oben geöffnete Parabel approximiert werden.

(2) Eine Gewichtsänderung kann unabhängig von den Änderungen anderer Gewichte erfolgen.

3.3.2 Temporal Resilient Backpropagation

Aufgrund der oben genannten Probleme des Lernalgorithmus Backpropagation fordert [Rie93] eine gewichtsspezifischen Schrittweite (*strukturelle Adaption*), die sich dynamisch an die lokal „sichtbare“ Form der Fehlerfunktion anpaßt (*temporäre Adaption*) und nicht (oder nicht nur) von der Höhe der partiellen Ableitung abhängt. Der Lernalgorithmus *RPROP* (*Resilient Backpropagation* oder kürzer *Resilient Propagation*) erfüllt diese Forderungen [Rie93, RB93, Rie94a, Rie94b]. Er berücksichtigt nur das Vorzeichen der partiellen Ableitung, um die Richtung der Gewichtsänderung vorzugeben. Die Höhe der Gewichtsänderung steigt oder fällt in Abhängigkeit von nicht vorhandenen oder vorhandenen Vorzeichenwechseln der partiellen Ableitung. RPROP basiert auf Backpropagation; entsprechend baut der hier vorgestellte Algorithmus auf Temporal Backpropagation auf und wird daher als *Temporal RPROP* bezeichnet.

Wie beim Lernalgorithmus Temporal Quickprop sei die gemäß Temporal Backpropagation (siehe Algorithmus 3.7) in einer Epoche p akkumulierte Summe der Gewichtsänderungen $\sum_{k=1}^{K} \nabla_k w_{i,j}^l(m)$ im folgenden mit $S^p(\nabla_k w_{i,j}^l(m))$ bezeichnet. Daneben benötigt man die Funktion sign, die für $x \in \mathbb{R}$ definiert ist durch

$$\text{sign}(x) \stackrel{def}{=} \begin{cases} +1 & \text{für } x > 0 \\ 0 & \text{für } x = 0 \\ -1 & \text{sonst} \end{cases}.$$

Algorithmus 3.9 (*Temporal Resilient Backpropagation*)
Der Algorithmus *Temporal Resilient Backpropagation* (*Temporal RPROP*) arbeitet nach dem Prinzip des epochenweisen Lernens (Algorithmus 3.5) und basiert auf dem Lernalgorithmus Backpropagation (Algorithmus 3.7).

(1) Initialisiere für jedes Gewicht $w_{i,j}^{(l)}(m)$ einen individuellen *Updatewert* $\triangle^p w_{i,j}^{(l)}(m)$ mit $\triangle^0 w_{i,j}^{(l)}(m) = \triangle_{\text{init}}$.

(2) Initialisiere für jedes Gewicht $w_{i,j}^{(l)}(m)$ die akkumulierte Summe der Gewichtsänderungen $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ mit $S^{(0)}(\nabla_k w_{i,j}^{(l)}(m)) = 0$.

(3) Wiederhole die folgenden Schritte (a) bis (c) für alle Epochen $p = 1, 2, \ldots$ bis eine maximale Epochenzahl erreicht oder eine zuvor gewählte Fehlerschranke unterschritten ist:

(a) Berechne $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ für jedes Gewicht nach dem Algorithmus Temporal Backpropagation.

(b) Bestimme die Gewichtsänderungen für die p-te Epoche $\nabla^p w_{i,j}^{(l)}(m)$ für jedes Gewicht nach folgender Regel:

- **Fall A**: Wenn $\left(S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) \cdot S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))\right) > 0$, dann setze

$$\begin{aligned} \triangle^p w_{i,j}^{(l)}(m) &\stackrel{def}{=} \min\{\triangle^{p-1} w_{i,j}^{(l)}(m) \cdot \eta^+, \triangle_{\max}\} \text{ und} \\ \nabla^p w_{i,j}^{(l)}(m) &\stackrel{def}{=} \text{sign}\left(S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))\right) \cdot \triangle^p w_{i,j}^{(l)}(m). \end{aligned}$$

- **Fall B**: Wenn $\left(S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) \cdot S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))\right) < 0$, dann setze

$$\begin{aligned} \triangle^p w_{i,j}^{(l)}(m) &\stackrel{def}{=} \max\{\triangle^{p-1} w_{i,j}^{(l)}(m) \cdot \eta^-, \triangle_{\min}\}, \\ S^{(p)}(\nabla_k w_{i,j}^{(l)}(m)) &\stackrel{def}{=} 0 \text{ und} \\ \nabla^p w_{i,j}^{(l)}(m) &\stackrel{def}{=} 0. \end{aligned}$$

- **Fall C**: Sonst, also wenn $\left(S^{(p-1)}(\nabla_k w_{i,j}^{(l)}(m)) \cdot S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))\right) = 0$, setze

$$\begin{aligned} \triangle^p w_{i,j}^{(l)}(m) &\stackrel{def}{=} \triangle^{p-1} w_{i,j}^{(l)}(m) \text{ und} \\ \nabla^p w_{i,j}^{(l)}(m) &\stackrel{def}{=} \text{sign}\left(S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))\right) \cdot \triangle^p w_{i,j}^{(l)}(m). \end{aligned}$$

(c) Ändere jedes Gewicht um $\nabla^p w_{i,j}^{(l)}(m)$.

Dabei gelte für den *minimalen Updatewert* $\triangle_{\min}$, den *maximalen Updatewert* $\triangle_{\max}$ und den *initialen Updatewert* $\triangle_{\text{init}}$ mit $\triangle_{\min}, \triangle_{\max}, \triangle_{\text{init}} \in \mathbb{R}^+$ die Beziehung $\triangle_{\min} < \triangle_{\text{init}} < \triangle_{\max}$. Für den *Zunahmefaktor* η^+ und den *Abnahmefaktor* η^- mit $\eta^+, \eta^- \in \mathbb{R}^+$ gelte $\eta^- < 1 < \eta^+$ und $\frac{1}{\eta^-} > \eta^+$.

Schritt (3b) legt die Höhe und die Richtung einer Gewichtsänderung fest: Wechselt $S^{(p)}(\nabla_k w_{i,j}^{(l)}(m))$ das Vorzeichen nicht, so wird die Schrittweite (Gewichtsänderung) größer. Ändert sich das Vorzeichen (ein Minimum wurde übersprungen) so wird im übernächsten Schritt eine kleinere Gewichtsänderung in die entgegengesetzte Richtung durchgeführt.

In den Originalbeschreibungen von RPROP wird für den Zunahmefaktor η^+ und den Abnahmefaktor η^- anstelle der stärkeren Bedingungen $\frac{1}{\eta^-} > \eta^+$ und $\eta^- < 1 < \eta^+$ lediglich die Erfüllung von $\eta^- < 1 < \eta^+$ verlangt. Allerdings ist es sinnvoll, daß eine Zunahme der Schrittweite bei der Gradientenverfolgung langsamer erfolgt als eine Abnahme [Roj96]. Zudem kann es für $\frac{1}{\eta^-} = \eta^+$ in manchen symmetrischen Umgebungen um ein Minimum zu einem Oszillieren kommen (siehe schematische Darstellung in Abbildung 16). Daher wird hier zusätzlich $\frac{1}{\eta^-} > \eta^+$ gefordert, um eine rasche Konvergenz zu unterstützen.

Als Vorteile von RPROP wurden im Vergleich mit anderen Lernalgorithmen (Backpropagation, Backpropagation mit Momentumterm, Quickprop, SuperSAB, Backpercolation) festgestellt (siehe z.B. [Rie93, RB93, Rie94a, Rie94b, SB97]):

(1) Konvergenz in Aufgabenstellungen, in denen Backpropagation nicht oder selten (je nach Gewichtsinitialisierung, d.h. Startpunkt) und auch SuperSAB und Quickprop seltener konvergieren,

(2) schnellere Konvergenz,

(3) geringere Abhängigkeit der Konvergenzgeschwindigkeit und des Trainingsergebnisses von Parametern des Lernalgorithmus und

(4) speziell bei TDNN eine bessere Generalisierungsfähigkeit als Backpropagation (trotz optimaler Einstellung der Lernrate) bei erzwungenen kleinen Gewichtsänderungen (d.h. $\triangle_{\max} = 0.01$ statt üblicherweise $\triangle_{\max} = 50.0$).

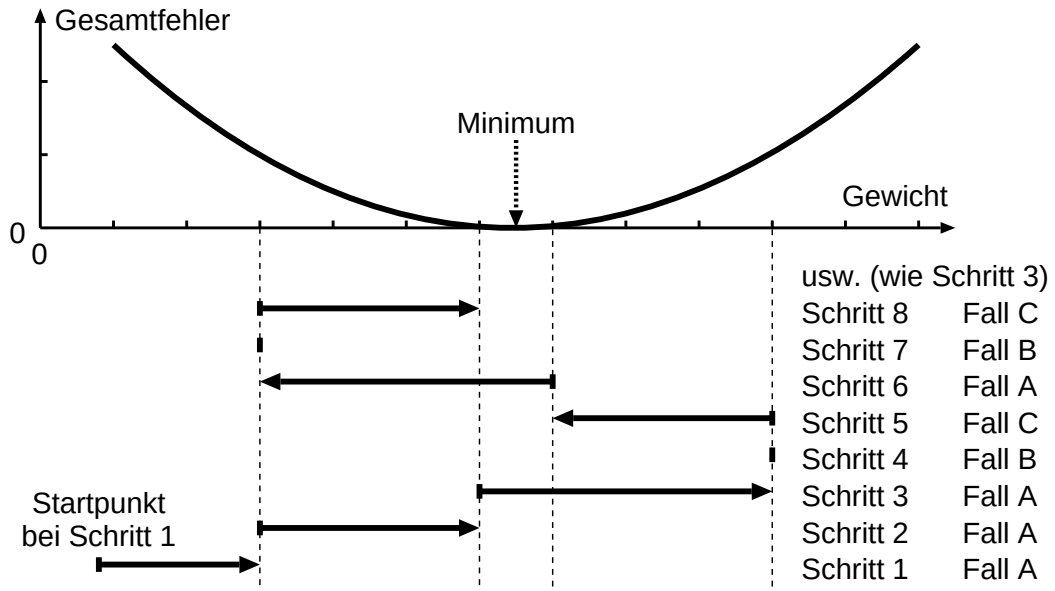


Abbildung 16: Beispiel für die Oszillation bei RPROP mit bestimmten Parameterkonstellationen

Punkt (1) ist von allgemeiner Bedeutung; in vielen praktischen Aufgabenstellungen kann jedoch auch bei der Verwendung von Backpropagation immer eine Konvergenz beobachtet werden.

Die in Punkt (2) angesprochene schnellere Konvergenz ist beispielsweise wichtig, wenn ein Neuronales Netz online (d.h. während seiner Anwendung) für eine spezifische Aufgabenstellung trainiert werden muß. Auch wenn (wie z.B. bei der Strukturoptimierung des Netzes oder der Selektion der geeignetsten Merkmale) sehr viele Trainingsdurchgänge durchgeführt werden müssen, ist diese Eigenschaft erfreulich, jedoch nicht zwingend notwendig.

Punkt (3) ist als sehr wichtig einzuschätzen, da es das Ziel vieler Anwendungen ist, Verfahren zu entwickeln, die robust in dem Sinne sind, daß die Wahl bestimmter Parameter (darunter auch die Lernparameter) einen untergeordneten Einfluß auf das Trainingsergebnis hat. Der Initialupdatewert $\triangle_{\text{init}}$ hat bei Aufgabenstellungen mit hoher Epochenzahl kaum Einfluß auf das Ergebnis.

Besonders interessant für den Einsatz von TDNN ist natürlich Punkt (4). Die in [Rie93] festgestellte verbesserte Generalisierungsfähigkeit bei TDNN (dort wird ein Benchmark-Problem aus dem Bereich der Spracherkennung untersucht) konnte z.B. für die Werkzeugzustandsüberwachung in Drehmaschinen bestätigt werden [Sic00]. Eine Erklärung für dieses Verhalten von RPROP wird auch in [Rie93] (Autor ist der Entwickler dieses Lernalgorithmus) nicht gegeben. Ein möglicher Grund könnte jedoch sein, daß bei RPROP die Gewichte nahe an der Eingabeschicht (z.B. zwischen Eingabe- und erster verdeckter Schicht) schneller adaptiert werden als bei

Backpropagation, da die Höhe der Gewichtsänderung nur vom Vorzeichen des Gradienten abhängt. Gerade diese Gewichte beschreiben den mehr oder weniger großen Einfluß zeitlich aufeinanderfolgender Eingangsmuster, sie sind also für die Erfassung zeitlicher Entwicklungen besonders wichtig.

3.3.3 Anmerkungen zu weiteren Lernalgorithmen für TDNN

Bisher wurden in Abschnitt 3 drei Lernalgorithmen für TDNN ausführlich beschrieben: Temporal Backpropagation, Temporal Quickprop und Temporal RPROP (siehe Abbildung 17). Daneben gibt es eine Vielzahl weiterer Algorithmen, von denen einige in diesem Abschnitt kurz angesprochen werden sollen.

Prinzipiell läßt sich für fast jeden Trainingsalgorithmus für MLP ein ähnlicher Algorithmus für TDNN konstruieren, wobei zusätzlich jedoch die verzögerte Weitergabe der Aktivierungen zu berücksichtigen ist. Als Kriterium für die Bewertung eines Lernalgorithmus können verwendet werden:

- die Generalisierungsfähigkeit der mit diesem Algorithmus trainierten Netze,
- die Abhängigkeit des Trainingsergebnisses von konkreten Werten von Lernparametern,
- die Konvergenzgeschwindigkeit,
- die numerische Stabilität,
- der Berechnungsaufwand für die Gewichtsänderungen,
- der Speicherverbrauch und
- der Implementierungsaufwand.

Ein Verfahren zum musterweisen Trainieren der Gewichte wird in [LDL92a, LDL95, LLD93a] vorgestellt. Dort wird Gleichung 45 fortgesetzt durch

$$\delta_j^{(l)}(k) \approx \sum_{u \in \mathcal{U}_{l+1}} \sum_{m=0}^{d^{(l+1)}-1} \delta_u^{(l+1)}(k) \cdot \frac{\partial s_u^{(l+1)}(k+m)}{\partial s_j^{(l)}(k)} \tag{56}$$

$$= \ldots$$

$$= \sum_{u \in \mathcal{U}_{l+1}} \sum_{m=0}^{d^{(l+1)}-1} \delta_u^{(l+1)}(k) \cdot w_{j,u}^{(l+1)}(m) \cdot \sigma'\left(s_j^{(l)}(k)\right), \tag{57}$$

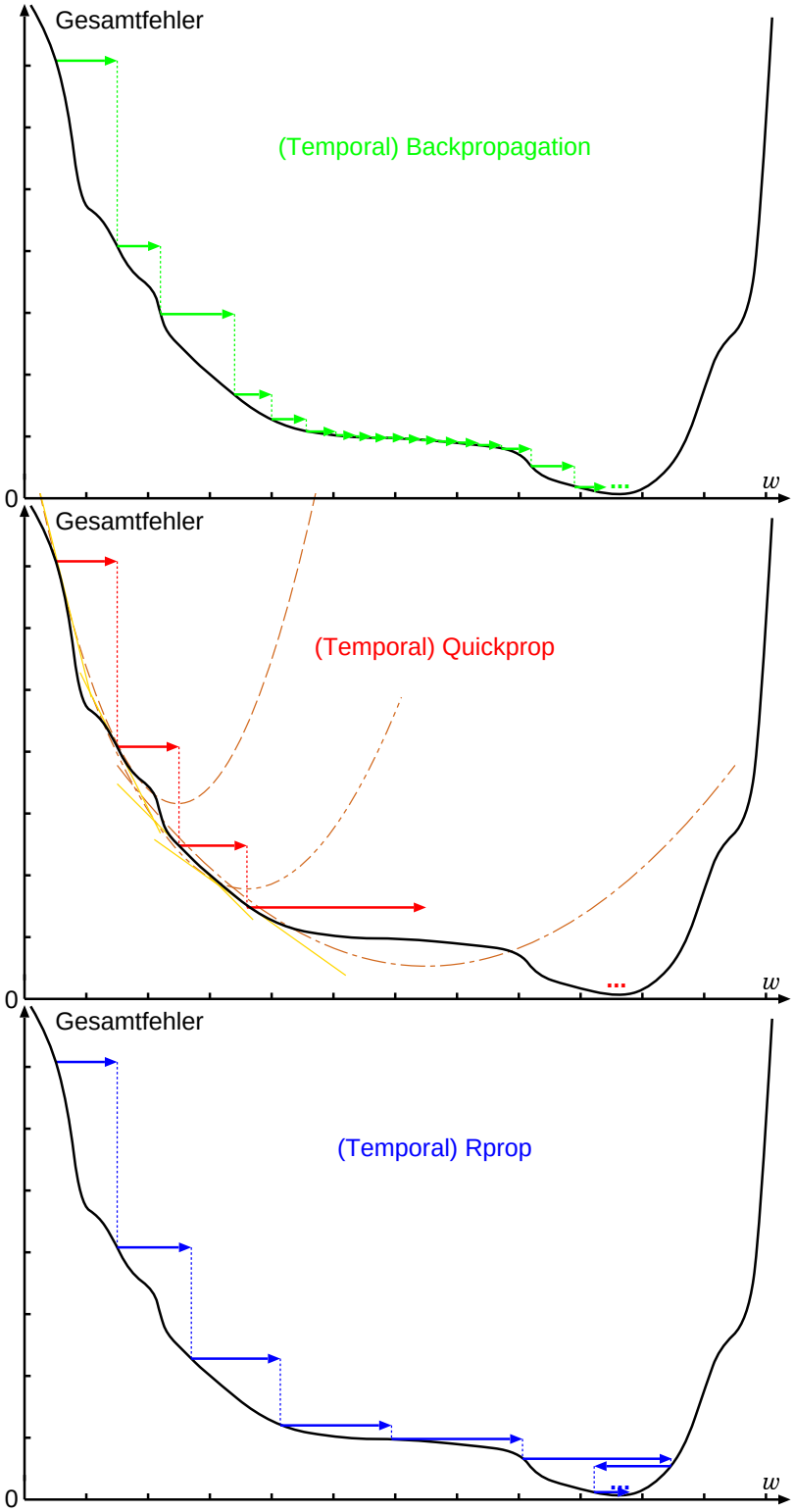


Abbildung 17: Beispiel für die Arbeitsweise von Backpropagation, Quickprop und RPROP bei gleichem Startpunkt und gleicher Startschrittweite

d.h. mit $\delta_u^{(l+1)}(k)$ anstelle von $\delta_u^{(l+1)}(k+m)$. Dies führt im Algorithmus 3.7 zu einer etwas anderen Berechnungsvorschrift für das Fehlersignal $\delta_j^{(l)}(k)$, nämlich zu

$$\delta_j^{(l)}(k) \stackrel{def}{=} \begin{cases} \sigma'\left(s_j^{(l)}(k)\right) \cdot \left(t_j(k) - a_j^{(l)}(k)\right) & \text{für } l = L \\ \sigma'\left(s_j^{(l)}(k)\right) \cdot \sum\limits_{u \in \mathcal{U}_{l+1}} \delta_u^{(l+1)}(k) \cdot \mathbf{w}_{j,u}^{(l+1)} & \text{für } l \in \{2, \ldots, L-1\} \end{cases} \tag{58}$$

mit dem Skalar $\delta_u^{(l+1)}(k)$ anstelle des Vektors $\boldsymbol{\delta}_u^{(l+1)}(k)$. Wie man leicht sieht, werden hier Auswirkungen der aktuellen Aktivierungszustände auf den Fehler zu verschiedenen Zeitpunkten nicht ausreichend berücksichtigt. Natürlich ist die Implementierung einfacher, da eine Speicherung von Fehlersignalen entfällt. Das Verfahren entspricht der Anwendung von Standard-Backpropagation auf TDNN und ist aus folgendem Grund als problematisch einzuschätzen: Bei großen Lernraten kann das musterweise Adaptieren der Gewichte zu Instabilität führen; nur bei sehr kleinen Lernraten ändern sich die Gewichte in einem Zeitraum, der der Länge des rezeptiven Fensters des TDNN entspricht, so wenig, daß der tatsächliche Gradient ausreichend angenähert wird. Somit hängt es stark von der Aufgabenstellung ab, ob mit Standard-Backpropagation ähnlich gute Ergebnisse wie mit Temporal Backpropagation für TDNN erzielt werden können. Eine Variante von Backpropagation zum musterweisen Trainieren von TDNN, bei der die Fehlerauswirkungen berücksichtigt werden, stellt beispielsweise [DD93] vor.

Das Training des zeitlich entfalteten Netzes mit Standard-Backpropagation wurde bereits in Abschnitt 3.2 erwähnt. Ein experimenteller Vergleich dieses Verfahrens mit Temporal Backpropagation wird in [Wan93a] durchgeführt. Als größter Nachteil stellt sich die hohe Neuronen- und Gewichtszahl heraus. Daher wird in [CG96] ein Verfahren zum Training eines *teilweise zeitlich entfalteten Netzes* beschrieben. Auch dieses Netz besitzt noch gemeinsam verwendete Gewichte (siehe Abschnitt 2.2), wodurch das Lokalitätsprinzip verletzt ist.

Eine Reihe bekannter Ideen zur Verbesserung von Backpropagation für MLP, wie beispielsweise Backpropagation mit Momentum, Verfahren nach Silva und Almeida, Delta-Bar-Delta, SuperSAB und viele mehr (siehe z.B. [Roj96, Zel94]) können in ähnlicher Form zur Erweiterung von Temporal Backpropagation für TDNN herangezogen werden (auch Quickprop und RPROP sind dieser Gruppe von Weiterentwicklungen von Backpropagation zuzuordnen). Der Lernalgorithmus QRPROP (siehe [Roj96]) kombiniert die Ideen von Quickprop und RPROP: Ist das Vorzeichen des Gradienten (bzw. das Vorzeichen der akkumulierten Gewichtsänderungen) in zwei aufeinanderfolgenden Epochen gleich, so wird die Strategie von RPROP verwendet. Im anderen Fall, wenn also ein (lokales) Minimum übersprungen wurde, so erfolgt ein Schritt gemäß der Strategie von Quickprop.

Daneben gibt es eine Reihe von „klassischen“ Verfahren der nichtlinearen Optimierung (speziell handelt es sich hier um eine multivariate Minimierung ohne Ne-

benbedingungen), die auch zum Training von MLP eingesetzt werden. Zur Bestimmung der Laufrichtung in der Fehlerfläche werden beispielsweise (siehe u.a. [Bis95, Rie94a, Roj96, She97])

- Quasi-Newton-Verfahren (BFGS: Broyden-Fletcher-Goldfarb-Shanno, DFP: Davidon-Fletcher-Powell) oder
- Konjugierte-Gradienten-Verfahren (Hestenes-Stiefel, Polak-Ribiere, Fletcher-Reeves)

verwendet. Zusätzlich wird ein geeignetes Verfahren zur Minimierung in Laufrichtung benötigt (z.B. Verfahren nach Dennis-Schnabel oder nach Brent). Diese Verfahren basieren auf der Idee, daß zur Näherung einer beliebigen Funktion in der Nähe eines Minimums eine quadratische Funktion verwendet werden kann (sog. *Second-Order-Verfahren*). Einige Second-Order-Verfahren benötigen eine Hesse-Matrix der zweiten partiellen Ableitungen der Fehlerfunktion oder eine Näherung dieser Matrix. Eine Methode zur exakten Berechnung der Hesse-Matrix für TDNN unter Berücksichtigung der verzögerten Aktivierungen wurde erst kürzlich in [CZ98] vorgestellt.

Als ein signifikantes Resultat umfassender Benchmarktests mit MLP in [She97] stellt sich jedoch heraus, daß „lokale Minima ein noch viel größeres Hindernis für bestimmte Second-Order-Verfahren – speziell Quasi-Newton-Verfahren [...] und (in einem etwas geringeren Ausmaß) Konjugierte-Gradienten-Verfahren [...] – sind, als für Methoden, die dem konventionellen Backpropagation verwandt sind.“ Dieses Ergebnis kann durch eigene Untersuchungen bestätigt werden [May99].

Weitere Trainingsalgorithmen speziell für TDNN werden beispielsweise in [BT96, BWLT94, YB97] beschrieben.

Zuletzt sollte nicht unerwähnt bleiben, daß eine ganze Reihe weiterer Faktoren Einfluß auf die Konvergenz und die Generalisierungsleistung hat (siehe z.B. [Roj96]): das Fehlermaß, eine Vorverarbeitung der Trainingsdatenmenge (z.B. Dekorrelation der Eingangsdaten), die Vorgehensweise bei der Gewichtsinitialisierung usw.

4 Strukturoptimierung für Neuronale Netze

Neben der Adaption der Gewichte, einer kontinuierlichen Optimierungsaufgabe, sind bei einer Anwendung eines TDNN auch noch verschiedene diskrete Optimierungsaufgaben zu lösen:

- die Wahl einer geeigneten Zahl verdeckter Schichten,

- die Bestimmung einer optimalen Anzahl von Neuronen in den verdeckten Schichten,
- die Festlegung einer Verbindungsstruktur und
- die Suche nach den besten Werten für die Anzahl von Zeitschritten, um die die Aktivierungszustände in Speicherelementen zu verzögern sind.

Die genannten Fragestellungen betreffen die Netztopologie (Netzstruktur) und sie sind von essentieller Bedeutung für die Modellbildung (siehe beispielsweise Abschnitte 2.2 und 3.1). Die Strukturoptimierung hinsichtlich bestimmter, vorgegebener Kriterien (z.B. der Generalisierungsleistung oder der Konvergenzgeschwindigkeit in der Lernphase) ist nach [KD97] sogar „zur Zeit eines der zentralen Forschungsprobleme im Bereich der Neuronalen Netze“.

Warum ist die Strukturoptimierung ein nicht-triviales Problem? Antworten auf diese Frage liefern beispielsweise [BW93b, Man94, RT95, Yao93]:

- Ist der Lösungsraum nicht beschränkt, so kann es prinzipiell unendlich viele mögliche Strukturen geben.
- Es handelt sich um eine diskrete Optimierungsaufgabe, bei der die Zielfunktion (Bewertungsfunktion für die Performanz einer Struktur), die z.B. die Generalisierungsleistung des Netzes oder die Anzahl von Verbindungen berücksichtigt, nicht differenzierbar ist.
- Ähnliche Strukturen können sehr unterschiedliche Performanz zeigen, während sehr verschiedene Strukturen manchmal sehr ähnlich zu bewerten sind (in der englischsprachigen Literatur sind diese Eigenschaften als *deceptive* und *multimodal* bezeichnet).
- Die Bewertung einer Netzstruktur kann zu sehr ungenauen (verrauschten) Ergebnissen führen, z.B. da das Training der Netze mit einer zufälligen Gewichtsinitialisierung beginnt (beim musterweisen Lernen ist beispielsweise auch die Musterreihenfolge von Bedeutung, beim inkrementellen Lernen die Reihenfolge der Mustersequenzen).

Die für die Strukturoptimierung bekannten Verfahren (im allgemeinen angewandt für Mehrlagige Perzeptren) lassen sich in drei Klassen einteilen: destruktive, konstruktive und hybride Verfahren.

Destruktive Verfahren (*pruning*; siehe beispielsweise [Bis95, Ree93, Zel94]) gehen von sehr großen Netzstukturen aus und versuchen, sukzessive verschiedene Elemente (Neuronen, Verbindungen usw.) zu löschen oder Gewichte an Verbindungen sehr klein zu halten (was einem Löschen nahekommt). Zur ersten Gruppe

destruktiver Verfahren gehören beispielsweise Algorithmen, die die Verbindungen mit den betragsmäßig kleinsten Gewichten löschen (z.B. *magnitude based pruning*), Algorithmen, die die Verbindungen mit minimalem Einfluß auf den Ausgabefehler des Netzes löschen (z.B. *OBS: optimal brain surgeon, OBD: optimal brain damage*), Skelettierungs-Algorithmen (für Neuronen) usw. Bei der zweiten Gruppe von destruktiven Verfahren wird üblicherweise ein zusätzlicher Term zur Fehlerfunktion addiert (*penalty term*); zu dieser Gruppe zählen beispielsweise *weight decay* und verwandte Algorithmen. Bei einem Vergleich destruktiver Verfahren (*magnitude based pruning, OBS, OBD,* Skelettierung) stellt [Zel94] jedoch fest: „Enttäuschenderweise konnte keines der getesteten Verfahren durch das Ausdünnen [gemeint ist das Löschen von Verbindungen] die Generalisierungsleistung der Netze auf unbekannten Testdaten entscheidend verbessern. "

Konstruktive Verfahren (*growing*; siehe beispielsweise [Bis95, Moh94, MP96, SH95]) beginnen mit sehr kleinen Initialstrukturen und vergrößern diese allmählich durch Hinzunahme von Verbindungen und Neuronen. Zur Gruppe der konstruktiven Algorithmen gehören beispielsweise Cascade-Correlation, Cascade2, Tiling-Algorithmus, Upstart-Algorithmus, Dynamic Node Creation, FNNCA (Feedforward Neural Network Construction Algorithm) und FlexNet.

Sowohl rein destruktive als auch rein konstruktive Verfahren realisieren im allgemeinen eine weitgehend lokale Suche nach einer optimalen Netzstruktur im Raum der prinzipiell vorhandenen Möglichkeiten von Netzstrukturen (Suchraum). Die meisten hybriden Verfahren, die destruktive und konstruktive Ansätze kombinieren, versuchen, diesen Nachteil zu vermeiden. Zu dieser Gruppe von Verfahren zählen beispielsweise unterschiedliche Verfahren, die auf Evolutionären Algorithmen (z.B. auf Genetischen Algorithmen oder Evolutionärer Programmierung) basieren. Zu dieser Gruppe von Verfahren gehören beispielsweise ENZO und ENZO-M [Bra94, BR96, Bra97], AnnaEleonora [Man94], NeuroGENESYS [HS92], GNARL [ASP94], NetGen [KS97], sGA [DM92] usw.

Evolutionäre Verfahren bieten eine hohe Flexibilität: Mit der Strukturoptimierung läßt sich nicht nur (wie in dieser Arbeit) die Merkmalsselektion kombinieren, auch die Parameter von Lernalgorithmen oder geeignete Aktivierungsfunktionen können gewählt werden. Diese Verfahren sind zwar oft mit hohem Rechenaufwand verbunden, aber gleichzeitig für eine parallele Ausführung geradezu prädestiniert. Auch das Trainieren des Gewichte erfolgt in manchen der Verfahren mit Hilfe Evolutionärer Algorithmen (siehe z.B. [ASP94, DM92, KS97, MW94, PP97, Yao93]). Sogar das Einbeziehen der Evolutionsparameter selbst in den Evolutionsprozeß ist möglich. Die Vorteile eines evolutionären Ansatzes werden beispielsweise beim Vergleich des Systems ENZO mit verschiedenen destruktiven (*magnitude based pruning, OBS, unit-OBS*) und konstruktiven (Cascade-Correlation, FlexNet) Verfahren in [RBL97] deutlich.

Eine Reihe Evolutionärer Verfahren zur Strukturoptimierung für Neuronale Netze (meist für Mehrlagige Perzeptren) wurde in den letzten Jahren entwickelt (siehe beispielsweise [Bra95, Bra97, BK92, DM92, HS92, Man94, PP97, RBL97, RT95, VZL97]; weitere Referenzen sind in [Bra95, Bra97, BW93a] zu finden). Wenige dieser Verfahren basieren auf dynamischen Netzparadigmen: Eine Strukturevolution wird beispielsweise in [ASP94, HM94, MW94] für rekurrente Netzparadigmen oder in [MW94] für Netzparadigmen mit gleitendem Eingangsfenster untersucht. Ein für Time-Delay-Netze geeignetes Verfahren zur Strukturoptimierung und / oder Merkmalsauswahl auf der Basis Evolutionärer Verfahren wird in [Sic00, Wei98] beschrieben.

5 Modellbildung mit Neuronalen Netzen

In diesem Abschnitt werden zunächst einige Empfehlungen für die Durchführung von Trainingsexperimenten mit Neuronalen Netzen gegeben, bevor auf konkrete Bewertungskriterien für trainierte Netze eingegangen wird.

5.1 Anmerkungen zur Durchführung von Versuchen

Folgende allgemeine Kriterien sind bei der Entwicklung eines neuronalen Modells (z.B. ein Regler basierend auf dem Paradigma TDNN) zu beachten:

(1) Das Modell soll nicht nur für Trainings-, sondern vor allem auch für unbekannte Testdaten genaue Ergebnisse liefern (*Generalisierungsfähigkeit*).

(2) Bei einem erneuten Trainingsdurchgang mit wieder neu zufällig initialisierten Gewichten sollen ähnlich gute Ergebnisse erzielt werden können (*Reproduzierbarkeit*).

(3) Das Modell soll möglichst unabhängig von geringen Variationen der Netzstruktur (z.B. Zahl verdeckter Schichten, Zahl innerer Knoten) und der Lernparameter (z.B. minimale und maximale Schrittweite bei RPROP, Epochenzahl) sein (*Robustheit*).

Zur Bewertung der *Generalisierungsfähigkeit* der neuronalen Modelle werden die trainierten Netze mit unbekannten Testmustern getestet. Noch sicherer ist es, dabei gleichzeitig eine *k-fache Kreuzvalidierung* (*k-fold cross-validation*) durchzuführen: Der (Gesamt-)Mustersatz wird dabei in k gleich große Teile aufgeteilt und jeder Teil wird als Testdatenmenge für ein Netz verwendet, das mit den jeweils anderen Daten trainiert wird. Aus den Bewertungsmaßen der k einzelnen Trainingsdurchgänge wird

dann ein Durchschnittswert berechnet. Die Anwendung des Prinzips der Kreuzvalidierung ist insbesondere dann zu empfehlen, wenn insgesamt relativ wenige Muster zur Verfügung stehen.

Die für TDNN üblicherweise verwendeten Lernalgorithmen (siehe Abschnitt 3) können nicht garantieren, in einem globalen Minimum der Fehlerfläche zu konvergieren. Je „zerklüfteter“ eine Fehlerfläche ist (u.a. abhängig von der Zahl trainierbarer Gewichte), desto leichter konvergiert der Lernalgorithmus in einem lokalen Minimum, das unter Umständen weit entfernt vom globalen Minimum liegt. Wiederholt man nun einen Trainingsdurchgang mehrfach, so kann man Streuungen der Ergebnisse beobachten. D.h., bei einem Trainingsdurchgang können gute, bei einem erneuten Trainingsdurchgang auch schlechte Ergebnisse für Trainings- und Testdaten erzielt werden. Der Grund hierfür ist die Initialisierung der Gewichte mit zufällig erzeugten, betragsmäßig kleinen Startwerten bei Beginn des Trainings. Bei Lernalgorithmen, die nach jedem Muster (bei musterweisem Lernen) oder bei jeder Mustersequenz (bei inkrementellem Lernen) die Gewichte adaptieren, ist zudem die Reihenfolge, in der die Muster präsentiert werden, von Bedeutung. Bei einem „guten“ neuronalen Modell sollten die Streuungen der Ergebnisse gering sein (hohe *Reproduzierbarkeit*). Nur dann kann man mit hoher Wahrscheinlichkeit davon ausgehen, daß ein beliebig ausgewähltes Netz aus einem der Trainingsdurchgänge auch in einer späteren Anwendung, d.h. für weitere unbekannte Muster, genaue Ergebnisse liefert. Im allgemeinen wäre es falsch, aus einer Menge von Trainingsdurchgängen eines Neuronalen Netzes das bezüglich unbekannter Testdaten beste Netz auszuwählen.

Um ein neuronales Modell zu bewerten, ist es also notwendig, mehrere Trainingsdurchgänge durchzuführen. Eine sinnvolle Anzahl von Wiederholungen hängt von der Höhe der Streuungen der Ergebnisse ab. Bei einer Überanpassung sind die Streuungen für Testdaten oft deutlich größer als die Streuungen für Trainingsdaten. Neben den Durchschnittswerten der Ergebnisse aller Trainingsdurchgänge sollten auch empirische Standardabweichungen, Konfidenzintervalle o.ä. für die untersuchten Bewertungskriterien (z.B. Schätzfehler oder Klassifikationsraten) angegeben werden. Eine Menge von mit einer identischen Netzarchitektur durchgeführten Trainingsdurchgängen wird im folgenden als *Versuch* bezeichnet, ein einzelner Trainingsdurchgang als *Test* (oder *Versuchswiederholung*).

Erst auf der Basis mehrerer Tests ist ein sinnvoller Vergleich zweier Netzarchitekturen (Versuche) möglich. Eine bloße Betrachtung beispielsweise der Differenz zweier gemittelter Ergebnisse genügt allerdings nicht. Stattdessen muß eine Aussage darüber getroffen werden, inwieweit ein bestimmtes beobachtetes Phänomen im statistischen Sinne signifikant ist oder nicht. Ein statistischer Test ermöglicht beispielsweise auch eine Aussage darüber, mit welcher Wahrscheinlichkeit beim Vergleich zweier Versuchsergebnisse, z.B. anhand der mittleren Schätzfehler aus mehreren Tests, eine (Null-)Hypothese (z.B. „zwei Netzarchitekturen sind gleichwertig“) fälschlicherweise abgelehnt wird („falscher Alarm“, *Fehler 1. Art*) oder auch

fälschlicherweise akzeptiert wird („versäumte Gelegenheit“, *Fehler 2. Art*) [Sac97]. Die Aussagen sind um so signifikanter, je höher die Zahl an Versuchswiederholungen ist. Diese Zahl muß dabei nicht notwendigerweise in zwei Versuchen, die verglichen werden, identisch sein.

Um ein neuronales Modell zu optimieren, bieten sich verschiedene Ansatzpunkte:

- die Auswahl einer geeigneten Menge von Eingangsgrößen (Merkmalen),
- die Optimierung der Netzstruktur (z.B. Zahl von Neuronen in verdeckter Schichten oder maximale Verzögerungen in den Verbindungen) und
- eine optimale Einstellung der Lernparameter (z.B. maximaler Updatewert bei Temporal RPROP).

Geringfügige Modifikationen eines Parameterwertes, der die einem neuronalen Modell zu einem beliebigen Zeitpunkt zur Berechnung der Ausgabewerte zur Verfügung stehende Information bestimmt (z.B. Merkmale oder maximale Verzögerungen), sollten jeweils gleichartige Änderungen (Verbesserungen oder Verschlechterungen) der Ergebnisse für Lerndaten bzw. unbekannte Testdaten bewirken. Insbesondere soll die Wahl eines bezüglich der Ergebnisse für Lerndaten „guten“ Modells auch für unbekannte Testdaten genaue Ergebnisse liefern. Würde man neuronale Modelle nur hinsichtlich der Resultate für Testdaten optimieren, so würde die Testdatenmenge effektiv als Trainingsdatenmenge verwendet. In diesem Fall müßte eine dritte, unabhängige Datenmenge für einen abschließenden Test bereitgestellt werden.

Für alle anderen Ansatzpunkte (z.B. Parameter von Lernalgorithmen) wird hier gefordert, daß ein „gutes“ neuronales Modell eine möglichst geringe Empfindlichkeit gegenüber kleineren Variationen von Parameterwerten besitzen sollte (*Robustheit*). Die geforderte Robustheit soll eine leichte Handhabbarkeit des neuronalen Modells ermöglichen.

Besondere Erwähnung verdient an dieser Stelle noch der Lernparameter „Epochenzahl“. In Abschnitt 3.1 wird beschrieben, wie sich der oben erwähnte Effekt der Überanpassung vermeiden läßt, wenn das Training nach einer relativ niedrigen Epochenzahl abgebrochen wird. Der Grund hierfür ist, daß Netze, bei denen mit fortschreitender Trainingsdauer allmählich eine Überanpassung auftritt, bei kürzeren Epochenzahlen häufig zwar schlechtere Ergebnisse für Trainingsdaten, aber bessere Ergebnisse für unbekannte Testdaten zeigen. Erfolgt die Festlegung einer geeigneten Epochenzahl jedoch mit Blick auf die Resultate für diese unbekannten Testdaten, so ist dieses Verfahren als sehr problematisch anzusehen. Besser sind Modelle, bei denen nach einer gewissen „Mindestepochenzahl“ eine Fortsetzung des Trainings nicht zu einer Verschlechterung der Ergebnisse für unbekannte Testdaten führt. D.h., die Trainingsdauer sollte insgesamt einen eher geringen Einfluß haben. Eine schnelle

Konvergenz (und damit kurze Trainingszeit) ist dabei für Anwendungen, bei denen die Neuronalen Netze vor ihrem Einsatz trainiert werden, zweitrangig.

Wieso wurden diese allgemeinen, scheinbar so selbstverständlichen Anforderungen an neuronale Modelle hier so ausführlich erörtert? Leider ist die Bewertung neuronaler Modelle anhand der angegebenen Kriterien in der Praxis nicht immer üblich: In einer in [Fle96] beschriebenen Studie wurden 61 praxisorientierte Artikel der führenden Zeitschriften *Neural Networks* und *Neural Computation* ausgewertet. Dabei wurde festgestellt, daß nur in 72% der Veröffentlichungen explizit die Verwendung unterschiedlicher Trainings- und Testdatenmengen angegeben war. In lediglich 57% der Veröffentlichungen wurde die Reproduzierbarkeit mit Hilfe mehrfacher Trainingsdurchgänge getestet, wobei nur in 28% der Fälle Varianzen oder Konfidenzintervalle angegeben waren. Beim Vergleich zweier Netze wurden nur in 5% der Artikel statistische Bewertungsmaßstäbe verwendet. In 95% der Fälle wurde nicht klar, wie Modellparameter optimiert wurden.

5.2 Bewertungskriterien für Neuronale Modelle

Zur Bewertung eines trainierten Netzes können verschiedenste Kriterien herangezogen werden. Naheliegend ist beispielsweise, den gesamten quadrierten Fehler zu verwenden, der beim Training des Netzes optimiert wird (siehe Definition 3.2). Leider ist dieses Fehlermaß wenig anschaulich. Häufig ist es sinnvoll, zur Bewertung durchschnittliche und maximale Fehler für Trainings- und unbekannte Testdaten zu bestimmen und zu vergleichen.

Zur Bewertung eines trainierten Netzes werden insgesamt $P \in \mathbb{N}$ Muster von (möglicherweise) mehreren Mustersequenzen untersucht. Bei der Bestimmung der Fehler für Trainingsdaten entspricht P der Anzahl der Muster einer Lernaufgabe, die mit K bezeichnet wurde (vgl. Definition 3.1). In jedem Versuch werden $R \in \mathbb{N}$ Tests durchgeführt. Im folgenden bezeichne $y_1^{(r)}(p)$ mit $r \in \{1, \ldots, R\}$ und $p \in \{1, \ldots, P\}$ die tatsächliche Ausgabe des ersten Ausgabeneurons für das p-te Muster im r-ten Test (vgl. Definition 2.1). Die erwünschte Ausgabe (Soll-Ausgabe) des ersten Ausgabeneurons für das p-te Muster wird – wie bisher – mit $t_1(p)$ notiert; sie ist natürlich in allen Tests eines Versuchs gleich. Der Einfachkeit halber wird hier von Netzen mit nur einem Ausgabeneuron ausgegangen. Bei mehreren Ausgabeneuronen sind die Bewertungskriterien analog anwendbar.

Nun können verschiedene Kriterien zur Bewertung neuronaler Modelle definiert werden.

Definition 5.1 (*Kriterien zur Bewertung eines Tests*)
Der *durchschnittliche Fehler* $\emptyset_r$ *bei einem Test* $r \in \{1, \ldots, R\}$ ist definiert durch

$$\emptyset_r \stackrel{def}{=} \frac{1}{P} \sum_{p=1}^{P} \left| t_1(p) - y_1^{(r)}(p) \right| .$$

Der *maximale Fehler* $\max_r$ *bei einem Test* $r \in \{1, \ldots, R\}$ ist definiert durch

$$\max_r \stackrel{def}{=} \max_p \left\{ \left| t_1(p) - y_1^{(r)}(p) \right| \right\} .$$

Der durchschnittliche Fehler eines Tests beschreibt das (Gesamt-)Verhalten eines Netzes in einem bestimmten Test (Trainingsdurchgang) r eines Versuchs; der maximale Fehler gibt die schlechteste Schätzung in diesem Test an (*worst-case*-Verhalten). Beide Größen werden sowohl für Trainingsdaten als auch für unbekannte Testdaten bestimmt, um die Approximations- und die Generalisierungsfähigkeit der neuronalen Modelle zu beschreiben.

Definition 5.2 (*Kriterien zur Bewertung eines Versuchs – Teil 1*)
Der *Mittelwert der durchschnittlichen Fehler aller Tests eines Versuchs* (kurz: *mittlerer durchschnittlicher Fehler*) $\mu_{(\emptyset_r)}$ ist definiert durch

$$\mu_{(\emptyset_r)} \stackrel{def}{=} \frac{1}{R} \sum_{r=1}^{R} \emptyset_r .$$

Die *Streuung der durchschnittlichen Fehler aller Tests eines Versuchs* (oder *empirische Standardabweichung der durchschnittlichen Fehler*) $\sigma_{(\emptyset_r)}$ ist definiert durch

$$\sigma_{(\emptyset_r)} \stackrel{def}{=} \sqrt{\frac{1}{R-1} \sum_{r=1}^{R} \left(\emptyset_r - \mu_{(\emptyset_r)} \right)^2} .$$

In analoger Weise sind der *minimale durchschnittliche Fehler* $\min_{(\emptyset_r)}$ *eines Versuchs*, der *maximale durchschnittliche Fehler* $\max_{(\emptyset_r)}$ *eines Versuchs* und der *Median der durchschnittlichen Fehler* $\mathrm{med}_{(\emptyset_r)}$ *eines Versuchs* definiert.

Der mittlere durchschnittliche Fehler ist ein erwartungstreuer Schätzer für den Erwartungswert der Verteilung der durchschnittlichen Fehler; die Streuung der durchschnittlichen Fehler ist ein erwartungstreuer Schätzer für die Standardabweichung. Beide Werte zusammen sind ein Maß für die Reproduzierbarkeit von Trainingsergebnissen. Um unterschiedliche Versuche besser vergleichen zu können, kann die Streuung bezogen auf den Mittelwert angegeben werden, d.h. der Quotient aus $\sigma_{(\emptyset_r)}$ und $\mu_{(\emptyset_r)}$. Minimaler und maximaler durchschnittlicher Fehler beschreiben zusammen die *Spannweite* der bei den Tests erzielten Ergebnisse und der Median gibt einen Hinweis auf die *Symmetrie* der Verteilung [Sac97].

Definition 5.3 (*Kriterien zur Bewertung eines Versuchs – Teil 2*)
Der *Mittelwert der maximalen Fehler aller Tests eines Versuchs* (kurz: *mittlerer maximaler Fehler*) $\mu_{(\max_r)}$ ist definiert durch

$$\mu_{(\max_r)} \stackrel{def}{=} \frac{1}{R} \sum_{r=1}^{R} \max_r .$$

Die *Streuung der maximalen Fehler aller Tests eines Versuchs* (oder *empirische Standardabweichung der maximalen Fehler*) $\sigma_{(\max_r)}$ ist definiert durch

$$\sigma_{(\max_r)} \stackrel{def}{=} \sqrt{\frac{1}{R-1} \sum_{r=1}^{R} \left(\max_r - \mu_{(\max_r)}\right)^2}.$$

In analoger Weise sind der *minimale maximale Fehler* $\min_{(\max_r)}$ *eines Versuchs*, der *maximale maximale Fehler* $\max_{(\max_r)}$ *eines Versuchs* und der *Median der maximalen Fehler* $\mathrm{med}_{(\max_r)}$ *eines Versuchs* definiert.

Da es sich bei den verwendeten Lernalgorithmen um stochastische Optimierungsverfahren handelt, deren Ergebnis von einer zufälligen Initialisierung der synaptischen Gewichte oder – bei musterweisem Lernen – auch von der Präsentationsreihenfolge der Muster abhängt, kann nicht garantiert werden, daß diese Verfahren in einem globalen Minimum der Fehlerfläche konvergieren. Vielmehr ist zu erwarten, daß das Trainingsergebnis nach einer bestimmten, fest gewählten Epochenzahl nur mehr oder weniger nah in der Nähe des globalen Optimums liegt. Infolgedessen wird für den Vergleich zweier Versuche, der hier auf dem Vergleich der in den beiden Versuchen erzielten mittleren durchschnittlichen Fehler sowie auf den Streuungen der durchschnittlichen Fehler basiert, ein *statistischer* Test benötigt, der eine Aussage darüber ermöglicht, ob ein Versuch mit (hoher) Wahrscheinlichkeit besser als ein anderer ist oder nicht. Der Test untersucht die *Signifikanz* der Differenz zweier mittlerer durchschnittlicher Fehler. Kann man nun vereinfachend annehmen, daß die durchschnittlichen Fehler in jedem Versuch näherungsweise normalverteilt sind, so läßt sich dieses Problem abstrakt als „Vergleich zweier Mittelwerte unabhängiger Stichproben aus angenähert normalverteilten Grundgesamtheiten bei unbekannten und eventuell verschiedenen Varianzen“ charakterisieren (sog. *Behrens-Fisher-Problem*) [Bos97].

In Anlehnung an den in [Bos97] beschriebenen *t-Test* (einseitig) kann nun das folgende Verfahren zum Vergleich zweier Versuche definiert werden (siehe auch [Sac97]). Die mittleren durchschnittlichen Fehler, die Streuungen der durchschnittlichen Fehler und die jeweilige Anzahl der Tests in zwei Versuchen A und B werden zur Unterscheidung mit entsprechenden zusätzlichen Indizes versehen.

Algorithmus 5.4 (*Verfahren zum Vergleich zweier Versuche*)
Zwei Versuche A und B sollen verglichen werden; die Größen $\mu^{(A)}_{(\varnothing_r)}$ und $\mu^{(B)}_{(\varnothing_r)}$ (mittlere durchschnittliche Fehler), $\sigma^{(A)}_{(\varnothing_r)}$ und $\sigma^{(B)}_{(\varnothing_r)}$ (Streuungen der durchschnittlichen Fehler)

sowie $R^{(A)}$ und $R^{(B)}$ (Anzahl der Tests) sind gegeben. Zuerst werden der *Wert der Stichprobenfunktion*

$$\hat{t} \stackrel{def}{=} \frac{\mu^{(A)}_{(\emptyset_r)} - \mu^{(B)}_{(\emptyset_r)}}{\sqrt{\frac{\left(\sigma^{(A)}_{(\emptyset_r)}\right)^2}{R^{(A)}} + \frac{\left(\sigma^{(B)}_{(\emptyset_r)}\right)^2}{R^{(B)}}}}$$

und die *Zahl der Freiheitsgrade*

$$\nu \stackrel{def}{=} \min\{R^{(A)}, R^{(B)}\} - 1$$

bestimmt. Dann werden die beiden (Teil-)Aussagen

$$A1 : \hat{t} > t_{\nu;\alpha} \qquad \text{und} \qquad A2 : \hat{t} < -t_{\nu;\alpha}$$

überprüft, wobei der Wert $t_{\nu;\alpha}$ $(0 < \alpha < 1)$ einer Tabelle der t-*Verteilung* (*Student-Verteilung*) zu entnehmen ist.

Trifft nun $A1$ zu und $A2$ nicht, so heißt Versuch B *besser* als Versuch A *auf einem Signifikanzniveau von* α. Trifft $A2$ zu und $A1$ nicht, so heißt umgekehrt A *besser* als B *auf einem Signifikanzniveau von* α. Treffen beide Teilaussagen nicht zu, so kann im Vergleich von A und B keine Entscheidung auf einem bestimmten, vorgegebenen Signifikanzniveau α getroffen werden.

Tabelle 3: Ausschnitt aus einer Tabelle der t-Verteilung

$\nu \setminus \alpha$	0.1	0.05	0.01	0.005	0.001
4	1.533	2.135	3.747	4.604	7.173
9	1.383	1.833	2.821	3.250	4.297
14	1.345	1.761	2.624	2.977	3.787
19	1.328	1.729	2.539	2.861	3.579
24	1.318	1.711	2.492	2.797	3.467

Beispielsweise ist also ein Versuch B dann besser als ein Versuch A, wenn der mittlere durchschnittliche Fehler geringer und gleichzeitig die Streuungen ausreichend klein sind. Das Ergebnis des Vergleichs ist um so zuverlässiger, je kleiner α gewählt wird. Üblicherweise wird α mit 0.01 oder 0.05 angesetzt. In [Sac97] wird $R^{(A)} \geq 6$ und $R^{(B)} \geq 6$ empfohlen. Verschiedene Werte von $t_{\nu;\alpha}$ sind Tabelle 3 zu entnehmen [Sac97].

Beispiel 5.5
Gegeben sind zwei Versuche A ($\mu^{(A)}_{(\emptyset_r)} = 43.51$, $\sigma^{(A)}_{(\emptyset_r)} = 1.41$ und $R^{(A)} = 25$) und B ($\mu^{(B)}_{(\emptyset_r)} = 45.15$, $\sigma^{(B)}_{(\emptyset_r)} = 1.43$ und $R^{(B)} = 25$), die miteinander verglichen werden. Der

Wert der Stichprobenfunktion ist somit $\hat{t} \approx -4.08$. Als Signifikanzniveau wird $\alpha = 0.01$ gewählt, die Anzahl der Freiheitsgrade ist $\nu = 24$. Der benötigte Wert der t-Verteilung ist $t_{24;0.99} = 2.492$. Damit ist die Teilaussage $A1$ falsch, die Teilaussage $A2$ wahr und demnach der Versuch A besser als der Versuch B auf einem 1%-Signifikanzniveau.

Dieses Beispiel gibt eine ungefähre Vorstellung davon, wie nah bei einer Wiederholungszahl von 25 die mittleren durchschnittlichen Fehler zweier Versuche bei vorgegebenen empirischen Standardabweichungen beieinanderliegen können, obwohl ein Unterschied der Versuche auf einem 1%-Signifikanzniveau feststellbar ist.

Werden Neuronale Netze für Klassifikationsaufgaben eingesetzt, so können auch verschiedene andere Kriterien zur Bewertung herangezogen werden.

6 Einsatz Neuronaler Netze als Regler

Die folgenden Ausführungen stützen sich im wesentlichen auf [MSW90, And98, Wan98, NKK96].

Neuronale Regelung gibt es beinahe so lange, wie Neuronale Netze erforscht werden. Mitte der sechziger Jahre wurde von Widrow und Smith erstmals ein Neuronales Netz als Neuronaler Regler eingesetzt. Sie wendeten Widrow und Hoffs Adaline-Netz auf das Problem der Regelung eines *inversen Pendels* an. In den siebziger Jahren entwickelte Albus CMAC, eine komplexe auf Neuronalen Netzen basierende Reglerarchitektur, für die Regelung eines Roboterarmmanipulators. Barto, Sutton und Anderson veröffentlichten Mitte der achtziger Jahre ihre Resultate bzgl. der Methode des adaptiven Kritikers. Seit der Publikation des Backpropagation-Algorithmus ist die Zahl der Veröffentlichungen und Anwendungen auf dem Gebiet der Neuronalen Regelung sprunghaft angewachsen.

Die zentrale Problematik dieser Methoden ist bisher die unzureichende Analyse ihrer Stabilität, Empfindlichkeit und Robustheit. Dies ist bei vielen veröffentlichten Beispielanwendungen, wie dem inversen Pendel und dem Problem *Truck Backer-Upper* zu verschmerzen. Bei dem zweiten Benchmark-Problem handelt es sich um das simulierte Einparken eines Lastwagens (rückwärts an ein Ladedock, mit Anhänger). Wenn die Pendelstange doch einmal umkippt bzw. der Lastwagen in einer Simulation nicht korrekt rückwärts eingeparkt wird, hat dies keine Konsequenzen. Bei realen Anwendungen können aber Menschenleben oder teure Maschinen auf dem Spiel stehen. Deshalb ist die Erforschung Neuronaler Regler noch lange nicht abgeschlossen.

Die Aufgaben eines Neuronalen Reglers entsprechen denen eines herkömmlichen Reglers. Während jedoch ein normaler Regler durch eine mathematisch-physikalische Analyse des zu regelnden Systems erzeugt werden muß, werden Neu-

ronale Regler mit Hilfe der Lernfähigkeit der ihnen zugrundeliegenden konnektionistischen Modelle gebildet.

6.1 Klassifikation der Neuronalen Reglerarchitekturen

Eine Möglichkeit, Neuronale Regler in verschiedene Klassen aufzuteilen, besteht darin, sie nach der Art, wie das Fehlersignal für den Regler berechnet wird, zu klassifizieren. Dabei kann man vier Hauptklassen unterscheiden:

- **Überwachte Regler:** Neuronale Netze werden trainiert, das Verhalten eines anderen Reglers nachzuahmen.
- **Verstärkend lernende Regler:** Neuronale Regler werden nach dem Prinzip des *Reinforcement-Learning* trainiert.
- **Vorhersagende Regler:** Regler verwenden einen Neuronalen Emulator der Regelstrecke und eine Optimierungsfunktion, um zukünftige Werte des Regelsignals vorherzusagen.
- **Selbstanpassende Regler:** Neuronale Regler, deren freie Parameter so adaptiert werden, daß eine gegebene Kostenfunktion minimiert wird.

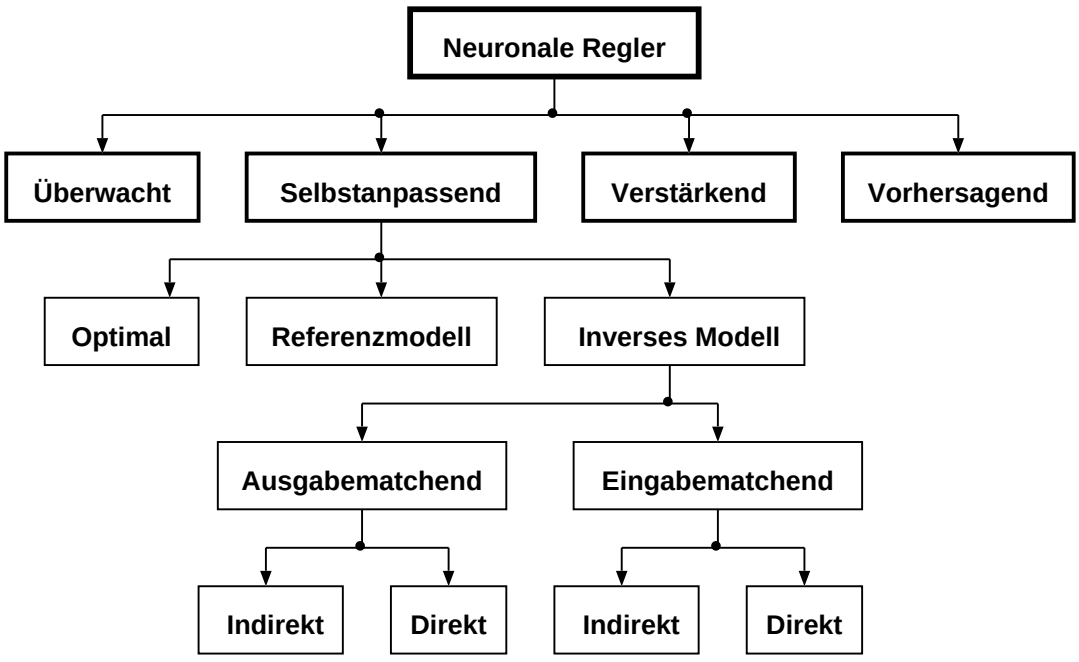


Abbildung 18: Klassifizierung Neuronaler Regler

Die vollständige Klassifizierung aus [And98] ist in Abbildung 18 dargestellt. Im folgenden werden aber nur die eben genannten (Haupt-)Klassen näher erläutert.

6.2 Überwachte Regler oder Klone

Ein Neuronales Netz wird mit dem bekannten Regelverhalten eines anderen Reglers trainiert. Diese Art des Neuronalen Reglers kann online oder offline trainiert werden.

Bei ersterem Verfahren verwendet man meist den in Abbildung 19 dargestellten Aufbau. Bei der auf den Reglereingang rückgekoppelten Ausgabe der Regelstrecke muß es sich nicht um ein elektronisches Signal handeln. Dies ist beispielsweise der Fall, wenn es sich bei dem Online-Regler um einen Menschen handelt, dessen Verhalten nachgebildet werden soll. Zu dem vorhandenen Regelkreis wird der Neuronale Regler parallel geschaltet, wobei er als Eingabe die Ausgabe der Regelstrecke erhält. Daraus berechnet er eine Reglerausgangsgröße. Diese wird mit dem Ergebnis des Online-Reglers verglichen, und die Differenz zwischen beiden Werten ist das Fehlersignal, mit welchem die Gewichte des Neuronalen Netzes angepaßt werden.

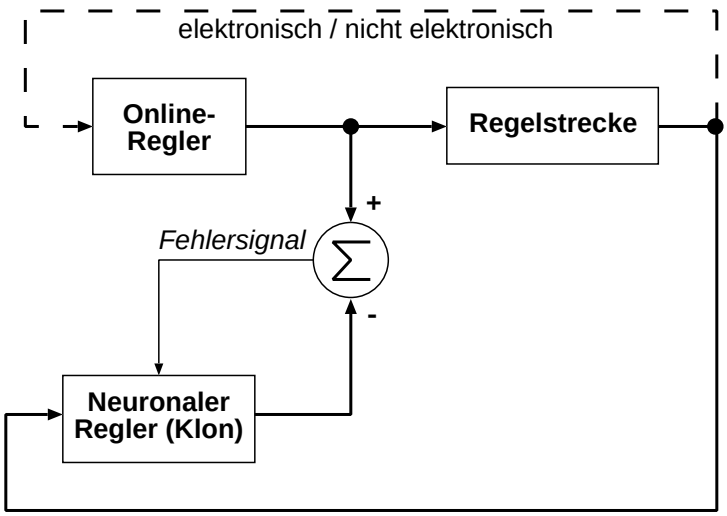


Abbildung 19: Online erzeugter Klon

Manchmal ist es aber sinnvoll, offline zu trainieren, da das Neuronale Netz so schneller lernen kann (siehe Abbildung 20). Dazu gewinnt man aus einem bestehenden Regelkreis Lern- und Testdaten, die aus Eingangs- und Ausgangswerten des beobachteten Reglers bestehen. Diese Daten bilden die feste Lernaufgabe für den Neuronalen Regler.

Da in beiden Fällen eine (mehr oder weniger) exakte Kopie des Originalreglers entsteht, bezeichnet man ein in dieser Weise trainiertes Netz auch als Klon. Nach dem Training ersetzt ein Klon den vorhandenen Regler.

Die Erzeugung eines Klons ist nützlich, falls der zu ersetzende Regler ein Mensch ist, bzw. wenn der Regelalgorithmus sehr rechenintensiv ist, aber eine von einem Neuronalen Netz erlernbare Ein-/Ausgabeabbildung besitzt. Ziel könnte beispielsweise in diesem Fall sein, einen Neuronalen Regler mit kürzerer Reaktionszeit zu

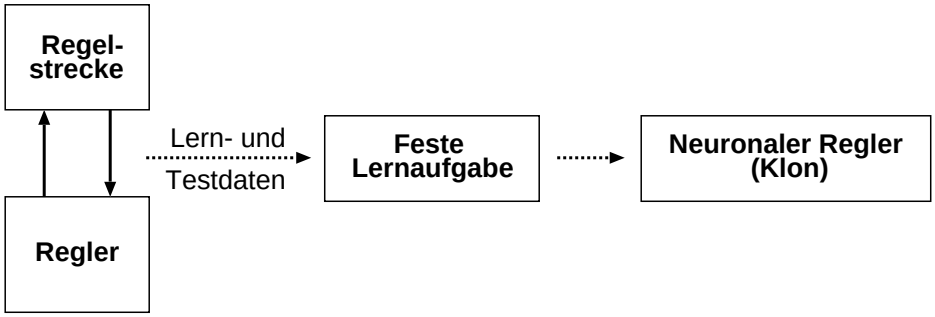


Abbildung 20: Offline erzeugter Klon

finden, der bestimmten Echtzeitbedingungen genügt. Auch wenn die Regelstrecke durch einen zufällig initialisierten Neuronalen Regler beschädigt werden könnte, ist Klonen empfehlenswert.

6.3 Verstärkend lernende Regler

In vielen Anwendungen gibt es keinen vorhandenen Regler, der kopiert werden könnte, d.h. keine feste Lernaufgabe. Aus diesem Grunde verwendet man Neuronale Regler, die mit einem Algorithmus für *verstärkendes Lernen* (*reinforcement learning*) trainiert werden. Dazu muß man nicht die korrekte Ausgabe zu einem Eingabemuster kennen, sondern man muß die Ausgabe bzw. die durch sie in der Umgebung des Netzwerkes hervorgerufene Veränderung als gut oder schlecht bewerten können. Verstärkendes Lernen kann also als Lernen mit Hilfe eines Kritikers statt eines Lehrers bezeichnet werden. Ein Lehrer sagt seinem Schüler, wie er etwas machen soll. Dagegen teilt ein Kritiker nur mit, ob man etwas gut oder schlecht gemacht hat. Auf einen Neuronalen Regler angewandt, sieht der Aufbau wie in Abbildung 21 aus. Der Kritiker ist in den Regelkreis derart eingeschaltet, daß er die Eingaben des Neuronalen Reglers (die Regeldifferenz) und dessen Ausgaben (die Reglerausgangsgröße) beobachten kann. Je nach Größe der Regeldifferenz, die sich aus Soll- und Istwert berechnet, beurteilt der Kritiker die Entscheidung des Neuronalen Netzes als gut oder schlecht und teilt dies dem Neuronalen Regler mit Hilfe des Reinforcement-Signals mit.

Wann eine Ausgabe als gut oder schlecht bewertet wird, hängt von der jeweiligen Umgebung bzw. Aufgabe des Netzes ab. Bezüglich einer Regelungsaufgabe ist es z.B. üblich, eine Ausgabe als gut zu klassifizieren, wenn die Regelung noch nicht versagt hat, und anderenfalls als schlecht zu bewerten.

Nachteilig bei dieser binären Bewertung einer Ausgabe ist, daß sie dem Netz nur sehr wenig Informationen zur Veränderung seiner Gewichte liefert. Außerdem wird möglicherweise nicht das gesamte vorhandene Wissen genutzt.

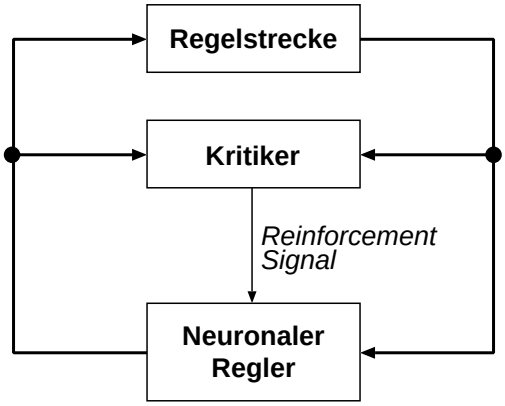


Abbildung 21: Lernen mit einem Kritiker

Die Veränderung der Netzwerkstruktur wird so vorgenommen, daß die excitatorischen Gewichte, die verstärkend auf die aktuelle Ausgabe wirken, „bestraft“ werden, wenn die Ausgabe mit „schlecht“ bewertet wird, bzw. „belohnt“ werden, wenn der Ausgabe die Bewertung „gut“ zugeordnet wird. Mit inhibitorischen Gewichten, die abschwächend auf die aktuelle Ausgabe wirken, wird dementsprechend entgegengesetzt verfahren. Die „Belohnung“ bzw. „Bestrafung“ von Gewichten äußert sich in einer Erhöhung bzw. Erniedrigung ihres Betrages.

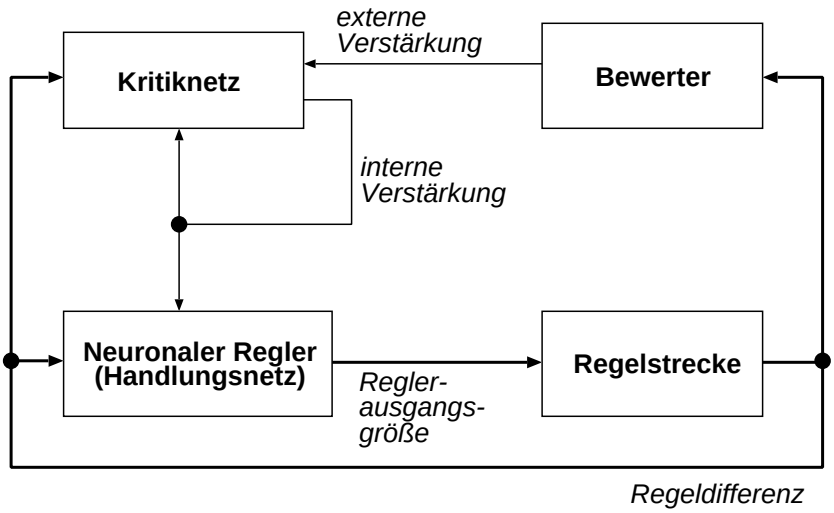


Abbildung 22: Handlungs- und Kritiknetz

Eine Ausweitung dieses Ansatzes stellt die Verwendung eines adaptiven Kritiknetzes neben einem Handlungsnetz dar. Der eigentliche Regelkreis besteht wieder aus dem Neuronalen Regler (dem sog. Handlungsnetz) und der Regelstrecke. Ein Bewerter beurteilt die Ausgabe der Regelstrecke als gut oder schlecht und teilt dies dem Kritiknetz mit (externe Verstärkung). Dieses lernt anhand der externen Verstärkung und der jeweils aktuellen Ausgabe der Regelstrecke die Bewertung des nächsten Systemzustandes vorherzusagen. Das als Ausgabe produzierte interne reelle Verstärkungs-

signal dient nun sowohl zur Gewichtsänderung im Kritiknetz als auch im Handlungsnetz. Der Vorteil dieses Ansatzes ist, daß die vorhandene Information dazu benutzt wird, dem Neuronalen Regler mit einem reellwertigen Fehlersignal mehr als nur die Zustände gut oder schlecht mitzuteilen. Dies erleichtert die adäquate Anpassung der Gewichte. Diese Vorgehensweise soll dafür sorgen, daß das aus beiden Teilnetzen gebildete System lernt, „schlechte“ Zustände zu erkennen und zu vermeiden.

Der Vorteil der beiden vorgestellten Ansätze liegt im „Lernen durch Probieren“. D.h., die verwendete Lernaufgabe muß nicht von Beginn an vorliegen, sondern kann sukzessive in Realzeit erzeugt werden, indem der Neuronale Regler verschiedene Ansätze ausprobiert. Voraussetzung dafür ist allerdings, daß der Regler seine Aktionen austesten kann, ohne z.B. die Versuchsanordnung zu zerstören.

6.4 Vorhersagende Regler

Vorhersagende Neuronale Regler benutzen einen Neuronalen Emulator der Regelstrecke und einen Optimierungsalgorithmus, der eine Kostenfunktion minimiert, um zukünftige Reglersignale zu berechnen. Die Kostenfunktion J könnte beispielsweise folgendermaßen definiert sein:

$$J\left(v(k), v(k+1), \ldots, v(k+N-1)\right) \stackrel{def}{=} \sum_{i=1}^{N} \left(r(k+i) - \hat{y}(k+d+i)\right)^2 + \sum_{i=1}^{N} \lambda_i \cdot \left(v(k+i-1) - v(k+i-2)\right)^2 .$$

Dabei ist $d \in \mathbb{N}$ die zeitliche Verzögerung der Regelstrecke, $N \in \mathbb{N}$ die Anzahl von Zeitschritten, die durch die Kostenfunktion berücksichtigt wird, $\lambda_i \in \mathbb{R}$ mit $i = 1, \ldots, N$ die relative Bedeutung künftiger Änderungen des Regelsignals, $v : \mathbb{N} \to \mathbb{R}$ das Regelsignal, das berechnet wird, $\hat{y} : \mathbb{N} \to \mathbb{R}$ die Ausgabe des Neuronalen Emulators der Regelstrecke und $r : \mathbb{N} \to \mathbb{R}$ das Referenzsignal, das den Sollwert bereitstellt.

Der Emulator kann fest gewählt werden oder auch online adaptiert werden. Die Optimierungsroutine erzeugt Regelsignale $(v(k), v(k+1), \ldots, v(k+N-1))$, wobei $v(k)$ direkt auf die Regelstrecke angewandt werden kann (siehe Abbildung 23). Alternativ kann $v(k)$ aber auch dazu benutzt werden, einen Neuronalen Regler zu trainieren, der seinerseits dann das Regelsignal für die Regelstrecke erzeugt (siehe Abbildung 24).

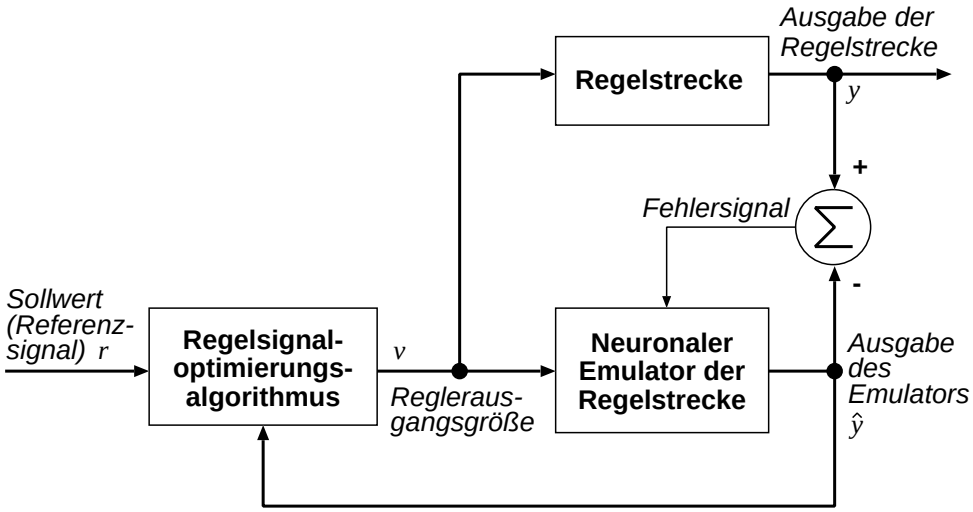


Abbildung 23: Vorhersagender Regler – Typ 1

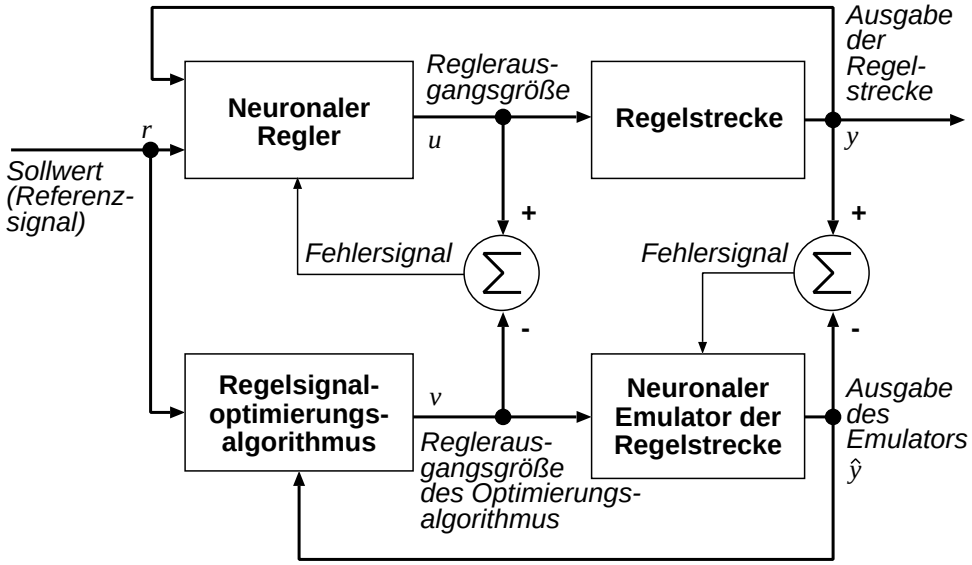


Abbildung 24: Vorhersagender Regler – Typ 2

6.5 Adaptive Regler

Als adaptiv werden solche Regler bezeichnet, deren freie Parameter so angepaßt werden, daß eine gegebene Kostenfunktion minimiert wird. Dies wird meist mit Gradientenabstiegsverfahren erreicht. Abbildung 25 zeigt den prinzipiellen Aufbau eines solchen Reglers. Der Neuronale Regler berechnet aus den Soll- und Istwerten das Regelsignal für die Regelstrecke. Soll-, Istwert und Regelsignal werden einem kostenfunktionsoptimierenden Algorithmus zugeleitet. Dieser Algorithmus berechnet gemäß einer vorgegebenen Kostenfunktion ein Fehlersignal, welches dem Neuronalen Regler übermittelt wird. Dieser paßt dann seine Gewichte dementsprechend an.

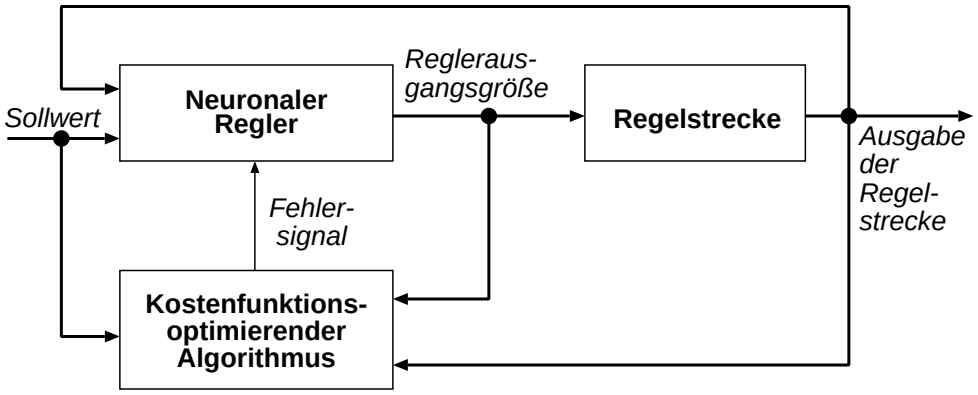


Abbildung 25: Adaptiver Regler

7 Module für Neuronale Netze in ICONNECT

Im folgenden Abschnitt werden die in ICONNECT (Version 2.0) realisierten Module vorgestellt, die es ermöglichen, TDNN (bzw. MLP oder MLP-sw) für Klassifikations- oder Regelungsaufgaben einzusetzen. Dabei handelt es sich zunächst um das Modul TDNN, mit dem ein TDNN (bzw. MLP oder MLP-sw) simuliert wird. Zur Steuerung des Datenflusses für dieses Modules sind die Module Repeat, zum Wiederholen einmal eingelesener Daten, und Pre-NN, eine Art Multiplexer-Modul, zuständig.

7.1 Das Modul TDNN

Das Modul TDNN (siehe Abbildung 26) simuliert ein TDNN (siehe Definition 2.1). Durch Setzen der Zeitverzögerungen aller Schichten auf Eins erhält man ein statisches MLP, das bei geeigneter Aufbereitung der Eingangsdaten auch als MLP-sw eingesetzt werden kann (siehe Abschnitt 2.2).

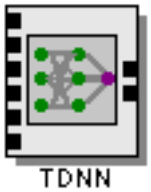


Abbildung 26: Das Modul TDNN

7.1.1 Parameter des Moduls TDNN

Die Parameter des Moduls TDNN (siehe Abbildung 27) lassen sich grob in Parameter, die die Netzstruktur betreffen, und Parameter zum Trainieren des Netzes aufteilen. Die Netzstruktur beschränkt sich auf zwei bis vier Neuronenebenen, die schichtweise vollständig miteinander verbunden werden. Die Eingabemaske beschränkt die Zahl der Neuronen in jeder Schicht auf 32768.

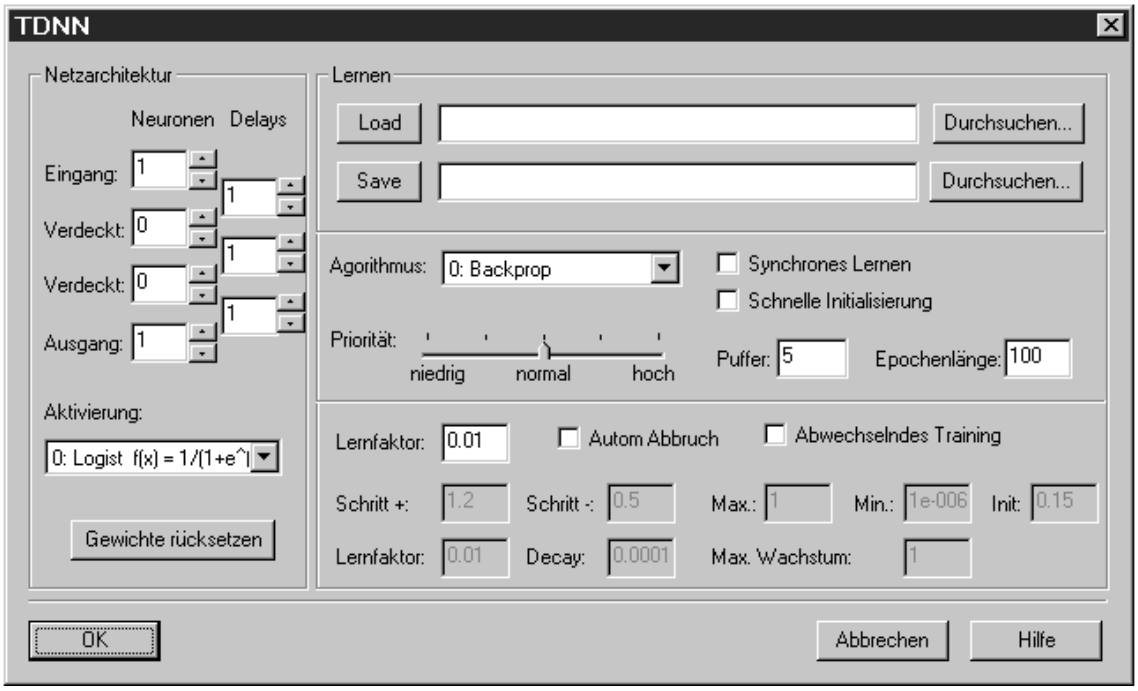


Abbildung 27: Parameter des Moduls TDNN

Im Dialog existiert für jede Schicht ein Eingabefeld, in dem die Anzahl an Neuronen in dieser Schicht eingestellt werden kann. Für ein zweischichtiges Netz sind nur die Neuronenzahlen der Eingabe- und Ausgabeschicht nötig. Die Neuronenzahlen der verdeckten Schichten müssen hier auf Null gesetzt werden. Für Netze mit drei Schichten ist zusätzlich die Zahl der Neuronen in der ersten verdeckten Schicht (oberes Eingabefeld) auf einen Wert grösser Null zu setzen, für ein Netz mit vier Schichten die Neuronenzahl der beiden verdeckten Schichten. Entsprechend verhält es sich mit den Zeitverzögerungen. Für ein Netz bestehend aus n Schichten sind $n-1$ Eingabefelder von oben gesehen auszufüllen. Die Eingabefelder für nicht benutzte Verbindungen müssen bei den Verzögerungen auf Eins gesetzt werden. So ist z.B. ein TDNN bestehend aus 3 Schichten mit 7,4,2 Neuronen und den Verzögerungen 5 zwischen Eingabe- und verdeckter Schicht, sowie einer Verzögerung von 3 zwischen verdeckter und Ausgabeschicht (d.h. ein Netz mit der Struktur $7 \stackrel{5}{\rightarrow} 4 \stackrel{3}{\rightarrow} 2$, vgl. Abschnitt 2.1), folgendermaßen einzugeben: Neuronenzahlen 7,4,0,2, Verzögerungen 5,3,1. Die in Abbildung 27 dargestellte Einstellung entspricht einem Netz mit der Struktur $1 \stackrel{1}{\rightarrow} 1$.

Mit zur Netzarchitektur gehört auch die verwendete Aktivierungsfunktion. Diese ist in allen Neuronen eines Netzes gleich. Das Modul TDNN stellt folgende Funktionen zur Verfügung: logistische Aktivierung, Tangens Hyperbolicus und „schnelle“ Aktivierung (siehe Tabelle 1 und Abbildungen 2 und 3). Diese Funktionen erfüllen alle Bedingungen, um beim Training mittels Gradientenabstieg verwendet werden zu können. Charakteristisch für jede Funktion sind dabei der Wertebereich sowie die Ableitung am Punkt $x = 0$. Je größer dieser Wert ist, umso schneller passen sich die Gewichte zu Beginn eines Trainings, wenn die Aktivierungen der Neuronen nach einer zufälligen Initialisierung der Gewichte nahe Null sind, an (siehe Abschnitt 2.1).

Dieses Initialisieren der Verbindungsgewichte mit zufälligen, kleinen Werten erfolgt automatisch, sobald ein neues Netz erzeugt wurde, oder auch nach einem Klick auf den Button „Gewichte rücksetzen“.

Zum Lernen sind folgende Parameter verfügbar. Zunächst muß einer von vier zur Verfügung stehenden Lernalgorithmen ausgewählt werden. Dies sind im Einzelnen: *(Temporal) Backpropagation* (siehe Abschnitt 3.2), *(Temporal) Resilient Backpropagation* (RPROP, siehe Abschnitt 3.3.2), *(Temporal) Quickpropagation* (QPROP, siehe Abschnitt 3.3.1) und *(Temporal) Resilient Quickpropagation* (QRPROP, siehe [Roj96]). Der Namenszusatz *Temporal* rührt aus der Verwendung der Algorithmen in TDNN her. Setzt man die einzelnen Verzögerungen auf Eins, entspricht die Funktion jedes Algorithmus dem jeweils entsprechenden „klassischen“ Gradientenabstiegsverfahren für MLP, d.h. ohne Namenszusatz.

Die zu den Algorithmen gehörenden Lernparameter sind im Bereich rechts unten im Dialog einzustellen. Bei Wahl eines Algorithmus aus dem Dropdown-Feld werden automatisch alle nichtzugehörigen Felder deaktiviert. Die Bedeutung der einzelnen Parameter ist mittels Tabelle 4 aus der Beschreibung der Trainingsalgorithmen in Abschnitt 3 zu erkennen.

Bei zeitverzögerten Netzen tritt bei den ersten Eingaben in das Netz das Problem auf, daß sich die Ausgabe des Netzes auch auf frühere Aktivierungen der Neuronen bezieht. Diese enthalten allerdings noch keine sinnvollen Werte, so daß es anfangs zu zufälligen, sprunghaften Ausgaben des Netzes kommen kann. Dies will man vor allem bei Regelungsaufgaben vermeiden. Zu diesem Zwecke existiert die Checkbox „Schnelle Initialisierung“. In diesem Modus wird die erste Eingabe so oft durch das Netz geschickt, bis alle Aktivierungen mit (einigermaßen) „sinnvollen“ Werten initialisiert sind. Das oben beschriebene Verhalten findet dann nicht mehr statt.

Da der Lernvorgang in einem eigenen Thread implementiert wurde, kann man über den Schiebregler „Priorität“ die Priorität des Lernthreads in fünf Stufen einstellen. Die einzelnen Schritte sind dabei

- `THREAD_PRIORITY_LOWEST`,

Tabelle 4: Zuordnung der Namen der Lernparameter

Name im Dialog	Name in der Dokumentation
Backpropagation	
Lernfaktor	Lernrate
Quickprop	
Lernfaktor	Lernrate
Decay	Weight-Decay-Faktor
max. Wachstum	max. Wachstumsfaktor
RProp und QRProp	
Schritt +	Zunahmefaktor
Schritt −	Abnahmefaktor
Max	maximaler Updatewert
Min	minimaler Updatewert
Init	initialer Updatewert

- `THREAD_PRIORITY_BELOW_NORMAL`,
- `THREAD_PRIORITY_NORMAL`,
- `THREAD_PRIORITY_HIGHEST` und
- `THREAD_PRIORITY_TIME_CRITICAL`.

Die Priorität Time-Critical wurde verwendet, da ICONNECT-Signalgraphen per Default ebenfalls in dieser Prioritätsklasse laufen. Da aber vor allem beim Training des Netzes die gesamte Rechenzeit verwendet wird, sollte beachtet werden, daß der Rechner nicht mehr bedienbar ist, sobald die Anzahl der Prozesse der Stufe time-critical (nicht der Threads) mit voller CPU-Nutzung gleich oder größer ist als die Anzahl der zur Verfügung stehenden Prozessoren. Die Begriffe Prozeße und Threads wurden dabei wie unter Windows NT üblich verwendet: Ein Prozeß erhält seinen eigenen virtuellen Speicher und wird beim Starten eines Programms erstellt. Während des Ablaufs des Programms können Unterprogramme im selben Speicherbereich als Threads gestartet werden, um eine parallele Abarbeitung zu ermöglichen.

Die Trainingsdaten gelangen über einen Puffer zum Lernthread. Die Größe des Puffers kann über das entsprechende Eingabefeld eingestellt werden, und hängt stark von der Priorität des Lernprozesses und der sonstigen Rechnerauslastung ab. Werte unter 100 sollten vermieden werden. Kommt es zu einem Pufferüberlauf, wird der Lernthread abgeschossen und wieder neu gestartet. Dadurch gehen einige Trainingsmuster verloren.

Will man dies auf jeden Fall vermeiden, kann man den Checkbutton „Synchrones

Lernen“ anwählen. Damit wird der Lernthread mit dem Hauptthread so synchronisiert, daß jedesmal, nachdem ein neues Muster eingelesen wurde, auch ein Lernschritt stattfindet. Wird das Netz „offline“ trainiert, ist dieser Lernmodus dem puffernden Lernen vorzuziehen.

Um mit dem Lernen beginnen zu können, muß das Netz noch wissen, wieviele Muster es jeweils zu einer Epoche zusammenfassen soll. Dies wird im Eingabefeld „Epochenlänge“ angegeben. Steht dort eine Null, stellt jeweils ein Paket (Superblock) eine Epoche dar. Ansonsten werden immer soviele Muster zu einer Epoche zusammengefaßt, wie in diesem Feld angegeben.

Das Feld „Epochenlänge“ wird deaktiviert, wenn man die Checkbox „Abwechselndes Lernen“ anwählt. In diesem Modus können dem Netz nach jeder Epoche Muster präsentiert werden, die nicht zum Training zur Verfügung standen. Diese Muster werden hier als „unbekannte Muster“ bezeichnet, im Gegensatz zu den „bekannten Mustern“, die zum Training verwendet wurden (siehe Abschnitt 5.1). Da sich hier zwei unterschiedliche Epochenlängen ergeben können, findet keine Einstellung im Modul TDNN statt; stattdessen werden diese im Parameterdialog des Moduls PreNN (siehe Abschnitt 7.3) eingestellt, welches für diesen Modus dem Modul TDNN vorausgeschaltet werden muß.

Die Checkbox „Automatischer Abbruch“ sorgt dafür, daß das Training abgebrochen wird, sobald sich der Epochenfehler während des Trainings kaum noch verbessert. Dieser Epochenfehler berechnet sich wie folgt: Zunächst wird für jedes Muster die Summe der quadratischen Differenzen aus der tatsächlichen und der Sollausgabe über alle Ausgangsneuronen berechnet. Diese Fehler werden in jeder Epoche aufsummiert und bilden den Epochenfehler. Beim Epochenfehler, der ausgegeben wird, und auch beim hier verwendeten Epochenfehler handelt es sich um einen auf eine Epoche der Länge 100 normierten Epochenfehler. Um feststellen zu können, ob das Training abgebrochen werden soll, wird ein gleitendes Mittel über jeweils 50 Epochenfehler berechnet. Abgebrochen wird, wenn sich der Trend nach 50 Epochenfehlern um weniger als 0.01 verbessert hat. Diese Parameter sind allerdings nicht über den Dialog einstellbar, sondern müssen über `#define`-Anweisungen im Sourcecode angepaßt werden.

Über die Buttons „Load“ und „Save“ können die Gewichte bereits trainierter Netze geladen bzw. gespeichert werden. Der Dateiname ist in den jeweils nebenstehenden Eingabefeldern anzugeben. Dieser kann entweder direkt eingegeben werden, oder mittels Klick auf den jeweiligen Button „Durchsuchen“ aus einem Filedialog ausgewählt werden. Die Dateien sollten auf „.nnw“ enden. Die Filedialoge filtern automatisch diese Dateien.

7.1.2 Ein- und Ausgänge des Moduls TDNN

Die Eingänge des Moduls sind in Tabelle 5 aufgeführt, die Ausgänge in Tabelle 6. Die genaue Bedeutung der Ein- und Ausgänge wird im folgenden Abschnitt erklärt.

Tabelle 5: Eingänge des Moduls TDNN

Name	Type	Beschreibung
Eingabe	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Dateineingang für Trainingsmuster
Sollausgabe	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Teachinput, Sollausgaben
Lernen Ein/Aus	`TYPEINFO {Typeinfo}` `SWORD[1] {TIME_DOMAIN}`	aktiviert (1) bzw. deaktiviert (0) den Trainingsmodus
Gewichte laden	`TYPEINFO {Typeinfo}` `SWORD[1] {TIME_DOMAIN}`	Lädt die Gewichte aus einer im Dialog festgelegten Datei
Gewichte speichern	`TYPEINFO {Typeinfo}` `SWORD[1] {TIME_DOMAIN}`	Speichert die Gewichte in einer im Dialog festgelegten Datei
Reset	`TYPEINFO {Typeinfo}` `SWORD[1] {TIME_DOMAIN}`	Setzt die Gewichte auf zufällige Werte zwischen -1 und 1

Tabelle 6: Ausgänge des Moduls TDNN

Name	Type	Beschreibung
Ausgabe	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Die Ausgabe des Netzes
Fehler	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Der normierte Epochenfehler der letzten Epoche

7.1.3 Funktionsweise des Moduls TDNN

Das Modul TDNN kennt zwei Betriebszustände. Zunächst einen sogenannten Verarbeitungszustand, und dann einen Trainingszustand. Letzterer läßt sich erweitern zu einem verifizierenden Trainieren. Welcher der beiden Hauptzustände aktiv ist, läßt sich über den Eingang „Lernen Ein/Aus“ festlegen. Erhält dieser Eingang eine Eins, wird das Training gestartet, bei einer Null wieder gestoppt.

Im Verarbeitungszustand nimmt das Netz über den Eingang „Eingabe“ die Eingabe in das Netz blockweise entgegen. Die Länge eines Blockes muß dabei gleich sein mit der Anzahl der Neuronen in der Eingabeschicht. Diese Eingabe wird anschließend durch das Netz zur Ausgabeschicht propagiert, und als Block der Länge gleich der Anzahl der Neuronen in der Ausgabeschicht über den Ausgang „Ausgabe“ ausgegeben. Paketinformationen bleiben dabei erhalten. Über den zweiten Ausgang „Fehler“ werden keine Werte ausgegeben. An Eingängen ist in diesem Modus nur noch der Eingang „Lernen Ein/Aus“ nötig, der dann auf Null stehen muß.

Im Trainingszustand ist zusätzlich der Eingang „Sollausgabe“ erforderlich. An diesem Eingang erwartet das Netz Blöcke der Länge gleich der Anzahl der Neuronen in der Ausgabeschicht, die es als Sollausgaben zu den Trainingsmustern am Eingang „Eingabe“ verwendet. Beim Start des Signalgraphen in ICONNECT wird ein eigenständiger Thread zum Trainieren gestartet, dem die Trainingsmuster mit den Sollausgaben gepuffert übergeben werden. Wie im Verarbeitungszustand wird auch hier die Netzausgabe über den Ausgang „Ausgabe“ ausgegeben. Zusätzlich wird hier nach jeder Epoche der Epochenfehler, normiert auf 100 Muster, über den Ausgang „Fehler“ ausgegeben. Dieses Normieren ermöglicht einen Vergleich des Fehler auch bei unterschiedlichen Epochenlängen (Berechnung des Epochenfehlers siehe Abschnitt 7.1.1, automatischer Abbruch).

Wurde ein Dateiname im Parameterdialog (siehe Abbildung 27) zum Speichern des Netzes angegeben (Dateiendung „.nnw“), wird der Epochenfehler zusätzlich in einer Datei mitprotokolliert, die denselben Namen trägt, im selben Verzeichnis liegt, aber auf „.asc“ endet. Nach jedem Epochenfehler folgt ein „,“, dem die Epochenlänge in Klammern folgt. Ist über die entsprechende Checkbox „Automatischer Abbruch“ aktiviert, folgt in derselben Zeile nach einem Tabulator der Wert des Trends, und nach einem weiteren Tabulatur die Differenz zum Trend vor 50 Werten (siehe Abschnitt 7.1.1). Durch das Trennen der Werte durch Kommata und Tabulatoren ist ein einfacher Import der Datei als Textfile in Microsoft Excel möglich, um die Fehlerverläufe graphisch darstellen zu können. Bevor allerdings mit dem Schreiben dieser oben beschriebenen Zeilen begonnen wird, folgt ein einmaliger Eintrag zu Beginn. Dort sind Informationen über die Netzstruktur, den verwendeten Lernalgorithmus, sowie alle Parameter aller Lernalgorithmen zu finden (siehe Abbildung 28).

Optional steht beim Trainingsvorgang aber auch noch die Möglichkeit zur Verfügung, während des Trainings die Generalisierungsfähigkeit des Netzes zu überprüfen. Zusammen mit dem Modul Pre-NN (siehe Abschnitt 7.3) wird dem Netz dann nach jeder Epoche ein zweiter Satz an Mustern und Sollausgaben präsentiert. Bei diesem zweiten Satz sollte es sich um Muster handeln, die nicht zum Training verwendet wurden. Dieser Modus ist aktiv, wenn im Eigenschaften-Dialog (siehe Abbildung 27) die Checkbox „Abwechselndes Lernen“ aktiviert wurde. Die Epochenlängen der beiden Mustersätze werden dann im Parameterdialog des Moduls Pre-NN eingestellt. Das Modul TDNN erkennt dann anhand des Umschaltens des

```
Epochenlaenge: abwechselndes Training
Netzstruktur: 20-15-4-1
Delays: 1-1-1
Aktivierungsfunktion: 0: Logist  f(x) = 1/(1+e^(-x))
Algorithmus: 1: R-Prop

Lernparameter:
BStep: 0.010000

QDecay: 0.000100
QStep:  0.010000
QDecay: 0.000100

RMax:       1.000000
RMin:       0.000001
RInit:      0.010000
RMinusStep: 0.500000
RPlusStep:  1.200000

Epochenfehler:
40.593386,(6972)
40.489613,(819)
40.023464,(6972)
40.542356,(819)
....
```

Abbildung 28: Beispiel eines Trainingsprotokolls

Lernmodus durch das Modul Pre-NN diese Epochenlängen. Über den Ausgang „Fehler“ werden jetzt abwechselnd die Fehler für die bekannten bzw. unbekannten Muster ausgegeben.

Alle übrigen Eingänge des Moduls TDNN sind optional. Wird an „Gewichte Laden“ bzw. „Gewichte Speichern“ eine Eins geschickt, lädt bzw. speichert das Netz seine aktuellen Gewichte in der im Parameterdialog (siehe Abbildung 27) festgelegten Datei. Wurde dort kein Name angegeben, erscheint ein Dateimanager. Die ausgewählte Datei wird anschließend automatisch in den Dialog übernommen. Erhält der Eingang „Reset“ eine Eins, werden alle Gewichte zufällig mit kleinen Werten zwischen -1 und 1 initialisiert.

7.2 Das Modul Repeat

Das Modul Repeat (siehe Abbildung 29) wiederholt ein einmal eingelesenes Paket (Superblock) beliebig, also auch unendlich oft.

Abbildung 29: Das Modul Repeat

7.2.1 Parameter des Moduls Repeat

Abbildung 30 zeigt die Parameter des Moduls Repeat. Es kann die Anzahl der Wiederholungen eines Pakets eingestellt werden.

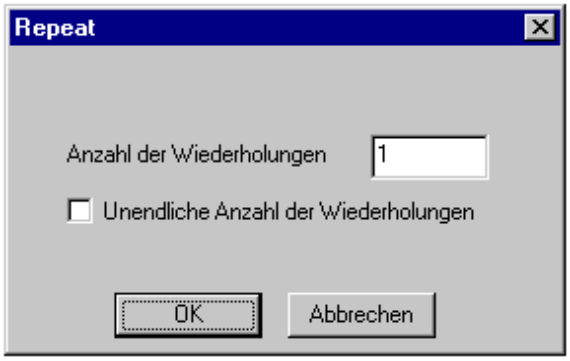


Abbildung 30: Parameter des Moduls Repeat

7.2.2 Ein- und Ausgänge des Moduls Repeat

Die Ein- und Ausgänge des Moduls Repeat sind in Tabelle 7 aufgeführt.

7.2.3 Funktionsweise des Moduls Repeat

Das Modul Repeat liest nach dem Start zunächst über seinen Eingang genau ein Paket ein. Danach nimmt es keine weiteren Eingaben mehr entgegen und gibt dieses Paket so oft auf seinem ersten Ausgang wieder aus, wie im Parameterdialog eingestellt wurde (siehe Abbildung 30), also auch unendlich oft. Die Ausgabe bildet dabei wieder genau ein Paket. Im Fall von unendlich vielen Wiederholungen wird also nie ein Stop-Status geschickt, d.h., das Paket wird nicht abgeschlossen. Auch

Tabelle 7: Ein- und Ausgänge des Moduls Repeat

Name	Type	Beschreibung
Signal In	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Einlesen eines Pakets
Name	**Type**	**Beschreibung**
Signal Out	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Datenausgang
Repeats	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Anzahl der vollständigen Wiederholungen

wenn das Modul Repeat seine Ausgabe beendet hat, liest es keine weiteren Daten mehr ein.

Über den zweiten Ausgang des Moduls wird die aktuelle Anzahl an bereits stattgefundenen Wiederholungen ausgegeben.

7.3 Das Modul Pre-NN

Das Modul Pre-NN (siehe Abbildung 31) wird vom Modul TDNN benötigt, um während des Trainings den Fehler zu ermitteln, den das Netz auf unbekannten Trainingsmustern macht. Dadurch wird eine Abschätzung der Generalisierungsfähigkeit des Netzes während des Trainingsverlaufs möglich. Dieser kann, falls nötig, entsprechend vorzeitig abgebrochen werden, um eine Überanpassung zu vermeiden.

Abbildung 31: Das Modul Pre-NN

7.3.1 Parameter des Moduls Pre-NN

Abbildung 32 zeigt die Parameter des Moduls Pre-NN. Es sind die Epochenlängen für Trainings- bzw. Textmuster anzugeben (bekannte bzw. unbekannte Muster).

Abbildung 32: Parameter des Moduls Pre-NN

7.3.2 Ein- und Ausgänge des Moduls Pre-NN

Die Eingänge des Moduls sind in Tabelle 8 aufgeführt, die Ausgänge in Tabelle 9.

Tabelle 8: Eingänge des Moduls Pre-NN

Name	Type	Beschreibung
bekannte Muster	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Einlesen der bekannten Muster
bekannte Sollausgabe	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Einlesen der bekannten Sollausgaben
unbekannte Muster	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Einlesen der unbekannten Muster
unbekannte Sollausgabe	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Einlesen der unbekannten Sollausgaben

Tabelle 9: Ausgänge des Moduls Pre-NN

Name	Type	Beschreibung
Merkmale	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Ausgabe der Merkmale
Lernmodus	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Ein- und Ausschalten des Lernmodus beim nachfolgenden Modul TDNN
Sollausgaben	`TYPEINFO {Typeinfo}` `DOUBLE[] {TIME_DOMAIN}`	Ausgabe der Sollausgaben

7.3.3 Funktionsweise des Moduls Pre-NN

Das Modul Pre-NN hat die Aufgabe, zwischen jeweils zwei Datenströmen hin- und herzuschalten. Die ersten beiden Eingänge sind für die bekannten Muster mit den zugehörigen Sollausgaben gedacht. Eingänge drei und vier analog für unbekannte Muster und deren Sollausgaben. Die Epochenlängen für die bekannten und unbekannten Muster können dabei unterschiedlich sein. Sie werden im Parameterdialog eingestellt (siehe Abbildung 32).

Das Modul besitzt drei Ausgänge. Der erste und der dritte werden verwendet, um die bekannten oder unbekannten Muster- und Sollausgabenströme weiterzugeben. Mit dem zweiten Ausgang wird der Lernmodus im Modul TDNN ein- und ausgeschaltet. Aus dem Umschalten des Lernmodus erkennt das Modul TDNN die Epochenlängen. (Parameter „Abwechselndes Lernen“ im Modul TDNN).

Literatur

[And98] Andersen, H. C. A.: *The Controller Output Error Method*; Dissertation; University of Queensland; St Lucia 4072, Australia, Juni 1998

[ASP94] Angeline, P. J.; Saunders, G. M.; Pollack, J. P.: *An Evolutionary Algorithm that Constructs Recurrent Neural Networks*; in: *IEEE Transactions on Neural Networks*; 1994; Bd. 5 (1); S. 54 – 65

[Bis95] Bishop, C. M.: *Neural Networks for Pattern Recognition*; Clarendon Press, Oxford, 1995

[BK92] Boers, E. J. W.; Kuiper, H.: *Biological Metaphors and the Design of Modular Artificial Neural Networks*; Diplomarbeit; Leiden University, Departments of Computer Science and Experimental and Theoretical Psychology, 1992

[Bod90] Bodenhausen, U.: *The Tempo-Algorithm: Learning in a Neural Network with Variable Time-Delays*; in: *International Joint Conference on Neural Networks (IJCNN '90),* Washington D.C.; 1990; Bd. 1; S. 597 – 600

[Bos97] Bosch, K.: *Lexikon der Statistik – Nachschlagewerk für Anwender*; R. Oldenbourg Verlag, München, Wien, 1997; 2. Aufl.

[BPR97] Behnke, S.; Pfister, M.; Rojas, R.: *Recognition of Handwritten Digits Using Structural Information*; in: *Proceedings of the 1997 International Conference on Neural Networks (ICNN '97),* Houston; 1997; Bd. 3; S. 1391 – 1396

[BR96] Braun, H.; Ragg, T.: *ENZO: Evolution of Neural Networks*; Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 1996; User Manual and Implementation Guide, Version 1.0

[Bra94] Braun, H.: *Evolution – A Paradigm for Constructing Intelligent Agents*; in: *Proceedings of the International ZiF-Conference: Prerational Intelligence – Phenomenology of Complexity Emerging in Systems of Simple Interacting Agents,* Bielefeld (Hg. Cruse, H.; Ritter, H.; Dean, J.); 1994; Bd. 2; S. 31 – 40

[Bra95] Branke, J.: *Evolutionary Algorithms for Neural Network Design and Training*; in: *1st Nordic Workshop on Genetic Algorithms and its Applications,* Vaasa; 1995; S. 145 – 163

[Bra97] Braun, H.: *Neuronale Netze: Optimierung durch Lernen und Evolution*; Springer-Verlag, Berlin, Heidelberg, New York, 1997

[BT96] Back, A. D.; Tsoi, A. C.: *Aspects of Adaptive Learning Algorithms for FIR Feedforward Networks*; in: *Proceedings of the International Conference on Neural Information Processing (ICONIP '96),* Hong Kong; 1996; Bd. 2; S. 1311 – 1316

[BW91] Bodenhausen, U.; Waibel, A.: *The Tempo 2 Algorithm: Adjusting Time-Delays by Supervised Learning*; in: *Advances in Neural Information Processing Systems* (Hg. Lippmann, R. P.; Moody, J. E.; Touretzky, D. S.); Morgan Kaufmann Publishers, San Mateo, 1991; Bd. 3; S. 155 – 161

[BW93a] Bodenhausen, U.; Waibel, A.: *Application Oriented Automatic Structuring of Time-Delay Neural Networks for High Performance Character and Speech Recognition*; in: *Proceedings of the 1993 IEEE International Conference on Neural Networks (ICNN '93),* San Francisco; 1993; Bd. 3; S. 1627 – 1632

[BW93b] Braun, H.; Weisbrod, J.: *Evolving Neural Networks for Application Oriented Problems*; in: *Proceedings of the 2nd Annual Conference on Evolutionary Programming,* La Jolla; 1993; S. 62 – 71

[BWLT94] Back, A.; Wan, E. A.; Lawrence, S.; Tsoi, A. C.: *A Unifying View of Some Training Algorithms for Multilayer Perceptrons with FIR Filter Synapses*; in: *Neural Networks for Signal Processing IV* (Hg. Vlontzos, J.; Hwang, J.-N.; Wilson, E.); IEEE Press, New York, 1994; S. 146 – 154; (Proceedings of the NNSP '94 IEEE Workshop, Ermioni)

[CG96] Cancelliere, R.; Gemello, R.: *Efficient Training of Time Delay Neural Networks for Sequential Patterns*; in: *Neurocomputing*; 1996; Bd. 10 (1); S. 33 – 42

[CGHC97] Clouse, D. S.; Giles, C. L.; Horne, B. G.; Cottrell, G. W.: *Time-Delay Neural Networks: Representation and Induction of Finite-State Machines*; in: *IEEE Transactions on Neural Networks*; 1997; Bd. 8 (5); S. 1065 – 1070

[Chi95] Chi, Z.: *MLP Classifiers: Overtraining and Solutions*; in: *Proceedings of the IEEE International Conference on Neural Networks,* Perth; 1995; Bd. 5; S. 2821 – 2824

[CP98] Coggins, K. M.; Principe, J.: *Detection and Classification of Insect Sounds in a Grain Silo Using a Neural Network*; in: *Proceedings of the 1998 International Joint Conference on Neural Networks (IJCNN '98),* Anchorage; 1998; S. 1760 – 1765

[CU93] Cichocki, A.; Unbehauen, R.: *Neural Networks for Optimization and Signal Processing*; John Wiley & Sons, Chichester, New York, 1993

[CZ98] Cholewo, T. J.; Zurada, J. M.: *Exact Hessian Calculation in Feedforward FIR Neural Networks*; in: *Proceedings of the 1998 International Joint Conference on Neural Networks (IJCNN '98),* Anchorage; 1998; S. 1074 – 1077

[DD93] Day, S. P.; Davenport, M. R.: *Continuous-Time Temporal Back-Propagation with Adaptable Time Delays*; in: *IEEE Transactions on Neural Networks*; 1993; Bd. 4 (2); S. 348 – 354

[DGT98] Denby, B.; Golé, P.; Tarniewicz, J.: *TDNN Approach to Measuring Raindrop Sizes and Velocities*; in: *Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN '98),* Skövde (Hg. Niklasson, L.; Bodén, M.; Ziemke, T.); Springer Verlag, London, 1998; Bd. 1; S. 269 – 274

[DM92] Dasgupta, D.; McGregor, D. R.: *Designing Application-Specific Neural Networks using the Structured Genetic Algorithm.*; in: *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks: COGANN-92,* Baltimore; 1992; S. 87 – 96

[Ell93] Elliot, D. L.: *A Better Activation Function for Artificial Neural Networks*; Technischer Bericht TR-93-8; University of Maryland, Institute for Systems Research, 1993

[Elm90] Elman, J. L.: *Finding Structure in Time*; in: *Cognitive Science*; 1990; Bd. 14 (2); S. 179 – 211

[Fle96] Flexer, A.: *Statistical Evaluation of Neural Network Experiments: Minimum Requirements and Current Practice*; in: *Proceedings of the 13th European Meeting on Cybernetics and Systems Research (Cybernetics and Systems '96),* Wien (Hg. Trappl, R.); 1996; S. 1005 – 1008

[Ham87] Hamming, R. W.: *Digitale Filter*; VCH Verlagsgesellschaft, Weinheim, New York, 1987

[Hay94] Haykin, S.: *Neural Networks – A Comprehensive Foundation*; Macmillan College Publishing Company, New York, 1994

[HB98] Howell, J. A.; Buxton, H.: *Learning Gestures with Time-Delay RBF Networks*; in: *Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN '98),* Skövde (Hg. Niklasson, L.; Bodén, M.; Ziemke, T.); Springer Verlag, London, 1998; Bd. 1; S. 239 – 244

[HLW93] Hwang, J.-N.; Li, H.; Wang, C.-J.: *A Limited Feedback Time-Delay Neural Network*; in: *Proceedings of the 1993 International Joint Conference on Neural Networks (IJCNN '93),* Nagoya; 1993; Bd. 1; S. 271 – 274

[HM94] Happel, B. L. M.; Murre, J. M. J.: *Design and Evolution of Modular Neural-Network Architectures*; in: *Neural Networks*; 1994; Bd. 7 (6–7); S. 985 – 1004

[Hof93] Hoffmann, N.: *Kleines Handbuch Neuronale Netze – Anwendungsorientiertes Wissen zum Lernen und Nachschlagen*; Friedrich Vieweg & Sohn Verlagsgesellschaft, Braunschweig, Wiesbaden, 1993

[HS92] Harp, S. A.; Samad, T.: *Optimizing Neural Networks with Genetic Algorithms*; in: *Proceedings of the 54th American Power Conference,* Chicago; 1992; Bd. 2; S. 1138 – 1143

[HSW89] Hornik, K.; Stinchcombe, M.; White, H.: *Multilayer Feedforward Networks are Universal Approximators*; in: *Neural Networks*; 1989; Bd. 2; S. 359 – 366

[KD97] Korczak, J.; Dizdarevic, E.: *Genetic Search for Optimal Neural Networks*; in: *3rd Conference Neural Networks and Their Applications,* Kule; 1997; S. 30 – 45

[KDMM98] Klaassen, A. J.; Driancourt, X.; Muller, S.; Muller, J.-D.: *Classifying Regional Seismic Signals Using TDNN-alike Neural Networks*; in: *Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN '98),* Skövde (Hg. Niklasson, L.; Bodén, M.; Ziemke, T.); Springer Verlag, London, 1998; Bd. 2; S. 809 – 814

[Kei98] Keilhofer, J. P.: *Untersuchung von Methoden zur Fusion von Sensordaten auf unterschiedlichen Ebenen bei der Werkzeugzustandsüberwachung*; Diplomarbeit; Universität Passau, Lehrstuhl für Rechnerstrukturen, 1998

[KS97] Kwaśnicka, H.; Szerszon, P.: *NetGen – Evolutionary Algorithms in Designing Artificial Neural Networks*; in: *3rd Conference Neural Networks and Their Applications,* Kule; 1997; S. 671 – 676

[KWL94] Kreesuradej, W.; Wunsch, D. C.; Lane, M.: *Time Delay Neural Network for Small Time Series Data Sets*; in: *Proceedings of the 1994 World Congress on Neural Networks,* San Diego; 1994; Bd. 2; S. 248 – 253

[LD95] Lin, D.-T.; Dayhoff, J. E.: *Network Unfolding Algorithm and Universal Spatiotemporal Function Approximation*; Technischer Bericht TR-95-6; University of Maryland, Institute for Systems Research, 1995

[LDL92a] Lin, D.-T.; Dayhoff, J. E.; Ligomenides, P. A.: *Adaptive Time-Delay Neural Network for Temporal Correlation and Prediction*; in: *SPIE – Intelligent Robots and Computer Vision XI*; 1992; Bd. 1826; S. 170 – 181

[LDL92b] Lin, D.-T.; Dayhoff, J. E.; Ligomenides, P. A.: *A Learning Algorithm for Adaptive Time-Delays in a Temporal Neural Network*; Technischer Bericht TR-92-59; University of Maryland, Institute for Systems Research, 1992

[LDL94] Lin, D.-T.; Dayhoff, J. E.; Ligomenides, P. A.: *Prediction of Chaotic Time Series and Resolution of Embedding Dynamics with the ATNN*; in: *Proceedings of the 1994 World Congress on Neural Networks,* San Diego; 1994; Bd. 2; S. 231 – 236

[LDL95] Lin, D.-T.; Dayhoff, J. E.; Ligomenides, P. A.: *Trajectory Production with the Adaptive Time-Delay Neural Network*; in: *Neural Networks*; 1995; Bd. 8 (3); S. 447 – 461

[LGHK97] Lin, T.-N.; Giles, C. L.; Horne, B. G.; Kung, S.-Y.: *A Delay Damage Model Selection Algorithm for NARX Neural Networks*; in: *IEEE Transactions on Signal Processing*; 1997; Bd. 45 (11); S. 2719 – 2730; (Special Issue on Neural Networks)

[LHTG96] Lin, T.; Horne, B. G.; Tiňo, P.; Giles, C. L.: *Learning Long-Term Dependencies in NARX Recurrent Neural Networks*; in: *IEEE Transactions on Neural Networks*; 1996; Bd. 7 (6); S. 1329 – 1338

[Lin94] Lin, D.-T.: *The Adaptive Time-Delay Neural Network: Characterization and Applications to Pattern Recognition, Prediction and Signal Processing*; Dissertation; University of Maryland, Faculty of the Graduate School, 1994

[LLBC97] Lavagetto, F.; Lepsøy, S.; Braccini, C.; Curinga, S.: *Lip Motion Modeling and Speech Driven Estimation*; in: *Proceedings of the 1997 International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97),* München; 1997; Bd. 1; S. 183 – 186

[LLD93a] Lin, D.-T.; Ligomenides, P. A.; Dayhoff, J. E.: *Learning with the Adaptive Time-Delay Neural Network*; Technischer Bericht TR-93-49; University of Maryland, Institute for Systems Research, 1993

[LLD93b] Lin, D.-T.; Ligomenides, P. A.; Dayhoff, J. E.: *Spatiotemporal Topology and Temporal Sequence Identification with an Adaptive Time-Delay Neural Network*; in: *SPIE – Intelligent Robots and Computer Vision XII*; 1993; Bd. 2055; S. 536 – 545

[Man94] Maniezzo, V.: *Genetic Evolution of the Topology and Weight Distribution of Neural Networks*; in: *IEEE Transactions on Neural Networks*; 1994; Bd. 5 (1); S. 39 – 53

[May99] Maydl, W.: *Verwendung Neuronaler Netze (NARX und TDNN) als nichtlineare, dynamische Regler*; Diplomarbeit; Universität Passau, Lehrstuhl für Rechnerstrukturen, 1999

[MKB97] Marti, U.-V.; Kaufmann, G.; Bunke, H.: *Cursive Script Recognition with Time Delay Neural Networks Using Learning Hints*; in: *Proceedings of the 7th International Conference on Artificial Neural Networks (ICANN '97),* Lausanne (Hg. Gerstner, W.; Germond, A.; Hasler, M.; Nicoud, J.-D.); Nr. 1327 in Lecture Notes in Computer Science; Springer Verlag, Berlin, Heidelberg, New York, 1997; S. 973 – 978

[Moh94] Mohraz, K.: *Neuronale Netze mit dynamischer Architektur*; Diplomarbeit; Universität Erlangen – Nürnberg, IMMD – Lehrstuhl für Programmier- und Dialogsprachen sowie Compiler, 1994

[MP96] Mohraz, K.; Protzel, P.: *FlexNet – A Flexible Neural Network Construction Algorithm*; in: *Proceedings of the 4th European Symposium on Artificial Neural Networks (ESANN '96),* Brüssel; 1996; S. 111 – 116

[MSW90] Miller, W. T.; Sutton, R. S.; Werbos, P. J. (Hg.): *Neural Networks for Control*; The MIT Press, 1990

[MW94] McDonnell, J. R.; Waagen, D.: *Evolving Recurrent Perceptrons for Time-Series Modeling*; in: *IEEE Transactions on Neural Networks*; 1994; Bd. 5 (1); S. 24 – 38

[NKK96] Nauck, D.; Klawonn, F.; Kruse, R.: *Neuronale Netze und Fuzzy-Systeme*; Friedrich Vieweg & Sohn Verlagsgesellschaft, Braunschweig, Wiesbaden, 1996; 2. Aufl.

[NMSG93] Neumerkel, D.; Murray-Smith, R.; Gollee, H.: *Modelling Dynamic Processes with Clustered Time-Delay Neurons*; in: *Proceedings of the 1993 International Joint Conference on Neural Networks (IJCNN '93),* Nagoya; 1993; Bd. 2; S. 1765 – 1768

[PP97] Pujol, J. C. F.; Poli, R.: *Evolution of the Topology and the Weights of Neural Networks Using Genetic Programming with a Dual Representation*; Technischer Bericht CSRP-97-7; University of Birmingham, School of Computer Science, 1997

[RB93] Riedmiller, M.; Braun, H.: *A Direct Method for Faster Backpropagation Learning: The RPROP Algorithm*; in: *Proceedings of the 1993 IEEE International Conference on Neural Networks (ICNN '93),* San Francisco; 1993; Bd. 1; S. 586 – 591

[RBL97] Ragg, T.; Braun, H.; Landsberg, H.: *A Comparative Study of Neural Network Optimization Techniques*; in: *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA '97),* Norwich; 1997; S. 343 – 347

[Ree93] Reed, R.: *Pruning Algorithms – A Survey*; in: *IEEE Transactions on Neural Networks*; 1993; Bd. 4 (5); S. 740 – 747

[Rie93] Riedmiller, M.: *Untersuchungen zu Konvergenz und Generalisierungsfähigkeit überwachter Lernverfahren mit dem SNNS*; in: *Tagungsband zum Workshop Simulation Neuronaler Netze mit SNNS (SNNS '93),* Stuttgart (Hg. Zell, A.); 1993; S. 107 – 116

[Rie94a] Riedmiller, M.: *Advanced Supervised Learning in Multi-layer Perceptrons – From Backpropagation to Adaptive Learning Algorithms*; in: *International Journal of Computer Standards and Interfaces*; 1994; Bd. 16 (3); S. 265 – 278; (Special Issue on Artificial Neural Networks)

[Rie94b] Riedmiller, M.: *RPROP – Description and Implementation Details*; Technischer Bericht; Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 1994

[Roj96] Rojas, R.: *Neural Networks – A Systematic Introduction*; Springer-Verlag, Berlin, Heidelberg, New York, 1996

[RT95] Roberts, S. G.; Turega, M.: *Evolving Neural Network Structures: An Evaluation of Encoding Techniques*; in: *Artificial Neural Nets and Genetic Algorithms,* Alès (Hg. Pearson, D. W.; Steele, N. C.; Albrecht, R. F.); Springer Verlag, Wien, New York, 1995; S. 96 – 99

[Sac97] Sachs, L.: *Angewandte Statistik – Anwendung statistischer Methoden*; Springer-Verlag, Berlin, Heidelberg, New York, 1997; 8. Aufl.

[Sar96] Sarle, W. (Hg.): *Neural Nets: A List of Frequently Asked Questions (FAQ)*; USENET: comp.ai.neural-nets, 1996; erhältlich durch anonymous ftp unter `ftp://ftp.sas.com/pub/neural/FAQ.html`

[SB97] Sbirrazzuoli, N.; Brunel, D.: *Computational Neural Networks for Mapping Calorimetric Data: Application of Feed-forward Neural Networks to Kinetic Parameters Determination and Signals Filtering*; in: *Neural Computing & Applications*; 1997; Bd. 5 (1); S. 20 – 32

[SBF+98a] Sicheneder, A.; Bender, A.; Fuchs, E.; Mandl, R.; Mendler, M.; Sick, B.: *Tool-supported Software Design and Program Execution for Signal Processing Applications Using Modular Software Components*; in: *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT '98),* Aalborg (Hg. Margaria, T.; Steffen, B.); 1998; S. 61 – 70; (BRICS Notes Series NS-98-4)

[SBF+98b] Sicheneder, A.; Bender, A.; Fuchs, E.; Mandl, R.; Sick, B.: *A Framework for the Graphical Specification and Execution of Complex Signal Processing Applications*; in: *Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing (ICASSP '98),* Seattle; 1998; Bd. 3; S. 1757 – 1760

[SDB97] Schittenkopf, C.; Deco, G.; Brauer, W.: *Two Strategies to Avoid Overfitting in Feedforward Networks*; in: *Neural Networks*; 1997; Bd. 10 (3); S. 505 – 516

[SH95] Setiono, R.; Hui, L. C. K.: *Use of a Quasi-Newton Method in a Feedforward Neural Network Construction Algorithm*; in: *IEEE Transactions on Neural Networks*; 1995; Bd. 6 (1); S. 273 – 277

[She97] Shepherd, A. J.: *Second Order Methods for Neural Networks – Fast and Reliable Training Methods for Multi-Layer Perceptrons*; Springer-Verlag, London, Berlin, Heidelberg, 1997

[Sic00] Sick, B.: *Signalinterpretation mit Neuronalen Netzen unter Nutzung von modellbasiertem Nebenwissen am Beispiel der Verschleißüberwachung von Werkzeugen in CNC-Drehmaschinen*; Nr. 629 in Fortschritt-Berichte VDI, 10: Informatik/Kommunikationstechnik; VDI-Verlag, Düsseldorf, 2000

[Tso98] Tsoi, A. C.: *Recurrent Neural Network Architectures – An Overview*; in: *Adaptive Processing of Sequences and Data Structures* (Hg. Giles, C. L.; Gori, M.); Nr. 1387 in Lecture Notes in Computer Science; Springer Verlag, Berlin, Heidelberg, New York, 1998; S. 1 – 26; (Proceedings of the International Summer School on Neural Networks, Vietri sul Mare, Salerno, 1997)

[UDC+92] Ulbricht, C.; Dorffner, G.; Canu, S.; Guillemyn, D.; Marijuán, G.; Olarte, J.; Rodríguez, C.; Martín, I.: *Mechanisms for Handling Sequences with Neural Networks*; in: *Intelligent Engineering Systems through Artificial Neural Networks* (Hg. Dagli, C. H.; Burke, L. I.; Shin, Y. C.); ASME Press, New York, 1992; Bd. 2; S. 273 – 278; (Proceedings of the 2nd Artificial Neural Networks in Engineering Conference (ANNIE '92), St. Louis)

[VZL97] Voß, H.; Zamzow, T.; Lohmann, R.: *Anwendungsbezogene Optimierung Neuronaler Netze mittels Struktur-Evolution und unvollständiger Induktion*; in: *2. internationaler Workshop Neuronale Netze in Ingenieuranwendungen,* Stuttgart (Hg. Kröplin, B.); 1997; S. 15 – 31

[Wan93a] Wan, E. A.: *Finite Impulse Response Neural Networks with Applications in Time Series Prediction*; Dissertation; Stanford University, Department of Electrical Engineering, 1993

[Wan93b] Wan, E. A.: *Modeling Nonlinear Dynamics with Neural Networks: Examples in Time Series Prediction*; in: *Proceedings of the 5th Workshop on Neural Networks: Academic/Industrial/NASA/Defense (WNN '93 / FNN '93),* San Francisco; 1993; S. 327 – 332

[Wan93c] Wan, E. A.: *Time Series Prediction by Using a Connectionist Network with Internal Delay Lines*; in: *Time Series Prediction: Forecasting the Future and Understanding the Past* (Hg. Weigend, A. S.; Gershenfeld, N. A.); SFI Studies in the Sciences of Complexity; Addison-Wesley Publishing, Reading, 1993; S. 195 – 217

[Wan98] Wan, E. A.: *Control Systems: Classical, Neural, and Fuzzy*; Oregon Graduate Institute, 1998

[WB96] Wan, E. A.; Beaufays, F.: *Diagrammatic Derivation of Gradient Algorithms for Neural Networks*; in: *Neural Computation*; 1996; Bd. 8(1); S. 182 – 201

[WB98] Wan, E. A.; Beaufays, F.: *Diagrammatic Methods for Deriving and Relating Temporal Neural Network Algorithms*; in: *Adaptive Processing of Sequences and Data Structures* (Hg. Giles, C. L.; Gori, M.); Nr. 1387 in Lecture Notes in Computer Science; Springer Verlag, Berlin, Heidelberg, New York, 1998; S. 63 – 98; (Proceedings of the International Summer School on Neural Networks, Vietri sul Mare, Salerno, 1997)

[Wei98] Weiss, F.: *Strukturfindung für Neuronale Time-Delay-Netze mit Hilfe verteilter genetischer Algorithmen*; Diplomarbeit; Universität Passau, Lehrstuhl für Rechnerstrukturen, 1998

[WG94] Wu, D.; Gowdy, J. N.: *Tunable Time Delay Neural Networks for Isolated Word Recognition*; in: *Proceedings of the 1994 International Symposium on Speech, Image Processing, and Neural Networks (ISSIPNN '94),* Hong Kong; 1994; Bd. 1; S. 105 – 108

[WHH+89] Waibel, A.; Hanazawa, T.; Hinton, G.; Shikano, K.; Lang, K. J.: *Phoneme Recognition Using Time-Delay Neural Networks*; in: *IEEE Transactions on Acoustics, Speech, and Signal Processing*; 1989; Bd. 37 (3); S. 328 – 339

[WSVY97] Waibel, A.; Suhm, B.; Vo, M. T.; Yang, J.: *Multimodal Interfaces for Multimedia Information Agents*; in: *Proceedings of the 1997 International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97),* München; 1997; Bd. 1; S. 167 – 170

[Yao93] Yao, X.: *Evolutionary Artificial Neural Networks*; in: *International Journal of Neural Systems*; 1993; Bd. 4 (3); S. 203 – 222

[YB97] Yu, H.-Y.; Bang, S.-Y.: *An Improved Time Series Prediction by Applying the Layer-by-Layer Learning Method to FIR Neural Networks*; in: *Neural Networks*; 1997; Bd. 10 (9); S. 1717 – 1729

[YK98] Yazdizadeh, A.; Khorasani, K.: *Nonlinear System Identification Using Embedded Dynamic Neural Networks*; in: *Proceedings of the 1998 International Joint Conference on Neural Networks (IJCNN '98),* Anchorage; 1998; S. 378 – 383

[Zel94] Zell, A.: *Simulation Neuronaler Netze*; Addison-Wesley Publishing, Bonn, Paris, Reading, 1994