Untersuchungen zur Parallelisierbarkeit von Backpropagation-Netzwerken auf dem CNAPS Neurocomputer

Diplomarbeit

von

SVEN WAHLE

Oktober 1994

an der Universität Karlsruhe (T.H.)
Institut für Logik, Komplexität und Deduktionssysteme

Referent : Prof. Dr. Alex Waibel
Betreuer : Dipl. Inform. Tilo Sloboda

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe erstellt habe. Alle verwendeten Quellen sind im Literaturverzeichnis aufgeführt.

Karlsruhe, den 1. Oktober 1994

Vorwort

Im letzten Jahrzehnt hat das Gebiet der neuronalen Netze eine Renaissance erlebt. Seiner ersten Blüte von Mitte der 40er bis Ende der 60er Jahre wurde durch Minsky und Paperts Buch über die Berechenbarkeitsgrenzen einfacher neuronaler Netzmodelle [MP69] ein abruptes Ende gesetzt. Erst Mitte der 80er Jahre wurde die Theorie wiederentdeckt und durch den Backpropagation-Algorithmus [RHW86] wurden neuronale Netze zu einem Werkzeug mit dessen Hilfe viele Klassifikations-, Wiedererkennungs- und Vorhersageaufgaben auf eine sehr einfache Art und Weise gelöst werden konnten. Neben klassischen Mustererkennungs und Regelungsaufgaben, sowie der Modellierung biologischer Prozesse wurden neuronale Netze in vielen Gebieten, zum Beispiel der Sprachverarbeitung [WL89], der Spieltheorie, der Musik oder den Wirtschaftswissenschaften mit Erfolg eingesetzt. Da der Aufwand des Einlernens neuronaler Netze mit der Anzahl ihrer Neuronen, der Anzahl der synaptischen Verbindungen und der Menge der Lernbeispiele stark ansteigt, sind der aktuellen Forschung Grenzen gesetzt, was die Größe der behandelbaren Probleme anbelangt. Als Lösung für diese Problematik bietet sich der Einsatz von massiv parallelen Rechnern an, da diese, den künstlichen neuronalen Netze ähnlich, aus einer großen Menge einfacher Recheneinheiten bestehen, die über lokale Verbindungen miteinander kommunizieren können, und somit für eine Verwendung prädestiniert sind.

Im Rahmen dieser Arbeit sollen nun zum einen die allgemeinen theoretischen Möglichkeiten der Parallelisierung neuronaler Netze, zum anderen die Eignung eines speziellen Parallelrechners, der CNAPS von Adaptive Solutions, für zwei einfache Klassen von neuronalen Netzen, MLPs und TDNNs, untersucht werden.

Gliederung

Zunächst werden die Grundlagen neuronaler Netze rekapituliert (Kapitel 1). Nach einer Einführung in das Gebiet der Parallelrechner und der Vorstellung der CNAPS (Kapitel 2) werden die Problematiken der Implementierung neuronaler Netze in Festkommaarithmetik (Kapitel 3) erläutert. Danach sollen die Möglichkeiten zur Parallelisierung neuronaler Netze erörtert werden und mittels des daraus hervorgehenden Klassifizierungsschemas einige bisher implementierte parallele Simulatoren für Neuronale Netze bezüglich ihrer Leistungsdaten eingeordnet werden (Kapitel 4).

Anschließend werden die beiden in dieser Arbeit implementierten Simulatoren erläutert, detailliert analysiert und danach mit ihren Leistungsmerkmalen einander gegenübergestellt und verglichen (Kapitel 5). Den Abschluß bilden eine Zusammenfassung der Ergebnisse dieser Arbeit sowie ein Ausblick auf noch offene Fragestellungen, die nicht mehr behandelt werden konnten (Kapitel 6).

Inhaltsverzeichnis

1. Neuronale Netze **1**
1.1 Was ist ein neuronales Netz ? 1
1.2 Definitionen 2
1.3 Der Backpropagation-Algorithmus 3
1.4 Varianten 4
1.4.1 Aktivierungsfunktionen 4
1.4.2 Fehlerfunktionen 4
1.4.3 Gewichtsänderungen 4
1.4.4 Lernverfahren 5
1.5 Einbettung von neuronalen Netzen 5
1.6 Eine alternative Notation 9
1.7 TDNNs 10

2. Parallelrechner **13**
2.1 Einleitung 13
2.2 Grundlagen 13
2.2.1 Klassifikation von Parallelrechnern 13
2.2.2 Maßzahlen für Algorithmen 14
2.3 Der CNAPS Neurocomputer 15
2.3.1 Systemübersicht 15
2.3.2 Aufbau des einzelnen Prozessors 16
2.3.3 Hard- und Software 17
2.3.4 Zusammenfassung 19

3. Berechnung von neuronalen Netzen mit beschränkter Genauigkeit **21**
3.1 Einleitung 21
3.2 Ein neuronales Netz in Festkomma-Darstellung 21
3.3 Darstellungsgenauigkeit für Gewichte 22
3.3.1 Notation 23
3.3.2 Rundungsoperatoren 24
3.3.3 Experimente 28
3.3.4 Auswertung 31
3.3.5 Zusammenfassung 32

4. Möglichkeiten der Parallelisierung neuronaler Netze **33**
4.1 Einleitung 33
4.2 Aufgabenstellungen bei der Berechnung neuronaler Netze 33
4.3 Parallelisierungsmöglichkeiten für neuronale Netze 34
4.3.1 Netzparallelität 34
4.3.2 Mustersatzparallelität 35

4.3.3 Schichtparallelität 36
4.3.4 Neuronenparallelität 37
4.3.5 Gewichtsparallelität 39
4.4 Parallelisierung in realen Systemen 42

5. Zwei parallele Simulatoren für neuronale Netze **45**
5.1 Einleitung 45
5.2 Aufgabenstellung 45
5.2.1 Zielsetzung 45
5.2.2 Eigenschaften der Hardware 46
5.2.3 Verwendbare Parallelitätsebenen 46
5.3 Neuronaler Netz Algorithmus I 47
5.3.1 Kurzübersicht 47
5.3.2 Verteilung der Neuronen 47
5.3.3 Detailerläuterung des Algorithmus 48
5.3.4 Speicherbedarf 51
5.3.5 Laufzeit 51
5.3.6 TDNNs 55
5.3.7 Realisierung des Algorithmus 56
5.4 Neuronaler Netz Algorithmus II 57
5.4.1 Kurzübersicht 57
5.4.2 Verteilung der Neuronen 57
5.4.3 Detailerläuterung des Algorithmus 58
5.4.4 Speicherbedarf 61
5.4.5 Laufzeit 63
5.4.6 TDNNs 68
5.4.7 Realisierung des Algorithmus 68
5.5 Vergleich der beiden Algorithmen 69
5.6 Wertung der Algorithmen 70

6. Resümee **73**
6.1 Ergebnisse 73
6.2 Ausblick 74

1. Neuronale Netze

1.1 Was ist ein neuronales Netz ?

Ein neuronales Netz ist „grob gesprochen" ein als Funktion interpretierbarer gerichteter, gewichteter Graph. Sind die Knoten dieses Graphen, die sogenannten *Neuronen*, in größeren Teilmengen so organisiert, daß der Graph multipartit ist, wird das Netz als geschichtetes neuronales Netz und diese Teilmengen als *Schichten* bezeichnet. Zumeist sind zwei dieser Schichten gegenüber den anderen ausgezeichnet: die *Eingabeschicht* und die *Ausgabeschicht*. Die verbleibenden Schichten werden dann als *versteckte Schichten* bezeichnet. Ein Beispiel für ein solches Netz ist in Abbildung 1.1 dargestellt.

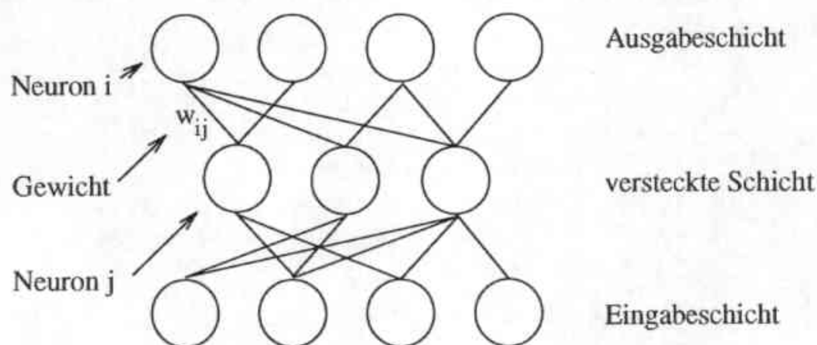


Abbildung 1.1: Ein neuronales Netz

Ein neuronales Netz wird dann in folgender Weise als Funktion betrachtet: Für einen Eingabevektor $\mathbf{i} = (i_1, i_2, \dots i_m)$ wird der Funktionswert des neuronalen Netzes berechnet, indem man die Komponenten als Ausgabewerte der Neuronen der Eingabeschicht betrachtet und dann für die Neuronen der nachfolgenden Schichten sukzessive bis zur Ausgabeschicht deren Ausgabewerte (*Aktivierungen*) berechnet. Die Aktivierungen der Neuronen der Ausgabeschicht bilden dann den Ausgabevektor $\mathbf{o} = (o_1, o_2, \dots o_n)$, den Funktionswert des neuronalen Netzes für den Eingabevektor $\mathbf{i}$. Die Aktivierung der einzelnen Neuronen berechnet sich hierbei, indem man die Summe ihrer mit den Verbindungsgewichten gewichteten Vorgängeraktivierungen und des Schwellwertes (*Bias*, θ) des Neurons bildet und diese dann mittels einer sogenannten *Aktivierungsfunktion* f_{act} auf einen Wert in ein beschränktes Intervall (zumeist [0,1] oder [-1, 1]) abbildet. Um die Darstellung zu vereinheitlichen und Lernalgorithmen kompakt herleiten zu können, wird der Schwellwert oft auch auf folgende Weise dargestellt: Jedem Neuron, das einen Schwellwert besitzt, wird statt diesem ein zusätzliches sogenanntes Schwellwertneuron hinzugefügt, das die Aktivierung konstant Eins hat

und dessen Verbindung mit dem Schwellwert gewichtet ist (Abbildung 1.2)

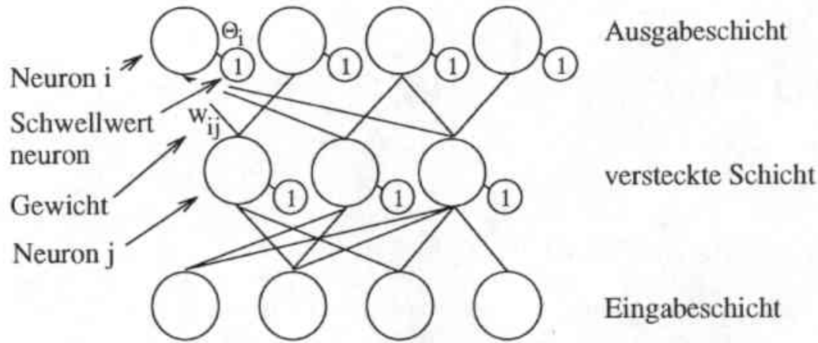


Abbildung 1.2: Ein neuronales Netz mit Schwellwert-Neuronen

1.2 Definitionen

Um im folgenden etwas besser argumentieren zu können, hier nun ein paar Definitionen zur Beschreibung neuronaler Netze:

$\mathcal{P}(i)$	:	Menge aller Vorgänger des Neurons i
$\mathcal{P}^+(i)$	:	Menge aller Vorgänger des Neurons i ohne das Schwellwertneuron
$\mathcal{S}(i)$	:	Menge aller Nachfolger eines Neurons i
f_{act}	:	die Aktivierungsfunktion des Neurons
ω_{ij}	:	das Gewicht auf der Verbindung von Neuron j zu Neuron i
θ_i	:	der Schwellwert des Neurons i

Somit ergibt sich der Eingabewert net_i des Neurons i zu

$$net_i = \sum_{j \in \mathcal{P}^+(i)} \omega_{ij} a_j + \theta_i = \sum_{j \in \mathcal{P}(i)} \omega_{ij} a_j$$

und seine Aktivierung zu

$$a_i = f_{act}(net_i)$$

Dieser Sachverhalt ist noch einmal anschaulich in den Abbildungen 1.3 und 1.4 dargestellt.
Außerdem bezeichnen

$\mathcal{N}$	:	Menge aller Neuronen des Netzes
$\mathcal{N}_\mathcal{I}$	:	die Eingabeschicht
$\mathcal{N}_\mathcal{O}$	:	die Ausgabeschicht
$\mathcal{N}_\mathcal{H}$	:	sämtliche versteckten Neuronen
$\mathcal{N}_{\mathcal{H}_i}$	:	eine versteckte Schicht

Es gilt:

$$\begin{aligned} \mathcal{N} &= \mathcal{N}_\mathcal{I} \uplus \mathcal{N}_\mathcal{H} \uplus \mathcal{N}_\mathcal{O} \\ &= \mathcal{N}_\mathcal{I} \uplus \mathcal{N}_{\mathcal{H}_1} \uplus \ldots \uplus \mathcal{N}_{\mathcal{H}_n} \uplus \mathcal{N}_\mathcal{O} \end{aligned}$$

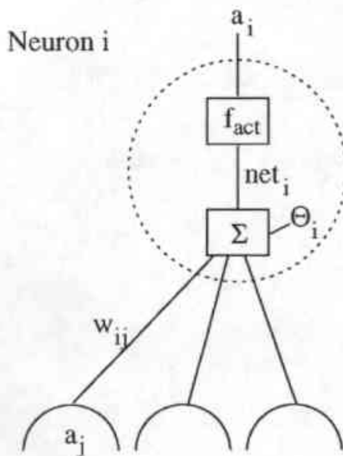


Abbildung 1.3: Ein Neuron mit Schwellwert

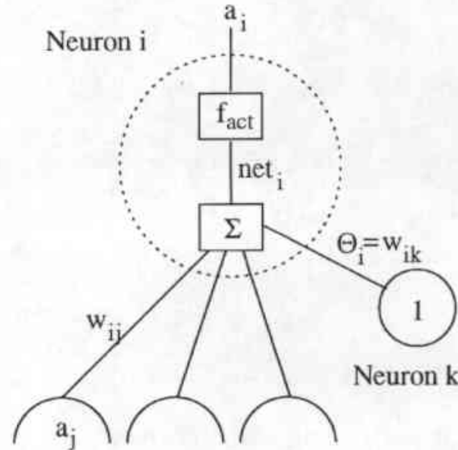


Abbildung 1.4: Ein Neuron mit Schwellwertneuron

$$
\begin{aligned}
|\mathcal{N_I}| &= m \\
|\mathcal{N_O}| &= n
\end{aligned}
$$

1.3 Der Backpropagation-Algorithmus

Da das Netz auf die oben geschilderte Art und Weise eine Funktion von $I\!R^m$ nach $I\!R^n$ darstellt, in der die Gewichte frei wählbare Parameter sind, kann man versuchen diese so zu wählen, daß sich für bestimmte Eingabevektoren für diese Funktion bestimmte Ausgabevektoren an der Ausgabeschicht ergeben. Als Lösungsansatz für diese Aufgabe kann man sich eine *Fehlerfunktion* $E(\mathbf{i}, \mathbf{t})$ definieren, die den Abstand der Ausgabe des Netzes vom gewünschten Ausgabevektor (Sollwert) bestimmt und dann bezüglich dieser Fehlerfunktion für eine *Mustermenge* $\mathcal{M}$, bestehend aus Paaren von Eingabevektoren mit gewünschten Ausgabevektoren, ein Gradientenabstiegsverfahren durchführen. Wählt man jetzt zum Beispiel die Fehlerfunktion $E = MSE$ (*Mean Squared Error*)

$$E(\mathbf{i}, \mathbf{t}) = MSE(\mathbf{i}, \mathbf{t}) = \frac{1}{2}\sum_{i=0}^{n}(t_i - o_i)^2,$$

wobei $\mathbf{i} = (i_1, i_2, \ldots i_m)$ den Eingabevektor, $\mathbf{o} = (o_1, o_2, \ldots o_n)$ den Ausgabevektor des Netzes bei Eingabe von $\mathbf{i}$ und $\mathbf{t} = (t_1, t_2, \ldots t_n)$ den gewünschten Ausgabevektor bezeichnen, und führt man die Abkürzung

$$\delta_i = -\frac{\partial E}{\partial net_i}$$

für den *rückpropagierter Fehler* eines Neurons ein, ergibt sich:

$$
\begin{aligned}
\forall i \in \mathcal{N_O} &: \quad \delta_i = (t_i - o_i) f'_{act}(net_i) \\
\forall i \in \mathcal{N_H} &: \quad \delta_i = (\sum_{k \in \mathcal{S}(i)} \delta_k \omega_{ki}) f'_{act}(net_i)
\end{aligned}
$$

und somit als Änderungswerte (*updates*) für die einzelnen Gewichte des Netzes

$$\Delta\omega_{ij} := \frac{-\partial E}{\partial \omega_{ij}} = \delta_i a_j$$

Auf diese Weise gliedert sich der *Backpropagation*-Algorithmus in 3 wesentliche Teile:

①	Vorwärtsschritt:	Die Berechnung der Aktivierungen aller Neuronen
②	Rückwärtschritt:	Die Berechnung der rückpropagierten Fehler aller Neuronen
③	Änderungsschritt:	Änderung der Gewichte

1.4 Varianten

1.4.1 Aktivierungsfunktionen

Als Aktivierungsfunktionen erfreuen sich folgende Funktionen recht großer Beliebtheit:

$$\begin{aligned} f_{act1} &= sig(x) = \frac{1}{1+e^{-x}} \\ f_{act2} &= tanh(x) = 2\frac{1}{1+e^{-2x-1}} \end{aligned}$$

Hierbei gilt für die Ableitungen:

$$\begin{aligned} sig'(x) &= (1-sig(x))sig(x) \\ tanh'(x) &= 1-tanh(x)^2 \end{aligned}$$

1.4.2 Fehlerfunktionen

Als Alternative zur Standard-Fehlerfunktion MSE gibt es neben vielen anderen Funktionen mit unterschiedlichsten Eigenschaften auch folgende, *cross entropy* genannte, Fehlerfunktion[1]:

$$CE(\mathbf{i},\mathbf{t}) = \sum_{i=0}^{n}((t_i)\log(o_i) + (1-t_i)\log(1-o_i))$$

Hierbei ergeben sich im *Backpropagation*-Algorithmus die Werte wie folgt:

$$\begin{aligned} \forall i \in \mathcal{N}_{\mathcal{O}} &: \delta_i = (t_i - o_i) \\ \forall i \in \mathcal{N}_{\mathcal{H}} &: \delta_i = (\sum_{k\in\mathcal{S}(i)} \delta_k \omega_{ki}) f'_{act}(net_i) \end{aligned}$$

1.4.3 Gewichtsänderungen

Eine Möglichkeit ein Netz einzulernen besteht darin, ihm die Muster $(\mathbf{i},\mathbf{t})$ aus $\mathcal{M}$ nacheinander zu präsentieren, für jedes Muster jeweils einen Gradientenabstiegsschritt durchzuführen und zu hoffen, daß das Netz nach einigen Epochen einen Zustand angenommen hat, in dem es alle Eingabevektoren aus $\mathcal{M}$ auf die jeweils gewünschten Ausgabevektoren abbildet. Als Alternative zu dieser heuristischen Methodik läßt sich die Fehlerfunktion aber auch statt auf ein einzelnes Muster auf einen Teil der Mustermenge oder die komplette Mustermenge definieren und dementsprechend das Gradientenabstiegsverfahren durchführen.
Es seien $\Delta^{(\mathbf{i},\mathbf{t})}\omega_{ij}$ die für das einzelne Muster $(\mathbf{i},\mathbf{t})$ berechnete Gewichtsänderung und $\mathcal{B}$ eine Teilmenge der Mustermenge $\mathcal{M}$. Dann lassen sich die folgenden drei Vorgehensweisen unterscheiden:

[1] Dies ist die Formulierung für $f_{act}: \boldsymbol{R} \rightarrow [0,1]$. Für andere f_{act} existieren andere, wirkungsäquivalente Ausprägungen dieser Formel.

- *Musterlernen (learning by pattern, pattern update)*

  Fehlerfunktion: $E(\mathbf{i}, \mathbf{t})$

  Gewichtsänderungswert: $\Delta\omega_{ij} = \Delta^{(\mathbf{i},\mathbf{t})}\omega_{ij} = \delta_i a_j$

- *Blocklernen (batch update)*

  Fehlerfunktion: $\sum_{(\mathbf{i},\mathbf{t})\in\mathcal{B}} E(\mathbf{i}, \mathbf{t})$

  Gewichtsänderungswert: $\Delta\omega_{ij} = \sum_{(\mathbf{i},\mathbf{t})\in\mathcal{B}} \Delta^{(\mathbf{i},\mathbf{t})}\omega_{ij}$

- *Epochenlernen (learning by epoch, epoch update)*

  Fehlerfunktion: $\sum_{(\mathbf{i},\mathbf{t})\in\mathcal{I}} E(\mathbf{i}, \mathbf{t})$

  Gewichtsänderungswert: $\Delta\omega_{ij} = \sum_{(\mathbf{i},\mathbf{t})\in\mathcal{M}} \Delta^{(\mathbf{i},\mathbf{t})}\omega_{ij}$

1.4.4 Lernverfahren

Neben der mathematisch korrekten Vorgehensweise, dem Addieren des in Abschnitt 1.3 hergeleiteten Gewichtsänderungs-Wertes, gibt es einen ganzen Zoo an verschiedenen Möglichkeiten und Verfahren die Gewichtsänderung für eine einzelne Verbindung vorzunehmen. Diese Verfahren versuchen mittels einfacher Berechnungen Insuffizienzen des Standard-*Backpropagation*-Algorithmus zu beseitigen. Die bekanntesten hiervon sind:

- gemeinsames Skalieren aller Gewichtsänderungen (Lernrate)
- Wiederverwendung des Gradienten (*gradient reuse*) [HSH91]
- Quickprop [Fah88]
- Rprop [Rie92]
- Verwendung eines Impuls-Terms (*momentum term*) [PNH86]

Für die letzten drei Verfahren muß neben dem Gewicht noch mindestens ein anderer Wert pro Verbindung gespeichert werden. All diesen Verfahren gemein ist, daß sie wie der *Backpropagation*-Algorithmus die lokale Berechnungseigenschaft besitzen, d.h., daß sie die Aktivierung und den rückpropagierten Fehler eines Neurons respektive die Gewichtsänderung einer Verbindung aus den Werten der adjazenten Neuronen/Verbindungen berechnen.

1.5 Einbettung von neuronalen Netzen

Für eine Berechnung geschichteter Neuronaler Netze auf Parallelrechnern ist es häufig von Vorteil, bestimmte Schichtgrößen zu haben, die sich ohne „Restneuronen" auf die einzelnen Prozessoren verteilen lassen. Im folgenden wird daher die Einbettung neuronaler Netze in größere betrachtet und wie man dabei die gleichen Aktivierungen und rückpropagierten Fehler für die eingebetteten Neuronen, und die gleichen Gewichtsänderungen für die eingebetteten Gewichte erhalten kann, die sich für das ursprüngliche Netz ergeben hätten.

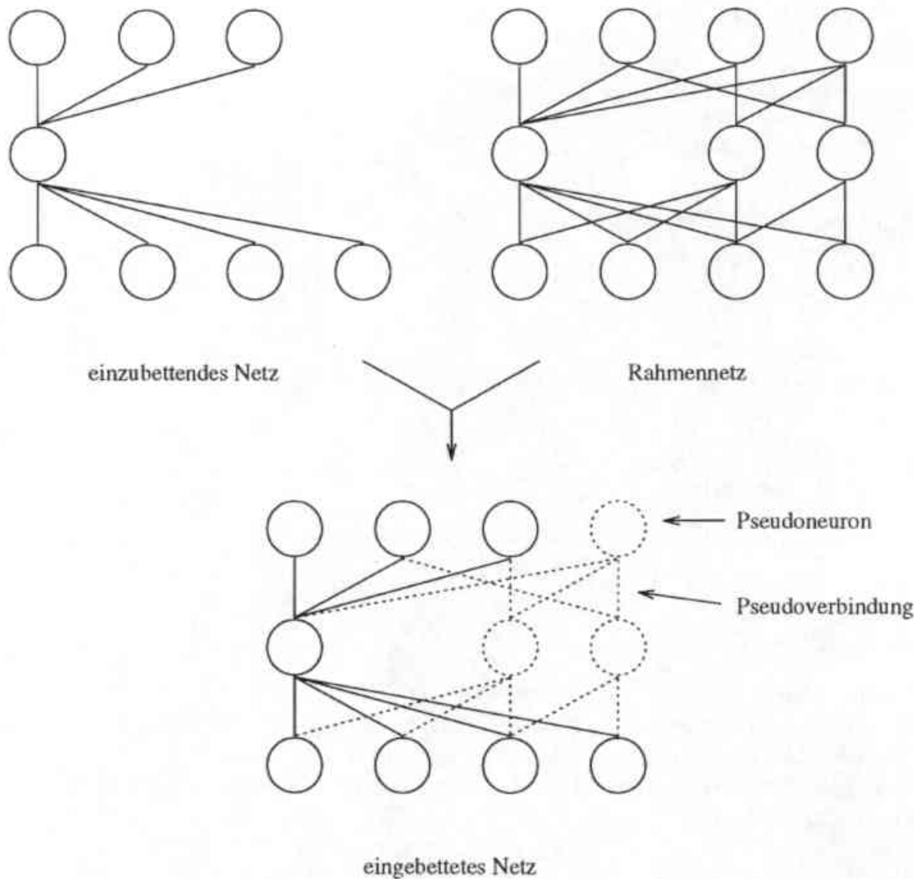


Abbildung 1.5: Ein eingebettetes neuronales Netz

Nehmen wir beispielweise an, das ursprüngliche Netz hätte den Aufbau aus Abbildung 1.5 und werde in eine größere Netzstruktur (im folgenden *Rahmennetz* genannt) eingebettet, indem sogenannte *Pseudoneuronen* und *Pseudoverbindungen* unter der Beachtung folgender 4 Regeln hinzufügt werden:

- Die Pseudoverbindungen seien mit mindestens einem Pseudoneuron verbunden.
- Die Pseudoverbindungen werden mit Null gewichtet
- Die Aktivierungsfunktionen der Pseudoneuronen werden folgendermaßen definiert:

$$f_{act} \equiv 0$$

  womit dann auch

$$f'_{act} \equiv 0$$

- Die Muster aus $\mathcal{M}$ werden entsprechend der Längen der neuen Eingabe- und Ausgabeschicht umdefiniert:

$$(\mathbf{i}, \mathbf{t}) = ((i_1, \ldots, i_m)(i_1, \ldots, i_n)) \in \mathcal{M} \Rightarrow$$

$$(\mathbf{i}_{neu}, \mathbf{t}_{neu}) = ((i_1, \ldots, i_m, 0, \ldots, 0), (t_1, \ldots, t_n, 0, \ldots, 0)) \in \mathcal{M}_{neu}$$

Im folgenden wird nun gezeigt, daß diese vier Forderungen genügen, um für die alten Neuronen und Gewichte die gleichen Ergebnisse im *Backpropagation*-Algorithmus wie bei einer Nichteinbettung zu garantieren.
Es bezeichnen:

$\mathcal{N}^{alt}$	:	die Menge der Neuronen im alten Netz
$\mathcal{N}^{pseudo}$	:	die Menge der Pseudoneuronen
$\mathcal{N}^{neu}$	:	die Menge der Neuronen im neuen, erweiterten Netz
$\mathcal{E}^{alt}$	:	die Menge der Verbindungen im alten Netz
$\mathcal{E}^{pseudo}$	:	die Menge der Pseudoverbindungen
$\mathcal{E}^{neu}$	:	die Menge der Verbindungen im neuen, erweiterten Netz
$\mathcal{P}^{alt}(i)$	:	die Menge der Vorgänger des Neurons i vor der Netzerweiterung
$\mathcal{P}^{neu}(i)$	:	die Menge der Vorgänger des Neurons i im neuen, erweiterten Netz

$\mathcal{S}_i$, net_i, a_i und δ_i seien ebenfalls entsprechend benannt.

Hierbei gilt

$$\mathcal{N}^{pseudo} \uplus \mathcal{N}^{alt} = \mathcal{N}^{neu}$$
$$\mathcal{E}^{pseudo} \uplus \mathcal{E}^{alt} = \mathcal{E}^{neu}$$
$$\mathcal{P}^{alt}(i) \subset \mathcal{P}^{neu}(i)$$
$$\mathcal{S}^{alt}(i) \subset \mathcal{S}^{neu}(i)$$
$$\forall e_{ij} \in \mathcal{E}^{pseudo} : i \in \mathcal{N}^{pseudo} \vee j \in \mathcal{N}^{pseudo}$$

Mit diesen Definitionen gilt:
$\forall i \in \mathcal{N}^{alt}$:

$$\begin{aligned} net_i^{neu} &= \sum_{j \in \mathcal{P}^{neu}(i)} \omega_{ij} a_j \\ &= \sum_{j \in \mathcal{P}^{alt}(i)} \omega_{ij} a_j + \sum_{j \in \mathcal{P}^{neu}(i) \setminus \mathcal{P}^{alt}(i)} \omega_{ij} a_j \\ &= \sum_{j \in \mathcal{P}^{alt}(i)} \omega_{ij} a_j + \sum_{j \in \mathcal{P}^{neu}(i) \setminus \mathcal{P}^{alt}(i)} 0 * a_j \\ &= \sum_{j \in \mathcal{P}^{alt}(i)} \omega_{ij} a_j \\ &= net_i^{alt} \end{aligned}$$

und somit

$$\begin{aligned} a_i^{neu} &= a_i^{alt} & \forall i \in \mathcal{N}^{alt} \\ a_i^{neu} &= 0 & \forall i \in \mathcal{N}^{pseudo} \end{aligned}$$

Mit der obigen Definition der neuen Ausgabevektoren gilt dann für die Ausgabeneuronen des neuen Netzes:
$\forall i \in \mathcal{N}_O^{alt}$:

$$\begin{aligned} \delta_i^{neu} &= (t_i - o_i) * f'_{act}(net_i) \\ &= \delta_i^{alt} \end{aligned}$$

und
$\forall i \in \mathcal{N}_{\mathcal{O}}^{pseudo}$:

$$\begin{aligned}\delta_i^{neu} &= (t_i - o_i) * f'_{act}(net_i) \\ &= 0 * 0 \\ &= 0\end{aligned}$$

sowie für alle versteckten Neuronen des neuen Netzes:
$\forall i \in \mathcal{N}_{\mathcal{H}}^{alt}$:

$$\begin{aligned}\delta_i^{neu} &= (\sum_{k \in \mathcal{S}^{neu}(i)} \delta_k \omega_{ki}) f'_{act}(net_i) \\ &= (\sum_{k \in \mathcal{S}^{alt}(i)} \delta_k \omega_{ki} + \sum_{k \in \mathcal{S}^{neu}(i) \setminus \mathcal{S}^{alt}(i)} \delta_k \omega_{ki}) f'_{act}(net_i) \\ &= (\sum_{k \in \mathcal{S}^{alt}(i)} \delta_k \omega_{ki} + \sum_{k \in \mathcal{S}^{neu}(i) \setminus \mathcal{S}^{alt}(i)} \delta_k * 0) f'_{act}(net_i) \\ &= (\sum_{k \in \mathcal{S}^{alt}(i)} \delta_k \omega_{ki}) f'_{act}(net_i) \\ &= \delta_i^{alt}\end{aligned}$$

und
$\forall i \in \mathcal{N}_{\mathcal{H}}^{pseudo}$:

$$\begin{aligned}\delta_i^{neu} &= (\sum_{k \in \mathcal{S}^{neu}(i)} \delta_k \omega_{ki}) f'_{act}(net_i) \\ &= (\sum_{k \in \mathcal{S}^{neu}(i)} \delta_k 0) * 0 \\ &= 0\end{aligned}$$

Somit ergibt sich im Änderungsschritt für die Gewichte des neuen Netzes folgendes:
$\forall e_{ij} \in \mathcal{E}^{alt}$:

$$\begin{aligned}\Delta^{neu}\omega_{ij} &= \delta_i^{neu} a_j^{neu} \\ &= \delta_i^{alt} a_j^{alt} \\ &= \Delta^{alt}\omega_{ij}\end{aligned}$$

und
$\forall e_{ij} \in \mathcal{E}^{pseudo}$:

$$\begin{aligned}\Delta^{neu}\omega_{ij} &= \delta_i^{neu} a_j^{neu} \\ &= \begin{cases} 0 * a_j^{alt} & j \in \mathcal{N}^{alt}, i \in \mathcal{N}^{pseudo} \\ \delta_i^{alt} * 0 & j \in \mathcal{N}^{pseudo}, i \in \mathcal{N}^{alt} \\ 0 * 0 & j \in \mathcal{N}^{pseudo}, i \in \mathcal{N}^{pseudo} \end{cases} \\ &= 0\end{aligned}$$

Das heißt, daß die Verbindungen des eingebetteten alten Netzes nach einer Musterpräsentation dieselben Werte wie im nichteingebetteten Netz haben und, daß auf den Pseudoverbindungen des neuen Netzes die Null als Gewicht erhalten geblieben ist. Somit kann das Training eines neuronalen Netzes unter Erhalt sämtlicher Operationen auch auf einem eingebetteten neuronalen Netz vorgenommen werden, indem man einfach:

- keine Pseudoverbindungen zwischen schon existierenden Neuronen einfügt
- die Gewichte auf den Pseudoverbindungen auf Null setzt
- die Aktivierungsfunktionen der Pseudoneuronen zu Null setzt
- die Ein- und Ausgabevektoren mit Nullen auf die neue Ein- und Ausgabeschichtlängen erweitert

Die hier gestellten Bedingungen können, wie aus den Formeln ersichtlich, sogar noch etwas relaxiert werden, indem man für die Schwellwert-Neuronen der Pseudo-Neuronen die Forderung $f_{act} \equiv 0$ streicht und wie im übrigen Netz $f_{act} \equiv 1$ einsetzt. Da diese Schwellwert-Neuronen als einzigen Nachfolger ein Pseudo-Neuron haben ergibt sich als Änderungswert für den Schwellwert:

$$\begin{aligned} \Delta^{neu}\omega_{ij} &= \delta_i a_j \\ &= 0 * 1 \\ &= 0 \end{aligned}$$

1.6 Eine alternative Notation

Die Definition eines neuronalen Netzes als gewichteter Graph ist allgemein und für eine große Klasse von neuronalen Netzen anwendbar. Eine Implementation eines neuronalen Netzes gemäß dieser Definition würde allerdings auch die Darstellung als gewichteten Graphen im Rechner bedeuten. Für die meistverwendete Klasse von neuronalen Netzen, die mehrfach geschichteten neuronalen Netze (*Multi Layer Perceptron*, MLP), ist es jedoch möglich die durch das Netz gebildete Funktion durch einfache Vektoralgebra zu beschreiben.
Seien

l	:	die Anzahl der Schichten
$\mathbf{s}_k$	:	der Vektor der Schwellwerte der Neuronen der Schicht k ($k = 1 \ldots l-1$)
$\mathbf{i}$	:	die Aktivierungen der Eingangsschicht (Schicht 0)
$\mathbf{h}_k$	:	die Aktivierungen der k-ten versteckten Schicht ($k = 1 \ldots l-2$)
$\mathbf{o}$	:	die Aktivierungen der Ausgabeschicht (Schicht l)
$\mathbf{W}_k$	:	Die Matrix der Verbindungen zwischen Schicht $k-1$ und Schicht k

Dann ergibt sich der Funktionswert eines MLP zu:

$$\begin{aligned} \mathbf{h}_1 &= f_{act}(\mathbf{W}_1\mathbf{i} + \mathbf{s}_1) \\ \mathbf{h}_k &= f_{act}(\mathbf{W}_k\mathbf{h}_{k-1} + \mathbf{s}_k) \quad k = 2 \ldots l-2 \\ \mathbf{o} &= f_{act}(\mathbf{W}_{l-1}\mathbf{h}_{l-1} + \mathbf{s}_{l-1}) \end{aligned}$$

1.7 TDNNs

Time-Delay Neural Nets (TDNNs) [WHH⁺89] sind eine spezielle Art von neuronalen Netzen, die entwickelt wurden, um zeitvariante Signale, z.B. Sprachsignale oder Bildfolgen, auszuwerten. Anhand eines praktischen Beispiels, dem Auswerten eines Sprachsignals durch Abbildung einer Folge von Frequenzvektoren auf Subphoneme, soll nun das TDNN erklärt werden.

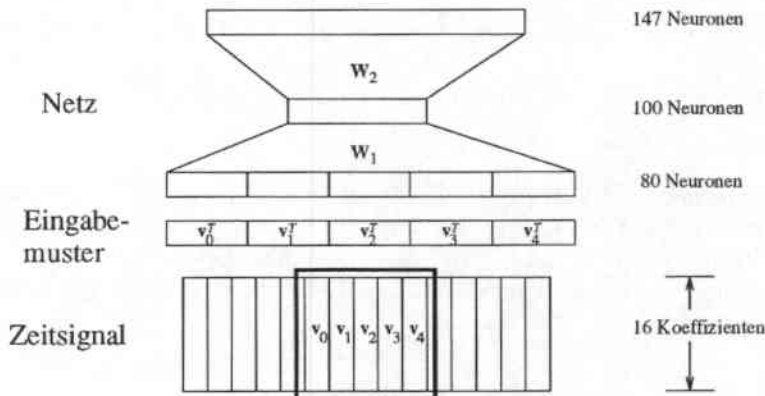


Abbildung 1.6: Ein MLP zur Subphonemerkennung

Ein einfacher Lösungsansatz für diese Aufgabe mittels eines geschichteten neuronalen Netzes ist in Abbildung 1.6 dargestellt. Das auszuwertende Sprachsignal wurde hierbei alle 10ms digitalisiert und in 16 Koeffizienten, die jeweils einen bestimmten Frequenzbereich repräsentieren (*Melscale-*Koeffizienten), zerlegt. Der Eingabevektor für das vollverbundene Netz sind dann fünf aufeinanderfolgende Frequenzvektoren ($\Rightarrow 5*16 = 80$ Eingabeneuronen), die als Eingabefenster $\mathbf{f} = (\mathbf{v}_0^T \ldots \mathbf{v}_4^T)$ bezeichnet werden sollen. Hiermit soll dem Netz ermöglicht werden auch den Verlauf der Frequenzen über die Zeit hinweg bewerten zu können. Die versteckte Schicht wurde hier mit 100 und die Ausgabeschicht mit 147 Neuronen ($49 * 3$ = Anzahl Phoneme $*$ Anzahl Subphoneme pro Phonem) verwirklicht. Die Gewichtsmatrizen im Bild sind mit $\mathbf{W}_1$ (Gewichte zwischen Eingabeschicht und versteckter Schicht) und $\mathbf{W}_2$ (Gewichte zwischen versteckter Schicht und Ausgabeschicht) bezeichnet.
Dieses MLP berechnet sich also folgendermaßen:

$$\begin{aligned}\mathbf{h}_1 &= f_{act}(\mathbf{W}_1\mathbf{f} + \mathbf{s}_1)\\ \mathbf{o} &= f_{act}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{s}_2)\end{aligned}$$

Der Lösungsansatz mittels eines TDNNs (siehe Abbildung 1.7) mit beispielsweise drei *Time-Delays* besteht darin, daß im Prinzip die gleiche Netzstruktur verwendet wird, aber statt einer drei versteckte Schichten existieren, in die als Eingabe $\mathbf{f}_{TD=-1} = (\mathbf{v}_{-1}^T \ldots \mathbf{v}_3^T)$, $\mathbf{f}_{TD=0} = (\mathbf{v}_0^T \ldots \mathbf{v}_4^T)$ und $\mathbf{f}_{TD=1} = (\mathbf{v}_1^T \ldots \mathbf{v}_5^T)$ mittels Gewichts-Mehrfachverwendung (*Weight Sharing*) eingespeist werden. Hierbei werden für die Gewichtung der Eingabeaktivierungen bis zur jeweiligen versteckten Schicht für alle drei Eingabeschichten die gleiche Gewichtsmatrix (in der Abbildung die Gewichtsmatrix $\mathbf{W}_1$) und die gleichen Schwellwerte verwendet. Durch dieses Vorgehen soll eine zeitliche Invarianz der Gewichte aus $\mathbf{W}_1$ erreicht werden.

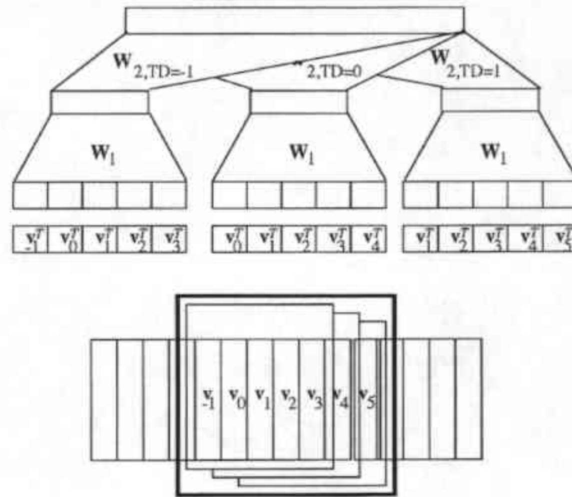


Abbildung 1.7: Ein TDNN mit 3 Time-Delays zur Phonemerkennung

Dieses spezielle TDNN mit 3 Time-Delays berechnet sich also folgendermaßen:

$$\begin{aligned} \mathbf{h}_{1,TD=k} &= f_{act}(\mathbf{W}_1 \mathbf{f}_{TD=k} + \mathbf{s}_1) \qquad (k \in \{-1,0,1\}) \\ \mathbf{o} &= f_{act}\Big(\sum_{k \in \{-1,0,1\}} \mathbf{W}_{2,TD=k} \mathbf{h}_{1,TD=k} + \mathbf{s}_2\Big) \end{aligned}$$

Hierbei bezeichnen $\mathbf{h}_{1,TD=-1}$, $\mathbf{h}_{1,TD=0}$ und $\mathbf{h}_{1,TD=1}$ die Aktivierungsvektoren der jeweiligen versteckten Schicht.
Bei der Berechnung der Gewichtsänderung für die hierbei mehrfach verwendeten Gewichte in $\mathbf{W}_1$ werden einfach die Gewichtsänderungen, die für die jeweiligen Verwendungen entstanden wären, aufsummiert und dann auf das Gewicht addiert. Somit ergibt sich als Änderungswert für das solchermaßen dreimal verwendete Gewicht ω_{ij} zwischen den Neuronen $j_k (k \in \{-1,0,1\})$ der versteckten Schichten -1, 0, 1 und den Neuronen $i_k (k \in \{-1,0,1\})$ der Eingabeschichten -1, 0 und 1:

$$\Delta\omega_{ij} = \sum_{k \in \{-1,0,1\}} \delta_{i_k} a_{j_k}$$

Der Aufbau, die Berechnung und das Training von TDNNs mit einer anderen Anzahl an Time-Delays erfolgt analog.

2. Parallelrechner

2.1 Einleitung

In diesem Kapitel werden zunächst einige für die weiteren Betrachtungen notwendige Begriffe aus dem Bereich Parallelrechner erläutert. Zum einen soll hier auf die Klassifikation von Parallelrechnern eingegangen werden, zum anderen sollen die zur Beurteilung von parallelen Algorithmen notwendigen Begriffe der Beschleunigung und Effizienz erläutert werden. Danach soll der Parallelrechner auf dem diese Arbeit durchgeführt wurde, die CNAPS von Adaptive Solutions, vorgestellt werden. Bei der Erläuterung der Architektur dieses SIMD-Neurocomputer wird insbesondere auch auf die für die Implementierung von neuronalen Netzen wichtigen Details eingegangen. Zum Abschluß sollen dann noch einmal die wichtigsten Daten der CNAPS zusammengefaßt werden.

2.2 Grundlagen

2.2.1 Klassifikation von Parallelrechnern

Nach Flynn [Fly72] lassen sich Rechner bezüglich der Parallelität bei der Verarbeitung von Befehls- und Datenstrom in 4 Klassen einteilen. Dieses sind:

- *SISD* (*Single Instruction, Single Data*)
- *MISD* (*Multiple Instruction, Single Data*)
- *SIMD* (*Single Instruction, Multiple Data*)
- *MIMD* (*Multiple Instruction, Multiple Data*)

Alle sequentiellen Rechner fallen hierbei in die Klasse SISD, die Klasse MISD wird meist als leer angesehen. Die Gruppe der Parallelrechner besteht dann einerseits aus Multiprozessorsystemen (MIMD-Rechner), in denen in vielen Prozessoren gleichzeitig verschiedene Daten mittels verschiedener Befehle bearbeitet werden können und der Klasse der Feld- und Vektorrechner (SIMD-Rechner), die nur über einen einzigen Befehlstrom verfügen und damit viele Daten gleichzeitig bearbeiten[1]. *Feldrechner* bestehen im allgemeinen aus einer Steuereinheit, die die Befehle des Befehlsstroms decodiert und einer großen Anzahl (typischerweise 2^8 bis 2^{16}) sehr einfacher Datenverarbeitungseinheiten. Meistens sind diese Einheiten mittels eines einfachen Netzes (Matrix, x-Netz, Torus, Hyperkubus) verbunden, über das zeitgleich Daten ausgetauscht werden können. Der Vorteil solcher Architekturen gegenüber MIMD-Rechnern besteht zum einen darin, daß die Herstellungskosten bei gleicher Menge an Prozessoren niedriger sind, da die einzelnen Datenverarbeitungseinheiten keine Decodierlogik enthalten und folglich nur aus einer arithmetisch-logischen Einheit mit ein wenig Speicher bestehen. Zum anderen ist bei einer Kommunikation respektive

[1] Dies ist eine extrem vergröberte Sicht der Dinge. Es gibt einige parallele Architekturen (Vektorrechner, systolische Felder) die in Grenzbereiche fallen und je nach Belieben und Auslegung der obigen Definition von verschiedenen Autoren in verschiedene Klassen eingeordnet werden. Die hier gemachten Angaben beziehen sich auf [Ung93].

einem Datenaustausch keine zusätzliche Synchronisation der Prozessoren notwendig, da die Prozessoren schon über den Befehlsstrom synchronisiert sind. Hierdurch ist mit Feldrechnern eine sehr feingranulare Bearbeitung von Problemen mit Parallelisierung auf Suboperationsebene möglich, während auf MIMD-Rechnern nur eine Parallelisierung auf Block-, oder höherer Ebene möglich ist [Ung93].

2.2.2 Maßzahlen für Algorithmen

Um parallele Algorithmen bezüglich ihrer Leistungsfähigkeit zu beurteilen, gibt es neben der normalen Laufzeitabschätzung noch die Begriffe der Beschleunigung und der Effizienz. Es bezeichnen:

T_1 : Die Laufzeit zur Lösung eines gegebenen Problems auf einem sequentiellen Rechner

T_n : Die Laufzeit zur Lösung eines gegebenen Problems auf einem parallelen Rechner mit n Prozessoren

Dann sind die *Beschleunigung* S_n und die *Effizienz* E_n definiert durch:

$$\begin{aligned} S_n &:= \frac{T_1}{T_n} \\ E_n &:= \frac{S_n}{n} \end{aligned}$$

Die Beschleunigung gibt an, inwieweit die Lösung eines Problems durch ein paralleles System mit einer gegebenen Menge an Prozessoren beschleunigt werden konnte. Die höchstmögliche Beschleunigung beim Einsatz von n Prozessoren und Betrachtung im O-Kalkül beträgt $S_n = n$ ([2]). Den Bezug zwischen Höhe der erreichten Beschleunigung und der Menge der eingesetzten Prozessoren stellt die Größe E_n her. Hierdurch wird der Ausnutzungsgrad der einzelnen Prozessoren bezüglich des zu lösenden Problems angegeben. $E_n = 1$ bedeutet eine optimale Ausnutzung aller Prozessoren und kann nur für $T_n = n$ erreicht werden.
Da ein Algorithmus meistens nicht vollständig parallelisierbar ist und noch Anteile sequentiellen Codes enthält, läßt sich unter der Annahme eines sequentiellen Anteils f für obige Definitionen das sogenannte *Amdahlsche Gesetz* herleiten:

$$\begin{aligned} T_n &\geq fT_1 + \frac{(1-f)T_1}{n} \\ S_n &\leq \frac{T_1}{fT_1 + \frac{(1-f)T_1}{n}} \\ &\leq \frac{1}{f} \end{aligned}$$

Durch diese Formeln werden die bei sequentiellem Anteil f schnellstmögliche Laufzeit und höchstmögliche Beschleunigung eines Algorithmus beschrieben. An der Abschätzung für die Beschleunigung ist zu erkennnen, daß schon bei einem sequentiellen Anteil von 10% selbst mit unendlich vielen Prozessoren höchstens eine Beschleunigung um den Faktor 10 zu erreichen ist. Hieraus ergibt sich unter anderem, daß sequentieller Code in Parallelrechnersystemen möglichst von dedizierten,

[2] Von einigen Autoren wird in diesem Zusammenhang der Begriff der superlinearen Beschleunigung für $S_n > n$ verwendet. Diese Beschleunigungen sind aber nur auf implementierungstechnischer Basis aufgrund synergetischer Effekte zu messen.

schnellen sequentiellen Recheneinheiten ausgeführt werden sollte, da dieser sonst bei schon kleinen Anteilen einen Flaschenhals für die Beschleunigung von Algorithmen darstellt. Durch Verdopplung oder Verdreifachung der Geschwindigkeit des sequentiellen Prozessors kann so die höchstmögliche Beschleunigung zu recht geringen Kosten im Vergleich zu einem Gesamtsystem verdoppelt oder verdreifacht werden.

2.3 Der CNAPS Neurocomputer

Die CNAPS ist ein Rechner, der zur Klasse der Feldrechner gehört und speziell für die Berechnung neuronaler Netze konzipiert wurde. In den folgenden Abschnitten wird neben der allgemeinen Beschreibung auch auf einige Details des Rechners eingegangen werden, da diese für die später zu beschreibenden Implementierungen eine maßgebliche Rolle spielen.

2.3.1 Systemübersicht

Der grobe Aufbau der CNAPS-Servers ist in Abbildung 2.1 dargestellt. Die Hauptkomponenten des Systems sind:

- Der Parallel-Rechner
  - Prozessorvektor
  - Sequencer (CSC)
  - Programmspeicher
  - Dateispeicher
- Der Frontrechner
  - Kontrolleinheit (CP)
  - Kontrolleinheits-Speicher (CP-Speicher)

Die CNAPS besteht also im wesentlichen aus zwei Teilen: Einem normalen sequentiellen Frontrechner nach dem von-Neumann-Prinzip, sowie dem Parallelrechner, mit allen Prozessoren gemeinsamen Programmspeicher und Dateispeicher. Im folgenden sollen kurz die Aufgaben und das Zusammenwirken der einzelnen aktiven Einheiten beschrieben werden:

Prozessorvektor: Das Prozessorfeld besteht aus bis zu 512 einzelnen Prozessoren, die zeitgleich die Befehle ausführen, die sie über den Befehlsbus aus der Steuereinheit (CSC, Sequencer) erhalten. Die Prozessoren sind untereinander mittels eines 8 Bit breiten IN-, eines 8 Bit breiten OUT-Busses, sowie einer bidirektionalen 2 Bit Nachbarschafts-Kommunikationsstruktur (Ring) verbunden.

Sequencer (CSC): Der Sequencer übernimmt zum einen die Steuerung des Kontrollflusses, zum anderen lenkt er die Kommunikation des Prozessorfeldes mit dem restlichen System. Jeder der aus dem 512 KB großen Programmspeicher gelesenen 64-Bit Befehle wird in 2 Hälften geteilt. Die letzten 32 Bit werden direkt über den Befehlsbus an das Prozessorfeld geschickt, die anderen 32 Bit verbleiben im Sequencer und steuern dort den Befehlszähler, die Schleifenzähler, das interne Rechenwerk und die Anbindung des IN und OUT-Busses an den 32 MB Dateispeicher.

Kontrolleinheit (CP): Der Frontrechner stellt die Anbindung des aus CSC, Prozessorfeld, Dateispeicher und Programmspeicher bestehenden Parallelrechners an das Ethernet her. Er besteht aus einer VME-Bus Karte mit 68030 Prozessor mit 4MB RAM auf dem als Betriebssystem

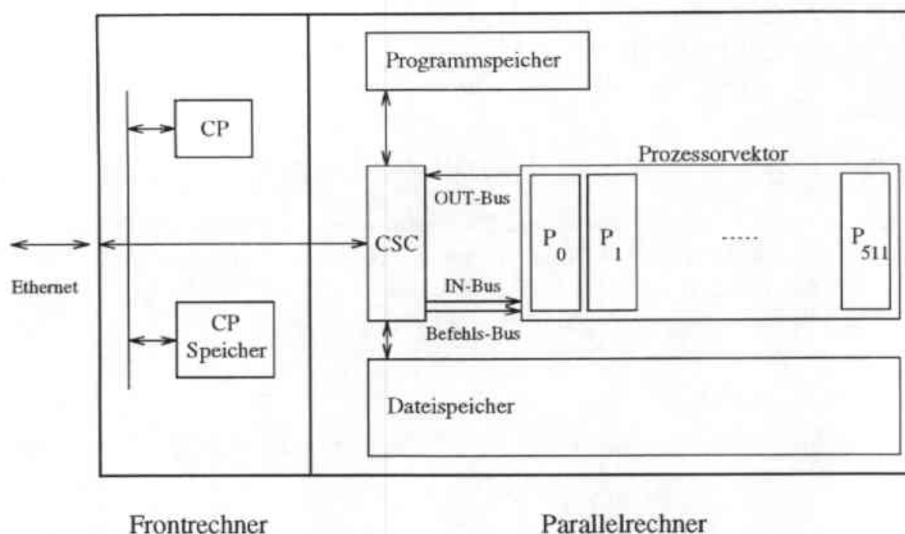


Abbildung 2.1: Übersichtszeichnung des CNAPS-Neurorechners

VXWorks läuft. Durch Software, die auf der Workstation läuft, kann dieser Rechner in der CNAPS angewiesen werden, Dateien (z.B. große Gewichtsdateien) aus dem normalen UNIX-Dateisystem zur besseren und schnelleren Zugreifbarkeit in den Dateispeicher zu bringen oder aus dem Dateispeicher dorthin zurückzuschreiben. Auf dieselbe Art und Weise werden auch vom Parallelrechner auszuführende Programme in den Programmspeicher der parallelen Einheit transferiert und gestartet.

2.3.2 Aufbau des einzelnen Prozessors

Da der einzelne Prozessor nur als Recheneinheit benutzt wird, sind in ihm keine Einheiten zur Steuerung des Kontrollflusses vorhanden. Die einzige Möglichkeit in Abhängigkeit der von ihm berechneten Daten zu agieren (oder besser: nicht zu agieren) besteht darin, aufgrund eines Aktivierungsbits bei als konditional gekennzeichneten Befehlen „nicht zuzuhören". Dieses Bit kann aufgrund eines Testergebnisses gesetzt, gelöscht oder mit ihm über normale Logikoperationen verknüpft werden, so daß auch mehrfach geschachtelte if-Anweisungen in Abhängigkeit von lokalen Daten für die einzelnen Prozessoren möglich sind.
Der einzelne Prozessor besteht im wesentlichen aus:

A- und B-Bus: Die beiden internen Busse sind 16 Bit breit und dienen dazu die Daten zwischen den verschiedenen Recheneinheiten zu transportieren. Über sie können zeitgleich die beiden Eingänge von Addierer, Multiplizierer oder der Logikeinheit beschickt werden.

Registersatz: 32 16-Bit Datenregister, ca. 20 zur Verwendung für den Benutzer frei

Adreßregister: 1 (!) 12-Bit Adreßregister. Hierfür existieren noch ein Keller der Größe 1 zum Abspeichern und Wechseln einer zweiten Adresse, sowie ein Offsetregister, das angibt in welchen Schritten das Adreßregister beim Inkrementierungsbefehl erhöht werden soll.

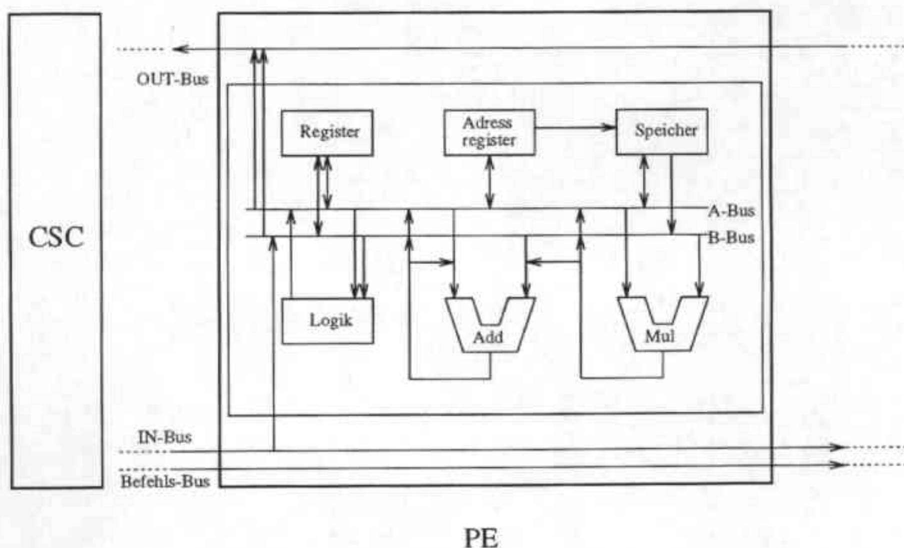


Abbildung 2.2: Übersicht über einen einzelnen Prozessor

Speicher: 4 KB Speicher, byte- oder wortweise adressierbar. Zusätzlich existiert eine Speziallogik mittels der spärlich besetzte Matrizen, die gewisse Regelmäßigkeiten aufweisen, platzsparend im Speicher gehalten werden können, aber auf die wie auf ein normales Feld zugegriffen werden kann.

Logikeinheit: logische 16 Bit Verknüpfungen, 16 Bit Schieberegister, Test des Logikausgangs

Addierer: 32-Bit Addierer/Subtrahierer für vorzeichenbehaftete Ganzzahlen. Da die internen Busse nur 16 Bit breit ausgelegt sind, müssen die Eingänge entweder in 2 Zyklen geladen werden, oder der Ausgang des Addierers an einen der Eingänge gelegt werden.

Multiplizierer: Die Multiplikationseinheit kann in 2 verschiedenen Modi arbeiten:

1. 8 Bit unsigned x 16 Bit signed Multiplikation (1 Zyklus)
2. 16 Bit signed x 16 Bit signed Multiplikation (2 Zyklen, mit Benutzung des Addierers) mit 24 Bit Ergebnis (die oberen 8 Bit im Addierer können dann als Überlaufbits bei der Akkumulation von vielen Multiplikationsergebnissen (z.B. der Berechnung von net_i) verwendet werden)

2.3.3 Hard- und Software

Hardwaretechnische Realisierung: Das ganze System ist in einem Towergehäuse der Größe einer normalen Workstation untergebracht, das mittels Ethernetkabel an ein lokales Netz oder wahlweise über einen VME-Bus an einen dedizierten Rechner angeschlossen werden kann. Innerhalb dieses Gehäuses ist neben statischem Programmspeicher und dynamischem Dateispeicher die Recheneinheit auf einer Platine mit 8 Chips, in die jeweils 64 Prozessoren integriert wurden, sowie einem Chip für den CSC untergebracht.

Varianten: Es existieren zwei größere Varianten der CNAPS:

CNAPS I:

- Prozessorzahl: 256
- Taktgeschwindigkeit: 15 Mhz
- Nennleistung: 3.84 Milliarden Multiply+Accumulate/ sec
- Programmspeicher: 64 KB (8K Befehle)

CNAPS II:

- Prozessorzahl: 512
- Taktgeschwindigkeit: 20 Mhz
- Nennleistung: 10.24 Milliarden Multiply+Accumulate/ sec
- Programmspeicher: 512 KB (64K Befehle)

Zudem sind auch Varianten mit 64 und 128 Prozessoren als Einsteckplatinen statt eines kompletten CNAPS-Servers mit Frontrechner und Ethernetanbindung möglich.

Software: Geliefert wird die CNAPS mit einem CNAPS-C Übersetzer (eine um Festkomma-Konstrukte und Parallelanweisungen erweiterte Untermenge von ANSI-C) mit Inline-Assembler und Bibiotheken für schnelle Matrixmultiplikation, sowie einem Debugger und einem CNAPS-Simulator auf Microcodeebene. Außerdem werden Simulatoren für MLPs mit 16 Bit sowie 32 Bit Gewichten, mit und ohne Impuls-Term, mitgeliefert.

Programmiermodell: Die parallelen Konstrukte von CNAPS-C sind sogenannte *domains* (ähnlich denen der Sprache C^* [Thi90] für Parallelrechner), Vektoren von Strukturen die über eine gewisse Menge an Prozessoren definiert werden können. Alle in der Struktur vorhanden Variablen können dann innerhalb eines *domains* parallel bearbeitet werden. Da der Parallelrechner der CNAPS über keinen dedizierten sequentiellen Prozessor mit eigenem Speicher verfügt, der den sequentiellen Programmteil ausführt, übernimmt einer der Prozessoren des Prozesorfeldes diese Aufgabe. Um einen größeren globalen Speicher zur Verfügung stellen zu können, wird durch den Übersetzer ein fester Teil des lokalen Speichers jedes einzelnen Prozessors des Feldes belegt und diese kleinen Speicherstücke dem Benutzer als großer linear adressierbarer globaler Speicher präsentiert (siehe Abbildung 2.3). Als Folge hiervon muß für jeden Zugriff auf eine globale Variable diese aus dem jeweiligen lokalen Prozessorspeicher geholt werden, so daß das Rechnen mit globalen Variablen langsamer als mit parallelen Variablen ist.

Die oben erwähnten Einschränkungen von CNAPS-C gegenüber normalen ANSI-C bestehen weniger in Konstrukten als in den Variablentypen und ihren Verknüpfungsarten, da diese aufgrund der minimal ausgelegten Prozessoren sehr eingeschränkt sind. Es gibt nur Ganzahlen und Festkommazahlen mit 16 oder weniger Bit. Fließkommazahlen, transzendente Funktionen oder Division fehlen mangels Hardwareunterstützung ganz. Die einzige Möglichkeit zur Division von Festkommazahlen besteht in einem (im Vergleich zu Hardwaredivision) langsamen numerischen Verfahren. Ein weiterer Mangel ist das Fehlen eines dynamischen parallelen Speicherverwaltungssystems, so daß speicherintensive Algorithmen in den Größen ihrer Datenfelder fest codiert und gegebenfalls für neue Größen neu übersetzt werden müssen. Der Zugriff auf Dateien ist nur über eine begrenzte Menge an festen Dateizeigern möglich, deren Zuordnung zu Dateien im Filesystem oder dem Dateispeicher allerdings vom Benutzer vor dem Beginn eines Programmes festgelegt werden kann. Ein letzes Problem ist das Fehlen jeglicher Bildschirmausgabe-Funktionen. Zur Analyse eines Programmes muß dieses entweder auf Microcode-Ebene im Debugger verfolgt werden (ein recht mühsames und problematisches Unterfangen für größere Programme), oder alle relevanten Zwischenwerte in eine Datei geschrieben und diese nach Ablaufen des Algorithmus ausgewertet werden.

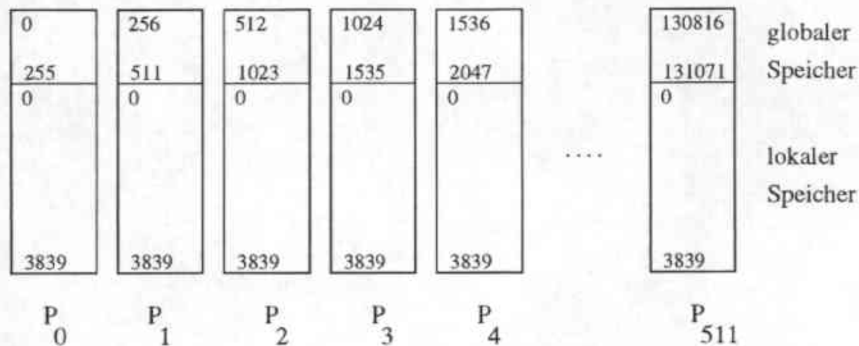


Abbildung 2.3: Das CNAPS-C Speichermodell

2.3.4 Zusammenfassung

Zum Abschluß noch einmal die wichtigsten Eckdaten für das Entwickeln von Algorithmen auf der CNAPS auf einen Blick:

- Schneller globaler Bus (1 Byte pro Zyklus)
- Langsame Nachbarschafts-Kommunikation (2 Bit pro Zyklus)
- 4 KB Datenspeicher pro Prozessor
- 64 K Befehlsworte Programmspeicher
- 32 MB Dateispeicher mit schnellem Buszugriff (1 Byte pro Zyklus)
- Zahlenverarbeitung nur im Festkommaformat
- keine Division
- verkürzte 16x16 Bit Multiplikation (24 Bit Ergebnis !)

Da hier nur ein sehr grober Überblick über das System gegeben werden konnte, sei der interessierte Leser auf die Handbücher [Ada93] sowie Artikel zu Leistungsfähigkeit und Design der CNAPS [McC90] verwiesen.

3. Berechnung von neuronalen Netzen mit beschränkter Genauigkeit

3.1 Einleitung

Das Einlernen von neuronalen Netzen ist besonders bei großen Netzen mit vielen Verbindungen und großen Mustermengen ein sehr zeitintensives Unterfangen. Eine der Möglichkeiten zur Geschwindigkeitssteigerung der Rechnung an sich, unabhängig von der Architektur des zugrundeliegenden Rechners (eine Thematik, die im nächsten Kapitel behandelt wird), ist die Verwendung von Festkommazahlen. Diese können im allgemeinen schneller als Fließkommazahlen miteinander multipliziert und addiert werden, da Angleichung der Exponenten und Renormalisierung entfallen. Zudem benötigen Festkommazahlen weniger Speicher als Fließkommazahlen, da sie ja ohne Exponenten auskommen: eine Eigenschaft, die insbesondere im Hinblick auf parallele Architekturen mit wenig lokalem Speicher, wie z.B. der CNAPS, von Vorteil ist. Die Frage, die sich dabei stellt, ist nun, wieviel (eigentlich: wiewenig) Vor-, respektive Nachkommastellen die verwendeten Festkommazahlen haben dürfen, damit die mit ihnen realisierten neuronalen Netze ein ähnliches Lern- und Trainingsverhalten wie eine Implementierung mit Fließkommazahlen zeigen. Wünschenswert wäre hier natürlich eine möglichst niedrige Bitzahl, damit zum einen kostbarer Speicher, zum anderen aber auch Rechenzeit gespart werden kann.
Zuerst wird im folgenden ein neuronales Netz in einer Festkomma-Implementierung vorgestellt und dabei die Darstellung der einzelnen im Backpropagation-Algorithmus benötigten Werte erläutert. Die Darstellung der Gewichte, die bezüglich des Speicherverbrauchs eines neuronalen Netzes wichtigste Größe, benötigt in dieser Implementierung 24 Bit. Da eine niedrigere Bitzahl die ebengenannten Vorteile hat und der größte zeitliche und speichertechnische Aufwand des Backpropagation-Algorithmus in der Multiplikation der Aktivierungen mit den Gewichten besteht, werden als zweites verschiedene Rundungsverfahren vorgestellt, mit denen die Zahl der Bits einer Gewichtsänderung und damit die Anzahl der Bits eines Gewichts von 24 auf 16 vermindert werden kann. Die einzelnen Verfahren werden kurz in den sie unterscheidenden Merkmalen vorgestellt und ihre Güte anhand einiger praktischer Experimente überprüft.

3.2 Ein neuronales Netz in Festkomma-Darstellung

Im folgenden wird die Darstellung von Festkommazahlen in der Notation x_y erfolgen. Dies bedeutet, daß die zugehörige Variable als vorzeichenbehaftete Festkommazahl mit x Vorkommastellen und y Nachkommastellen dargestellt ist. Somit umfaßt die zugehörige Zahl den Wertebereich $[-2^{x-1}, 2^{x-1})$ in Schritten von 2^{-y}.
Für eine Implementierung eines neuronalen Netzes genügt es nach Baker [BH88], wenn die Variablen des *Backpropagation*-Verfahrens folgende Darstellung haben:

$$i_i, a_i, t_i \quad : \quad 1_15$$
$$\omega_{ij} \quad : \quad 4_19^1$$
$$net_i, \delta_i \quad : \quad 4_12$$

Da der Wertebereich von f_{act} Kapitel 1 folgend zwischen -1.0 und 1.0 liegt, genügt es, für $a_i = f_{act}(net_i)$ ein Vorkommabit (das Vorzeichen) und 15 Nachkommabits zu verwenden. Dieselbe Darstellung ist daher auch für i_i und t_i zu wählen. Die Vorkommagenauigkeit für ω_{ij} ist intuitiv dadurch erklärbar, daß der dadurch realisierte Wertebereich $[-8.0, 8.0)$ der relevante Definitionsbereich für $f_{act} = tanh$ respektive $f_{act} = sig$ ist, und nur so ein Gewicht bei genügend großer Aktivierung des vorhergehenden Neurons sein nachgeschaltetes Neuron hinreichend beeinflußen kann. Zu ähnlichen Ergebnissen sind Hoehfeld und Fahlmann in Experimenten zur Darstellungsgenauigkeit von Gewichten im Cascade-Correlation-Algorithmus [HF91] und Baker in Untersuchungen zur Darstellungsgenauigkeit in MLPs [BH88][HH90] gekommen. Die Nachkommagenauigkeit der Gewichtsdarstellung folgt aus der Tatsache, daß der Multiplizierer der CNAPS nur 24-Bit Ergebnisse liefert. Die Darstellungsgenauigkeit der δ_i schließlich ist ebenso nur experimentell begründbar. Mit diesen Werten stellt sich die Berechnung eines dreischichtigen Netzes wie folgt dar:

Berechnung			Darstellungsgenauigkeiten		
$hiddennet_i$	$=$	$(4_12)(\sum_j \omega_{ij} * i_j)$	4_12	$=$	$(4_12)(\sum(4_19 * 1_15))$
$hiddenact_i$	$=$	$f_{act}(hiddennet_i)$	1_15	$=$	$f_{act}(4_12)$
$outnet_i$	$=$	$(4_12)(\sum_j \omega_{ij} * hiddenact_j)$	4_12	$=$	$(4.12)(\sum(4_19 * 1_15))$
$outact_i$	$=$	$f_{act}(outnet_i)$	1_15	$=$	$f_{act}(4_12)$
$outerr_i$	$=$	$outact_i - t_i$	1_15	$=$	$1_15 - 1_15$
$out\delta_i$	$=$	$(4_12)(outerr_i * f'_{act}(outnet_i)$	4_12	$=$	$(4_12)(1_15 * 1_15)$
$hiddenerr_i$	$=$	$(4_12)(\sum \omega_{ij} * out\delta_j)$	4_12	$=$	$(4_12)(\sum(4_12 * 4_12))$
$hidden\delta_i$	$=$	$(4_12)(outerr_i * f'_{act}(hiddennet_i)$	4_12	$=$	$(4_12)(1_15 * 1_15)$
$learndelta_i$	$=$	$(4_12)\delta_i * learningrate$	4_12	$=$	$(4_12)(4_12 * 4_12)$
ω_{ij}	$=$	$\omega_{ij} + (4_19)a_j * learndelta_i$	4_19	$=$	$4_19 + (4_19)(1_15 * 4_12)$

Die in diesem Algorithmus vorkommenden Summierungen von Produkten bei der Bildung von net_i und err_i werden in einem 32 Bit Register akkumuliert. Die einzelnen Produkte werden dabei nach der Multiplikation auf 24 Bit reduziert, indem die hinteren Bits abgeschnitten werden, um in den oberen 8 Bits des Akkumulationsregisters noch Platz für eventuelle Überläufe zu haben. Diese Eigenschaft des Multiplizierers der Hardware führt auch zum fehlendem Bit Genauigkeit bei der Gewichtsdarstellung. Eine Gewichtsänderung ist das Ergebnis folgender Multiplikation:

$\Delta\omega_{ij}$	$=$	$a_j * learndelta_i$	5.19	$=$	$(1.15 * 4.12)$

Da aber zwei vorzeichenbehaftete Zahlen miteinander multipliziert wurden, hat die sich ergebende Zahl nur eine Genauigkeit von 4.19[2], d.h. Bit 23 und Bit 22 dieser 5_19 Zahl sind immer gleich.

3.3 Darstellungsgenauigkeit für Gewichte

Von verschiedenen Autoren sind theoretische [HH90] und praktische [HH90][BH88][HF91] Untersuchungen auf unterschiedlichen Problemstellungen zur Darstellungsgenauigkeit von Gewichten in neuronalen Netzen gemacht worden.

[1] Diese Darstellung ist offensichlicherweise kein Vielfaches von 8, aber natürlich wird in der Implementierung immer aus Einfachheit mit Zahlen in 4.20 Darstellung gerechnet. Das fehlende Bit Genauigkeit ergibt sich aufgrund einer später zu erläuternden speziellen Einschränkung der Hardware auf der die Versuche durchgeführt wurden

[2] Der einzige Spezialfall a_j = -1.0 und $learndelta_i$ = -4.0, für den diese Aussage nicht zutrifft, kann in dieser Implementierung nicht vorkommen, da die Aktivierungsfunktion nie -1.0 als Wert annimmt.

Von Holt [HH90] sind Methoden entwickelt worden, die Fehlerpropagierung bei der Verwendung beschränkter Genauigkeiten in neuronalen Netzen zu berechnen. Mithilfe dieser Methoden wurde von ihm der Einfluß des Fehlers auf das Berechnen und das Einlernen zweier MLPs vorhergesagt und die theoretischen Ergebnisse anhand von Simulationen verifiziert. Die von ihm verwendeten MLPs waren zum einen ein 100-100-100 MLP mit 100 zufällig generierten Muster und zum anderen ein 2-3-1 MLP mit den Mustern des XOR-Problems. Für das Berechnen stellten sich hier eine Darstellung der Gewichte mit 8 Bit (4_4), für das Einlernen eine Genauigkeit von 14 bis 16 Bit (4_10 bis 4_12) als kritisch und minimal erforderlich heraus.
Baker [BH88] hat seine Untersuchungen anhand einer einfachen Zeichenerkennungsaufgabe mit einem 144-30-26 MLP und 286 Trainingsmustern durchgeführt. Hierbei erwies sich die Darstellung der Gewichte als 16 Bit Festkommazahl (4_12) nicht nur als ausreichend, sondern benötigte auch gegenüber einer Fließkomma-Implementierung weniger Muster-Präsentationen.
Höhfeld und Fahlmann [HF91] schließlich haben die erforderlichen Darstellungsgenauigkeiten im Cascade-Correlation Algorithmus, einer Methode zur Konstruktion recht spezieller neuronaler Netze, die in ihrer Struktur stark von normalen MLPs abweichen, an 3 Problemstellungen (sonar, spirals, 6-parity) untersucht. Auch hier stellten sich Bitgenauigkeiten von 13 bis 15 Bit als notwendig für die Konvergenz der Netze heraus (Hier wurden allerdings keine festen Wertebereiche für die Gewichte festgelegt, sondern mittels eines Skalierungsfaktors für das ganze Netz eine Normalisierung der Gewichte in den Bereich [-1,1) vorgenommen). Mit einem speziellen Verfahren, dem *Stochastic weight update*, konnten diese Genauigkeiten weiter bis zu 7 Bit reduziert werden, wobei allerdings der Cascade-Correlation Algorithmus für derart eingeschränkte Darstellungsgenauigkeiten größere Netze konstruiert hat.
Ausgehend vom oben angegebenen Berechnungsschema und unserer 23 Bit-Darstellung (4_19), werden im folgenden fünf Rundungsverfahren zur Reduzierung der benötigten Gewichtsbits von 23 Bit auf 16 Bit vorgestellt und das Verhalten an drei Aufgabenstellungen für neuronale Netze untersucht.

3.3.1 Notation

Die einzelne Gewichtsänderung ist Ergebnis der Multiplikation zweier 16-Bit Zahlen gemäß folgender Gleichung:

$$\boxed{\Delta\omega_{ij} = learndelta_i * a_j} \quad \boxed{4_19 = (4_19)(4_12 * 1_15)}$$

Um die Darstellung der Gewichte in der Bitzahl weiter zu vermindern, müßte man die sich bei der Gewichtsänderung ergebenden niedrigstwertigen Bits in irgendeiner Art und Weise abschneiden, aber gleichzeitig versuchen, die Konvergenz des Netzes zu erhalten.
Um die Situation noch einmal zu verdeutlichen,:

$$\Delta\omega = \mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3 = \underbrace{\square\square\square\square}_{b_{22}\ \ b_{19}}.\underbrace{\square\square\square\square\square\square\square\square\square\square\square\square}_{b_{18}\ \ b_7}\underbrace{\square\square\square\square\square\square\square}_{b_6\ \ b_0}$$

Der Bitvektor $\mathbf{u}_1 = (b_{22} \ldots b_{19})$ beschreibt hierbei die Vorkommabits, $\mathbf{u}_2 = (b_{18} \ldots b_7)$ die Nachkommabits und $\mathbf{u}_3 = (b_6 \ldots b_0)$ die abzuschneidenden Bits der einzelnen Gewichtsänderung. Ein paar Zahlen, die im folgenden eine Rolle spielen, seien hier explizit erwähnt:

$$\begin{array}{rcrcl} 2^0 & = & 1.000000 & = & 0001.000000000000\ 0000000 \\ -2^0 & = & -1.000000 & = & 1111.000000000000\ 0000000 \\ 2^{-13} & = & 0.000122 & = & 0000.000000000000\ 1000000 \\ -2^{-13} & = & -0.000122 & = & 1111.111111111111\ 1000000 \\ 2^{-14} & = & 0.000061 & = & 0000.000000000000\ 0100000 \\ -2^{-14} & = & -0.000061 & = & 1111.111111111111\ 1100000 \end{array}$$

Die Darstellung der negativen Zahlen wird im folgenden, wie auch in diesem Beispiel, immer im Zweierkomplement erfolgen, da dieses die Darstellung ist, auf der auch der Rechner operiert. Der Punkt („.") in den Zahlen symbolisiert die Skalierung der jeweiligen Festkommazahl, d.h. die Anzahl ihrer Vor- und Nachkommastellen.

3.3.2 Rundungsoperatoren

Es sollen nun fünf Möglichkeiten diese 4_19-Zahl auf eine 4_12-Zahl zu runden mit ihren Vor- und Nachteilen vorgestellt werden. Zuerst wird dabei der jeweilige Operator in seiner Funktionsweise erläutert. Danach werden seine Eigenschaften, u.a. der maximal verursachte Fehler (err_{max}), der durchschnittlich verursachte Fehler ($err_{average}$) und der Mittelwert ($mean$) der von ihm gerundeten Zahlen, sowie Implementierungsmöglichkeiten vorgestellt.

3.3.2.1 Abschneiden von $b_6 \ldots b_0$

Operator:

$$R_{cut}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = \mathbf{u}_1.\mathbf{u}_2$$

Eigenschaften:

$mean$	$err_{average}$	err_{max}
2^{-13}	2^{-13}	2^{-12}

Bemerkungen:
Durch einfaches Abschneiden werden die negativen Änderungen aufgrund ihrer Darstellung gegenüber den positiven Änderungen bevorzugt. Da das Entfernen der letzten Bits der Anwendung eines Gaussoperators entspricht, also auf die größte darstellbare kleinere Zahl abgebildet wird, werden negative Gewichtsänderungen zwischen -2^{-12} und -2^{-19} auf -2^{-12}, positive zwischen 2^{-12} und 2^{19} hingegen auf Null abgebildet. Dieses Ungleichgewicht führt bei einer symmetrischen Veteilung der zu rundenden 4_19-Werte mit Mittelwert Null zu einer Verteilung der gerundeten 4_12-Werte mit Mittelwert $2^{-12}/2 = 2^{-13}$. Der durchschnittlich gemachte Fehler bei Gewichtsänderungen bei unendlich langem, gleichverteiltem $\mathbf{u}_3$ beträgt 2^{-13}, während der maximale Fehler 2^{-12} ist.

Realisierung:
Dieser Operator bedarf keines zusätzlichen Aufwandes bei der Implementierung, da beim Schieben nach links automatisch Nullen nachgeschoben werden.

3.3.2.2 Abschneiden mit *jamming*

Operator:

$$R_{jam}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = \begin{cases} \mathbf{u}_1.b_{18}\ldots b_8 0 & b_0\ldots b_6 = 0\ldots 0 \\ \mathbf{u}_1.b_{18}\ldots b_8 1 & b_0\ldots b_6 \neq 0\ldots 0 \end{cases}$$

Eigenschaften:

$mean$	$err_{average}$	err_{max}
0	2^{-13}	2^{-12}

Bemerkungen:
Um die letzten Bits, die bei R_{cut} einfach abgeschnitten werden, nicht unbeachtet zu lassen, kann

man für den Fall, daß einige der abzuschneidenden Bits gesetzt waren, das letzte der noch verbliebenen Bits setzen. Der Vorteil der Verwendung dieses Operators bei neuronalen Netzen ist folgender: Sollte eine Gewichtsänderung so klein sein, daß sie auf Null gerundet würde, so wird einfach die kleinstmögliche Gewichtsänderung durchgeführt. Ein weiterer Vorteil dieses Operators gegenüber R_{cut} ist, daß der Mittelwert der gerundeten Zahlen Null ist. Ein Beispiel für die Anwendung dieses Operators ist in Tabelle 37.2 zu sehen. Hier ist dann auch das Manko dieses Operators ersichtlich: Der maximale Fehler, der bei der Anwendung des Operators gemacht wird, beträgt wie bei R_{cut} in einzelnen Fällen bis zu 0.75, d.h. im Fall unserer Gewichtsänderung $2^{-12} - 2^{-19}$.

Realisierung:
Dieser Operator kann in der Hardwareimplementierung über beliebig viele Nachkommabits recht einfach realisiert werden, indem man das Schieberegister mit folgender Zusatzlogik für das Rechtsschieben versieht:

$$\text{Setze das letzte Bit} = \begin{cases} 0 & \text{falls eine 0 rausgeschoben wurde} \\ 1 & \text{falls eine 1 rausgeschoben wurde} \end{cases}$$

x		$R_{cut}(x)$	
binär	dezimal	binär	dezimal
01.11	1.75	01.	1
01.10	1.5	01.	1
01.01	1.25	01.	1
01.00	1.0	01.	1
00.11	0.75	00.	0
00.10	0.5	00.	0
00.01	0.25	00.	0
00.00	0.0	00.	0
11.11	-0.25	11.	-1
11.10	-0.5	11.	-1
11.01	-0.75	11.	-1
11.00	-1.0	11.	-1
10.11	-1.25	10.	-2
10.10	-1.5	10.	-2
10.01	-1.75	10.	-2

Tabelle 3.1 Runden durch Abschneiden

x		$R_{jam}(x)$	
binär	dezimal	binär	dezimal
01.11	1.75	01.	1
01.10	1.5	01.	1
01.01	1.25	01.	1
01.00	1.0	01.	1
00.11	0.75	01.	1
00.10	0.5	01.	1
00.01	0.25	01.	1
00.00	0.0	00.	0
11.11	-0.25	11.	-1
11.10	-0.5	11.	-1
11.01	-0.75	11.	-1
11.00	-1.0	11.	-1
10.11	-1.25	11.	-1
10.10	-1.5	11.	-1
10.01	-1.75	11.	-1

Tabelle 3.2 Runden durch Abschneiden mit jamming

3.3.2.3 Exaktes Runden

Operator:

$$R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = R_{cut}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3 + \underbrace{0\ldots0}_{|\mathbf{u}_1|}.\underbrace{0\ldots0}_{|\mathbf{u}_2|}\underbrace{10\ldots0}_{|\mathbf{u}_3|})$$

Eigenschaften:

$mean$	$err_{average}$	err_{max}
0	2^{-14}	2^{-13}

Bemerkungen:
Um einen hohen maximalen Fehler wie bei R_{jam} und R_{cut} zu vermeiden kann man die Gewichtsänderung exakt runden und somit den maximalen Fehler auf 2^{-13} begrenzen. Im Unterschied

zu R_{jam} werden bei diesem Operator allerdings wieder sehr kleine Zahlen auf Null abgebildet.

Realisierung:
Hierfür ist neben dem Ausmaskieren des Ergebnisses eine zusätzliche Addition von 2^{-13} nötig.

x		$R_{round}(x)$	
binär	dezimal	binär	dezimal
01.11	1.75	10.	2
01.10	1.5	10.	2
01.01	1.25	01.	1
01.00	1.0	01.	1
00.11	0.75	01.	1
00.10	0.5	01.	1
00.01	0.25	00.	0
00.00	0.0	00.	0
11.11	-0.25	00.	0
11.10	-0.5	11.	-1
11.01	-0.75	11.	-1
11.00	-1.0	11.	-1
10.11	-1.25	11.	-1
10.10	-1.5	10.	-2
10.01	-1.75	10.	-2

Tabelle 3.3 Exaktes Runden

x		$R_{roundlift}(x)$	
binär	dezimal	binär	dezimal
01.11	1.75	10.	2
01.10	1.5	10.	2
01.01	1.25	01.	1
01.00	1.0	01.	1
00.11	0.75	01.	1
00.10	0.5	01.	1
00.01	0.25	01.	1
00.00	0.0	00.	0
11.11	-0.25	11.	-1
11.10	-0.5	11.	-1
11.01	-0.75	11.	-1
11.00	-1.0	11.	-1
10.11	-1.25	11.	-1
10.10	-1.5	10.	-2
10.01	-1.75	10.	-2

Tabelle 3.4 Exaktes Runden mit Aufwertung

3.3.2.4 Exaktes Runden mit Aufwertung sehr kleiner Gewichtsänderungen

Operator:

$$R_{roundlift}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = \begin{cases} R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) & R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) \neq 0 \\ \underbrace{0\ldots0}_{|\mathbf{u}_1|}.\underbrace{0\ldots01}_{|\mathbf{u}_2|} & R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = 0 \quad \text{und} \quad \mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3 > 0 \\ \underbrace{1\ldots1}_{|\mathbf{u}_1|}.\underbrace{1\ldots1}_{|\mathbf{u}_2|} & R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = 0 \quad \text{und} \quad \mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3 < 0 \end{cases}$$

Eigenschaften:

Für $R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = 0$ gilt

mean	$err_{average}$	err_{max}
0	$2^{-13} + 2^{-14}$	2^{-12}

Für $R_{round}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) \neq 0$ gilt

mean	$err_{average}$	err_{max}
0	2^{-14}	2^{-13}

Bemerkungen:
Dieser Operator rundet den Wert exakt und wertet danach alle echten Gewichtsveränderungen, die auf Null gerundet wurden auf die betragsmäßig kleinstmögliche (positive oder negative) Änderung auf. Daher ist für den Fall einer Anwendung auf sehr kleine Zahlen der durchschnittliche Fehler höher als bei einer Anwendung auf große Zahlen.

Realisierung
Neben der zusätzlichen Addition wie bei R_{round} sind noch zwei Vergleiche und gegebenfalls Ändern der bisherigen Null-Gewichtsänderung auf die minimal mögliche nötig (jeweils für den minimal kleinsten positiven und negativen Wert)

3.3.2.5 Zufallsbasiertes Runden (*Stochastic weight update*)

Operator:

$Z(p) : [0,1] \rightarrow \{0,1\}$ sei ein Zufallszahlengenerator, der mit der Wahrscheinlichkeit p die Zahlen 0 oder 1 erzeugt :

$$\begin{aligned} \mathrm{P(Z(p)=1)} &= \mathrm{p} \\ \mathrm{P(Z(p)=0)} &= \mathrm{1\text{-}p} \end{aligned}$$

Desweiteren seien $R_{floor}(\Delta\omega_{ij})$ und $R_{ceil}(\Delta\omega_{ij})$ Operatoren, die $\omega_{ij} = \mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3$ auf die jeweils nächstkleinere respektive nächstgrößere 4_12-Zahl runden.
Außerdem sei der Operator $\mathrm{O}(\Delta\omega_{ij})$ mit

$$O(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) := 0.b_6 \ldots b_0$$

definiert.
Dieser Operator wandelt die abzuschneidenden Bits $\mathbf{u}_3$ in eine Zahl zwischen 0.0 und 1.0 um, die proportional zur Differenz von $\Delta\omega_{ij}$ und $R_{floor}(\Delta\omega_{ij})$ ist.

Stochastic weight update ist dann folgendermaßen erklärt:

$$R_{stoch}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = \begin{cases} R_{ceil}(\Delta\omega_{ij}) & Z(O(\Delta\omega_{ij})) = 1 \\ R_{floor}(\Delta\omega_{ij}) & Z(O(\Delta\omega_{ij})) = 0 \end{cases}$$

Eigenschaften:

$mean$	$err_{average}$	err_{max}
0	2^{-14}	2^{-12}

Bemerkungen:
In R_{jam} und $R_{roundlift}$ wurde die jeweilige Gewichtsänderung, egal wie klein sie auch sein mochte, auf den in der verminderten Genauigkeit betragsmäßig kleinstmöglichen Wert abgebildet. Dieses hat den Nachteil, daß z.B. in einem neuronalen Netz, in dem nur noch Gewichtsänderungen unterhalb der Darstellungsgrenze vorkommen, eben diese wesentlich größer durchgeführt werden. Somit führt z.B. ein zehnmaliges Ändern mit 0.1 zu einem zehnmaligen Ändern eines Gewichtes mit 1.0. Als Änderung dieser Vorgehensweise könnte man, statt zehnmal das Gewicht mit 1.0 zu ändern, es nur einmal mit 1.0 und neunmal mit 0.0 ändern. Somit hätte man im Mittel den gleichen Wert erreicht, wie er auch im normalen Verfahren ohne verminderte Genauigkeit vorkommen würde. Dieses ist die Idee auf der Stochastic weight update [HF91] beruht. Ein weiterer Vorteil liegt darin, daß selbst bei sehr kleinen Gewichtsänderungen eine gewisse Wahrscheinlichkeit besteht, daß das zugehörige Gewicht sich ändert. So wird zusätzlich das Festfahren der Gewichte („stuck units")[Fah88] während des Einlernen eines Netzes vermieden.
Realisierung:
Der Operator kann durch das Addieren einer gleichverteilten Zufallszahl Z_{random} mit gleicher Anzahl an Bits wie $\mathbf{u}_3$ und nachfolgendem Abschneiden der Restbits realisiert werden:

$$R_{stoch}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3) = R_{floor}(\mathbf{u}_1.\mathbf{u}_2\mathbf{u}_3 + \underbrace{0\ldots0}_{|\mathbf{u}_1|}.\underbrace{0\ldots0}_{|\mathbf{u}_2|}\underbrace{Z_{random}}_{|\mathbf{u}_3|})$$

x		$R_{stoch}(x)$	
binär	dezimal	binär	dezimal
01.11	1.75	01.+Z(0.11)	1.Z(0.75)
01.10	1.5	01.+Z(0.10)	1.Z(0.50)
01.01	1.25	01.+Z(0.01)	1.Z(0.25)
01.00	1.0	01.+Z(0.00)	1.Z(0.00)
00.11	0.75	00.+Z(0.11)	0.Z(0.75)
00.10	0.5	00.+Z(0.10)	0.Z(0.50)
00.01	0.25	00.+Z(0.01)	0.Z(0.25)
00.00	0.0	00.+Z(0.00)	0.Z(0.00)
11.11	-0.25	11.+Z(0.11)	-1.Z(0.75)
11.10	-0.5	11.+Z(0.10)	-1.Z(0.50)
11.01	-0.75	11.+Z(0.01)	-1.Z(0.25)
11.00	-1.0	11.+Z(0.00)	-1.Z(0.00)
10.11	-1.25	10.+Z(0.11)	-2.Z(0.75)
10.10	-1.5	10.+Z(0.10)	-2.Z(0.50)
10.01	-1.75	10.+Z(0.01)	-2.Z(0.25)

Tabelle 3.5 Zufallsbasiertes Runden

3.3.3 Experimente

3.3.3.1 Die Benchmarks

Um die vorgestellten Rundungsverfahren untereinander und im Vergleich mit der 24 Bit Darstellung bewerten zu können, nehmen wir uns ein paar Aufgabenstellungen für Neuronale Netze und stellen experimentell fest, wie sich die einzelnen Methoden verhalten. Als Maßstab sollen uns dabei zwei rein akademische Aufgaben und ein praktisches Problem aus dem Bereich der Spracherkennung dienen:

Encoder: Im diesem rein akademischen Problem geht es darum, die Vektoren $(1,0,\ldots 0)$, $(0,1,\ldots 0)$, $\ldots,(0,0,\ldots 1)$ zu autoassoziieren. Die Darstellung der Nullen und Einsen erfolgt durch 0.0 und 1.0 (für $f_{act} = sig$) respektive durch -1.0 und 1.0 (für $f_{act} = tanh$). Das Netz sollte hierbei, wenn möglich, alle Eingabevektoren auswendig lernen und wieder reproduzieren können. Typische Netzgrößen für kleine n sind $2^n - n - 2^n$, die Anzahl der Trainingsmuster beträgt $|\mathcal{N}_\mathcal{I}|$.

Parity: Bei dieser ebenfalls künstlichen Problemstellung soll ein neuronales Netz lernen für beliebige binäre Eingabevektoren die Parität zu bestimmen. Die Nullen und Einsen werden wie beim Encoder dargestellt. Auch hier ist durch die Trainingsmenge das zu lernende Problem vollständig gegeben und es wird ebenfalls nur die Fähigkeit zur Unterscheidung der einzelnen Muster vom Netz gefordert. Typische Netzgrößen sind hier n-n-1, da das Problem recht schwierig ist (Durch die Veränderung einer einzelne Eingabe muß am Netzausgang die zum bisherigen Ergebnis komplementäre Ausgabe erscheinen). Die Anzahl der Eingabevektoren beträgt $2^{|\mathcal{N}_\mathcal{I}|}$. Für den Fall von nur 2 Eingabeneuronen ergibt sich das bekannte XOR-Problem.

Phonemerkennung: In diesem praktischen Problem geht es darum, aus einer Folge von 7 Frequenzvektoren mit jeweils 16 Melscale-Koeffizienten, das zum jeweiligen Zeitpunkt am wahrscheinlichsten gesprochene von 49 Phonemen herauszufinden. Als Trainingsmenge dienen 216

Sätze verschiedener Sprecher (Resource Management Task), die zusammen eine Menge von 71805 Mustern ergeben. Die unabhängige Testmenge umfaßt 50 Sätze mit insgesamt 17610 Mustern. Unter Verwendung von 200 versteckten Neuronen sind in dieser Darstellung mit dem einfachen Backpropagationverfahren in einer Fließkomma-Implementierung 40% Erkennungsquote zu erreichen (optimale Lernrate 0.001, 80 Epochen). Das Netz ist nicht in der Lage die Mustermenge auswendig zu lernen und behält damit einen Restfehler während des Trainings.

3.3.3.2 Die Ergebnisse

Die nun folgenden Tabellen geben die Versuchsergebnisse für jeweils ein bestimmtes Problem mit einem bestimmten Netz und verschiedenen Lernraten an. Um das Problem der festgefahreren Gewichte (*stuck units*) zu umgehen, wurde in allen Versuchen zusätzlich zum bisher geschilderten Verfahren noch eine Translation der Ableitung um 0.01 (*derivative offset*) [Fah88] vorgenommen. Dieser Wert hat sich in der vorliegenden Implementierung experimentell als günstig herausgestellt.

Da es in den ersten zwei Problemen um die Fähigkeit geht eine Mustermenge überhaupt lernen zu können, wird diese ein wesentliches Merkmal zur Beurteilung der einzelnen Rundungsverfahren sein. Weitere Merkmale sind die Anzahl der benötigten Epochen, die Stabilität des sich ergebenden Netzes, sowie das allgemeine Lernverhalten.

Die für diese Merkmale in den Tabellen benutzten Spezialnotationen bedeuten:

n	Das Netz hat nach n Epochen die Mustermenge gelernt
-	Das Netz hat keine Anzeichen gemacht die Mustermenge auch nur annäherungsweise zu lernen
\m	Das Netz stagniert mit m Mustern Erkennungsrate
~	Das Netz ist nicht stabil, sondern zeigt (sehr große) Schwankungen in der Erkennungsleistung während des Trainings

Für das dritte Problem ist nicht so sehr ausschlaggebend, ob das Netz die Trainingsmenge auswendig lernt (da dieses schon in der Fließkomma-Implementierung nicht möglich ist), sondern welche Erkennungsleistung sich auf der unabhängigen Testmenge ergibt. Ein weiteres Kriterium zur Beurteilung wird die Konvergenz des Netzes sein. Es bedeuten:

x	Das Netz ist bei der eingestellten Lernrate bei der größten im Lernverlauf erreichten Erkennungsleistung konvergiert und hat dort eine Erkennungsleistung von x Prozent gehabt
(x)	Das Netz ist bei der eingestellten Lernrate nicht bei der größten im Lernverlauf erreichten Erkennungsquote konvergiert, sondern auf einem niedrigeren Niveau. Die größte im Lernverlauf erreichte Erkennungsquote betrug x Prozent

Die fett geschriebenen Zahlen in den Tabellen geben die jeweils beste erreichte Leistung eines Operators an.

	Lernrate				
	0.01	0.05	0.1	0.5	1.0
24 Bit	1200	200	**100**	310	-
round	420	100	**50**	\5	\3
roundlift	420	100	**50**	\4	\3
stoch	440	100	**50**	330	\3
jam	360	100	**50**	\4	\4
cut	580	100	**40**	\5	\3

Tabelle 3.6: Lernverhalten des 8-3-8 Encoder Netzes (8 Muster)

	Lernrate				
	0.01	0.05	0.1	0.2	0.4
24 Bit	\14	480	200	**100**	-
round	\15~	300	140	**120**	\6
roundlift	1300	160	**80**	140	\6
stoch	2300	140	140	**60**	\9
jam	900	140	100	**80**	\6
cut	\9	200	180	**100**	\6

Tabelle 3.7: Lernverhalten des 16-5-16 Encoder Netzes (16 Muster)

	Lernrate				
	0.01	0.05	0.1	0.2	0.4
24 Bit	\31	1200	540	**500**	-
round	\30	**440**~~	\30	\9	\7
roundlift	3600	500~	**360**~	\8	\6
stoch	2800	420~	**320**~~	\8	\7
jam	3800	**360**~	\28	\14	\6
cut	—	\20	**\21**	\6	\6

Tabelle 3.8: Lernverhalten des 32-6-32 Encoder Netzes (32 Muster)

	Lernrate					
	0.01	0.1	0.2	0.4	0.6	1.0
24 Bit	-	-	\63	**380**	\60	\53
round	-	-	**\63**	\60	\58	\47~
roundlift	-	-	**\63**	\61	\58	\49~
stoch	-	-	**\63**	\60	\58	\47~
jam	-	-	**\63**	\63	\56	\45~
cut	\57	-	**\62**	\61	\60	\57~

Tabelle 3.9: Lernverhalten des 6-8-1 Parity Netzes (64 Muster)

	Lernrate					
	0.01	0.1	0.2	0.4	0.6	1.0
24 Bit	-	**\63**	**\63**	**\63**	\54	\45
round	-	-	**\62**	\58	\60	\52
roundlift	-	-	**\63**	\62	\60	\44
stoch	-	-	**\62**	\56	\59	\40
jam	-	-	**\62**	\61	\61	\45
cut	\57	-	**\59**	\58	**\59**	\52

Tabelle 3.10: Lernverhalten des 6-6-1 Parity Netzes (64 Muster)

	Lernrate					
	0.005	0.01	0.02	0.05	0.1	0.2
24 Bit	37.42	42.78	**43.21**	39.85	35.9	(18.4)
round	32.48	**35.83**	33.45	32.51	(25.37)	(15.34)
roundlift	29.05	31.69	**33.2**	32.85	(28.18)	(15.18)
stoch	25.3	(27.87)	**(28.19)**	(27.98)	(24.53)	(13.23)
jam	29.29	31.66	**33.35**	31.81	(27.64)	(13.99)
cut	(3.15)	(6.21)	(10.55)	**(13.81)**	(11.77)	(8.43)

Tabelle 3.11: Lernverhalten des 112-200-49 Netzes zur Phonemerkennung

3.3.4 Auswertung

Im folgenden werden die fünf Rundungsoperatoren untereinander und zum Vergleich das Verhalten der 24 Bit Darstellung betrachtet:

cut: Das einfache Abschneiden hat sich bei kleinen Netzen (Tabellen 3.6, 3.7) nicht nachteilig ausgewirkt. Bei größeren Netzen und schwierigeren Aufgaben hingegen haben sich Mängel in der Lernfähigkeit des Netzes gezeigt (Tabellen 3.8, 3.10). Bei der Phonemerkennung schließlich ist das Netz nicht einmal konvergiert.

stoch: Das Verhalten der Netze dieses Operators ist auf den Parity- und Encoder-Netzen ähnlich denen der anderen Rundungsoperatoren. Auf der Phonemerkennungs-Aufgabe jedoch konvergieren die Netze sehr schlecht und haben insgesamt auch schlechtere Erkennungsraten als die der übrigen drei Operatoren.

jam: Die mit diesem Operator berechneten Netze zeigen insgesamt ein gutes Konvergenzverhalten und erreichen bei der Phonemerkennung eine, verglichen mit einer Fließkommadarstellung allerdings recht niedrige, Erkennungsquote von 33 %

roundlift: Im wesentlichen verhalten sich die Netze dieses Operators äquivalent denen von R_{jam}

round: Auf den Parity- und Encoder-Problemen zeigt sich ein ähnliches Verhalten wie bei R_{jam} und $R_{roundlift}$. Bezüglich der Konvergenz bei der Phonemerkennung schneidet dieser Operator jedoch deutlich besser ab. Seine maximale Erkennungsquote liegt um den Faktor 2 (1 Bit) tiefer und ist um 2% höher als die der beiden letztgenannten Operatoren. In Tabelle 3.11 sind deutlich die Vorteile der korrekten Rundung gegenüber diesen bei besonders niedrigen Lernraten zu erkennnen.

24 Bit: Bei geringen Lernraten konvergieren die mit der vollen Genauigkeit berechneten Netze wesentlich langsamer als die mit 16 Bit Gewichten und Rundungsoperatoren berechneten Netze

(Tabellen 3.6, 3.7). Dies ist darin begründet, daß die durch die niedrigen Lernraten entstehenden niedrigen Gewichtsänderungen korrekt vorgenommen werden, während bei den 16 Bit Netzen aufgrund der Rundungsoperatoren wesentlich größere Gewichtsänderungen enstehen. Diese größeren Gewichtsänderungen führen dazu, daß die Netze sich wie bei höheren Lernraten verhalten, was bei den Encoder- und Parity-Beispielen vorteilhaft bezüglich der Konvergenzgeschwindigkeit ist. Ansonsten zeigen die mit 24 Bit Genauigkeit berechneten Netze ein stabileres und besseres Konvergenzverhalten (Tabellen 3.8, 3.9). Auf der Phonemerkennungs-Aufgabe schließlich ist die erreichte Erkennungsqutote um 8% höher als die des Operators R_{round} und noch um 3% höher als die mit Fließkommadarstellung in 80 Epochen erreichbare (Das Netz in Fließkommadarstellung war nach den 80 Epochen allerdings noch am Lernen, während das mit 24 Bit Genauigkeit gerechnete Netz schon konvergiert war[3]).

3.3.5 Zusammenfassung

Im Gegensatz zu den in der Literatur üblichen Ergebnissen, in denen eine 16 Bit Darstellung der Gewichte für die jeweilig betrachteten Aufgaben und Netze für eine gute Konvergenz ausreicht, hat sich für eine Festkomma-Implementierung nach Baker am Phonemerkennungs-Task gezeigt, daß für optimale Erkennungsquoten höhere Genauigkeiten notwendig sind. Sollten jedoch aus rechen- oder speichertechnischen Gründen 16 Bit Gewichte notwendig sein, so sollte als Rundungsoperator R_{round} gewählt werden, da dieser im Gegensatz zu R_{jam} oder $R_{roundlift}$ ein besseres Verhalten bei kleinen Gewichtsänderungen zeigt. Der Operator R_{stoch}, der in Hoehfelds Experimenten am Cascade-Correlation-Algorithmus eine wesentlich niedrigere Darstellungsgenauigkeit für Gewichte ermöglichte, hat in den hier mit MLPs durchgeführten Experimenten kein besseres Verhalten als die anderen Operatoren gezeigt und bei der Phonemerkennung sogar ein deutlich schlechteres.

[3] Aufgrund einer gerade wieder mal anstehenden Demo konnte das Netz leider nicht zuende trainiert werden. Da aber mit der Festkommadarstellung 43% erreichbar sind, steht zu vermuten, daß auch das Netz in Fließkommadarstellung diese Zahl nach weiteren 80 bis 100 Epochen erreicht. An dieser Stelle ein großes Danke an Hermann Hild und Ivica Rogina, die mir ein klein bischen Rechenzeit auf ihren total überlasteten Maschinen gewährt haben.

4. Möglichkeiten der Parallelisierung neuronaler Netze

4.1 Einleitung

In diesem Kapitel sollen die verschiedenen Parallisierungsebenen, die bei Berechnung und Training neuronaler Netze möglich sind, vorgestellt werden. Hierfür werden als erstes die verschiedenen Aufgabenstellungen aufgelistet, die in Zusammenhang mit neuronalen Netzen in der Praxis auftreten. Danach werden dann die bei neuronalen Netzen möglichen Parallelisierungen erläutert, die jeweilige theoretische Beschleunigung errechnet, sowie ihre Nutzbarkeit für die unterschiedlichen Aufgabenstellungen und die beiden großen Parallelrechnerklassen, SIMD und MIMD, ermittelt. Zum Abschluß werden beispielhaft einige der bisher existenten parallelen Simulatoren für neuronale Netze mit ihren Leistungsdaten und den in ihnen genutzten Parallelitätsformen vorgestellt.

4.2 Aufgabenstellungen bei der Berechnung neuronaler Netze

Folgende Aufgabenstellungen sind im Zusammenhang mit neuronalen Netzen häufig anzutreffen:

- **Aufgabenstellung 1:**
  Berechne den Ausgabewert eines festen Netzes für eine Mustermenge
  (Netz ist trainiert und soll nur noch in einer Echtzeitanwendung eingesetzt werden)

- **Aufgabenstellung 2:**
  Trainiere ein Netz mit einer Mustermenge (Muster, Block-/Epochenlernen)
  (Die Netzarchitektur, das Lernverfahren und seine Parameter sind gefunden und das Netz soll nur noch auf eine neue Mustermenge eintrainiert werden)

- **Aufgabenstellung 3:**
  Trainiere mehrere leicht unterschiedliche Netze (unterschiedliche Lernparameter, unterschiedlich große versteckte Schichten) auf einer Mustermenge
  (Das ungefähre Netz ist gefunden und unter den Varianten soll das beste bestimmt werden)

- **Aufgabenstellung 4:**
  Trainiere mehrere stark unterschiedliche Netze auf einer Mustermenge
  (Für ein gegebenes Problem soll ein entsprechendes Netz gefunden werden)

- **Aufgabenstellung 5:**
  Trainiere mehrere stark unterschiedliche Netze auf stark unterschiedlichen Mustermengen
  (Für mehrere gegebene Probleme sollen entsprechende Netze gefunden werden)

4.3 Parallelisierungsmöglichkeiten für neuronale Netze

Um nun die Parallelisierungsmöglichkeiten zu ermitteln, wird erst einmal die letzte Aufgabenstellung zugrundegelegt. Danach wird dann zu prüfen sein, inwieweit und unter welchen Bedingungen die Parallelitätsebenen auf die anderen Aufgabenstellungen übernommen werden können.
Folgende fünf Parallelisierungsmöglichkeiten existieren für neuronale Netze:

- Netzparallelität
- Musterparallelität
- Schichtparallelität
- Neuronenparallelität
- Gewichtsparallelität

In den folgendenden fünf Abschnitten werden diese Parallelisierungsebenen sowie mögliche Kombinationsformen besprochen.

4.3.1 Netzparallelität

Wenn eine Menge von unterschiedlichen Netzen auf verschiedenen Trainingssätzen zu berechnen oder zu trainieren ist, so kann dies im einfachsten Fall parallel auf verschiedenen Rechnern durchgeführt werden. Sofern die Netze von einer halbwegs ähnlichen Struktur sind, ist dieses unter Umständen auch auf einem SIMD-Rechner realisierbar.

Mögliche Beschleunigung:

Sei t_i die Zeit, die das i-te Netz zum Trainieren braucht und n die Anzahl der Prozessoren. Bei einer Anzahl an Trainingsaufgaben, die größer als die zugrundeliegende Menge an Prozessoren ist, und optimalem Jobscheduling ergibt sich folgendes:

$$
\begin{aligned}
T_n &= \frac{(\sum_i t_i)}{n} \\
S_n &= n
\end{aligned}
$$

Diese Parallelisierungstechnik ist üblich zur Durchsatzsteigerung bei den Aufgabenstellungen 3, 4 und 5. Hierfür müssen die Trainingsläufe für die verschiedenen Netze nur auf unterschiedlichen Rechnern gestartet werden und können so parallel berechnet werden.
Ein Beispiel für diese Parallelität ist in Abbildung 4.1 zu sehen. Hier werden auf vier Prozessoren (P_1 bis P_4) einzelne, sehr verschiedene Netze mit unterschiedlich großen Mustermengen trainiert. Falls das Training für diese Netze annähernd gleich lang dauert, beträgt die Beschleunigung S_4 in diesem Beispiel also ungefähr vier.

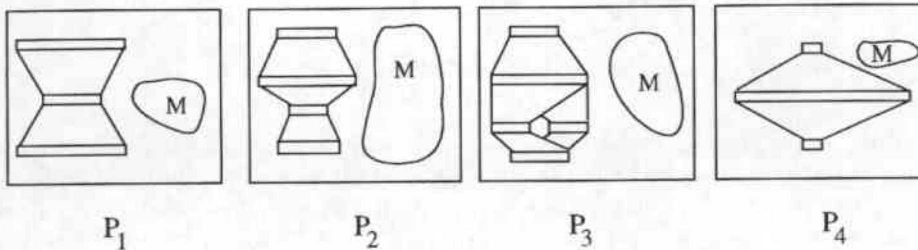


Abbildung 4.1: Netzparallelität

4.3.2 Mustersatzparallelität (*pattern parallelism, training set parallelism*)

Sobald Blocklernen oder Epochenlernen als Lernmethode benutzt werden soll, können die Präsentationen der einzelnen Muster der Mustermenge vollkommen getrennt voneinander durchgeführt werden. Die für die einzelnen Muster errechneten Gewichtsänderungen müssen erst beim Gewichtsveränderungsschritt für den Block respektive die Epoche wieder zusammengeführt werden.

Mögliche Beschleunigung:

Es seien

m	:	die Größe der Mustermenge
s	:	die Anzahl der jedem Prozessor zu präsentierenden Muster
t_1	:	die Zeit für die Präsentation eines Musters
t_2	:	die Zeit zur Zusammenführung der einzelnen Änderungswerte, der Durchführung der Gewichtsveränderung und der Verteilung der sich neu ergebenden Gewichte

Desweiteren seien Epochenlernen und eine hinreichend große Menge an Prozessoren angenommen. Dann ist:

$$T_n = s * t_1 + t_2$$

$$S_n = \frac{(t_1 * m)}{(st_1 + t_2)}$$

Da die zur Präsentation eines Muster benötigte Zeit zumeist kleiner ist als die Zeit zur Aufsummierung, muß hier eine sinnvolle Anzahl an Prozessoren bestimmt werden, die jeder für sich eine gewisse Menge an Eingangsmustern präsentieren und dann ihre Änderungs-Werte entsprechend untereinander aufsummieren. Dieses Aufsummieren kann abhängig von der Kommunikationsstruktur der Recheneinheiten des zugrundeliegenden Rechners linear (Ring), proportional zur Wurzel (Feld) oder logarithmisch (Baum) in der Anzahl der Prozessoren erfolgen. Etwas spezieller formuliert sehen die Zeitabschätzungen folgendermaßen aus:

$$T_n = s * t_1 + t_{zusammenf\ddot{u}hrung}(\tfrac{m}{s})$$

$$S_n = \frac{(t_1 * m)}{(s * t_1 + t_{zusammenf\ddot{u}hrung}(\frac{m}{s}))}$$

Für die drei möglichen Arten der Zusammenführung der einzelnen Änderungswerte ergeben sich dann die folgenden Zeiten und optimalen Teilmengengrößen:

$$\begin{array}{llll} \text{Ring:} & T_n = s*t_1 + t_2\frac{m}{s} & \Rightarrow & s = \sqrt{\frac{mt_2}{t_1}} \\ \text{Feld:} & T_n = s*t_1 + t_2\sqrt{\frac{m}{s}} & \Rightarrow & s = (\frac{\sqrt{m}t_2}{2t_1})^{\frac{2}{3}} \\ \text{Baum:} & T_n = s*t_1 + t_2\log(\frac{m}{s}) & \Rightarrow & s = \frac{t_2}{t_1} \end{array}$$

Da m in praktischen Anwendungen recht groß sein kann ist, sind die spektakulärsten in der Literatur besprochenen Beschleunigungen auf diese Art und Weise erzielt worden [Sin90a] [WZ90]. Fraglich ist hierbei allerdings, inwieweit das Lernverfahren an sich beschleunigt wird, da durch das Verwenden von Epochenlernen gegenüber Musterlernen eine insgesamt höhere Anzahl an Eingangsmusterpräsentationen notwendig ist. Zur Lösung dieser Problematik bietet sich der Einsatz von Blocklernen an. Hierfür kann dann über eine Abschätzung des Lernverhaltens bezüglich der Blockgröße die für eine maximale Beschleunigung des gesamten Lernverfahrens optimale Anzahl an Prozessoren bestimmt werden [PM92].
Ein Beispiel für die Mustersatzparallelität ist in Abbildung 4.2 zu sehen. Hier werden von acht Prozessoren (P_1 bis P_8) gleichzeitig die Änderungswerte für jeweils ungefähr ein Achtel der Mustermenge berechnet. Die Zusammenführung der Änderungswerte der einzelnen Teilmengen zum Abschluß einer Epoche erfolgt über einen Baum. Auf diesem Wege werden auch die neuen Gewichte nach dem Änderungsschritt für eine neue Epoche wieder verteilt.

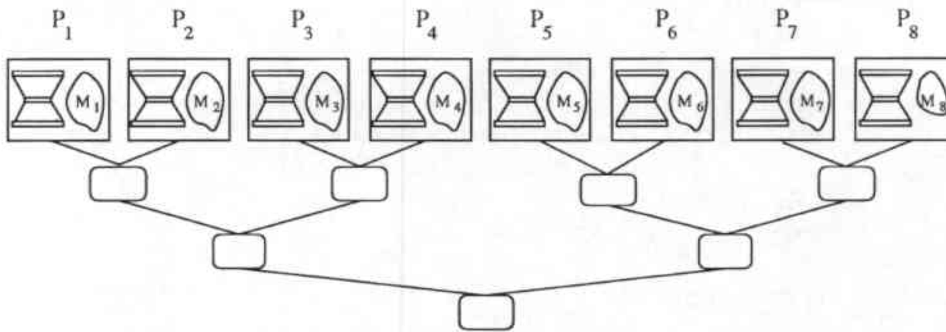


Abbildung 4.2: Mustersatzparallelität mit logarithmischer Summierung

4.3.3 Schichtparallelität[1]

Da viele Netze aus verschiedenen Schichten aufgebaut sind, kann die Berechnung der Aktivierungs- und rückpropagierten Fehlervektoren der einzelnen Schichten respektive der Gewichtsänderungen getrennt nach den verschiedenen Gewichtsmatrizen vorgenommen werden. Für ein MLP zum Beispiel ergibt sich dabei eine Pipeline, durch die die einzelnen Aktivierungsvektoren fließen. Da ein Muster diese Pipeline schon betritt, bevor das vorhergehende überhaupt die Ausgangsschicht erreicht, bzw. seine Änderungswerte berechnet werden konnten, ist dieses Verfahren nur anwendbar, falls die Änderungswerte über eine gewisse Menge von Eingangsmustern summiert werden sollen (Block-/ Epochenlernen) oder die Gewichte gar nicht geändert, sondern nur die Ausgaben des Netzes für eine Menge von Eingabemustern berechnet werden sollen.

[1] Die in der Literatur [Sin90b] erwähnte Lernphasenparallelität, d.h. die Parallelität von Vorwärts und Rückwärtsschritt im Lernverfahren, ist eine niedrigere Form von Schichtparallelität und soll hierunter subsumiert werden.

Mögliche Beschleunigung:

Sei t_{1k} die Zeit, die benötigt wird, um die Aktivierungen von Schicht k zu berechnen und t_{2k} die Zeit, die benötigt wird, um die Fehlerwerte von Schicht k sowie die Gewichtsänderungswerte der adjazenten Neuronen zu berechnen. Unter der Annahme von Epochenlernen ergibt sich:

$$T_n = \max(\max_k t_{1k}, \max_k t_{2k})$$

$$S_n = \frac{\sum_k t_{1k}+t_{2k}}{\max(\max_k t_{1k}, \max_k t_{2k})}$$

Die maximal erreichbare Beschleunigung mit dieser Technik für MLPs mit 2 bis 4 Schichten gleicher Größe liegt damit bei ca. 1 bis 3 beim Berechnen von Netzausgaben und bei 2 bis 5 für das Training. Für TDNNs ergeben sich hierbei aufgrund der hohen Anzahl an Gewichtsmatrizen relativ hohe Werte: Für ein dreischichtiges TDNN mit nahezu gleichgroßen Schichten und fünf Time-Delays ca. 10 für die Netzausgabeberechnung und 15 für das Training. In Abbildung 4.3 ist eine mögliche Nutzung dieser Parallelitätsebene für die Berechnung eines TDNN mit drei Time-Delays dargestellt. Hierfür werden die sechs Gewichtsmatrizen des TDNNs auf sechs unterschiedliche Prozessoren (P_1 bis P_6) verteilt. Zur Berechnung der Ausgabevektoren dieses TDNNs werden die Eingabevektoren der Eingabemenge nacheinander in die unteren Schicht dieser Prozessor-Pipeline eingespeist. Zu jedem Zeitschritt der Pipeline wechseln dann die auf den Prozessoren berechneten Aktivierungsvektoren zum nachfolgenden Prozessor, um dort weiterverarbeitet zu werden. Auf diese Weise wird die Berechnung des TDNNs ungefähr um den Faktor sechs beschleunigt.

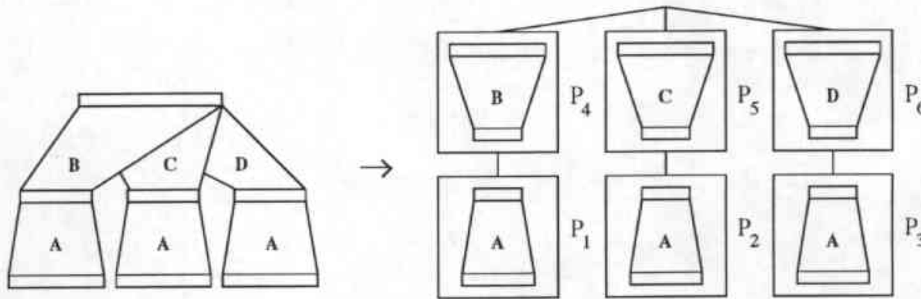


Abbildung 4.3: Schichtparallelität, 1 Schicht pro Prozessor

4.3.4 Neuronenparallelität

Im Prinzip können in neuronalen Netzen alle Neuronen einer Schicht gleichzeitig berechnet werden. Um dies zu unterstützen, kann jedem Neuron ein Prozessor oder auch mehreren Neuronen ein Prozessor zugewiesen werden. Diese Art ein neuronales Netz aufzuteilen wird *Vertical slicing* genannt (im Gegensatz zur Schichtparallelität, dem *Horizontal slicing* eines Netzes) und ist wohl die meistverwandte Methode neben der Mustermengenparallelität. Ein Beispiel für die Verteilung eines einfachen dreischichtigen Netzes mit einem Neuron pro Prozessor ist in Abbildung 4.4 zu sehen.

Mögliche Beschleunigung:

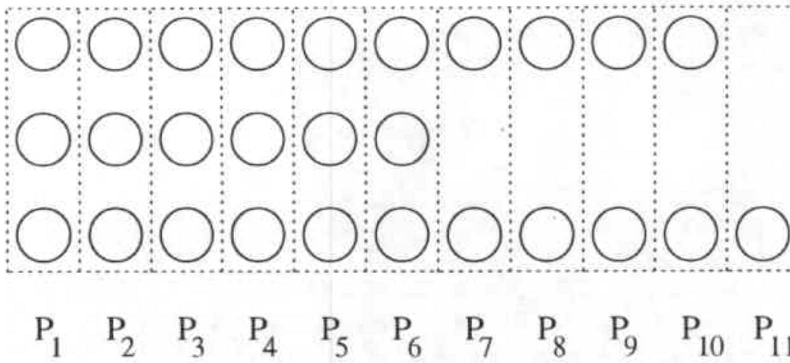


Abbildung 4.4: Neuronenparallelität, 1 Neuron pro Prozessor

Das zu berechnende Netz sei vollverbunden. Es bezeichnen

l	:	die Anzahl der Schichten des Netzes
n_i	:	die Anzahl der Neuronen der Schicht i
d_i	:	die Anzahl der Neuronen in Schicht i pro Prozessor (Neuronendichte)
t_{fwd}	:	die Zeit zur Berechnung einer Verbindung im Vorwärtsschritt
t_{bwd}	:	die Zeit zur Berechnung einer Verbindung im Rückwärtsschritt
t_{updt}	:	die Zeit zur Berechnung einer Verbindung im Gewichtsänderungsschritt

Unter Vernachlässigung der Zeit für die Addierung des Schwellwertes und die Berechnung der Aktivierungsfunktion gilt:

$$T_n = \sum_{i=1}^{l-1} t_{fwd} n_i d_{i+1} + \sum_{i=3}^{l} t_{bwd} n_i d_{i-1} + \sum_{i=1}^{l-1} t_{updt} n_i d_{i+1}$$

$$S_n = \frac{\sum_{i=1}^{l-1} t_{fwd} n_i n_{i+1} + \sum_{i=3}^{l} t_{bwd} n_i n_{i-1} + \sum_{i=1}^{l-1} t_{updt} n_i n_{i+1}}{\sum_{i=1}^{l-1} t_{fwd} n_i d_{i+1} + \sum_{i=3}^{l} t_{bwd} n_i d_{i-1} + \sum_{i=1}^{l-1} t_{updt} n_i d_{i+1}}$$

Diese Art der Parallelisierung benötigt bei sehr kleinem d_i, ein schnelles, gut synchronisiertes Konzept zur Verteilung der auf den einzelnen Prozessoren errechneten Aktivierungen (SIMD), während bei größeren d_i auch grobgranulare Techniken (MIMD) eingesetzt werden können.
Im normalen Backpropagation-Algorithmus z.B. beträgt die Anzahl der Operationen pro Prozessor im Vorwärtsschritt $d_i * n_i$ Multiplikationen und $d_i * n_i$ Additionen, denen n_i Kommunikationsoperationen gegenüberstehen. Für den Fall kleiner d_i bewegt sich also die Anzahl der zu tätigenden Kommunikationsschritte in ungefähr der gleichen Größenordnung wie die Anzahl der Rechenschritte. Hierfür ist eine sehr enge Synchronisation (SIMD) erforderlich, um nicht die Kommunikation zum Flaschenhals für die Berechnung werden zu lassen. Die maximale Beschleunigung für die Schicht-zu-Schicht-Berechnung ergibt sich dann bei entsprechend gut gelöster Kommunikation für 1 Neuron pro Prozessor und beträgt $O(n_i)$. Für den Fall höherer d_i hingegen kann auch ein gröberes Konzept zur Verteilung und entsprechend gröbere Rechnerarchitekturen (MIMD) genommen werden, da die Kommunikationszeiten gegenüber den Rechenzeiten nicht mehr ins Gewicht fallen. Hierfür kann dann bei gegebenen Kommunikations- und Rechengeschwindigkeiten über die Minimierung einfacher Gleichungen die optimale Anzahl an Neuronen pro Prozessor, d.h. die für eine maximale

Beschleunigung notwendige Anzahl an Prozessoren gewonnen werden [BM89]. Die Beschleunigung ergibt sich dann zu $O(n)$

4.3.5 Gewichtsparallelität

Ein Neuron besitzt eine Anzahl von eingehenden Verbindungen, deren Gewichte jeweils mit der Aktivierung eines anderen Neurons verknüpft werden müssen. Da die auf einer Verbindung auszuführenden Operationen unabhängig von denen der anderen Verbindungen sind (Lokalitätseigenschaft des Backpropagationverfahrens), können diese für ein einzelnes Neuron parallel zueinander ausgeführt werden. Zwei Beipiele dieser Parallelitätsform sind in Abbildung 4.5 zu sehen. Hier werden die Multiplikationen der Aktivierungen mit den Gewichten auf jeweils vier verschiedenen Prozessoren vorgenommen. Die Summierung erfolgt dann im linken Bild über die Nachbarschaftskommunikation der Prozessoren. Im rechten Bild wird dafür ein zweistufiger Addierbaum eingesetzt.

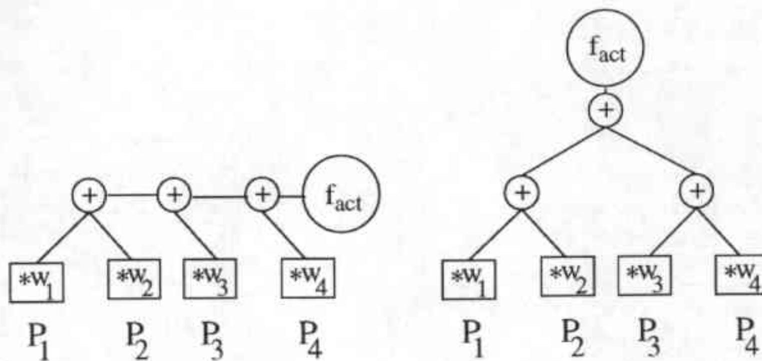


Abbildung 4.5: Gewichtsparallelität

Mögliche Beschleunigung:

Da die Operationen auf den Kanten (mindestens eine Multiplikation) zumeist aufwendiger sind als die Summierung der einzelnen Werte, ergibt sich hierdurch eine Beschleunigung des Lernverfahrens. Nach der Errechnung müssen die einzelnen Werte der Kanten des Neurons im Vorwärtsschritt und im Rückwärtsschritt des Trainings aufsummiert werden. Dies kann, abhängig von der Kommunikationsstruktur des Rechners und der Gewichtsverteilung, auf verschiedene Arten geschehen, im schnellsten Fall aber logarithmisch.

Wie in Abschnitt 4.3.4 sei das Netz auch hier vollverbunden. Es bezeichnen

l	:	die Anzahl der Schichten des Netzes
n_i	:	die Anzahl der Neuronen der Schicht i
t_1	:	die Zeit zur Berechnung einer Verbindung im Vorwärtsschritt (ohne Addition)
t_2	:	die Zeit zur Berechnung einer Verbindung im Rückwärtsschritt (ohne Addition)
t_3	:	die Zeit zur Berechnung einer Verbindung im Veränderungs-Schritt
t_{add}	:	die Zeit zur Berechnung einer Addition
$t_{sum}(n)$	:	die Zeit zur Aufsummierung der einzelnen Kantenwerte zu net_i (z.B. $t_{sum}(n) = n$ oder $t_{sum}(n) = log(n)$)

Die Addierung des Schwellwertes und die Berechnung der Aktivierungsfunktion werden auch hier der Einfachheit wegen vernachlässigt. Es gilt:

$$T_n = \sum_{i=1}^{l-1}(t_1 + t_{sum(n_i)})n_{i+1} + \sum_{i=3}^{l}(t_2 + t_{sum}(n_i))n_{i-1} + \sum_{i=1}^{l-1} t_3 n_{i+1}$$

$$S_n = \frac{\sum_{i=1}^{l-1}(t_1+t_{add})n_i n_{i+1} + \sum_{i=3}^{l}(t_2+t_{add})n_i n_{i-1} + \sum_{i=1}^{l-1} t_3 n_i n_{i+1}}{\sum_{i=1}^{l-1}(t_1+t_{sum}(n_i))n_{i+1} + \sum_{i=3}^{l}(t_2+t_{sum}(n_i))n_{i-1} + \sum_{i=1}^{l-1} t_3 n_{i+1}}$$

Bei Verwendung dieser Parallelität[1] kann auch bei großen Schichten eine Beschleunigung in der Höhe der Schichtgröße nur erreicht werden, wenn die einzelnen Operationen auf den Kanten gegenüber der Addition sehr aufwendig sind und logarithmisch aufsummiert wird. Für das normale Backpropagation-Verfahren trifft dieses nicht zu, da hier die Operation auf den einzelnen Kanten nur eine Multiplikation ist. Folglich ist der Einsatz dieser Parallelisierungsvariante hierfür in Bezug auf Beschleunigung ($S_n \approx O(\frac{n_i}{log(n_i)})$) gegenüber der Neuronenparallelität ($S_n \approx O(n_i)$) nicht gerechtfertigt. In Kombination mit anderen Parallelisierungsvarianten jedoch ergeben sich hier sehr gute Möglichkeiten.

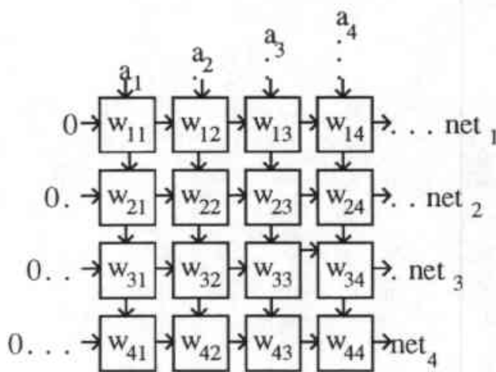


Abbildung 4.6: Ein systolisches Feld mit linearer Verteilung und Summierung

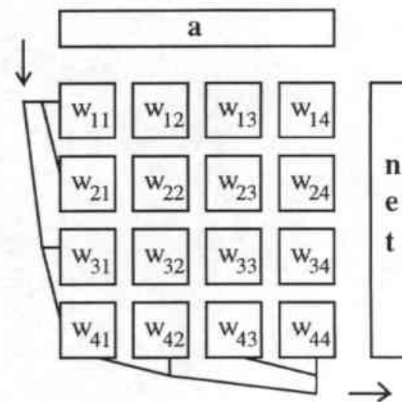


Abbildung 4.7: Ein Feldrechner mit logarithmischer Verteilung und Summierung

In Verbindung mit Neuronenparallelität ist zum Beispiel der Einsatz eines systolischen Feldes [Vir93] oder eines Feldrechners möglich, auf dem die Gewichtsmatrix kanonisch verteilt ist. Im Rechenverfahren wird dann der Aktivierungsvektor der niedrigeren Schicht von oben in die Prozessormatrix eingespeist und auf die Spalten der Prozessormatrix verteilt. Dann wird für alle Gewichte gleichzeitig die Multiplikation mit den jeweiligen Aktivierungen durchgeführt, über die Zeilen aufsummiert und der so berechnete Vektor der Eingabewerte der Neuronen der höheren Schicht ausgegeben. Die Verteilung einer einzelnen Aktivierung in einer Spalte der Prozessormatrix kann hierbei in Abhängigkeit von der zur Verfügung stehenden Kommunikationsstruktur linear

[1] Man beachte: Bei einer Berechnung des Netzes allein mit Gewichtsparallelität werden die einzelnen Neuronen nacheinander berechnet und nur für jedes einzelne die Multiplikation mit den Gewichten parallel vorgenommen.

(Ring), logarithmisch (Baum) oder direkt (Bus) erfolgen. Die Aufsummierung der in einer Zeile der Prozessormatrix berechneten Verbindungwerte ist in linearem (Ring) oder logarithmischem (Baum) Zeitaufwand möglich. Zwei Beispiele hierfür sind in den Abbildungen 4.6 und 4.7) zu sehen. In Abbildung 4.6 ist eine 4x4 Gewichtsmatrix auf ein ebensogroßes systolisches Feld verteilt. Die Berechnung geschieht, wie in systolischen Feldern üblich, durch zeitversetztes Einspeisen des Aktivierungsvektors und der anfänglich auf Null gesetzten Summen der einzelnen Zeilen sowie schrittweisem Weiterreichen der einzelnen Teilsummen nach links und der einzelnen Aktivierungen nach unten. Nach vier Schritten ergibt sich der erste Wert des Summenvektors und nach acht Schritten der letzte Wert an der rechten Seite des Feldes. In Abbildung 4.7 ist dieselbe Matrix auf ein Prozessorfeld verteilt. Hier geschieht die Verteilung der Aktivierungen und die Aufsummierung über die in jeder Zeile und jeder Spalte vorhandenen Bäume.
Im allgemeinen Fall ergibt sich mit $t_{spread}(n)$ ($t_{spread}(n) = n, log(n)$ oder 1 für Ring, Baum oder Bus) als Zeit zur Verteilung der Aktivierungen folgendes:

$$T_n = \sum_{i=1}^{n-1}(t_1 + t_{sum}(n_i) + t_{sp.}(n_{i+1})) + \sum_{i=3}^{n}(t_2 + t_{sum}(n_i) + t_{sp.}(n_{i-1})) + \sum_{i=1}^{n-1}(t_3 + t_{sp.}(n_i) + t_{sp.}(n_{i+1})$$

$$S_n = \frac{\sum_{i=1}^{n-1}(t_1+t_{add})n_i n_{i+1} + \sum_{i=3}^{n}(t_2+t_{add})n_i n_{i-1} + \sum_{i=1}^{n-1} t_3 n_i n_{i+1}}{\sum_{i=1}^{n-1}(t_1+t_{sum}(n_i)+t_{sp.}(n_{i+1})) + \sum_{i=3}^{n}(t_2+t_{sum}(n_i)t_{sp.}(n_{i-1})) + \sum_{i=1}^{n-1}(t_3+t_{sp.}(n_i)+t_{sp.}(n_{i+1}))}$$

Für $t_{spread}(n) = 1$ und $t_{sum} = log(n)$ ist dies die wohl schnellste Möglichkeit zur Schicht-zu-Schicht Berechnung in neuronalen Netzen. Die hierbei erreichte Beschleunigung beträgt ungefähr $O(\frac{n_{i-1}n_i}{log(n_i)})$.

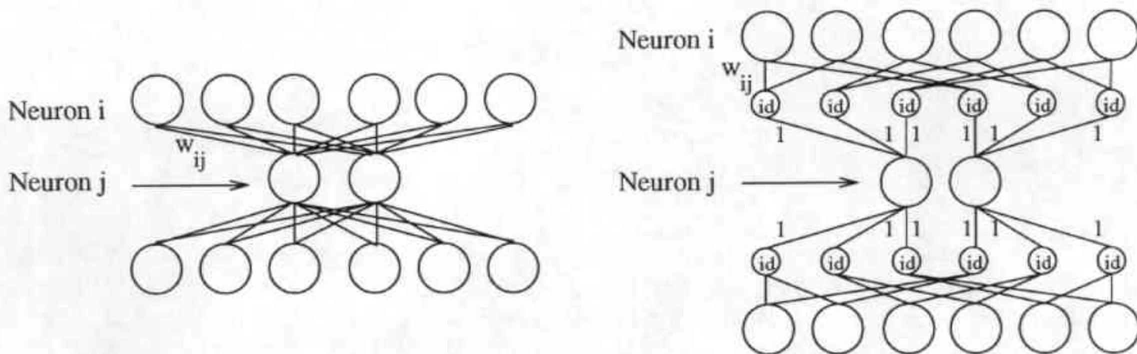


Abbildung 4.8: Neuronensplitting

Eine mildere Form dieser Parallelitätsvariante ist das sogenannte *Neuronensplitting*. Hierbei werden die einzelnen Neuronen sehr kleiner Schichten in mehrere Teilneuronen mit $f_{act} = id$ zerlegt und dann das Netz mittels Neuronenparallelität berechnet. Auf diese Weise werden mehrere Verbindungen desselben Neurons gleichzeitig berechnet und somit de facto Gewichtsparallelität durch Neuronenparallelität erreicht. Der Vorteil dieses Verfahrens liegt in der besseren Verteilung der Verbindungen auf den Rechner. Dieses führt zum einen zu einem geringeren Speicherbedarf pro Prozessor, was auf Parallelrechnern, die nur einen geringen Speicher pro Prozessor besitzen, von großem Nutzen sein kann. Zum anderen ist unter gewissen Bedingungen auf diese Weise eine Erhöhung der Berechnungsgeschwindigkeit zu erreichen, da gleichzeitig mehrere Prozessoren an

den Verbindungen der Neuronen der kleinen Schichten rechnen [Kol94]. Ein Beispiel ist in Abbildung 4.8 zu sehen. Hierbei werden die beiden Neuronen der versteckten Schicht in jeweils acht kleinere Neuronen zerlegt. Wenn in einer praktischen Implementierung die Neuronen mit jeweils einem Neuron pro Prozessor abgelegt werden und die Gewichte jeweils bei den Prozessoren der höheren Schicht gespeichert sind, sind für die inneren beiden Prozessoren statt 8 (=6+2) Verbindungen nur noch 4 Verbindungen gespeichert (Die mit Eins gewichteten Verbindungen müssen nicht explizit gespeichert werden). Für die äußeren Prozessoren hingegen sind statt der vorherigen zwei Verbindungen jetzt 4 Verbindungen abgelegt. Auf diese Weise sind die Gewichte homogener verteilt und benötigen im Maximalfall nur noch 4 statt 8 Speicherplätze.

Eine weitere Möglichkeit die verschiedenen Parallelitätsebenen zu nutzen besteht darin, das komplette Berechnungsschema für das Netz als Datenflußgraphen darzustellen und diesen dann von einem Datenflußrechner berechnen zu lassen [Smo89]. Die maximale Parallelität für das Netz ist damit direkt durch das mit dem Graphen formulierte Berechnungsschema gegeben und die Beschleunigung unterhalb dieser Grenze direkt durch die Anzahl der Prozessoren des Datenflußrechners bestimmt.

4.4 Parallelisierung in realen Systemen

In der folgenden Tabelle, Tabelle 4.1, sind die Einsatzmöglichkeiten der einzelnen Parallelisierungsmöglichkeiten für die unterschiedlichen Aufgabenstellungen dargestellt:

	Parallelisierungsebenen				
	Netz	Mustersatz	Schicht	Neuronen	Gewichte
Aufgabe 1		x	x	x	x
Aufgabe 2		(x)	(x)	x	x
Aufgabe 3	x	(x)	(x)	x	x
Aufgabe 4	x	(x)	(x)	x	x
Aufgabe 5	x	(x)	(x)	x	x

Tabelle 4.1: Nutzbarkeit der Parallelitätsebenen für die 5 Aufgabenstellungen

Der Einsatz von Mustersatz oder Schichtparallelität ist, wie schon geschildert, vom jeweils gewählten Lernverfahren (Musterlernen oder Block-/Epochenlernen) abhängig und somit nur bedingt möglich. In der Tabelle ist diese eingeschränkte Verwendbarkeit durch Klammerung symbolisiert. Da die beschriebenen Parallelisierungsebenen vollkommen unabhängig voneinander und frei kombinierbar sind, können aufgrund dieser Tabelle nun für einzelne Aufgabenstellungen direkt die verwendbaren Parallelitäten abgelesen werden.

Im Falle von Aufgabenstellung 3 mit Epochenlernen wäre zum Beipiel eine Implementierung folgender Art denkbar: Da Neuronen- und Gewichtsparallelität nutzbar sind, wäre eine Schicht-zu-Schicht Verbindung mittels eines systolischen Feldes berechenbar. Aufgrund der ebenso nutzbaren Schichtparallelität wäre dann ein neuronales Netz über eine Pipeline von mehreren systolischen Feldern, jedes für jeweils eine Gewichtsmatrix, berechenbar. Durch Ausnutzung der Mustersatzparallelität wäre dann eine Kopplung sehr vieler Rechner dieser Art über einen Baum möglich, der die Aufsummierung der Gewichtsänderungen und die Verteilung der neuen Gewichte vornimmt. Da auch mehrere Netze zu trainieren sind und somit Netzparallelität nutzbar ist, wäre schließlich eine Erledigung der Gesamtaufgabe mit mehreren Systemen dieser Art denkbar.

In der nächsten Tabelle, Tabelle 4.2, ist die Nutzbarkeit der verschiedenen Parallelitäten auf den zwei großen Klassen von Parallelrechnern dargestellt:

	Parallelisierungsebenen				
	Netz	Mustersatz	Schicht	Neuronen	Gewichte
SIMD	(x)	x	x	x	x
MIMD	x	x	x	(x)	-

Tabelle 4.2: Nutzbarkeit der Parallelitätsebenen für SIMD- und MIMD-Rechner

Netzparallelität kann auf SIMD-Rechnern nur bei Verwendung sehr ähnlich strukturierter Netze verwendet werden, da sonst ein zu großer Teil der Prozessoren während der Berechnungen für Netze anderer Struktur brachliegt (Diese Einschränkung ist in der Tabelle durch Klammerung gekennzeichnet). Auf MIMD-Rechnern ist sie ohne diese Einschränkung nutzbar. Neuronenparallelität hingegen ist auf SIMD-Rechnern sehr einfach wegen der vielen gleich strukturierten Neuronen einer Schicht zu verwenden, während sie sich auf MIMD-Rechnern nur bei groben *Vertical slicing* lohnt. Eine Verteilung von einem oder zwei Neuronen pro Prozessor bringt hier zusätzlichen Abstimmungsaufwand mit sich, der sich nur schwer durch die gewonnene Leistung wieder wettmachen läßt (siehe Abschnitt 4.3.4).
Auf SIMD-Rechnern können folglich nur die Nebenläufigkeit vieler ähnlicher Einheiten (Neuronen, Gewichte, Schichten) ohne größere Leistungseinbußen ausgenutzt werden, da zwar die die feingranulare Abstimmung durch das SIMD-Prinzip unterstützt wird, aber eine weitgehende Entkopplung der Prozessoren (und damit die Möglichkeit zur Berechnung auch nicht ähnlicher Netze) wegen des allen Prozessoren gemeinsamen Befehlsstromes nicht möglich ist. Auf MIMD-Rechnern hingegen können zwar ohne Probleme grob granulare Parallelitäten verwirklicht werden, aber eine feinere Abstimmung bringt große Leistungsverluste aufgrund des hohen benötigten Kommunikations- und Abstimmungsaufwandes mit sich.

	Rechner	Leistung (MCPS/MCUPS)	Aufgaben stellung	Parallelitätsebenen Netz	Muster	Neuronen	Gewicht
Wahle	HP 9000	14.1/6.3	1,2				
Wahle	10 DEC 3000	87/36	5	Nz			
[EMS90]	4 T800	k.A./0.3	2			Ne	
[ZMMV93]	Paragon	k.A./36	2		M		
[NF89]	DAP610	k.A./100-160	2			Ne	Ge
[NF89]	DAP610	k.A./25-40	2			Ne	Ge
[Kol94]	MP1	k.A./68	5	Nz	M	Ne	Ge
[Mac92]	MP1	348/129	1,2		M	Ne	
[ZMMV93]	MP2	972/360	1,2		M	Ne	
[Sin90a]	CM-2	1300/k.A.	2		M		
[ZMMW89]	CM-2	182/40	2		M	Ne	
[McC90]	CNAPS	9700/2370	1,2			Ne	
Wahle	CNAPS	k.A./126	2			Ne	
Wahle	CNAPS	k.A./75	3			Ne	
[SSA+93]	MY-Neupower	12800/1260	1,2			Ne	
[Vir93]	Mantra	400/133	1,2			Ne	Ge
[MBK+92a]	RAP	558/102	1,2			Ne	
[MBK+92b]	MUSIC-20	515/246	1,2			Ne	
[WZ90]	GF11	k.A./901	2		M		
[RRA+93]	Synapse	5100/k.A.	1,2			Ne	

Tabelle 4.3: Simulatoren neuronaler Netze: Leistung und genutzte Parallelitätsformen

In Tabelle 4.3 sind nun ein paar Leistungsdaten von bisher implementierten parallelen Simulatoren für Neuronale Netze mit den in ihnen angestrebten Zielen und verwendeten Parallelitäten dargestellt. Die verwendeten Parallelitäten sind innerhalb der Tabelle zur schnelleren Lesbarkeit mit

	Prozessoren	Anordnung	Gewichtsdarstellung
Mantra	40x40	systolisches Feld	33 Bit float
RAP	40 DSP	Ring	float
MUSIC-20	60 DSP	Ring	float
CNAPS	512	Ring/Bus	16(32) Bit / 32 Bit Akku
MY-Neupower	512	Bus	16 Bit
MasPar MP1	128x128	xnet	32 Bit float
CM2	2^{16} 1bit	Hyperkubus	32 Bit float
GF11	511	Benes-Netzwerk	float
Paragon	66 i860XP	Matrix	float
Synapse	2*(4*4)	systolisches Feld	16(32) Bit / 48 Bit Akku
DAP610	64x64, 1Bit	Matrix	8 Bit / 16 Bit

Tabelle 4.4: Parallelrechnerarchitekturen im Überblick

Kürzeln gekennzeichnet. In Tabelle 4.4 sind in einer kurzen Übersicht ein paar wichtige Daten der jeweils benutzten Rechnertypen zusammengestellt.
Die hier angegebenen Leistungsdaten sind mit extremer Vorsicht zu genießen, da sie in vielen Fällen auf Spezialnetzen und Spezialproblemstellungen gemessen wurden, die nicht der Realität entnommen wurden. Vielmehr sind diese häufig auf die jeweiligen Verfahren zugeschnitten, um eine möglichst hohe Leistung veröffentlichen zu könnnen. Die Leistungsangabe des SNNS (Stuttgarter Neuronale Netze Simulator) [ZMMV93][Mac92] wurde zum Beispiel auf einem 128x128x128 Netz gemessen, das in 128 Kopien auf der MasPar verteilt war und auf einer genügend großen Trainingsmenge mit Epochenlernen trainiert wurde, so daß die Vereinigung der Gewichtsänderungswerte der einzelnen Netzkopien gegenüber der Zeit zum Berechnen der Änderungswerte nicht ins Gewicht fiel. Die Leistung von Singer [Sin90a] ist ebenfalls auf der Basis eines 128-128-128 Netz errechnet worden, wobei allerdings noch zusätzlich von einer Trainingsmenge mit 65536 Trainingsmustern ausgegangen wurde (eine CM-2 hat genau 2^{16} Prozessoren). Die Maximalleistung der CNAPS [McC90] wurde mit 512 Prozessoren auf einem 1900-500-12 Netz erreicht. Für den verwendeten Algorithmus sind bei diesem speziellen Netz nahezu alle Prozessoren permanent mit Gewichtsberechnungen und Veränderungen beschäftigt. Die Anzahl der Gewichte pro Prozessor erreicht hierbei 1900+12 (= 3824 Byte) und stößt an die Grenze der Prozessorspeichers von 4KB. Da diese Daten nur in sehr unzureichendem Maße die Leitungsdaten/-grenzen (Netzgrößen, Netzarchitekturen, Berechnungsgeschwindigkeit, Trainingsgeschwindigkeit, Lernalgorithmen, Mustermengengrößen) aufzeigen, wäre eine andere Vorgehensweise bei der Bewertung der verschiedenen Verfahren auf den einzelnen Rechnerarchitekturen sicher von Vorteil. Eine praktischere Herangehensweise wäre zum Beispiel einen Satz verschiedener Problemstellungen, Netze, Lernverfahren und Mustermengengrößen anzunehmen, sie vollständing miteinander zu kombinieren und die so entstehende Menge von Tests als Benchmark für die einzelnen Verfahren zu nehmen. Auf diese Weise wäre es wesentlich einfacher möglich, einen Rechner und die Implementierungen in Hinblick auf Ausstattung, Leistungsfähigkeit und vor allem bezüglich der Grenzen zu beurteilen.

5. Zwei parallele Simulatoren für neuronale Netze

5.1 Einleitung

In diesem Kapitel sollen zwei Methoden zur Berechnung neuronaler Netze vorgestellt werden. Hierfür werden zuerst der angestrebte Leistungsumfang des zu entwickelnden Simulators umrissen und noch einmal die gegebenen Hardwareeigenschaften der CNAPS aus Kapitel 2 aufgelistet. Mithilfe der in Kapitel 4 entwickelten Parallelisierungsebenen und ihrer Verwendbarkeit für unterschiedliche Aufgaben und Parallelrechnerarchitekturen sollen dann die hier nutzbaren Parallelitäten ermittelt werden. Danach werden die beiden auf Basis dieser Parallelitäten und der gegebenen Hardwarearchitektur implementierten Simulatoren für neuronale Netze vorgestellt. Ein besonderes Augenmerk soll hier den beiden wichtigen Leistungsdaten der Algorithmen gelten: Der Größen der berechenbaren Netze und den Trainingszeiten. Zum Abschluß sollen die beiden Methoden einander gegenübergestellt und in einem direkten Vergleich die jeweiligen Vor- und Nachteile noch einmal detailliert erläutert werden.

5.2 Aufgabenstellung

5.2.1 Zielsetzung

Der in dieser Arbeit zu entwickelnde Simulator für neuronale Netze auf dem CNAPS-Neurocomputer sollte folgende Leistungsmerkmale aufweisen:

- möglichst hohe Trainingsgeschwindigkeit
- Verarbeitung möglichst großer Netze
- Aufgabenstellungen:
  - Training eines einzelnen Netzes (Aufgabenstellung 2)
  - Training mehrerer leicht unterschiedlicher Netze (Aufgabenstellung 3)
- Netztypen: vollverbundene MLPs und TDNNs beliebiger Größen, wahlweise mit schichtübergreifenden Verbindungen (*shortcuts*) zwischen Eingabe- und Ausgabeschicht
- Lernverfahren: Standard-Backpropagation mit Skalierung der Gewichtsänderung durch eine Lernrate
- leichte Änderbarkeit bezüglich der Fehlerfunktion
- variable Aktivierungsfunktion

- Funktionsapproximation und Klassifikation
- automatische Verteilung des Netzes auf den Rechner

Inbesondere da

- insgesamt nur 2 MB Speicher zur Verfügung stehen und das Abspeichern der Gewichtsänderungswerte oder anderer zusätzlicher Daten für jede Verbindung nur zusätzlichen Speicher beanspruchen und damit die Größe der verarbeitbaren Netze drastisch verringern würde
- Probleme bei der Akkumulation aller Gewichtsänderungen einer Epoche in der Festkomma-Darstellung entstehen würden
- keine hardwaremäßigen Divisionen möglich sind

wurde auf kompliziertere und speicherintensive Varianten des Backpropagation-Lernverfahrens, wie sie in Kapitel 1 genannt wurden, verzichtet.

5.2.2 Eigenschaften der Hardware

In der folgenden Liste sind noch einmal die für die folgenden Implementierungen wichtigsten Merkmale der CNAPS aufgeführt:

- SIMD-Rechner
- Schneller globaler Bus
- Langsame Nachbarschafts-Kommunikation
- 4KB Speicher pro Prozessor
- 64K Befehlsworte Programmspeicher
- 32MB Dateispeicher mit schnellem Buszugriff (1 Byte pro Zyklus)
- Zahlenverarbeitung nur im Festkommaformat

5.2.3 Verwendbare Parallelitätsebenen

Um einen Simulator für neuronale Netze auf einem Parallelrechner zu entwerfen, ist es nötig, die bei gegebener Aufgabenstellung und Hardwarearchitektur nutzbaren Parallelitätsebenen zu kennen. In den soeben formulierten Anforderungen sind als Architektur ein SIMD-Rechner und als Problemstellungen die Aufgaben Nr.2 und Nr.3 vorgegeben. Die in Kapitel 5 hierfür entwickelten Möglichkeiten zur Parallelisierung sind in der folgenden Tabelle aufgeführt:

	verwendbare Parallelitätsebenen				
	Netz	Mustersatz	Schicht	Neuronen	Gewichte
SIMD -Rechner	(x)	x	x	x	x
Aufgabe 2		(x)	(x)	x	x
Aufgabe 3	x	(x)	(x)	x	x

Vereinigt man die verwendbaren Parallelitätsebenen von Aufgabenstellung und Rechner ergibt sich:

	verwendbare Parallelitätsebenen				
	Netz	Mustersatz	Schicht	Neuronen	Gewichte
SIMD + Aufgabe 2		(x)	(x)	x	x
SIMD + Aufgabe 3	(x)	(x)	(x)	x	x

Die in dieser Tabelle vorhandenen Einschränkungen bedürfen noch einer näheren Betrachtung: Die effiziente Verwendung von Netzparallelität bei SIMD-Rechnern war in Kapitel 2 abhängig gemacht worden von der Ähnlichkeit der zu trainierenden Netze. Da Aufgabenstellung 3 diese Ähnlichkeit beinhaltet, kann diese Parallelitätsform genutzt werden.
Der Einsatz von Mustersatzparallelität und Schichtparallelität hing in Kapitel 2 von der Verwendung von Epochen-/Blocklernen oder Musterlernen ab. Da aus Speichermangel bei großen Netzen beide Lernmethoden nicht in Frage kommen, fallen diese Parallelisierungsebenen hingegen weg. Zudem entfällt diese Parallelisierungsmöglichkeit auch bei einem Training von kleinen Netzen, die wenig Speicher benötigen, da die Summierung der Gewichtsänderungen den Transport der kompletten Gewichtsdaten der einzelnen Netze innerhalb des Rechners erfordert. Dieses ist jedoch mittels der mageren CNAPS-Kommunikationsstrukturen (Bus oder 2-Bit Nachbarschaftsverbindungen) nicht in akzeptabler Zeit zu bewerkstelligen.
Es verbleiben also folgende Parallelisierungsmöglichkeiten für den zu entwerfenden Simulator:

	verwendbare Parallelitätsebenen				
	Netz	Mustersatz	Schicht	Neuronen	Gewichte
SIMD + Aufgabe 2				x	x
SIMD + Aufgabe 3	x			x	x

Hiermit sind nun die durch die Aufgabenstellung, die Hardware und die verwendbaren Parallelitätseben gegebenen Rahmenbedingungen für die zu erstellenden Algorithmen abgesteckt. Im folgenden werden die beiden in dieser Arbeit entwickelten und implementierten Verfahren zur Berechnung neuronaler Netze vorgestellt, erläutert und analysiert. Dabei soll wie folgt vorgegangen werden: Zuerst wird jeweils eine Kurzübersicht über die wesentlichen Punkte des Algorithmus gegeben. Danach wird die dem Algorithmus zugrundeliegende Verteilung der Neuronen auf die einzelnen Prozessoren anhand eines Beispiels dargestellt und darauf aufbauend die Funktionsweise des Algorithmus erläutert. Anschließend werden jeweilig die maximalen Größen berechenbarer Netze ermittelt und detaillierte Laufzeitmessungen und Leistungsanalysen vorgenommen. Zum Abschluß werden dann die für TDNNs notwendigen Erweiterungen erläutert und kurz die softwaretechnische Realisierung des jeweiligen Algorithmus vorgestellt.

5.3 Neuronaler Netz Algorithmus I

5.3.1 Kurzübersicht

Zielsetzung dieses Algorithmus

- Training eines einzelnen neuronalen Netzes (Aufgabenstellung 2)

Gewählter Lösungsweg

- Verwendete Parallelitäten: Neuronenparallelität
- Verteilung der Neuronen: *Vertical Slicing* mit einem Neuron pro Prozessor
- Verteilung der Gewichte: Gewicht ω_{ij} liegt bei Neuron i
- Berechnungsschema: Aktivierungsvektor-Gewichtsmatrix-Multiplikation mittels *broadcasting* der Koeffizienten des Aktivierungsvektors über globalen Bus

5.3.2 Verteilung der Neuronen

Die Verteilung eines Netzes bei *Vertical Slicing* mit einem Neuron pro Prozessor ist in Abbildung 5.1 dargestellt. In diesem Beispiel wurden die 6 Eingabeneuronen (i_j), die 4 versteckten Neuronen (h_j) und die 7 Ausgabeneuronen (o_j) eines 6-4-7 Netzes auf zehn Prozessoren (P_k), beginnend mit Prozessor 0, verteilt.

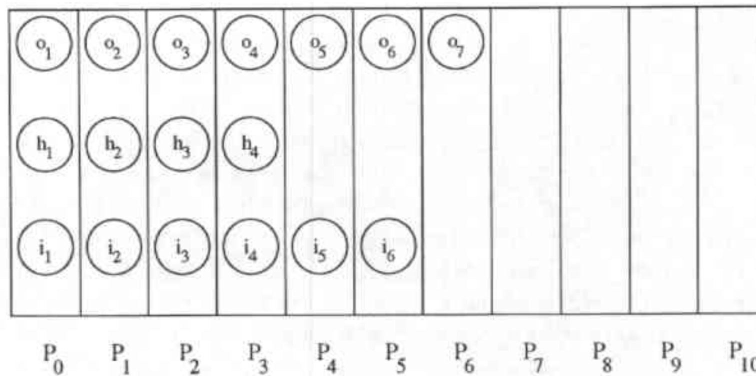
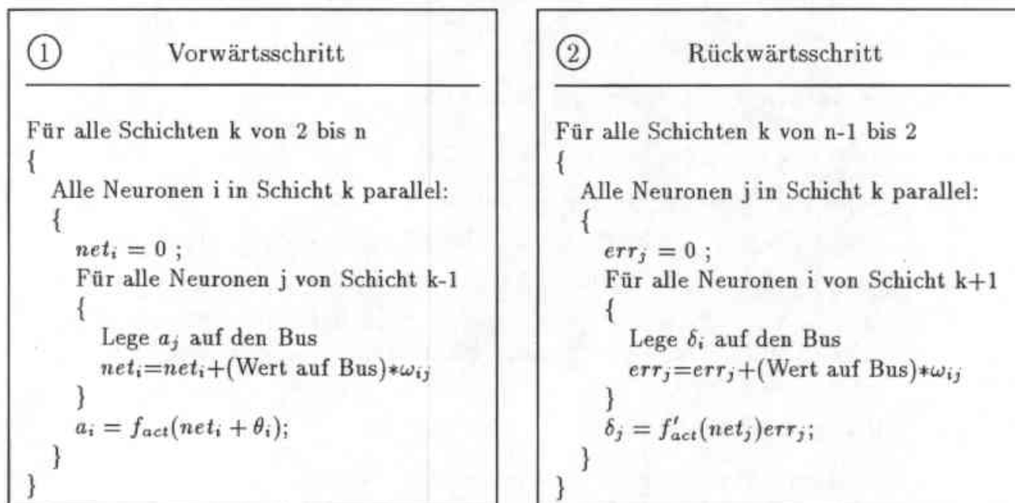


Abbildung 5.1: Verteilung der Neuronen bei *Vertical Slicing* mit einem Neuron pro Prozessor

5.3.3 Detailerläuterung des Algorithmus

Der zeitlich größte Aufwand des *Backpropagation*-Algorithmus besteht in der Multiplikation der Gewichtsmatrizen mit den Aktivierungsvektoren (Vorwärtsschritt) und den Fehlervektoren (Rückwärtsschritt), sowie der Änderung der Gewichtsmatrizen (Änderungs-Schritt), die von den notwendigen Datenflüssen ebenso einer Vektor-Matrix-Multiplikation gleicht. In diesem Algorithmus wird die Multiplikation über *broadcasting* der Aktivierungsvektoren über den Bus durchgeführt, wobei die einzelnen Zeilen der Gewichtsmatrix auf jeweils einem Prozessor liegen.
Die Berechnung der einzelnen Werte in den 3 Schritten des *Backpropagation*-Algorithmus geschieht dann wie in den Darstellungen 5.1 und 5.2.
Durch das in den Algorithmen verwendete „Alle Neuronen x in Schicht k parallel:"-Konstrukt wird ein Index x als parallel ausgezeichnet. Alle Variablen, die innerhalb der Klammerung des „Alle



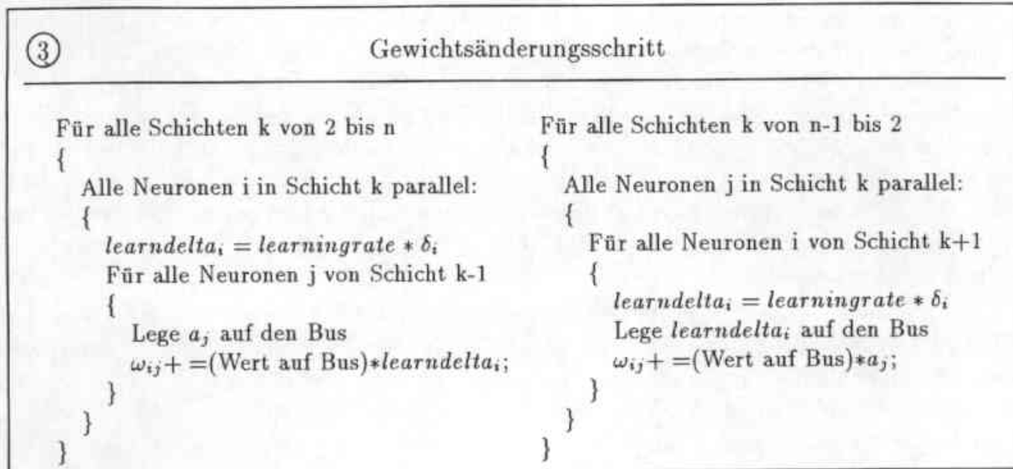
① Vorwärtsschritt

Für alle Schichten k von 2 bis n
{
  Alle Neuronen i in Schicht k parallel:
  {
    $net_i = 0$;
    Für alle Neuronen j von Schicht k-1
    {
      Lege a_j auf den Bus
      $net_i = net_i + (\text{Wert auf Bus}) * \omega_{ij}$
    }
    $a_i = f_{act}(net_i + \theta_i)$;
  }
}

② Rückwärtsschritt

Für alle Schichten k von n-1 bis 2
{
  Alle Neuronen j in Schicht k parallel:
  {
    $err_j = 0$;
    Für alle Neuronen i von Schicht k+1
    {
      Lege δ_i auf den Bus
      $err_j = err_j + (\text{Wert auf Bus}) * \omega_{ij}$
    }
    $\delta_j = f'_{act}(net_j) err_j$;
  }
}

Algorithmendarstellung 5.1 : Vorwärts- und Rückwärtsschritt von *Backpropagation*-Algorithmus I



③ Gewichtsänderungsschritt

```
Für alle Schichten k von 2 bis n
{
  Alle Neuronen i in Schicht k parallel:
  {
    learndelta_i = learningrate * δ_i
    Für alle Neuronen j von Schicht k-1
    {
      Lege a_j auf den Bus
      ω_ij + =(Wert auf Bus)*learndelta_i;
    }
  }
}
```

```
Für alle Schichten k von n-1 bis 2
{
  Alle Neuronen j in Schicht k parallel:
  {
    Für alle Neuronen i von Schicht k+1
    {
      learndelta_i = learningrate * δ_i
      Lege learndelta_i auf den Bus
      ω_ij + =(Wert auf Bus)*a_j;
    }
  }
}
```

Algorithmendarstellung 5.2 : Gewichtsänderungsschritt von *Backpropagation*-Algorithmus I

...“-Konstruktes diesen Index tragen, liegen dann parallel auf verschiedenen Prozessoren vor und werden auch parallel bearbeitet. So bedeutet zum Beispiel die Zeile „$net_i = 0$“ innerhalb der „Alle...“-Klammerung, daß alle Prozessoren ihre lokale Variable *net* auf Null setzen. Insbesondere sei darauf hingewiesen, daß das „Alle ...“-Konstrukt keine sequentielle Schleife über alle Neuronen darstellt, sondern nur angibt, welche Variablen auf allen Prozessoren vorhanden sind und parallel berechnet werden sollen.

Das Verfahren einer einzelnen Schicht-zu-Schicht Berechnung im Vorwärtsschritt (1) ist noch einmal graphisch in Abbildung 5.2 dargestellt. In diesem Beispiel wird der Vektor der gewichteten Summe der Vorgängeraktivierungen $(net_{h_1}, net_{h_2}, net_{h_3})$ einer versteckten Schicht mit 3 Neuronen

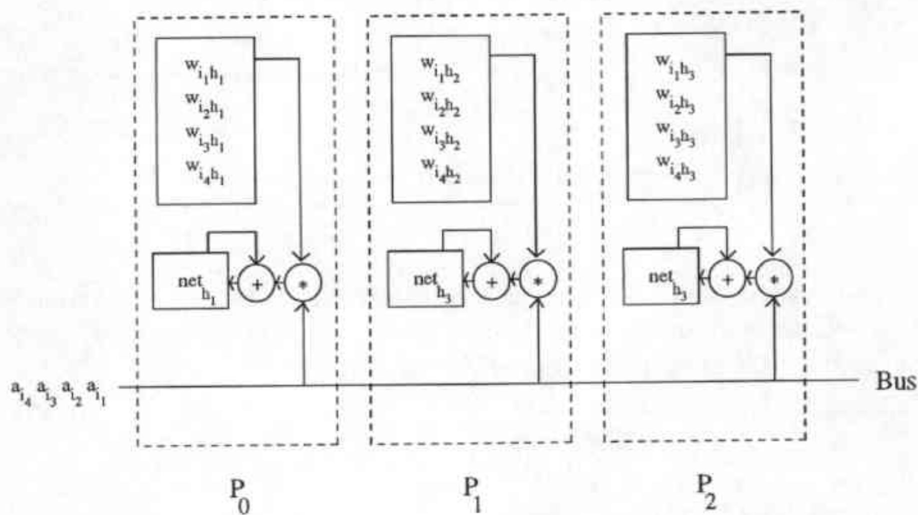


Abbildung 5.2: Die Berechnung des Vorwärtsschrittes

aus den Aktivierungen einer Eingabeschicht mit 4 Neuronen berechnet. Dabei werden die Aktivierungen der Neuronen der Eingabeschicht, wie im Algorithmus zum Vorwärtsschritt (1) angegeben, nacheinander auf den allen Prozessoren zugänglichen IN-Bus gelegt. In jedem Zeitschritt wird dann lokal in jedem Prozessor für das jeweilige versteckte Neuron die auf dem Bus liegende Aktivierung mit dem entsprechenden Gewicht multipliziert und auf die bisher errechnete Teilsumme addiert. Nachdem auf diese Weise alle Aktivierungen der niederen Schicht auf den Bus gelegt worden sind, liegt in jedem Prozessor in der Variablen net_i die Summe aller gewichteten Aktivierungen seiner Vorgängerneuronen vor. Für dieses Vorgehen sind die Gewichte der Verbindungen zu den Vorgängerneuronen, wie aus der Abbildung ersichtlich, in der Reihenfolge in der sie benötigt werden, in einem Feld abgelegt.

Die Berechnung einer Verbindung im Rückwärtsschritt (2) erfolgt auf die gleiche Weise, jedoch werden bei einer Berechnung des Fehlers einer Schicht die rückpropagierten Fehler der Neuronen der nächsthöheren Schicht nacheinander auf den Bus gelegt. Für diesen Schritt müssen die Gewichte in der Reihenfolge der Neuronen der nächsthöheren Schicht abgelegt sein.

Die Ablage aller Gewichte eines Netzes in Algorithmus I ist am Beipiel eines 4-4-3 Netzes in Abbildung 5.3 dargestellt. Die nur im Vorwärtsschritt benötigte Gewichtsmatrix **A** enthält für jedes der vier versteckten Neuronen (auf $P_0, \ldots, P_3$) die Gewichte zu den vier Eingabeneuronen. Die im Vorwärtsschritt benötigte Matrix B enthält für jedes der drei Ausgabeneuronen (auf $P_0, \ldots, P_2$) die Gewichte zu den vier versteckten Neuronen. Für den Rückwärtsschritt ist diese Matrix noch einmal transponiert abgelegt. Hier sind dann für jedes der versteckten Neuronen (auf $P_0, \ldots, P_3$) die Gewichte zu den drei Ausgabeneuronen nacheinander abgelegt.

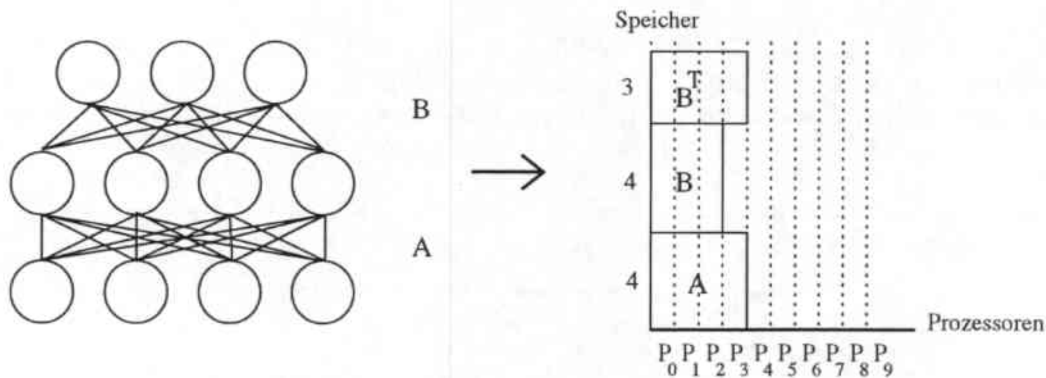


Abbildung 5.3: Verteilung der Gewichtsmatrizen am Beipiel eines 4-4-3 Netzes

Für jedes Neuron liegen somit im Speicher seines Prozessors die Gewichte der Verbindungen zu seinen jeweiligen Vorgängerneuronen und Nachfolgerneuronen. Beide müssen dann auch im Gewichtsänderungsschritt (3) entsprechend angepaßt werden[1].

[1] Wollte man nur eine Sorte der Gewichtsmatrizen ändern und die andere durch Umspeichern erzeugen, so würde dies eine Transponierung erfordern. Diese Transponierung aber würde einen (in der Kantenlänge der Matrix) quadratischen Aufwand bedeuten. Eine Änderung beider Matrizen, wie in Gewichtsänderungsschritt(3) angegeben, erfordert hingegen nur einen linearen Aufwand.
Allerdings ist bei dieser Vorgehensweise Vorsicht geboten: die Reihenfolge der Operationen und Rundungen muß für beide Matrizen exakt dieselbe sein, da sonst W und W^T unterschiedlich verändert würden, die Gleichung $W=(W^T)^T$ nach ein paar weiteren Schritten nicht einmal näherungsweise stimmte und auf diese Weise das Netz divergierte.

5.3.4 Speicherbedarf

5.3.4.1 Berechnung des Speicherbedarfs

Da für das oben beschriebene Verfahren die Gewichtsmatrizen für den Vorwärtsschritt in normaler und für den Rückwärtsschritt in transponierter Form auf dem Rechner verteilt liegen müssen, sind die Gewichte, die im Rückwärtsschritt benötigt werden (also alle mit Ausnahme der Gewichte zwischen der Eingabeschicht und anderen Schichten) zweifach vorhanden.

Der maximale Speicherbedarf für die Gewichte eines Neurons (bei dieser Neuronenverteilung der Speicherbedarf auf Prozessor 0) beläuft sich dann auf:

$$(\sum_{i=1}^{l-1} n_i + \sum_{i=3}^{l} n_i) * size_{weights}$$

5.3.4.2 Realisierbare Netzgrößen

Nach Abzug des Speicherplatzes für Aktivierungen, rückpropagierte Fehler und Verwaltungsinformationen verbleiben ca. 3400 Byte pro Prozessor. Damit sind folgende Netzgrößen berechenbar :

Gewichte	Netzstruktur				
	100-n-100	300-n-300	500-n-500	300-n-100	500-n-50
16 Bit	512 (1500)	512 (1100)	512 (700)	512 (1300)	512 (1150)
24 Bit	512 (933)	512 (33)	133	512 (733)	512(580)
32 Bit	512 (650)	250	x	450	300

Tabelle 5.1: Berechenbare Netzgrößen für MLPs mit Algorithnmus I

In Tabelle 5.1 ist für die verschiedenen Gewichtsdarstellungen von 16, 24 und 32 Bit und einige beispielhafte Netzstrukturen jeweils die größtmögliche Anzahl versteckter Neuronen angegeben. Die aktuelle Implementierung läßt maximal 512 Neuronen pro Schicht zu. Die Werte in Klammern geben die Anzahl an versteckten Neuronen an, die mit diesem Verfahren bei einem Rechner mit unendlich vielen Prozessoren realisierbar wären.

5.3.5 Laufzeit

5.3.5.1 Theoretische Betrachtungen zur Laufzeit

Die Laufzeit des Lernverfahrens wird bestimmt durch die Zeit zur Übertragung der einzelnen Aktivierungen der unteren Schicht über den globalen Bus, der Zeit zur Multiplikation mit den einzelnen Kantengewichten und der Aufsummierung in einem Akkumulator. Hinzu kommen natürlich die Zeiten zum Laden der einzelnen Eingabevektoren und der gewünschten Ausgabevektoren.
Es ergibt sich also:

$$T_n = t_{loadin} + t_{loadout} + t_{calculate}$$

wobei

$$\begin{aligned} t_{calculate} &= \sum_{i=1}^{l-1} t_{fwd} n_i + \sum_{i=3}^{l} t_{bwd} n_i + \sum_{i=1}^{l-1} t_{updt} n_i + \sum_{i=3}^{l} t_{updt} n_i \\ t_{loadin} &= t_{ld} * n_1 \\ t_{loadout} &= t_{ld} * n_n \quad \text{oder} \quad t_{ld} * 1 \end{aligned}$$

Hierbei bezeichnet t_{ld} die Zeit zum Einlesen eines Bytes aus dem Dateispeicher (die Eingabevektoren und gewünschten Ausgabevektoren der Muster wurden in der Repräsentation 1_7 abgespeichert). Die Zeit $t_{loadout}$ ist abhängig von der Art des Netzes, das man trainieren möchte. Für einen Klassifikator muß nur die Nummer des „richtigen" Ausgabeneurons gelesen werden, aufgrund dessen dann **t** festgelegt wird, während für einen Funktionsapproximator ganze Vektoren **t** eingeladen werden müssen.
Die Implementation der innersten Schleifen erfolgte in Microcode, da nur so höchstmögliche Effizienz gewährleistet werden konnte und die auf Matrizenmultiplikation zugeschnittenen Spezialmodi und Befehle der CNAPS-Prozessoren voll ausgenutzt werden konnten (durch Implementierung dieser innersten Schleifen in Mikrocode wurde eine Beschleunigung von ca. 7 gegenüber dem gleichen Programm in CNAPS-C erreicht). Die innere Schleife des Vorwärts-/Rückwärtsschrittes, die alle Prozessoren gleichzeitig ausführen, sieht dann wie folgt aus:

```
LOOP ( num_col );
{
  xmit.2seq lo;
  xmit.2seq inA mul.16 memB.16 addaddL muladdR;
  mul.16 addaddL muladdR incbase;
}
```

Um diese Zeilen zu verstehen muß man ein wenig über die Bedeutung der einzelnen Mikrobefehle wissen:

`xmit.2seq`: Dieser Befehl ist nur im sogenannten sequentiellen Zuteilungsmodus anwendbar, in dem die einzelnen Prozessoren nacheinander jeweils einen Wert auf den OUT-Bus legen. Für diesen Modus existiert eine spezielle Steuerleitung, die vom jeweiligen rechten Nachbarn eines Prozessors lesbar ist, mit der ein Prozessor anzeigt, daß er seinen Wert schon auf den Bus gelegt hat.
Die Sequentialisierung geschieht dann folgendermaßen: Der Prozessor, dessen linker Nachbar diese Steuerleitung gesetzt hat und der seine Aktivierung noch nicht auf den Bus gegeben hat legt die Aktivierung, die vor Beginn der Schleife in ein spezielles Ausgaberegister geschrieben wurde, mittels des `xmit`-Befehles auf den OUT-Bus. Zusätzlich wird durch diesen Befehl seine Steuerleitung gesetzt, die signalisiert, daß er seinen Wert abgegeben hat. Wenn man nun vor Beginn der Schleife a_0 auf den Bus legt und die Steuerleitung des Prozessors 0 setzt, legen auf diese Weise alle Prozessoren nacheinander ihre Aktivierungen auf den OUT-Bus. Die jeweilige auf den Bus gelegte Aktivierung wird (wenn man den Sequencer entsprechend anweist) über eine 2-Register Pipeline wieder in den IN-Bus geleitet und erscheint dort exakt 3 Zyklen nach dem sendenden `xmit`.

`lo`: Um einen 16 Bit Wert über den 8 Bit breiten IN-Bus zu laden muß das erste Byte zwischengelagert werden. Dies geschieht mit dem `lo`-Befehl. Der `inA`-Befehl holt dann das zweite Byte und stellt den kompletten 16 Bit Wert zur Verfügung.

`inA`: Der Wert auf dem IN-Bus wird mit dem vom `lo`-Befehl zwischengespeicherten Wert zusammengefügt und auf den internen A-Bus gelegt.

`memB.16`: Das 16 Bit Gewicht, das an der Speicheradresse liegt, auf die das Adreßregister zeigt, wird gelesen und auf den B-Bus gelegt.

`mul.16`: Der A-Bus und der B-Bus werden miteinander multipliziert. Eine 16-Bit Multiplikation benötigt 2 Zyklen, wobei die sich ergebenden Teilergebnisse im Addierer summiert werden und dort auch verbleiben, damit im nächsten Schritt der nächste Teilsummand von net_i addiert werden kann.

addaddL: Der Wert des Addiererausgangs wird auf den linken Addierereingang gelegt.

muladdR: Der Wert des Multipliziererausgangs wird auf den rechten Addierereingang gelegt.

Die innere Schleife eines Gewichtsveränderungs-Schrittes ist ungleich komplizierter, da in ihr zwei Werte multipliziert und auf ein Gewicht addiert werden, das man aus dem Speicher holen und wieder zurückschreiben muß. Wenn man diese Schleife auf Lesbarkeit auslegt und eine allgemeine Skalierung von Aktivierung und $\delta_i * learningrate$ animmm, enthält sie 34 Befehle. Für spezielle Skalierungen der Festkommazahlen im Algorithmus (a_i = 1_15, *learningdelta* = 3_13) und unter Verlust von einem Bit Genauigkeit sind hier allerdings bis zu 19 Befehle zu erreichen[2] [3].
Das Einladen der Muster geschieht ebenfalls im oben erwähnten sequentiellen Modus, in dem die Werte über den Bus aus dem im CNAPS-Server vorhandenen Dateispeicher nacheinander in die einzelnen Prozessoren geladen werden. Die Anzahl der Befehle in dieser Schleife beträgt ebenfalls 3. Somit ergibt sich:

$$t_{fwd} = t_{bwd} = 3 \text{ Zyklen}, \; t_{updt} = 34 \text{ Zyklen und } t_{load} = 3 \text{ Zyklen.}$$

$$T_n = \sum_{i=1}^{l-1} 37n_i + \sum_{i=3}^{l} 37n_i + 3 * n_1 + (3 * n_n \quad \text{oder} \quad 3)$$

5.3.5.2 Gemessene Laufzeiten

Die so errechneten Laufzeiten geben nur eine grobe Abschätzung des Lernverhaltens an. Zusätzlich müssen nämlich noch die Aktivierungsfunktionen mittels einer linear interpolierten Wertetabelle berechnet und diverse Prozeduraufrufe, Fehlerberechnungen und Multiplikationen mit Lernraten getätigt werden. All diese Operationen finden zwar pro Muster und pro Schicht nur einmal statt, kosten aber Wesentliches an Zeit, da sie nicht maschinencodiert sind und ihre Ausführungszeit von der Güte des CNAPS-C Übersetzers abhängt.
In Abbildung 5.4 ist die Laufzeit des Programms zum Trainieren eines 112-n-147 Netzes für 5000 Muster und 10 Epochen im Klassifizierungsmodus zur Subphonemerkennung angegeben. Unterteilt wurde hier die Laufzeit in die Laufzeit mit Auswertung (d.h. Berechnung von MSE) und die reine Lernlaufzeit (Ohne Auswertung). Letztere wurde wiederum unterteilt in den Anteil für die im Lernschritt notwendigen Microcoderoutinen, den Anteil der ebenfalls in Microcode realisierten Laderoutinen und den in CNAPS-C realisierten Anteil für die Berechnung der Aktivierungsfunktion, deren Ableitung, die Fehlerberechnung und die Behandlung der Schwellwerte.
Die Zeit zur Auswertung hat nahezu die Größe der Lernzeit, da die Auswerteroutine für jedes Trainingsmuster in einer Schleife über die Ausgabeneuronen den maximalen und den zweithöchsten Wert bestimmt. Ein Subphonem wird nur dann als richtig erkannt gewertet, falls diese beiden eine gewisse Differenz aufweisen. Der Code für diese Routine ist der leichten Änderbarkeit zuliebe vollständig in CNAPS-C geschrieben und die Laufzeit somit von der Güte des Übersetzers abhängig.

[2] Um genau zu sein:

Skalierung		Gewichtsdarstellung und Rundungsmethode						
		16 Bit					23 Bit	32 Bit
a_i	*learndelta*$_i$	*jam*	*round*	*roundlift*	*stoch*	*cut*	-	-
1_15	4_12	32	31	52	49	31		
1_15	3_13	20	19	37	33	19	32	23

In dieser Tabelle sind die benötigten Zyklenzahlen für die Änderungsschleifenrümpfe bei Verwendung von voller Genauigkeit (23 Bit) und den in Kapitel 3 eingeführten Rundungsoperatoren dargestellt. Die drei Rundungsmethoden *jam*,*round* und *cut* unterscheiden sich hierbei nicht wesentlich in ihrer Laufzeit. Die Variante 16 Bit mit *Stochastic weight update* erfordert zusätzliche Zyklen zum Erzeugen einer Zufallszahl und die Variante *roundlift* zum Aufwerten der Gewichtsänderungen. Die Variante 32-Bit hat zwar nur eine Genauigkeit von 23 Bit aus den im Kapitel zur beschränkten Genauigkeit beschriebenen Gründen, verbraucht aber 32 Bit Speicher pro Gewicht. Der Vorteil hierbei liegt darin, daß der Prozessor nicht mehr zwischen dem 8 und dem 16 Bit Speicherzugriffsmodus hin- und hergeschaltet werden muß und somit wertvolle Zyklen gespart werden können.

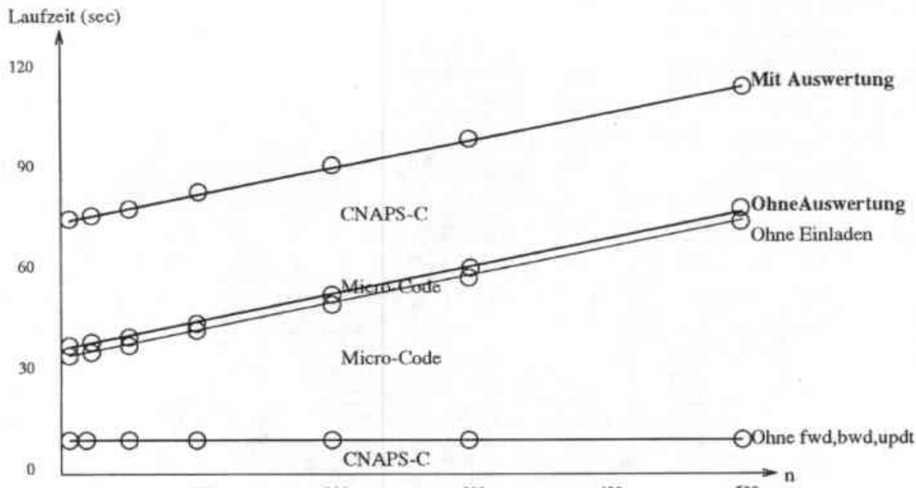


Abbildung 5.4: Laufzeitprofil eines 112-n-147 Netzes

Die aus diesen Zeiten ermittelten Leistungsdaten sind in Tabelle 5.2 dargestellt. Für die spezielle Skalierung ($a_i = 1_15$, $learndelta_j = 4_12$) mit dem daraus resultierenden kleineren t_{updt} ergeben sich die Werte in Tabelle 5.3.

	Größe der versteckten Schicht							
	500	300	200	100	50	30	20	10
Ohne Auswertung	82	63	50.1	29.7	16.8	10.4	7.2	3.7
Mit Auswertung	56	39.5	29.3	16.3	8.7	5.3	3.7	1.9

Tabelle 5.2: Leistungsdaten von Algorithmus I für allgemeine Festkommadarstellungen (in MCUPS)

	Größe der versteckten Schicht							
	500	300	200	100	50	30	20	10
Ohne Auswertung	126.3	94.1	72.0	42.4	23	14.5	9.7	5.1
Mit Auswertung	82.8	57.0	41.2	22.6	11.8	7.3	4.9	2.5

Tabelle 5.3: Leistungsdaten von Algorithmus I für die spezielle Festkommadarstellung (in MCUPS)

[3] Im Simulator von Adaptive Solutions für Neuronale Netze mit 16-Bit Gewichten wird derselbe Algorithmus, aber eine leicht andere Zahlendarstellung verwendet. Dadurch kann ein Spezialbefehl verwendet werden, der das Schieben der Zahl nach der Multiplikation erspart. Zudem wird $f_{act} = sig(x)$ verwendet, was eine unsigned-signed Multiplikation, die in einem Zyklus durchgeführt werden kann, erlaubt. Da diese Operationen zudem ohne das Hin- und Herschalten der verschiedenen Prozessormodi durchgeführt werden können, enthält der Schleifenkörper der Gewichtsänderungsschleife nur 3 (!) Befehlszeilen. Der Simulator von Adaptive mit 32 Bit Gewichtsdarstellung und $f_{act} = tanh(x)$ hingegen weist hier 20 Befehle auf.

5.3.6 TDNNs

5.3.6.1 Verteilung der Neuronen

Ein Beispiel für die Verteilung der Neuronen für *Vertical slicing* mit einem Neuron pro Prozessor bei TDNNs ist in Abbildung 5.5 zu sehen. Das hier verteilte TDNN wurde aus einem 6-4-7 MLP mit 2 Time-Delays hergeleitet. Daher sind sowohl die Eingabeschicht ($i_{1,j}$,$i_{2,j}$), als auch die versteckte Schicht ($h_{1,j}$,$h_{2,j}$) zweifach vorhanden.

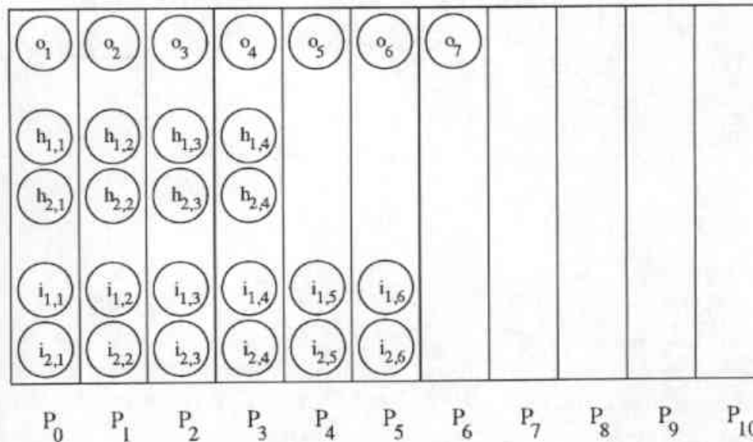


Abbildung 5.5: *Vertical Slicing* mit einem Neuron pro Prozessor als TDNN

5.3.6.2 Speicherbedarf

Im Gegensatz zum dreischichtigen MLP sind beim daraus entwickelten TDNN mehrere versteckte Schichten vorhanden, die alle über verschiedene Gewichtsmatrizen mit der Ausgabeschicht verbunden sind (siehe Abschnitt 1.7, S. 10). All diese Matrizen werden auch für den Rückwärtsschritt benötigt und müssen damit, wie oben erläutert, doppelt gespeichert werden. Der Speicherbedarf von Prozessor 0 (unter Vernachlässigung des zusätzlich benötigten Speichers für Aktivierungen, rückpropagierte Fehler und Schwellwerte für die versteckten Schichten) beläuft sich dann auf:

$$(n_1 + \#timedelays * n_2 + \#timedelays * n_3) * size_{weights}$$

5.3.6.3 Realisierbare Netzgrößen

In Tabelle 5.4 sind die mit Algorithmus I berechenbaren TDNNs dargestellt. Für einige Netzstrukturen ist für drei, fünf und sieben Time-Delays die jeweils maximale Anzahl an versteckten Neuronen angegeben, die auf der CNAPS bei diesem Algorithmus verwendet werden kann.

5.3.6.4 Detailerläuterung der TDNN-Erweiterung des MLP-Algorithmus

Da die Eingabevektoren für die verschiedenen Eingabeschichten jeweils einen Ausschnitt aus einem gesamten Eingabefenster darstellen, liegt es nahe dieses Eingabefenster für ein Muster einmal einzuladen und über die Nachbarschafts-Kommunikation die einzelnen Eingabevektoren zu erzeugen,

Gewichte	Netzstruktur				
	100-n-100	300-n-300	500-n-500	300-n-100	500-n-50
	#Time-Delays = 3				
16 Bit:	430	166	x	366	350
24 Bit:	244	x	x	177	161
32 Bit:	150	x	x	83	66
	#Time-Delays = 5				
16 Bit:	220	x	x	180	190
24 Bit:	106	x	x	66	76
32 Bit:	50	x	x	10	20
	#Time-Delays = 7				
16 Bit:	128	x	x	100	121
24 Bit:	47	x	x	19	40
32 Bit:	7	x	x	x	x

Tabelle 5.4: Berechenbare Netzgrößen für TDNNs mit Algorithmus I

anstatt die Eingabevektoren eines Musters einzeln im Dateispeicher zu halten und einzeln einzuladen. Ein Beispiel hierfür in Anlehnung an das TDNN von Seite 11 ist in Abbildung 5.6 dargestellt. Die Frequenzvektoren $\mathbf{v}_{-1}, \ldots, \mathbf{v}_5$ des kompletten Eingabefensters wurden koeffizientenweise bei Prozessor P_0 beginnend abgelegt. Da jeder Frequenzvektor 16 Koeffizienten besitzt, liegen diese 112 Koeffizienten auf den Prozessoren P_0 bis P_{111}. Die für die drei Eingabeschichten benötigten Eingabevektoren $(\mathbf{v}_{-1}^T \ldots \mathbf{v}_3^T)$, $(\mathbf{v}_0^T \ldots \mathbf{v}_4^T)$ und $(\mathbf{v}_1^T \ldots \mathbf{v}_5^T)$ können dann durch einfaches Linksschieben des kompletten Eingabefensters um 16 Prozessoren erzeugt werden.

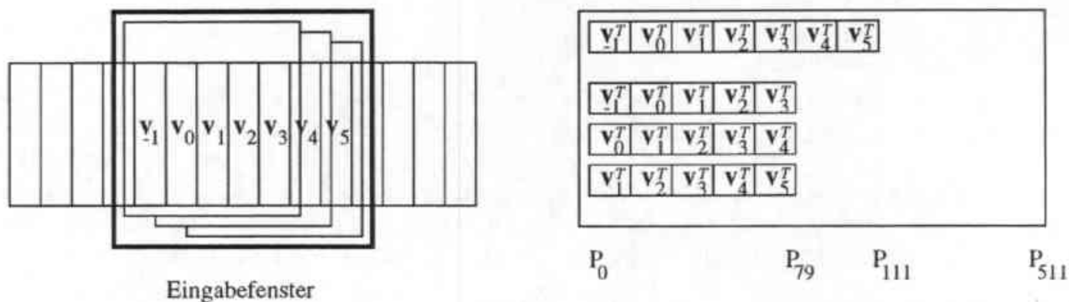


Abbildung 5.6: Die Erzeugung der einzelnen Eingabeschichten für das TDNN

5.3.7 Realisierung des Algorithmus

Die oben beschriebenen Algorithmen wurden in CNAPS-C mit den in Kapitel 3 erläuterten Darstellungen der einzelnen Werte des *Backpropagation*-Algorithmus für dreischichtige MLPs und TDNNs, wahlweise mit schichtübergreifenden Verbindungen zwischen der Eingabeschicht und der Ausgabeschicht (*shortcut connections*), realisiert. Um die hardwaregegebenen Möglichkeiten zu nutzen wurden die Routinen zur Multiplikation der Gewichtsmatrizen mit den Aktivierungsvektoren im Vorwärtsschritt, der transponierten Gewichtsmatrizen mit den Fehler-Vektoren im Rückwärtsschritt sowie die Veränderung der Gewichtsmatrizen in Microcode mit den oben angegebenen

Zyklenzahlen implementiert. Für das Erreichen einer möglichst hohen Geschwindigkeit (und vor allem, da das Hardwarekonstrukt für schnelle Schleifen im CNAPS-Server 1 nur mit festdefinierten Schleifendurchläufen arbeitete!) wurde der Code so gestaltet, daß die Größen der einzelnen Schichten, des Eingabefensters und die zu verwendende Gewichtsänderungsfunktion fest bei der Übersetzung in den Code eingefügt werden. Der Quellcode hat eine Größe von ungefähr 2350 CNAPS-C Zeilen mit zusätzlichen 150 Zeilen Microcode, sowie weiteren 400 Zeilen für die fünf übrigen in Kapitel 3 beschriebenen Gewichtsveränderungsfunktionen und kann in ca. 70 sec für ein neues Netz übersetzt werden. Eine Übersetzung mit sämtlichen Debugroutinen und Probeausgaben dauert ca. 180 sec.

5.4 Neuronaler Netz Algorithmus II

5.4.1 Kurzübersicht

Zielsetzung dieses Algorithmus

- Training einer Menge von mehreren ähnlichen neuronalen Netzen (Aufgabenstellung 3)
- Training möglichst großer neuronaler Netze

Gewählter Lösungsweg

- Verwendete Parallelitäten: Netzparallelität, Neuronenparallelität
- Verteilung der Netze: Teilung des Prozessorvektors in kleinere Teilbereiche (Subvektoren) von denen jeder ein Netz erhält
- Verteilung der Neuronen: *Vertical Slicing* mit mehreren Neuron pro Prozessor
- Verteilung der Gewichte: Gewicht ω_{ij} bei Neuron i oder j, bestimmt durch Algorithmus zur Gewichtsspeicherminimierung
- Berechnungsschema: Aktivierungsvektor-Gewichtsmatrix-Multiplikation mittels Rotation des Aktivierungsvektors über Nachbarschaftskommunikation innerhalb des jeweiligen Subvektors

5.4.2 Verteilung der Neuronen

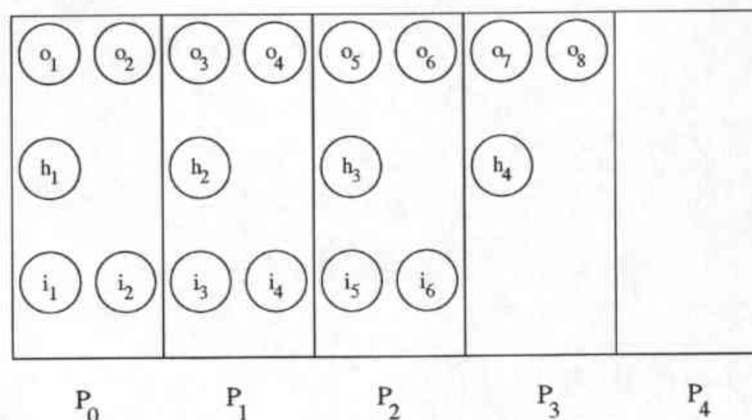
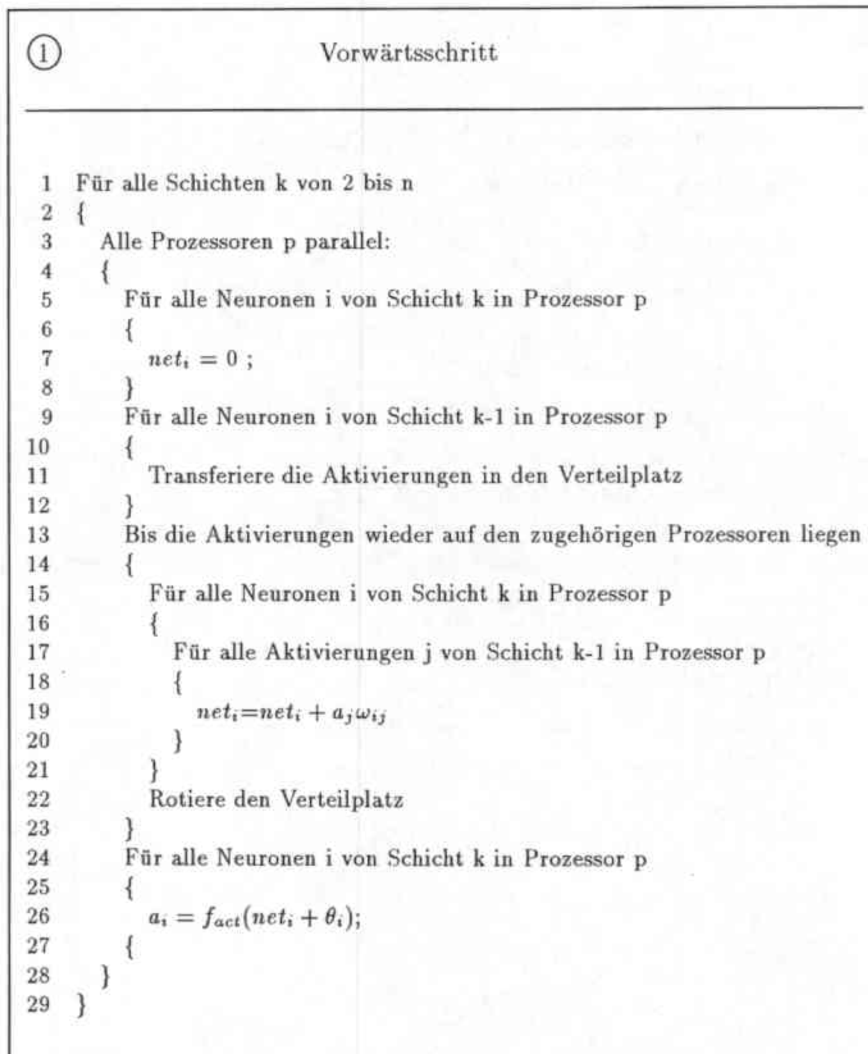


Abbildung 5.7: Verteilung der Neuronen bei *Vertical Slicing* mit mehreren Neuronen pro Prozessor

Die Verteilung bei *Vertical Slicing* mit mehreren Neuronen pro Prozessor ist in Abbildung 5.7 dargestellt. In diesem Beispiel ist die Verteilung eines 6-4-8 Netzes auf 5 Prozessoren zu sehen. Die Neuronen der Eingabe- (i_j) und Ausgabeschicht (o_j) sind jeweils mit zwei, die Neuronen der versteckten Schicht (h_j) mit einem Neuron pro Prozessor verteilt. Welche Neuronen-Dichten von Algorithmus II zur Verteilung der Neuronen der einzelnen Schichten sinnvoll sind und welche verwendet werden, wird in den Abschnitten „Speicherbedarf“ (Abschnitt 5.4.4) und „Laufzeit“ (Abschnitt 5.4.5) behandelt werden.

5.4.3 Detailerläuterung des Algorithmus

Die Berechnung der Werte des *Backpropagation*-Algorithmus im Vorwärtsschritt geschieht dann bei Algorithmus II wie in Darstellung 5.3.



```
①                         Vorwärtsschritt

 1  Für alle Schichten k von 2 bis n
 2  {
 3    Alle Prozessoren p parallel:
 4    {
 5      Für alle Neuronen i von Schicht k in Prozessor p
 6      {
 7        net_i = 0 ;
 8      }
 9      Für alle Neuronen i von Schicht k-1 in Prozessor p
10      {
11        Transferiere die Aktivierungen in den Verteilplatz
12      }
13      Bis die Aktivierungen wieder auf den zugehörigen Prozessoren liegen
14      {
15        Für alle Neuronen i von Schicht k in Prozessor p
16        {
17          Für alle Aktivierungen j von Schicht k-1 in Prozessor p
18          {
19            net_i=net_i + a_j w_ij
20          }
21        }
22        Rotiere den Verteilplatz
23      }
24      Für alle Neuronen i von Schicht k in Prozessor p
25      {
26        a_i = f_act(net_i + θ_i);
27      {
28    }
29  }
```

Algorithmusdarstellung 5.3 : Der Vorwärtsschritt in *Backpropagation*-Algorithmus II

Im wesentlichen werden also alle auf dem jeweiligen Prozessor vorhandenen Verbindungen berechnet, die Aktivierungen der Neuronen der unteren Schicht einen Prozessor weitergeschoben, wieder

berechnet, und dies solange durchgeführt, bis jede Aktivierung der unteren Schicht an jedem Neuron der oberen Schicht vorbeigekommen ist und somit das präsynaptische Potential für jedes Neuron der oberen Schicht berechnet werden konnte.

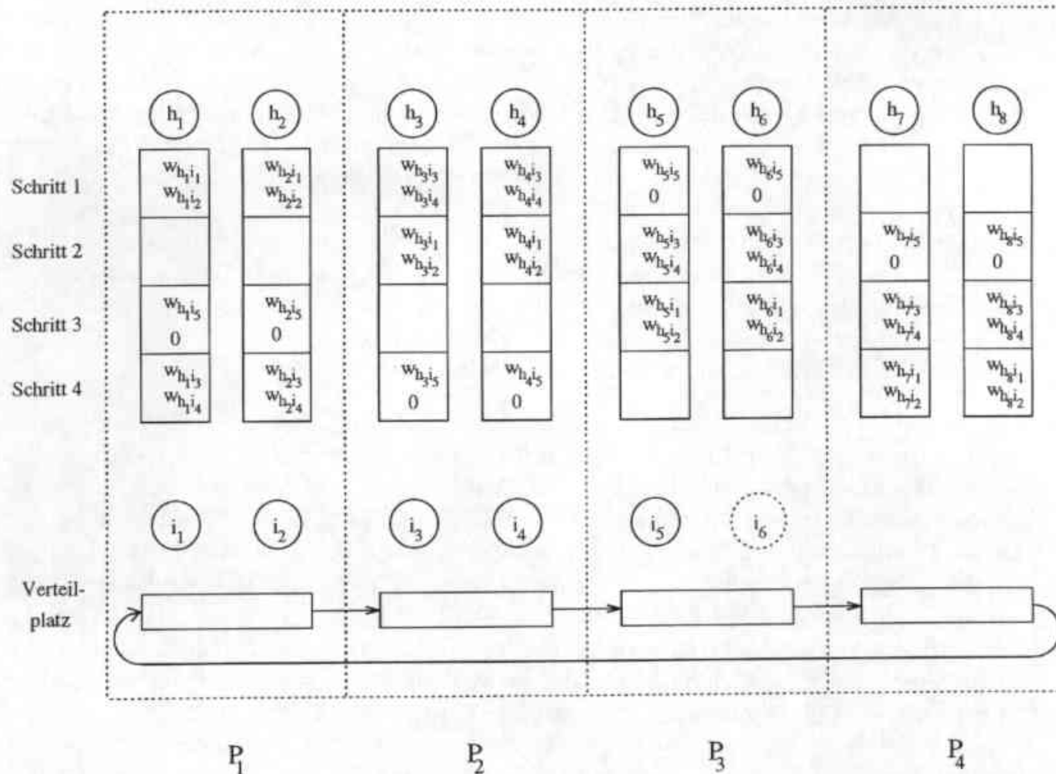


Abbildung 5.8: Die Berechnung des Vorwärtsschrittes

Ein Beispiel für eine solche Berechnung einer 5-8 Schicht-zu-Schicht-Verbindung im Vorwärtsschritt ist anschaulich in Abbildung 5.8 dargestellt. Zu Beginn der Berechnung werden hier die Aktivierungen der Neuronen i_1 bis i_6 in den sogenannten *Verteilplatz* kopiert (vergl. Zeile 9-12 des Algorithmus). Dieses hat den Vorteil, daß nicht die kompletten Neuronen (mit Aktivierung, Schwellwert, Fehlerwert,...), sondern nur die Aktivierungswerte rotiert werden müssen. Danach werden die Verbindungen zwischen den im jeweiligen Prozessor vohandenen Neuronen berechnet (vergl. Zeile 15-20). In Schritt 1 sind dies in Prozessor 1 die Verbindungen zwischen den Neuronen i_1 und i_2 sowie h_1 und h_2. Die Gewichte der Verbindungen sind dabei für jedes Neuron der oberen Schicht in der benötigten Reihenfolge in einem Feld abgelegt. Nachdem alle Verbindungen innerhalb der Prozessoren berechnet sind, werden die Aktivierungen im Verteilplatz alle einen Prozessor nach rechts rotiert und die in den Neuronen der oberen Schicht vorhanden net_i-Teilsummen weiter vervollständigt. Nach 4 Schritten schließlich sind auf diese Weise die Aktivierungen der unteren Schicht wieder auf ihrem Ausgangsprozessor angekommen und die Variablen net_i der oberen Neuronen enthalten die komplette Summe ihrer gewichteten Vorgängeraktivierungen. In dieser Abbildung ist nebenbei auch ersichtlich, wozu die in Kapitel 2 betrachtete Einbettung von neuronalen Netzen nötig ist. Falls die Anzahl der Neuronen einer Schicht für die Verteilung mit einer gewissen Menge von Neuronen pro Prozessor nicht ausreicht, wird einfach die noch fehlende Anzahl Neu-

ronen hinzugefügt (in der Abbildung Neuron i_6), d.h. das Netz in ein Netz mit einer größeren Schicht eingebettet. Die Gewichte der Verbindungen dieses Pseudoneurons sind hierbei, wie in 1.5 gefordert, auf Null gesetzt. Auf diese Weise wird durch die Einbettung eines Netzes in ein größeres, passenderes Netz, eine Sonderbehandlung der Prozessoren mit weniger Neuronen während der zeitkritischen Berechnungen vermieden und so die Implementierung des Algorithmus wesentlich vereinfacht.
Auf dieselbe Art und Weise wie im Vorwärtsschritt können dann auch die rückpropagierten Fehler im Rückwärtschritt für die Neuronen der unteren Schicht berechnet werden. Hierbei werden die bei der Rückpropagierung in der unteren Schicht entstehenden Zwischensummen von gewichteten Fehlerwerten der oberen Schicht rotiert. Bei jeder Rotation werden weitere Summanden hinzugefügt, bis schließlich die komplette gewichtete Summe der rückpropagierten Fehler der Nachfolgerneuronen entstanden ist und die Summen wieder auf den Prozessoren, auf denen ihre zugehörigen Neuronen liegen, angekommen sind.
Im Änderungsschritt schließlich wird ebenso verfahren. Hier werden in der unteren Schicht die Aktivierungen an den Neuronen der oberen Schicht (mit ihren rückpropagierten Fehlern) vorbeigeschoben und in jedem Schritt die Gewichte der Verbindungen zwischen den im jeweiligen Prozessor vorhandenen Neuronen der unteren und oberen Schicht geändert.
Der eben beschriebene Algorithmus hat folgende Vorteile:

Mehrere Netze: Da die Aktivierungen und Zwischensummen der Neuronen über die Nachbarschaftsverbindungen geschoben werden und kein Datenaustausch über den nur exklusiv zuteilbaren Bus stattfindet, ist es möglich den Prozessorvektor in Subvektoren zu unterteilen und auf jedem einzelnen ein Netz mit anderen Parametern (Lernraten, Translation der Ableitung, ...) laufen zu lassen. Aufgrund der in Abschnitt 1.5 beschriebenen Netzeinbettungen ist es sogar möglich Netze mit verschiedener Architektur zu trainieren, solange sie alle in ein gemeinsames Rahmennetz eingebettet worden sind. Dies Vorgehen wäre in Algorithmus I nur unter zeitlichen Verlusten in Höhe der Anzahl der gleichzeitig zu trainierenden Netze möglich, da sämtliche Aktivierungen über den exklusiv zuteilbaren Bus gesendet werden müßten. Ein Beispiel hierfür ist in Abbildung 5.9 zu sehen. Hier wurde ein Prozessorvektor mit 16 Prozesoren in vier Subvektoren mit jeweils vier Prozessoren unterteilt. Auf allen vier Subvektoren wird jeweils ein 4-4-3 Netz trainiert, wobei allerdings in den Subvektoren 2,3 und 4 kleinere versteckte Schichten vorliegen. Diese Netze wurden mit den vier Regeln aus Abschnitt 1.5 (S. 5) in das 4-4-3 Netz unter Hinzunahme von Pseudoverbindungen und Pseudoneuronen eingebettet.

Speicher/Zeitersparnis: Alle Matrizen, die zwischen den einzelnen Schichten in diesem Verfahren existieren, können für den Vorwärts-, den Rückwärts- und den Gewichtsveränderungs-Schritt verwendet werden und müssen somit auch nur einmal vorhanden sein. Hierdurch wird nicht nur Speicherplatz, sondern auch Zeit gespart, da die Gewichte einer Matrix auch nur einmal geändert werden müssen. Im Algorithmus I müssen ja bei einer Gewichtsänderung nicht nur die Matrizen, sondern auch ihre Transponierten angepaßt werden.

Gewichtsverteilung: Statt im Vorwärtsschritt die Aktivierungen der unteren Schicht zu rotieren und die Gewichte bei den Neuronen der oberen Schicht zu speichern, ist es natürlich ebenso möglich die Teilsummen der oberen Schicht zu rotieren und die Gewichte bei den Neuronen der unteren Schicht zu speichern. Durch diese Wahlmöglichkeit kann man die Gewichtsmatrizen so abspeichern, daß sie möglichst wenig Speicher verbrauchen und nicht, wie bei Verfahren 1, immer bei den Neuronen der oberen Schicht. Ein Beispiel hierfür ist in Abbildung 5.10 dargestellt. Hier müssen bei Rotieren der oberen Schicht (Gewichtsverteilung 1) für jedes der vier Neuronen der unteren Schicht nur zwei Gewichte abgelegt werden, während bei Rotation der unteren Schicht (Gewichtsverteilung 2) pro Neuron der oberen Schicht vier Gewichte abgelegt sind. In diesem Fall wäre also unter speichertechnischen Gesichtspunkten die Rotation der oberen Schicht und damit Gewichtsverteilung 1 vorzuziehen.

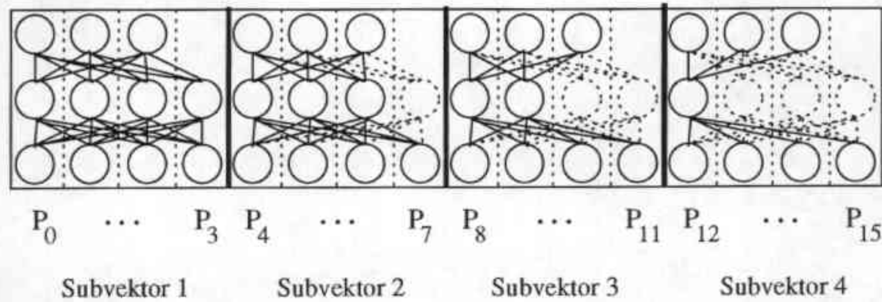


Abbildung 5.9: Netzparallelität durch Unterteilung des Prozessorvektors in Subvektoren und Einbettung der Netze

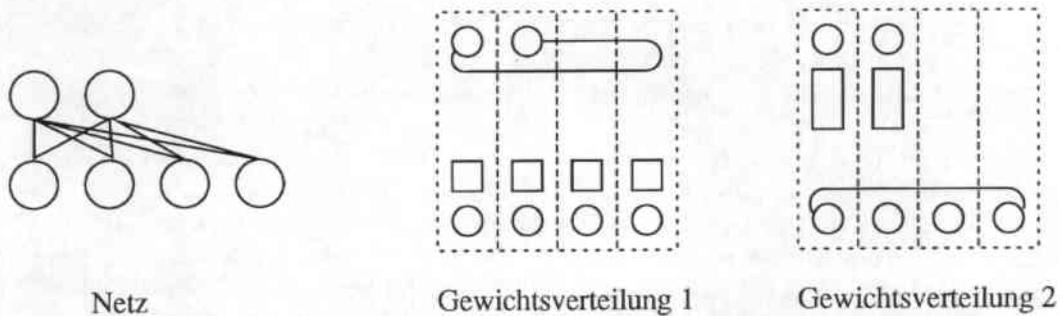


Abbildung 5.10: Unterschiedliche Gewichtsverteilung durch Berechnungsänderung

5.4.4 Speicherbedarf

5.4.4.1 Berechnung des Speicherbedarfs

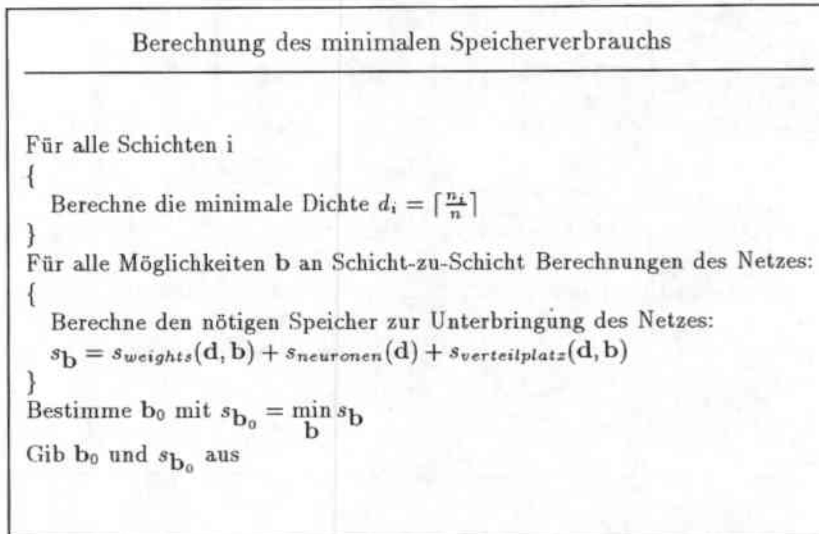
Im Gegensatz zu Algorithmus I, in dem die Verteilung der Gewichte durch den Algorithmus vorgeschrieben war, gibt es hier zwei Möglichkeiten die Verteilung der Gewichte zu steuern:

1. Die Dichte $d_i (i = 1, \ldots, l)$ mit der die Neuronen der Schicht i in jedem Prozessor abgelegt werden

2. Die Art jeder einzelnen Schicht-zu-Schicht Berechnung (siehe Abbildung 5.10)

Um nun zu überprüfen, ob ein Netz auf eine Menge von n Prozessoren mit gegebenem Speicher s paßt, berechnet man einfach den minimal notwendigen Speicher pro Prozessor.
Im folgenden sei $\mathbf{b} = (b_1, \ldots, b_{l-1})$ ein boolescher Vektor dessen Koeffizient b_i angibt, ob bei der Berechnung von Schicht i zu Schicht $i+1$ die untere oder obere Schicht rotiert wird. Die Größen $s_{weights}(\mathbf{d}, \mathbf{b})$, $s_{neuronen}(\mathbf{d})$, $s_{verteilplatz}(\mathbf{d}, \mathbf{b})$ mögen den pro Prozessor benötigten Speicher für

Gewichte, Neuronen (Aktivierungen, Fehler, Schwellwert, ...) und den Verteilplatz bezeichnen. Die Berechnung des speicherminimalen Layouts eines Netz kann dann wie in Darstellung 5.4 vorgenommen werden.



Berechnung des minimalen Speicherverbrauchs

Für alle Schichten i
{
  Berechne die minimale Dichte $d_i = \lceil \frac{n_i}{n} \rceil$
}
Für alle Möglichkeiten **b** an Schicht-zu-Schicht Berechnungen des Netzes:
{
  Berechne den nötigen Speicher zur Unterbringung des Netzes:
  $s_{\mathbf{b}} = s_{weights}(\mathbf{d}, \mathbf{b}) + s_{neuronen}(\mathbf{d}) + s_{verteilplatz}(\mathbf{d}, \mathbf{b})$
}
Bestimme $\mathbf{b}_0$ mit $s_{\mathbf{b}_0} = \min_{\mathbf{b}} s_{\mathbf{b}}$
Gib $\mathbf{b}_0$ und $s_{\mathbf{b}_0}$ aus

Algorithmusdarstellung 5.4 : Berechnung des speicherminimalen Layouts eines Netzes

Der Beweis für die Korrektheit dieser Vorgehensweise folgt aus der Tatsache, daß bei festem **b** jedes Netz, das in auch nur einer Schicht mit einer höheren Dichte als der minimalen verteilt wird, zu höherem $s_{neuronen}(\mathbf{d})$ und zu gleichem oder höheren $s_{weights}(\mathbf{d}, \mathbf{b})$ und $s_{verteilplatz}(\mathbf{d}, \mathbf{b})$ führt, also mehr Speicher pro Prozessor benötigt. Somit verbleibt als freier Spielraum zur Parameterwahl die Art der Berechnung jeder einzelnen Schicht-zu-Schicht Verbindung. Nach Abbildung 5.10 sollte es genügen, jede Schicht-zu-Schicht Verbindung so zu berechnen, daß man einfach die kleinere Schicht rotieren läßt. Dies stimmt jedoch nur, wenn man ausschließlich die Gewichtsmatrizen betrachtet. Zum einen besteht nämlich ein Unterschied darin, ob Aktivierungen (mit 16 Bit pro Neuron) oder Teilsummen (mit 32 Bit pro Neuron) rotieren, wodurch sich dann die Größe des Verteilplatzes berechnet. Zum anderen muß auch beachtet werden, daß für versteckte Neuronen und Ausgabeneuronen mehr Speicherplatz (Aktivierung, Fehler und Schwellwert) als für die Eingabeneuronen (nur Aktivierungen) benötigt wird. Somit müssen alle 2^{l-1} Möglichkeiten von **b** auf ihren Speicherbedarf durchgerechnet werden.
Als Illustration für diesen Algorithmus ist in der Tabelle 5.5 die Verteilung von drei Netzen gezeigt, die alle auf 512 Prozessoren mit 16 Bit Gewichten berechnet werden sollen. Für jedes Netz sind die minimalen Dichten, aufgrund dieser Dichten für jede Berechnungsart der notwendige Speicher und die daraufhin gewählte Berechnungsart angegeben.

5.4.4.2 Realisierbare Netzgrößen

Auch hier verbleiben nach Abzug des Speicherplatzes für Aktivierungen, Fehlerwerte und Verwaltungsinformationen ca. 3400 Byte pro Prozessor. Die maximal möglichen Schichtgrößen für MLPs sind in Tabelle 5.6 angegeben.
Wie aus der Tabelle ersichtlich, sind mit dieser Methode Netzgrößen möglich, die mit Algorithmus I nicht berechenbar waren, da dort die Lage der Gewichte bei den jeweiligen Neuronen der höheren Schicht vorgeschrieben war und die im Rückwärtsschritt benötigten Matrizen doppelt gespeichert werden mußten.

Netz-struktur	d			b			
	d_1	d_2	d_3	(0,0)	(0,1)	(1,0)	(1,1)
2000-100-10	4	1	1	4292	4112	1068	**888**
10-5000-10	1	10	1	10436	**564**	20236	10436
1000-10-1000	2	1	2	2104	4064	**136**	2104

Tabelle 5.5: Speicherverbrauch pro Prozessor für verschiedene Berechnungsarten (in Byte)

Gewichte	Netzstruktur				
	100-n-100	300-n-300	500-n-500	300-n-100	500-n-50
16 Bit	4096	1024	838	2048	1536
24 Bit	2560	556	556	1024	1024
32 Bit	2048	512	421	1024	737

Tabelle 5.6: Berechenbare Netzgrößen für MLPs mit Algorithmus II

5.4.5 Laufzeit

5.4.5.1 Theoretische Betrachtungen zur Laufzeit

Die zur Berechnung notwendige Zeit setzt sich bei diesem Algorithmus aus der Zeit zum Berechnen der einzelnen Verbindungen, der Zeit zum Rotieren der Daten im Verteilplatz und der Zeit zum Einladen der Muster zusammen. Also ergibt sich:

$$T_n = t_{loadin} + t_{loadout} + t_{calculate},$$

wobei nach dem bisher beschriebenen Schema gilt:

$$\begin{aligned} t_{calculate} = & \sum_{i=1}^{l-1} \max(p_{rot,i}, p_{stat,i}) * (t_{rotate} e_i + d_{i+1} d_i t_{fwd}) + \\ & \sum_{i=2}^{l-1} \max(p_{rot,i}, p_{stat,i}) * (t_{rotate} e_i + d_{i+1} d_i t_{bwd}) + \\ & \sum_{i=1}^{l-1} \max(p_{rot,i}, p_{stat,i}) * (t_{rotate} e_i + d_{i+1} d_i t_{updt}) \end{aligned}$$

Hierbei ist $p_{rot,i}$ die Anzahl der Prozessoren, auf die die jeweils rotierende Schicht bei der Schicht-zu-Schicht Berechnung i verteilt ist, $p_{stat,i}$ die Anzahl der Prozessoren, auf die die feststehende Schicht verteilt ist, e_i die Dichte der zu bewegenden Schicht ($e_i = d_{i+1} * b_i + d_i * (1 - b_i)$) und t_{rotate} die Zeit zum Rotieren eines Neurons.
Da diese Formel zur weiteren Betrachtung ein wenig unhandlich ist, nehmen wir uns die Formel, die die Zeit für eine einzelne Schicht-zu-Schicht Berechnung beschreibt und setzen für die jeweilige Zeit zur Verbindungsberechnung in Vorwärts-, Rückwärts- oder Veränderungsschritt ein allgemeines t_{conn}:

$$t_{layertolayer} = \max(p_{rot,i}, p_{stat,i})(t_{rotate} e_i + d_{i+1} d_i t_{conn})$$

Wie aus der Formel ersichtlich, hat das bisher beschriebene Vorgehen insbesondere bei kleinen Schichten den Nachteil, daß ein großer Teil der Prozessoren während der Berechnung leerläuft (dies zeigt sich auch in Abbildung 5.8 an den leeren Kästchen).

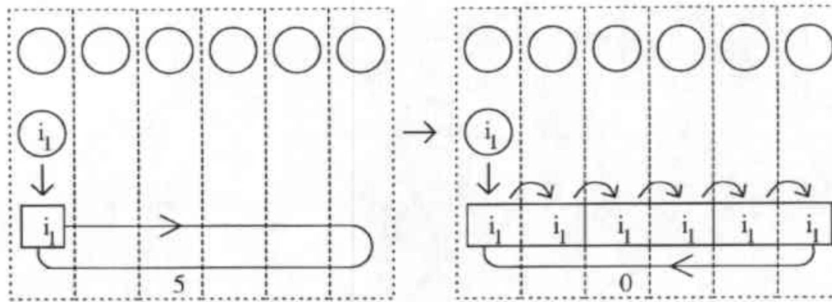


Abbildung 5.11: Verteilung der Aktivierungen durch Vorabkopieren

Um dieses zu verhindern, kann man die Aktivierungen der zu rotierenden Schicht vorab gemäß Abbildung 5.11 verteilen und danach die Rotationen und Berechnungen ausführen. Durch diese Trennung der einzelnen Schicht-zu-Schicht Berechnung in einen Verteil- und einen Berechnungsschritt ist zum einen eine vollständige Auslastung der Prozessoren während des Berechnungsschrittes gewährleistet, zum anderen ergeben sich aber auch implementierungstechnische Vorteile, da während der Berechnung nicht mehr zwischen Prozessoren, die gerade Aktivierungen der unteren Schicht vorliegen haben und solchen, bei denen das nicht der Fall ist, unterschieden werden muß[4]. Sollen Werte in der rotierenden Schicht berechnet werden, so wird dies Prinzip natürlich umgekehrt angewandt. Erst werden die Teilsummen im Berechnungsschritt mit Rotationen berechnet und dann die dabei entstandenen kleinen Vektoren aus Teilsummen in einem Sammelschritt aufaddiert.

Hiermit sieht $t_{calculate}$ dann etwas nach oben abgeschätzt folgendermaßen aus:

$$t_{layertolayer} = p_{stat,i} t_{rotate} e_i + p_{rot,i}(t_{rotate} e_i + d_{i+1} d_i t_{conn})$$

Nun noch zu einem kleinen aber wichtigen Detail, das bisher keine Beachtung gefunden hat: die CNAPS hat keine Möglichkeiten zum Rotieren über eine beliebige Menge von Prozessoren hinweg, sondern besitzt nur eine ringförmige Nachbarschaftskommunikation! Somit müssen also die normalerweise durch Rotation an der linken Seite des Prozessorfeldes eingehenden Aktivierungswerte der Neuronen der unteren Schicht durch Vorabkopieren ersetzt werden und daraus resultierend die doppelte Menge an Aktivierungswerten rotiert werden. Ein Beispiel dafür ist in Abbildung 5.12 zu sehen. Die Berechnung der 2-6 Schicht-zu-Schicht Verbindung wird hier wie folgt vorgenommen: Die Aktivierungen der beiden Neuronen der unteren Schicht (i_1,i_2) werden erst einmal in den Verteilplatz kopiert. Dann werden sie in der Vorabverteilung nach rechts bis zum Ende der oberen Schicht vervielfältigt. Zusätzlich werden sie noch einmal nach links kopiert, um für die Neuronen der oberen Schicht, die bei einer Rotation normalerweise entstehenden Aktivierungen bereitzustellen. Damit jede Aktivierung der unteren Schicht an jedem Neuron der oberen Schicht vorbeikommt, muß während der Berechnung der, jetzt allerdings doppelt so große, Verteilplatz dann nur noch einmal nach rechts geschoben werden.

[4] Um dieses Problem zu umgehen könnte man auch die „zu klein“ geratenen Schichten der zu berechnenden Netze mit Pseudoneuronen auffüllen, so daß alle Schichten die gleiche Prozessoranzahl besitzen. Man würde aber mit einer solchen Einbettung natürlich den pro Prozessor nötigen Gewichtsspeicher erhöhen. Da gerade dieser eine bezüglich der Aufgabenstellung kritische Größe ist, wurde hierauf verzichtet.

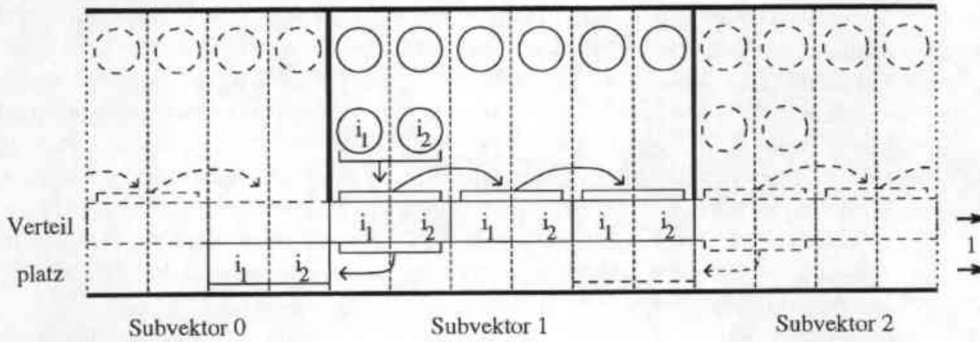


Abbildung 5.12: Verteilung der Aktivierungen zur Verwirklichung von Rotation

5.4.5.2 Gemessene Laufzeiten

Da die obige Laufzeitabschätzung im Gegensatz zu Algorithmus I sehr grob war, wollen wir uns nun das Verhalten dieses Algorithmus in der Praxis etwas genauer anschauen. In Abbildung 5.13 ist das Laufzeitprofil eines 112-n-147 Netzes mit variabler Anzahl an versteckten Knoten mit der Verteilung $\mathbf{d} = (1, 1, 1)$ und dem Berechnungsschema $\mathbf{b} = (1, 0)$[5] dargestellt. Gemessen wurde die Trainingszeit wie bei Algorithmus I auf 10 Epochen á 5000 Musterpräsentationen im Klassifizierungsmodus.

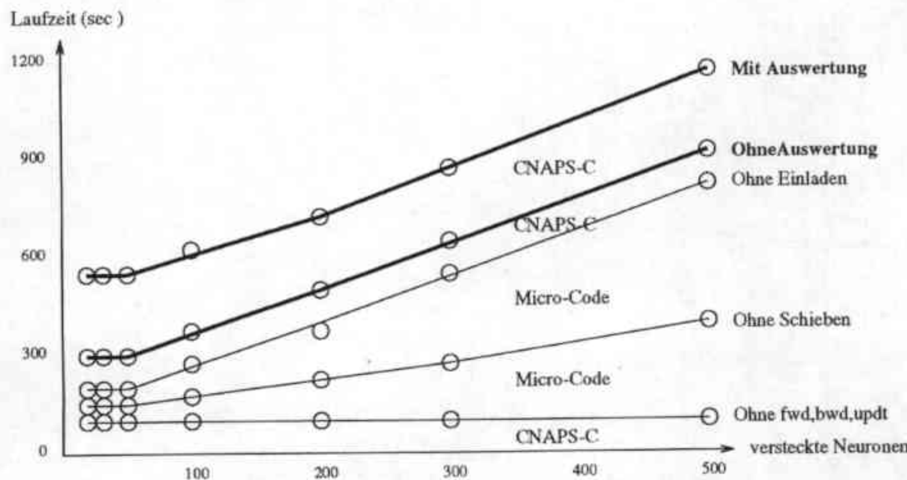


Abbildung 5.13: Laufzeitprofil eines 112-n-147 Netzes

Durch Verwendung des gleichen Berechnungsschemas für alle Netze skalieren sich die Trainingszeiten auch hier nahezu linear mit Anzahl der versteckten Neuronen. Die im Vergleich zu Algorithmus

[5] Dieses Berechnungschema ist für $n > 147$ suboptimal, aber erlaubt eine konsistente Betrachtung der Laufzeiten, da ein festes Berechnungsschema verwendet wird und nicht das für die jeweiligen Schichtgrößen speicherminimale Berechnungsschema. Die Berechnungszeiten zum Training eines Netzes mit 500 versteckten Neuronen, das nach dem oben vorgestellten Algorithmus verteilt wurde, liegen um ca. 25% niedriger.

I wesentlich höheren Zeiten sind bedingt durch die Tasache, daß die Berechnung der Verbindungen von zwei Schleifen (Zeilen 16-21) umgeben ist, die für jeden Schiebeschritt initialisiert werden müssen. Zudem wird hier pro Prozessor net_i für mehrere Neuronen berechnet, so daß die sich bildenen Zwischensummen aus dem lokalen Speicher geholt und dahin auch wieder zurückgeschrieben werden müssen, während net_i bei Algorithmus I, der ja nur ein Neuron pro Prozessor berechnet, im Addierer akkumuliert werden könnte. Gerade bei der hier gewählten Dichte 1 überwiegt dieser Aufwand die eigentliche Verbindungsberechnung, die in 3 (fwd/bwd) respektive ca. 20-30 (updt) Zyklen pro Verbindung abgewickelt werden kann. Die Schleife über alle Neuronen eines Prozessors führt auch bei Einladen und Auswerten zu erheblichen Mehraufwand, zumal diese Routinen in CNAPS-C programmiert sind und der Übersetzer, der leider nicht optimierend ist und keine Variableneliminierung oder Verwendung von Registern kennt, einen unnötigen zusätzlichen Aufwand für die Schleifenbearbeitung einführt.
Eine weitere Schwachstelle ist die mit nur 2 Bit ausgelegte Verbindung zwischen den Prozessoren, die zudem einen speziellen Modus erfordert, so daß für jedes zum Nachbarn übertragene Bit ein Zyklus verbraucht wird[6]. Da sich im Verteilplatz entweder 16-Bit Aktivierungen oder 32-Bit Teilsummen befinden, ist der zum Schieben der Werte notwendige Kommunikationsaufwand bei dieser Dichte sogar größer als der Aufwand für die Berechnung der Verbindungen inklusive der sie steuernden Schleifen. In Algorithmus I hingegen kann die komplette Kommunikation und Beschaffung der Daten aufgrund des speziellen Sequentialisierungsmodus und der 8 Bit breiten OUT- und IN-Busse schon während der Berechnung geschehen.
In Abbildung 5.14 ist ersichtlich, wie sich die Trainingszeit verhält, wenn das Netz mit höheren Dichten verteilt wird. Hierzu wurde die CNAPS in 3 bis 25 Subvektoren aufgeteilt und mit der obigen Prozedur das für die jeweilige Subvektorlänge nötige speicherminimale Layout des Netzes ermittelt. Die mit $\mathbf{d} = (1,1,1)$ größte realisierbare Zahl an Subvektoren ist 3 ($3 * 147 = 431 < 512$, $4 * 147 = 588 > 512$), während 25 Subvektoren die speicherbedingte Obergrenze darstellt (ein Netz wird dann auf nur noch 20 Prozessoren verteilt, die Dichte beträgt $\mathbf{d}=(6,5,8)$).

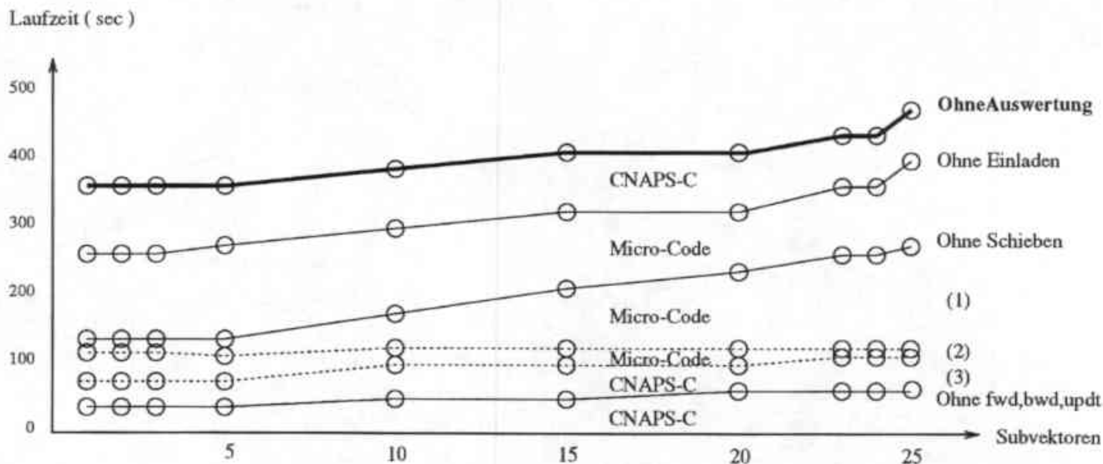


Abbildung 5.14: Laufzeitprofil für das Training mehrerer 112-100-147 Netze

Die Zeit zum Einladen eines Musters aus der Mustermenge bleibt ebenso wie die Zeit zum Auswerten konstant, da die Routine im Prinzip eine Schleife über alle Neuronen der Eingabe respek-

[6] Diese Angabe gilt für eine allgemeine Schleife. Für größere Bitmengen und mit Ausrollen der Schleife sind 2 Bit pro Zylus zu erreichen.

tive Ausgabeschicht ist (bei sehr genauer Betrachtung stimmt dies nur näherungsweise, da die Anordung der Schleifen bei einer höheren Anzahl von Neuronen und der kleineren Anzahl an zu durchlaufenden Prozessoren zu einer etwas geringeren Zeit führt).
Der Aufwand zum Schieben des Verteilungsfeldes ist zwar abhängig von **d** und **b**, die sich aufgrund der Größe des Subvektors bestimmen, aber im wesentlichen konstant, da es den gleichen Aufwand bedeutet, ob man eine Schicht mit $d_i = 1$ einhundert Prozessoren weit schiebt oder eine Schicht mit $d_i = 100$ einen Prozessor weit. Die leichten Varianzen der Kurve kommen dadurch zustande, daß aufgrund der verschiedenartigen Layouts unterschiedliche Mengen an Pseudoneuronen zum Auffüllen des letzten Prozessors eingefügt werden.
Die Berechnung der Aktivierungen (Ohne Schieben - Ohne fwd,bwd,updt) gliedert sich in 3 Teilgebiete:

- die eigentliche Berechnung der auf einem Prozessor gerade berechenbaren Verbindungen im innersten Schleifenkörper (1) (vergl.: Algorithmus Zeile 19)
- die Initialisierung und Abwicklung der innersten 2 Schleifen (2) (vergl.: Algorithmus Zeile 15-18, 20-21)
- die kompletten Aufrufschemata in denen bestimmt wird, welcher Aktivierungsvektor mit welcher Gewichtsmatrix verknüpft wird und die Aufrufe der entsprechenden Subroutinen(3)

In Abbildung 5.14 sieht man deutlich, daß der Aufwand zur eigentlichen Verbindungsberechnung (1) linear in der Dichte steigt, während der Aufwand zur Initialisierung und Durchführung der innersten Schleifen (2) mit kleinerer Subvektorlänge und damit geringerer Anzahl an Rotationen bei einer Schicht-zu-Schicht Berechnung sinkt. Für die Verteilung auf 3 Subvektoren ($\mathbf{d} = (1,1,1)$) wird nämlich genau eine Verbindung pro Durchlauf der innersten zwei Schleifen berechnet (und dieses pro Schicht-zu-Schicht Verbindung für durchschnittlich 100 Schiebeschritte), wohingegen bei 25 Subvektoren ($\mathbf{d} = (6,5,8)$) im Durchschnitt 35 Verbindungen in den Durchläufen der beiden innersten Schleifen berechnet werden und nur 20 Schiebeschritte notwendig sind. Dementsprechend sinkt die Zeit für die Ausführung der Schleifen an sich von 3 Subvektoren zu 25 Subvektoren auf ein Fünftel ab ($\frac{100}{20}$), während die Zeit zur Verbindungsberechnung auf das ungefähr 7-fache steigt (100 Schiebeschritte $*$ 1 Verbindung $= 100 \rightarrow 20$ Schiebeschritte $*$ 35 Verbindungen $= 700$). Das Verhältnis des Aufwands der Schleifeninitialisierung zum echten Verbindungsberechnungsaufwand gestaltet sich also umso günstiger, je dichter die Neuronen auf die Prozessoren verteilt werden.
Aus den so gemessenen Werten berechnen sich die in Tabelle 5.7 dargestellten Leistungsdaten von Algorithmus II: Für diese Messungen wurde die CNAPS in die entsprechende Anzahl Subvektoren

	Subvektoren					
	3	5	10	15	20	25
ein Netz	3.6	3.6	3.5	3.2	3.3	2.7
mehrere Netze	10.8	18	34.8	48.4	66.8	68.9

Tabelle 5.7: Leistungsdaten von Algorithmus II für allgemeine Festkommadarstellungen (in MCUPS)

zerteilt und die Zeit für die Berechnung eines Netzes gemessen. Die dabei erreichte Leistung ist in der ersten Zeile eingetragen. Bei einer Aufteilung in z.B. 20 Subvektoren können aber nicht nur ein sondern gleichzeitig 20 Netze berechnet werden. Die dadurch erreichte Gesamtleistung der CNAPS ist in der zweiten Zeile eingetragen.
Die Werte in Tabelle 5.7 wurden mit einer Veränderungsroutine von 34 Befehlen Größe gemessen. Bei einer Verwendung der günstigeren Skalierung mit kleinerem Schleifenrumpf im Veränderungsschritt dürften die Leistungsdaten aus Tabelle 5.8 erreicht werden.

	Subvektoren					
	3	5	10	15	20	25
ein Netz	3.67	3.67	3.64	3.4	3.6	3.0
mehrere Netze	11.0	18.4	36.4	51.0	72.0	75.0

Tabelle 5.8: Leistungsdaten von Algorithmus II für die spezielle Festkommadarstellung (in MCUPS)

Bemerkenswert hierbei ist, daß das einzelne Netz bei einer Verkleinerung der zur Verfügung stehenden Prozessorzahl von 147 auf 20 (also $\frac{1}{7}$) nur um den Faktor 1.33 mehr Trainingszeit benötigt. Der eigentlich zu erwartende Faktor 7 spiegelt sich aber nur in der eigentlichen Verbindungsberechnungszeit wieder ($\frac{700}{100}$). Diese schlechte Skalierung mit der Dichte der Neuronen ist eine Auswirkung der Tatsache, daß bei einer extrem niedrigen Zyklenzahl zur Verbindungsberechnung die Kontrolle der Berechnung bei kleinen Dichten einen vollkommen unproportionalen Teil der Zeit verschlingt, solange sie nicht von einem dedizierten schnellen sequentiellen Prozessor ausgeführt wird oder, wie bei Algorithmus I, in Hardware gegossen und damit inflexibel ist. Dieses wird deutlich, wenn man sich das Verhältnis von Verbindungsberechnung zu den Kontrollroutinen der Verbindungsberechnung ((1) : (2)+(3)) anschaut: Dieses beträgt bei **d**=(1,1,1) ca. 1:4 und und erreicht erst bei einer Dichte **d**=(6,5,8) ein akzeptables Verhältnis von 2:1.

5.4.6 TDNNs

Die Implementierung des TDNNs für diesen Algorithmus weist keine Besonderheiten wie in Algorithmus I auf. Da hier die Neuronen der Eingabeschicht auch mit mehr als einem Neuron pro Prozessor verteilt sein können, kann zur Verteilung der Eingabevektoren für die einzelnen Eingabeschichten keine Nachbarschaftskommunikation genutzt werden. Stattdessen wird das komplette Eingabefenster eingelesen und die einzelnen Eingabevektoren innerhalb des Prozessorvektors über den Bus an die jeweiligen Eingabeschichten verteilt. Der Speicherbedarf wird wie für MLPs errechnet, wobei hier natürlich berücksichtigt werden muß, daß die Matrizen zwischen der versteckten und der Ausgabe-Schicht mehrfach vorhanden sind. In der Tabelle 5.9 sind einige Maximalgrößen für berechenbare TDNNs angegeben.

5.4.7 Realisierung des Algorithmus

Der oben beschriebene Algorithmus wurde wie Algorithmus I in CNAPS-C mit den in Kapitel 3 dargelegten Darstellungen der einzelnen Werte des Backpropagation-Algorithmus implementiert. Um ein möglichst kleines CNAPS-Programm schreiben zu müssen (der Programmspeicher der CNAPS ist extrem klein und die Übersetzungszeiten des Compilers für Programme, die mehr als ein paar hundert Zeilen umfassen, inakzeptabel hoch), wurde nur die eigentliche Berechnung des Netzes auf der CNAPS implementiert. Als Eingabe dient diesem CNAPS-Kernprogramm eine Datei, in der die Berechnung des Netzes in einer Befehlsliste durch eine Reihe von Matrix-Vektor-Operationen beschrieben ist und in der die Gewichtsmatrizen schon so aufbereitet liegen, daß sie direkt in die Prozessoren geladen werden können.
Diese Datei wird von einem Aufbereitungsprogramm erzeugt, das aus einer gegebenen Netzbeschreibung mit initialen Gewichten und einem vorgegebenen Layout (d.h. Schichtgrößen, Anzahl der Subvektoren,**d** und **b**), die nötige Liste an Matriz-Vektor Kommandos erstellt, das Netz einbettet und die in den jeweiligen Prozessoren zu den einzelnen Zeitpunkten im Algorithmus benötigten Gewichte berechnet. Zusätzlich kann dieses Programm auch die speicherminimalen Layouts einer gegebenen Netzbeschreibung für eine bestimmte Menge an Subvektoren oder Prozessoren, sowie

Gewichte	Netzstruktur				
	100-n-100	300-n-300	500-n-500	300-n-100	500-n-50
	#Time-Delays = 3				
16 Bit:	2048	512	418	1024	1200
24 Bit:	1024	278	278	512	804
32 Bit:	1024	209	209	512	530
	#Time-Delays = 5				
16 Bit:	1024	277	277	1024	1024
24 Bit:	512	184	184	512	594
32 Bit:	512	138	138	512	512
	#Time-Delays = 7				
16 Bit:	1024	206	206	512	935
24 Bit:	512	138	138	512	512
32 Bit:	512	103	103	125	477

Tabelle 5.9: Berechenbare Netzgrößen für TDNNs mit Algorithmus II

das Layout für die größtmögliche Anzahl an gleichzeitig zu trainierenden Netzen bestimmen.
Der Quellcode für den CNAPS-Kern hat eine Größe von ungefähr 4300 CNAPS-C-Codezeilen mit zusätzlichen 900 Zeilen Microcode, und benötigt zum Übersetzen auf einer HP-Workstation ca. 19 Minuten. Eine einzige Übersetzung mit Debugroutinen und Probeausgaben (also die typische Übersetzung während der Programmentwicklung) dauert 37 Minuten (!) und erzeugt ein ausführbares Programm von ungefähr 55000-59000 Befehlszeilen (zur Erinnerung: die maximal mögliche Anzahl an Befehlen ist 65536). Der Quellcode für das Aufbereitungsprogramm hat ungefähr 3900 C-Codezeilen und wird von der gleichen HP-Workstation mit einem C-Übersetzer (cc) in nur 6 sec. compiliert.

5.5 Vergleich der beiden Algorithmen

Zum Abschluß wollen wir die Eigenschaften der beiden implementierten Algorithmen zusammenfassen und die beiden Verfahren einander gegenüberstellen:

Algorithmus I:

Vorteile:

- hohe Geschwindigkeit
- hohe Effizienz für große Netze
- kleines verständliches Programm, nur kleine Schleifen in Microcode
- geringe Übersetzungszeit (1 min)
- volle Ausnutzung der hardwaregegebenen Kommunikationsmechanismen

Nachteile:

- nur moderat große Netze berechenbar (doppelte Gewichtsspeicherung, festgelegte Gewichtsverteilung)
- niedrige Effizienz für kleine Netze

Algorithmus II:

Vorteile:

- auch sehr große und ungleichmäßig geschichtete Netze berechenbar
- hohe Effizienz für kleine und mittlere Netze bei hohen Dichten durch Netzparallelität

Nachteile:

- Langsam (2-Bit Nachbarschaftskommunikation, langsames CNAPS-C Compilat, Overhead durch Schleifenköpfe bei niedrigen Neuronen-Dichten)
- hohe Übersetzungszeit (37 min !)
- hohe Größe des compilierten Programmes (55000 - 59000 Befehle von 64K möglichen, somit schlechte Erweiterbarkeit)
- viele komplizierte Schleifen in Mikrocode

5.6 Wertung der Algorithmen

Algorithmus I ist aufgrund der Ausnutzung der von der CNAPS zur Verfügung gestellten Kommunikationsstruktur in der Lage, MLPs mittlerer bis großer Ausprägung in wesentlich schnellerer Zeit, als auf normalen Workstations üblich, zu trainieren. Das Trainieren von TDNNs hingegen ist nur mit sehr eingeschränkten Größen möglich, da der recht geringe Speicher pro Prozessor in Zusammenhang mit der festen Gewichtszuordnung enge Grenzen setzt. Auch für das Einlernen von kleinen Netzen ergeben sich Schwächen, da dabei ein großer Teil der Prozessoren brachliegt und das Potential der CNAPS nicht annähernd ausgeschöpft wird.
Algorithmus II hat die Fähigkeit, dieses Potential zu nutzen, indem er durch die flexiblere Verteilung der Gewichte größere Netze ermöglicht und durch Netzparallelität auch normalerweise brachliegende Prozessoren einsetzen kann. Die in ihm erreichte Beschleunigung für viele kleine Netze ist sehr gut, kann aber ebenso durch normale Verteilung der neuronalen Netze auf einem Workstation-Cluster erreicht werden (siehe 4.3.1). Für die großen Netze hingegen ist die erreichte Geschwindigkeit kaum höher als die einer normalen Workstation, da ein großer Teil der Rechenzeit in sequentiellen Kontrollbefehlen, nicht optimiert übersetztem CNAPS-C Code und sehr dürftig ausgelegter Nachbarschaftskommunikation verlorengeht.
Würden die wesentlichen Mankos der CNAPS, nämlich

- die Ausführung der sequentiellen Code-Anteile auf einer der parallelen Einheiten,
- die 2-Bit Nachbarschaftskommunikation,
- das sequentielle Einlesen der Daten über den Bus,
- und der (nicht modulfähige, langsame), nicht optimierende Übersetzer

beseitigt, indem man

- den sequentiellen Code auf einem schnellen sequentiellen vorgeschalteten (ca. 4 mal schnelleren) Prozessor mit eigenem Datenspeicher ausführt,
- 16-Bit Nachbarschaftskommunikation einführt (16 Bit pro Zyklus),
- die einzelnen Prozessoren mit eigenen Bussen an Festplatten anbindet,
- und den Übersetzer bezüglich Optimierung (und natürlich auch Übersetzungszeit und Modulfähigkeit, da sonst eine Programmweiterentwicklung unmöglich ist) wesentlich verbessert,

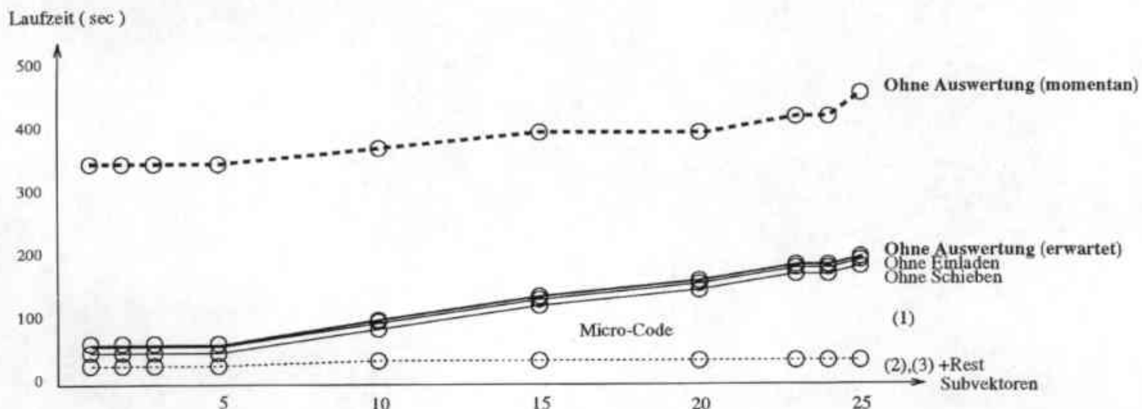


Abbildung 5.15: Erwartetes Laufzeitprofil für das Training mehrerer 112-100-147 Netze

so würde das Laufzeitprofil von Algorithmus II wie in Abbildung 5.15 aussehen.
Aufgrund der lokalen Plattenanbindung würden die Ladezeiten auf die Anzahl an Eingabeneuronen pro Prozessor reduziert. Die Zeit zur Durchführung der inneren Schleifen würde auf ca. ein Viertel der bisherigen Zeit reduziert und der restliche sequentielle CNAPS-Code könnte mindestens viermal schneller ausgeführt werden (wahrscheinlich schneller, da die einzelnen globalen Variablen nicht, wie zur Zeit, umständlich aus den einzelnen Prozessoren geholt werden müßten). Das Schieben der einzelnen Aktivierungen würde um den Faktor 16 beschleunigt werden. Einzig der Code zur Auswertung (und Fehlerberechnung der Ausgabeschicht) kann nur relativ wenig beschleunigt werden, da er ja flexibel gehalten werden und in CNAPS-C geschrieben werden muß. Auf diese Weise wären die Zeiten für Algorithmus II auf den wirklich parallelen Teil reduziert und der sequentiell bedingte Overhead beseitigt, so daß ein Vergleich dieses Verfahrens mit auf anderen Rechnern implementierten *Vertical Slicing*-Ansätzen über Nachbarschaftskommunikation [MBK+92b] [MBK+92a] [Kol94] [Mac92] auch für die Verteilung mit kleinen Neuronen-Dichten konkurrenzfähig wäre.
Nach Aussagen der Marketing-Manager von Adaptive, die zu Beginn des Jahres ihre Kunden in Europa besucht und die Entwicklungslinie für die nächsten zwei Jahre vorgestellt haben, ist der Einsatz eines sequentiellen Prozessors schon einmal angedacht worden. Die Anbindung der einzelnen Prozessoren des Prozessorfeldes an lokale Platten soll ab der nächsten Chip-Generation verwirklicht werden und auch ein modulfähiger, optimierender Übersetzer ist zur Zeit in der Entwicklung.
Fraglich ist hierbei allerdings inwieweit der Einsatz eines sequentiellen Prozessors auch verwirklicht wird und in welchem Zeitraum die übrigen Verbesserungen für das CNAPS-System erwartet werden können.

6. Resümee

6.1 Ergebnisse

In dieser Arbeit wurden die Möglichkeiten der Implementierung neuronaler Netze auf dem CNAPS-Neurorechner untersucht. Die Untersuchungen gliederten sich hierbei in zwei wesentliche Teile: zum einen in die Experimente zur Festkommaimplementierung von neuronalen Netzen und zum anderen in die Untersuchungen zur prinzipiellen Eignung der CNAPS für das Training von neuronalen Netzen, insbesondere von MLPs und TDNNs mit hohen Schichtgrößen.
In den Experimenten zur Festkomma-Implementierung hat sich gezeigt, daß eine Darstellungsgenauigkeit der Gewichte von 16 Bit für die Aufgabe der Phonemerkennung im Resource-Management-Task nicht ausreicht, sondern 24 Bit für eine optimale Erkennungsleistung erforderlich sind. Dieses Ergebnis steht im Widerspruch zu den bisher publizierten Ergebnissen, in denen allgemein eine Genauigkeit von 16 Bit als hinreichend erachtet wird. Für diese Diskrepanz sind zwei wesentliche Gründe anzuführen: Einerseits unterscheidet sich die Größe der Mustermenge beim Resource-Management-Task um einige Größenordnungen von denen der in der Literatur betrachteten Probleme. Andererseits sind während des Trainings für eine optimale Erkennungsleistung sehr niedrige Lernraten erforderlich. Eine Tatsache, die besonders für Festkommaimplementierungen mit niedriger Auflösungsgenauigkeit kritisch ist.
In den Untersuchungen zur prinzipiellen Eignung der CNAPS für das Training von neuronalen Netzen hat sich gezeigt, daß die CNAPS weniger ein allgemeiner Parallelrechner ist, als vielmehr ein Rechner, der für die Multiplikation von vorzeichenlosen 8-Bit Vektoren mit vorzeichenbehafteten 16-Bit Matrizen mittels eines speziellen Algorithmus konzipiert wurde. Aufgrund dieser Tatsache ist die Berechnung und das Training von mehrschichtigen MLPs mit sigmoiden Aktivierungfunktionen und 16-Bit Gewichten um wesentliches schneller, als auf jedem derzeit erhältlichen Rechner. Für allgemeine Aktivierungsfunktionen und die Verwendung von 32-Bit Gewichten müssen jedoch schon Leistungseinbußen um den Faktor 4 hingenommen werden. Auch bei den Größen der MLPs müssen hier Abstriche gemacht werden, da der lokale Speicher mit 4KB relativ niedrig ist und durch den Algorithmus eine Doppelspeicherung von vielen Gewichten erforderlich ist. Noch weit größere Einschränkungen ergeben sich hieraus für das Training von TDNNs. Da bei TDNNs mehrere verschiedene Gewichtsmatrizen zwischen versteckter Schicht und Ausgabeschicht existieren, also genau jene Matrizen, die doppelt gespeichert werden müssen, sind hier nur noch sehr eingeschränkte Schichtgrößen möglich. Zusätzliche Schwierigkeiten entstehen hierdurch auch bei Verwendung anderer Lernverfahren und anderer Gewichtsänderungsstrategien. Im Gegensatz zum normalen *Backpropagation*-Algorithmus mit Musterlernen müssen hier nämlich häufig zwei oder mehr Werte pro Verbindung gespeichert werden, was dann aufgrund der Beschränkung des Speichers sich in noch viel stärkerem Maße auf die Schichtgrößen auswirkt.
Aber es sind nicht nur speicherbedingte Probleme, die die Implementierung anderer Lernverfahren erschweren. Es sind auch Fragen des Systemdesigns und der Rechnerarchitektur, die hier Probleme bereiten. Insbesondere dadurch, daß der Microcode und die Prozessoren in ihrer Auslegung vollkommen auf Algorithmus I zugeschnitten sind, und daß die einzige Kommunikationsmöglichkeit der Prozessoren untereinander und mit dem Dateispeicher ein einzelner exklusiv verwendbarer Bus

ist, ist eine effiziente Implementierung anderer Lernverfahren und Fehlerfunktionen nur schwer möglich.
Weitere Probleme ergeben sich aber auch aus der Zahlendarstellung der CNAPS durch (16-Bit) Festkommazahlen. Hierdurch sind nur beschränkte Wertebereiche in recht geringer Genauigkeit realisierbar, was Division und Multiplikation mit sehr hohen oder sehr niedrigen Zahlen, sowie die Anwendung unbeschränkter transzendenter Funktionen (wie z.B. $exp(x)$, $log(x)$) nahezu unmöglich macht.
Als letztes Problem sei noch das Softwaresystem genannt. Das Fehlen von Debugging-Möglichkeiten auf höheren Ebenen als Microcode, ein Compiler der nur 16-Bit Zahlen kennt, nicht optimierend und nicht modulfähig ist und zur Übersetzung eines durchschnittlich großen Programmes 37 min. braucht, sowie die Tatsache, daß wesentliche Teile von Programmen, um akzeptable Leistung zu erreichen, in Microcode geschrieben werden müssen, lassen die Systementwicklung auf der CNAPS zu einem sehr langwierigen (und nervenaufreibenden) Prozeß werden.
Abschließend läßt sich also sagen, daß die CNAPS für die Berechnung und das Training von zwei- und dreischichtigen, vollverbundenen *Backpropagation*-Netzen mit relativ hohen Lernraten, sehr gut geeignet ist. Für die effziente Implementierung anderer Lernverfahren, hoher Netzgrößen, anderer Netzarten und komplizierterer Fehlerfunktionen ist sie jedoch aufgrund des beschränkten Speichers, der Festkommadarstellung, der sehr spärlich ausgestatteten Prozessoren und des ebenso spärlichen Kommunikationssystems kaum geeignet.

6.2 Ausblick

In dieser Arbeit wurden im Kapitel zur Darstellungsgenauigkeit von Gewichten nur drei Aufgabenstellungen für MLPs betrachtet. Hier sind sicher noch wesentlich intensivere Versuche und vor allem auch statistische Analysen notwendig, um die exakten Abhängigkeiten von Mustermengengröße, Lernraten und der Darstellungsgenauigkeit sowohl für Gewichte, als auch anderer Variablen des *Backpropagation*-Algorithmus festzustellen. Von besonderem Interesse ist hier, welche exakten Bitgenauigkeiten für die einzelnen Aufgabenstellungen nötig sind und inwieweit sich diese mit einer höheren Anzahl an versteckten Neuronen oder vielleicht auch Verfahren wie *Stochastic weight update* vermindern lassen. Wichtig ist dies auch im Hinblick auf TDNNs, da bei der Aufgabe der Phonemerkennung auf dem Resource-Management-Task in einer Fließkommaimplementierung noch niedrigere Lernraten für optimale Erkennungsquoten, als für MLPs verwendet werden müssen, sodaß hier eventuell noch höhere Genauigkeiten für Gewichte als 24 Bit notwendig sind. In diesem Fall würden dann auch die Darstellungen anderer Zahlen im Backpropagation-Algorithmus relevant werden (vor allem die Variable *learndelta*, das Produkt aus Lernrate und rückpropagiertem Fehler, dürfte für niedrige Lernraten kritisch sein).
Ein weiteres Ziel zukünftiger Aktivitäten sollte auch die Suche nach einer Rechnerarchitektur sein, die, im Gegensatz zur CNAPS, zum einen vom Gewichtsspeicher her für MLPs und TDNNs mit großen Schichten ausgelegt ist (hierbei sollte nicht nur der Bereich der Spracherkennung, sondern auch die Bild- und Bildfolgenverarbeitung in Erwägung gezogen werden) und zum anderen auch die Möglichkeit zur effizienten Implementierung anderer Lernverfahren und Fehlerfunktionen besitzt. Eine gute Möglichkeit stellt hier sicher die Verwendung von DSPs dar, die zu recht geringen Kosten gute Leistung bieten [MBK+92b] [MBK+92a]. Bei diesen Rechnern entfallen sowohl die Problematik der Festkommaimplementierung, als auch die der zu geringen Gewichtsspeichergrößen. Zudem dürfte sich, aufgrund der langen Erfahrung mit DSPs, hier auch die softwaretechnische Seite besser gestalten. Eine zweite Möglichkeit wäre aber auch die Implementierung von TDNNs auf der Synapse von Siemens [RRA+93]. Dieser Rechner verfügt über eine ähnliche Maximalleistung wie die CNAPS, besitzt aber wesentlich mehr Gewichtsspeicher, kann teilweise mit höheren Genauigkeiten rechnen und ist vom Gesamtsystem her besser ausgestattet, als dies bei der CNAPS der Fall ist.

Literaturverzeichnis

[Ada93] Adaptive Solutions, Inc.
CNAPS Manuals 1-5
Adaptive Solutions, Inc., 1400 N.W. Compton Drive, Suite 340,Beaverton, OR 97006, Version 3.0, Oktober 1993

[BH88] Tom Baker und Dan Hammerstrom
Modifications to Artificial Neural Networks Models for Digital Hardware Implementation
Technical report, Dept. of Computer Science and Engineering, Oregon Graduate Center, 1988

[BM89] F. Baiardi und R. Mussardo
Parallel Implementation of a Multi-Layer Perceptron
Neurocomputing, F68:161–166, 1989

[EMS90] Hans Peter Ernst, Branislav Mokry und Zoltan Schreter
A Transputer Based General Simulator for Connectionist Models
In *Parallel Processing in Neural Systems and Computers*, S. 283–286, 1990

[Fah88] Scott E. Fahlmann
Faster Learning Variations on Back-Propagation : An Empirical Study
In D. E. Touretzky, Hrsg., *Proceedings of the 1988 Connectionist Summer School.* Morgan Kaufmann, 1988

[Fly72] M. Flynn
Some Computer Organizations and Their Effectiveness
IEEE Transactions on Computers, C-21(9):948–960, 1972

[HF91] Marcus Hoehfeld und Scott E. Fahlmann
Learning with Numerical Precision Using the Cascade-Correlation Algorithm
Technical report, Carnegie Mellon University, Pittsburgh, 1991

[HH90] Jordan L. Holt und Jenq-Neng Hwang
Finite Precision Error Analysis of Neural Network Hardware
Technical report, Department of Elect. Engr., FT-10, University of Washington,Seattle, WA 98195, 1990

[HSH91] D. Hush, J. Salas und B. Horne
Error Surfaces for Multilayer Prceptrons
IEEE International Joint Conference on Neural Networks, Seattle, I:759–764, Juli 1991

[Kol94] Detlef Koll
Untersuchung effizienter Methoden der Parallelisierung neuronaler Netze auf SIMD Rechnern
Diplomarbeit, Universität Karlsruhe, Februar 1994

[Mac92] Niels Mache
Entwicklung eines massiv parallelen Simulatorkerns für neuronale Netze auf der MasPar MP-1216
Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Februar 1992

[MBK+92a] Nelson Morgan, James Beck, Phil Kohn, Jeff Bilmes, Eric Allman und Joachim Beer
The Ring Array Processor: A Multiprocessing Peripheral for Connectionist Approaches
Journal of Parallel and Distributed Computing, 14:248–259, 1992

[MBK+92b] Urs A. Muller, Bernhard Baumle, Peter Kohler, Anton Gunzinger und Walter Guggenbuhl
Achieving Supercomputer Performance for Neural Net Simulation with an Array of Digital Signal Processors
IEEE Micro, 12:55–65, Oktober 1992

[McC90] Hal McCartor
Back Propagation Implementation on the Adaptive Solutions CNAPS Neurocomputer Chip
In *Neural Information Processing Systems 3*, S. 1029–1031, 1990

[MP69] Marvin Minsky und Papert
Perceptrons : An Introduction to Computational Geometry
MIT Press, Cambridge, Massachusetts, 1969

[NF89] Fernando J. Nuñez und Jose A. B. Fortes
Performance of Connectionist Learning Algorithms on 2-D SIMD Processor Arrays
In *Neural Information Processing Systems 2 (NIPS 2)*, S. 810–817, 1989

[PM92] Hélène Paugam-Moisy
Optimal Speedup Conditions for a Parallel Back-Propagation Algorithm
In *CONPAR 92 - VAPP V*, September 1992

[PNH86] D. Plaut, S. Nowlan und G. Hinton
Experiments on Learning by Back Propagation
Technical Report CS-86-126, Carnegie Mellon University, Pittsburgh, 1986

[RHW86] D. E. Rumelhart, G. E. Hinton und R. J. Williams
Learning Internal Representation by Error Propagation
In *Parallel Distributed Processing*, Kapitel 7, S. 318–362. MIT Press, Cambridge, Massachusetts, 1986

[Rie92] Martin Riedmiller
Schnelle adaptive Lernverfahren für mehrschichtige Feedforward-Netzwerke - Vergleich und Weiterentwicklung -
Diplomarbeit, Universität Karlsruhe, Februar 1992

[RRA+93] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann und M. Weßeling
Synapse - A General-Purpose Neurocomputer
Technical report, Siemens Corporate Research and Development Devision, Februar 1993

[Sin90a] Alexander Singer
Exploiting the Inherent Parallelism of Artificial Neural Networks to Achieve 1300 Million Interconnects per second
In *INNC*, S. 656–660, 1990

[Sin90b] Alexander Singer
Implementations of Artificial Neural Networks on the Connection Machine
Parallel Computing, 14:305–315, 1990

[Smo89] Ira G. Smotroff
Dataflow Architectures : Flexible Platforms for Neural Network Simulation
In *Neural Information Processing Systems 2 (NIPS 2)*, S. 818–825, 1989

[SSA+93] Yuji Sato, Katsunari Shibata, Mitsuo Asai, Masaru Ohki, Mamoru Sugie, Takahiro Sakaguchi, Masashi Hashimoto und Yoshihiro Kuwabara
Development of a High-Performance, General Purpose Neuro-Computer Composed of 512 Digital Neurons
In *International Joint Conference on Neural Networks*, S. 1967–1970, 1993

[Thi90] Thinking Machines Corporation
C Reference Manual, Version 6.0*
Thinking Machines Corp., Cambridge, MA, 1990

[Ung93] Theo Ungerer
Parallelrechner und Parallelprogrammierung
Universität Karlsruhe, Fakultät für Informatik, Oktober 1993

[Vir93] Marc A. Viredaz
Mantra I: An SIMD Processor Array for Neural Computation
In Peter Paul Spies, Hrsg., *Euro-ARCH*, S. 99–110. Springer, Oktober 1993

[WHH+89] Alexander Waibel, Toshiyuki Hanazawa, Geoffry Hinton, Kiyohiro Shikano und Kevin Lang
Phonem Recognition using Time-Delay Neural Networks
In Alex Waibel und Kai-Fu Lee, Hrsg., *Readings in Speech Recognition*, Kapitel 7, S. 393–404. Morgan Kaufmann, San Mateo, California, 1989

[WL89] Alexander Waibel und Kai Fu Lee
Connectionist Approches
In *Readings in Speech Recognition*, Kapitel 7, S. 393–404. Morgan Kaufmann, San Mateo, California, 1989

[WZ90] Michael Whitbrock und Marco Zagha
An Implementation of Backpropagation Learning on GF11, a Large SIMD Parallel Computer
Parallel Computing, 14:329–345, 1990

[ZMMV93] Andreas Zell, Günter Mamier, Niels Mache und Michael Vogt
Simulation Neuronaler Netze auf SIMD und MIMD-Parallelrechnern
In Peter Paul Spies, Hrsg., *Euro-ARCH*, S. 111–122. Springer, Oktober 1993

[ZMMW89] Xiru Zhang, Michael Mckenna, Jill P. Mesirov und David L. Waltz
An Efficient Implementation of the Back-propagation Algorithm on the Connection Machine CM-2
In *Neural Information Processing Systems 2 (NIPS 2)*, S. 801–809, 1989