Parallele Backpropagation

Diplomarbeit

Ulrich Middelberg

Universität Osnabrück
Fachbereich Mathematik / Informatik
Arbeitsgruppe Praktische Informatik
Prof. Dr. Oliver Vornberger

Betreuer: Dipl.-Math. Frank M. Thiesing

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Hiermit versichere ich, Ulrich Middelberg, die vorliegende Arbeit „Parallele Backpropagation“ selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Osnabrück, den 11. September 1995

Ulrich Middelberg

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mir bei der Erstellung dieser Arbeit zur Seite standen.

Ich danke Herrn Prof. Dr. Oliver Vornberger für eine Reihe von Anregungen und Vorschlägen zum Inhalt und zur Ausführung der Arbeit.

Bei Herrn Dipl.-Math. Frank M. Thiesing möchte ich mich ganz besonders für eine intensive Betreuung und viele praktische Hinweise bedanken, die mir die Entwicklung dieser Arbeit erleichtert haben.

Herrn Dr. Frank Lohmeyer und Herrn Dipl.-Inform. Volker Schnecke danke ich für eine Menge Tips und Ratschläge, durch die ich so manches Problem lösen konnte.

Für die Benutzung der Parallelrechner CGel und GC/PP danke ich dem Paderborn Center for Parallel Computing (PC^2).

Kapitel 1

Einleitung

Kaum ein anderer Wissenschaftszweig hat sich in den letzten Jahren einem so großen Wandel unterzogen wie der der künstlichen neuronalen Netze. In den fünfziger Jahren wurden sie von Neurobiologen entworfen, um das Zusammenspiel natürlicher Nervenzellen zu modellieren. Inzwischen haben jedoch die Forschungsaktivitäten den ursprünglichen Kreis verlassen. Heutzutage stellt die Beschäftigung mit künstlichen neuronalen Netzen ein Paradebeispiel interdisziplinärer Forschung dar. Während Psychologen, Informatiker und Mathematiker eher an den Grundlagen und der Funktionsweise interessiert sind, versuchen vor allem Ingenieure und Wirtschaftswissenschaftler sie für ihre Zwecke gewinnbringend anzuwenden.

Neben Klassifikation und Mustererkennung werden künstliche neuronale Netze auch zur Zeitreihenprognose eingesetzt. Ein großer Vorteil gegenüber herkömmlichen Methoden besteht darin, daß durch das Training ein künstliches neuronales Netz aus einer exemplarischen Problembeschreibung heraus wesentliche Zusammenhänge extrahiert.

Mitunter erweist sich die Trainingsphase als sehr zeitintensiv, weshalb vielfach Anstrengungen unternommen werden, die Dauer zu verkürzen. Neben qualitativen Verfahren, die es erlauben, das Training in weniger Lernschritten abzuwickeln, bietet sich der Natur künstlicher neuronaler Netze folgend eine parallele Abarbeitung des Lernverfahrens an.

Als ein sehr flexibel einsetzbarer Typ künstlicher neuronaler Netze hat sich das Mehrschicht-Perzeptron-Netz erwiesen. Backpropagation ist ein Lernverfahren, mit dem derartige Netze trainiert werden.

Im Rahmen der hier vorliegenden Arbeit entstand ein Programm zum Training eines Mehrschicht-Perzeptron-Netzes mit dem Backpropagation-Algorithmus. Um die Lerndauer zu verringern, wurde zum einen die Trainingsphase auf verschieden Arten parallelisiert und diese miteinander verglichen, zum andern eine qualitative Verbesserung des Lernverfahrens integriert.

Eine thematische Übersicht dieser Arbeit zeigt Abbildung 1.1. Die wichtigsten Konzepte und Vokabeln aus dem Bereich der künstlichen neuronaler Netze werden im folgenden Kapitel dargestellt. Bausteine eines künstlichen neuronalen Netzes sind die Neuronen, ihre Funktionalität und Anordnung bestimmt dessen Arbeitsweise.

Kapitel 3 befaßt sich mit dem Mehrschicht-Perzeptron-Netz. Ausführlich wird auf das Training mit dem Backpropagation-Algorithmus eingegangen. Eine qualitative Verbesserung des Backpropagation-Verfahrens, der MouseProp-Algorithmus, wird vorgestellt. Den Abschluß des Kapitels bilden die Grundzüge der Zeitreihenprognose mit Mehrschicht-Perzeptron-Netzen.

Mittelpunkt der Arbeit stellt Kapitel 4 dar. Nachdem Strategien zur Aufteilung der Trainingsaufgabe und deren parallele Abarbeitung vorgestellt und diskutiert werden, geht es um die Implementierung verschiedener Varianten des parallelen Backpropagation-Algorithmus. Hierfür stehen zwei Standardumgebungen der Parallelverarbeitung zur Verfügung. Das parallele Batch-Training wurde unter PVM und unter Parix, das parallele Online-Training unter Parix realisiert. Experimentelle Speedup-Messungen zeigen deutliche Unterschiede der verschiedenen Ansätze.

Kapitel 5 faßt noch einmal die wichtigsten Ergebnisse zusammen und deutet mögliche Ergänzungen in einem Ausblick an. Detaillierte Hinweise zur Installation und zur Bedienung des hier entwickelten Programms `bp` zum Training von Mehrschicht-Perzeptron-Netzen befinden sich in Anhang A.

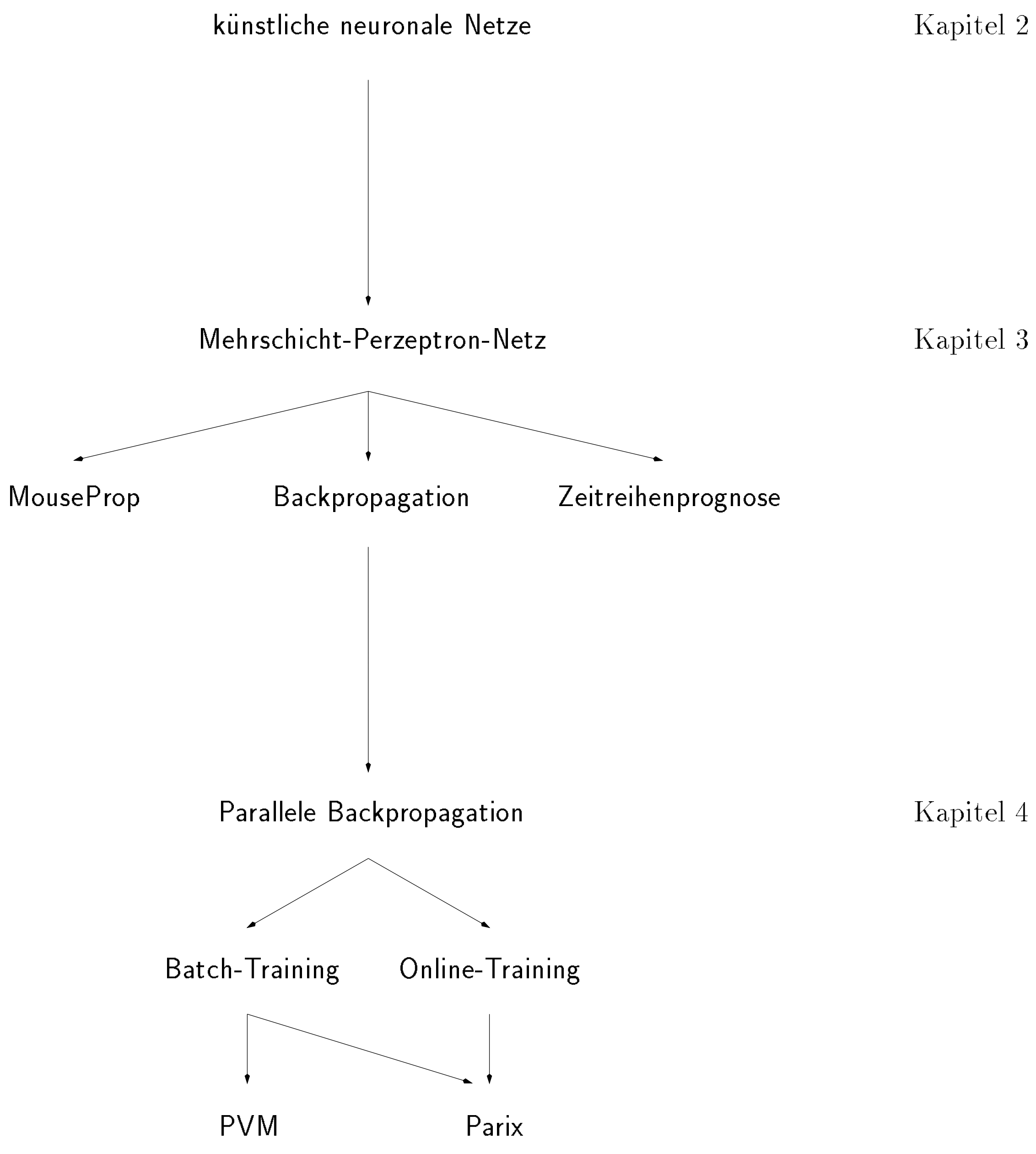


Abbildung 1.1: Thematische Übersicht

Kapitel 2

Künstliche Neuronale Netze

2.1 Überblick

Kaum ein anderes Wissenschaftsfeld hat in den letzten Jahren so viel Interesse auf sich gezogen, wie die Künstlichen Neuronalen Netze (KNN). Dabei ist es nach raschen anfänglichen Erfolgen in den 50er Jahren lange Zeit still um sie gewesen. Erst zu Beginn der achtziger Jahre erfuhren sie dank den Entwicklungen von HOPFIELD [7] und KOHONEN [9] eine Renaissance. Der von RUMELHART et al. 1987 vorgestellte Backpropagation-Algorithmus [20] verhalf den KNN zu ihrem Durchbruch. Sprunghaft entstand eine unüberschaubare Anzahl an möglichen Anwendungen künstlicher neuronaler Netze.

Allen Netzmodellen gemein ist das Grundelement, das *Neuron*. Ein KNN erhält man durch Verschalten mehrerer identischer Neuronen. Dies ergibt eine Netztopologie, die zusammen mit der Arbeitsweise der Neuronen die Funktionalität des KNN bestimmt. Die Benutzung künstlicher neuronaler Netze geschieht in zwei Schritten: Bevor eine *Abfrage* ein „sinnvolles" Ergebnis liefert, wird das KNN anhand von Beispielen *trainiert*.

2.2 Das Neuron als informationsverarbeitendes Element

2.2.1 Ein abstraktes Modell

Die ursprüngliche Motivation der Erforschung der KNN war ein besseres Verständnis der Zusammenhänge des Nervensystems. Mit künstlichen neuronalen Netzen wurden biologische neuronalen Netze als informationsverarbeitende Systeme modelliert. Wesentliche Eigenschaften biologischer Nervenzellen dienten als Vorbild bei dem Entwurf künstlicher Neuronen.

Ein derartiges Neuron besitzt folgenden grundlegenden Aufbau:

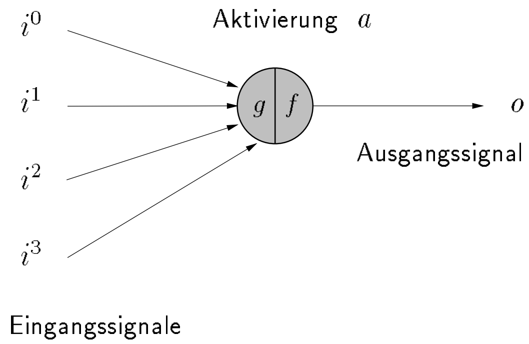


Abbildung 2.1: Ein generisches Neuron

- Mehrere *Eingangskanäle*, über die externe Signale $i^0, \ldots, i^{n-1}$ empfangen werden. Die eingehenden Signale werden in der Regel zu einem Vektor zusammengefaßt.
- Das eigentliche Berechnungselement, welches in einem zweistufigen Verfahren zunächst die *Aktivierung* bestimmt und diese in ein *Ausgangssignal* transformiert.
- Ein *Ausgangskanal*, über den das *Ausgangsignal* zu den Eingangskanälen anderer Neuronen gelangen kann.

Der Informationsfluß in einem künstlichen Neuron beginnt mit der Reduktion des Eingangsvektors zu einem skalaren Wert, der Aktivierung. Diese Reduktion vollzieht sich mit Hilfe einer sogenannten *Integrationsfunktion* g, welche im einfachsten Fall die Argumente aufaddiert:

$$a \quad := \quad g(i^0, \ldots, i^{n-1}) = i^0 + \ldots + i^{n-1}$$

Die *Aktivierungsfunktion* bzw. *Transferfunktion* f liefert das Ausgangssignal:

$$o \quad := \quad f(a) = f(g(i^0, \ldots, i^{n-1}))$$

Hierfür wird oft die die sogenannte Treppenfunktion verwandt:

$$f(x) \quad := \quad \begin{cases} 1 & : \quad \text{für } x \geq 0 \\ 0 & : \quad \text{für } x < 0 \end{cases}$$

Das Ausgangssignal kann sowohl Eingangssignal für weitere Neuronen sein, als auch über eine Rückkopplung dem Neuron selbst wieder zur Verfügung stehen.

2.2.2 Das McCulloch-Pitts Neuron

Die ersten veröffentlichten Arbeiten auf dem Gebiet der KNN stammen von McCulloch und Pitts [10]. Sie stellten zu Beginn der vierziger Jahre ein Netz vor, dessen Neuronen aus binären Schaltelementen bestanden. Die Signale, die ein solches Neuron verarbeitet, sind somit auch binärer Natur.

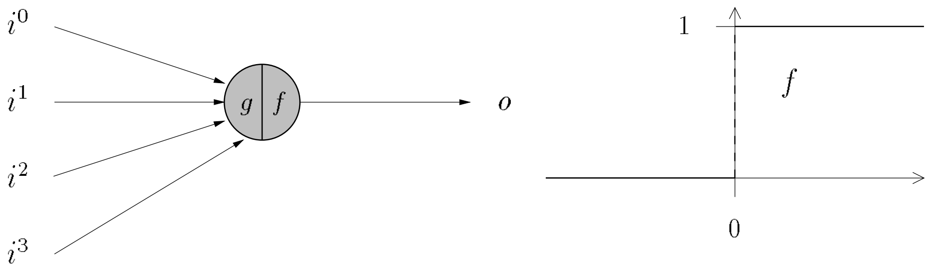


Abbildung 2.2: Ein Neuron von McCulloch und Pitts

Zur Berechnung der Aktivierung wird zusätzlich ein sogenannter Schwellwert θ herangezogen:

$$g(i^0, \ldots, i^{n-1}) \quad := \quad i^0 + \ldots + i^{n-1} - \theta$$

Die Ausgabe errechnet sich wie folgt:

$$o \quad := \quad \begin{cases} 1 & : \quad \text{für } \sum_{j=0}^{n-1} i^j \geq \theta \\ 0 & : \quad \text{für } \sum_{j=0}^{n-1} i^j < \theta \end{cases}$$

Ein derartiges Neuron kann bereits einfache boolesche Funktionen wie OR ($\theta = 1$) oder AND ($\theta = n$) berechnen.

2.2.3 Heutige Neuronen

Ausgehend von dem Modell von McCulloch und Pitts wurden Versuche unternommen, die Funktionalität eines Neurons zu erweitern. Die wesentliche Beschränkung, nur binäre Signale zuzulassen, kann wegfallen, wird statt der Treppenfunktion eine sogenannte Sigmoide zur Transformation verwandt:

$$f_T(x) \quad := \quad \frac{1}{1 + e^{-\frac{x}{T}}}$$

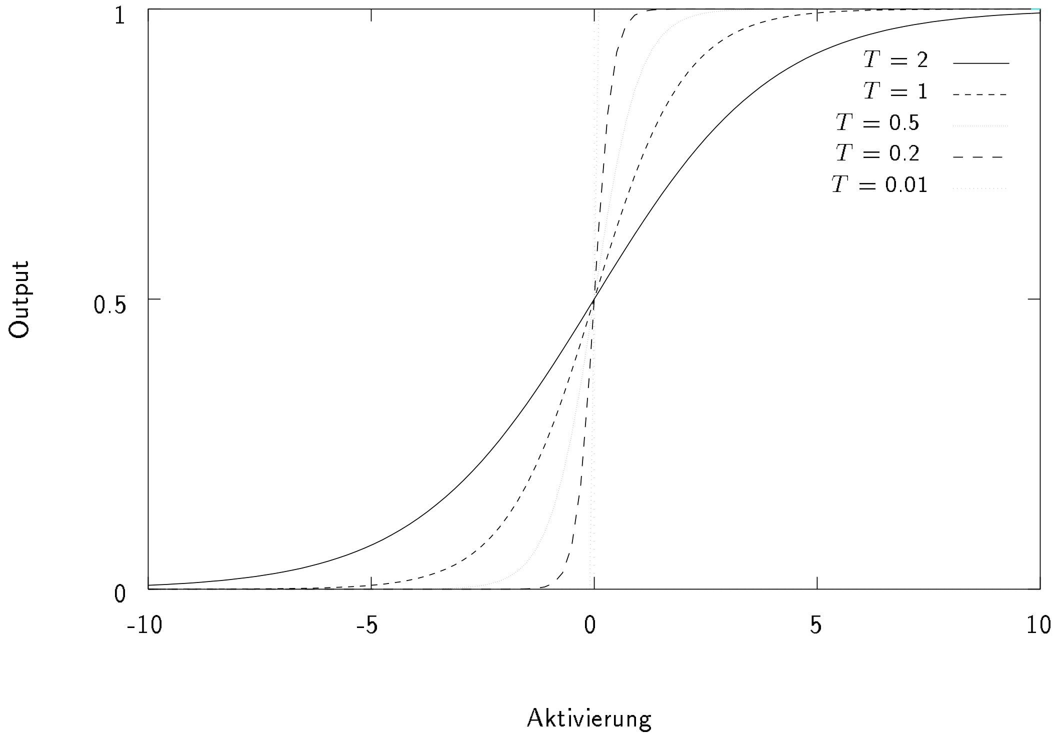


Abbildung 2.3: Sigmoide Funktionenschar

Durch den Parameter T läßt sich die Steigung der Funktion variieren. Große Werte für T haben einen relativ flachen Verlauf zur Folge, während kleine Werte zu einem der Treppenfunktion ähnlichen Graph führen. Abbildung 2.3 zeigt die Funktionsgraphen für verschiedene T.

Ein entscheidender Vorteil dieser Klasse von Transferfunktionen ist die Differenzierbarkeit. Damit kann eine Vielzahl von Verfahren der nichtlinearen Optimierung Anwendung bei den KNN finden. Ein prominentes Beispiel ist der in Abschnitt 3.3 vorgestellte Backpropagation-Algorithmus, der Information über die Ableitung von f benötigt.

Zwei andere häufig verwendete Aktivierungsfunktionen sind der Tangens hyperbolicus und der logistische Sinus. Ihre Funktionsgraphen ähneln dem der Sigmoiden, auch sind sie beide differenzierbar.

Das in Abbildung 2.4 dargestellte Neuron verarbeitet Signale aus dem reellen Intervall $]0,1[$. Dies erleichtert enorm die Kodierung der Problemstellung, da Werte, statt sie in mehrere binäre Signale zu wandeln, nur noch skaliert werden müssen.

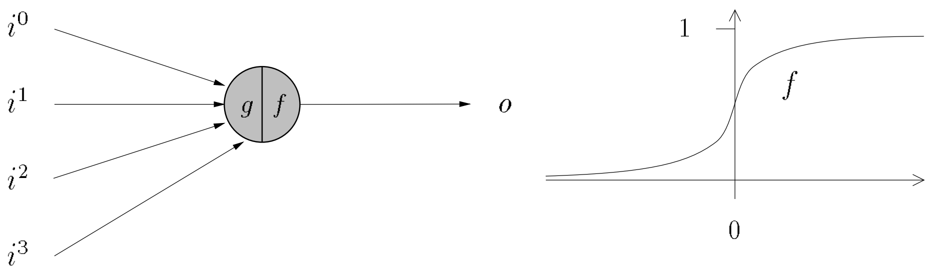


Abbildung 2.4: Ein „modernes Neuron“

2.3 Kopplung von Neuronen: Ein künstliches neuronales Netz

2.3.1 Grundlegende Funktionsweise

Ein KNN ist eine Menge von gleichartigen Neuronen, die über *gewichtete Verbindungen* miteinander vernetzt sind. Dargestellt wird die Topologie als Graph, wobei die Knoten die Neuronen, die Kanten die Verbindungen repräsentieren. In der Regel sind die Verbindungen bidirektional, d.h. sie fungieren beidermaßen als Eingabe- und Ausgabekanal.

Soll nun für ein gewisses Neuron k der Ausgabewert o_k bestimmt werden, so berechnen sich dessen Eingabewerte i_k^j aus der Ausgabe o_j aller mit k verbundenen Neuronen, jeweils multipliziert mit dem Gewicht der Verbindung von j nach k, w_{kj} (siehe Abbildung 2.5):

$$i_k^j = w_{kj} o_j$$

Der Ausgabewert für Neuron k berechnet sich als:

$$o_k = f(a_k) = f(g(w_{k0}o_0, \ldots, w_{kn-1}o_{n-1}))$$

Der Vektor aller Ausgabewerte o_j wird als *Netzzustand* bezeichnet. Bevor ein KNN in Betrieb gesetzt wird, erhalten die Neuronen initiale Ausgabewerte. Dieser Netzzustand stellt die *Eingabe* an das Netz dar und ist Ausgangspunkt einer *Abfrage*.

Der weitere Ablauf hängt von der Dynamik des Netzes ab. Im *synchronen* Modus führen alle Neuronen im Gleichtakt ihren Berechnungen aus. In der Regel steht das Resultat nach einem derartigen Schritt fest, der Netzzustand repräsentiert die *Ausgabe* des KNN.

Im Gegensatz dazu spricht man von einem *asynchronen* Betrieb, wenn jeweils nur ein Neuron ausgewählt wird, seinen Ausgabewert zu berechnen. Diese Vorgehensweise benötigt mehrere

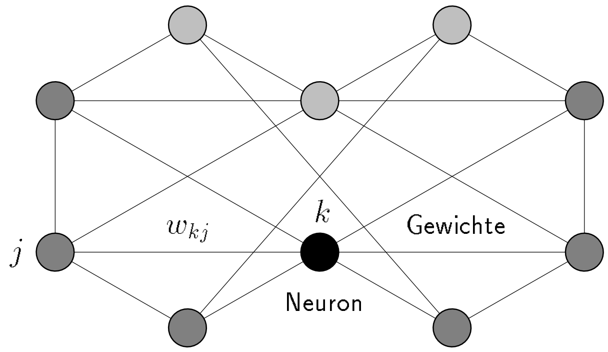


Abbildung 2.5: Darstellung der Topologie eines KNN

Schritte, bis ein Ergebnis feststeht. Erst wenn keines der Neuronen mehr seinen Ausgabewert ändert, hat das Netz einen *stabilen* Zustand erreicht.

Mitunter besitzen nicht alle Neuronen eines KNN die gleiche Funktionalität. Beispielsweise führen *Eingabeneuronen* keine Berechnungen aus, sondern bekommen einen Wert zugewiesen, den sie als ihren Ausgabewert anderen Neuronen zur Verfügung stellen. Dementsprechend reduziert sich die Dimension des Eingabevektors auf die Anzahl der Eingabeneuronen. Eine Verkleinerung des Ausgabevektors erhält man durch Kennzeichnung gewisser Neuronen als *Ausgabeneuronen*, ohne allerdings ihre Funktion einzuschränken.

Zu einer vollen Beschreibung eines KNN gehören neben der Topologie des Netzes auch Angaben zu

- der *Charakteristik* der verwandten Neuronen:
  - Integrationsfunktion g: Sie gibt Aufschluß darüber, ob das Neuron mit einem Schwellwert arbeitet oder nicht.
  - Transferfunktion f: Aus ihr ergibt sich die Art der verarbeiteten Signale.
  - Verwendung von Eingabe- bzw. Ausgabeneuronen
- der *Dynamik* des Netzes: synchroner bzw. asynchroner Betrieb.
- den zur Verfügung stehenden *Trainingsverfahren*.

Aufgrund der beinahe unüberschaubaren Vielfalt von KNN, würde eine tiefergehende Betrachtung den Rahmen dieser Arbeit sprengen. Für einen umfassenden Überblick der verschiedenen KNN Architekturen sei auf [6] und [18] verwiesen.

2.3.2 Training künstlicher neuronaler Netze

Ein Grund für den häufigen Einsatz auf KNN basierender Anwendungen [26] ist die Art und Weise, wie Probleme modelliert werden. Es ist zwar hilfreich, aber nicht notwendig, Annahmen bezüglich der Modellzusammenhänge zu treffen. Statt dessen reicht es aus, Daten zu beschaffen, die das Problem exemplarisch beschreiben.

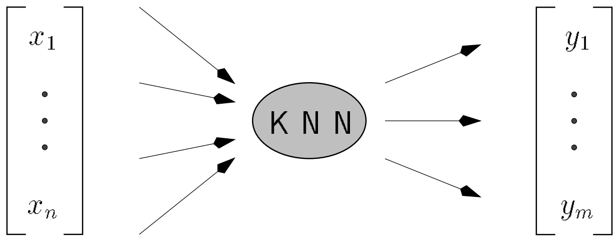


Abbildung 2.6: Informationsfluß in einem KNN

Eine abstrakte Sichtweise des Informationsfluß in einem KNN zeigt Abbildung 2.6. Ein KNN bildet einen n-dimensionalen Vektor x auf einen Vektor y der Dimension m ab. Hierbei sind die Berechnungen, die im Innern des KNN passieren, für den Anwender nicht von Belang.

Das Training künstlicher neuronaler Netze beinhaltet die Anwendung eines Lernverfahrens auf die zur Verfügung stehende Datenmenge. Steht die Problemlösung für die bekannten Daten a priori gar nicht fest (z.B. die Herausbildung von Merkmalen zur Segmentierung einer Menge von Daten), so wird das KNN mit einem *unüberwachten* Lernverfahren trainiert. Sind dagegen die Beispiellösungen bekannt, so kann zu jeder Eingabe explizit der Fehler berechnet werden, der in das weitere Training einfließt. Diese Methode wird als *überwachtes* Lernen bezeichnet.

Ein klassisches Problem hierfür, das exklusive Oder (XOR) wird folgendermaßen für den in Abschnitt A vorgestellten Simulator `bp` beschrieben:

```
## XOR Problem
##
## Wahrheitstabelle:
##
##        | 0 1
##      --|----
##      0 | 0 1
##      1 | 1 0
##
# pairs=      4
# genpairs=   0
```

```
# inputsize=  2
# outputsize= 1
0 0, 0;
0 1, 1;
1 0, 1;
1 1, 0;
```

Für dieses Problem ist der Eingabevektor x zweidimensional, der Ausgabevektor y eine skalare Größe. Zwei miteinander assoziierte Vektoren x,y heißen *Trainingspaar*.

Wird ein KNN zur Approximation eigesetzt (siehe Abschnitt 3.5), so ist das Ziel des Trainings ein KNN, welches für jedes Paar (x, y) bei Eingabe von x ein Resultat y' nahe dem Sollwert y ausgibt. Nahe deshalb, weil ein exaktes Ergebnis gar nicht im Vordergrund steht, sondern vielmehr die Fähigkeit des KNN zur Generalisierung, welche erfahrungsgemäß mit einer zu genauen Reproduktion verloren geht.

Generalisierung bedeutet in diesem Zusammenhang, daß das KNN einem nicht trainierten Eingabevektor einen Ausgabevektor zuordnet, den der Anwender als „sinnvoll" erachtet. In diesem Fall hat das KNN durch das Training die den Beispielen immanenten Zusammenhänge gelernt, die dem Anwender auf den ersten Blick verborgen bleiben.

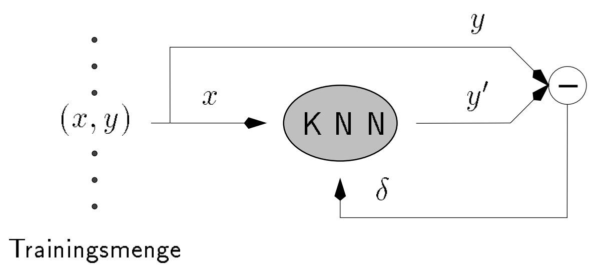


Abbildung 2.7: Überwachtes Lernen in einem KNN

Der eigentliche Lernvorgang geschieht durch Anpassung der Verbindungsgewichte und, falls vorhanden, der Schwellwerte anhand der vom KNN produzierten Fehler (siehe Abbildung 2.7). Für jedes Trainingspaar (x, y) wird zu dem Eingabevektor x die Ausgabe des KNN y' berechnet. Die Abweichung δ vom Sollwert y fließt in die Korrektur der Gewichte ein. Gesucht ist eine Konstellation, die den Fehler über alle Trainingspaare, die die *Trainingsmenge* bilden, minimiert. Das einmalige Präsentieren aller Trainingspaare mit anschließendem Aktualisieren der Gewichte wird als *Epoche* bezeichnet. Ob das KNN nur „auswendig" gelernt hat, zeigt sich nach dem Training bei dem Generalisierungstest, der Validierung. Die *Testmenge* enthält vorhandene aber nicht trainierte Paare, die dem KNN präsentiert werden. Die hierbei beobachtete Abweichung ist ein verläßlicheres Maß für den Lernerfolg als allein der Fehler auf der Trainingsmenge.

Kapitel 3

Das Mehrschicht Perzeptron Netz

3.1 Überblick

Das Mehrschicht Perzeptron Netz (MP-Netz) gehört aufgrund seiner Flexibilität zu den am häufigsten eingesetzten KNN [26], da es sich sowohl zur Klassifikation als auch zur Funktionsapproximation eignet. Gerade letztere Eigenschaft macht das MP-Netz zur Prognose von Zeitreihen interessant, da konventionelle Methoden ein immenses Fachwissen voraussetzen, um Annahmen bezüglich der Modellparameter treffen zu können.

Im folgenden Kapitel wird neben den Grundlagen des MP-Netzes ausführlich auf ein Verfahren zum Training von MP-Netzen, den *Backpropagation-Algorithmus*, eingegangen. Eine Erweiterung des Verfahrens, welche während des Trainings dem Netz Neuronen hinzufügt, wird vorgestellt. Zum Abschluß werden die Grundzüge der Zeitreihenprognose mit MP-Netzen erläutert.

3.2 Aufbau und Funktionsweise

Das Mehrschicht Perzeptron Netz gehört zur Kategorie der *vorwärtsgerichteten* KNN. Diese Netztopologie zeichnet sich durch einen kreisfreien Graphen aus, dessen Kanten gerichtet sind. Es wird somit zwischen eingehenden (rezeptiven) und ausgehenden (projektiven) Verbindungen unterschieden.

Die Neuronen eines MP-Netzes sind, ihrer Funktionalität entsprechend, hierarchisch in Schichten angeordnet. Neuronen, die keine rezeptiven Verbindungen besitzen, werden zur *Eingabeschicht* zusammengefaßt, Neuronen ohne projektive Verbindungen bilden die *Ausgabeschicht*. Neuronen, die sich zwischen diesen zwei Schichten befinden, werden *verborgene* Neuronen genannt und sind in einer oder mehreren verborgenen Schichten organisiert. Verbunden sind ausschließlich Neuronen zweier benachbarter Schichten. Abbildung 3.1 zeigt ein MP-Netz. Die Eingabeneuronen sind, da sie keine Berechnungen ausführen, anders dargestellt: Pfeile deuten die Orientierung der Kanten an.

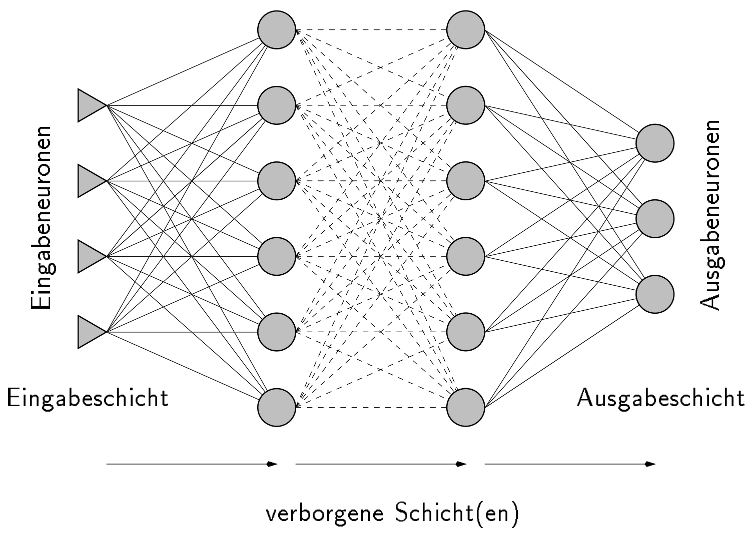


Abbildung 3.1: Topologie eines MP-Netzes

Die verwendeten Neuronen entsprechen dem in Abschnitt 2.2.3 vorgestellten Typ. Es wird mit Schwellwert gearbeitet. Als Transferfunktion kommt die Sigmoide mit $T = 1$ (siehe Abbildung 2.3) zum Einsatz. Die Berechnung der Ausgabewerte geschieht in jeder Schicht synchron, von Schicht zu Schicht sequentiell beginnend mit der ersten verborgenen. Im einzelnen wird die Abfrage des Netzes in Abschnitt 3.3.2 beschrieben.

3.3 Der Backpropagation-Algorithmus

3.3.1 Überblick

Der von Rumelhart et al. [20] vorgestellte Backpropagation-Algorithmus gehört zu der Kategorie überwachter Lernverfahren. Der entscheidende Fortschritt ist die Möglichkeit, auch für die verborgenen Neuronen einen Abbildungsfehler und damit Gewichtskorrekturen berechnen zu können. Vom Standpunkt der nichtlinearen Optimierung her betrachtet, entspricht der Backpropagation-Algorithmus dem Gradientenabstieg, angewandt auf die Besonderheiten eines MP-Netzes [1].

Das Training eines MP-Netzes mit dem Backpropagation-Verfahren beginnt mit der Initialisierung der Gewichte und Schwellwerte mit Zufallwerten aus dem Intervall $]-1, 1[$. Der eigentliche

Algorithmus gliedert sich in zwei sich abwechselnde Schritte. Das Resultat eines Eingabevektors wird in der *Forward-Phase* berechnet. Die *Backward-Phase* dient dazu, den Abbildungsfehler und damit die Gewichtskorrektur für jede Schicht zu bestimmen. Das Anwenden des Backpropagation-Algorithmus auf alle Trainingspaare wird als *Epoche* bezeichnet. Durch wiederholtes Trainieren stellt sich eine fehlerminimierende Konstellation der Gewichte ein.

3.3.2 Forward-Phase

Ausgehend von der ersten verborgenen Schicht werden die jeweils neuen Ausgabewerte für eine Schicht berechnet. Ein Neuron benötigt hierfür die Ausgabewerte der *Vorgänger* sowie die Gewichte der *rezeptiven* Kanten.

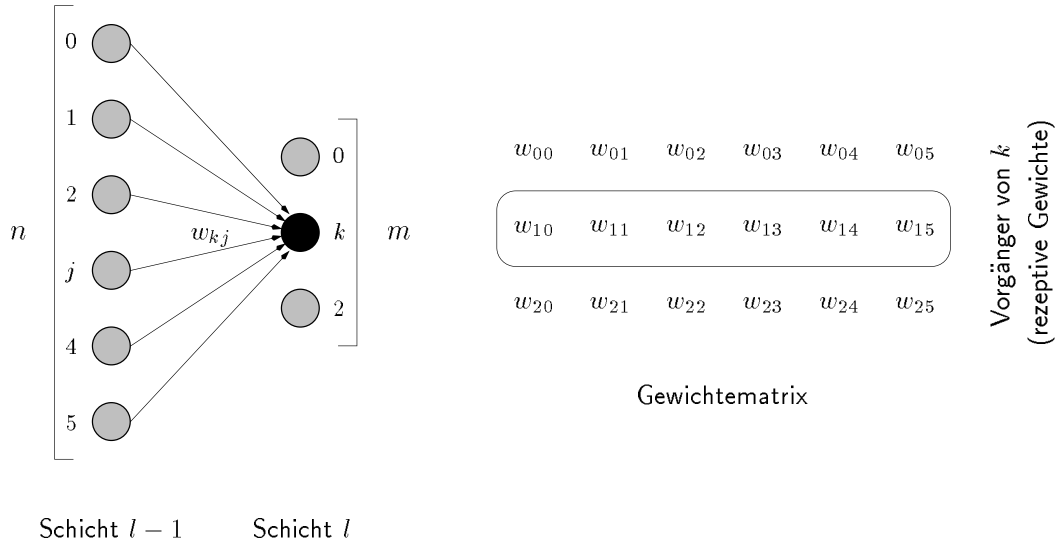
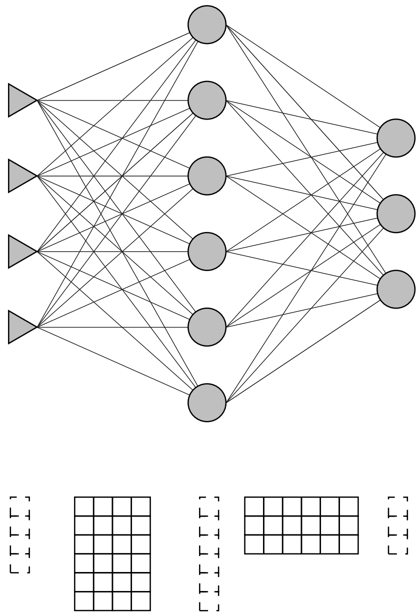


Abbildung 3.2: Forward-Phase

Neuron k in Abbildung 3.2 führt folgende Operationen aus:

$$
\begin{aligned}
a_k &= \left(\sum_{j=0}^{n-1} w_{kj} o_j\right) - \theta_k \\
o_k &= f(a_k) = \frac{1}{1+e^{-a_k}}
\end{aligned}
$$

Diese Schritte können für alle m Neuronen unabhängig voneinander ausgeführt werden. Die für Neuron k relevanten Gewichte befinden sich in der *k-ten Zeile* der Gewichtematrix. Im wesentlichen entspricht die Bestimmung der Aktivierung für alle Neuronen der Schicht l einer Multiplikation der Matrix mit dem Vektor der Ausgabewerte der Schicht $l-1$.



$$i \xrightarrow{h = F(w_h \cdot i)} h \xrightarrow{o = F(w_o \cdot h)} o$$

Abbildung 3.3: Matrix-Vektor-Operationen in der Forward-Phase

Abbildung 3.3 stellt die Gesamtsicht für ein dreischichtiges MP-Netz dar. Die Benennung der Schichten orientiert sich an ihrer Funktionalität: input, hidden und output. Die Schwellwerte können als zusätzliche Gewichte zu einem Neuron mit einer konstanten Aktivierung von 1 aufgefaßt werden. Die Abbildung $F((x_1, \cdots, x_n)) := (f(x_1), \cdots, f(x_n))$ ist eine vektorielle Verallgemeinerung der Transferfunktion f. Das Resultat für einen Eingabevektor x berechnet sich wie folgt:

$$y' = F(w_o \cdot F(w_h \cdot x))$$

3.3.3 Backward-Phase

Die zweite Phase des Backpropagation-Algorithmus dient zur Korrektur der Verbindungsgewichte. Hierfür wird an den Neuronen der Ausgabeschicht das Resultat der Eingabe x mit dem Sollwert y verglichen. Die Differenz ergibt den Fehler δ_k des jeweiligen *Ausgabeneurons*:

$$\delta_k \quad = \quad (y_k - o_k) \ \underbrace{o_k \ (1 - o_k)}_{f'(a_k)}$$

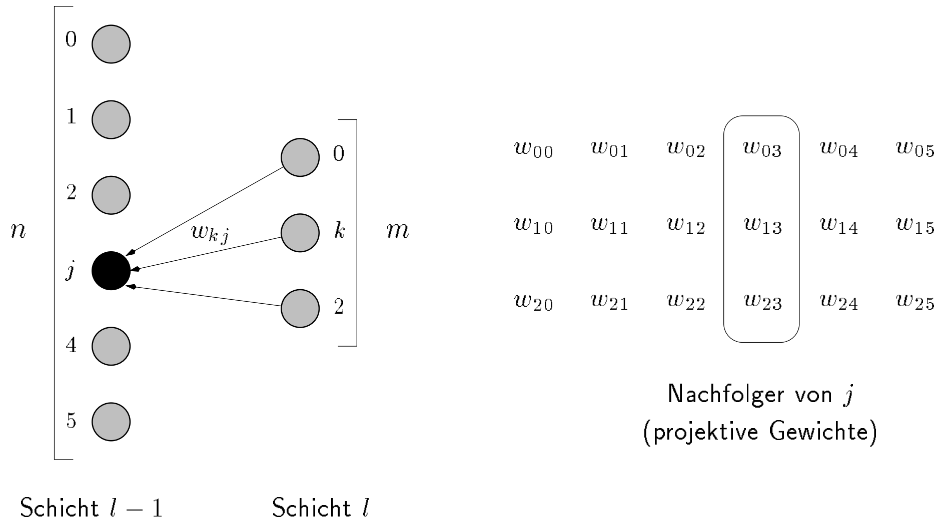


Abbildung 3.4: Backward-Phase

Die Bestimmung des Fehlers der verborgenen Neuronen basiert auf der Rückverfolgung des Fehlers der Ausgabeschicht, da auch verborgene Neuronen ihren Anteil am Zustandekommen des Ergebnisses haben. Neuron j in Abbildung 3.4 benötigt dazu die Fehlerwerte seiner *Nachfolger* und die Gewichte der *projektiven* Kanten. Der Fehler δ_j für *verborgene Neuronen* berechnet sich als:

$$\delta_j \quad = \quad \left(\sum_{k=0}^{m-1} \delta_k w_{kj} \right) \ \underbrace{o_j \ (1 - o_j)}_{f'(a_j)}$$

Die Identität $f'(a_j) = o_j \ (1 - o_j)$ gilt für die Sigmoide als Transferfunktion und erleichtert die Berechnungen, da die Ausgabewerte noch aus der Forward-Phase stammen. Die Bestimmung der Fehlerwerte endet bei der ersten verborgenen Schicht, da Eingabeneuronen keine Berechnungen ausführen.

Analog zur Forward-Phase ist ein Teil der Berechnungen, die zur Berechnung des Fehlers verborgener Neuronen dienen, als Multiplikation der transponierten Gewichtematrix mit dem Fehlervektor anzusehen. Diese Vektor-Matrix-Operation stellt Abbildung 3.5 dar, wobei o den Fehlervektor der Ausgabeschicht, h den der verborgenen Schicht bezeichnet.

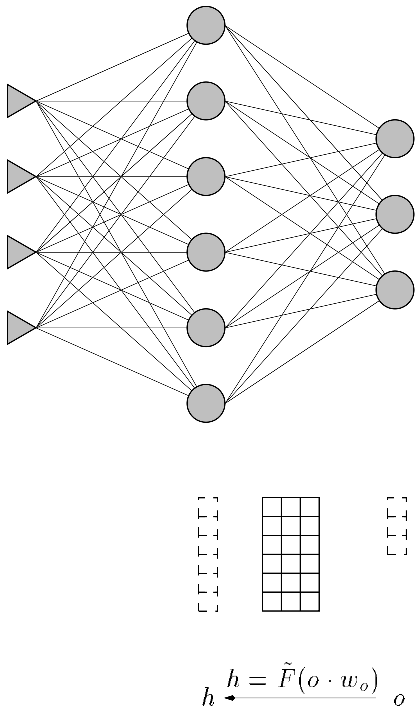


Abbildung 3.5: Vektor-Matrix-Operationen in der Backward-Phase

Die Gewichtsänderungen nehmen die Neuronen bezüglich der *rezeptiven Kanten* auf ihrer Eingangsseite vor. Die Änderungen berechnen sich gleichermaßen für verborgene Neuronen und Ausgabeneuronen:

$$\begin{aligned} \Delta w_{kj} &= \eta \, \delta_k \, o_j \\ \Delta \theta_k &= \eta \, \delta_k \end{aligned}$$

Der Faktor η wird als *Lernrate* bezeichnet und gibt die Schrittweite des Gradientenabstiegs an. Üblicherweise wird für η ein Wert zwischen null und eins gewählt, der während des Trainings für alle Neuronen gleich bleibt. Ein optimaler Wert kann a priori nicht angegeben werden, sondern sich nur durch Experimente herausstellen.

3.3.4 Varianten des Backpropagation-Algorithmus

Ein Problem aller Gradientenverfahren stellt die Existenz lokaler Minima der zu minimierenden Funktion dar. Häufig kommt die Suche in einem Suboptimum zum Stehen. Um dem vorzubeugen bietet der Backpropagation-Algorithmus einige Parameter, die es erlauben, den Gradientenabstieg zu manipulieren.

Eine der wichtigsten Stellgrößen ist die Lernrate η. In der Theorie wird mittels einer hinreichend kleinen Lernrate in jedem Fall das globale Minimum erreicht [6]. Eine kleine Lernrate verlängert die ohnehin langen Trainingszeiten jedoch erheblich.

Einige Verfeinerungen des Backpropagation-Algorithmus versuchen durch Einbeziehung weiterer Informationen als nur des momentanen Gradienten, Aufschluß über die Beschaffenheit der Fehlerfunktion zu gewinnen [1]. Dadurch sollen sich gegenseitig aufhebende und damit unnötige Gewichtsänderungen vermieden werden. Im Vergleich zum ursprünglichen Backpropagation-Algorithmus wird insbesondere bei regelmäßigen Fehlerverläufen ein Optimum in deutlich weniger Schritten erreicht.

Das einfachste Verfahren dieser Art stellt die Verwendung eines *Momentums* bzw. *Impulses* dar. Zu der aus den Fehlern der Neuronen resultierenden Gewichtsänderung wird ein Bruchteil der letzten aufaddiert:

$$\Delta^t w_{kj} = \eta\, \delta_k\, o_j + \underbrace{\alpha\, \Delta^{t-1} w_{kj}}_{\text{Impuls/Momentum}}$$

Ein optimaler Wert für α kann ebenfalls nicht a priori angegeben werden. Er sollte kleiner als eins gewählt werden, da ansonsten der Betrag des Korrekturterms stark anwächst (geometrische Reihe).

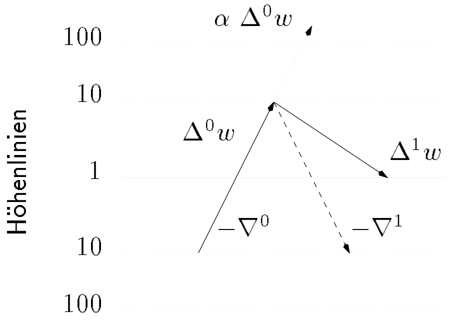


Abbildung 3.6: Gewichtskorrektur mit einer Impulsrate α von 0.5

Abbildung 3.6 illustriert die Korrektur für ein einfaches Beispiel. Die durchgezogenen Pfeile stellen die realisierten Änderungen der Gewichte dar. Die Änderung geschieht in einer dem Gradienten ∇ entgegengesetzten Richtung. Der gestrichelte Vektor entspricht der Änderung ohne

Impuls, zusammen mit dem gepunktet dargestellten Momentum ergeben beide die realisierte Korrektur.

Komplexere Verfahren verwenden beispielsweise Methoden zweiter Ordnung, bzw. adaptive Variation der Parameter η und α. Eine detaillierte Beschreibung der Verfahren findet sich in [1] und [6].

Eine der ältesten Modifikationen des Backpropagation-Algorithmus besteht darin, die während der Backward-Phase berechneten Gewichtsänderungen unmittelbar wirksam werden zu lassen, so daß die neuen Gewichte für das nächste Trainingspaar bereits relevant sind *(Online-Training)*. Demgegenüber bezeichnet das *Batch-Training* das Akkumulieren der Änderungen über alle Paare der Trainingsmenge. Erst nach einer vollständigen Epoche werden die Gewichte aktualisiert.

Beide Varianten differieren stark sowohl bezüglich der Anzahl der benötigten Epochen, als auch des Trainingsresultats, also der berechneten Gewichte. Außerdem sind die Modifikationen unterschiedlich effektiv. Während die Abfolge der Trainingspaare für das Batch-Training unerheblich ist, ist es für das Online-Training wichtig, daß für jede Epoche die Paare in zufälliger Reihenfolge trainiert werden.

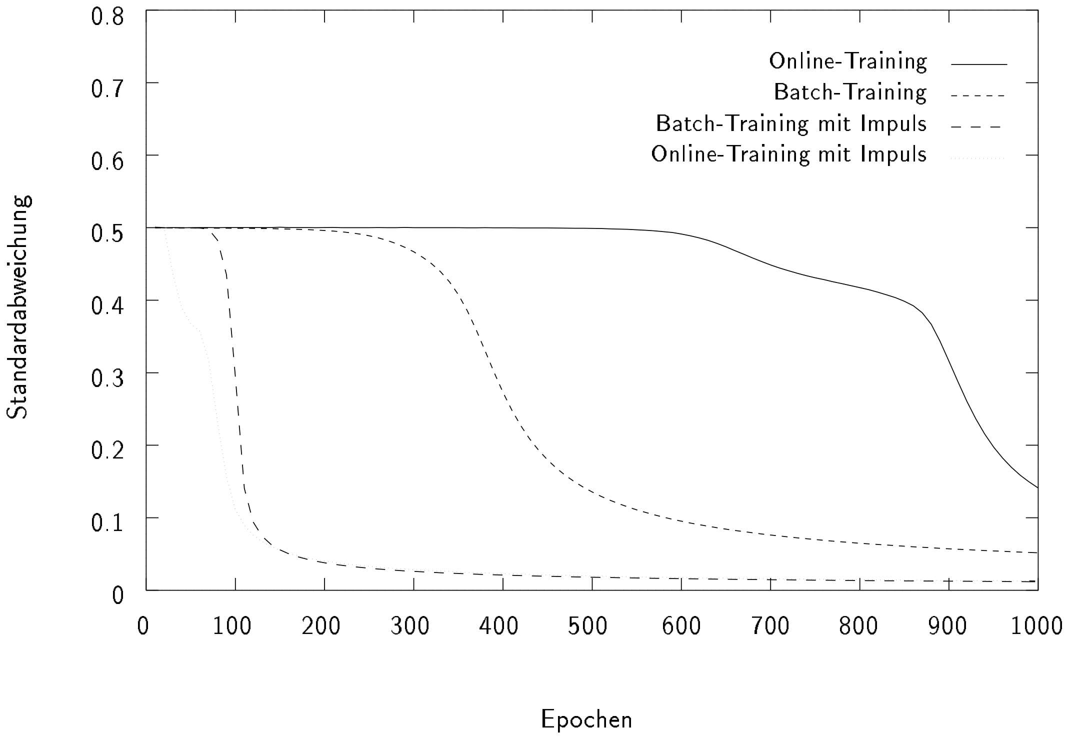


Abbildung 3.7: Fehlerverlauf beim Training des XOR-Problems

Abbildung 3.7 zeigt den Fehlerverlauf während des Trainings des XOR-Problems (siehe Abschnitt 2.3.2) für vier Experimente. Trainiert wurde ein MP-Netz mit zwei Eingabeneuronen, einem Ausgabeneuron und zwei Neuronen in der verborgenen Schicht. Zur Anwendung kamen sowohl das Online- als auch das Batch-Training zum einen mit einer Lernrate η von 1.0, zum andern mit einer Lernrate von η von 2.0 zusammen mit einer Impulsrate α vom 0.9. Es zeigen sich deutliche Unterschiede in der Konvergenzgeschwindigkeit der verschiedenen Ansätze, weshalb eine geeignete Wahl der Lernparameter die Anzahl der erforderlichen Epochen erheblich verkürzt.

Ein wesentliches Problem bleibt allerdings die Tatsache, daß ein kleiner Fehler auf der Trainingsmenge nicht unbedingt mit einem guten Generalisierungsverhalten, also einem kleinen Fehler auf der Testmenge einhergeht. Vielmehr verhält es sich mit beiden Zielen in der Regel gegenläufig, d.h. der Fehler bezüglich der Testmenge steigt während des Trainings an, statt ebenfalls zu sinken. Dieser Effekt wird als *Overfitting* bezeichnet. Das im folgenden Abschnitt vorgestellte Verfahren versucht, dieses Problem auf einer konstruktiven Art zu beheben.

3.4 Der „MouseProp“ Algorithmus

3.4.1 Motivation

Trotz ihrer leichten Anwendbarkeit, beherbergen MP-Netze ein hartnäckiges Problem. Zu einer gegebenen Datenmenge eine passende Anzahl verborgener Neuronen anzugeben ist Sache des Anwenders. Die Erfahrung lehrt, daß verhältnismäßig kleine MP-Netze in der Lage sind, wesentliche Merkmale zu extrahieren und somit gut zu generalisieren. Jedoch kann die Trainingsmenge nicht beliebig genau reproduziert werden. Hierfür ist eine größere Anzahl verborgener Neuronen notwendig. Hieraus resultieren allerdings auch MP-Netze mit einer verstärkten Tendenz zum Auswendiglernen der Trainigspaare einhergehend mit unbefriedigender Verallgemeinerungsfähigkeit. Konstruktionsalgorithmen versuchen nun, die positiven Eigenschaften sowohl der kleinen als auch der großen MP-Netze zu vereinigen [27]. Mit einem verhältnismäßig kleinem MP-Netz beginnend werden in der Trainingsphase dem Netz verborgene Neuronen hinzugefügt.

Das hier vorgestellte Verfahren ersetzt während des Trainings in regelmäßigen Abständen ein Neuron durch zwei neue. Dieses Aufspalten wird auf Neuronen angewendet, die durch stark oszillierende Gewichtsänderungen auffallen. Dahinter verbirgt sich die Hoffnung, daß zwei Neuronen die Aufgabe des einen besser erfüllen. Neuronen werden also nur dort zugefügt, wo Bedarf besteht. Ein derartig gewachsenes MP-Netz weist aufgrund seiner Entwicklung ein deutlich besseres Generalisierungsvermögen auf als ein Netz mit einer von Anfang an großen Zahl verborgener Neuronen [5].

Die Ausführungen stützen sich im wesentlichen auf die Arbeit von GLÖCKNER [5]. Ein empirischer Vergleich dieses Algorithmus mit MP-Netzen fester Topologie findet in Abschnitt 3.5.2 statt.

Der Erfolg des MouseProp-Algorithmus hängt entscheidend von den Kriterien ab, die ein aufzuspaltendes Neuron auszeichnen. Ein Maß für die „Oszillation eines Neurons" wird im folgenden Abschnitt vorgestellt.

3.4.2 Auswählen eines aufzuspaltenden Neurons

Während einer Epoche werden die durch den Backpropagation-Algorithmus berechneten Gewichtsänderungen für alle T Trainingspaare protokolliert, um für jede Verbindung von Neuron j zu Neuron k folgende numerischen Größen zu kennen:

$$\begin{aligned} \mathsf{as}(k,j) &:= \left|\sum^{T} \Delta w_{kj}\right| \\ \mathsf{sa}(k,j) &:= \sum^{T} |\Delta w_{kj}| \end{aligned}$$

Hierbei bezeichnet $\mathsf{as}(k,j)$ die absolute Summe und $\mathsf{sa}(k,j)$ die Summe der absoluten Änderungen von w_{kj}. Ein Wert nahe null für $\mathsf{as}(k,j)$ deutet auf sich gegenseitig auslöschende Änderungen hin und entspricht einer hohen Oszillation. Um verschiedene Verbindungen miteinander vergleichen zu können, wird ein normiertes Maß für die Oszillation der Gewichtsänderungen einer Verbindung eingeführt:

$$\mathsf{osc}(k,j) := 1 - \frac{\mathsf{as}(k,j)}{\mathsf{sa}(k,j)}$$

Hieraus wird die relative Oszillation eines Neurons j bezüglich seiner *projektiven* Gewichte abgeleitet, indem über alle *Nachfolger* k gemittelt wird:

$$\mathsf{osc}_{\mathsf{p}}(j) := \frac{\sum_k \mathsf{sa}(k,j)\,\mathsf{osc}(k,j)}{\sum_k \mathsf{sa}(k,j)}$$

Die relative Oszillation allein ist nicht geeignet, einen Kandidaten aus der Menge aller verborgenen Neuronen zu bestimmen, da für alle Neuronen, die sich in einem Fehlerminimum befinden, eine starke Oszillation der Gewichte beobachtet wird. Deshalb werden nur solche Neuronen in Betracht gezogen, die einen überdurchschnittlichen relativen Fehler aufweisen. Erst aus dieser Menge wird als aufzuspaltendes Neuron das mit der höchsten relativen Oszillation ausgewählt.

3.4.3 Aufspalten eines Neurons

Die Änderung der Topologie wirkt sich nur lokal aus und beeinträchtigt nicht das Abbildungsverhalten eines MP-Netzes.

Abbildung 3.8 zeigt die Aufteilung der Gewichte beim Aufspalten eines Neurons j in zwei Neuronen j' und j". Die Gewichte der rezeptiven Kanten w_j werden dupliziert. Der Vektor der projektiven Gewichte v_j wird so auf die beiden Neuronen aufgeteilt, daß die Summe den ursprünglichen Vektor v_j ergibt. Der jeweilige Anteil wird für alle Komponenten per Zufallsgenerator ermittelt.

Das auf diese Weise entstandene Netz zeigt zum vorhergehenden keine Unterschied bezüglich des Abbildungsverhaltens (monoton). Allein die Fehler der Neuronen j' und j" entsprechen nicht mehr dem des Originals j (fehlerlokal).

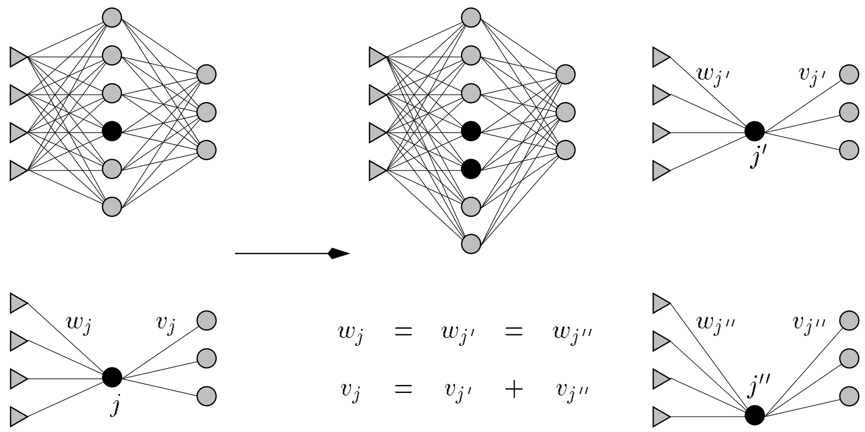


Abbildung 3.8: Aufspalten eines Neurons

3.4.4 Bewertung des Verfahrens

Für den Benutzer des MouseProp-Algorithmus kommen zwei weitere Trainingsparameter hinzu, deren optimaler Wert nur experimentell bestimmt werden kann. Die Anzahl der Epochen zwischen zwei Aufspaltungen sollte nicht zu klein bemessen sein, damit die Gewichte Gelegenheit erhalten, sich neu einstellen. Eine zu hohe Anzahl verlangsamt das Training unnötig. Ebenso bleibt die initiale Größe der verborgenen Schicht des MP-Netzes Sache des Anwenders.

Für das Problem, die optimale Netzkonfiguration zu finden, ist dieses Verfahren keine Lösung. Dafür bietet es eine gewisse Stabilität gegenüber Overfitting bei einer großen Anzahl verborgener Neuronen. Außerdem wird das Training im Vergleich zu einem von Anfang an großem MP-Netz beschleunigt. Zum einen dadurch, daß ein Fehlerminimum in weniger Epochen erreicht wird, zum andern benötigt eine Epoche selbst weniger Zeit.

3.4.5 Evaluation mittels des SBIC

Zur Modellierung einer Problemstellung bieten sich in der Regel mehrere Varianten an. Sie unterscheiden sich zum Beispiel in der Anzahl der einstellbaren Parameter und des Abbildungsfehlers. Welches Modell der wirklichen Situation am ehesten gerecht wird, ist schwer zu entscheiden.

Eine Hilfestellung bietet das Verfahren von SCHWARZ [21]. Es ermittelt für jede Modellvariante einen Koeffizienten, das *sbic*. Die Formulierung des Schwarz'schen Bayesianischen Informationskriteriums für autoregressive Modelle (siehe Abschnitt 3.5.1) setzt den Modellfehler in Relation zu der Anzahl der Modellparameter und der Anzahl der Datenpunkte:

$$sbic = \frac{1}{2}\left[1 + \ln(2\,\pi) + 2\,\ln(rms) + \frac{p\,\ln(n)}{n}\right]$$

Das *sbic* eines Modells steht für dessen Wirklichkeitsnähe, wobei ein kleiner Wert eine hohe Übereinstimmung mit der Realität ausdrückt. Diese Eigenschaft wird in [15] ausgenutzt, um MP-Netze konstanter Topologie mit einer unterschiedlichen Anzahl an verborgenen Neuronen zu vergleichen. Die Anzahl p der Modellparameter berechnet sich als Summe aller Gewichte und Schwellwerte. Der Modellfehler *rms* (*root mean square error*) entspricht der Standardabweichung aller n Trainingspaare. In der Regel besitzt das *sbic* ein einziges globales Minimum, was einer optimalen Netztopologie entspricht.

Das *sbic* kann somit auch ein Anhaltspunkt dafür sein, ob sich das Hinzufügen eines Neurons nennenswert gelohnt hat. Eine Berechnung des *sbic* unmittelbar vor jedem Aufspalten erlaubt Rückschlüsse auf den Erfolg der letzten Aufspaltung. Einen beispielhaften Verlauf des *sbic* während des Trainings mit dem MouseProp-Algorithmus zeigt Abbildung 3.14 in Abschnitt 3.5.2.

3.5 Zeitreihenprognose mit MP-Netzen

3.5.1 Autoregressive Modelle

Eine *Zeitreihe* beschreibt die Entwicklung einer Variablen x, die an diskreten Zeitpunkten $t = 1 \ldots T$ beobachtet wurde. Ein autoregressives Modell versucht, einen Zusammenhang für x_t aufgrund der Historie abzuleiten:

$$x_t = f(x_{t-1}, x_{t-2}, \ldots, x_{t-d}) \qquad 1 \leq t \leq T$$

Ziel ist es, *eine* geeignete Funktion f zu finden, die für *alle* bekannten Datenpunkte ein zufriedenstellendes Ergebnis erzielt. Die eigentliche Aufgabe besteht nun darin, den Abbildungsfehler zu minimieren, indem Parameter der Funktion f variiert werden. Am einfachsten gestaltet sich dies für lineare Funktionen. Für sie liefert die Methode der kleinsten Quadrate optimale Werte, jedoch ist das Ergebnis häufig nicht zufriedenstellend, zum Beispiel weil der tatsächliche Zusammenhang nichtlinearer Natur ist.

Eine spezielle Kategorie nichtlinearer Funktionen stellen die MP-Netze dar. Vor allem zeichnen sie sich als *universelle Approximatoren* aus [8]. D.h. zu jeder Trainingsaufgabe existiert ein MP-Netz, welches nach dem Training mit dem Backpropagation-Algorithmus die Trainingspaare beliebig genau annähert [25].

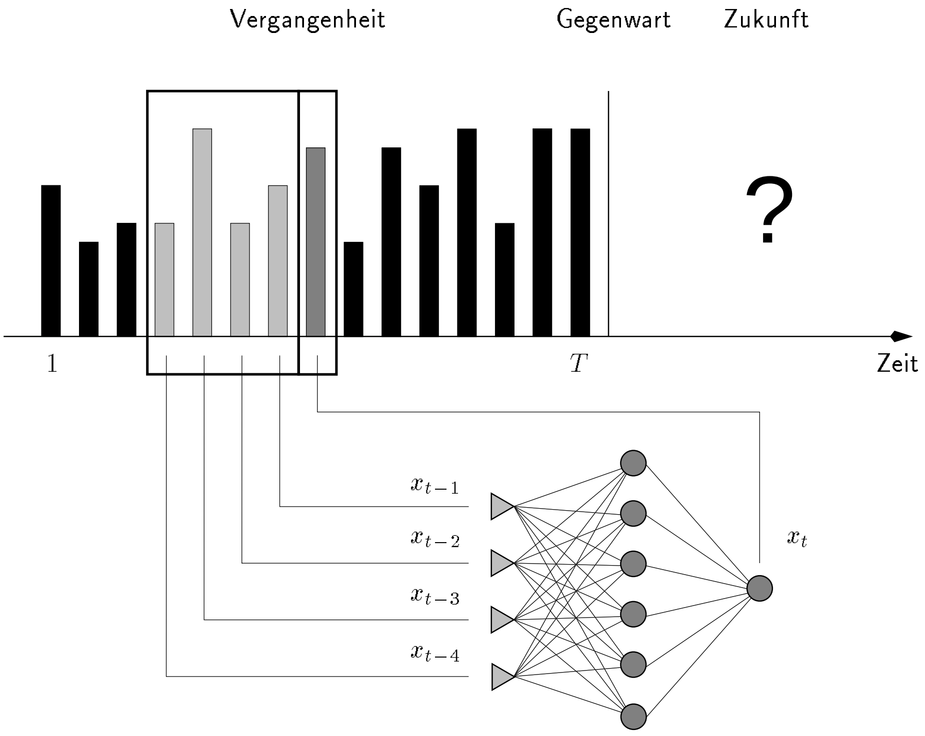


Abbildung 3.9: Zeitreihenprognose mit einem MP-Netz

Die Modellierung eines autoregressiven Prozesses mit Hilfe eines MP-Netzes zeigt Abbildung 3.9. Um den Wert für x_t vorherzusagen, werden die der vorherigen $d(=4)$ Zeitpunkte der Eingabeschicht präsentiert. Die Ausgabeschicht besteht aus nur einem Neuron und gibt den Wert zum Zeitpunkt t an. Die Prognose des zukünftigen Wertes x_{T+1} geschieht durch Eingabe der Werte x_T, x_{T-1}, x_{T-2} und x_{T-3}.

Für den Fall, daß es sich um eine Zeitreihe von Vektoren der Dimension n handelt, besteht die Möglichkeit, jeden Vektor auf n Eingabe- bzw. Ausgabeneuronen abzubilden. In der Praxis wird jedoch ein anderer Weg verfolgt. Um die Komplexität der Trainingsaufgabe eines einzelnen

Netzes zu reduzieren, werden n MP-Netze darauf trainiert, jeweils nur eine Komponente des Ergebnisvektors x_t zu prognostizieren.

Die Anzahl der verborgenen Neuronen bleibt Sache des Anwenders. Sie kann beispielsweise mit dem Verfahren aus Abschnitt 3.4.5 optimiert werden.

Zusätzliche Einflußfaktoren neben der eigentlichen Zeitreihe werden miteinbezogen, indem sie auf weitere Eingabeneuronen abgebildet werden. Ein MP-Netz ist in der Lage, die Relevanz der Informationen zu erkennen und sie zu berücksichtigen bzw. sie zu ignorieren. Allerdings ist es hilfreich, die Eingabedaten in einer Vorverarbeitung auf redundante Elemente hin zu untersuchen und mit diesen Anhaltspunkten die Trainingsmenge nominell zu verkleinern.

3.5.2 Verhulst-Prozess

In dem folgenden Abschnitt wird die Prognose einer „synthetischen" Zeitreihe untersucht. Die der Zeitreihe zugrundeliegenden Daten werden aus der VERHULST-Gleichung generiert:

$$x_{t+1} = x_t \cdot k \cdot (1 - x_t)$$

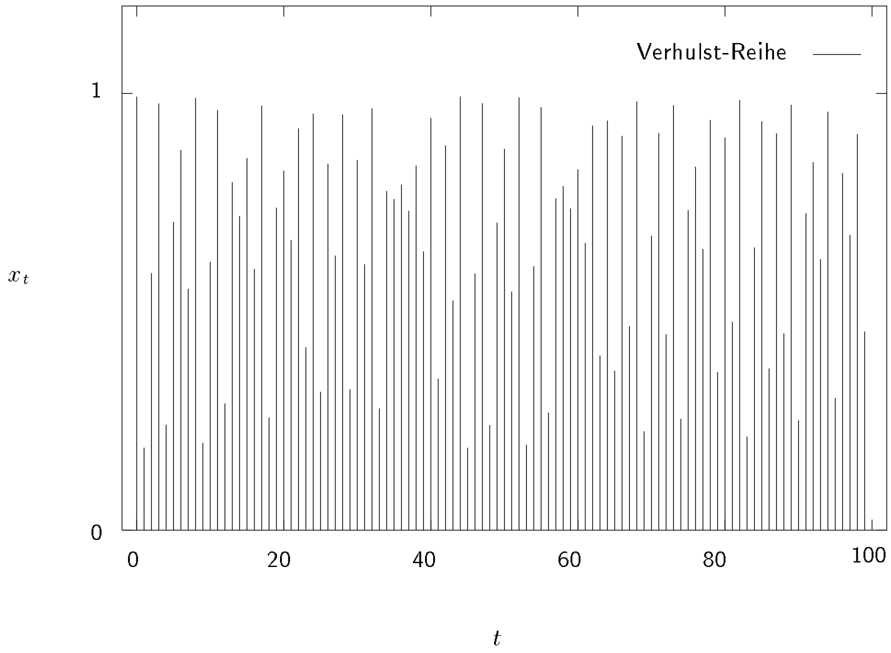


Abbildung 3.10: Eine VERHULST-Reihe

Bekannter als die Gleichung selbst, dürfte das FEIGENBAUM-Diagramm sein, welches für wechselnde Parameter k die zugehörigen Fixpunkte angibt [17]. Werte um 3,8 für k führen zu einem Verhalten, welches als *deterministisches Chaos* bezeichnet wird. Die aus der VERHULST-Gleichung generierte Zeitreihe weist in diesem Bereich eine große Ähnlichkeit mit einem Zufallsprozeß auf. Einem Betrachter der Zeitreihe offenbart sich nicht der doch einfache Zusammenhang (siehe Abbildung 3.10).

Es werden drei MP-Netze mit einer unterschiedlichen Anzahl verborgener Neuronen trainiert. Die Problemstellung ist für alle drei gleich. Bei Eingabe der letzten drei Werte der Zeitreihe soll der nächste ausgegeben werden. Hieraus resultieren drei Eingabe- und ein Ausgabeneuron. Die Zahl der verborgenen Neuronen beträgt einmal sechs, einmal 24. Für das dritte Netz wird der MouseProp-Algorithmus angewendet. Die verborgene Schicht umfaßt zu Beginn des Trainings sechs, danach 24 Neuronen.

Um mit dem verarbeiteten Signaltyp kompatibel zu sein, sind die Daten der Zeitreihe so skaliert, daß nur Werte aus dem Intervall [0,1] angenommen werden. Die Trainingsmenge besteht aus 31 Paaren, die Testmenge aus 7 Paaren.

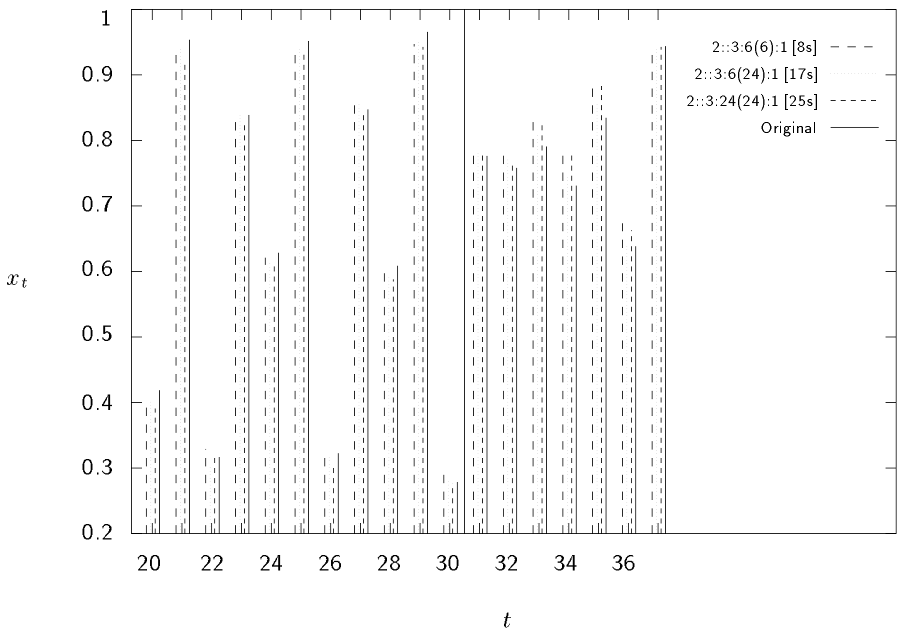


Abbildung 3.11: Prognose einer VERHULST-Reihe

Nach 4000 Epochen wird für jedes Netz eine Abfrage durchgeführt. Abbildung 3.11 zeigt für jedes MP-Netz den Prognosewert. Die jeweils rechte Säule gibt zum Vergleich den Originalwert an. Die Netztopologie wird nach folgendem Muster beschrieben:

```
<Anzahl der aktiven Schichten>:
:<Anzahl der Eingabeneuronen>
:<initiale Anzahl verborgener Neuronen>
      (<maximale Anzahl verborgener Neuronen>)
    .
    .
    .
:<Anzahl der Ausgabeneuronen>
```

In den eckigen Klammern neben der Netztopologie ist die Dauer des Trainings angegeben.

Offensichtlich sind alle MP-Netze in der Lage, die Trainingsaufgabe zu reproduzieren. Doch zeigen sich auf der Testmenge ($t > 30$) deutliche Unterschiede. Hier schneidet das Netz, welches mit dem MouseProp-Algorithmus trainiert wurde, am besten ab.

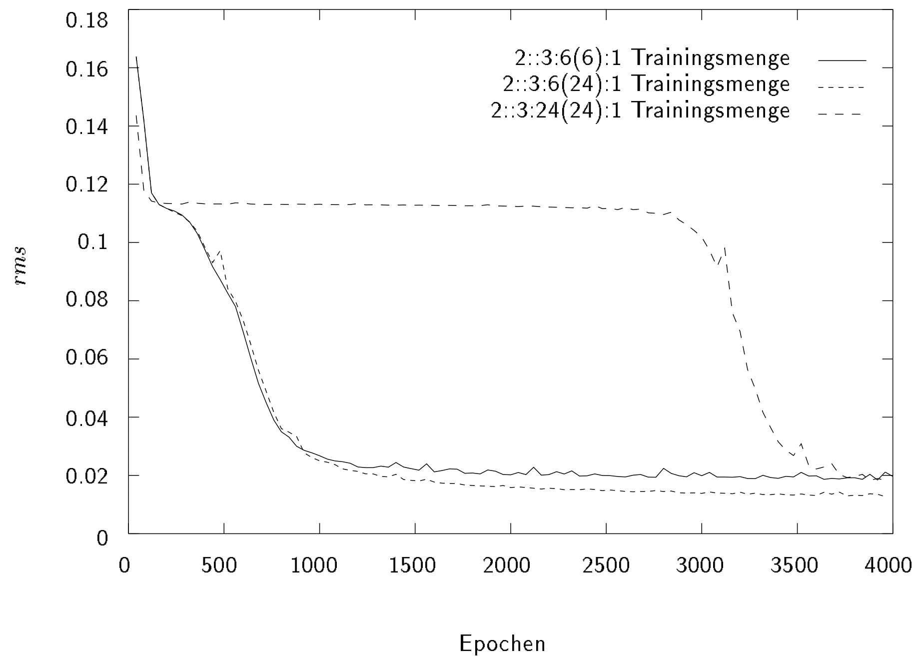


Abbildung 3.12: Fehlerverlauf auf der Trainingsmenge

Als sehr aufschlußreich zeigt sich die Verfolgung des Fehlers auf der Trainingsmenge (Abbildung 3.12) und der Testmenge (Abbildung 3.13). Auffällig ist hier, daß das kleine (`2::3:6(6):1`) und das mittlere (`2::3:6(24):1`) Netz bereits nach rund 1000 Epochen Fehlerwerte in der Nähe eines Minimums erreicht haben. Für das große Netz (`2::3:24(24):1`) ist dies erst nach 3500 Epochen der Fall. Der verhältnismäßig späte Abbruch des Trainings für das kleine und mittlere

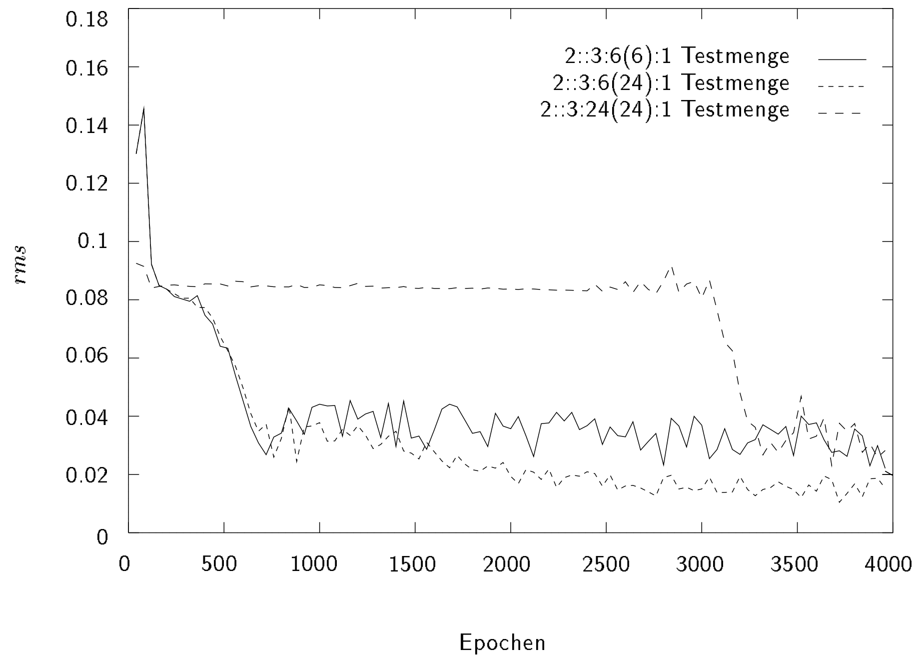


Abbildung 3.13: Fehlerverlauf auf der Testmenge

Netz führt nicht zu einem Overfitting. Das kleine MP-Netz profitiert allerdings auch nicht von der langen Trainingsdauer, da es eine anhaltend starke Oszillation des Fehlers der Testmenge bei gleichbleibend hohem Niveau aufweist. Insgesamt gesehen zeigt sich das mittlere MP-Netz als die beste Wahl. Es schlägt die anderen beiden sowohl bezüglich der Anzahl der Epochen als auch des erreichten Fehlerminimums.

Soll für das mittlere MP-Netz eine optimale Zahl der verborgenen Neuronen gefunden werden, so bietet das *sbic* eine Hilfestellung. Abbildung 3.14 stellt die Entwicklung des *sbic* zusammen mit dem Fehlerwert *rms* für das mit dem MouseProp-Algorithmus trainierte MP-Netz dar. Vor jedem Aufspalten wird der *sbic* berechnet und ausgegeben. Der minimale Wert des Kriteriums liegt hier bei 1500 Epochen, welches einer Anzahl von 13 verborgenen Neuronen entspricht. Dieser Wert ist allerdings abhängig von der Anzahl der Epochen zwischen zwei Aufspaltungen.

Das Abfallen des *sbic* in der ersten Hälfte des Trainings ist im wesentlichen auf die starke Reduzierung des Fehlers zurückzuführen. Der Anstieg in der zweiten Hälfte des Trainings deutet darauf hin, daß die Aufnahme zusätzlicher Neuronen den Fehler nicht im gleichen Maße herabsenken kann.

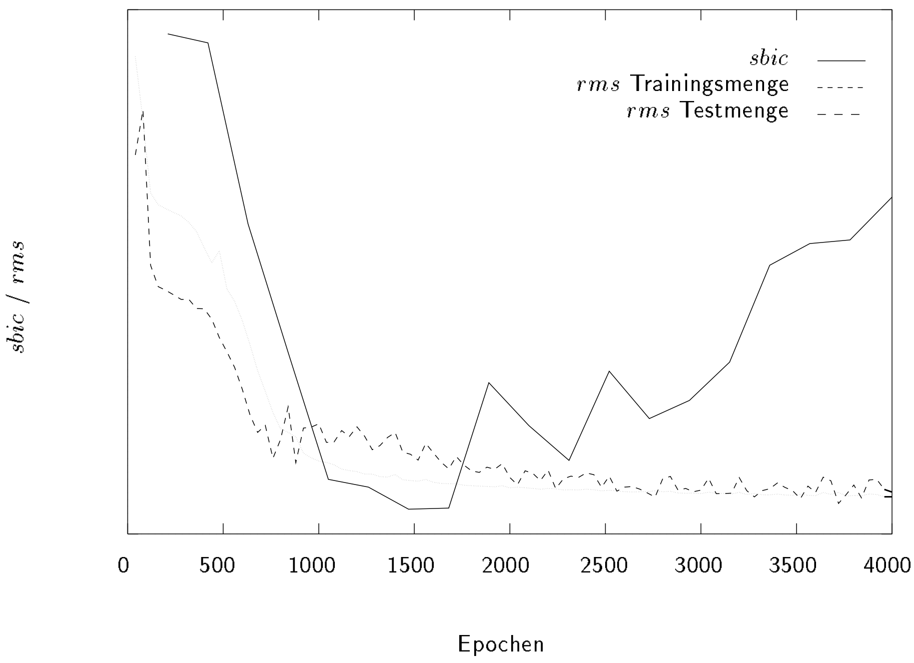


Abbildung 3.14: Verlauf des *sbic* während des Trainings

3.5.3 Natürliche Zeitreihen

In Abschnitt 3.5.2 wird gezeigt, daß MP-Netzen der funktionale Zusammenhang einer chaotischen Zeitreihe antrainiert werden kann. Die Prognose von Zeitreihen, wie sie in der Wirklichkeit auftreten, findet nach dem gleichen Prinzip statt. Unterstützend kommt hinzu, daß MP-Netze in der Lage sind, aus verrauschten Trainingsdaten, die deterministische Komponente zu extrahieren [15].

Die zu prognostizierende Variable ist in vielen Fällen Bestandteil eines größeren Systems, dessen Grenzen schwer abzustecken sind. Häufig werden MP-Netze im wirtschaftlichen Bereich eingesetzt, da dort auf eine lange Erfahrung auf dem Gebiet der Modellierung zurückgegriffen werden kann und umfassende Datenmengen vorliegen [24].

Aus der Berücksichtigung vieler Erklärungsgrößen neben der eigentlichen Zeitreihe resultieren verhältnismäßig große Netze und umfangreiche Trainingsmengen. Beides führt zu einer langen Trainingsdauer, die sich im Bereich von einigen Stunden bewegt. Eine Möglichkeit, diese Zeit zu verkürzen, bietet die Parallelisierung des Backpropagation-Algorithmus, welche im folgenden Kapitel ausführlich behandelt wird.

Kapitel 4

Parallele Backpropagation

4.1 Überblick

Die langen Laufzeiten des Backpropagation-Algorithmus signifikant zu verkürzen ist das Ziel einer effizienten Parallelisierung. Nach der Vorstellung grundlegender Konzepte paralleler Systeme werden mögliche Strategien zur Aufteilung des Trainings und dessen paralleler Abarbeitung präsentiert. Die Implementation für das Parallelrechnerbetriebssystem PARIX und das Message-Passing-Modell PVM werden vorgestellt und zum Abschluß deren Effizienz für verschiedene Architekturen verglichen.

4.2 Parallele Systeme

Das Konzept jeder Art von Parallelrechner ist die Verwendung vieler (kleiner) Prozessoreinheiten, die zusammengenommen mehr leisten als ein zentraler Hochleistungsprozessor. Ein großer Vorteil paralleler Systeme ist ihr nahezu lineares Preis/Leistungsverhältnis, was für Ein-Prozessor-Maschinen vergleichbarer Rechenleistung nicht besteht. Die Nutzbarmachung des Verarbeitungspotentials eines parallelen Rechners ist allerdings von der Formulierung effizienter paralleler Algorithmen abhängig.

Wesentliche Unterscheidungsmerkmale paralleler Rechner sind neben der Prozessorhardware die Kontrolle der *Programmausführung* und die Art der *Speicherorganisation*. Ein SIMD-Rechner (*single instruction, multiple data*) zeichnet sich durch die synchrone Programmausführung aus. Auf allen Prozessoren wird zur gleichen Zeit der gleiche Programmcode auf unterschiedliche Daten angewendet. Demgegenüber charakterisiert das MIMD-Modell (*multiple instruction, multiple data*) einen Rechner, dessen Prozessoren unterschiedliche Programme ausführen und auf lokalen Daten operieren. Erhält jeder Prozessor eines MIMD-Rechners das gleiche Programm, so entspricht dies dem SPMD-Modell (*single program, multiple data*).

In der Regel erfordet die parallele Abarbeitung eines Problems den Austausch von Daten (zum Beispiel Teilergebnisse) unter den Prozessoren. Wird der zur Verfügung stehende Speicherplatz global verwaltet (*shared memory*), so hat jeder Prozessor Zugriff auf jeden Bereich des Speichers. Der Informationsaustausch findet über den gemeinsamen Speicher statt. Interprozessorkommunikation ist dann erforderlich, wenn jeder Prozessor seinen eigenen lokalen Speicherbereich (*distributed memory*) besitzt. Informationen werden als Nachrichten unter den Prozessoren ausgetauscht (*message passing*). Sowohl gemeinsamer Speicher als auch message passing erfordert ein Verbindungsnetzwerk unter den Prozessoren.

Neben *massiv parallelen Systemen* mit bis zu tausend Prozessoren werden inzwischen auch vermehrt vernetzte UNIX-Workstations zur Parallelverarbeitung eingesetzt. Obwohl spezielle Parallelrechner bessere Voraussetzungen beispielsweise bei der Kommunikation bieten, besitzt ein Workstationverbund den Vorteil, daß die benötigte Hardware bereits vorhanden und durch den alltäglichen Gebrauch kaum ausgelastet ist.

4.3 Parallelisierungsstrategien

4.3.1 Partitionierung des experimentellen Raumes

Das Training eines MP-Netzes mit dem Backpropagation-Algorithmus bietet mehrere Ansätze der Parallelisierung. Der vermutliche einfachste Weg verfolgt die Idee, völlig verschiedene, voneinander unabhängige Experimente auf die Prozessoren zu verteilen.

Die Parameter eines Experiments sind beispielsweise die Netztopologie, die Trainingsmuster und Lernparameter. Dieser Ansatz erfordert kaum Kommunikation, auf jedem Prozessor des Parallelrechners läuft der sequentielle Algorithmus. Diese Idee ist jedoch untauglich, falls die Experimente in einer bestimmten Reihenfolge ablaufen sollen.

Die Evolution optimaler MP-Netze hinsichtlich ihrer Topologie und ihrer Gewichte mit Hilfe genetischer Algorithmen macht sich diesen Ansatz zunutze [2].

4.3.2 Partitionierung der Trainingsmenge

Eine grobkörnige Parallelisierung bietet das Unterteilen der Trainingsmenge in gleichmäßige Stücke. Auf allen Prozessoren befindet sich eine identische Kopie des MP-Netzes. Nachdem jeder Prozessor den Backpropagation-Algorithmus auf seine lokale Trainingsmenge angewendet hat, werden die gesammelten Gewichtskorrekturen untereinander ausgetauscht. Auf diese Art läßt sich das Batch-Training (siehe Abschnitt 3.3.4) vor allem für große Trainingsmengen sehr effektiv parallelisieren [3],[13], da nur einmal pro Epoche kommuniziert wird.

Die lokalen Gewichtsänderungen addieren sich zu der globalen Gewichtsänderung auf. Diese erhält jeder Prozessor zurück und kann damit seine Kopie des MP-Netzes aktualisieren.

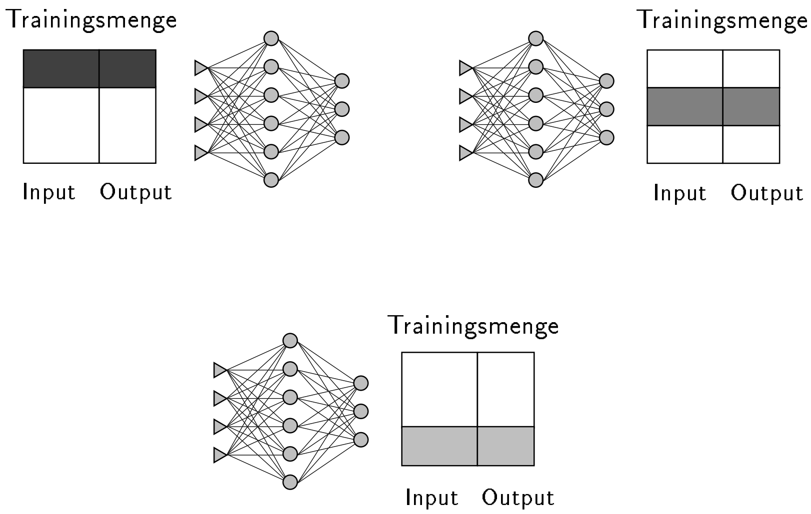


Abbildung 4.1: Paralleles Batch-Training : Trainingsphase

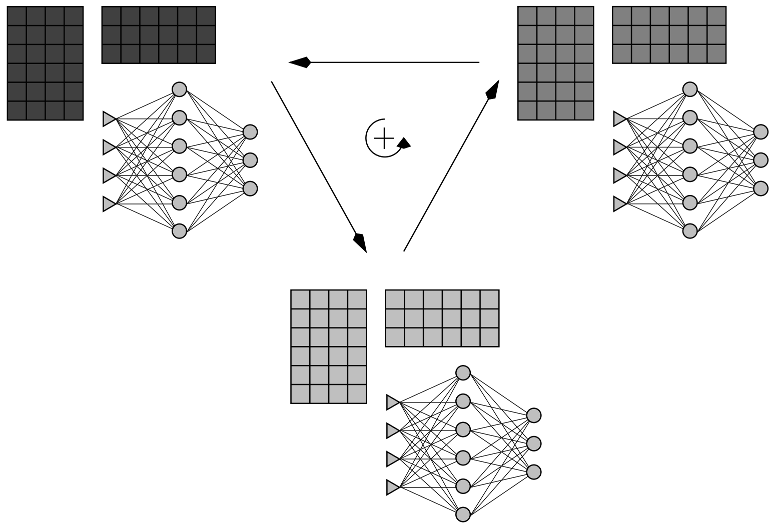


Abbildung 4.2: Paralleles Batch-Training : Angleichen der Gewichte

Die Trainingspaare werden zu Programmbeginn den Prozessoren zugeteilt. Während des Trainings erfolgt keine Neuverteilung, da die jeweilige Prozessorlast konstant bleibt. Auch die Integration des MouseProp-Algorithmus (siehe Abschnitt 3.4) hat darauf keine Auswirkungen. Implementiert ist dieser Ansatz unter PARIX und PVM.

4.3.3 Partitionierung des MP-Netzes

Für das Online-Training (siehe Abschnitt 3.3.4) ist eine noch feinkörnigere Parallelisierung nötig, da bereits nach einem Musterpaar die Gewichte aktualisiert werden. Im Gegensatz zum vorherigen Ansatz sind nicht die zu berechnenden Aufgaben verteilt (*domain decomposition*), sondern jedes einzelne Problem wird parallel angegangen (*function decomposition*).

Die funktionalen Einheiten eines MP-Netzes sind die Neuronen. Sie gilt es zu verteilen. Von einer Partitionierung in *„Querrichtung“* ist abzuraten, da sie die Anzahl der einsetzbaren Prozessoren auf die Anzahl der Schichten begrenzt. Außerdem läßt sie aufgrund der hierarchischen Struktur eines MP-Netzes nur eine Verwendung des Batch-Trainings zu [14].

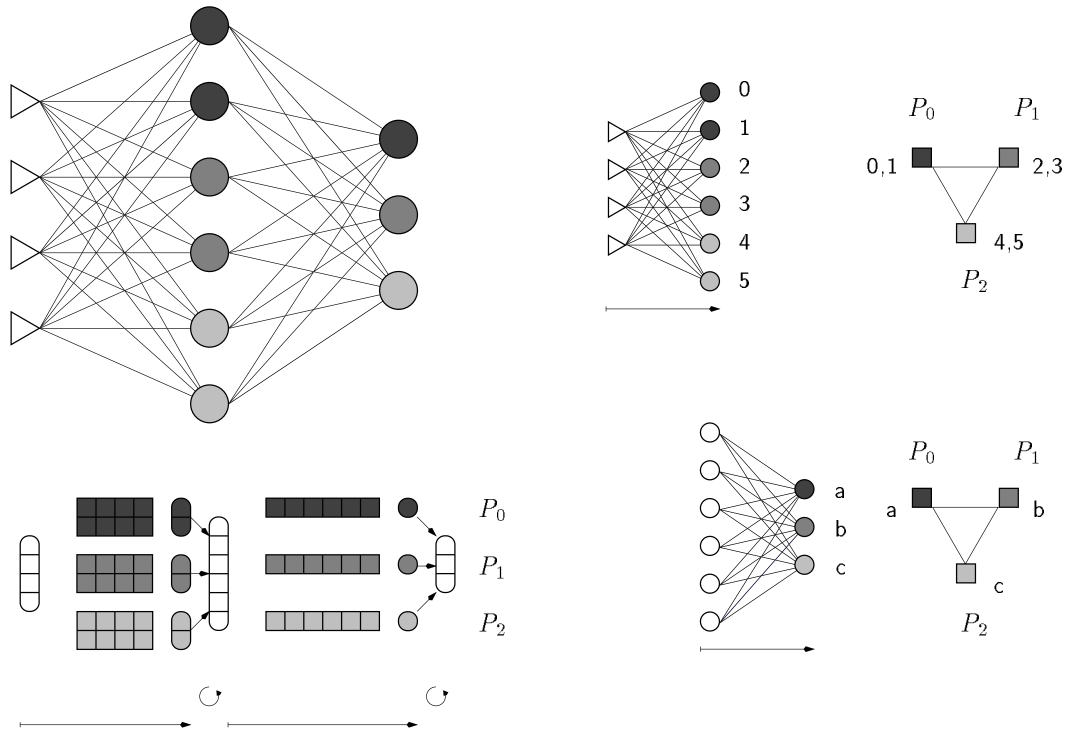


Abbildung 4.3: Parallele Forward-Phase

Die Partitionierung *längs* der Verbindungen erlaubt eine Parallelisierung des Online-Trainings. Für jede Schicht erhält jeder Prozessor die Zuständigkeit für einige Neuronen. Neben Schwellwert und Ausgabewert eines Neurons bekommt ein Prozessor auch die rezeptiven Gewichte zugewiesen.

Durch diese Aufteilung verringert sich der Speicherbedarf auf jedem Prozessor. Damit bietet sich die Möglichkeit, sehr große MP-Netze zu bearbeiten, die auf nur einem Prozessor nicht zu halten wären. Die Praxis zeigt jedoch, daß die Zeitkomplexität ein viel größeres Hindernis darstellt als die Speicherkomplexität. Schon das Training verhältnismäßig kleiner MP-Netze erweist sich als sehr zeitaufwendig.

In der *Forward-Phase* berechnet jeder Prozessor für seine Partition der jeweiligen Schicht die Ausgabewerte der ihm zugeteilten Neuronen (siehe Abbildung 4.3). Die dazu notwendigen rezeptiven Gewichte sind lokal vorhanden. Die Ausgabewerte der vorherigen Schicht sind jedem Prozessor bekannt. Damit erfolgt die Matrix-Vektor-Multiplikation sowie das Anwenden der Transferfunktion parallel. Das Resultat, also der Ausgabevektor der aktuellen Schicht liegt verteilt vor. Durch einen *All-to-All-Broadcast* [28] erhält jeder Prozessor die nicht von ihm berechneten Komponenten. Dieser Ausgabevektor wird für die im Anschluß stattfindenden Berechnungen der folgenden Schicht benötigt. Insgesamt ist also pro Schicht ein Broadcast erforderlich.

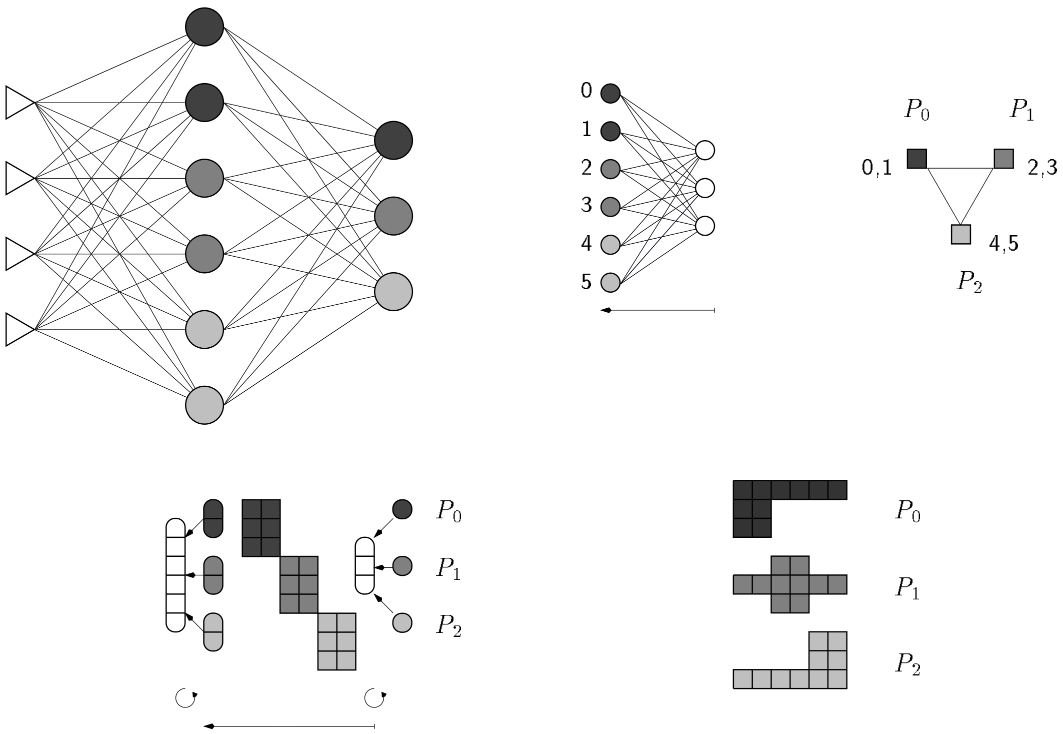


Abbildung 4.4: Parallele Backward-Phase nach YOON et al.

Die für die Fehlerberechnung in der Backward-Phase benötigten Gewichte entsprechen den *Spalten* der Gewichtematrix, lokal vorhanden sind aber nur die *Zeilen*. Um dennoch den Fehlervektor parallel berechnen zu können, werden in dem Ansatz von YOON et al. [28] neben den rezeptiven Gewichten auch die projektiven Gewichte eines Neurons gespeichert (siehe Abbildung 4.4). Die Berechnung der Fehlervektoren in der Backward-Phase verläuft ähnlich wie die der Ausgabevektoren in der Forward-Phase. Entsprechend der Zuständigkeit berechnet jeder Prozessor seinen Anteil. Durch einen All-to-All-Broadcast wird dieser zu einem auf jedem Prozessor identischen Vektor vervollständigt. Auch hier findet die Berechnung der Vektor-Matrix-Operation parallel statt.

Allerdings handelt man sich einen gewissen *Overhead* ein. Die projektiven Gewichte der verborgenen Neuronen müssen von dem jeweils zuständigen Prozessor aktualisiert werden. Dadurch verzögert sich die Backward-Phase. Trotz dieses Mehraufwands wurde die Idee von Yoon et al. erfolgreich auf moderat parallelen Systemen umgesetzt [14].

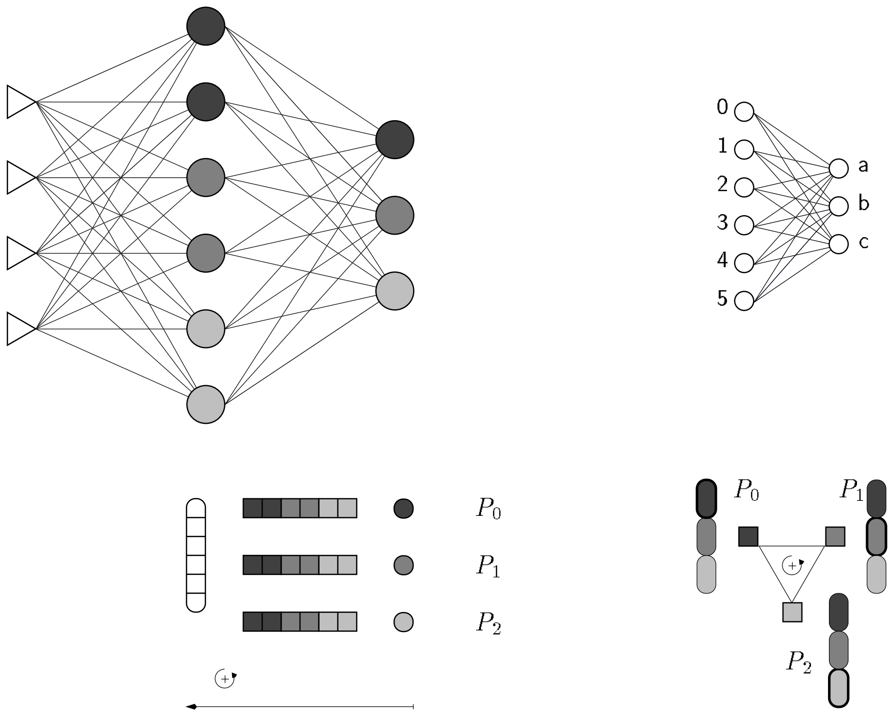


Abbildung 4.5: Parallele Backward-Phase nach Morgan et al.

Einen anderen Weg, die Matrix-Vektor-Multiplikation zu parallelisieren, verfolgt die Idee von Morgan et al. [11]. Statt die Spaltensumme zu berechnen, wird hier über die Zeilen aufaddiert. Jeder Prozessor bestimmt für *alle* Komponenten des Fehlervektors die Teilsumme aus den Produkten, die mit den vorhandenen Gewichten bestimmt werden können (siehe Abbildung 4.5).

Prozessor P_0 müßte für den Fehler von Neuron 0 und Neuron 1 folgende Summen berechnen:

$$\delta_a w_{a0} + \delta_b w_{b0} + \delta_c w_{c0} \qquad \text{und} \qquad \delta_a w_{a1} + \delta_b w_{b1} + \delta_c w_{c1}$$

Stattdessen berechnet P_0 den Vektor:

$$(\delta_a w_{a0},\ \delta_a w_{a1},\ \delta_a w_{a2},\ \delta_a w_{a3},\ \delta_a w_{a4},\ \delta_a w_{a5})^T$$

Entsprechende Vektoren berechnen die anderen Prozessoren. Die komponentenweise Addition der verteilten Vektoren gibt den korrekten Ergebnisvektor der Matrixmultiplikation. In einer *All-to-All-Akkumulation*, wie sie auch beim parallelen Batch-Training verwendet wird, tauschen die Prozessoren die Partialsummen untereinander aus und addieren sie dabei auf.

Die Integration des MouseProp-Algorithmus macht eine dynamische Lastverteilung erforderlich. Das zusätzliche Neuron bekommt der Prozessor, welcher die geringste Anzahl an verborgenen Neuronen der betreffenden Schicht besitzt.

4.3.4 Kommunikationsaufwand

Im folgenden Abschnitt werden die drei Parallelisierungsideen vor dem Hintergrund, daß die Informationen via Message-Passing ausgetauscht werden, verglichen. Die zeitlichen Kosten einer Kommunikation setzen sich aus einem relativ hohen konstanten Anteil und einem von dem zu verschickenden Volumen abhängigen Teil zusammen. Kommunikation bezeichnet in diesem Zusammenhang eine globale Operation, wie etwa All-to-All-Broadcast oder All-to-All-Akkumulation über alle Prozessoren.

Das Volumen einer Kommunikation bei der Parallelisierung des Online-Trainings bleibt für den Ansatz von MORGAN et al. konstant, während es bei YOON et al. mit zunehmender Prozessoranzahl sinkt. Das Kommunikationsvolumen bei der Parallelisierung des Batch-Trainings ist vergleichsweise groß, da statt Vektoren vom Ausmaß einer Schicht Daten im Umfang eines ganzen MP-Netzes verschickt werden. Allerdings bleibt das Volumen unbeeinflußt von der aufzuteilenden Problemgröße, also der Zahl der Trainingspaare. Dieses ergibt einen nahezu beliebig geringen Kommunikationsanteil an der Gesamtlaufzeit.

Die Anzahl der Kommunikationsaufrufe ist bei beiden Varianten des Online-Trainings gleich. Es ist ein Austausch unter allen Prozessoren nach dem Abarbeiten einer Schicht notwendig. Im Vergleich zum Batch-Training bedeutet dies eine vielfach höhere Kommunikationslast, da dort nur nach jeder Epoche kommuniziert wird. Abbildung 4.6 verdeutlicht dies für ein dreischichtiges MP-Netz mit einer verborgenen Schicht und einer relativ kleinen Trainingsmenge von vier Paaren, die auf zwei Prozessoren parallel abgearbeitet wird. Das Größenverhältnis der Blöcke zueinander ist willkürlich gewählt. Dennoch wird deutlich, daß sich das Batch-Training vor allem für große Trainingsmengen sehr effizient parallelisieren läßt.

Für die Parallelisierung des Online-Trainings zeigt sich eine sehr kleine Stückelung in Kommunikations- und Berechnungsphasen, die aufgrund der Randbedingungen des

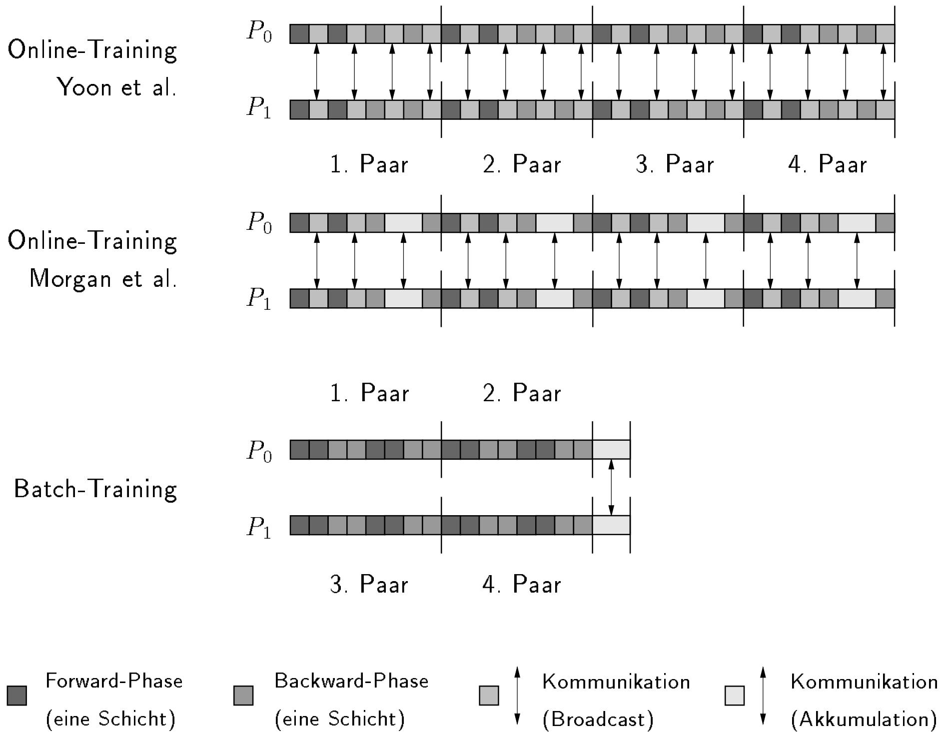


Abbildung 4.6: Kommunikation vs. Kalkulation

Backpropagation-Algorithmus nicht parallel zueinander ablaufen können. Auch die Berechnungsphasen selbst sind verhältnismäßig kurz. Zu kleine Häppchen machen die Parallelisierung unrentabel, weshalb sich das Online-Training erst für große MP-Netze mit einer entsprechenden Anzahl Neuronen lohnt.

Der hohe Kommunikationsanteil und auch die enge Verzahnung zwischen Kommunikation und Kalkulation haben zur Folge, daß die Parallelisierung des Online-Trainings nur auf solchen Architekturen Sinn macht, die über entsprechend leistungsstarke Kommunikationswege verfügen. Aus diesem Grund sind beide Ansätze des Online-Trainings auf Parallelrechnern unter Parix (siehe Tabelle 4.1) implementiert worden.

4.4 Parallele Implementationen

4.4.1 Parix

Parix [16] (PARallel extensions to UnIX) stellt dem Programmierer ein einheitliches Betriebssystem für alle Parallelrechner der Firma Parsytec zur Verfügung. Sowohl Transputer- als auch PowerPC-basierte MIMD-Rechner mit verteiltem Speicher können damit betreiben werden. Es bietet zum einen die volle Funktionalität eines UNIX-Betriebssystems, zum anderen eine optimale Unterstützung des Message-Passing durch die Parallelhardware.

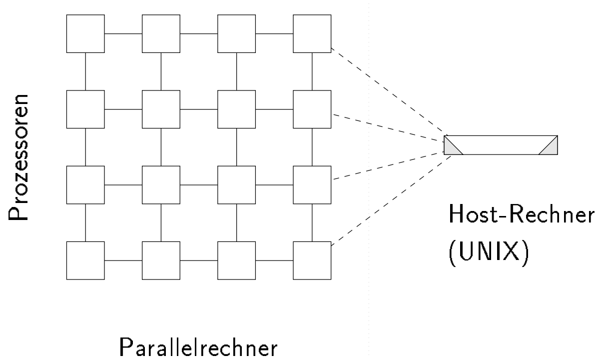


Abbildung 4.7: Ein Parallelrechner unter Parix

Eine UNIX-Workstation, der Host-Rechner (siehe Abbildung 4.7) versorgt den Parallelrechner mit Programmcode und Daten. Ein-/Ausgabefunktionen können von beliebigen Prozessoren des Parallelrechners ausgeführt werden und lösen einen *Remote Procedure Call* auf dem Host-Rechner aus. Auf diese Art wird der Zugriff auf Dateisystem und Netzwerkkomponenten des Host-Rechners realisiert.

Die Prozessoren des Parallelrechners sind als zweidimensionales Gitter verschaltet, über das benachbarte Prozessoren miteinander kommunizieren. Diese Topologie wird standardmäßig von Parix unterstützt. Darüberhinaus bietet Parix die Möglichkeit, eine Verbindung zwischen zwei beliebigen Prozessoren zu etablieren (*virtuelle Links*). Das *Routing* eines virtuellen Links wird vom Betriebssystem verwaltet. Eine Menge virtueller Links kann zu einer *virtuellen Topologie* zusammengefaßt werden. Die geläufigsten Topologien (Pipe, Ring, Gitter, Baum, etc.) können durch Bibliotheksfunktionen erzeugt werden.

Eine parallele Applikation läuft auf einer rechteckigen Prozessorpartition, deren Ausmaße beim Programmstart angegeben werden. Alle Prozessoren der Applikation erhalten initial den gleichen Programmcode (SPMD-Modell). Jeder Prozessor kennt seine Gitterposition. Diese kann genutzt werden, um differenziert weiteren Programmcode nachzuladen (MIMD-Modell).

Parix unterstützt sogenannte *Threads*. Hierbei handelt es sich um leichtgewichtige Prozesse, die im Kontext des erzeugenden Prozesses asynchron ablaufen. Ein Thread hat Zugriff auf alle globalen Variablen der Applikation und kann parallel zum Hauptprogramm eigenständig operieren. Threads stellen eine vergleichsweise günstige Resource dar, die häufig zur Abwicklung

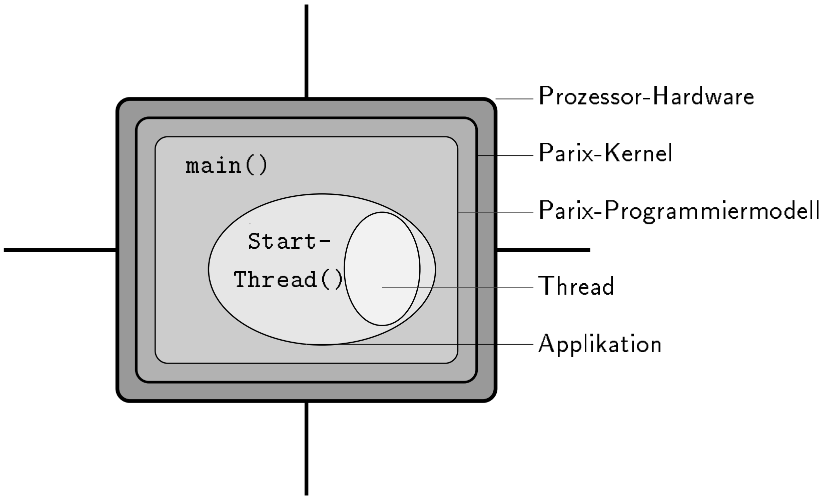


Abbildung 4.8: Threads unter PARIX

asynchroner Kommunikation eingesetzt wird. Zur Implementierung paralleler Programme steht sowohl ein ANSI-C als auch ein Fortran 77 Cross-Compiler einschließlich Bibliotheken für das jeweilige Zielsystem auf dem Host-Rechner zur Verfügung. Nur die Programmausführung selbst findet auf dem Parallelrechner statt. Für PARIX entwickelte Programme sind quelltextkompatibel, d.h. um ein Programm einer anderen als der bisherigen Architektur zugänglich zu machen, ist nur ein Übersetzungsvorgang notwendig.

Rechner	Prozessortyp	MByte	Prozessoren/Knoten	Standort
MC-2	T800	2 / 4	64	Uni Osnabrück
GCel	T805	4	1024	PC^2 Paderborn
PowerXplorer	MPC 601	8	4	Uni Osnabrück
GC/PP	MPC 601	64	96	PC^2 Paderborn

Tabelle 4.1: Unter PARIX eingesetzte Hardware

Tabelle 4.1 zeigt eine Auflistung der Parallelrechner, für die unter PARIX mit einem parallelen Backpropagation-Algorithmus ein MP-Netz trainiert wurde. Die Rechner vom Typ GC sind am Paderborn Center for Parallel Computing (PC^2) beheimatet. Die PowerPC-basierten Systeme sind in Knoten organisiert, in denen jeweils ein T805 Transputer zur Kommunikation eingesetzt wird. Der GC/Power Plus verfügt in seiner jetzigen Ausbaustufe über 96 Knoten mit jeweils 2 MPC 601 Prozessoren.

4.4.2 Realisation unter Parix

Die parallele Implementation sowohl des Batch- als auch des Online-Trainings benutzen denselben Rahmen an Funktionen und Variablen. Durch das Betriebssystem bekommen alle Prozessoren die Parameter, mit denen das PARIX-Programm gestartet wurde. In einem ersten Schritt wird eine alle Prozessoren umspannende virtuelle Ringtopologie etabliert, auf der die Kommunikation stattfindet. Diese wird für eine gerade Anzahl Prozessoren in mindestens einer Richtung optimal auf die vorhandene Gittertopologie abgebildet (Abbildung 4.9). Die Ausführung von Ein-/Ausgabefunktionen findet nur auf einem Knoten statt. Dieser lädt alle relevanten Daten wie beispielsweise Netzgewichte oder Trainingsdaten und verteilt diese über den Ring. Eine Baumtopologie ist zwar besser geeignet, Daten zu verteilen, doch fällt diese Phase im Vergleich zur Trainingsdauer kaum ins Gewicht. Viel entscheidender ist eine Umsetzung der globalen Operationen All-to-All-Broadcast bzw. All-to-All-Akkumulation.

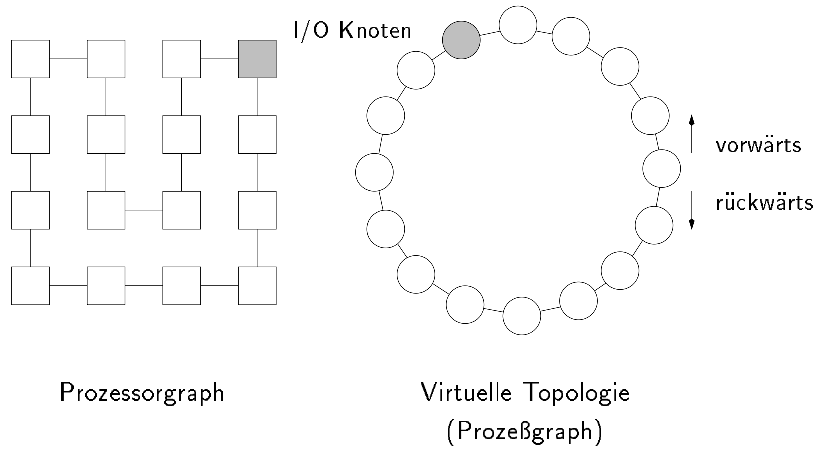


Abbildung 4.9: Ringtopologie

Jeder Prozessor hat zu Beginn des Trainings sowohl das komplette MP-Netz für das Batch-Training als auch sämtliche Trainingspaare für das Online-Training in seinem lokalen Speicher. Dieses erlaubt die gemeinsame Benutzung eines Großteils der Funktionen.

Für das Batch-Training berechnet jeder Prozessor seinen Ausschnitt der Trainings- und der Testmenge. Durch eine Akkumulation über alle Prozessoren werden nach einer deutlich kürzeren Epoche die Gewichtsänderungen angeglichen. Hierzu verschickt jeder Prozessor seine Änderungen für Schwellwerte und Gewichte *zu* seinem *vorderen* Nachbarn. *Von* seinem *hinteren* Nachbarn empfängt er dessen Änderungen. Diese schickt er im folgenden Schritt weiter nach vorne, addiert sie zu seinen Änderungen und empfängt die nächsten.

Für p Prozessoren erfordert die Akkumulation $p-1$ derartiger Kommunikationsschritte auf einer Ringtopologie. Auf einem zweidimensionalen Torus ließe sich diese Zahl auf $2(\sqrt{p}-1)$ reduzieren

[13]. Doch ist eine Einbettung dieser Topologie in die Prozessortopologie in der Regel nicht ohne Kantendehnung möglich. Die dadurch entstehenden Verluste treiben die Kosten wieder in einen Bereich, den eine Ringkommunikation verursacht.

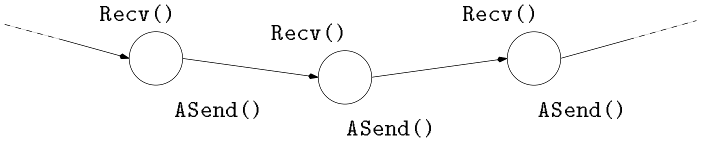


Abbildung 4.10: Kommunikation auf dem Ring

Um das Senden und Empfangen möglichst zügig zu bewerkstelligen, werden gleichzeitig zwei Kommunikationskanäle eines Prozessors bedient. Die PARIX-Funktion `ASend()` startet einen Thread, der im Hintergrund Daten zum Vordermann verschickt. Gleichzeitig empfängt das Hauptprogramm mittels `Recv()` Daten vom Hintermann. Die Benutzung der PARIX-Funktion `Send()` statt `ASend()` würde das Hauptprogramm blockieren bis sich ein Abnehmer für die Daten findet. Um keinen *Deadlock* zu provozieren, müßte dann in $2(p-1)$ Schritten alternierend kommuniziert werden.

Das `Recv()` synchronisiert die Prozessoren und sorgt dafür, daß das nächste `ASend()` mit korrekten Daten aufgerufen wird. Ohne diese Synchronisation würde `ASend()` im schlimmsten Fall $(p-1)$-mal den gleichen Vektor verschicken.

Der All-to-All-Broadcast beim Online-Training geschieht nach dem gleichen Prinzip, nur daß ein Prozessor eingehende Werte statt sie aufzuaddieren entsprechend der Partitionierung in seine lokalen Daten einsortiert. Da hierbei keine Fließkommaberechnungen anfallen, kann die gesamte Kommunikation ungestört und somit zügiger als die globale Akkumulation abgearbeitet werden.

Für die während und nach dem Training stattfindende Fehlerberechnung der Trainings- und Testmenge hat sich auch für das Online–Training eine Partionierung der Musterpaare als vorteilhaft erwiesen. Zwar ist hierfür ein Broadcast notwendig, um auf allen Prozessoren wieder identische Schwellwerte und Gewichte zu haben, doch entfällt abgesehen von der abschließenden Akkumulation des Fehlers weitere zeitraubende Kommunikation.

Die Integration des MouseProp-Algorithmus ist relativ problemlos. Die Berechnung der Oszillation eines Neurons kann parallel geschehen, die Prozessoren tauschen die Werte mit einem All-to-All-Broadcast untereinander aus. Die weitere Ermittlung eines aufzuspaltenden Neurons geschieht sequentiell. Der einfachste Weg, die Gewichte und Schwellwerte auf allen Prozessoren konsistent zu halten, besteht darin, mit einem Broadcast die MP–Netze anzugleichen, das Neuron aufzuspalten und für das quasi neue MP–Netz eine Lastverteilung zu ermitteln. Die im Anschluß stattfindende Bestimmung des *sbic*, erfordert die Berechnung der Fehlerwerte auf Trainings- und Testmenge. Der eigentlich dazu nötige Broadcast entfällt, da bereits für den Aufspaltungsvorgang die Gewichte und Schwellwerte ausgetauscht worden sind.

Nach Abschluß des Training werden noch einmal Schwellwerte und Gewichte angeglichen, um sie zum Beispiel über den I/O-Knoten in eine Datei zu schreiben.

Implementiert wurde der parallele Backpropagation-Algorithmus in ANSI-C unter PARIX in der Version 1.2 für reine Transputersysteme bzw. 1.3 für PowerPC-basierte Parallelrechner.

4.4.3 PVM

PVM (Parallel Virtual Machine) läßt ein heterogenes Netz von UNIX-Workstation und spezieller Parallelhardware als einen einzigen MIMD-Parallelrechner mit verteiltem Speicher erscheinen. Im Gegensatz zu PARIX handelt es sich bei PVM nicht um ein Betriebssystem, sondern um eine reine Laufzeitumgebung, die auf bestehenden Konzepten aufbaut. Unter UNIX-Workstations werden Daten via TCP/IP transferiert, auf Parallelrechnern werden dortige Message-Passing-Protokolle (zum Beispiel PARIX) benutzt (siehe Abbildung 4.11).

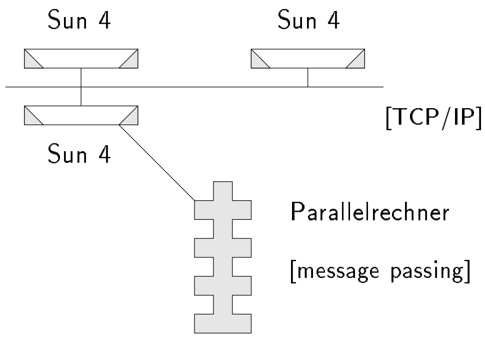


Abbildung 4.11: Virtueller Parallelrechner unter PVM

Das PVM-System besteht im wesentlichen aus zwei Teilen: Auf jedem beteiligten Rechnerknoten bearbeitet ein *PVM-Daemon* (`pvmd`) Anfragen von PVM-Applikationen (*task*). Diese Anfragen stammen von Funktionen der *PVM-Bibliothek*. Sie enthält alle Routinen, die zur Koorperation zwischen den Tasks einer Applikation nötig sind. Die Bibliothek unterstützt neben blockierender und nichtblockierender Task-zu-Task-Kommunikation auch Operationen auf Prozeßgruppen, wie Broadcasts und Synchronisationsbarrieren. Unterstützt werden die Programmiersprachen C, C++ und Fortran 77.

Eine Applikation besteht aus einer Menge von verteilten Tasks, die über ein Message-Passing-Protokoll kommunizieren. Tasks ebenso wie die beteiligten Rechner werden von PVM dynamisch verwaltet und können von der Applikation aus kontrolliert werden. Dieses erlaubt eine Programmierung im MIMD bzw. SPMD Modell. PVM kennt keine eigentliche Prozessortopologie, es sind beliebige Task-zu-Task Verbindungen möglich. Die Identifikation geschieht über eine eindeutige Task-ID (*TID*).

PVM ist für mehr als 50 Architekturen als Public Domain Software frei verfügbar. Dazu kommen noch mehrere kommerzielle Implementationen für Parallelrechner. Herausragender Vorteil der Formulierung paralleler Algorithmen mit PVM ist die Portabilität der Programme. Diese können komfortabel auf einem Workstation-Netz implementiert und getestet werden. Erst die eigentliche Anwendung benutzt eine parallele Architektur. Diese Portabilität hat natürlich gerade bei kommunikationsintensiven Algorithmen ihren Preis (siehe Abschnitt 4.5.2).

4.4.4 Realisation unter PVM

Unter PVM wurde eine parallele Variante des Batch-Trainings implementiert. Um auch hier das SPMD-Konzept verfolgen zu können wird das eigentliche Programm `bp` von einem anderen PVM-Programm (`mpil`) aus gestartet. Dieses bekommt neben dem auszuführenden Programm und dessen Argumenten auch die Anzahl der Tasks, die durch die PVM-Funktion `pvm_spawn()` gestartet werden. Das PVM-Laufzeitsystem verteilt diese Tasks auf die beteiligten Rechner der virtuellen Maschine (siehe Abbildung 4.12).

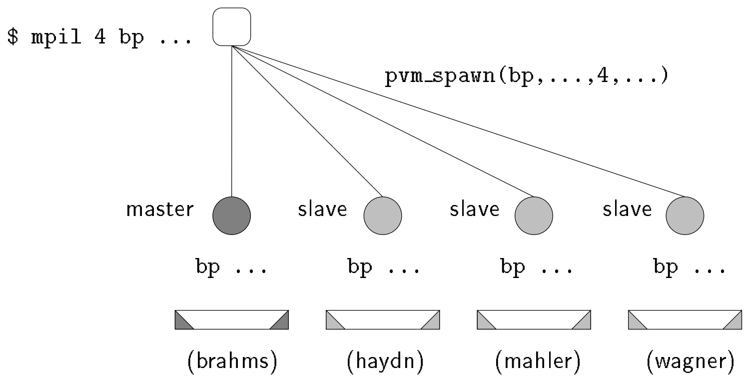


Abbildung 4.12: SPMD–Modell unter PVM

Die für die Kommunikation notwendigen Tasks-IDs empfangen die `bp`-Prozesse von ihrem Vater, der danach terminiert. Erst jetzt beginnt die eigentliche parallele Applikation. Analog zur Implementation unter PARIX wird eine Task bestimmt, die unter anderem die Dateioperationen erledigt. Über diese *Master*-Task läuft auch jegliche Kommunikation.

Alle `bp` Tasks werden Mitglied einer Gruppe. Sie dient zur Synchronisation der Prozesse. Die Master-Task verteilt Netzkonfiguration, Gewichte und Schwellwerte sowie Test- und Trainingsdaten. Nachdem jede Task die Größe ihrer Partition ermittelt hat, beginnt das Training. Nach Abarbeitung der lokalen Trainingsmenge findet eine All-to-All-Akkumulation der Gewichtsänderungen statt.

Da unter PVM jede Kommunikation a priori gleich teuer ist und auf einem Workstation-Verbund alle Rechner sich einen Kommunikationsweg teilen müssen, zielt die Realisierung der All-to-All-Akkumulation darauf ab, die globale Anzahl der Kommunikationen zu minimieren. Deshalb sammelt die Master-Task die Partialsummen auf, addiert sie zusammen und schickt das Ergebnis an die restlichen Tasks zurück (siehe Abbildung 4.13).

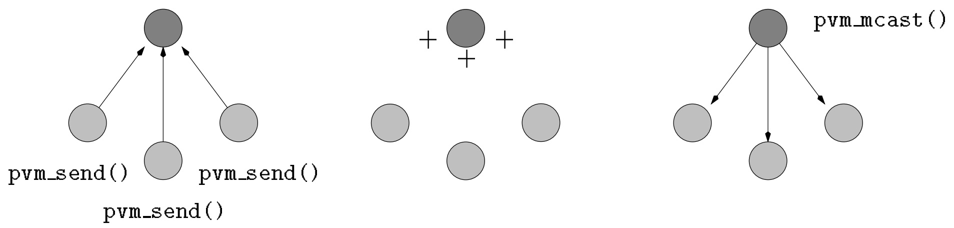


Abbildung 4.13: Globale Akkumulation unter PVM

Da nicht zwei Prozessoren gleichzeitig senden bzw. empfangen können, benötigt diese Operation für p Prozessoren $2(p-1)$ einzelne Kommunikationsaufrufe. Virtuelle Topologien wie unter Parix sind nicht vorgesehen. Auf Workstation-Clustern machen sie auch keinen Sinn, da sie nicht die Kommunikationsleistung steigern.

Implementiert wurde das parallele Batch-Training unter PVM 3.3.7 auf einem Workstation-Cluster und unter PVM/PARIX, einer PVM 3.2 Implementation für Parallelrechner unter PARIX.

4.5 Ergebnisse

4.5.1 Experimentelle Speedups

Der *Speedup*-Faktor s eines parallelen Rechners gibt an, wieviel mal schneller ein paralleler Algorithmus im Vergleich zum sequentiellen abgearbeitet werden kann:

$$s_p = \frac{t_1}{t_p}$$

Hierbei steht t_1 für die *sequentielle* Laufzeit auf einem Prozessor und t_p für die Laufzeit des *parallelen* Programms auf p Prozessoren. Für Algorithmen, die keine Überlappung von Kommunikation und Berechnung zulassen, setzt sich die parallele Laufzeit t_p folgendermaßen zusammen:

$$t_p = \frac{t_1}{p} + t_{comm}$$

Da die Gesamtdauer der Kommunikationen t_{comm} in jeden Fall echt größer null ist, kann der Speedup s_p nie den Wert p erreichen. Ein dem Speedup verwandtes Kriterium ist die *Effizienz* e, die ein parallels Programm erreicht:

$$e_p = \frac{s_p}{p}$$

In der Praxis sinkt für eine feste Problemgröße die Effizienz mit zunehmender Prozessoranzahl. Dies liegt zum einen daran, daß die Problemhappen eines Prozessors kleiner werden, zum andern aber die Zeit für die Kommunikation im besten Fall konstant bleibt, meistens eher steigt.

Die Trainingsdaten für das *parallele Batch-Training* stammen aus dem Verhulst-Prozeß (siehe Abbildung 3.10). Die Aufgabe entspricht der aus Abschnitt 3.5.2. Bei Präsentation der letzten drei Werte der Zeitreihe soll der zukünftige Wert prognostiziert werden. Trainiert wird ein MP-Netz mit 3 Eingabeneuronen, 6 Neuronen in der verborgenen Schicht und einem Ausgabeneuron. Die Zahl der zu trainierenden Paare beträgt 128, 1280 und 12800. Die Laufzeitmessungen wurden auf 3 Architekturen durchgeführt.

Ein Workstationverbund von Sun Sparc 10/20 kam unter PVM 3.3.7 zum Einsatz. Auf dem GCel und dem GC/PP (siehe Tabelle 4.1) wurde sowohl unter PARIX aus auch unter PVM trainiert. Der MC-2 und der PowerXplorer werden hier nicht betrachtet, da sie sich weitgehend nur bezüglich der Anzahl der Prozessoren von den anderen beiden Parallelrechnern unterscheiden.

Für vier Workstations zeigt sich eine moderate Beschleunigung des Trainings (siehe Abbildung 4.14). Wesentlich effektiver sind jedoch echte Parallelrechner. Auf dem GCel (siehe Abbildung 4.15) wird unter PARIX eine maximale Beschleunigung von knapp 67 auf 128 Prozessoren erreicht. Unter PVM knickt der Speedup-Graph sehr früh ab. Eine Ursache hierfür ist die mangelnde Unterstützung der lokalen Kommunikationsfähigkeiten, die unter PARIX ausgenutzt werden. Eine ähnliche Situation zeigt sich bei der letzten Architektur, dem GC/PP(siehe Abbildung 4.16). Auch hier läßt sich unter PARIX eine ganz passable Beschleunigung erreichen, während unter PVM der Speedup bereits bei 16 Knoten stagniert.

Für alle betrachteten Architekturen wird allerdings erst ab einer Problemgröße von mehr als 10000 Trainingsmustern ein respektabler Speedup erreicht. Weniger Trainingspaare sind bei einer großen Prozessoranzahl nicht rentabel.

Die Trainingsdaten für das *parallele Online-Training* kommen ebenfalls aus dem Umfeld der Zeitreihenprognose. Die trainierten Netze weisen 583 Eingabeneuronen, 256 verborgene Neuronen und ein Ausgabeneuron auf. Die Trainingsmenge besteht aus 20 Paaren. Die zugrundeliegende Zeitreihe beschreibt den Abverkauf eines Artikels in einem Supermarkt [22], [23]. Beide Varianten des parallelen Online-Trainings wurden unter PARIX auf dem GCel und dem GC/PP verwandt.

Abbildung 4.17 zeigt für jede Architektur den Speedup-Graphen beider Varianten. Für die gewählte Netztopologie zeichnen sich kaum Unterschiede zwischen der Idee von YOON et al.

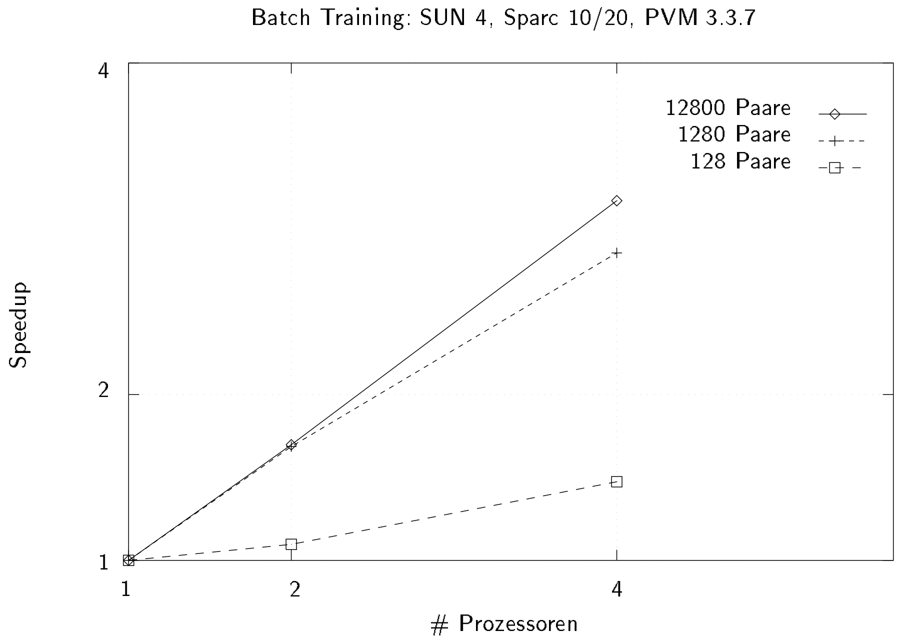


Abbildung 4.14: Speedups für das parallele Batch-Training auf einem Workstationverbund

und der von MORGAN et al. ab. Auf dem GC-el wird immerhin noch ein Speedup von 20 auf 32 Prozessoren erzielt. Auf dem GC/PP wird bereits bei 16 Knoten die maximale Beschleunigung erreicht.

4.5.2 Vergleich und Bewertung der Parallelisierungen

Für das parallele Batch Training ergeben sich gerade bei großen Trainingsmengen gute Beschleunigungswerte. Durch eine hinreichend große Anzahl an Trainingspaaren kann auch bei einer großen Prozessorzahl eine hohe Effizienz erzielt werden. Die für Kommunikation verwandte Zeit richtet sich allein nach der Größe der zu verschickenden MP–Netze und der Anzahl der Prozessoren. Dadurch kann ein günstiges Verhältnis zwischen Kommunikationsdauer und Rechenzeit erreicht werden.

Der Vergleich der Speedup-Graphen zeigt, daß die Verwendung von PVM unter PARIX auf Parallelrechnern nicht zu empfehlen ist, da insbesondere bei großen Prozessorzahlen sehr viel an Zeit durch Kommunikation verloren geht (siehe Abbildungen 4.15 und 4.16). Zwar enthält der PVM-Standard einige globale Funktionen wie etwa `pvm_gather()`, die hier verwendeten globalen Operationen müssen jedoch unter PVM explizit formuliert werden. Um portabel zu programmieren bietet sich stattdessen der MPI-Standard [12] an. Zum Funktionsumfang von MPI gehören kollektive Funktionen, die bereits das bieten, was unter PVM nachgebildet werden muß. Eine MPI-Implementation kann diese globalen Operationen auf dem jeweiligen Zielsystem optimal unterbringen, sodaß weniger Overhead entsteht.

Die Speedup-Graphen für das parallele Online-Training zeigen sowohl bei der Variante von Yoon et al. als auch bei der von Morgan et al. die Folgen des Kommunikationsaufwands. Die für diese Parallelisierung entscheidende Größe ist die Anzahl der Neuronen pro Schicht. Offensichtlich ist das zu verteilende Rechenpotential schon bei einer kleinen Anzahl Prozessoren erschöpft. Durch eine größere Anzahl verborgener Neuronen läßt sich eine höhere Effektivität erzielen. Eine Verbesserung bringt auch eine größere Anzahl von Ausgabeneuronen, doch ist diese durch die Problemstellung vorgegeben und dementsprechend schwieriger abzuändern als die Anzahl verborgener Neuronen.

Bei Problemen dieser Größenordnung zeigt sich kaum ein Unterschied zwischen den beiden Varianten. Der Vorteil einer zügigeren Kommunikation bei der Idee von Yoon et al. geht zu Lasten eines erhöhten Rechenaufkommens, da einige Gewichtsänderungen doppelt auszuführen sind. Je höher die Anzahl verborgener Neuronen ist, desto mehr fallen diese Berechnungen ins Gewicht und kompensieren den ursprünglichen Vorteil.

Aufgrund der Rahmenbedingungen des Backpropagation-Algorithmus sind für alle Parallelisierungen globale Kommunikationen erforderlich. Die Kommunikationszeit wächst linear mit der Anzahl der Prozessoren, während die Berechnungsaufgaben im gleichen Maß schrumpfen. Der Speedup fällt dann wieder ab, wenn die globale Kommunikation eines Ergebnisses länger dauert als dessen Berechnung.

Transputer verfügen über eine für heutige Verhältnisse geringe Rechenleistung (1,5 MFLOPS), können aber verhältnismäßig schnell kommunizieren. PowerPC basierte Rechner verfügen über die gleiche Kommunikationsleistung bei vielfach höherer Verarbeitungsgeschwindigkeit (80 MFLOPS). Kommunikationen sind damit auf dem GC/PP relativ teuer, weshalb sich auf dem GCel bessere Beschleunigungen erreichen lassen.

Die Parallelisierung des Batch-Trainings erzielt unter PVM auf einem Workstation-Verbund gute Ergebnisse. Ausgezeichnete Resultate sind unter Parix erreicht worden. Die Parallelisierung des Online-Trainings zeigt auf moderat parallelen Systemen eine ansprechende Beschleunigung. Insgesamt betrachtet, ist also durch die Parallelisierung des Backpropagation-Algorithmus die Trainingsdauer von MP-Netzen erheblich verkürzt worden.

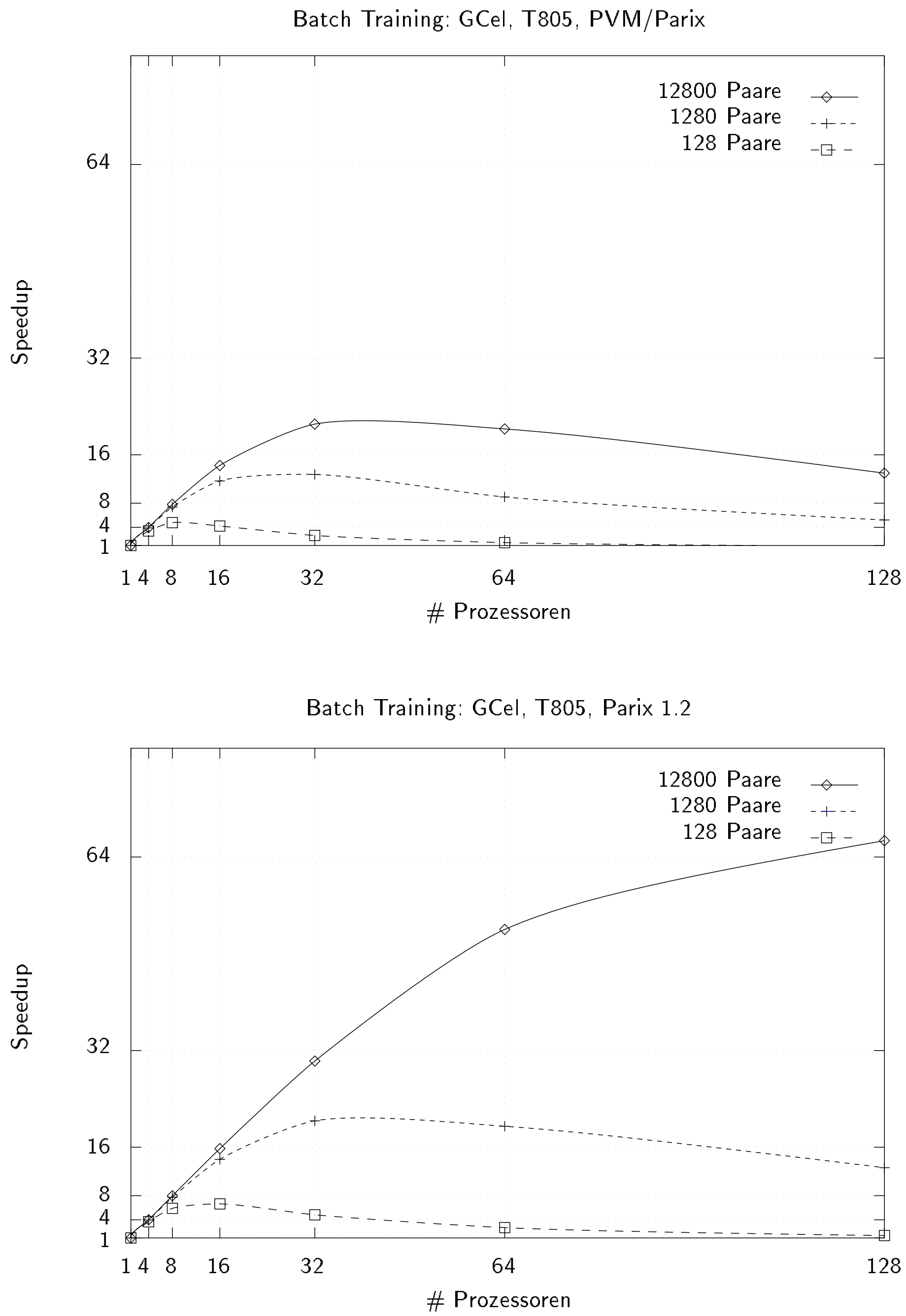
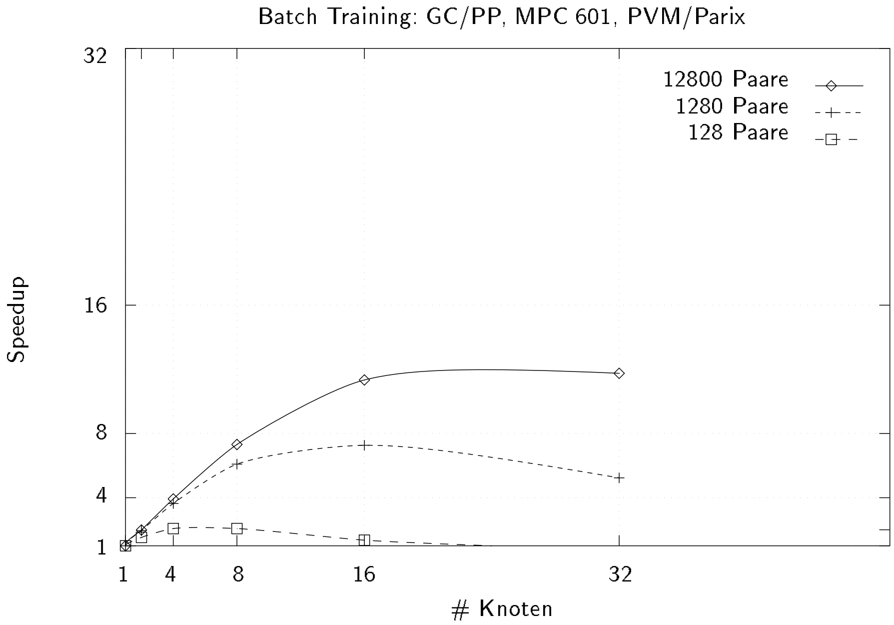


Abbildung 4.15: Speedups für das parallele Batch-Training auf dem GCel



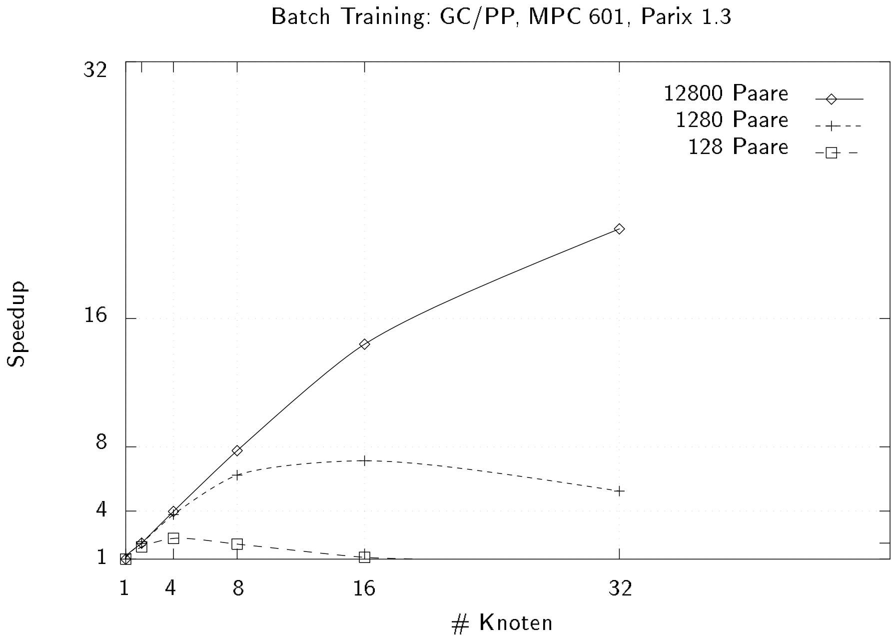
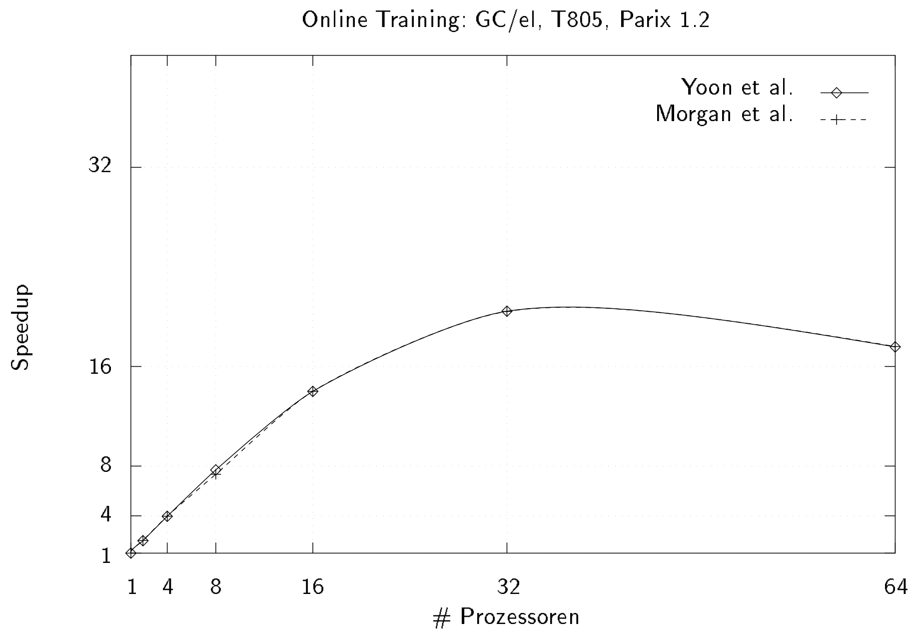


Abbildung 4.16: Speedups für das parallele Batch-Training auf dem GC/PP



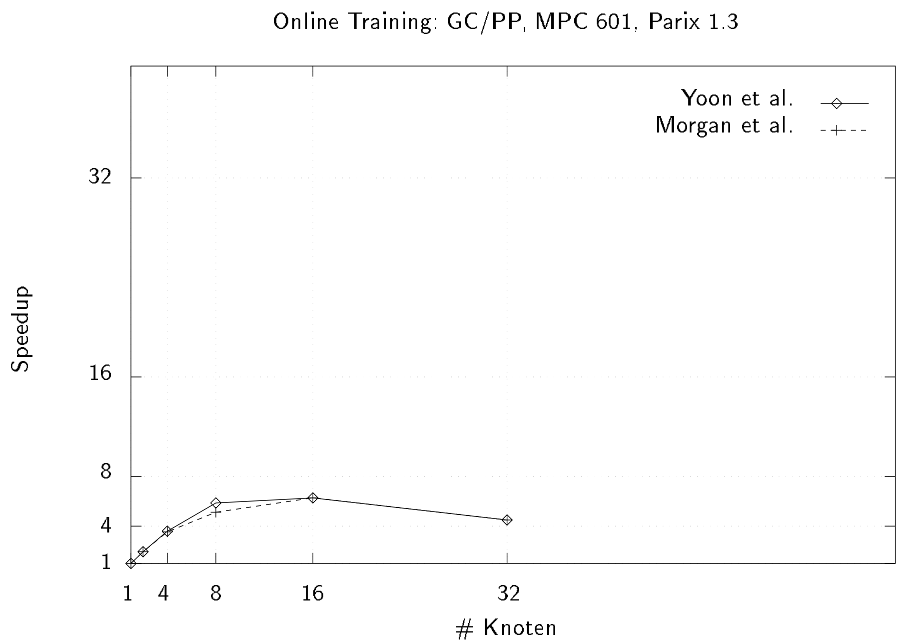


Abbildung 4.17: Speedups für das parallele Online-Training

Kapitel 5

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Erstellung eines Programms zur zeitlichen Beschleunigung des Trainings von Mehrschicht-Perzeptron-Netzen. Hierzu wurden mögliche Parallelisierungsansätze des Backpropagation-Algorithmus diskutiert, implementiert und verglichen.

Das im Rahmen dieser Arbeit entstandene Programm `bp` bietet umfangreiche Möglichkeiten zum Training von Mehrschicht-Perzeptron-Netzen. Neben einem Momentumterm unterstützt `bp` das Online- und das Batch-Training. Eine qualitative Verbesserung der Trainingsergebnisse wird durch das Aufspalten verborgener Neuronen mit dem MouseProp-Algorithmus erzielt. Eine wesentliche Verkürzung der Trainingszeit bietet die Parallelisierung des Backpropagation-Algorithmus. Das parallele Batch-Training unter PVM auf einem Workstation-Verbund erzielt gute Ergebnisse. Ausgezeichnete Resultate sind unter PARIX auf Transputersystemen erreicht worden. Die Parallelisierung des Online-Trainings zeigt auf moderat parallelen Systemen eine ansprechende Beschleunigung.

Sowohl PVM als auch PARIX haben sich als komfortable Programmierumgebungen zur Erstellung der parallelen Version von `bp` erwiesen. Für den GCel und den GC/PP standen sowohl eine PVM- als auch eine PARIX-Implementation zu Verfügung. Es stellte sich heraus, daß die Portabilität von PVM sehr zu Lasten der Effektivität geht und deshalb PVM auf Parallelrechnern für diese Anwendung ungeeignet ist. Eine Alternative zu PVM hinsichtlich portabler Programmierung paralleler Systeme stellt der MPI-Standard dar. Dieser unterstützt Operationen für globale Kommunikationen, wie sie auch von `bp` benötigt werden. Kommerzielle MPI-Implementationen können die Gegebenheiten eines Parallelrechners berücksichtigen und globale Operationen optimal auf die zugrundeliegende Architektur abbilden. Aus diesem Grund erscheint eine Portierung des parallelen Programms `bp` nach MPI als sehr sinnvoll.

Die Erweiterung von `bp` um zusätzliche Lernalgorithmen für MP-Netze wie zum Beispiel QPROP oder RPROP bietet sich an. Sie ermöglichen ein Training in noch weniger Epochen. Ihre Parallelisierung erfordert allerdings einen höheren Aufwand.

Einen vom Standpunkt der Parallelverarbeitung her interessanten Ansatz stellt das Training eines MP-Netzes mit Unterstützung genetischer Algorithmen dar. Sowohl die Bestimmung optimaler Lernparameter und Netztopologie als auch die Optimierung der Gewichte selbst kann mit Hilfe genetischer Algorithmen erfolgen und verspricht einen hohen Grad an Parallelität.

Abschließend kann festgehalten werden, daß das Programm `bp` eine deutliche Verringerung der langen Trainingszeiten von Mehrschicht-Perzeptron-Netzen ermöglicht, indem unter Ausnutzung der natürlichen Parallelität künstlicher neuronaler Netze effiziente Strategien eines parallelen Backpropagation-Algorithmus entwickelt wurden.

Anhang A

bp — Ein Simulator für MP-Netze

A.1 Installation

A.1.1 Voraussetzungen

- Das Makefile zum Übersetzen von `bp` benötigt `gmake`. Andere `make` Kommandos führen zu Fehlermeldungen.
- `bp` wurde in ANSI-C programmiert und läßt sich in seiner sequentiellen Form auf allen UNIX-Architekturen, die über einen ANSI-C Compiler wie zum Beispiel den `gcc` verfügen, übersetzen.
- Für das parallele Programm auf einem Workstation-Cluster wird PVM in einer Version > 3.3 benötigt. Parix sowohl in der Version 1.2 als auch inder Version 1.3 enthält alle notwendigen Programme und Bibliotheken zum Übersetzen von `bp` für Parallelrechner.

A.1.2 Dateistruktur

- `bp/man/man1/bp.1` — Hilfeseite mit einer Kurzbeschreibung von `bp`
- `bp/make` — Shellscript, Abkürzung für `gmake -f GnuMakefile`
- `bp/Arch.mak` — Architekturspezifische Vereinbarungen für `bp/Bp.mak`
- `bp/Bp.mak` — Regeln zum Übersetzen von `bp`.
- `bp/GnuMakefile` — Frontend für `bp/Bp.mak`
- `bp/SEQUENT/`, `bp/PVM/` und `bp/PARIX/` — Kataloge in denen zu jeder Architektur Objekt-Dateien und Programme abgelegt werden
- `bp/src/` — Katalog mit Quelltexten

A.1.3 Makefile

Für die UNIX-Architekturen `HP700`, `m68k` (NeXT), `i486` (Linux), `sun3`, `sun4` und `rs6000` sind bereits Anpassungen in die Datei `bp/Arch.mak` aufgenommen worden. Ist die Ausgabe des Kommandos `arch` nicht in dieser Menge enthalten, so wird versucht, `bp` mit dem `gcc` zu übersetzen.

Targets

`make ?`
listet alle verfügbaren Targets auf. Die in `bp/GnuMakefile` gesetzte Variable `XDIR` enthält den Katalog, wo das ausführbare Progamm abgelegt wird. Voreingestellt ist

- `$(HOME)/bin/$(ARCH)` — für sequentielle Programme unter UNIX
- `$(HOME)/pvm3/bin/$(PVM_ARCH)` — für PVM-Programme
- `$(HOME)/parix/bin/$(ARCH)` – für PARIX-Programme

A.2 Benutzung

A.2.1 Name

`bp` - ein Programm zur Simulation eines Mehrschicht-Perzeptron-Netzes (MP-Netz, *feedforward multilayer perceptron*), welches mit dem Backpropagation Algorithmus trainiert wird.

A.2.2 Aufruf

```
bp {-c <Konfiguration>}
   {-r <Dateiname>}
   {-q <Dateiname1> <Dateiname2>}
   {-t [/s[=<alle #Epochen>]]
       [/l=<Lernrate>]
       [/p=<alle #Epochen>]
       [/i=<Impulsrate>]
       [/b]
        /e=<#Epochen> | /g[=<Fehler>]
       <Dateiname>}
   {-l <prio> <Dateiname>}
   {-w[a] <Dateiname>}
```

Die Reihenfolge der Parameter in der Kommandozeile ist maßgebend für die Abarbeitung.

A.2.3 Netztopologie

`bp -c <Topologie>` legt ein MP-Netz (siehe Abbildung 3.1) an und initialisiert die Schwellwerte (*bias*) und Gewichte (*weights*) mit Zufallszahlen aus dem Intervall]-1,1[. Die `<Topologie>` besitzt folgende Syntax:

```
<Anzahl der aktiven Schichten>:
:<Anzahl der Neuronen der Eingabeschicht>
:<initiale Anzahl der Neuronen der ersten verborgenen Schicht>
 (<maximale Anzahl der Neuronen der ersten verborgenen Schicht>)
   .
   .
   .
:<initiale Anzahl der Neuronen der letzten verborgenen Schicht>
 (<maximale Anzahl der Neuronen der letzten verborgenen Schicht>)
:<Anzahl der Neuronen der Ausgabeschicht>
```

Beispiel

`$ bp -c "3:30:15(25):10(20):1"`
initialisiert ein MP-Netz, welches 3 aktive Schichten (neben der inaktiven Eingabeschicht) besitzt. Die Eingabeschicht besitzt 30 Neuronen, die erste vorborgene Schicht besitzt 15 Neuronen und kann maximal deren 25 aufnehmen. Die zweite verborgene Schicht besitzt 10 Neuronen und maximal 20. Die Ausgabeschicht besitzt ein Neuron. Die Angabe einer maximalen Anzahl, verborgener Neuronen ist für des Training mit dem MouseProp-Algorithmus relevant (siehe Abschnitt 3.4).

Anmerkungen

- Die Ausmaße eines Netzes unterliegen keiner Beschränkung.
- Da viele Shells die Klammern `()` als Metazeichen auffassen, ist man gut beraten, die Topologie zu schützen, in dem sie zum Beispiel in Anführungszeichen eingefaßt wird, um nicht mit der Shell in Konflikt zu geraten.

A.2.4 Netzgewichte einlesen

`bp -r <Dateiname>` initialisiert die Schwellwerte und Gewichte eines MP-Netzes mit den Werten aus der Datei `<Dateiname>`. Wurde zuvor noch kein MP-Netz angelegt, so erledigt dies `-r` mit der in der Datei abgespeicherten Topologie. Ansonsten wird überprüft, ob die Werte mit dem bisherigen MP-Netz verträglich sind.

Beispiel

```
$ bp -r xor.mp
```
liest Gewichte und Schwellwerte aus der Datei xor.mp.

Anmerkungen

- Hat <Dateiname> den Wert stdin, so erwartet bp -r seine Daten über die Standardeingabe.
- In der Regel wird man nur eine der beiden Befehle -c oder -r benutzen; einzige Ausnahme: Die maximale Anzahl der Neuronen pro Schicht eines abgespeicherten Netzes soll geändert werden.

Beispiel

```
$ bp -c "2::6:3(3):1" -w small.mp
$ bp -c "2::6:3(12):1" -r small.mp -w big.mp
```

A.2.5 Abfrage

bp -q <Dateiname1> <Dateiname2> führt eine Netzabfrage durch, d.h. gemäß der Netztopologie werden jeweils <Anzahl der Neuronen der Eingabeschicht> Werte über die Datei <Dateiname1> eingelesen, durch das Netz propagiert und <Anzahl der Neuronen der Ausgabeschicht> Werte mit Leerzeichen getrennt in die Datei <Dateiname2> geschrieben.

Die Abfrage endet mit dem Erreichen des Dateiendes von <Dateiname1> bzw. mit einem fehlgeschlagenen Leseversuch.

Anmerkungen

- Hat <Dateiname1> den Wert stdin, so erwartet bp -q seine Daten über die Standardeingabe.
- Hat <Dateiname2> den Wert stdout, so schreibt bp -q seine Daten in die Standardausgabe.

A.2.6 Training

```
bp -t [/s[=<alle #Epochen>]]
      [/l=<Lernrate>]
```

```
[/p=<alle #Epochen>]
[/i=<Impulsrate>]
[/b]
 /e=<#Epochen> | /g[=<Fehler>]
<Dateiname>
```

trainiert ein MP-Netz mit den in `<Dateiname>` aufgelisteten Trainingspaaren. Das einmalige Training aller Paare heißt Epoche. Das Training endet, falls ein Abbruchkriterium erfüllt ist:

`/e=<#Epochen>`

ist wahr, wenn `<#Epochen>` Epochen trainiert wurden.

`/g[=<Fehler>]`

ist wahr, wenn die Wurzel des mittleren quadratischen Fehlers der Testmenge gestiegen ist und die Differenz zum letzten Wert größer als `<Fehler>` ist, falls dieser angegeben wurde.

Standardmäßig wird der gewöhnliche Backpropagation Algorithmus mit Online-Training ohne Momentum/Impuls und mit einer Lernrate η von 1.0 verwendet (siehe Abschnitt 3.3.4). Die optionalen Parameter `s`,`l`,`i` und `b` dienen dazu, Abweichungen davon genauer zu spezifizieren.

MouseProp-Algorithmus

Die Angabe von `/s[=<alle #Epochen>]` veranlaßt `bp` dazu, während des Trainings verborgene Neuronen aufzuspalten. Ein schlecht trainiertes verborgenes Neuron wird aufgespalten, in der Hoffnung, daß zwei Neuronen das zu schaffen vermögen, was ein Neuron überfordert. Dieses Aufspalten geschieht monoton und fehlerlokal: Die Ergebnisse einer Abfrage bleiben unverändert (monoton) und die Fehlerwerte der nicht betroffenen Neuronen bleiben unverändert (fehlerlokal) (siehe Abschnitt 3.4).

Insbesondere „erbt“ ein derartig gewachsenes Netz die den „kleinen“ Netzen immanente Eigenschaft zur Verallgemeinerung, welche im Normalfall mit zunehmender Anzahl an verborgenen Neuronen verschwindet.

Die Angabe von `/s` sorgt dafür, daß `bp` in gleichmäßigen Abständen, die sich aus
`<Anzahl der Epochen> DIV <Anzahl aufzuspaltender Neuronen>`
ergeben, ein geeignetes Neuron ermittelt und aufgespaltet.

Durch `/s=<alle #Epochen>` läßt sich dieser Abstand explizit angeben. Die Anzahl aufzuspaltender Neuronen ergibt sich als Summe über alle verborgenen Schichten von
`<initiale Anzahl der Neuronen> - <maximale Anzahl der Neuronen>`,
welche mit der Netztopologie angegeben wird (siehe Abschnitt A.2.3).

Lernrate

Durch `/l=<Lernrate>` wird die zu verwendende Lernrate η gesetzt. Hierbei ist `<Lernrate>` ein positiver numerischer Wert, der als Schrittweite beim Gradientenabstieg (*gradient descent*) benutzt wird. In der Regel sollte die Lernrate möglichst klein gewählt werden, falls die zur Verfügung stehende Zeit dies zuläßt.

Protokoll des Fehlerverlaufs

Eine Verfolgung des Fehlers während des Trainings wird durch die Angabe von `/p=<alle #Epochen>` eingeschaltet. Nach `<alle #Epochen>` wird die Wurzel aus dem mittleren quadratischen Fehler sowohl für die Trainings- als auch für die Testmenge bestimmt und das Ergebnis in die Protokolldatei geschrieben (siehe Abschnitt A.2.8).

Impuls/Momentum

Zur Beschleunigung des Trainings erweist es sich bei manchen Problemstellungen als hilfreich, einen Bruchteil der letzten Gewichtsveränderung zur momentanen hinzuzuaddieren (Impuls/Momentum). Durch `/i=<Impulsrate>` wird mit Impuls gelernt. `<Impulsrate>` ist ein numerischer Wert, der den Bruchteil angibt. Sinnvolle Werte liegen zwischen 0 und 1.

Batch-Training

Sollen die berechneten Gewichtsänderungen erst am Ende einer Epoche wirksam werden, so ist `/b` anzugeben.

A.2.7 Aufbau einer Datei mit Trainingsdaten

Trainingsdaten sind die Musterpaare, mit denen ein MP-Netz trainiert wird (Trainingsmenge), bzw. mit denen ein Test auf Generalisierung durchgeführt wird (Testmenge) (siehe Abschnitt 2.3.2).

Zu Beginn der Datei dürfen beliebig viele Kommentarzeilen durch `##` eingeleitet werden. Nach den Kommentarzeilen sind folgende vier Zeilen obligatorisch:

```
# pairs=      <Anzahl der Trainingspaare>
# genpairs=   <Anzahl der Testpaare>
# inputsize=  <Anzahl der Eingabeneuronen>
# outputsize= <Anzahl der Ausgabeneuronen>
```

Hiernach folgen für jedes Trainingspaar jeweils
`<Anzahl der Eingabeneuronen>` + `<Anzahl der Ausgabeneuronen>`
Werte durch Whitespaces getrennt. Soll diese Datei kompatibel mit dem Xerion-Paket[1] sein, so muß nach den Werten für die Eingabeschicht ein Komma `,` folgen und am Ende eines Musterpaares ein Semikolon `;` stehen.

Beispiel

```
## Datei: xor.trn
##
## XOR Problem
##
## Wahrheitstabelle:
##
##        | 0 1
##      --|----
##      0 | 0 1
##      1 | 1 0
##
# pairs=      4
# genpairs=   0
# inputsize=  2
# outputsize= 1
0 0, 0;
0 1, 1;
1 0, 1;
1 1, 0;
```

Beispiel

`$ bp -c 2::2:2(2):1 -t /e=1000/l=2.0/i=0.9 xor.trn`
initialisiert ein MP-Netz mit 2 Eingabeneuronen, zwei verborgenen Neuronen und einem Ausgabeneuron. Anschließend wird es 1000 Epochen mit einer Lernrate von 2,0 und einer Impulsrate von 0,9 mit den Musterpaaren aus der Datei `xor.trn` trainiert.

[1]Xerion ist eine Sammlung von Funktionen zur Simulation künstlicher neuroneler Netze. Xerion wurde an der University of Toronto entwickelt und ist via `ftp://relay.cs.toronto.edu/pub/xerion` frei verfügbar.

A.2.8 Protokolldatei

Der Verlauf und das Resultat einer `bp` Sitzung wird in einer Protokolldatei festgehalten. `bp -l <prio> <Dateiname>` legt fest, wie ausführlich und wohin die Protokollmeldungen geschrieben werden sollen. `<prio>` sollte eine der folgenden Vokabeln sein:

`none`
schaltet das Protokollieren ganz aus,
`minimal`, `medium`, `detailed`
schaltet das Protokollieren mit zunehmender Priorität ein.

Der mittels `<Dateiname>` spezifizierten Datei werden die Protokolldaten der momentanen Sitzung angehängt. Die Protokolldaten selbst werden als Origami-Falten {{{ }}} geschrieben.

Voreinstellung ist `detailed` als Priorität und `bp.log` als Name der Protokolldatei.

Beispiel

Es folgt eine Sitzung mit minimalem Protokoll:

```
{{{  Wed Aug 30 12:36:12 1995
SESSION STARTED : Wed Aug 30 12:36:12 1995
PROGRAM VERSION : $Id: bp.c,v 4.2 1995/08/30 10:11:26 dumiddel Exp $
COMMANDLINE     : bp -l minimal bp.log -c 2::2:2(6):1 -r xor.ini.mp
                       -t /p=50/e=10000/b/s xor.trn -w xor.mp
bp.net_training: time: 1.791 s after 10000 epochs
bp.net_training: configuration: 2::2:6(6):1
bp.net_training: training_error: 0.0092954516435081
bp.net_training: test_error: 0.0000000000000000
bp.net_training: sbic: 10000 -1.1906957079849794
SESSION ENDED   : Wed Aug 30 12:36:14 1995
}}}
```

- `time`: gibt die Dauer des Trainings in Sekunden an
- `configuration`: beschreibt die Topologie des MP-Netzes nach dem Training.
- `training_error`: und `test_error`: stehen für die Wurzel der mittleren quadratischen Fehlers auf Trainings- bzw. Testmenge.
- `sbic`: gibt zu einer Epoche den Wert des *sbic* an (siehe Abschnitt 3.4.5).

Im Vergleich dazu die Sitzung mit ausführlichem Protokoll:

```
{{{  Wed Aug 30 12:27:06 1995
SESSION STARTED : Wed Aug 30 12:27:06 1995
PROGRAM VERSION : $Id: bp.c,v 4.2 1995/08/30 10:11:26 dumiddel Exp $
COMMANDLINE     : bp -l detailed bp.log -c 2::2:2(6):1 -r xor.ini.mp
                       -t /p=50/e=10000/b/s xor.trn -w xor.mp
bp.net_create: configuration: 2::2:2(6):1
bp.net_read:   configuration: 2::2:2(6):1
bp.net_training: file: xor.trn
bp.net_training: parameters for batch learning :
bp.net_training: split: 2000
bp.net_training: eta: 1.000000
bp.net_training: alpha: 0.000000
bp.net_training: max_error: 0.000000
bp.net_training: n_epochs: 10000
bp.net_training: pairs: 4
bp.net_training: genpairs: 0
bp.net_training: err_epochs: 50
bp.net_training: epoch: 50 error: 0.4996622701079532 0.0000000000000000
bp.net_training: epoch: 100 error: 0.4992786814394997 0.0000000000000000
...
bp.net_training: epoch: 1900 error: 0.0310810413518726 0.0000000000000000
bp.net_training: epoch: 1950 error: 0.0305270988669030 0.0000000000000000
bp.net_split: splitted neuron 1 in layer 0 with osc 0.723669
bp.net_training: sbic: 2000 -1.2614548476775369
bp.net_training: epoch: 2000 error: 0.0300006604114873 0.0000000000000000
bp.net_training: epoch: 2050 error: 0.0293746319216904 0.0000000000000000
...
bp.net_training: epoch: 9950 error: 0.0093272219041370 0.0000000000000000
bp.net_training: epoch: 10000 error: 0.0092954516435081 0.0000000000000000
bp.net_training: time: 1.847 s after 10000 epochs
bp.net_training: configuration: 2::2:6(6):1
bp.net_training: training_error: 0.0092954516435081
bp.net_training: test_error: 0.0000000000000000
bp.net_training: sbic: 10000 -1.1906957079849794
bp.net_training: query: 0.008719 0.990678 0.988943 0.007774
bp.net_write: file: xor.mp
bp.net_write: configuration: 2::2:6(6):1
SESSION ENDED   : Wed Aug 30 12:27:08 1995
}}}
```

A.2.9 Netzgewichte abspeichern

`bp -w <Dateiname>` sichert die durch Training gewonnenen Schwellwerte und Gewichte in der Datei `<Dateiname>`, aus der sie mittels `bp -r` wieder geladen werden können.

Beispiel

`$ bp -c 2::2:2(2):1 -t /e=1000/l=2.0/i=0.9 xor.trn -w xor.mp`
legt Platz für ein MP-Netz mit 2 Eingabeneuronen, zwei verborgenen Neuronen und einem Ausgabeneuron an. Anschließend wird es 1000 Epochen mit einer Lernrate von 2,0 und einer Impulsrate von 0,9 mit den Musterpaaren aus der Datei `xor.trn` trainiert. Hiernach werden Topologie, Schwellwerte und Gewichte in der Datei `xor.mp` gespeichert.

Anmerkungen

- Hat `<Dateiname>` den Wert `stdout`, so werden die Daten in die Standardausgabe geschrieben.
- `bp -wa <Dateiname>` schreibt die Schwellwerte und Gewichte im ASCII-Format. Dieses Format benötigt mehr Platz, ist dafür aber architekturunabhängig. `bp -r <Dateiname>` erkennt eigenständig, um welches Format es sich handelt.

A.2.10 bp — parallel

Die parallelen Implementationen von `bp` lassen sich prinzipiell genauso bedienen wie die sequentielle. Aufruf und Optionen sind dieselben, ebenso die Ausgabedateien. Neu ist nur die Art und Weise, wie das parallele Programm unter der jeweiligen Umgebung gestartet wird.

PVM 3.3

Da unter PVM nur das parallele Batch-Training implementiert wurde, führt ein fehlendes `/b` bei den Trainingsoptionen (siehe Abschnitt A.2.6) zu einem Programmabbruch.

Eine parallele virtuelle Maschine wird zum Beispiel durch das Kommando `pvm` etabliert. Danach befindet man sich in der PVM-Shell und kann Änderungen an der Konfiguration vornehmen.

Der Aufruf von `bp` geschieht über den Prefix `mpil` in der Kommandozeile:
`$mpil [<n_tasks>] bp ...`

`mpil` ist ein PVM-Programm, welches `bp` nachlädt. `<n_tasks>` gibt an, wieviele `bp`-Tasks gestartet werden sollen. Fehlt diese Angabe, so nimmt `mpil` stattdessen die Anzahl der an der virtuellen Maschine beteiligten Rechner. Nachdem alle `bp`-Tasks initialisiert wurden, terminiert `mpil`.

`mpil` kann auch außerhalb der PVM-Shell gestartet werden, sofern es sich im Suchpfad befindet. Die PVM-Version von `bp` kann ausschließlich über `mpil` aufgerufen werden und sollte sich nicht im Suchpfad befinden.

Unter PVM ist der jeweilige Heimatkatalog das Arbeitsverzeichnis, selbst wenn das PVM-Programm aus einem anderen Katalog heraus gestartet wurde. Aus diesem Grund sind alle Dateinamen relativ zum Heimatkatalog anzugeben.

Parix

Unter PARIX sind keine Besonderheiten zu beachten. `bp` läßt sich ganz gewöhnlich über das Kommando `run` aufrufen. Zum Beispiel startet

```
$run -f1 2 2 bp.px ...
```

`bp` auf dem PowerXplorer (siehe Tabelle 4.1).

Warnung

> *„Es sei darauf hingewiesen, daß die Benutzung dieses Produktes die Entropie im Universum erhöht. Obwohl es sich hierbei nicht um eine Straftat handelt, muß gesagt werden, daß dieser Prozeß ultimativ zum Hitzetod des Universums führt.“*

Aus: *Journal of Irreproducible Results*

Anhang B

bp — Auf Diskette

Literaturverzeichnis

[1] R. Battiti. **First- and Second-Order Methods for Learning: Between Steepest Descent and Newtons's Method.** *Neural Computation*, Vol. 4, pp. 141–166, MIT 1992.

[2] H. Braun, P. Zagorski. **ENZO–II – a Powerful Design Tool to Evolve Multilayer Feed Forward Networks.** *ICEC–94*, IEEE 1994.

[3] M. Cosnard, J.C. Mignot, H. Paugam-Moisy. **Implementation of Multilayer Neural Networks on Parallel Architectures.** *2nd International Specialist Seminar on The Design and Application of Parallel Digital Processors*, pp. 43–47, IEE 1991.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. **PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.** MIT Press 1994.

[5] I. Glöckner. **Monotonic incrementation of backpropagation networks.** *Proceedings of the International Conference on Artificial Neural Networks (ICANN'93)*, Amsterdam, p. 498, Springer Verlag 1993.

[6] J. Hertz, A. Krogh, R. Palmer. **Introduction to the Theory of Neural Computation.** Addison-Wesley 1991.

[7] J. Hopfield. **Neural Networks and physical systems with emergent collective computational abilities.** *Proceedings of the National Academy of Sciences*, Vol. 79, pp. 2554–2558, 1982.

[8] K. Hornik, M. Stinchcombe, H. White. **Multilayer Feedforward Networks are Universal Approximators.** *Neural Networks*, Vol. 2, pp. 359–366, Pergamon Press 1989.

[9] T. Kohonen **Self-organized formation of topologically correct feature maps.** *Biological Cybernetics*, Vol. 43, pp. 59–69, 1982.

[10] W. McCulloch, W. Pitts. **A logical calculus of the ideas immanent in nervous activity.** *Bulletin of Mathematical Biophysics*, Vol. 9, pp. 115–133, 1943.

[11] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, J. Beer. **The Ring Array Processor: A Multiprocessor Peripheral for Connectionist Applications.** *Journal of Parallel and Distributed Computing 14(3)*, pp. 248–259, Academic Press Inc. 1992.

[12] Message Passing Interface Forum. **MPI: A Message Passing Interface Standard.** University of Tennessee, Knoxville, TN, May 1994.

[13] H. Mühlenbein, K. Wolf. **Neural Network Simulation on Parallel Computers.** In D.J. Evans, G.R. Joubert, F.J. Peters, (Eds.), *Parallel Computing-89*, pp. 365–374, North Holland 1990.

[14] U.A. Müller, P. Kohler, A. Gunzinger. **Neural Net Simulation with MUSIC.** *IEEE International Conference on Neural Networks*, pp. 1737–1741, IEEE 1993.

[15] D. Nychka, S. Ellner, D. McCaffrey, A.R. Gallant. **Finding Chaos in Noisy Systems.** *Journal of the Royal Statistical Society*, Series B. 54, pp. 399–426, Royal Statistical Society 1992.

[16] Parsytec. **Parix V1.3 PowerPC Software Documentation.** 1994

[17] H.-O. Peitgen, H. Jürgens, D. Saupe. **Bausteine des Chaos — Fraktale.** Klett-Cotta/Springer-Verlag 1992.

[18] R. Rojas. **Theorie der neuronalen Netze — Eine systematische Einführung.** Springer Verlag 1993.

[19] F. Rosenblatt. **The perceptron: a probabilistic model for information storage and organisation in the brain.** *Psychological Review*, Vol 65, pp. 386–408, 1958.

[20] D.E. Rumelhart, G.E. Hinton, R.J. Williams. **Learning internal representations by error propagation.** In D.E. Rumelhart and J.L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, pp. 318–362, MIT 1987.

[21] G. Schwarz. **Estimating the Dimension of a Model.** *The Annals of Statistics*, Vol. 6, No. 2, pp. 461–464, 1978.

[22] F. M. Thiesing, U. Middelberg, O. Vornberger. **A Neural Network Approach for Predicting the Sale of Articles in Supermarkets.** *Proceedings of the 3rd European Congress on Intelligent Techniques and Soft Computing (EUFIT '95)*, Vol. 1, pp. 315–320, Aachen 1995.

[23] F. M. Thiesing, U. Middelberg, O. Vornberger. **Parallel Back-Propagation for Sales Prediction on Transputer Systems.** In B.M. Cook, M.R. Jane, P. Nixon, P.M. Welch, (Eds.), *Transputer Applications and Systems '95, Proceedings of the 1995 World Transputer Congress*, pp. 318–331, IOS Press 1995.

[24] V.R. Vemuri, R.D. Rogers. **Artificial Neural Networks — Forecasting Time Series.** IEEE Computer Society Press 5120-05, 1994.

[25] H. White. **Connectionist Nonparametric Regression: Multilayer Feedforward Networks Can Learn Arbitrary Mappings.** *Neural Networks*, Vol. 3, pp. 535–549, Pergamon Press 1990.

[26] B. Widrow, D.E. Rumelhart, M.A. Lehr. **Neural Networks: Applications in Industry, Business and Science.** *CACM*, Vol. 37, No. 3, pp. 93–105, 1994.

[27] M. Wynne-Jones. **Node Splitting: A Constructive Algorithm for Feed-Forward Neural Networks.** In J.E. Moody, S.J. Hanson and R.P. Lippmann (Eds.), *Neural Information Processing Systems 4*, pp. 1072–1079, Morgan Kaufmann Publishers 1992.

[28] H. Yoon, J.H. Nang, S.R. Maeng. **A distributed backpropagation algorithm of neural networks on distributed-memory multiprocessors.** *Proceedings of the 3rd symposium on the Frontiers of Massively Parallel Computation*, pp. 358–363, IEEE 1990.