Einsatz neuronaler Netze zur Lastbalancierung in Workstationclustern

B. Schnor · H. Langendörfer · S. Petri*
Institut für Betriebssysteme und Rechnerverbund
TU Braunschweig

Veröffentlicht in: H. Langendörfer (Herausgeber). *Praxisorientierte Parallelverarbeitung – Beiträge zum 3. Workshop über Wissenschaftliches Rechnen*, Braunschweig, Oktober 1994. Hanser Verlag. S. 154–165.

1 Einleitung

Workstation-Cluster werden häufig benutzt, um die Benutzer mit interaktiven Arbeitsplätzen zu versorgen. Da solche Installationen typischerweise nur eine geringe durchschnittliche CPU-Auslastung aufweisen [8], liegt es nahe, rechenintensive Prozesse auf zur Zeit unbelastete Maschinen auszulagern, um deren Ausführungszeiten zu minimieren. Aus der Literatur sind verschiedene Systeme bekannt, die dieses Auslagern möglichst transparent gestalten sollen [5, 12]. In konfigurationsheterogenen Umgebungen ergibt sich das Problem, daß für die Auslagerungsentscheidungen außer der Belastung der Rechner auch deren unterschiedlichen Grundgeschwindigkeiten berücksichtigt werden müssen. Da es sich bei den von uns erprobten Systemen als schwierig erwiesen hat, diesen Faktor zu bestimmen, haben wir ihn mit Hilfe neuronaler Netze lernen lassen.

In den folgenden Abschnitten skizzieren wir zunächst den grundsätzlichen Aufbau eines Lastverteilungssystems, und beschreiben den von uns verfolgten Ansatz für den Einsatz neuronaler Netze, sowie die damit erzielten Ergebnisse.

2 Lastverteilungssysteme

Ein Lastverteilungssystem besteht aus den drei Komponenten

- Informationsmanagement,
- Lastverteilungsstrategie und
- Export.

Wir erläutern diese Komponenten am Beispiel des am Institut für Betriebssysteme und Rechnerverbund der TU Braunschweig entwickelten YALB (Yet Another Load Balancing System) [10].

*Wir danken Henrik Carlsson, der unsere Ideen praktisch ausprobiert hat [1].

Das Informationsmanagement bestimmt aus den Ressource-Statistiken einen Lastindex als Maß für die momentane Auslastung eines Rechners und stellt diesen Lastindex der Lastverteilungsstrategiekomponente zur Verfügung. Die Lastverteilungsstrategiekomponente trifft aufgrund der Lastindizes die Lastverteilungsentscheidungen und bestimmt für neu eintreffende Aufträge einen geeignet erscheinenden Ausführungsort. Die Exportkomponente schließlich übernimmt den Datentransfer. Hierzu gehören neben Erzeugen und Starten des Prozesses auf dem ausgewählten Rechenknoten auch die Übertragung von Nachrichten an exportierte Prozesse, sowie die Übertragung von Ergebnissen zurück an den Aufrufer.

Ein Vergleich verschiedener Lastverteilungsstrategien findet sich in [4]. Die weiter unten zitierten Messungen wurden mit der sogenannten TopHost–Strategie durchgeführt, die einen Job auf den aktuell schnellsten Rechner exportiert. Als schnellster Rechner wird derjenige mit dem geringsten Lastindex angesehen. Das YALB benutzt den folgenden Lastindex:

$$\mathrm{I} := \alpha \cdot (1 + \mathit{load}),$$

wobei *load* die aktuelle Rechnerlast angibt und α ein Architekturfaktor für die Grundgeschwindigkeit des Rechners ist. Als für unterschiedliche Architekturen verläßliches Maß für die Last eines Rechners hat sich die Länge der CPU-Warteschlange, d.h. die Anzahl der lauffähigen Prozesse, herausgestellt [10]. Die Länge der CPU-Warteschlange wird alle 5 s bestimmt. Das YALB benutzt für *load* den über 30 s exponentiell geglätteten Wert.

Definition des Architekturfaktors

Da das YALB für konfigurationsheterogene Umgebungen entwickelt wurde, mußten die unterschiedlichen Grundgeschwindigkeiten der Rechner berücksichtigt werden. Ein mögliches Leistungsmaß sind z.B. die MIPS-Angaben der Hersteller. Doch angesichts der bekannten Schwächen bei der MIPS-Bestimmung haben wir uns zu eigenen Benchmarks entschlossen. Als Benchmark dient ein LaTeX-Lauf, eine in der Institutsumgebung häufige, größtenteils CPU-intensive Anwendung. Die Übersetzung eines Programmpakets mit Make und C-Compiler zeigte ein fast identisches Verhalten. Daher beschränken wir uns in den folgenden Untersuchungen auf die LaTeX-Anwendung. Der Architekturfaktor α^r eines Rechners r ist ein relatives Maß:

$$\alpha^r \quad = \quad \frac{\text{Laufzeit auf dem unbelasteten Rechner}}{\text{Laufzeit auf dem Referenzrechner}}.$$

Als Referenzrechner dient eine unbelastete Sun SLC. Die Laufzeit des LaTeX-Programms auf dem unbelasteten Referenzrechner beträgt 92 s. Da die SLC der langsamste Rechnertyp unter den betrachteten Maschinen ist, liegen alle Architekturfaktoren zwischen 1 und 0. Ein $\alpha^r = 0.5$ bedeutet also, daß der Rechner r doppelt so schnell wie der Referenzrechner ist.

Außer der Prozessorgeschwindigkeit sind auch Speicherausbau, Plattenausstattung und Fileservertyp weitere die Geschwindigkeit bestimmende Faktoren. So handelt es sich bei der Referenzmaschine um eine SLC mit 8 MB Hauptspeicher. Auf einer SLC mit 16 MB Hauptspeicher beträgt die mittlere Laufzeit 89.5 s. Dies entspricht einem α-Wert von 0.973. Tabelle 1 gibt einen Überblick über die in den folgenden Untersuchungen benutzten Rechner, ihren Speicherausbau, ihre Server-Aufgaben und die zugehörigen α-Werte. Von Rechnern der gleichen Architektur, mit gleichem Speicherausbau und gleicher Konfiguration (z.B. identischer Fileserver) wird angenommen, daß sie den selben α-Wert besitzen.

Wie sehr der α-Wert die Performance des YALBs beeinflußt, wurde durch Vergleichsmessungen untersucht. Hierzu werden auf den SLCs Skripte gestartet, die Jobs enthalten, die vom YALB auf die Rechner des Systems verteilt werden sollen. Die folgenden Speedup-Werte sind

Typ	Speicher	Festplatte intern	Festplatte extern	Aufgabe	α^r
$SS2_1$	40 MB	200 MB	1 GB	Fileserver (TeX, X11, ...), Swapserver	0.531
$SS2_2$	32 MB	200 MB	1 GB	Fileserver (Homedirectory)	0.518
ELC_1	16 MB				0.672
ELC_2	16 MB				0.674
IPC_1	24 MB	200 MB		Swapserver	0.850
IPC_2	36 MB	200 MB	1 GB	Swapserver	0.841
IPC_3	24 MB	200 MB	1 GB	Swapserver	0.850
SLC_1	16 MB				0.973
SLC_2	16 MB				0.973
SLC_3	16 MB				0.973
SLC_4	16 MB				0.973

Tabelle 1: Die Konfiguration der Testumgebung

Durchschnittswerte aus jeweils sechs Messungen mit Variationskoeffizienten kleiner als 4.5%. Gegenüber der lokalen Ausführung auf den SLCs erreicht man mit dem YALB um ca. 13% kürzere Ausführungszeiten. Werden die α-Werte gleichmäßig verfälscht, z.B. alle um 10% erhöht, so wird kein Performance-Verlust beobachtet. Dieses Ergebnis war zu erwarten, da der YALB-Index ein relatives Maß ist. Werden die α-Werte hingegen zufällig um 10% erhöht oder erniedrigt, wird ein Speedup zwischen 8.1% und 10.8% beobachtet. Bei Veränderungen um 20% lag der mit dem YALB erreichte Speedup nur noch zwischen 5.3% und 6.7%. Bei geringen Veränderungen um 5% lag der Speedup zwischen 11% und 12.8% und somit nahe dem besten erzielten Speedup.

Beobachtung: Die Performance des YALBs verschlechtert sich nur unwesentlich solange die α-Werte auf 5% genau bestimmt sind.

Im Fall von Änderungen im System, wie z.B. nach einem Fileserver-Ausfall oder Plattencrash, ist auch eine Änderung der α-Werte zu erwarten und somit eine Neuberechnung erforderlich. Dies gilt ebenfalls für jeden neuen Rechnertyp, der zum System neu hinzukommt. Mit dem hier vorgestellten Ansatz der Backpropagation-Netze liegt ein einfach einzusetzendes Werkzeug vor, daß einem Systemadministrator die Neukonfigurierung des Lastverteilungssystems im laufenden Betrieb erlaubt.

3 Modellbeschreibung

Zur Bestimmung des Architekturfaktors α sind Benchmarkläufe auf dem ansonsten unbelasteten System notwendig. Will man im Normalbetrieb Messungen durchführen, ohne das System exklusiv zur Verfügung zu haben, muß der Einfluß der durch Mitbenutzer erzeugten Last berücksichtigt werden. Eine erste Definition der Funktion $arch()$ lautet daher:

$$arch : (last, laufzeit_{Benchmark}) \mapsto \alpha^r \in \mathbf{R}^+.$$

Die wichtigsten Parameter, die die Last eines Rechners bestimmen, sind die Ausnutzungsgrade der vier Ressourcen CPU, Hauptspeicher, Festplatte und Netzwerk. Da die Ethernetauslastung des gegebenen Systems durchschnittlich nur 10% ist, wird die Netzlast im folgenden nicht berücksichtigt. Die Funktion $arch()$ wird daher wie folgt definiert:

Parameter	Definition
$runqueue_{CPU}$	Der Mittelwert der Länge der CPU-Runqueue während der Laufzeit des Benchmarkprogramms.
$swap_{in}$	Der Mittelwert der Anzahl der Seiten der eingelagerten Prozesse während der Laufzeit des Benchmarkprogramms.
$swap_{out}$	Der Mittelwert der Anzahl der Seiten der ausgelagerten Prozesse während der Laufzeit des Benchmarkprogramms.
$avenrun_{lokal}$	Der UNIX-Avenrun-Index für den Local-Server am Ende der Ausführung des Benchmarkprogramms.
$avenrun_{swap}$	Der UNIX-Avenrun-Index für den Swapserver am Ende der Ausführung des Benchmarkprogramms.
$avenrun_{file}$	Der UNIX-Avenrun-Index für den Fileserver am Ende der Ausführung des Benchmarkprogramms.
$syszeit$	Der im System-Modus verbrachte Anteil der Zeit während der Laufzeit des Benchmarkprogramms.
$laufzeit$	Die Laufzeit des Benchmarkprogramms.

Tabelle 2: Definition der Parameter

$$arch : (last_{CPU}, last_{Speicher}, last_{Festplatte}, laufzeit_{Benchmark}) \mapsto \alpha^r \in \mathbf{R}^+.$$

Zur Beschreibung der Lastparameter werden die vom UNIX Kern kontinuierlich erstellten Ressourcenstatistiken über die Auslastung des Systems verwendet. Zur Lastbewertung entfernter Rechner wird der UNIX-Avenrun-Index benutzt, der mit Hilfe der Funktion `rstat(3)` [11] ermittelt wird. Tabelle 2 zeigt die zur Lastbeschreibung benutzten Parameter.

Die Laufzeit des Benchmarkprogramms wird von der vorhandenen Systemlast beeinflußt. Zur Beschreibung der Systemlast wurde die prozentuale Angabe der im Systemmodus verbrachten CPU-Zeit (*syszeit*) verwandt. Dieser Parameter ist aber nicht unbedingt für verschiedene Architekturen vergleichbar, wie sich bei der Entwicklung des YALB-Systems [10] gezeigt hat. Daher wird das Modell sowohl mit als auch ohne diesen Parameter definiert.

Da die Länge der Plattenwarteschlange im Fall von Diskless-Client-Maschinen immer Null ist, konnte dieser Parameter nicht sinnvoll benutzt werden. Daher konnte die Aktivität des Dateisystems nicht berücksichtigt werden. Zur Bewertung der Hauptspeicherauslastung wurde die mittlere Anzahl der Speicherseiten der pro Zeiteinheit ein- bzw. ausgelagerten Prozesse ($swap_{in}$ bzw. $swap_{out}$) während der Laufzeit des Benchmark-Programms benutzt.

Während des Programmlaufs sind maximal drei Fileserver involviert:

1. Der Server, der die Programmbinaries hält,
2. der Server, der das Verzeichnis, in dem das Benchmark-Programm Ein- und Ausgaben macht, hält und ggf.
3. der Swapserver, falls Speicherplatz knapp wird.

Die Last dieser Server wird mit dem UNIX-Avenrun-Index approximiert ($avenrun_{lokal}$, $avenrun_{file}$, $avenrun_{swap}$).

Rechner	Anzahl der Muster
$SS2_1$	94
$SS2_2$	76
ELC_1	70
ELC_2	81
IPC_1	92
IPC_2	89
IPC_3	64
SLC_1	95
SLC_2	66
SLC_3	97
SLC_4	73

Tabelle 3: Die Anzahl ausgewählter Muster jedes Rechners

Als Benchmark-Programm wurde ein LaTeX-Lauf ausgewählt. Die erzeugte `dvi`-Datei ist ca. 330 KB groß. Während des Programmlaufs werden ca. 3,5 MB Speicher benötigt.

Die endgültige Definition der Funktion $arch()$ sieht somit folgendermaßen aus:

$$\begin{aligned} arch \quad : \quad & (runqueue_{CPU}, swap_{in}, swap_{out}, avenrun_{lokal}, avenrun_{swap}, \\ & avenrun_{file}, [syszeit,]\ laufzeit_{\text{\LaTeX}}) \mapsto \alpha^r \in \mathbf{R}^+. \end{aligned}$$

4 Bestimmen des Architekturfaktors mittels Neuronaler Netze

In diesem Abschnitt werden zwei Ansätze zur Bestimmung des Architekturfaktors beschrieben, die beide auf dem Backpropagation-Algorithmus [9] beruhen: die Funktionsannäherung und die Klassifizierung. Zur Simulation der neuronalen Netze wurde das Programmpaket **PlaNet** [7] benutzt.

Für jeden der untersuchten 11 Rechner wurden ungefähr 100 Benchmarkläufe durchgeführt. Um den Einfluß von stark abweichenden Meßwerten zu vermeiden, wurden danach nur diejenigen Messungen, deren Meßwerte in bestimmten Intervallen lagen, ausgewählt (vgl. Tabelle 3).

Die Meßwerte für die in Tabelle 2 angegebenen Parameter bilden die Mustermenge. Aus dieser wird im folgenden stets eine Teilmenge L ausgewählt. Mit den Mustern aus L wird das jeweilige Netz trainiert, getestet wird mit allen Mustern.

4.1 Bestimmen des α mittels Funktionsannäherung

Wie bei der Modellbildung beschrieben, wird der Architekturfaktor als Funktion der Lastparameter und der Laufzeit bestimmt. Mit Hilfe eines neuronalen Netzes soll diese Funktion approximiert werden. Abbildung 1 zeigt das verwendete Backpropagation-Netz. Die Anzahl der Eingabeknoten wird durch die Anzahl der Eingabeparameter vorgegeben. Da das Ergebnis der gesuchte Ausgabewert α ist, besteht die Ausgabeschicht nur aus einem einzelnen Knoten.

Die Anzahl der Knoten im Hidden-Layer wurde gemäß der Faustregel, daß diese Anzahl gleich dem Mittelwert aus der Anzahl der Eingabeknoten und Ausgabeknoten sein soll, gleich fünf

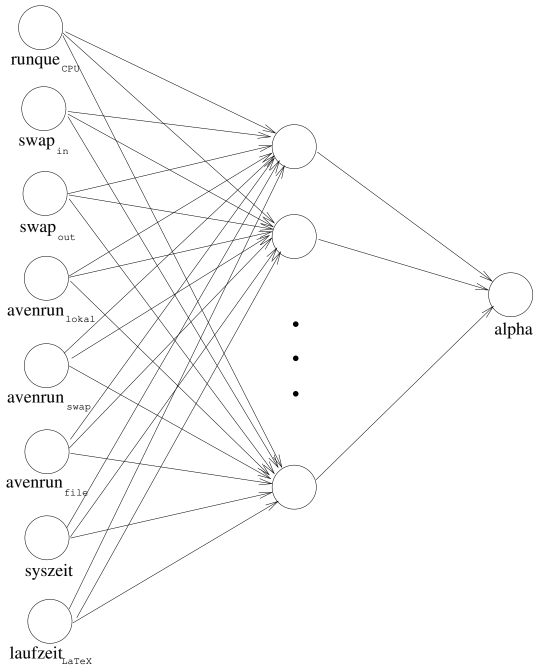

Abbildung 1: Die Struktur des für die Funktionsannäherung verwendeten Netzes

gesetzt. Es zeigte sich aber, daß diese Netze die Architekturfaktoren für unbekannte Rechnerarchitekturen nicht hinreichend genau bestimmen konnten und daß Netze mit zwei Knoten im Hidden-Layer bessere Ergebnisse liefern. Bei allen Trainingsläufen wurde die Lernkonstante η auf 0.01 und der Impulsterm γ auf 0.85 gesetzt. Jedes Muster der Trainingsmenge wurde 40 000 Mal an das Netz angelegt.

Nachdem das Netz trainiert ist, werden die Muster der Testmenge angelegt, um zu überprüfen, wie gut das Netz den Wert α^r approximieren kann. Für jeden Testrechner r werden dafür die zugehörigen Testmuster angelegt. Der Mittelwert aus den so bestimmten α-Werten wird mit $\tilde{\alpha}_f^r$ bezeichnet.

Zuerst wurde untersucht, ob ein Backpropagation-Netz prinzipiell in der Lage ist, die Lastinformationen korrekt zu bewerten und richtig auf den α-Wert zu schließen. Tabelle 4 zeigt die Ergebnisse, wobei die Rechner, deren Muster zur Trainingsmenge gehören, in Fettdruck geschrieben sind. Die von $Netz_1^f$ bestimmten Architekturfakoren weichen zwischen -2.36% und $+6.38\%$ von den richtigen Werten ab.

Weiterhin wurde untersucht, wie gut ein Netz den Architekturfaktor für eine Rechnerarchitektur, die nicht in der Trainingsmenge enthalten ist, approximiert. Aus der Testmenge L wurden hierfür die Muster der ELCs herausgenommen. Die Testergebnisse für $Netz_2^f$ in Tabelle 4 zeigen, daß $\tilde{\alpha}_f^{ELC_1}$ und $\tilde{\alpha}_f^{ELC_2}$ erheblich vom korrekten Wert abweichen. Die große Abweichung läßt sich durch die Tatsache erklären, daß die ELCs anders auf Systemlast reagieren als die übrigen Architekturen, d.h. die gleichen Werte für den Parameter *syszeit* führen zu unterschiedlichen Laufzeitverzögerungen. Die ELCs haben im Verhältnis zu den übrigen Architekturen kleinere Werte für den Parameter *syszeit*. Das Netz faßt diese Tatsache als eine kleinere Last auf, wodurch die ELCs langsamer zu sein scheinen, sie erhalten also zu große Architekturfaktoren. Die Abweichungen liegen in dem Intervall $[-3.87\%, +13.9\%]$.

Im Unterschied zu $Netz_2^f$ wurde das $Netz_3^f$ ohne den Parameter *syszeit* trainiert. $Netz_3^f$ bestimmt die Architekturfaktoren für die beiden ELCs deutlich besser als $Netz_2^f$. Die Fehler sind jedoch

		$Netz_1^f$		$Netz_2^f$		$Netz_3^f$	
Rechner	α^r	$\tilde{\alpha}_f^r$	Abw. [%]	$\tilde{\alpha}_f^r$	Abw. [%]	$\tilde{\alpha}_f^r$	Abw. [%]
$\mathbf{SS2}_1$	0.531	0.536	0.94	0.542	2.07	0.555	3.58
$\mathrm{SS2}_2$	0.518	0.549	5.98	0.551	6.37	0.558	7.72
$\mathbf{ELC}_1$	0.672	0.698	3.87	0.756	12.5	0.716	6.55
ELC_2	0.674	0.717	6.38	0.768	13.9	0.728	8.01
$\mathbf{IPC}_1$	0.850	0.834	-1.88	0.849	-0.12	0.851	0.12
IPC_2	0.841	0.828	-1.30	0.841	0.00	0.819	-2.62
$\mathbf{IPC}_3$	0.850	0.840	-1.18	0.842	-0.80	0.857	0.82
SLC_1	0.973	0.970	-0.31	0.964	-0.92	0.961	-1.23
SLC_2	0.973	0.950	-2.36	0.951	-2.26	0.929	-4.52
SLC_3	0.973	0.947	-2.60	0.951	-2.26	0.936	-3.80
$\mathbf{SLC}_4$	0.973	0.957	-1.64	0.964	-0.92	0.945	-2.88

Tabelle 4: Ergebnisse mit $Netz_1^f$, $Netz_2^f$ und $Netz_3^f$

größer als die gewünschten ±5 Prozent. Die Abweichungen liegen für $Netz_3^f$ in dem Intervall $[-4.52\%, +8.01\%]$.

Zusätzlich wurde eine SparcStation 10 (SS10) getestet. Dazu wurde $Netz_4^f$ mit Mustern je zweier SS2s, IPCs, ELCs und SLCs trainiert. Für die SS10 wurden 20 Testmuster bestimmt. Die Ergebnisse sind in den mit „ohne fiktive Muster“ bezeichneten Spalten in Tabelle 5 angegeben. Durch die Verwendung einer großen Trainingsmenge wurden die Ergebnisse für alle bekannten Architekturen verbessert. Der Architekturfaktor für die neue, wesentlich schnellere SS10 Architektur wird hingegen falsch bestimmt.

Damit das Netz auch schnellere Rechner zu erkennen lernt, wurden fiktive Muster erzeugt. Ausgehend von den vorhandenen Testmustern wurden neue Muster gemäß dem im Abschnitt 4.2 beschriebenen Verfahren erzeugt. Das Ergebnis bei der Benutzung fiktiver Muster wird für die SS10 deutlich besser. Insgesamt verschlechtert sich aber die Approximation der Architekturfaktoren, da die Abweichungen zwischen -4.86% und 12.8% liegen.

4.2 Klassifizierungsansatz

In diesem Abschnitt werden Netze benutzt, die in der Ausgabeschicht n Knoten besitzen, wobei jeder der Knoten einer Klasse von Architekturfaktoren entspricht (vgl. Abb. 2). Die Klasseneinteilung zeigt Tabelle 6. Sie wurde so gewählt, daß man einen Repräsentanten $\alpha_{i,rep}$ finden kann, der um weniger als $\delta = 5\%$ Prozent von der oberen ($\alpha_{i,max}$) und der unteren ($\alpha_{i,min}$) Klassengrenze abweicht. Dieser Ansatz kann nicht für beliebig kleine Architekturfaktoren eingesetzt werden, da man sonst unendlich kleine und unendlich viele Stufen erhalten würde.

Tabelle 6 zeigt, daß sich die Rechner nicht gleichmäßig über die Klassen verteilen. Daher wurden ca. 1000 neue Muster für 17 fiktive Rechner erzeugt. Neue Muster werden für jede Klasse $i \in \{1, 2, \cdots 16, 17\}$, jeden Rechner r und jedes Muster m des Rechners r hergestellt. Das neue Muster erhält dabei den Architekturfaktor $\alpha = \alpha_{i,rep}$ und die Laufzeit

$$laufzeit = \frac{laufzeit_m^r * \alpha_{i,rep}}{\alpha^r},$$

wobei $laufzeit_m^r$ die gemessene Laufzeit aus Muster m bezeichnet.

Rechner	α^r	ohne fiktive Muster		mit fiktiven Mustern	
		$\tilde{\alpha}^r_f$	Abw. [%]	$\tilde{\alpha}^r_f$	Abw. [%]
SS10	0.273	0.544	99.3	0.308	12.8
SS2$_1$	0.531	0.526	-0.94	0.517	-2.64
SS2$_2$	0.518	0.526	1.54	0.511	-1.35
ELC$_1$	0.672	0.672	0.00	0.708	5.36
ELC$_2$	0.674	0.681	1,04	0.714	5.93
IPC$_1$	0.850	0.832	-2.12	0.824	-3.06
IPC$_2$	0.841	0.821	-2.38	0.800	-4.86
IPC$_3$	0.850	0.854	0.47	0.822	-3.29
SLC$_1$	0.973	0.970	-0.31	0.964	-0.92
SLC$_2$	0.973	0.967	-0.62	0.948	-2.57
SLC$_3$	0.973	0.957	-1.64	0.938	-3.60
SLC$_4$	0.973	0.972	-0.10	0.950	-2.36

Tabelle 5: Ergebnisse mit $Netz_4^f$

Diese Muster wurden an die Netze 20 000 Mal angelegt. Dabei wurde die Lernkonstante $\eta = 0.005$ und der Impulsterm $\gamma = 0.85$ gesetzt.

Für jedes Testmuster, das an das Netz angelegt wird, erfolgt eine Ausgabe in einer der 17 Klassen. Nachdem alle Testmuster eines Rechners angelegt wurden, wird der Architekturfaktor bestimmt. Eine Möglichkeit ist, denjenigen Repräsentanten zu wählen, der zu der Klasse mit den meisten Ausgaben gehört. Doch der Vergleich zeigt, daß das gewichtete Mittel $\tilde{\alpha}^r_g$ aus den beiden Repräsentanten $\alpha_{i,rep}$, die zu den beiden Klassen mit den meisten Ausgaben gehören, den α-Wert besser approximiert. Dies trifft speziell für Architekturfaktoren zu, die nahe einer Klassengrenze liegen.

Wie auch beim Funktionsannäherungsansatz wurden verschiedene Trainingsmengen untersucht. Für bekannte Architekturen liegen die Abweichungen beim Klassifizierungsansatz im Intervall $[-4.35\%, 3.41\%]$, wobei die IPCs als etwas zu langsam und die ELCs als etwas zu schnell klassifiziert wurden.

Genauso wie für den Funktionsannäherungsansatz wurde auch hier untersucht, ob das Netz den Architekturfaktor für unbekannte Architekturen zuverlässig bestimmen kann. Als unbekannte Architekturen wurden ebenfalls die ELCs bzw. die SS10 getestet. Das Netz $Netz_2^k$ wurde mit derselben Trainingsmenge trainiert wie $Netz_2^f$, d.h. mit Mustern einer SS2, einer SLC und zweier IPCs. Dieses Netz kann alle bekannten Architekturen gut klassifizieren (Abweichungen maximal 3%). Die Architekturwerte für die beiden ELCs weichen um maximal 8.31% ab.

Trainiert man das Netz ohne den Parameter *syszeit*, so verbessert sich das Ergebnis wie beim Funktionsannäherungsansatz. Die Abweichungen für die ELCs verringern sich auf 3.87% bzw. 6.23%. Dieses Netz bestimmt den Architekturfaktor für die SS10 zu 0.264, was um nur 3.29% vom exakten Wert abweicht. Dieses Ergebnis ist deutlich besser als der beim Funktionsannäherungsansatz bestimmte α-Wert.

Bestimmung des Architekturfaktor ohne Zugang zum Kernel

Abschließend wurde untersucht, ob statt der selber aus dem Systemkern gelesenen Statistiken die Ausgaben der `rstat()`-Funktion genügen. Dies wäre zum Beispiel auf Maschinen wie den

Klasse i	$\alpha_{i,min}$	$\alpha_{i,max}$	$\alpha_{i,rep}$	Rechner
1	0.959	1.060	1.007	SLC
2	0.868	0.959	0.911	
3	0.785	0.868	0.824	IPC
4	0.710	0.785	0.746	
5	0.643	0.710	0.675	ELC
6	0.581	0.643	0.611	
7	0.526	0.581	0.552	$SS2_1$
8	0.476	0.526	0.500	$SS2_2$
9	0.431	0.476	0.452	
10	0.390	0.431	0.409	
11	0.353	0.390	0.370	
12	0.319	0.352	0.335	
13	0.289	0.319	0.303	
14	0.261	0.289	0.274	SS10
15	0.236	0.261	0.248	
16	0.214	0.236	0.224	
17	0.193	0.214	0.203	

Tabelle 6: Klasseneinteilung

HP Workstations hilfreich, die keine Funktionen zum Lesen der Kernelstatistiken zur Verfügung stellen. Anstatt der Parameter $runque_{CPU}$, $syszeit$, $swap_{ein}$ und $swap_{aus}$ wurde der Avenrun-Wert des lokalen Rechners verwandt.

Das Netz wurde mit Mustern der SLCs, IPCs und SS2s trainiert. Das Netz konnte die α-Werte für die ihm unbekannten ELCs erstaunlich genau mit einer Abweichung von nur -2.38% bzw. 1.19% bestimmen. Die Abweichung für die unbekannte SS10 beträgt hingegen -12.1%. Bekannte Architekturen wurden mit der vorgegebenen Genauigkeit bestimmt, sofern die Rechner keine Sonderfunktionen besaßen. Der α-Wert einer IPC, die als ftp-Server diente und deren α-Wert bisher zufriedenstellend bestimmt wurde – im besten Fall mit einer Abweichung von nur 0.71% – besaß nun eine Abweichung von $12,7\%$.

5 Andere Arbeiten auf dem Gebiet

In [2] werden die Ergebnisse einer Simulationsstudie beschrieben, in der Schlafintervalle und Schwellwerte der verschiedenen Komponenten eines Lastverteilungssystems durch neuronale Netze für die jeweils aktuelle Lastsituation adaptiert werden. Die Ergebnisse zeigen, daß die Parameter der untersuchten Lastverteilungsverfahren ihrem Optimum gut angenähert werden können.

Einen umfassenderen Ansatz verfolgt Mehra [6]. Er läßt die Abhängigkeit des Lastindex von den Ressource-Statistiken durch neuronale Netze lernen. Um die Trainingsmuster zu erzeugen, werden Benchmarks in verschiedenen Lastsituationen auf den beteiligten Rechnern durchgeführt. Zur deterministischen Erzeugung der Hintergrundlasten wird ein in den Systemkern integrierter Dynamic Workload Generator benutzt. Mehra simuliert preemptive Lastverteilung mit Prozeßmigration. Seine Ergebnisse sind ermutigend, aber das Verfahren ist für den Einsatz in Produktionsumgebungen unbrauchbar: Abgesehen von den Änderungen im Systemkern werden

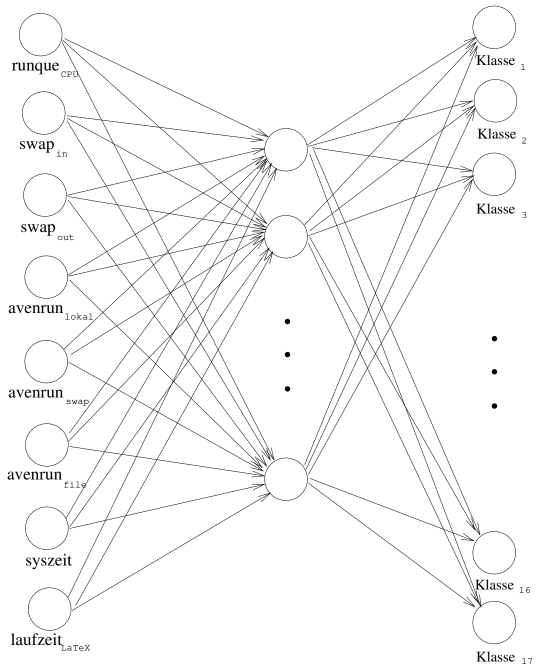

Abbildung 2: Die Struktur des für die Klassifizierung verwendeten Netzes

die Maschinen bei größeren Konfigurationsumstellungen (insbesondere Betriebssystem-Upgrade) zehn Tage lang exklusiv für die Erzeugung der Testmuster benötigt.

6 Zusammenfassung und Ausblick

Die im vorherigen Abschnitt beschriebenen Ergebnisse zeigen, daß neuronale Netze erfolgreich zur Bestimmung von Architekturfaktoren eingesetzt werden können. Hierzu genügen einfache Backpropagation-Netze.

Anstatt für jede Architektur ein eigenes architekturspezifisches Netz zu trainieren, wurde der Ansatz "Ein Netz für alle" untersucht. Es zeigte sich, daß die Wahl des Netzmodells und der Eingabeparameter großen Einfluß auf die Güte der Approximation hat. So hat sich das Klassifizierungsmodell, das mehr Differenzierungsmöglichgkeiten beinhaltet als der Funktionsannäherungsansatz, als das bessere Modell herausgestellt. Die beiden Ansätze können die Architekturfaktoren für bekannte Architekturen bis auf $\pm 5\%$ genau bestimmen. Aber im Vergleich zur Funktionsannäherung bestimmt der Klassifizierungsansatz die Architekturfaktoren für unbekannte Rechner besser.

Gleichbedeutend damit, daß das Netz einen Architekturfaktor gut approximiert, ist, daß das Netz gelernt hat, von den angelegten Eingabeparametern auf die aktuelle Belastung des Systems zu schließen. Das beweist, daß die angelegten Eingabeparameter die Lastsituation hinreichend gut beschreiben.

Die Anwendung eines Netzes auf eine unbekannte Architektur kann nur dann erfolgreich sein, wenn das Verhältnis von Eingabeparametern und Laufzeit ähnlich ist. Dies haben die Experimente mit dem Parameter *syszeit* gezeigt. Um ein Netz also für möglichst viele Architekturen benutzen zu können, muß man sich auf vergleichbare Parameter beschränken. Ein solcher de facto standardisierter Parameter scheint z.B. die CPU-Warteschlange zu sein. Weitere Standards z.B. für Avenrun-Werte wären wünschenswert.

Ein neuronales Netz gibt nur das wieder, was es früher einmal gelernt hat. Daher sind fiktive Muster hilfreich, die schnellere, bisher noch nicht vorhandene Architekturen beschreiben.

Da ein neuronales Netz lernen kann, die Last eines Rechners aus Statistikwerten der Ressourcenauslastung zu bestimmen, kann es auch einen Lastindex aufstellen, der die aktuelle Auslastung beschreibt. Dieser Ansatz wird derzeit von uns verfolgt. Der Einfluß vieler Parameter, wie z.B. der Belastung des Swapservers, wird von Lastverteilungssystemen im Lastindex meist nur durch eine adhoc-Regel berücksichtigt, ohne den genauen numerischen Zusammenhang zu kennen. Hier können neuronale Netze zu besseren Lastbeschreibungen führen.

Dieser Ansatz soll in einem Lastverteilungssystem benutzt werden, das zudem transparente Migration von kommunizierenden Prozessen unterstützt, um dynamischen Lastausgleich untersuchen zu können.

Literatur

[1] Henrik Carlsson. Konfiguration von Lastbalancierungssystemen. Diplomarbeit, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1994.

[2] Sabine Denecke. Entwurf und Implementierung einer adaptiven Komponente in Objektmigrationssystemen mit Hilfe Neuronaler Netze. Diplomarbeit, Institut für Betriebssysteme und Rechnerverbund, Uni Hildesheim, Februar 1994.

[3] H. Langendörfer. *Leistungsbewertung von Rechensystemen*. Hanser, München, 1992.

[4] H. Langendörfer und B. Schnor. *Verteilte Systeme*. Hanser, München, 1994.

[5] Michael J. Litzkow, Miron Livny und Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, S. 104–111. IEEE, Juni 1988. Available on ftp.cs.wisc.edu.

[6] Pankaj Mehra. *Automated Learning of Load-Balancing Strategies for a Distributed Computer System*. PhD thesis, University of Illinois at Urbana-Champaign, 1993. Available on ftp.ibr.cs.tu-bs.de.

[7] Y. Miyata. *A User's Guide to PlaNet*. University of Colorado, Boulder, 1991.

[8] Matt W. Mutka und Miron Livny. Profiling workstations' available capacity for remote execution. In *Proceedings of the 12-th IFIP WG 7.3 Symposium on Computer Performance*, 1987.

[9] D. Nauck, F. Klawonn und R. Kruse. *Neuronale Netze und Fuzzy-Systeme*. Vieweg, 1994.

[10] Stefan Stille. Lastbalancierung in verteilten Systemen. Diplomarbeit, Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, 1993.

[11] Sun Microsystems. *SunOS Reference Manual*, 1990. Revision A.

[12] Songnian Zhou, Jingwen Wang, Xiaohu Zheng und Pierre Delisle. UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. Bericht CSRI-257, CSRI, University of Toronto, April 1992.