DIPLOMARBEIT

Einsatz neuronaler Netze zum Entwurf einer Regelstrategie

von Bernhard-Christian Frenzel

vorgelegt am Lehrstuhl für Regelungstechnik der Universität Erlangen-Nürnberg

Institutsleiter: Prof. Dr. phil. nat. H. Schlitt

Tag der Annahme: 20.11.1993

Tag der Abgabe: 20.05.1994

INSTITUT UND
LEHRSTUHL FÜR REGELUNGSTECHNIK
UNIVERSITÄT ERLANGEN–NÜRNBERG
VORSTAND: PROF. DR. PHIL. NAT. H. SCHLITT

91058 Erlangen, den 20.11.93
Cauerstraße 7
Tel.: (09131) 85–7127/7130

D I P L O M A R B E I T

für

Herrn cand. el. Bernhard Frenzel

Einsatz neuronaler Netze zum Entwurf einer Regelstrategie

In dieser Arbeit soll untersucht werden, ob es möglich ist, mit neuronalen Netzen einen zeitdiskreten, nichtlinearen Zustandsregler zu entwerfen. Als Grundlage der Betrachtungen dient ein Aufsatz von D. H. Nguyen und B. Widrow (⋆), in dem die Methode der Fehlerrückvermittlung (sog. „Error Backpropagation") zum Finden einer Regelstrategie eingesetzt wird. Das dort beschriebene Verfahren ist darzustellen und anhand einiger Beispiele näher zu untersuchen, um die Möglichkeiten und Grenzen dieser Methode abschätzen zu können.

Es wird ausdrücklich auf die „Richtlinien zur Anfertigung von Studien– und Diplomarbeiten am Institut für Regelungstechnik der Universität Erlangen–Nürnberg" hingewiesen.

(⋆ D. H. Nguyen, B. Widrow, „Neural networks for self-learning control systems", Int. J. Control, 1991, Band 54, S. 1439 ff.)

Datum: Note:

....

(Dipl.–Ing. B. Wagner) (Priv.–Doz. Dr.–Ing. F. Dittrich)

ERKLÄRUNG

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen,

.....

Ratiborerstraße 4
8520 Erlangen

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren hat in den Bereichen der Wissenschaft und Technik die Verwendung von künstlichen neuronalen Systemen einen breiten Einzug gehalten. Die am meisten betroffenen Bereiche waren hierbei die typisch „menschlichen" Bereiche wie die Bildverarbeitung und die Spracherkennung. Wenn man bedenkt, daß der menschliche Körper eine Vielzahl von komplizierten Regelmechanismen beinhaltet, liegt es nahe, künstliche neuronale Systeme auch in technischen Regelaufgaben zu verwenden.

Diesem Gedanken folgend wurde von B. Widrow und D. Nguyen eine Abhandlung über ein Entwurfsverfahren für neuronale Regler für eine im allgemeinen nichtlineare Regelaufgabe verfaßt (siehe [13]). Die Regelaufgabe bestand darin, eine Regelstrecke von zufällig gewählten Anfangszuständen in einen gewünschten Endzustand zu überführen. Wichtig dabei war, daß der Regler diesen Endzustand möglichst genau, das heißt mit minimalem Fehler erreichte. Keine genauen Angaben wurden jedoch darüber gemacht, auf welchem Wege und durch Einnehmen welcher Übergangszustände dieser Wechsel vom Anfangs- in den Endzustand stattfand. Dieser Übergang ist beim Entwurf des Reglers von außen nicht beeinflußbar und ergibt sich durch Verwendung des Entwurfsverfahrens „von selbst".

Die Grundidee des Entwurfsverfahrens besteht nun in folgenden Gedankenansatz:

Zunächst wird mit Hilfe eines Emulators in Form eines neuronalen Netzes die zu regelnde Strecke nachgebildet.
Im Anschluß daran versetzt man diesen Emulator zunächst in einen zufälligen Anfangszustand und bildet einen Regelkreis unter Verwendung eines neuronalen Reglers. Ausgehend von dem zufällig gewählten Anfangszustand wird der Streckenemulator neue Folgezustände einnehmen. Unter Verwendung von Abbruchbedingungen, die im Laufe der Arbeit näher beschrieben werden, wird der Emulator bei einem Zustand angelangen, der als Endzustand festgesetzt wird. Anschließend wird der Fehler aus diesem Endzustand und dem Sollwert gebildet. Unter Verwendung eines Gradientenverfahrens wird dann dieser Fehler zum Einstellen der Reglerparameter benutzt und dient somit zur Minimierung des Fehlers im Endzustand.
Nun wird der Emulator wieder in den gleichen Anfangszustand versetzt und die Prozedur wiederholt. Dieser Vorgang wiederholt sich sooft, bis der Fehler des Endzustandes hinreichend klein ist. Der Regler hat dann den zufällig gewählten Anfangszustand „angelernt".
Dieser Vorgang wird nicht nur für einen, sondern für viele zufällig gewählte

Anfangszustände durchgeführt. Der Regler kann somit eine ganze Menge von Anfangszuständen in den Endwert mit minimiertem Fehler überführen.
Im Anschluß daran wird der Regler mit den derart fest eingestellten Parametern zusammen mit der ursprünglichen Strecke im Regelkreis eingesetzt.

Die Besonderheit ist also, daß die Reglerparameter nicht während der Regelung, also „on-line", adaptiert werden. Die eigentliche Adaption erfolgt vor dem Einsatz des Reglers. Dieser wird also nach dem Anlernvorgang mit unveränderlichen Parametern als statisches Gebilde zur Regelung verwendet.

Die Vorteile bei Verwendung eines neuronalen Reglers in Form eines statischen Gebildes sind, daß

- der Realisierungsaufwand durch die fehlende Adaptionseinrichtung sinkt und
- der Regler schnell und einfach herzustellen ist.

Der Nachteil dieser Vorgehensweise ist, daß

- vor der Adaption des Reglers bereits alle möglichen Streckenzustände, die während der Regelung von der Strecke eingenommen werden können, bekannt sein müssen.

Trotz dieses Nachteils konnte für die meisten der untersuchten Beispiele eine zufriedenstellende Regelung erreicht werden.

2 Übersicht

Die folgende, kurze Übersicht gibt den Aufbau der vorliegenden Arbeit wieder:

Die theoretischen Grundlagen befinden sich im Kapitel 3. Nach Beschreibung des Regelungsproblems in Abschnitt 3.1 werden im Abschnitt 3.2 die benötigten Grundlagen der neuronalen Netze dargestellt. Der Abschnitt 3.3 beschäftigt sich mit der Nachbildung der zu regelnden Strecke in Form eines neuronalen Netzes und der dabei auftretenden Struktur. Der Abschnitt 3.4 behandelt dann das Anlernen des neuronalen Reglers. Der Unterabschnitt 3.4.1 beschreibt das zentrale Verfahren dieser Arbeit.

Die behandelten Beispiele sind zusammen mit den Ergebnissen im Kapitel 4 aufgeführt. Der Abschnitt 4.2 beschäftigt sich mit der Regelung einer linearen Strecke, wobei auch das Auftreten einer Stellsignalbegrenzung mit berücksichtigt wird. Das Kapitel 4.3 hat dann die Regelung eines nichtlinearen Systems 3. Ordnung zum Thema, das zum Abschluß auf eine nichtlineare Strecke 4. Ordnung, einen Sattelschlepper, erweitert wird.
Nach einer Zusammenfassung und Bewertung ist die Realisierung des Verfahrens in Form eines Unterprogrammes in der Programmiersprache C im Anhang näher erläutert. Beschrieben wird sowohl die prinzipielle Struktur des Netzwerksimulators, in den das Unterprogramm eingebaut wurde, als auch die relevanten Erweiterungen.

Die Auswertung der Ergebnisse erfolgte mit Hilfe des Mathematikprogrammes Matlab 4.1 auf einer HP-Apollo-Workstation HP 9000/735 unter Verwendung des Betriebssystems HP-Unix Version 9.02. Auf derselben Maschine wurde auch die Programmentwicklung durchgeführt.

3 Theoretische Grundlagen

3.1 Das Regelungsproblem

Das in dieser Arbeit behandelte Regelungsproblem ist die Forderung, ein lineares oder nichtlineares System innerhalb einer endlichen Zeit unter Nebenbedingungen von einem frei wählbaren Anfangszustand in einen Endzustand zu überführen. Der Anfangszustand sei hierbei der zum Zeitpunkt $t = 0$ vorliegende Systemzustand $\underline{x}_0 = x(t = 0)$, der Endzustand sei der Systemzustand $\underline{x}_e$. Im Zeitkontinuierlichen liege weiterhin das zu regelnde System, die Regelstrecke also, in der allgemeinen Form der Gleichung (1)

$$\dot{\underline{x}}(t) = \tilde{F}(\underline{x}(t), \underline{u}(t)) \tag{1}$$

vor. Im folgenden wird die zugehörige, diskrete Beschreibung (2)

$$\underline{x}(k+1) = F(\underline{x}(k), \underline{u}(k)) \tag{2}$$

verwendet, da diese auch zur Nachbildung des Regelungsproblemes auf einem Rechner verwendet wurde.

Der strukturelle Aufbau zur Lösung des oben genannten Problems zeigt Abbildung 1.

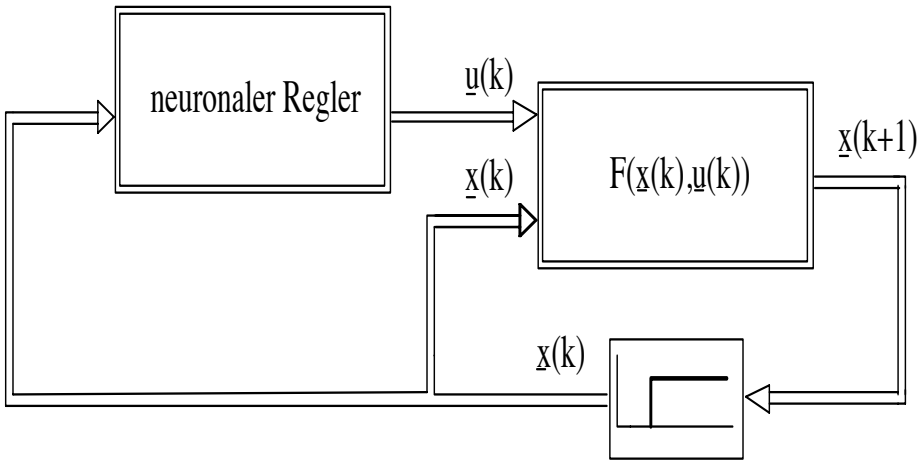

Abbildung 1: Strukturbild des Regelkreises

Wie aus Bild 1 ersichtlich, werden zur Regelung der durch Gleichung (2) modellierten Regelstrecke neuronale Regler verwendet. Prinzipiell ist auch ein anderer Ansatz zur Lösung des Regelungsproblems möglich. So ist beispielsweise das gleiche Problem exemplarisch von S. Kong und B. Kosko durch Einsatz von Fuzzy-Reglern diskutiert worden (siehe [11]). Die weiteren Ausführungen beschränken sich auf die Verwendung neuronaler Regler

und stützen sich maßgeblich auf eine Veröffentlichung von D. Nguyen und B. Widrow [13].

Der hier verwendete Ansatz fällt unter das weite Themengebiet der adaptiven Regelung. Die freien Parameter des neuronalen Reglers werden bei der Adaption mittels eines geeigneten Adaptionsverfahrens derart eingestellt, daß der Regler ein Stellsignal $\underline{u}_k$ liefert, das das Regelungsproblem löst. Das in dieser Arbeit verwendete Adaptionsverfahren ist der sogenannte „Backpropagation-Algorithmus".
Eine Adaption eines neuronalen Netzes wird auch als Anlernvorgang, die verwendeten Adaptionsverfahren auch als Lernalgorithmen bezeichnet.
Ist ein neuronaler Regler einmal angelernt, so wird dieser im folgenden als ein statisches Gebilde mit unveränderbaren Parametern im Regelkreis eingesetzt. Der Vorteil neuronaler Regler besteht nun darin, daß der Regler nicht für sämtliche im Regelkreis auftretenden Anfangszustände $\underline{x}_0$ angelernt werden muß. Es genügt eine bestimmte Anzahl von Anfangszuständen, den sog. Lernmustern, um eine zufriedenstellende Regelung für alle im Regelkreis auftretenden Anfangszustände zu erreichen. Bei Anfangszuständen, die nicht explizit gelernt wurden, aber den Lernmustern ähnlich sind, wird der Regler Stellsignale ähnlich denen der Lernmuster liefern.

Da beim Reglerentwurf sowohl der Regler als auch die Regelstrecke als neuronales Netz dargestellt werden müssen, werden im folgenden Abschnitt zunächst der Aufbau und die Struktur von Neuronen und die hier verwendeten Neuronennetze dargestellt. Außerdem wird der Backpropagation-Algorithmus als verwendetes Adaptionsverfahren erläutert.

3.2 Neuronale Netze

3.2.1 Aufbau eines Neurons

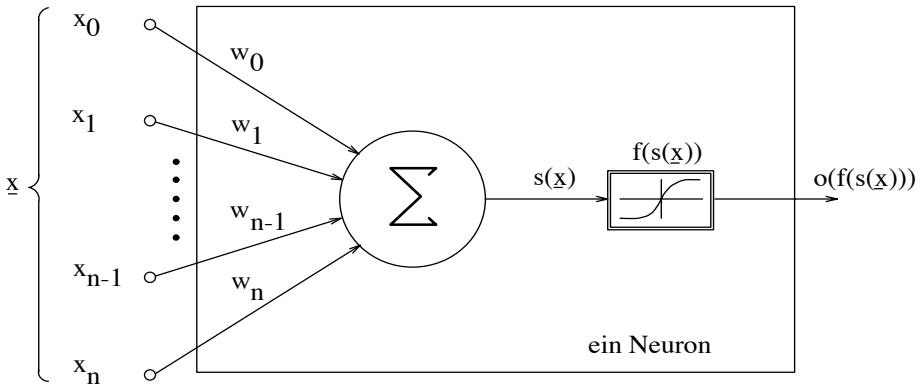

Abbildung 2: innere Struktur eines Neurons

Das kleinste Element eines neuronalen Netzes ist ein einzelnes Neuron. Den strukturellen Aufbau eines Neurons zeigt Abbildung 2, das in dieser Arbeit verwendete, abkürzende Symbol hierfür ist in Abbildung 3 dargestellt.

Die Anordnung aus Bild 2 zeigt ein sehr stark vereinfachtes Modell einer

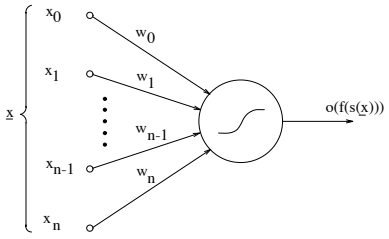

Abbildung 3: abkürzende Schreibweise für ein Neuron

menschlichen Nervenzelle (siehe [14]). Bei einem technisch realisierten Neuron liegen bei der Signalverarbeitung an den $n+1$ Eingängen $n+1$ verschiedene Eingangsgrößen zur Weiterverarbeitung an. Diese Eingangssignale sind entweder die Ausgangssignale anderer Neuronen oder Größen, die an das Neuron herangeführt werden, wie Meßgrößen oder ähnliches. Diese Signale, hier mit $x_0, x_1, \ldots, x_{n-1}, x_n$ bezeichnet, werden mit den zugehörigen Gewichtungen $w_0, w_1, \ldots, w_{n-1}, w_n$ multipliziert und aufsummiert. Das Ausgangssignal $o(f(s(\underline{x})))$, das als Aktivierung bezeichnet wird, ergibt sich dann durch Abbildung einer im allgemeinen nichtlinearen Funktion dieser Summe. Solche Funktionen werden als Aktivierungsfunktionen bezeichnet und sind meist differenzierbare Funktionen, woraus auch die Stetigkeit folgt. Die Forderung der Differentierbarkeit ist Voraussetzung zur Anwendung des

Backpropagation-Algorithmus und somit zwingend notwendig für alle in dieser Arbeit verwendeten Aktivierungsfunktionen.

3.2.2 mehrschichtige neuronale Netze

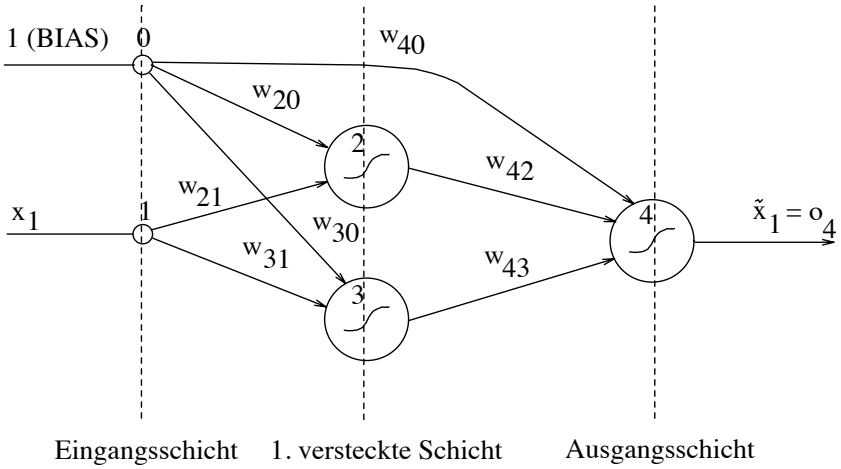

Abbildung 4: 2-schichtiges, neuronales Netz

Um eine Verschaltung mehrerer Neuronen zu einem neuronalen Netz zu strukturieren, werden die Neuronen in Schichten angeordnet. Diese Vorgehensweise dient der Übersichtlichkeit und Vereinfachung von Neuronenstrukturen. Dadurch entfernen sich die technischen Realisierungen neuronaler Netze jedoch wieder weiter von ihrem biologischen Vorbild, den Nervennetzen, die keine solchen geregelten Strukturen aufweisen. Der Aufbau eines einfachen, 2-schichtigen neuronalen Netzes ist in Abbildung 4 dargestellt. Die Eingangsschicht eines Netzes zählt nicht als vollwertige Schicht, da sie nur zur Aufteilung der Eingangssignale für die ersten Neuronen, die Neuronen der 1. versteckten Schicht, dient. Das von außen an das Netz herangeführte Signal ist hier die Größe x_1. Wahlweise kann ein weiterer Eingang, der konstant eins ist, an das Netz angelegt werden. Er wird auch als BIAS-Term bezeichnet. Er wirkt auf alle Neuronen des Netzes und erhöht den Freiheitsgrad, der neben der Struktur des Netzes und den verwendeten Aktivierungsfunktionen maßgeblich durch die Anzahl der Gewichte $\underline{w}$ bestimmt wird.

Neben dem hier vorgestellten, mehrschichtigen Netz gibt es noch eine ganze Reihe anderer Netzstrukturen, wie zum Beispiel die Netzstrukturen nach Elman, Jordan oder das Kaskaden-Korrelationsnetz (siehe [3]). Auf sie wird nicht weiter eingegangen, da das Entwurfsverfahren für neuronale Regler zunächst nur für mehrschichtige Netze der Art nach Abbildung 4 realisiert werden soll.

Die folgenden Überlegungen werden nun beispielhaft am Netz nach Abbildung 4 erläutert. Zielsetzung ist es, eine Abbildung der Art

$$x_{1e} = f(x_1) \tag{3}$$

mit Hilfe der Abbildung eines neuronalen Netzes

$$\tilde{x}_1 = n(x_1, \underline{w}) \tag{4}$$

möglichst gut, d. h. mit minimalem Fehler darzustellen. Der funktionale Zusammenhang nach Gleichung (3) muß nicht explizit gegeben sein. Eine Zahlenfolge von x_1 und die zugehörige Zahlenfolge x_{1e} sind zur Emulation der Abbildung durch das neuronale Netz aus dem interessierenden Zahlenbereich ausreichend. Die Adaption der Gewichte $\underline{w}$ nach Gleichung (4) geschieht nun durch Anwendung eines Lernalgorithmus, hier durch die Verwendung des Backpropagation-Algorithmus. Nach dem Anlernen eines mehrschichtigen, neuronalen Netzes mit Hilfe des Backpropagation-Algorithmus wird dieses auch als Backpropagation-Netz oder abkürzend als BP-Netz bezeichnet.

3.2.3 Der Backpropagation-Algorithmus

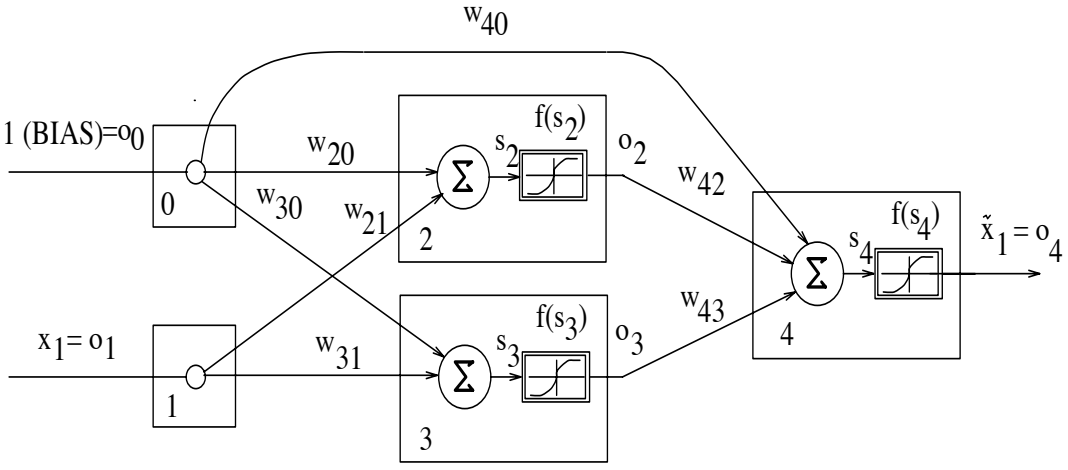

Abbildung 5: 2-schichtiges, neuronales Netz

Der Backpropagation-Algorithmus gliedert sich in zwei Abschnitte, den Feedforward-Schritt und den Backpropagation-Schritt. Der Feedforward-Schritt wird auch als Vorwärtsvermittlung und der Backpropagation-Schritt als Fehlerrückvermittlung bezeichnet.
Um beide Phasen des Algorithmus beispielhaft aufzuzeigen, ist das Netz aus Abbildung 4 noch einmal in ausführlicher Form in Abbildung 5 dargestellt.

Zu Beginn des Lernvorganges werden alle Gewichte $w_{20}, w_{21}, \ldots, w_{43}$ des Netzes zufällig initialisiert.
Zu Beginn des Feedforward-Schrittes wird anschließend an die Eingänge des Netzes, hier also an x_1, ein Eingangswert eines Lernmusters $(x_{1\nu}, x_{1e\nu})$, mit $\nu = 1 \ldots m$ angelegt, wobei m die Anzahl aller anzulernender Muster darstellt. Zunächst wird nur ein beliebiges, aber festes Lernmuster (x_1, x_{1e}) aus den m verschiedenen betrachtet.
Nach Festlegung der Aktivierungsfunktionen des Netzes können die Aktivierungen der Neuronen schrittweise berechnet werden, beginnend von der 1. versteckten Schicht bis hin zur Ausgangsschicht. Im allgemeinen werden sich die Aktivierungen der Neuronen der Ausgangsschicht von den gewünschten Aktivierungen, hier also x_{1e}, unterscheiden. Damit das Netz möglichst gut das System nachbildet, aus dem das Lernmuster ursprünglich gewonnen wurde, kann man den quadratischen Fehler

$$E = \frac{1}{2}(x_{1e} - \tilde{x}_1)^2 \tag{5}$$

minimieren. Die Grundidee des Backpropagation-Algorithmus ist es nun, die Fehlerfunktion E aus Gleichung (5) durch Veränderung der Gewichte des Netzes entlang des Gradienten der Fehlerfunktion E zu minimieren, weshalb der Backpropagation-Algorithmus auch als Gradienten-Abstiegsverfahren bezeichnet wird.

Der Backpropagation-Schritt beginnt daher mit der Berechnung des Gradienten der Fehlerfunktion (5) bezüglich des Ausganges des neuronalen Netzes $\tilde{x}_1$, der sich gerade zu

$$\frac{\partial E}{\partial \tilde{x}_1} = -(x_{1e} - \tilde{x}_1) = -e$$

ergibt. Um nun den Gradienten der Fehlerfunktion E in Abhängigkeit eines Gewichtes w_{ji} zu erhalten, wird die partielle Ableitung des quadratischen Fehlers nach dem Gewicht durch mehrfache Anwendung der Kettenregel der Differentation berechnet. Im Falle des Gewichtes w_{21} aus dem Netz aus Abbildung 5 führt das auf:

$$\frac{\partial E}{\partial w_{21}} = \frac{\partial E}{\partial \tilde{x}_1} \frac{\partial \tilde{x}_1}{\partial s_4} \frac{\partial s_4}{\partial o_2} \frac{\partial o_2}{\partial s_2} \frac{\partial s_2}{\partial w_{21}} \tag{6}$$

Mit den Beziehungen

$$\begin{aligned}
\frac{\partial E}{\partial \tilde{x}_1} &= -e \\
\tilde{x}_1 &= f(s_4) \\
s_4 &= w_{40} + w_{42} o_2 + w_{43} o_3 \\
o_2 &= f(s_2) \\
s_2 &= w_{20} + w_{21} o_1
\end{aligned}$$

ergibt sich somit aus Gleichung (6):

$$\frac{\partial E}{\partial w_{21}} = -e\, f'(s_4)\, w_{42}\, f'(s_2)\, o_1 \tag{7}$$

Hierbei sind die Größen s_2 und s_4 beim vorangegangenen Feedforward-Schritt berechnet worden. Entsprechend Gleichung (7) werden die Fehlergradienten bezüglich der restlichen Gewichte berechnet.

Es werde nun beispielsweise das Gewicht w_{21} proportional zum Fehlergradienten bezüglich w_{21} verändert, d.h. es gelte

$$w_{21neu} = w_{21} + \Delta w_{21} \tag{8}$$

mit

$$\Delta w_{21} = -\eta \frac{\partial E}{\partial w_{21}}. \tag{9}$$

Hierbei ist η ein Proportionalitätsfaktor, der in der Literatur als Lernfaktor bezeichnet wird.
Wenn die Gleichung (9) ein sinnvoller Ansatz zur Veränderung des Gewichtes w_{21} sein soll, so muß sich nach Durchführung des Schrittes (8) die Fehlerfunktion E verkleinert haben. Zur Überprüfung dieser Forderung kann nun der neue Fehler E_{neu} näherungsweise durch eine Taylorreihe dargestellt werden, die nach dem 1. Glied abgebrochen wird:

$$E_{neu} \approx E + \frac{\partial E}{\partial w_{21}} \Delta w_{21} \tag{10}$$

Setzt man nun den Ansatz (9) in Gleichung (10) ein, so folgt die Ungleichung

$$E_{neu} \approx E - \eta \left[\frac{\partial E}{\partial w_{12}} \right]^2 \leq E, \tag{11}$$

d. h. es ist sichergestellt, daß mit Gleichung (9) der Fehler abnimmt oder zumindest gleich bleibt.

Bei m unterschiedlichen Lernmustern werden bei Verallgemeinerung der Gleichung (9) m unterschiedliche Δw_{ji} je zugehörigem Gewicht w_{ji} berechnet. Die Veränderung der Gewichte w_{ji} kann nun nach jedem Lernmuster sofort analog Gleichung (8) erfolgen, oder erst nach Summation aller Gewichtsveränderungen bezüglich aller m Lernmuster. Es können auch Mischformen auftreten. So werden z.B. die Gewichte nach ξ Lernmustern, $\xi \in \{1 \dots m\}$, verändert.
Nach Abarbeitung aller m Lernmuster beginnt der Algorithmus wieder mit dem Feedforward-Schritt. Die Abarbeitung aller m Lernmuster wird als Lernepoche oder auch als Epoche bezeichnet.
Das Verfahren bricht ab, wenn der Fehler unter eine Abbruchschranke ε gesunken ist, d.h. wenn

$$E \leq \varepsilon$$

erfüllt ist.

Die Verallgemeinerung der eben angestellten Überlegungen führt auf die Iterationsgleichungen des Backpropagation-Algorithmus:

$$w_{jineu} = w_{ji} + \Delta w_{ji} = w_{ji} - \eta \frac{\partial E}{\partial w_{ji}}, \tag{12}$$

wobei sich für die Gewichte der Neuronen der Ausgangsschicht

$$\frac{\partial E}{\partial w_{ji}} = \underbrace{\underbrace{-\overbrace{(x_{qe} - \tilde{x}_q)}^{e}}_{\frac{\partial E}{\partial \tilde{x}_q}} f'_j(s_j)}_{\delta_j} o_i = -\delta_j \, o_i \tag{13}$$

ergibt. In Gleichung (13) sind die Ausgangsneuronen gesondert indiziert, wobei das q-te Ausgangsneuron dem j-ten Neuron des Netzes entspricht. Weiterhin ist $\tilde{x}_q$ die Aktivierung des q-ten Ausgangsneurons, x_{qe} der gewünschte Sollwert und $f'_j(s_j)$ die Ableitung der Aktivierungsfunktion des q-ten Ausgangsneurons an der Stelle s_j. Die Aktivierung des i-ten Neurons, dessen Ausgang über das Gewicht w_{ji} mit dem Eingang des q-ten Ausgangsneurons verbunden ist, wird mit o_i bezeichnet.
Für den Fehlergradienten bezüglich eines Gewichtes w_{ji} eines Neurons j, das nicht zur Ausgangsschicht gehört, folgt analog:

$$\frac{\partial E}{\partial w_{ji}} = \underbrace{-(\sum_k \delta_k w_{kj})}_{\frac{\partial E}{\partial o_j}} f'_j(s_j) o_i = -\delta_j o_i \tag{14}$$

Gegenüber Gleichung (13) hat sich der in Klammern stehende Ausdruck verändert. Er entspricht dem zurückpropagierten Gradienten des Fehlers E

bezüglich der Aktivierung o_j. Hierbei ist der Ausgang des j-ten Neurons mit dem Eingang des k-ten Neurons verbunden. Zum Beispiel ergibt sich in Abbildung (5) der Gradient des Fehlers bezüglich o_2 zu:

$$\begin{aligned} \frac{\partial E}{\partial o_2} &= \frac{\partial E}{\partial \tilde{x}_1}\frac{\partial \tilde{x}_1}{\partial s_4}\frac{\partial s_4}{\partial o_2} = -e\, f'(s_4)\, w_{42} \\ &= -(\sum_{k=4}^{4} \delta_k w_{k2}) \end{aligned} \tag{15}$$

wobei die Beziehung

$$\delta_4 = e\, f'(s_4)$$

unmittelbar aus Gleichung (13) folgt. Mit Beziehung (14) ergibt sich aus (15) dann sofort der Ausdruck (7) für den Gradienten des Fehlers bezüglich w_{21}.

Die Beziehungen (13) und (14) unterscheiden sich also nur formal, inhaltlich jedoch werden in beiden Gleichungen die Gradienten des Fehlers bezüglich der Aktivierung des betrachteten Neurons zur Berechnung von $\frac{\partial E}{\partial w_{ji}}$ herangezogen. Mit diesen Gleichungen wird also versucht, ein Minimum des Fehlers gemäß

$$Min(E) = \frac{1}{2} Min(\|\underline{x}_e - \underline{\tilde{x}}\|^2) \tag{16}$$

bezüglich der Variation der Gewichte $\underline{w}$ zu finden. Bei einem System mit n Ausgängen ergeben sich die Größen aus Gleichung (16) dann zu

$$\begin{aligned} \underline{x}_e &= (x_{1e} \quad x_{2e} \quad \ldots \quad x_{ne})^T \qquad \text{und} \\ \underline{\tilde{x}} &= (\tilde{x}_1 \quad \tilde{x}_2 \quad \ldots \quad \tilde{x}_n)^T. \end{aligned}$$

3.2.4 Anmerkungen zum Backpropagation-Algorithmus

Da der Backpropagation-Algorithmus ein Gradienten-Abstiegsverfahren ist, können prinzipiell nur lokale Minima der Fehlerfunktion E gefunden werden. Durch unterschiedliche Startwerte des Verfahrens, das heißt zum Beispiel durch unterschiedliche Initialisierung der Gewichte oder verschiedene Netzwerkstrukturen, werden im allgemeinen verschiedene Fehlerminima approximiert werden. Ein globales Minimum ist jedoch nur dann sicher erreicht, wenn der Fehler für alle Lernmuster identisch Null ist. Zur Überwindung lokaler Minima kann die Verwendung eines Momentum-Terms hilfreich sein. Hierbei wird neben der momentan aktuellen Änderung der Gewichte die vorherige Änderung der Gewichte des letzten Feedforward-Backpropagation-Zyklus mit berücksichtigt. Über die Globalität des gefundenen Minimums kann jedoch auch hier keine Aussage gemacht werden.

Bei Bildung des Fehlers nach Beziehung (16) können die einzelnen Anteile der Ausgangsgrößen unterschiedlich gewichtet werden. Mit $\alpha_\nu \in R, \quad \nu = 1, 2, \ldots, n$ erhält Gleichung (16) dann die Form

$$MIN(E) = \frac{1}{2} Min(\alpha_1(x_{e1} - x_1)^2 + \ldots + \alpha_n(x_{en} - x_n)^2). \qquad (17)$$

In der Praxis werden weiterhin in den Gleichungen (13) und (14) nicht die Werte der Summation s_j des j-ten Neurons verwendet, da der Speicheraufwand sich gegenüber dem einfachen Abspeichern der Aktivierungen o_j verdoppelte. Vielmehr wird f_j' in Abhängigkeit von der Aktivierung o_j berechnet. Zum Beispiel ergibt sich für $f(s) = tanh(s) = o$:

$$f'(s) = \frac{df}{ds} = 1 - o^2$$

oder für die häufig verwendete Sigmoidfunktion $f(s) = \frac{1}{1+e^{-s}} = o$:

$$f'(s) = \frac{df}{ds} = o\,(1 - o)$$

3.3 Die Modellierung der Regelstrecke als neuronales Netz

3.3.1 Einführende Bemerkungen

Um einen neuronalen Regler zur Regelung einer Regelstrecke der Art nach Abbildung 1 zur entwerfen, benötigt man die Regelstrecke selbst in Form eines neuronalen Netzes. Nun sind zwar oft die Systemgleichungen oder zumindest die Systemreaktionen einer Regelstrecke bekannt, aber nur in Spezialfällen wie in Abschnitt 4.2 kann das zugehörige neuronale Netz sofort angegeben werden. In allen anderen Fällen muß ein Netz angelernt werden, das die Regelstrecke nachbildet. Hat man erst einmal dieses Emulatornetz der Strecke angelernt, kann man mit einem dem Backpropagation-Algorithmus verwandten Verfahren,das in Abschnitt 3.4.1 vorgestellt wird, den neuronalen Regler anlernen. Nach dessen Entwurf wird dieser zur Regelung der ursprünglichen Strecke eingesetzt. Daraus ergibt sich, daß die Strecke möglichst gut durch ein neuronales Netz emuliert werden muß, da anderenfalls der Regler zwar den Emulator, nicht jedoch die ursprüngliche Strecke im Sinne des Regelziels vernünftig regeln kann.

Im folgenden werden nun zunächst Überlegungen zum Anlernen eines neuronalen Netzes zur Nachbildung des Systemverhaltens der zu regelnden Regelstrecke angestellt.

3.3.2 Das Anlernen der Strecke ohne Durchgriff

Liegen die Informationen über die Regelstrecke bereits in Form von Systemantworten vor, so können diese zusammen mit dem zugehörigen Eingabevektor direkt als Lernmuster verwendet werden. Liegt die beschreibende Systemabbildung $F(\underline{x}(k), \underline{u}(k))$ nach Gleichung (2)vor, so müssen daraus zunächst repräsentative Vektoren erzeugt werden. Diese sollten möglichst gleichverteilt dem zu interessierenden Bereich der Anfangswerte entnommen werden. Bei der Erzeugung wird der Einfachheit halber angenommen, daß die Systemzustände der Strecke direkt meßbar und rauschfrei sind.

Die Anzahl der Eingänge des Emulatornetzes ist nach Abbildung 1 gleich der Summe aus der Anzahl der Streckenzustände und der Stellsignale. Die Anzahl der Ausgänge ist gleich der Anzahl der Zustände der Regelstrecke. Frei wählbar ist die Anzahl der versteckten Schichten und deren Neuronenzahl. Zur Nachbildung eines Systemverhaltens einer Regelstrecke sind im allgemeinen mehrere Netzstrukturen zulässig, eine Regel hierfür gibt es nicht. Allerdings können bei einer zu geringen Anzahl von Neuronen im Netz bestimmte Problemstellungen nicht mehr zufriedenstellend gelöst werden (siehe [4]).

Nach Erzeugung der Lernmuster kann das Emulatornetz mit Hilfe des Backpropagation-Algorithmus nach Abschnitt 3.2.3 angelernt werden.

3.3.3 Die Verwendung eines Durchgriffes

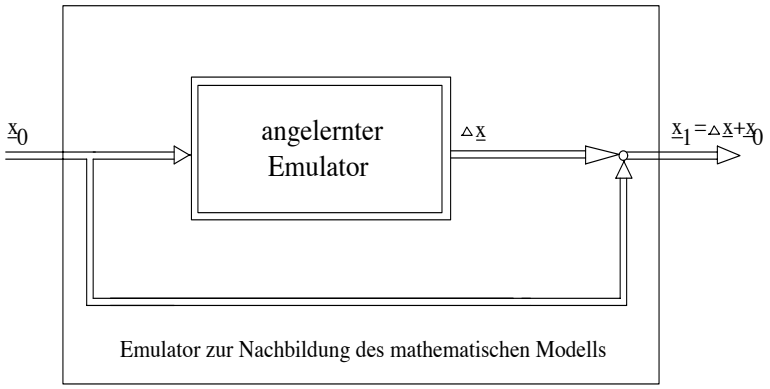

Abbildung 6: Der Emulator bei Verwendung eines Durchgriffes

Beim Anlernen eines Emulators zur Nachbildung des Systemverhaltens der Strecke kann es oft sinnvoll sein, nur die Veränderung $\Delta\underline{x}$ des Systemzustandes $\underline{x}$ anzulernen. Dies ist z.B. der Fall, wenn die Veränderung eines Systemzustandes x_ν mit $\nu \epsilon [1, 2, \ldots n]$ sehr klein gegenüber dem Startwert $x_{\nu 0}$ ist. Hierbei gilt für den zugehörigen Ausgangswert $x_{\nu 1}$ des Emulators

annähernd:

$$x_{\nu 1} \approx x_{\nu 0}$$

Ein Lernverfahren für den neuronalen Emulator wird somit versuchen, die Emulatorgewichte derart einzustellen, daß der Ausgangszustand $x_{\nu 1}$ näherungsweise dem Eingangszustand $x_{\nu 0}$ entspricht. Die eigentliche Information über die Abbildung des Systemzustandes durch den Emulator geht hierbei verloren oder ist stark fehlerbehaftet.

Zur Vermeidung solcher Effekte kann man beispielsweise nur die Änderung $\Delta\underline{x}$ des Systemzustandes anlernen. Die Struktur des Emulators ergibt sich dann aus Abbildung 6. Der neue Systemzustand $\underline{x}_1$ ist dann die Summe aus der Änderung $\Delta\underline{x}$ und eines Durchgriffes des ursprünglichen Systemzustandes $\underline{x}_0$. Bei der Durchführung des Backpropagation-Algorithmus zum Anlernen des Reglers muß dann dieser Durchgriff berücksichtigt werden.

3.3.4 Die Skalierung der Lernmuster

Beim Anlernen eines Emulators für die Regelstrecke sind oft Skalierungen der Lernmuster sinnvoll. So kann beispielsweise bei fehlender Skalierung ein Emulator mit sigmoiden Ausgangsneuronen, die die Aktivierungsfunktion

$$f(s) = 2M(\frac{1}{1+e^{-s}} - \frac{1}{2})$$

besitzen, keine Zustände anlernen, deren Betrag größer als M ist. Ist nun zum Beispiel der zu lernende Ausgangszustand $x_{1e} = 51$ und die Begrenzung $M = 1$, so wird der Fehler die Schranke S mit

$$S = \frac{1}{2}(51 - 1)^2 = 1250$$

nicht unterschreiten können, unabhängig von der Einstellung der Gewichte im Emulatornetz.

Die in dieser Arbeit entwickelte Routine berücksichtigt eine separate Skalierung sowohl für die Eingangssignale, als auch für die Ausgangssignale der Lernmuster. Beim Anlernen des Reglers müssen diese Skalierungen entsprechend beachtet werden (siehe Abschnitt 3.4.2).

3.4 Das Anlernen des neuronalen Reglers

3.4.1 Das Anlernverfahren

Das zentrale Verfahren dieser Arbeit ist das Anlernverfahren des neuronalen Reglers zur Lösung des Regelungsproblems nach Abschnitt 3.1. Es wurde von B. Widrow und D. Nguyen in [13] vorgestellt. Der Trainingsvorgang ist schematisch in Abbildung 7 dargestellt.

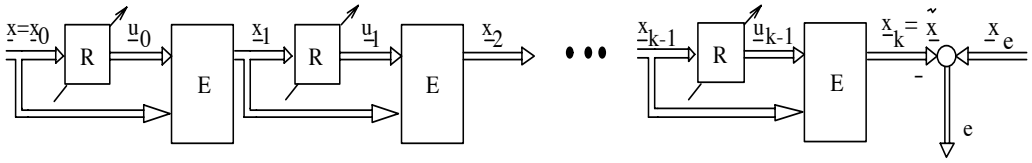

Abbildung 7: Der Trainingsalgorithmus zum Anlernen des Reglers; R = Regler; E = Emulator der Regelstrecke

Die Idee des Lernverfahrens für den Regler ist nun folgende:

Der neuronale Regler soll nach dem Entwurf in einem Regelkreis entsprechend Abbildung 1 eingesetzt werden. Dabei soll er die Streckenzustände vom Anfangszustand $\underline{x}_0$ beginnend über die Zwischenzustände $\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_{k-1}$ in den Endzustand $\underline{x}_k = \underline{\tilde{x}}$ überführen, der wiederum möglichst genau dem Sollwert $\underline{x}_e$ entsprechen soll. Dieser Vorgang ist in Abbildung 7 schematisch dargestellt, wobei anstelle der Strecke der Streckenemulator in Form eines neuronalen Netzes verwendet wird.
Nun kann diese Kette aus neuronalem Regler und Streckenemulator als ein großes Netz betrachtet werden, auf das nach Bildung des Fehlers $\underline{e} = \underline{x}_e - \underline{\tilde{x}}$ der Backpropagation-Algorithmus angewendet wird. Dies ist durch die Nachbildung der Strecke in Form eines neuronalen Netzes möglich geworden. Bei Verwendung dieses Verfahrens werden jeweils nur die Gewichte des Reglers eingestellt, die Gewichte des Emulators bleiben unverändert. Da real Regler und Emulator nur einmal vorliegen, werden somit bei einem Backpropagation-Zyklus k-mal die Gewichte des Reglers verändert.

Eine genauere Betrachtung des Verfahrens ergibt sich aus folgenden Überlegungen:

Die Regelstrecke habe insgesamt n unterschiedliche Zustände und es werden m unterschiedliche Anfangszustände $\underline{x}_{0\nu}, \nu \in \{1 \ldots m\}$ als Eingangswerte der Lernmuster zum Anlernen des Reglers erzeugt. Beispielhaft wird zunächst nur ein beliebiges, aber festes Lernmuster $(\underline{x} = \underline{x}_{0\nu}, \underline{x}_e)$ betrachtet.

Das Verfahren läßt sich in einen Feedforward- und in einen Backpropagation-Schritt aufteilen. Zu Beginn des Algorithmus werden die Gewichte des Reglers einmal zufällig initialisiert.

Der Feedforward-Schritt beginnt damit, daß die Eingangswerte $\underline{x} = \underline{x}_0$ des Lernmusters an die Reglereingänge gelegt werden. Der Feedforward-Schritt durch den Regler erzeugt daraufhin ein Stellsignal $\underline{u}_0$. Dieses Stellsignal $\underline{u}_0$ und der Anfangszustand $\underline{x}_0$ werden dann gemeinsam an den Eingang des Emulators der Regelstrecke gelegt. Der Ausgang des Emulators wird nach einem Feedforward-Schritt den neuen Zustand $\underline{x}_1$ annehmen, der wieder an den Reglereingang angelegt wird, und so weiter. Dieser Zyklus wiederholt sich insgesamt k mal bis die maximale Iterationszahl k_{max} erreicht wurde, oder eine Zustandsgröße eine Abbruchbedingung erfüllt hat. Danach befindet sich der Emulator der Regelstrecke im Zustand $\underline{x}_k = \tilde{\underline{x}}$.

Die Gewichte $\underline{w}$ des Reglers sollen nun durch einen Backpropagation-Schritt verändert werden. Diese Veränderung ist durch die das Reglersymbol kreuzenden Pfeile in Abbildung 7 angedeutet. Wie beim herkömmlichen Backpropagation-Schritt werden die Gewichte $\underline{w}$ derart adaptiert, daß sich der Fehler

$$E = \frac{1}{2}(\|\underline{x}_e - \tilde{\underline{x}}\|)^2$$

im nächsten Feedforward-Schritt verringert hat. Hierfür benötigt man den Gradienten des Fehlers der Stellsignale $\underline{u}_\nu$ des Reglers $\frac{\partial E}{\partial \underline{u}_\nu}$ mit $\nu = \{0, 1, \ldots, k-1, k\}$ für jeden der k Schritte. Berechnen kann man jedoch nur den Fehler im Endzustand $\tilde{\underline{x}} = \underline{x}_k$ mit $e = \|\underline{x}_e - \tilde{\underline{x}}\|$. Da aber die Regelstrecke in Form eines neuronalen Netzes vorliegt, kann leicht der Gradient des Fehlers E bezüglich der Eingangsgrößen $\underline{u}_{k-1}$ und $\underline{x}_{k-1}$ des Emulators der Regelstrecke berechnet werden.

Nimmt man beispielsweise Abbildung 5 als ein angelerntes Emulatornetz an, so ergibt sich für $\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial o_1}$ mit Gleichung (14)

$$\frac{\partial E}{\partial x_1} = -(\delta_2\, w_{21} + \delta_3\, w_{31}),$$

wobei sich δ_2 und δ_3 aus den Gleichungen (13) und (14) bestimmen lassen. Hierbei dient Abbildung 5 nur zur Veranschaulichung, da ein reales Emulatornetz ohne Berücksichtigung des BIAS-Term mindestens zwei Eingänge (einen Stellsignal- und einen Systemzustandeingang) besitzen muß.

Die Regelstrecke in Form eines neuronalen Netzes übersetzt also den Fehler des Endzustandes in den Fehlergradienten am Eingang des Emulators

bezüglich dessen Eingangsgrößen. Durch sukzessive Anwendung der Gleichungen (13) und (14) können nun die Gewichtsänderungen der Reglergewichte für alle k Schritte berechnet werden. Nach Addition aller k Gewichtsänderungen werden dann die Gewichte $\underline{w}$ des Reglers nach Beziehung (12) verändert.

Es sei an dieser Stelle ausdrücklich darauf hingewiesen, daß in der schematischen Darstellung von Abbildung 7 der Regler und der Emulator real nur einmal vorhanden sind. Die mehrfache Darstellung dient nur der Veranschaulichung des Verfahrens. In Wirklichkeit liegt eine Kreisstruktur nach Abbildung 1 vor.

Die maximale Anzahl an Zyklen k_{max} ist vor dem Entwurf festzulegen. Sie dient zur Vermeidung einer Endlosschleife des Verfahrens, falls keine der Abbruchbedingungen erfüllt wird. Abbruchbedingungen können beispielsweise das Verlassen des Definitionsbereiches einer Zustandsgröße oder das Erreichen des Sollwertes x_e sein. Diese Bedingungen müssen ebenfalls vor dem eigentlichen Entwurfsschritt explizit angegeben werden.

Liegen insgesamt m unterschiedliche Lernmuster vor, so wird das Verfahren für jedes der m Muster einzeln abgearbeitet. Als Einschränkung zum gewöhnlichen Backpropagation-Algorithmus werden jedoch die Gewichte nach Gleichung (12) nach Abarbeitung eines jeden Lernmusters sofort verändert.

3.4.2 Die Skalierung beim Anlernen des Reglers

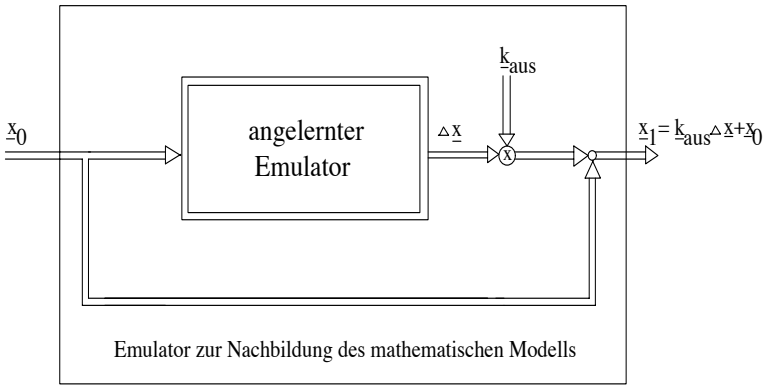

Abbildung 8: Der Emulator mit Durchgriff und Ausgangsskalierung

Es kann nun, wie in Abschnitt 3.3.4 bemerkt, notwendig sein, Skalierungen beim Entwurf des Emulators der Regelstrecke zu verwenden. Wird der nach Abbildung 8 angelernte Emulator zum Anlernen des Reglers verwendet, so

sind diese Skalierungen beim Entwurf des Reglers entsprechend zu berücksichtigen.
Die folgende Darstellung bezieht sich dabei auf den ν-ten Zustand des Streckenemulators.
Die Skalierung der Eingangswerte sei hier nun mit $SKAL_{ein\nu}$, die Skalierung der Ausgangswerte mit $SKAL_{aus\nu}$ bezeichnet. Es ist naheliegend, die Eingangszustände zum Anlernen des Reglers mit den gleichen Skalierungen $SKAL_{ein\nu}$ wie beim Anlernen des Emulators zu versehen. Um nun $x_{1\nu}$ zu bilden, muß man $k_{aus\nu}$ zu

$$k_{aus\nu} = \frac{SKAL_{aus\nu}}{SKAL_{ein\nu}} \tag{18}$$

wählen. Somit ergibt sich der Folgezustand aus:

$$x_{1\nu} = \frac{SKAL_{aus\nu}}{SKAL_{ein\nu}} \Delta x_\nu + x_{0\nu} \tag{19}$$

Bezeichnet man die nicht skalierten Größen mit $\Delta\hat{x}_\nu$ und $\hat{x}_{0\nu}$, so folgt mit

$$\Delta x_\nu = \frac{1}{SKAL_{aus\nu}} \Delta\hat{x}_\nu$$

und

$$x_{0\nu} = \frac{1}{SKAL_{ein\nu}} \hat{x}_{0\nu}$$

die Beziehung:

$$x_{1\nu} = \frac{1}{SKAL_{ein\nu}} (\Delta\hat{x}_\nu + \hat{x}_{0\nu})$$

Um $x_{1\nu}$ nun mit dem Endwert $x_{e\nu}$ vergleichen zu können, muß $x_{e\nu}$ auch entsprechend aus dem nicht skalierten Wert $\hat{x}_{e\nu}$ hervorgehen mit

$$x_{e\nu} = \frac{1}{SKAL_{ein\nu}} \hat{x}_{e\nu}.$$

Diese Skalierung der Sollwerte ist unbedingt zu berücksichtigen, da anderenfalls der Fehler

$$e_1 = \frac{\partial E}{\partial x_{1\nu}} = x_{e\nu} - x_{1\nu}$$

falsch berechnet wird. Der folgende Backpropagation-Algorithmus arbeitet dann nicht korrekt und berechnet eine falsche Änderung der Gewichte des Reglers.

3.4.3 Die Unterteilung des Anlernvorganges in Lektionen

Im Abschnitt 3.4.1 ist nichts über die Wahl der maximalen Anzahl der Iterationen $k \leq k_{max}$ des Feedforward-Schrittes gesagt worden. Der Fehler $e = \underline{x}_e - \underline{\tilde{x}}$ hängt jedoch entscheidend von der Wahl von k_{max} und vom gewählten Anfangszustand $\underline{x}_0$ ab. Ist die maximale Anzahl der Iterationen gering und der Anfangszustand $\underline{x}_0$ der Regelstrecke sehr weit vom Sollwert $\underline{x}_e$ entfernt, so kann beispielsweise bei Stellsignalbegrenzung des Reglers der Sollwert $\underline{x}_e$ nie erreicht werden. Daher ist immer darauf zu achten, daß k_{max} groß genug gewählt wird, das heißt daß das Regelungsproblem mit der maximal zulässigen Anzahl an Schritten prinzipiell lösbar sein muß.

Desweiteren können sich zu Beginn des Anlernvorganges des Reglers bei einer großen Anzahl maximal zulässiger Iterationsschritte die Zustände weit vom eigentlichen Sollwert $\underline{x}_e$ entfernen. Der daraus resultierende Fehler kann bei Regelstrecken mit mehreren Zustandsgrößen so groß werden, daß durch die begrenzte Zahlendarstellung in den Rechenmaschinen Zahlenüberläufe auftreten. In anderen Beispielen divergierte das Verfahren. Dies erklärt sich aus den bei großen Fehlern auftretenden Änderungen der Reglergewichte $\Delta\underline{w}$ nach Gleichung (12). Auch eine Verringerung der Lernrate ergab hierfür keine Abhilfe.

Ein weiteres Problem ergibt sich durch das Verwenden eines Emulators als Ersatz für die Regelstrecke. Die Zustände am Eingang des Streckenemulators können bei zu groß gewähltem k_{max} zusammen mit schlecht eingestellten Reglergewichten den Emulator in Zustände bringen, die außerhalb des Bereiches der Lernmuster liegen. Der Emulator wird dann unsinnige Folgezustände liefern und das Verfahren divergiert.

Deshalb ist beim Anlernen des Reglers für Regelstrecken höherer Ordnung der Anlernvorgang in Lektionen aufgegliedert worden. In den ersten Lektionen wurden nur Anfangswerte $\underline{x}_0$ ausgewählt, die sehr nahe dem gewünschten Sollwert $\underline{x}_e$ waren. Der Regler konnte also bei beliebiger Initialisierung der Reglergewichte die Regelstrecke innerhalb weniger Schritte in die Nähe des Sollwertes bringen. Die so angelernten Reglergewichte werden dann als Ausgangswerte zum Anlernen weiter vom Sollwert $\underline{x}_e$ entfernt liegender Anfangswerte $\underline{x}_0$ verwendet. Die maximal zulässige Schrittweite k_{max} wird dabei von Lektion zu Lektion weiter vergrößert. Die Anfangswerte der leichteren Lektionen sind hierbei als Teilmenge in der Menge der schwierigeren Anfangswerte der späteren Lektionen enthalten.
Der Regler lernt somit zunächst „leichtere" Anfangszustände in den Sollwert überzuführen, und dann darauf aufbauend immer „schwierigere" Anfangszustände.

4 Ausgewählte Beispiele

4.1 Übersicht

Die folgenden Beispiele sollen die bisherigen Überlegungen veranschaulichen. Um einen Zusammenhang zwischen dem dargestellten Entwurfsverfahren für neuronale Regler und bereits bekannten Verfahren herzustellen, wird zunächst die Deadbeat-Regelung einer linearen Strecke anhand des vorgestellten Verfahrens untersucht. Es wird ein I_2-System betrachtet, das zeitoptimal mit Hilfe eines Zustandsreglers geregelt werden soll. Für dieses System wird im Anschluß die Regelung unter Berücksichtigung einer Stellsignalbegrenzung diskutiert. Darauf folgend wird zur Erweiterung der Problematik die Regelung des Rückfahrverhaltens eines Autos betrachtet. Das Fahrzeug soll so geregelt werden, daß ein selbstständiges Einparken von verschiedenen Anfangsstellungen aus möglich ist. Zur Steigerung der Komplexität wird dann zum Abschluß noch das Rückfahrverhalten dieses Autos unter Berücksichtigung eines Hängers behandelt.

4.2 Die Deadbeat-Regelung einer linearen Strecke

4.2.1 Einführende Bemerkungen

Zunächst soll anhand eines einfachen, linearen Streckenmodells der Zusammenhang zwischen dem vorgestellten Verfahren und der zeitoptimalen Zustandsregelung dargestellt werden. Eine lineare Strecke bietet den besonderen Vorteil, daß der Entwurfsschritt des Anlernens des Emulators entfallen kann, da aus der beschreibenden, zeitdiskreten Zustandsdarstellung das neuronale Netz, welches die Strecke nachbildet, sofort abgelesen werden kann. Hierbei entfallen die Aktivierungsfunktionen $f(s(\underline{x}))$ in Bild 2 der einzelnen Neuronen, und die Neuronenausgänge $o(f(s(\underline{x})))$ ergeben sich einfach als gewichtete Summe der Eingangswerte $\underline{x}$.

Das nun folgende Beispiel macht von dem Satz 1 Gebrauch, der sich aus den Überlegungen in [7] und [10] ableiten läßt:

Satz 1 *Eine lineare, zeitdiskrete Regelstrecke der Ordnung n sei vollständig steuerbar. Dann existiert ein Zustandsregler mit der Zustandsrückführung* $\underline{k}^T$ *derart, daß mit Hilfe des Stellgesetzes*

$$u(k) = -\underline{k}^T \underline{x}(k)$$

die Zustandsgrößen $\underline{x}(k)$ *der Strecke innerhalb von n Schritten von jedem beliebigen Anfangszustand in jeden beliebigen Endzustand gebracht werden kann.*

Ein Regler mit diesen Eigenschaften wird auch als Deadbeat-Regler bezeichnet und ist hinsichtlich der Regelung zeitoptimal. Ein solcher Regler soll nun in Form eines neuronalen Netzes angelernt werden. Nach dem Anlernvorgang muß dann der Koeffizientenvektor $-\underline{k}^T$ den Gewichtungen $\underline{w}^T$ des Reglers entsprechen.

4.2.2 Das Streckenmodell

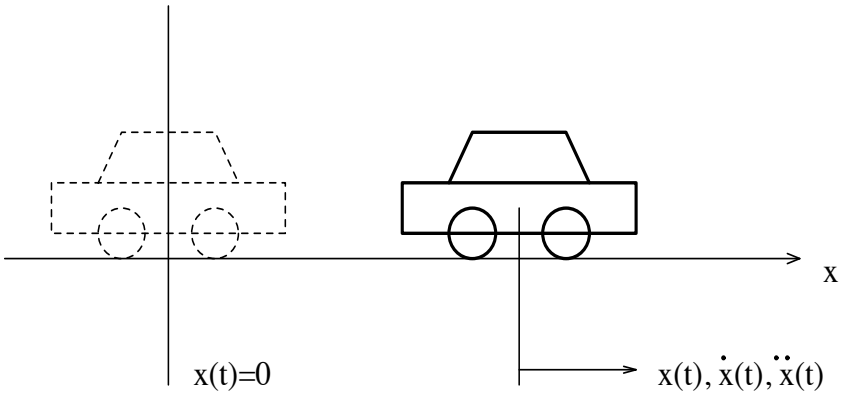

Abbildung 9: Veranschaulichung eines I_2-Systems

Als ein Streckenmodell wird ein I_2-System ohne Stellsignalbegrenzung mit der Beziehung

$$\ddot{x}(t) = u(t) \tag{20}$$

betrachtet. Zur Veranschaulichung einer solchen Strecke dient die Bewegung eines Fahrzeuges gemäß Bild 9, das sich reibungsfrei in eindimensionaler Richtung bewegen soll. Das Fahrzeug hat hierbei zum Zeitpunkt $t = 0$ eine Anfangsgeschwindigkeit von $\dot{x}(t = 0) = \dot{x}_0$ und befindet sich an der Stelle $x(t = 0) = x_0$. Der Zustand $\underline{x}(t)$ des Fahrzeuges kann mit Hilfe des Zustandsvektors

$$\underline{x}(t) = \begin{pmatrix} x(t) \\ \dot{x}(t) \end{pmatrix} = \begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix}$$

dargestellt werden.

Die zeitkontinuierliche Zustandsdarstellung der Bewegungsgleichung ergibt sich zu:

$$\begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}}_{\underline{\tilde{A}}} \begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} + \underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}_{\underline{\tilde{b}}} u(t) \tag{21}$$

Die diskretisierte Zustandsdarstellung folgt dann aus der sprunginvarianten Transformation mit

$$\underline{A}(T) = e^{\underline{\tilde{A}}T} = \underline{I} + \underline{\tilde{A}}T = \begin{pmatrix} 1 & T \\ 0 & 1 \end{pmatrix}$$

$$\underline{b} = \int_0^T \underline{A}(\tau)\underline{\tilde{b}}d\tau = \begin{pmatrix} \frac{T^2}{2} \\ T \end{pmatrix}$$

und der Abtastzeit $T = 1$ aus Gleichung (21) zu:

$$\begin{pmatrix} x_1(k+1) \\ x_2(k+1) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}}_{\underline{A}} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \underbrace{\begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix}}_{\underline{b}} u(k) \tag{22}$$

Aus Gleichung 22 läßt sich unmittelbar die Struktur des Emulators der Strecke in Form eines neuronalen Netzes gemäß Abbildung 10 angeben. Dieses

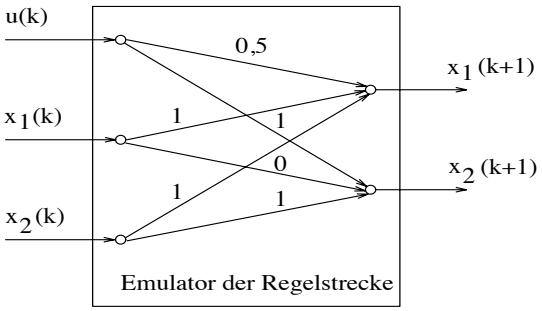

Abbildung 10: Struktur des Streckenemulators

neuronale Netz emuliert das mathematische Modellverhalten idealerweise exakt und kann nun als Emulatornetz der Regelstrecke zum Anlernen des Zustandsreglers verwendet werden.

4.2.3 Das Regelungsproblem

Das I_2-System soll nun innerhalb von $k = n = 2$ Schritten von einem zufällig gewählten Anfangszustand $\underline{x}_0 = \underline{x}(k = 0) = (x_{10} \quad x_{20})^T$ in den Endzustand $x_e = (x_{e1} \quad x_{e2})^T = (0 \quad 0)^T$ übergeführt werden. Ein Strukturbild der Regelstrecke in Regelungsnormalform (siehe [6],[16]) und des verwendeten Zustandsreglers zeigt Abbildung 11.

Welche Zustandsrückführungen $\underline{k} = (k_1 \quad k_2)^T$ sich zum Lösen des Regelungsproblems ergeben müssen, soll vorab anhand folgender Überlegung (siehe [7]) berechnet werden:

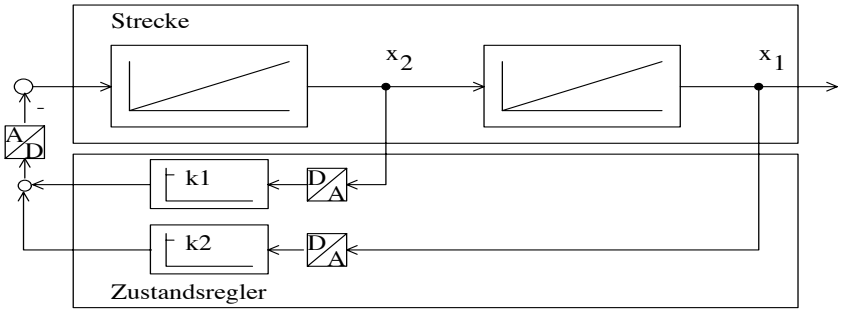

Abbildung 11: Strukturbild des Regelkreises

Die Regelstrecke befinde sich im Anfangszustand $\underline{x}_0$. Die geregelte Strecke wird im nächsten Schritt in den Folgezustand $\underline{x}_1$ übergehen mit

$$\underline{x}_1 = (\underline{A} - \underline{b}\,\underline{k}^T)\underline{x}_0 = \hat{\underline{A}}\,\underline{x}_0.$$

Der darauf folgende Zustand $\underline{x}_2$ sei der Endzustand mit

$$\underline{x}_2 = \hat{\underline{A}}\,\underline{x}_1 = \hat{\underline{A}}^2\,\underline{x}_0 = \underline{0} \tag{23}$$

Damit Gleichung (23) für beliebige Anfangswerte $\underline{x}_0$ erfüllt ist, muß

$$\hat{\underline{A}}^2 = \underline{0} \tag{24}$$

gelten. Die charakteristische Gleichung des Regelkreises lautet

$$det(z\underline{I} - \hat{\underline{A}}) = z^2 + a_1 z + a_0 = 0. \tag{25}$$

Nach dem Satz von Cayley-Hamilton (siehe [6]) erfüllt eine quadratische Matrix ihre eigene charakteristische Gleichung, das heißt es gilt weiterhin

$$\hat{\underline{A}}^2 + a_1\hat{\underline{A}} + a_0\underline{I} = \underline{0}.$$

Zur Erfüllung von Beziehung (24) muß für nichtverschwindendes $\hat{\underline{A}}$ also $a_0 = a_1 = 0$ sein, und somit folgt für die charakteristische Gleichung aus Beziehung (25):

$$det(z\underline{I} - \underline{A} + \underline{b}\,\underline{k}^T) = z^2 = 0 \tag{26}$$

Setzt man nun in Gleichung (26) die konkreten Größen für $\underline{A}$ und $\underline{b}$ aus Beziehung (22) ein und führt einen Koeffizientenvergleich durch, so folgt für den Rückführvektor k

$$\underline{k} = \begin{pmatrix} 1 \\ 1,5 \end{pmatrix}. \tag{27}$$

Für $\hat{\underline{A}}$ ergibt sich dann

$$\hat{\underline{A}} = \underline{A} - \underline{b}\underline{k}^T = \begin{pmatrix} 0,5 & 0,25 \\ -1 & -0,5 \end{pmatrix}.$$

Nach der Adaption des Reglers sollten sich also die Reglergewichte zu $\underline{w} = (-1 \quad -1,5)^T$ ergeben. Die Rückführkoeffizienten werden hierbei derart eingestellt, daß alle Pole der geregelten Strecke in den Ursprung der z-Ebene verschoben werden.

4.2.4 Der Anlernvorgang des Reglers

Da es sich bei dem anzulernenden Regler um einen Zustandsregler gemäß Abbildung 11 handelt, ist die Struktur des Reglers bereits festgelegt. Der geschlossene Regelkreis nach Abbildung 1 hat also die spezielle Form von Abbildung 12. Im Regelkreis sind die Voraussetzungen nach Satz 1

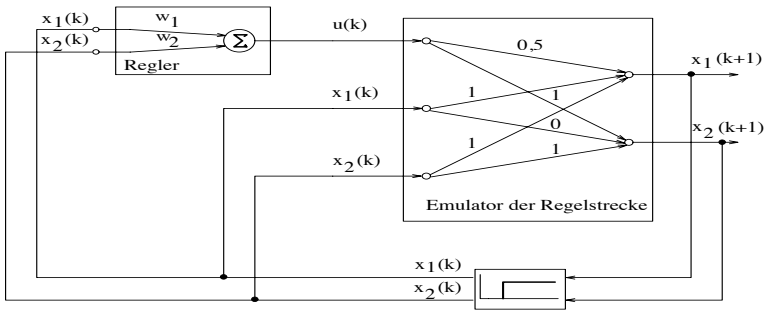

Abbildung 12: Strukturbild des Regelkreises

erfüllt. Daher läßt sich die Regelstrecke mit korrekt eingestellten Gewichten $\underline{w} = (w_1 \quad w_2)^T$ des Reglers von jedem beliebigen Anfangszustand $\underline{x}_0 = (x_{10} \quad x_{20})^T$ in den gewünschten Endzustand $\underline{x}_e = \underline{0}$ überführen. Die Anzahl der Schritte beträgt hierbei $k = n = 2$.

Zur Darstellung des Adaptionsalgorithmus zum Einstellen der Reglergewichte $\underline{w}$ nach Abschitt 3.4.1 ergibt sich die Struktur nach Bild 13. Im

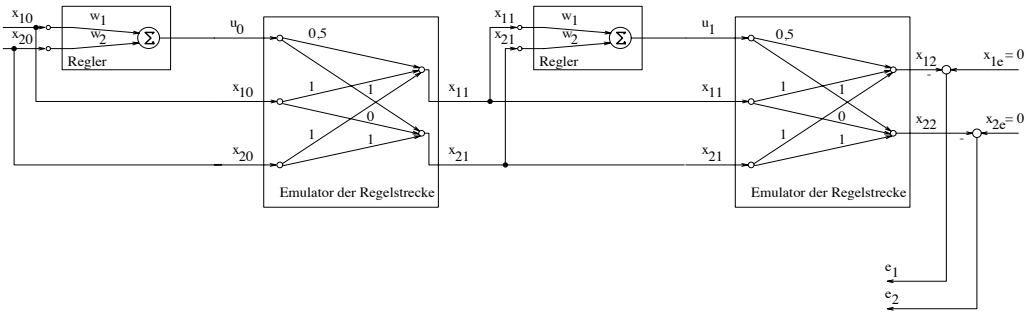

Abbildung 13: Der Feedforward-Schritt zum Anlernen des Reglers

Feedforward-Schritt wird nach zufälliger Initialisierung der Gewichte w_1 und w_2 ein Anfangszustand $\underline{x}_0 = (x_{10} \quad x_{20})^T$ an die Reglereingänge gelegt. Der

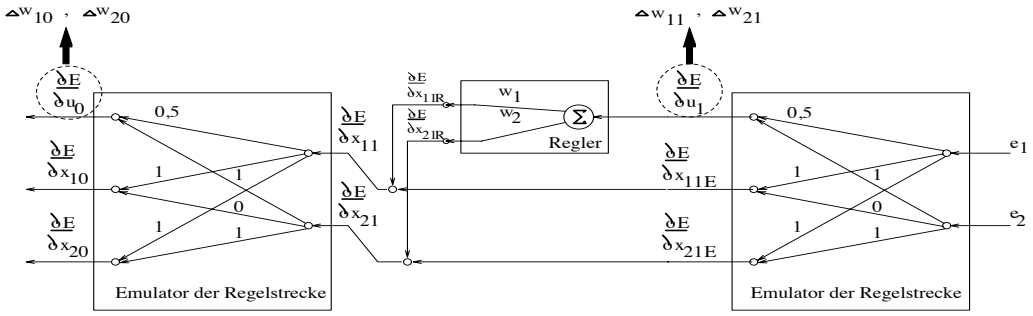

Abbildung 14: Der Backpropagation-Schritt zum Anlernen des Reglers

Emulator liefert dann zusammen mit dem Stellsignal u_0 und $\underline{x}_0$ den neuen Zustand $\underline{x}_1 = (x_{11} \quad x_{21})^T$, und daraus wiederum u_1 und $\underline{x}_2$.

Im Backpropagation-Schritt wird dann der Ausgangsfehler $\underline{e} = (e_1 \quad e_2)^T$ zur Einstellung der Reglergewichte w_1 und w_2 verwendet. Dieser Vorgang hat durch die linearen Aktivierungsfunktionen eine sehr einfache Form und ist schematisch in Abbildung 14 dargestellt.
Man erhält am Emulatoreingang die Größen $\frac{\partial E}{\partial u_1}, \frac{\partial E}{\partial x_{11E}}$ und $\frac{\partial E}{\partial x_{21E}}$. Mit $\frac{\partial E}{\partial u_1}$ werden die Gewichtsänderungen Δw_{11} und Δw_{21} im Regler berechnet. Außerdem erhält man am Reglereingang die Größen $\frac{\partial E}{\partial x_{11R}}$ und $\frac{\partial E}{\partial x_{21R}}$. Die Größen $\frac{\partial E}{\partial x_{11}}$ und $\frac{\partial E}{\partial x_{21}}$ ergeben sich dann aus

$$\frac{\partial E}{\partial x_{11}} = \frac{\partial E}{\partial x_{11E}} + \frac{\partial E}{\partial x_{11R}} \tag{28}$$

und

$$\frac{\partial E}{\partial x_{21}} = \frac{\partial E}{\partial x_{21E}} + \frac{\partial E}{\partial x_{21R}}. \tag{29}$$

Durch den Emulator werden $\frac{\partial E}{\partial x_{11}}$ und $\frac{\partial E}{\partial x_{21}}$ in die Größen $\frac{\partial E}{\partial u_0}, \frac{\partial E}{\partial x_{10}}$ und $\frac{\partial E}{\partial x_{20}}$ übersetzt. Aus $\frac{\partial E}{\partial u_0}$ können nun wieder die Gewichtsänderungen Δw_{10} und Δw_{20} berechnet werden. Am Ende des Backpropagation-Schrittes werden die Gewichte w_1 und w_2 gemäß Gleichung (12)

$$\begin{aligned} w_{1neu} &= w_1 + \Delta w_1 \\ w_{2neu} &= w_2 + \Delta w_2 \end{aligned}$$

verändert. Hierbei ist

$$\begin{aligned} \Delta w_1 &= \Delta w_{11} + \Delta w_{10} \\ \text{und} \qquad \Delta w_2 &= \Delta w_{21} + \Delta w_{20}. \end{aligned}$$

4.2.5 Ergebnisse

Mit Hilfe des in Abschnitt 4.2.4 beschriebenen Anlernvorganges ergeben sich die Reglergewichte $\underline{w}$ nach $k = 3$ Lernepochen zu

$$\underline{w} = \begin{pmatrix} -1 \\ -1,5 \end{pmatrix}. \tag{30}$$

Gleichung (30) entspricht hierbei exakt den theoretisch gefundenen Werten für den Rückführvektor $-\underline{k}$ nach Gleichung (27).
Die Erzeugung der Trainingsmuster erfolgte zufällig unter Verwendung des Programmpaketes Matlab. Die Muster lagen in den Wertebereichen

$$\begin{aligned} x_{10} &\in \{0 \ldots 100\} \\ x_{20} &\in \{-100 \ldots 100\}. \end{aligned}$$

Die Anzahl der erzeugten Trainingsmuster betrug beim Anlernen $m = 296$.
Nach dem Anlernvorgang wurde der Regler als ein statisches Gebilde in den Regelkreis nach Abbildung 12 eingesetzt. Die Phasenebenendarstellung und der zugehörige Stellsignalverlauf eines Rückfahrvorganges des Fahrzeuges für zwei zufällig ausgewählte Anfangszustände zeigen die Abbildungen 15 und 16.

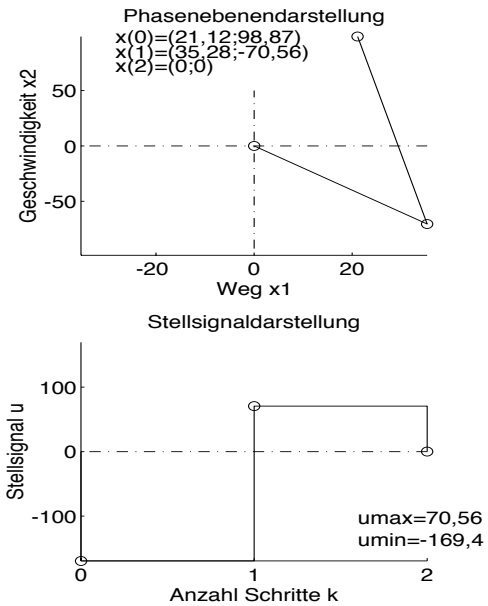

Abbildung 15: Phasenebenen- und Stellsignaldarstellung eines Rückfahrvorganges mit $x_{20} > 0$

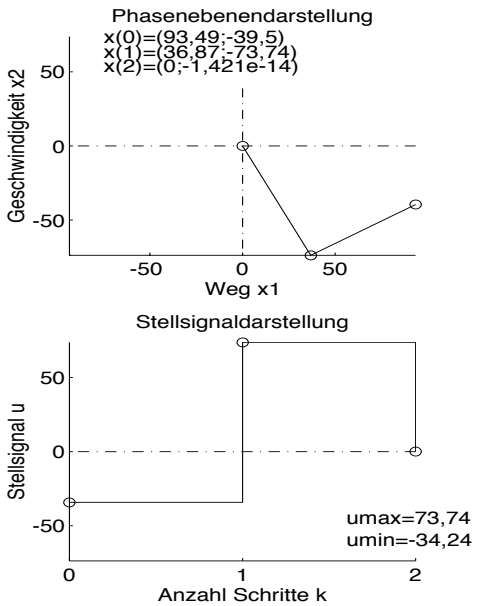

Abbildung 16: Phasenebenen- und Stellsignaldarstellung eines Rückfahrvorganges mit $x_{20} < 0$

4.2.6 Die Einführung einer Stellsignalbegrenzung

Die in den letzten Abschnitten durchgeführten Überlegungen sind in der Praxis nur beschränkt anwendbar, da bei realen Strecken Stellsignalbegrenzungen auftreten. Diese Stellsignalbegrenzung stellt eine Nichtlinearität der Strecke dar. Sind bei einer Regelung nicht mehr beliebig große Stellsignale zugelassen, so kann eine Regelstrecke n-ter Ordnung im allgemeinen nicht mehr innerhalb von n Schritten von jedem beliebigen Anfangszustand in jeden beliebigen Endzustand gebracht werden.

Im folgenden sei wieder die Regelstrecke nach Abschnitt 4.2.2 betrachtet, erweitert durch eine Stellsignalbegrenzung am Stellsignaleingang auf $M = \pm\frac{1}{2}$. Als Nebenbedingung gelte, daß die Regelstrecke für positive Anfangsauslenkungen $x_{10} \geq 0$ den Endwert $x_{1e} = 0$ erreicht, ohne diesen beim Rückfahrvorgang zu unterschreiten. Es ergibt sich die Forderung

$$x_{1\nu} \geq 0 \qquad \forall \nu \in \{0, 1, \ldots, k\}. \tag{31}$$

Zur Motivation dieser Forderung kann man sich unter Berücksichtigung der Wagenabmessungen eine Wand in Bild 9 links des Endwertes $x_{1e} = 0$ vorstellen.

Aus der Forderung (31) folgt, daß nicht beliebige Anfangszustände als Startwerte zugelassen werden können. Anschaulich werden zum Beispiel diese Zustände ausgenommen sein, bei denen das Fahrzeug nahe dem Zielwert $x_{1e} = 0$ ist und sich mit hoher Geschwindigkeit auf diesen zubewegt, das heißt die Zustände $\underline{x}_0 = (x_{10} \approx 0 \qquad x_{20} \ll 0)^T$. Unproblematisch sind hingegen Anfangszustände, bei denen $x_{20} > 0$ ist. Da sich das Fahrzeug zu Beginn von x_{1e} zunächst entfernt, wird x_{1e} trotz Stellsignalbegrenzung sicherlich nicht unterschritten werden. Es sei daher der Fall

$$x_{10} \geq 0,\ x_{20} \leq 0$$

betrachtet. Die Herleitung der folgenden Beziehung wird im Kontinuierlichen durchgeführt und kann entsprechend auf das Diskrete übertragen werden:

Gemäß Gleichung (20) gilt für die Strecke die Beziehung $\ddot{x}_1(t) = u(t)$. Durch zweifache Integration dieser Gleichung bezüglich der Zeit t folgt

$$x_2(t) = u(t)\,t + x_{20} \tag{32}$$

$$x_1(t) = \frac{1}{2}u(t)\,t^2 + x_{20}\,t + x_{10} \tag{33}$$

Durch Auflösen von Gleichung (32) nach der Zeit t und Einsetzen in die Gleichung (33) folgt

$$x_1(x_2(t)) = \frac{1}{2}\frac{(x_2(t) - x_{20})^2}{u(t)} + x_{20}\frac{x_2(t) - x_{20}}{u(t)} + x_{10} \geq 0\ \forall t \tag{34}$$

Von der Parabelschar nach Gleichung (34) wird nun die Parabel ausgewählt, für die sich das Stellsignal $u(t) = \pm M$ in der Stellsignalbegrenzung M befindet. Berücksichtigt man nun noch, daß der Endzustand identisch Null sein muß, so folgt aus Gleichung (34)

$$x_1(x_2(t)) = \pm \frac{1}{2M} x_2^2(t) \tag{35}$$

Da nach Voraussetzung nur positive Werte für x_1 zulässig sind, ergibt sich in der Phasenebene eine Parabel nach Abbildung 17. Alle Anfangszustände, die

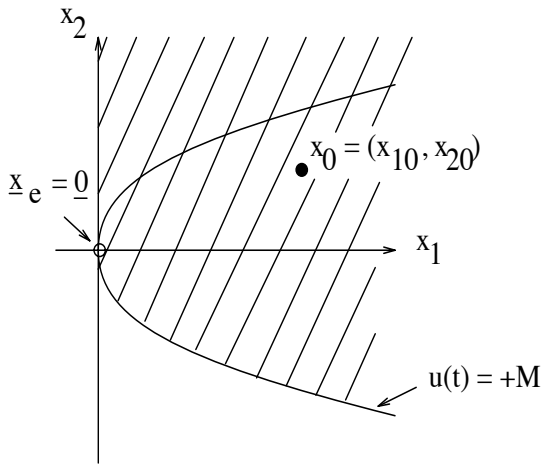

Abbildung 17: Phasenebenendarstellung der zulässigen Anfangszustände bei Stellsignalbegrenzung

sich in Bild 17 im schraffierten Bereich befinden, können noch ohne Überschwingen bei begrenztem Stellsignal in den Ursprung übergeführt werden. Somit ergibt sich aus Gleichung (35) die Bedingung für die Anfangszustände bei $x_{20} < 0$ zu

$$x_{10} \geq \frac{1}{2M} x_{20}^2. \tag{36}$$

Berücksichtigt man nun noch die Begrenzung mit $M = \frac{1}{2}$, so folgt aus Gleichung (36) schließlich

$$x_{10} \geq x_{20}^2. \tag{37}$$

Diese Beziehung ist bei der Wahl der Lernmuster zu berücksichtigen.

Beim Anlernen des Reglers wurde ein Zustandsregler entsprechend Bild 18 verwendet. Als Aktivierungsfunktion wurde eine Sigmoidfunktion gemäß

$$u(\tilde{u}(t)) = \frac{1}{1 + e^{-\tilde{u}(t)}} - \frac{1}{2} \tag{38}$$

eingesetzt. Somit ist das Reglerausgangssignal bereits auf $M = \pm\frac{1}{2}$ begrenzt. Die Stellsignalbegrenzung am Streckeneingang spricht daher nicht an und kann dadurch in Bild 18 unberücksichtigt bleiben.

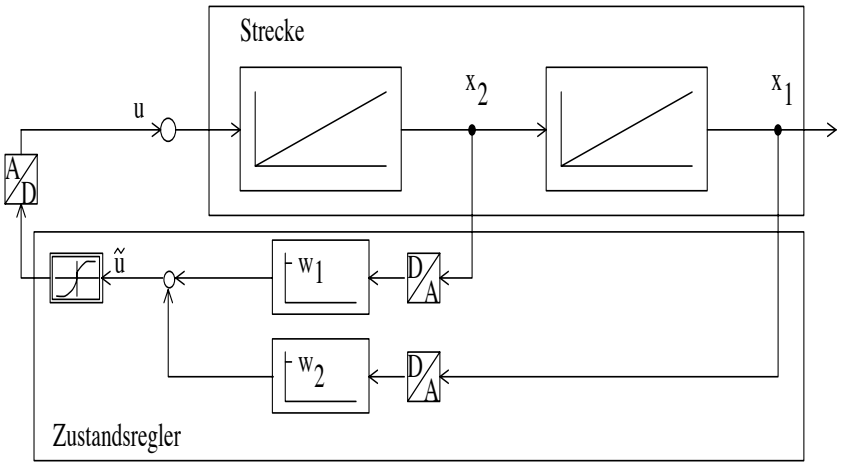

Abbildung 18: Strukturbild des Regelkreises mit Stellsignalbegrenzung

4.2.7 Ergebnisse bei Verwendung einer Stellsignalbegrenzung

Die Reglergewichte $\underline{w}$ ergeben sich mit dem Verfahren nach Abschnitt 3.4.1 unter Verwendung der Stellsignalbegrenzung nach (38) nun nach 100 Epochen zu

$$\underline{w} = \begin{pmatrix} -0,25365 \\ -6.24304 \end{pmatrix}. \tag{39}$$

Die Abbruchbedingung der einzelnen Rückfahrschritte war, daß der Zustand x_1 kleiner als der Endwert $x_{1e} = 0$ wird. In diesem Fall wurde nach jedem Trainingsmuster ein Backpropagation-Schritt für die Kette aus Emulator und Regler durchgeführt.
Die Anzahl der erzeugten Trainingsmuster betrug beim Anlernen $m = 838$. Die Muster lagen in den Wertebereichen

$$\begin{aligned} x_{10} &\in \{0 \ldots 400\} \\ x_{20} &\in \{-20 \ldots 20\}. \end{aligned}$$

Die Erzeugung erfolgte zufällig unter Berücksichtigung der Gleichung (37).

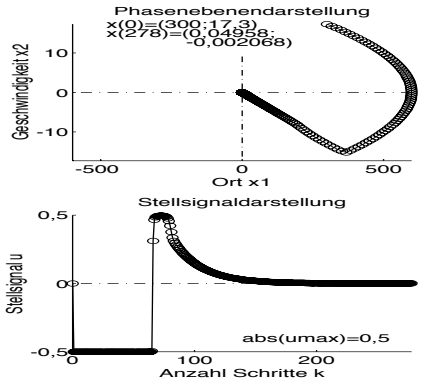

Abbildung 19: Phasenebenen- und Stellsignaldarstellung eines Rückfahrvorganges mit $x_{20} > 0$

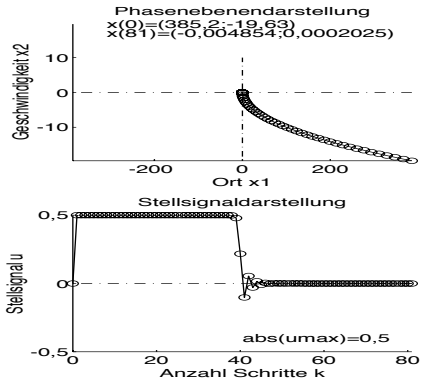

Abbildung 20: Phasenebenen- und Stellsignaldarstellung eines Rückfahrvorganges mit $x_{20} < 0$

Nach dem Anlernvorgang wurde der Regler in den Regelkreis nach Abbildung 18 eingesetzt. Die Phasenebenendarstellung und der zugehörige Stellsignalverlauf eines Rückfahrvorganges des Fahrzeuges für zwei zufällig ausgewählte Anfangszustände zeigen die Abbildungen 19 und 20. Die maximal zulässige Anzahl an Schritten wurde hierbei auf $k_{max} = 300$ festgelegt.

4.2.8 Ausblick

Der in Abschnitt 4.2.6 entworfene Regler ist zeitlich nicht optimal entworfen worden. Man könnte nun versuchen, eine zeitliche Optimierung dadurch zu erreichen, daß nach einem erfolgreichen Rückfahrvorgang des Fahrzeuges im nächsten Backpropagation-Schritt zur Fehlerbildung nicht der Endzustand $\underline{x}_k$ des Fahrzeuges, sondern der vorletzte Zustand $\underline{x}_{k-1}$ verwendet wird. Dieser Vorgang wird so lange wiederholt, bis der Fehler $\underline{e} = \underline{x}_e - \underline{x}_{k-1}$ unter eine vorgebbare Abbruchschranke gesunken ist. Die Größe $\underline{x}_e$ ist hierbei wieder der gewünschte Endzustand des Fahrzeuges. Im Anschluß daran wird der Zustand $\underline{x}_{k-2}$ zur Fehlerbildung herangezogen, und so fort. Das Verfahren ist beendet, wenn der Regler bei einem Zustand $\underline{x}_\nu$, $\nu \in \{0, 1, \ldots, k-1, k\}$ das Fahrzeug in den Grenzen der Fehlerschranke nicht mehr in den Sollwert $\underline{x}_e$ überführen kann. Es sind als Reglergewichte dann diejenigen zu verwenden, bei denen das Regelziel für sämliche Trainingsmuster mit kleinster Anzahl an Schritten erreicht wurde.

4.3 Die Regelung einer nichtlinearen Strecke

4.3.1 Einführende Bemerkungen

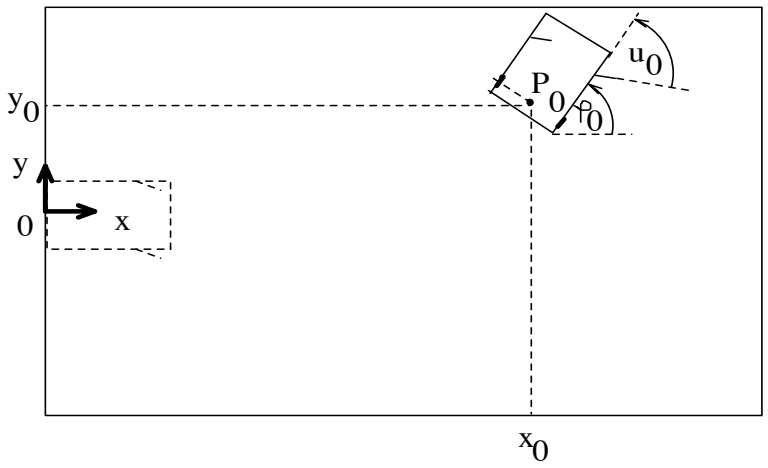

Abbildung 21: Beispiel einer nichlinearen Regelstrecke

Zur Behandlung eines Beispiels einer komplizierteren nichtlinearen Regelstrecke wird ein Auto verwendet, das sich nach Abbildung 21 im Ausgangszustand an der Stelle (x_0, y_0) befindet. Zusammen mit dem Ausgangswinkel φ_0 ist die Ausgangsstellung des Autos eindeutig festgelegt. Das Regelungsproblem besteht nun darin, einen Regler zu finden, der das Fahrzeug von zufällig gewählten Ausgangszuständen $\underline{x}_0 = (x_0, y_0, \varphi_0)$ mit minimalem Fehler in den Endzustand $\underline{x}_e = (x = 0, y = 0, \varphi = 0)$ überführt. Hierbei sind nur Rückwärtsfahrvorgänge zugelassen. Das Stellsignal u entspricht dabei dem Winkel der Vorderräder bezüglich der Längsachse des Fahrzeuges.

Beim Entwurf eines solchen Reglers muß zuächst eine Modellbildung des physikalischen Problems durchgeführt werden. Anschließend können daraus Trainingsmuster erzeugt werden, die zum Anlernen des Streckenemulators verwendet werden. Nach Entwurf des Streckenemulators wird dann der eigentliche Reglerentwurf nach Abschnitt 3.4.1 durchgeführt.

Entsprechend dieser Vorgehensweise gliedern sich die folgenden Abschnitte zunächst in die Modellbildung der Strecke, in die Emulation der Strecke als neuronales Netz, in den Reglerentwurf und abschließend in die Ergebnisse des Regelung.

4.3.2 Das Streckenmodell

Im folgenden werden die Bewegungsgleichungen des Autos für den diskreten Fall abgeleitet. Hierzu wird Abbildung 22 verwendet. Der Ausgangszustand

sei $\underline{x}_0 = (x_0\ y_0\ \varphi_0)^T$. Betrachtet wird der Punkt $Q_0 = (Q_{0x}\ Q_{0y})^T$ während

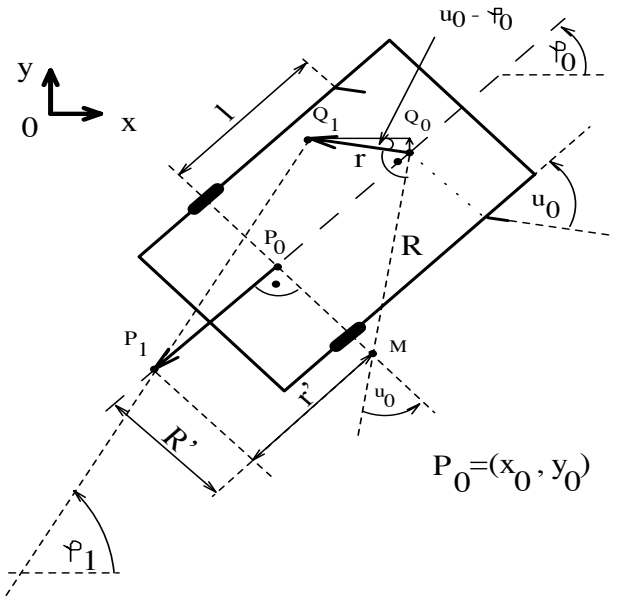

Abbildung 22: Ein Rückfahrschritt des Autos

eines Rückfahrschrittes der Länge r. Der Punkt Q_0 befindet sich nach einem Schritt im Punkt $Q_1 = (Q_{1x}\ Q_{1y})^T$, wobei sich der Zusammenhang

$$\begin{pmatrix} Q_{1x} \\ Q_{1y} \end{pmatrix} = \begin{pmatrix} Q_{0x} \\ Q_{0y} \end{pmatrix} - r \begin{pmatrix} \cos(\varphi_0 - u_0) \\ \sin(\varphi_0 - u_0) \end{pmatrix} \tag{40}$$

ergibt.
Um hierbei die Bewegung des Punktes $P_0 = (x_0\ y_0)^T$ zum Punkt $P_1 = (x_1\ y_1)^T$ zu ermitteln, wird eine Kreisbewegung des Fahrzeuges um den Punkt M angenommen. Für kleine Schrittweiten gilt dann der Zusammenhang

$$\frac{r'}{r} \approx \frac{R'}{R} \tag{41}$$

Aus den Winkelbeziehungen

$$\begin{aligned} \tan(u_0) &= \frac{l}{R'} \\ \sin(u_0) &= \frac{l}{R} \end{aligned}$$

folgt dann nach Gleichung (41) näherungsweise

$$r' = r\cos(u_0). \tag{42}$$

Entsprechend Gleichung (40) ergibt sich aus Beziehung (42) für den Punkt P_1:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \underbrace{r\cos(u_0)}_{r'} \begin{pmatrix} \cos(\varphi_0) \\ \sin(\varphi_0) \end{pmatrix} \tag{43}$$

Für die neue Winkelstellung des Fahrzeuges ergibt sich

$$\tan(\varphi_1) = \frac{Q_{1y} - y_1}{Q_{1x} - x_1} \tag{44}$$

Berücksichtigt man den Zusammenhang zwischen den Punkten P_0 und Q_0 gemäß

$$\begin{pmatrix} Q_{0x} \\ Q_{0y} \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + l \begin{pmatrix} \cos(\varphi_0) \\ \sin(\varphi_0) \end{pmatrix} \tag{45}$$

und setzt die Gleichungen (40) und (43) in die Beziehung (44) ein, so ergibt sich nach Auflösen für φ_1:

$$\varphi_1 = \arctan(\frac{l\sin(\varphi_0) + r\cos(\varphi_0)\sin(u_0)}{l\cos(\varphi_0) - r\sin(\varphi_0)\sin(u_0)}) \tag{46}$$

Die Verallgemeinerung der Beziehungen (43) und (46) führt somit auf die Bewegungsgleichungen (47) des Fahrzeuges:

$$\begin{aligned} x(k+1) &= x(k) - r\cos(\varphi(k))\cos(u(k)) \\ y(k+1) &= y(k) - r\sin(\varphi(k))\cos(u(k)) \\ \varphi(k+1) &= \arctan(\frac{l\sin(\varphi(k)) + r\cos(\varphi(k))\sin(u(k))}{l\cos(\varphi(k)) - r\sin(\varphi(k))\sin(u(k))}) \end{aligned} \tag{47}$$

4.3.3 Die Strecke als neuronales Netz

Ziel ist es nun wiederum, die Systemgleichungen des Fahrzeuges möglichst gut durch ein neuronales Netz nachzubilden.
In Anlehnung an [13] besteht die Struktur dieses mehrschichtigen Netzes aus vier Eingangsneuronen (für drei Zustandsgrößen und einem Stellsignal als Eingang), einer versteckten Schicht aus 25 Neuronen und einer Ausgangsschicht mit drei Neuronen, deren Anzahl sich aus der Anzahl der Zustandsgrößen ergibt. Als Aktivierungsfunktionen wurden die Tangenshyperbolicus-Funktionen gewählt. Weiterhin wurde der BIAS-Term verwendet.
Zum Erzeugen geeigneter Lernmuster zum Anlernen des Netzes wurde das Fahrzeug an $m = 986$ zufällig ausgewählten Positionen mit ebenfalls zufällig ausgewählten Ausgangswinkeln plaziert und ein Rückfahrschritt durchgeführt. Die dabei verwendeten Stellsignale u sind auch zufällig innerhalb der festgelegten Grenzen ausgewählt worden.Die Voreinstellungen der Größen l und r waren $l = 6$ und $r = 0,8$, die Voreinstellungen der restlichen

Größe	Bereich	Normierung	normierter Bereich
u	$\{-\frac{7}{18}\pi \dots \frac{7}{18}\pi\}$	$\frac{7}{18}\pi$	$\{-1 \dots 1\}$
x	$\{0 \dots 100\}$	100	$\{0 \dots 1\}$
y	$\{-50 \dots 50\}$	50	$\{-1 \dots 1\}$
φ	$\{-\pi \dots \pi\}$	π	$\{-1 \dots 1\}$

Tabelle 1: Liste der Wertebereiche der Ausgangszustände

Größe	Bereich	Normierung	normierter Bereich
Δx	$\{-0,8 \dots 0,8\}$	1	$\{-0,8 \dots 0,8\}$
Δy	$\{-0,8 \dots 0,8\}$	1	$\{-0,8 \dots 0,8\}$
$\Delta\varphi$	$\{-0,126 \dots 0,126\}$	0,18	$\{-0,7 \dots 0,7\}$

Tabelle 2: Liste der Wertebereiche der Zustandsänderungen

Größen zeigt Tabelle 1. Weiterhin wurde ein Durchgriff gemäß Abschnitt 3.3.3 verwendet. Die anzulernenden Zustände waren somit nicht die neue Stellung des Fahrzeuges, sondern nur die Änderung $(\Delta x, \Delta y, \Delta\varphi)$ bezüglich der zufällig gewählten Ausgangszustände. Deren Wertebereiche sind in Tabelle 2 dargestellt. Die Skalierung der Größen erfolgte unter Berücksichtigung der Überlegungen in 3.4.2 weitgehend empirisch.

4.3.4 Der Anlernvorgang des Reglers

Die Struktur des Reglers besteht aus drei Eingangsneuronen (entsprechend der Anzahl der Zustände), einer versteckten Schicht mit 25 Neuronen und einer Ausgangsschicht mit einem Neuron. Als Aktivierungsfunktion wurde der Tangenshyperbolicus, zur Erhöhung des Freiheitsgrades wurde der BIAS-Term verwendet.

Der Anlernvorgang des Reglers gestaltet sich nun ähnlich wie in Abschnitt 4.2.4. Der wesentliche Unterschied besteht in der Wahl der Lernmuster. Ein Anlernvorgang des ungelernten Reglers mit zufällig ausgewählten Anfangszuständen ohne Unterteilung in Lektionen scheiterte. Entsprechend Abschnitt 3.4.3 sind die Lernschritte daher in vier Lektionen unterteilt worden. Für die Lernmuster aller Lektionen wurde zunächst das Fahrzeug in den gewünschten Endzustand, hier also in den Ursprung gestellt. Durch die Umkehrung der Systemgleichungen (47) fährt das Fahrzeug nun eine festgelegte Anzahl an Schritten k vorwärts aus dem Ursprung heraus. Das dabei ver-

wendete Stellsignal u ist bei jedem Schritt zufällig ausgewählt worden. Die Endstellung des Fahrzeuges ist dann als Lernmuster zum Anlernen des Reglers verwendet worden. Diese Vorgehensweise hat den Vorteil, daß sicher innerhalb der Schrittzahl k eine Lösung des Regelungsproblems existiert.
Die einzelnen Lektionen unterscheiden sich nun in der Anzahl der Schritte k. Die Zuordnung zu den einzelnen Lektionen zeigt Tabelle 4.3.4. Ebenfalls

Lektion	Schritte k	Anzahl Lernmuster m
1	5	20
2	20	50
3	45	50
4	150	100

aufgeführt ist die Anzahl der Lernmuster je Lektion.
Die Umkehrungen der Systemgleichungen (47) sind in (48) angegeben.

$$\begin{aligned} \varphi(k+1) &= \arctan\left(\frac{l\sin(\varphi(k)) - r\cos(\varphi(k))\sin(u(k))}{l\cos(\varphi(k)) + r\sin(\varphi(k))\sin(u(k))}\right) \\ x(k+1) &= x(k) + r\cos(\varphi(k+1))\cos(u(k)) \\ y(k+1) &= y(k) + r\sin(\varphi(k+1))\cos(u(k)) \end{aligned} \tag{48}$$

Nach Beendigung der vierten Lektion war der Regler in der Lage, auch zufällig gewählte Anfangszustände in die gewünschte Endposition überzuführen.

4.3.5 Ergebnisse

In den Bildern 23 bis 25 sind die geregelten Rückfahrvorgänge zweier zufällig ausgewählter Anfangspositionen dargestellt. Als Regelstrecke wurde bei den Rückfahrvorgängen das mathematische Modell nach Gleichung (47) verwendet. Die Anfangs- und die Endposition des jeweiligen Rückfahrvorganges und die Anzahl der benötigten Rückfahrschritte sind in den Abbildungen angegeben. Bei den Angaben des x-Wertes ist zu berücksichtigen, daß durch die Schrittweite von $r = 0,8$ die Genauigkeit des Zustandes x nicht größer als r sein kann.
Das Bild 23 stellt eine Regelung eines einfachen Anfangszustandes dar. Durch die weite Entfernung der x-Position vom gewünschten Endwert hat der Regler genügend Spielraum, das Fahrzeug auf eine Horizontale durch den Ursprung zu bringen. Die Bilder 24 und 25 hingegen stellen eine schlecht geregelte Anfangsbedingung dar. Der Regler kann durch die Nähe der Anfangsposition

in x-Richtung das Fahrzeug nicht auf die Ideallinie in Form einer Horizontalen durch den Ursprung bringen. Weiterhin ist eine Rotation des Fahrzeuges zu Beginn des Rückfahrvorganges um fast 2π zu erkennen. Dies ergibt sich

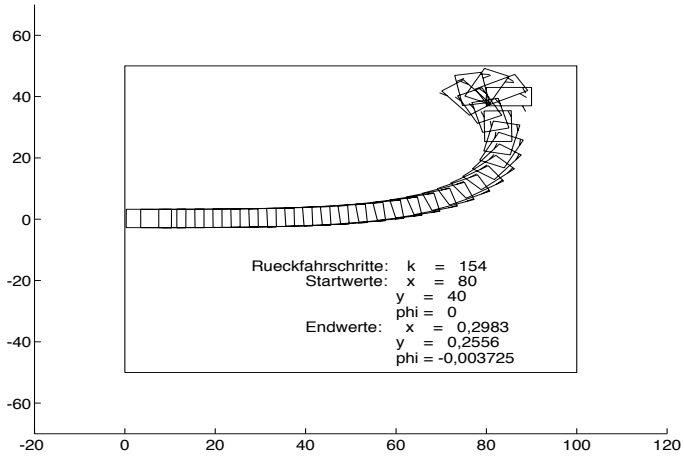

Abbildung 23: Ein Rückfahrvorgang des Autos

daraus, daß der Winkel φ beim Anlernvorgang die Bedingung

$$|\varphi| \leq \pi$$

erfüllen muß. Eine geringfügige Rotation im Gegenuhrzeigersinn ergäbe jedoch eine Verletzung dieser Bedingung aufgrund der Mehrdeutigkeit der Winkel.

Zustand	x	y	φ
absoluter Fehler (linearer Mittelwert)	$0,4012$	$0,1621$	$-0,0099$
absoluter Fehler (Betragsmaximum)	$0,7999$	$5,7250$	$0,3633$

Tabelle 3: Fehler der Endposition bei 1000 Rückfahrversuchen

In Tabelle 3 wurden die Fehler für 1000 Rückfahrversuche mit zufällig gewählten Anfangspositionen ausgewertet. Die Werte stellen den linearen Mittelwert des Betrages des absoluten Fehlers und das Betragsmaximum des absoluten Fehlers geordnet nach den Zustandsgrößen dar.

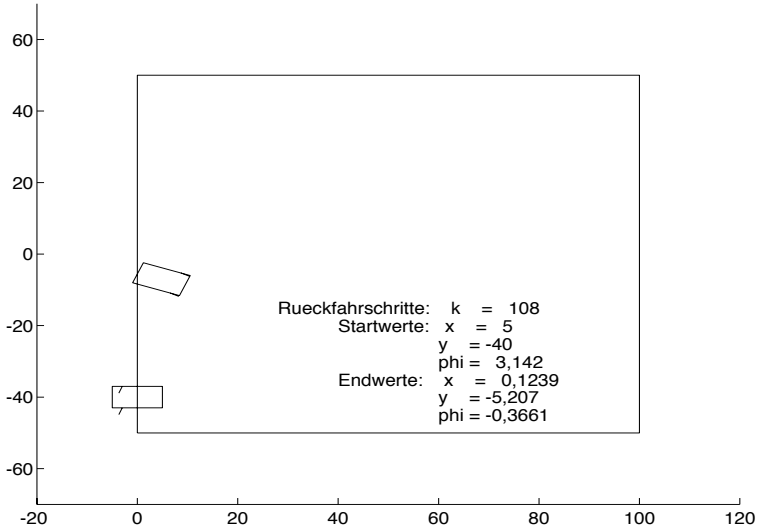

Abbildung 24: Anfangs- und Endwert eines Rückfahrvorganges

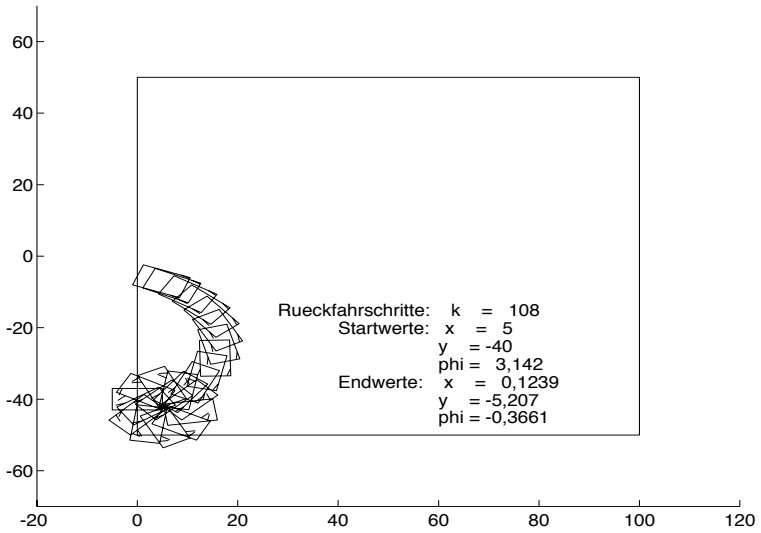

Abbildung 25: Ein Rückfahrvorgang des Autos

4.4 Die Regelung eines Sattelschleppers

4.4.1 Einführende Bemerkungen

Die Regelstrecke nach Abschnitt 4.3 wird nun dahingehend erweitert, daß ein Anhänger gemäß Abbildung 26 an das Auto gehängt wird. Diese Anordnung

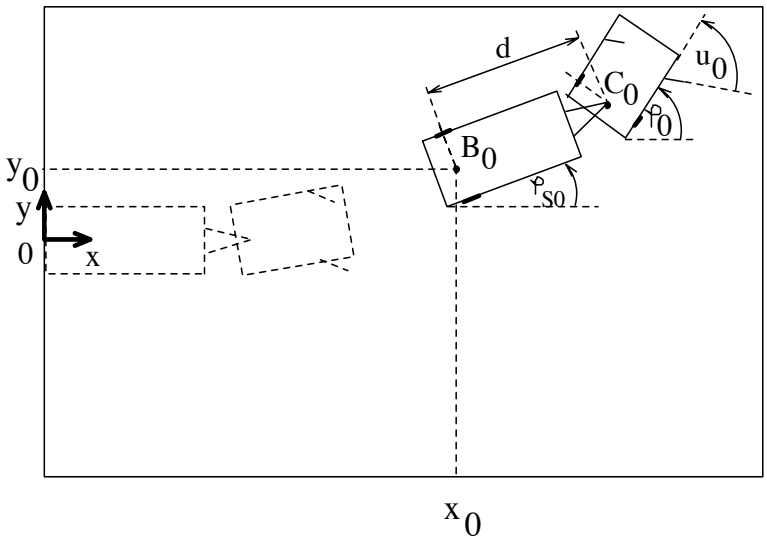

Abbildung 26: Ein Sattelschlepper als Beispiel einer nichtlinearen Regelstrecke

kann auch als ein Sattelschlepper interpretiert werden. Sie ist als Beispiel in [12] und [13] angeführt worden. Die Zustandsgrößen dieser Anordnung sind hier die Koordinaten des Punktes (x, y) und der Winkel φ_s des Hängers, sowie der Winkel der Zumaschine φ. Das Fahrzeug soll wie im letzten Beispiel nur rückwärts fahren und so von beliebigen Anfangspositionen in die Endposition, dem Koordinatenursprung, übergeführt werden. Das Stellsignal entspricht wiederum dem Einschlag der Vorderräder der Zugmaschine. Der Fehler bezüglich der Zustände (x, y, φ_s) soll im Endzustand minimal werden. Der Endzustand des Winkels φ der Zugmaschine interessiere hierbei nicht. Eine solche Problemstellung ergibt sich beispielsweise beim rückwärtigen Andocken eines Lastwagens an eine Laderampe.

Zum Entwurf des Reglers wird wieder zunächst ein mathematisches Modell der Strecke benötigt. Nach deren Nachbildung durch ein neuronales Netz wird der neuronale Regler stufenweise in Lektionen durch das Verfahren nach Abschnitt 3.4.1 angelernt. Zum Schluß dieses Abschnittes sind wieder die erzielten Regelergebnisse dargestellt.

4.4.2 Das Streckenmodell

Zur Herleitung der Modellgleichungen sei zunächst nur die Zugmaschine betrachtet. Deren Zustandsgleichungen entsprechen den Gleichungen des Autos nach Beziehung (47) bezüglich des Punktes C_0 in Abbildung 26. Setzt man nun für die Punkte P_0 und Q_0 aus Abbildung 22 die Punkte B_0 und C_0 aus Abbildung 26 ein, so lassen sich die Bewegungsgleichungen des Hängers unter Berücksichtigung der Beziehung

$$C_0 = B_0 + \begin{pmatrix} d\cos(\varphi_{s0}) \\ d\sin(\varphi_{s0}) \end{pmatrix}$$

analog zu den Überlegungen aus Abschnitt 4.3.2 herleiten. Insgesamt ergeben sich die Systemgleichungen zu

$$\begin{aligned}
x(k+1) &= x(k) - r\cos(\varphi_s(k))\cos(u(k))\cos(\varphi(k) - \varphi_s(k)) \\
y(k+1) &= y(k) - r\sin(\varphi_s(k))\cos(u(k))\cos(\varphi(k) - \varphi_s(k)) \\
\varphi(k+1) &= \arctan\left(\frac{l\sin(\varphi(k)) + r\cos(\varphi(k))\sin(u(k))}{l\cos(\varphi(k)) - r\sin(\varphi(k))\sin(u(k))}\right) \\
\varphi_s(k+1) &= \arctan\left(\frac{d\sin(\varphi_s(k)) - r\cos(\varphi_s(k))\cos(u(k))\sin(\varphi(k) - \varphi_s(k))}{d\cos(\varphi_s(k)) + r\sin(\varphi_s(k))\sin(u(k))\sin(\varphi(k) - \varphi_s(k))}\right)
\end{aligned} \tag{49}$$

Die zunächst verwendeten Bewegungsgleichungen für den Sattelschlepper nach [12] ergaben sich hierbei als fehlerhaft.

4.4.3 Die Strecke als neuronales Netz

Größe	Bereich	Normierung	normierter Bereich
u	$\{-\frac{7}{18}\pi \dots \frac{7}{18}\pi\}$	$\frac{7}{18}\pi$	$\{-1 \dots 1\}$
x	$\{0 \dots 100\}$	100	$\{0 \dots 1\}$
y	$\{-50 \dots 50\}$	50	$\{-1 \dots 1\}$
φ	$\{-\pi \dots \pi\}$	π	$\{-1 \dots 1\}$
φ_s	$\{-\pi \dots \pi\}$	π	$\{-1 \dots 1\}$

Tabelle 4: Liste der Wertebereiche der Ausgangszustände

Die Struktur des Emulators besteht aus fünf Eingangsneuronen (vier Zustandsgrößen und ein Stellsignaleingang), einer versteckten Schicht mit 45 Neuronen und einer Ausgangsschicht mit vier Neuronen. Als Aktivierungsfunktionen wurde wiederum der Tangenshyperbolicus gewählt. Ebenso wurde der BIAS-Term und ein Durchgriff verwendet. Der Anlernvorgang erfolgt ebenso wie in Abschnitt 4.3.3. Zum Erzeugen der Lernmuster wurde das

Größe	Bereich	Normierung	normierter Bereich
Δx	$\{-0,2 \dots 0,2\}$	$0,2$	$\{-1 \dots 1\}$
Δy	$\{-0,2 \dots 0,2\}$	$0,2$	$\{-1 \dots 1\}$
$\Delta\varphi$	$\{-0,031 \dots 0,031\}$	$0,04$	$\{-0,78 \dots 0,78\}$
$\Delta\varphi_s$	$\{-0,014 \dots 0,014\}$	$0,02$	$\{-0,71 \dots 0,71\}$

Tabelle 5: Liste der Wertebereiche Zustandsänderungen

Fahrzeug in $m = 728$ Anfangszustände gebracht. Die Wertebereiche der Anfangsstellungen zeigt Tabelle 4, deren Festlegung erfolgte in Anlehnung an [12]. Bei der Wahl der Anfangswerte ist ferner zu berücksichtigen, daß die Bedingung

$$|\varphi_s - \varphi| \leq \frac{\pi}{2}$$

eingehalten werden muß.
Die Wertebereiche der Ausgangsgrößen sind in Tabelle 5 dargestellt. Die Einstellungen sonstiger Größen sind der Tabelle 6 zu entnehmen.

Größe	Wert
Länge der Zumaschine l	6
Länge des Hängers d	14
Schrittweite r	0,2

Tabelle 6: Liste der verwendeten Variablen

4.4.4 Der Anlernvorgang des Reglers

Lektion	Schritte k	Anzahl Lernmuster m
1	20	50
2	90	20
3	300	20
4	450	20

Tabelle 7: Anzahl Schritte und Lernmuster der Lektionen 1 bis 4

Die Struktur des Regler entspricht derjenigen des Reglers aus Abschnitt 4.3.4, mit dem Unterschied, daß vier anstatt drei Eingangsneuronen verwendet werden. Ebenfalls wie nach Abschnitt 4.3.4 werden die Lernmuster der ersten vier

der insgesamt sechs Lektionen erzeugt. Das Vorwärtsfahren des Sattelschleppers wurde durch die Systemgleichungen (49) mit negativer Schrittweite r simuliert. Nähere Angaben zu der Anzahl der Schritte und Lernmuster je Lektion macht Tabelle 7, eine Darstellung der Lernmuster in Form der Stellungen des Sattelschleppers je Lektion zeigt Abbildung 27.

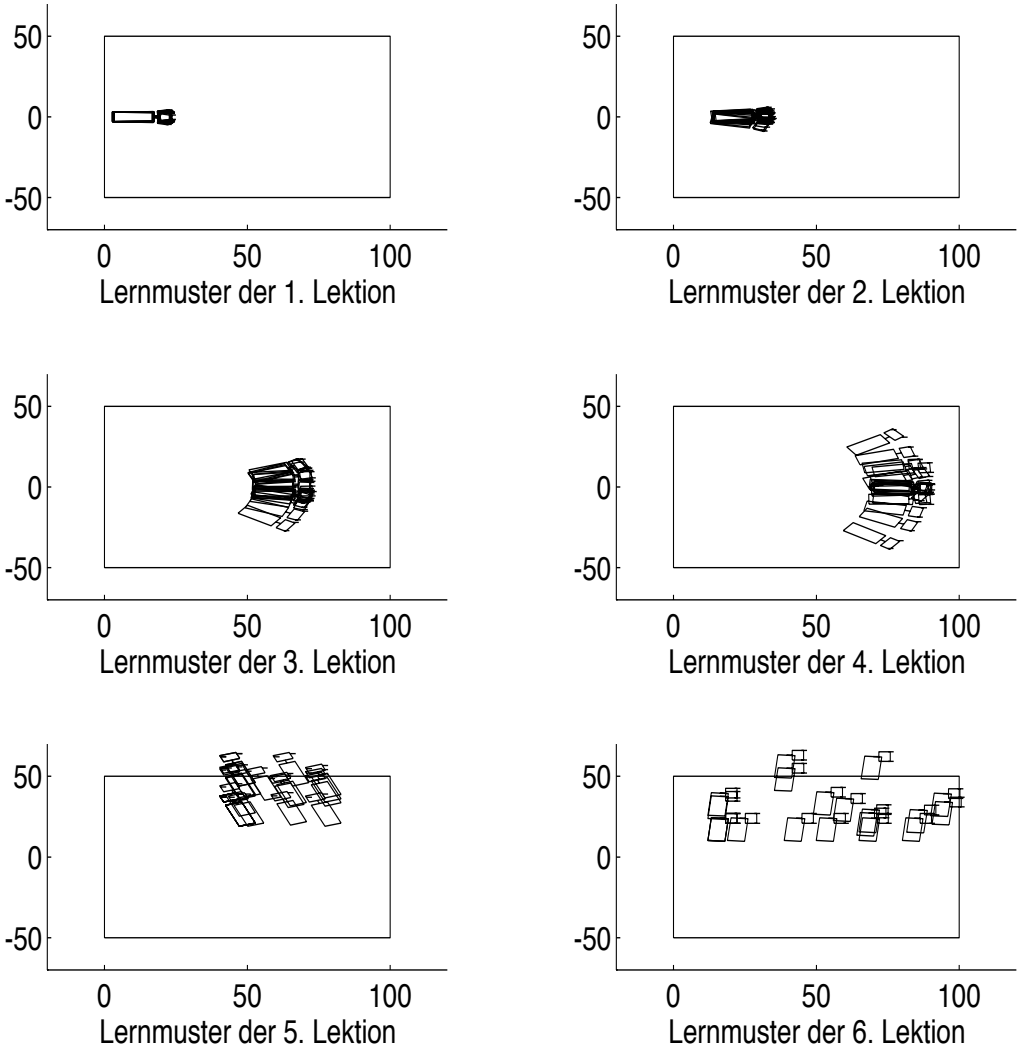

Abbildung 27: Darstellung der Lernmuster je Lektion

Die letzten zwei Lektionen bestanden aus zufälligen Anfangswerten, die der Regler noch nicht gelernt hatte. Sie wurden empirisch ermittelt.

4.4.5 Ergebnisse

Die Bilder 28 und 29 zeigen einen Rückfahrvorgang des Sattelschleppers, der sich beide Male in der gleichen Anfangsposition befindet. Wie bei allen folgenden Bildern ist nur jede 10. Position des Sattelschleppers dargestellt. Der verwendete Regler in Bild 28 wurde bis einschließlich Lektion 3 angelernt. Da bis zu dieser Lektion noch keine Lernmuster ähnlich der Anfangsposition in Abbildung 28 angelernt wurden, versagt die Regelung wie erwartet. Im Bild 29 hingegen wurde der Regler bis einschließlich Lektion 4 angelernt. Da nun auch Muster ähnlich der dargestellten Anfangsposition angelernt wurden, kann der Regler das Fahrzeug in die gewünschte Endposition überführen.

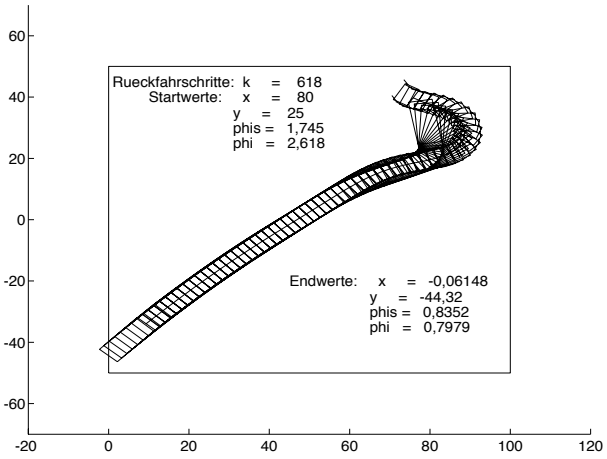

Abbildung 28: Rückfahrvorgang nach Lektion 3

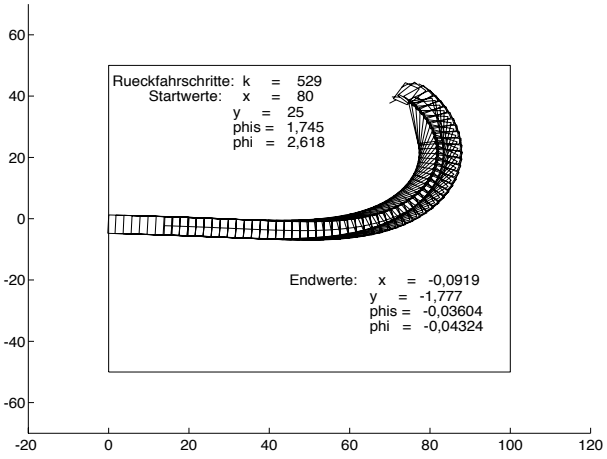

Abbildung 29: Rückfahrvorgang nach Lektion 4

Nach Beendigung der 6. Lektion war der Regler auch in der Lage, schwierigere Anfangszustände auszuregeln (siehe Bild 30). Die Bilder 31 und 32

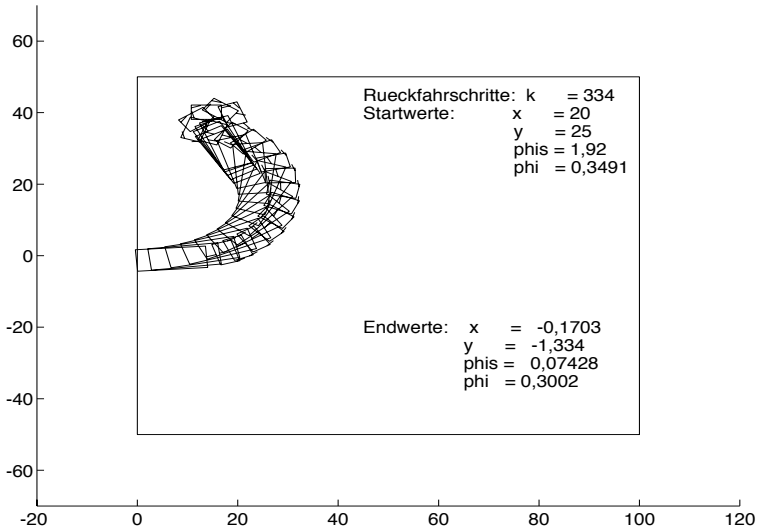

Abbildung 30: Ein Rückfahrvorgang nach Lektion 6

zeigen jedoch ebenso eine Anfangsposition, bei der die Regelung selbst nach Anlernen der 6. Lektion versagt.

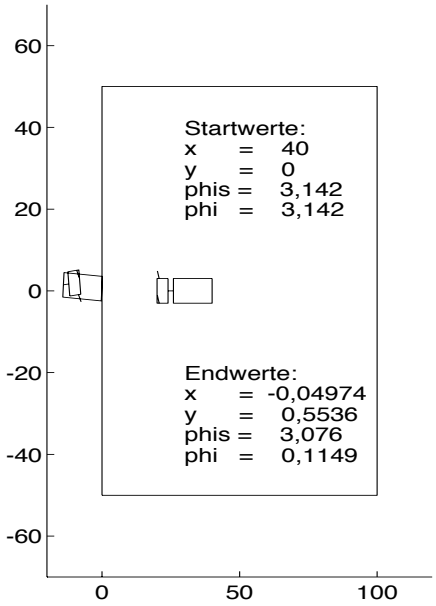

Abbildung 31: Start- und Endposition eines Rückfahrvorganges

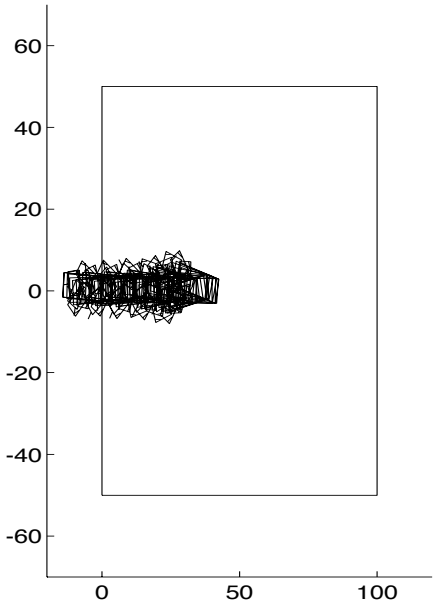

Abbildung 32: Rückfahrvorgang nach Lektion 6

Es wären zum Anlernen dieser Ausgangsstellung somit weitere Lektionen erforderlich, obwohl eine Vielzahl von Anfangspositionen bereits erfolgreich

ausgeregelt werden.

Zum praktischen Einsatz des Reglers müssen also zwei wichtige Punkte beachtet werden:

1. Es dürfen während der Regelung keine Zustände auftreten, die nicht in ähnlicher Form als Lernmuster vorgelegen haben.

2. Beim Anlernen des Reglers in Lektionen dürfen die Reglergewichte nur so verändert werden, daß Anfangspositionen bereits gelernter Lektionen auch weiterhin ausgeregelt werden. Der Regler darf also die bereits gelernten Zustände nicht wieder „verlernen“.

Der zweite Punkt kann einfach überprüft werden, in dem man die Regelergebnisse auch bezüglich bereits angelernter Anfangszustände überprüft. Der erste Punkt hingegen ist bedeutend schwieriger zu überprüfen. Man kann sich zwar dadurch behelfen, daß die letzte Lektion beim Anlernen des Reglers aus zufällig erzeugten Anfangswerten besteht. Bei komplizierteren Regelstrecken mit vielen Zustandsgrößen können jedoch Anfangszustände beim Anlernen unberücksichtigt bleiben, da durch den hohen Rechenaufwand die Anzahl der Stichproben der Anfangszustände stark beschränkt sind. Die Anzahl der Anfangszustände beim Anlernen einer Strecke vierter Ordnung lag hierbei beispielsweise zwischen 20 und 50 Zuständen (siehe Tabelle 7).

In der Abbildung 33 wurden die Abmessungen des Anhängers von

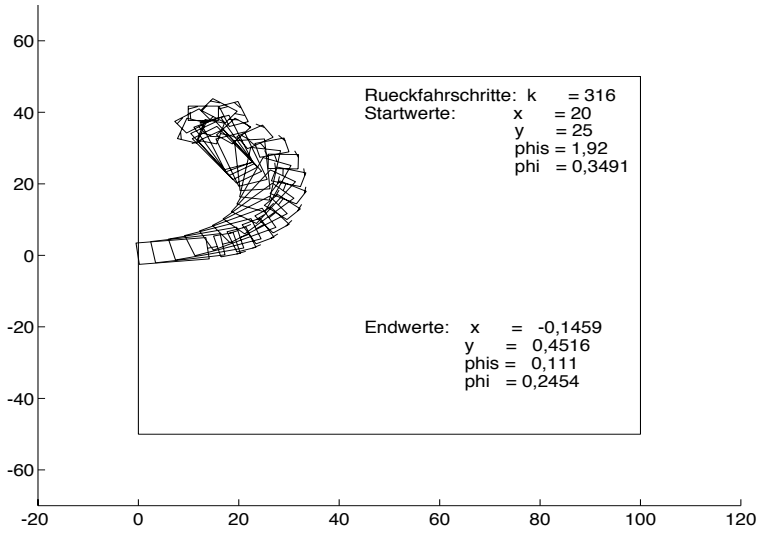

Abbildung 33: Rückfahrvorgang bei Hängerlänge $d = 10$

$d = 14$ auf $d = 10$ verändert und ein geregelter Rückfahrvorgang mit den

Startwerten nach Abbildung 30 durchgeführt. Gegenüber der Änderung der geometrischen Abmessungen des Hängers zeigte der Regler nach dem Anlernen der 6. Lektion ein robustes Verhalten. Der Rückfahrweg wich nach der Veränderung der Hängerlänge kaum vom ursprünglichen Rückfahrweg ab. Nach der Veränderung von d konnten wiederum nur solche Anfangszustände in den gewünschten Endwert $\underline{x}_e$ gebracht werden, die auch mit dem ursprünglichen Wert $d = 14$ erfolgreich geregelt werden konnten.

5 Zusammenfassung und Bewertung

Zunächst wurden in den Abschnitten des Kapitels 3 die Grundlagen des Verfahrens zum Reglerentwurf nach D. Nguyen und B. Widrow vorgestellt. Ausgehend von der Darstellung der verwendeten, neuronalen Netze wurde die Emulation der Regelstrecke in Form eines neuronalen Netzes dargelegt, die beim Reglerentwurf benötigt wird. Als verwendete Lernverfahren wurden der Backpropagation-Algorithmus und darauf aufbauend das Reglerentwurfsverfahren erörtert.

Als Beispiele dienten in Kapitel 4 Reglerentwürfe für Regelstrecken, deren Veranschaulichung aus dem Fahrzeugbereich stammten. Neben einem I_2-System ohne und mit Stellsignalbegrenzung wurden die geregelten Rückfahrvorgänge eines Autos und eines Sattelschleppers diskutiert. Dabei wurde bei dem I_2-System ein Zusammenhang des Reglerentwurfsverfahrens mit der zeitoptimalen Deadbeat-Regelung aufgezeigt. Bei Einführung einer Stellsignalbegrenzung dienten weiterhin allgemeine Überlegungen aus der Theorie der nichtlinearen Regelung zur Wahl der Anfangszustände zur prinzipiellen Lösbarkeit der Regelaufgabe.

Anhand der Ausführungen und Beispiele kann man erkennen, auf welche Art und Weise sich das Reglerentwurfsverfahren in bereits bekannte Methoden eingliedert. Bei Verwendung linearer Aktivierungsfunktionen im Regler kann ein Zustandsregler entstehen, dessen Rückführkoeffizienten durch Wahl der Abbruchbedingungen beim Anlernverfahren optimal ausgelegt werden können.
Das Verfahren selbst kann wiederum als eine geschickte Anwendung des Gradientenverfahrens betrachtet werden.

Ein neuer Aspekt bei diesem Reglerentwurf ist, daß der Verlauf des Überganges vom Ausgangszustand in den Endzustand im allgemeinen nicht in den Reglerentwurf mit eingeht. Dies ist darin begründet, daß eine Überprüfung des Fehlers immer erst dann durchgeführt wird, wenn der Systemzustand bestimmte Abbruchbedingungen erfüllt hat. Wie der Regler den Streckenemulator beziehungsweise die Strecke selbst in den Endzustand übergeführt hat, bleibt hierbei unberücksichtigt. Eingriffe in den Verlauf dieses Überganges kann man also auch nur indirekt durch geeignete Wahl der Abbruchbedingungen vornehmen. So kann man beispielsweise durch Festlegung einer maximalen Schrittzahl die Ausregelzeit begrenzen.

Der Regler ist bei der aufgezeigten Vorgehensweise also ein Gebilde, das aus einem stark abstrahierten, physikalischen Modell hervorgegangen ist. Werden bereits bei der mathematischen Modellierung eines Systems Fehler in Kauf

genommen, so ergibt sich durch die Nachbildung des mathematischen Modells durch ein neuronales Netz eine weitere Fehlerquelle. Dadurch besteht die Gefahr, daß der Regler im praktischen Einsatz nicht in der Lage ist, das physikalische System in der gewünschten Weise zu regeln. Es ist also immer nach dem Entwurf des Reglers noch zu überprüfen, ob der Regler neben dem Emulator auch die Strecke selbst in gewünschter Form regeln kann.
In der Praxis wird man daher versuchen, die Abstraktion beim Reglerentwurf zu reduzieren. Dies kann zum Beispiel dadurch erfolgen, daß die Lernmuster zum Entwurf des Streckenemulators nicht aus dem mathematischen Modell der Strecke, sondern aus Messungen am physikalischen Modell erzeugt werden.

Ein weiteres Problem ist, daß keine Aussagen über die Stabilität des Regelkreises gemacht werden können und bereits bei der Erzeugung der Lernmuster einige Überlegungen angestellt werden müssen, wie beispielsweise das Aufteilen der Lernmuster in für den Regler lernbare Lektionen. Die Lernmuster müssen also vor dem Entwurf „geschickt" gewählt werden.

Trotz dieser Nachteile bietet das Verfahren die Möglichkeit, gerade bei Regelstrecken niedrigerer Ordnung auf schnelle Art und Weise auch für nichtlineare Strecken einen Regler zu entwerfen, der die Strecke von einem Anfangszustand mit minimiertem Fehler in den Endzustand überführt. Bei entsprechender Wahl der Abbruchbedingungen kann das Verfahren ferner ein guter Ansatzpunkt zum Entwurf optimaler Regler sein.

6 Anhang (Die Erweiterungen des Netzwerksimulators)

6.1 Einführende Bemerkungen

Der Netzwerksimulator *fast* [1] (*fast* = forwiss artificial neural network simulation toolbox) ist ein am Institut für Regelungstechnik vorhandenes Programmpaket, das zum Anlernen von neuronalen Netzen verwendet wird. Es ist in der Programmiersprache C verfaßt und unter den Betriebssystemen UNIX und MS-DOS lauffähig. Das Anlernverfahren aus Abschnitt 3.4.1 ist auch in der Programmiersprache C umgesetzt und in *fast* eingebaut worden.

Die Art der Datenaufbereitung durch *fast* ist in Abbildung 34 schematisch dargestellt. Die zum Anlernen eines neuronalen Netzes erforderlichen Daten

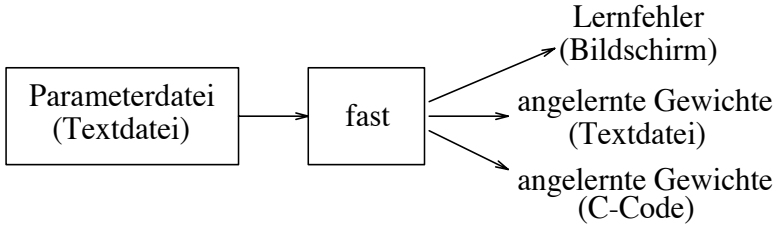

Abbildung 34: Datenaufbereitung des Netzwerksimulators *fast*

wie zum Beispiel der Netztyp, die Anzahl der versteckten Schichten, der Lernrate und so weiter werden über eine Parameterdatei eingelesen. Während des Anlernvorganges wird der aktuelle Lernfehler auf dem Bildschirm angezeigt. Nach Beendigung des Lernzyklus werden dann die Netzwerkgewichte in Form einer Textdatei oder eines Unterprogrammes in der Programmiersprache C abgelegt.

Der Reglerentwurf erfolgt nun wieder in zwei Schritten. Zunächst wird mit *fast* ein Backpropagation-Netz zur Nachbildung der Strecke angelernt und die angelernten Gewichte in Form einer Textdatei gespeichert. Im zweiten Schritt wird nach Laden dieser Emulatorgewichte wiederum mit *fast* der gewünschte Regler angelernt.

Aus Bild 34 ergibt sich für die weiteren Abschnitte folgende Gliederung:

1. Beschreibung der Parameterdatei und deren Erweiterungen
2. Strukturbeschreibung des Netzwerksimulators *fast* mit den neuen Programmodulen
3. Beschreibung der Gewichtsdatei (Textdatei) und deren Veränderungen

6.2 Die Parameterdatei

6.2.1 Die Funktionsweise

Ein einfaches Beispiel einer Parameterdatei, die das XOR-Problem (siehe [4]) beschreibt, zeigt Abbildung 35. Ein Befehl besteht aus der Anwei-

```
# Parameterdatei fuer das XOR-Problem mit 2 versteckten Neuronen
#
ANNType Backprop                  # Netztyp = Backpropagation-Netz
NInputs 2  NHidden 2  NOutputs 1 # 2 Eingaenge; erste versteckte
                                  # Schicht mit 2 Neuronen;
                                  # 1 Ausgangsneuron
LearningRate 0.2                  # Lernrate
ReportUpdate 5                    # Lernfehler nach jeweils
                                  # 5 Epochen anzeigen
Training                          # Trainingsmuster
0 0     -0.5
0 1      0.5
1 0      0.5
1 1     -0.5
```

Abbildung 35: Beispiel einer Parameterdatei

sung (Keyword) und meistens aus noch einem zugehörigen Parameter. So zeigt beispielsweise die Anweisung `ANNType` an, daß der folgende Parameter den Netzwerktypen bezeichnet, in dem Beispiel also ein mehrschichtiges Backpropagation-Netz. Werden Anweisungen weggelassen, so werden notwendige Größen intern in *fast* mit Standardgrößen vorbelegt. Diese sind in der Datei int.c festgelegt und können entsprechend verändert werden. Danach muß das Programmpaket *fast* allerdings neu in ausführbaren Maschinencode übersetzt werden. Eine vollständige Liste aller Anweisungen außer den hier hinzugefügten Erweiterungen ist in [1] zu finden.

6.2.2 Die Erweiterungen

6.2.2.1 Vorgehensweise bei den Erweiterungen Zur Realisierung des Anlernvorganges des neuronalen Reglers ist es erforderlich, neue Anweisungen in der Parameterdatei zuzulassen. Diese Anweisungen müssen zunächst vom Netzwerksimulator *fast* verarbeitet werden können. Hierfür ist eine Erweiterung der Eingabefunktionen des Programmes *fast* erforderlich. Die Vorgehensweise bei Erweiterungen des Befehlssatzes der Parameterdatei sei nun beispielhaft am Einbau der Anweisung `EmulatorMaxIter` aufgezeigt. Dieser Wert entspricht der Größe k_{max} in Abschitt 3.4.1.

Durch die weitgehend globale Festlegung der Variablen in *fast* müssen Einträge in mehreren Dateien vorgenommen werden. Zunächst muß in der Datei par.h in die alphabetisch geordnete Liste die Anweisung

```
{"EmulatorMaxIter", INT, CB, (VOID)&EmulatorMaxIter}
```

eingetragen werden.
`"EmulatorMaxIter"` entspricht der Anweisung, `INT` bedeutet, daß die Anweisung einen ganzzahligen Parameter besitzt. `CB` bedeutet, daß der Befehl auftreten darf, wenn `ANNType ControllerBackprop` ist. Diese Option wird immer dann in der Parameterdatei angegeben sein, wenn ein Regler nach Abschnitt 3.4.1 angelernt werden soll. Eine entsprechende Definition von `CB` ist in der Datei fst.h zu finden. `(VOID)&EmulatorMaxIter` ordnet der Anweisung eine Variable mit dem Namen `EmulatorMaxIter` zu. Die Deklaration dieser Variable muß in der Datei int.c mit der Anweisung

```
int EmulatorMaxIter = 1000;
```

durchgeführt werden, wobei 1000 der Vorbelegung entspricht. Damit alle Programmodule auf diese Variable zugreifen können, muß abschließend in der Datei var.h der Eintrag

```
extern int EmulatorMaxIter;
```

durchgeführt werden.

6.2.2.2 Die Liste der Erweiterungen Die folgenden Tabellen 8 und 9 enthalten die zur Realisierung des Verfahrens nach Abschnitt 3.4.1 nötigen Erweiterungen bezüglich der Befehle, die in der Parameterdatei anzugeben sind. Die Auflistung erfolgt in Anlehnung an [1] und ist als eine Ergänzung der dort aufgeführten Tabelle zu betrachten. Falls zu einem Befehl neben der Anweisung noch ein Parameter benötigt wird, ist dies durch die Art des Parameters in Klammern hinter der Anweisung vermerkt. Beispielsweise bedeutet der Befehl

```
EmulatorMaxIter <int>,
```

daß die Anweisung EmulatorMaxIter eine ganzzahligen Parameter benötigt. Die mit `(*)` markierten Befehle werden im allgemeinen nicht mehr explizit im Parameterfile angegeben. Da der Emulator bereits bei einem ersten

Anlerndurchlauf in Form einer Gewichtsdatei im Textformat erstellt wurde, werden die mit `(*)` gekennzeichneten Informationen mit abgespeichert. Weitere Erklärungen hierzu finden sich im Abschnitt 6.4.1.

Durch Hinzufügen eines neuen Parameters für die Netzstruktur der Form `ControllerBackprop` kann nun über die Anweisung

`ANNType ControllerBackprop`

in der Parameterdatei ein Regler nach Abschnitt 3.4.1 entworfen werden. Das Hinzufügen dieses neuen Parameters erfolgte hierbei durch Deklaration und Listeneintrag in den Dateien fst.h und par.h.
Weiterhin wurde die Tangenshyperbolicus-Funktion für die Aktivierungsfunktionen der versteckten Neuronen und der Ausgangsneuronen realisiert. Der Aufruf erfolgt im Parameterfile mit der Parameterangabe:

`Tanh`

Der Parameter kann sowohl für die Emulatorneuronen, als auch für andere Netze wie beispielsweise das des Reglers verwendet werden. Der Parameter ist als eine Ergänzung der Parameter in [1], Seite 9 zu verstehen.

	Befehl	Beschreibung
	`EmulatorErrorWeights <real>`	entspricht den α_ν in Gleichung (17) Voreinstellung: 1.0
	`EmulatorMaxIter <int>`	entspricht k_{max} in Abschnitt 3.4.1 Voreinstellung: 1000
	`EmulatorOutputBounds <real>`	gibt die Unter- bzw. Obergrenze der Wertebereiche an, in denen sich die Zustandsgrößen bewegen dürfen Voreinstellung: $-10^5 \quad 10^5$
	`EmulatorOutputScale <real>`	gibt die k_ν nach Gleichung (18) an Voreinstellung: 1.0
`(*)`	`EmulatorOutputType <string>`	legt die Aktivierungsfunktion der Ausgangsneuronen des Emulators fest; zulässige Typen siehe [1] Voreinstellung: Sigmoid
	`EmulatorPassThrough <boole>`	gibt die Verwendung eines Durchgriffes bei Realisierung des Emulators an (siehe 3.3.3) Voreinstellung: False

Tabelle 8: Liste der Erweiterungen der Parameterdatei

EmulatorStatesDontCare <int>	gibt an, für welche Zustandsgrößen eine Überprüfung des Wertebereiches erfolgt; 0 = Überprüfung des Wertebereiches 1 = keine Überprüfung Voreinstellung: 0
(*) EmulatorUnitType <string>	bestimmt die Aktivierungsfunktion der versteckten Neuronen des Emulators; zulässige Typen siehe [1] Voreinstellung: Sigmoid
(*) EmulatorUseBiasUnit <string>	gibt an, ob der BIAS-Term im Emulator verwendet werden soll Voreinstellung: True
EmulatorWeightFileIn <string>	Laden der Gewichte des Emulators, die in der Datei <string> abgespeichert sind
(*) NEmulatorHidden[1-5] <int>	Anzahl der Neuronen des Emulators je versteckter Schicht Voreinstellung: 0
(*) NEmulatorInputs <int>	Anzahl der Eingänge des Emulators
(*) NEmulatorOutputs <int>	Anzahl der Ausgänge des Emulators

Tabelle 9: Liste der Erweiterungen der Parameterdatei (Fortsetzung)

6.2.2.3 Die Erweiterungen am Beispiel des Sattelschleppers

Beispielhaft ist in Abbildung 36 die Parameterdatei zum Anlernen der ersten Lektion des Reglers für den Sattelschlepper dargestellt. Wie beim Anlernen anderer Netze muß auch hier die erste Anweisung das Lernverfahren festlegen. Danach werden die Angaben zum Emulatornetz über die Gewichtsdatei `emultruck.w` eingelesen. Die Angaben zum Reglernetz erfolgen in gleicher Weise wie beim Anlernen anderer Netze. Zu beachten ist, daß die Angabe der Skalierungen $k_{aus\nu}$ aus Abschnitt 3.4.2 in einer der Anweisung `EmulatorOutputScale` folgenden Zeile erfolgen muß. In der gleichen Weise werden die α_ν aus Abschnitt 3.2.3 nach der Anweisung `EmulatorErrorWeights` angegeben. Das Gleiche gilt, falls benötigt, für die Angaben bezüglich der Anweisung `EmulatorStatesDontCare`. Die Wertebereiche der Zustandsgrößen hingegen werden zeilenweise nach der Anweisung `EmulatorOutputBounds` entsprechend Bild 36 angegeben.
Die Syntax der restlichen Befehle ergibt sich entsprechend den Ausführungen in [1].

```
# Parameterdatei zum Anlernen des Reglers fuer den
# Sattelschlepper; Lektion Nummer 1
#
ANNType ControllerBackProp          # Lernverfahren = Verfahren nach
                                    # D. Nguyen und B. Widrow
EmulatorWeightFileIn emultruck.w    # Gewichtsdatei des Emulators der
                                    # Regelstrecke einlesen
EmulatorMaxIter 500                 # = kmax
NInputs 4 NHidden 25 NOutputs 1     # Angaben zur Reglerstruktur:
                                    # 4 Eingaenge; erste versteckte
                                    # Schicht mit 25 Neuronen;
                                    # 1 Ausgangsneuron
WeightFileOut wcont1.dat            # Ausgeben der Reglerparameter
                                    # nach dem Anlernen in Textdatei
ReportUpdate 1                      # Lernfehler nach jeder
                                    # Epoche anzeigen
UnitType Tanh                       # Aktivierungsfunktionen des Reglers
OutputType Tanh                     # = Tangenshyperbolicus
EmulatorPassThrough True            # Verwendung eines Durchgriffes beim
                                    # Regler

EmulatorOutputScale                 # Angabe der Skalierungen bezueglich
.002 0.004 0.00636619772368 0.01273239544735 # x, y, phis, phi

EmulatorOutputBounds                # Wertebereiche der Zustandsgroessen
 0 1                                # min. max. Wert des 1. Zustandes x
-1 1                                # min. max. Wert des 2. Zustandes y
-1 1                                # min. max. Wert des 3. Zustandes phis
-1 1                                # min. max. Wert des 4. Zustandes phi

EmulatorErrorWeights                # = alpha1 ... alpha4
1 1 1 0

MaxIter 100                         # Anzahl Epochen
Momentum 0.9                        # Momentum-Term
LearningRate 0.9                    # Lernrate

Training lesson1p.dat               # Trainingsmuster in Datei lesson1p.dat
```

Abbildung 36: Parameterdatei zum Anlernen eines Reglers

6.3 Der Netzwerksimulator *fast*

6.3.1 Struktur und Funktionsweise

Durch den großen Umfang des Programmpaketes wird an dieser Stelle nur die prinzipielle Struktur und die von der Änderung betroffenen Module des Netzwerksimulators dargestellt. Nach den folgenden Ausführungen sollte es jedoch prinzipiell möglich sein, neuerliche Veränderungen am Programm vorzunehmen.

Das Hauptprogramm main() in der Datei fst.c gliedert sich in drei grobe Blöcke nach Abbildung 37. Die Eingaben werden von der Funktion pro-

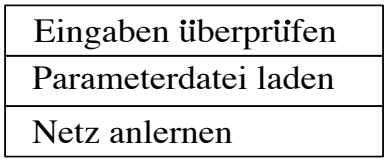

Abbildung 37: Struktur der Hauptroutine main()

cess_cmd_arguments() in fst.c überprüft. Sind diese korrekt, wird über die Funktion load_param_file() in dio.c die Funktion process_line() aktiviert, die zeilenweise die Parameterdatei einliest und analysiert. Anschließend wird die Funktion train_test_and_save() in der Datei cor.c aktiviert, die eigentliche Hauptroutine zum Anlernen des Netzes. Ihre Struktur zeigt Abbildung 38. Nach der Initialisierung der Neuronengewichte und dem Aufbau der Netz-

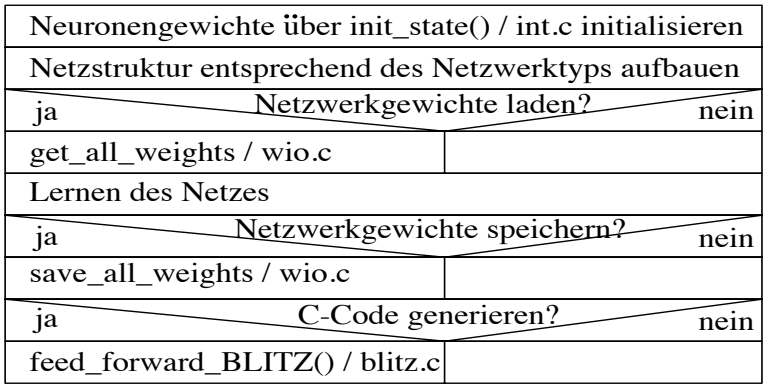

Abbildung 38: Struktur der Funktion train_test_and_save()/cor.c

struktur je nach Netzwerktyp erfolgt nach dem optionalen Laden der Netzwerkgewichte der Anlernvorgang des Netzes. Nach dem Anlernen können wahlweise die Netzwerkgewichte in Form einer Textdatei oder eines C-Codes abgespeichert werden.

Die größten Veränderungen und Ergänzungen wurden neben dem Laden der Parameterdatei und dem Abspeichern des Netzwerkgewichte im Aufbau der Netzstruktur und dem Lernvorgang des Netzes gemäß Abschnitt 3.4.1 vorgenommen. Die meisten neuen Routinen finden sich dabei in der neu erstellten Datei emul.c wieder. Im folgenden soll nun der Aufbau der Netzstruktur und der Anlernvorgang für den Fall des Reglerentwurfes näher betrachtet werden.

6.3.2 Struktur und Funktion der Erweiterungen

Die Struktur des Aufbauvorganges des Regler- und des Emulatornetzes gibt Abbildung 39 wieder. Zunächst wird in den Funktionen build_prop_net() und

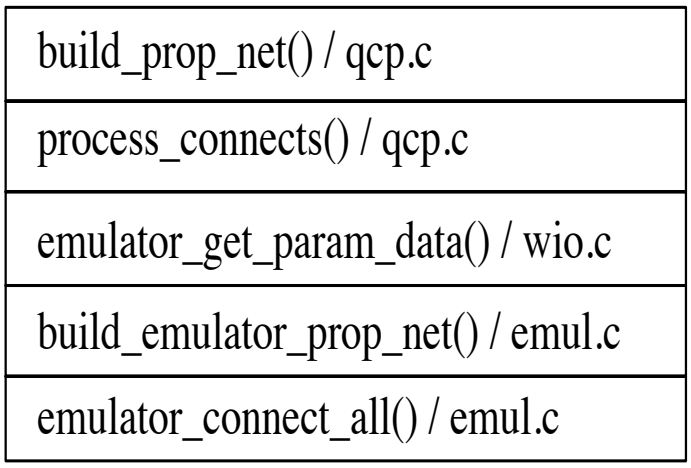

Abbildung 39: Struktogramm zum Aufbau der Netzstruktur

in process_connects() in der Datei qcp.c der Speicher für das Reglernetz reserviert und die zugehörige Struktur des Reglers aufgebaut. Anschließend werden in der Funktion emulator_get_param_data() in wio.c die Strukturinformationen des Emulators der Regelstrecke aus der Gewichtsdatei des Emulators entnommen. Der genaue Aufbau dieser Datei ist in Abschnitt 6.4.1 näher erläutert. In der Funktion build_emulator_prop_net() in emul.c wird dann der Speicher für das Emulatornetz reserviert und gegebenenfalls noch nicht vorbelegte Einstellungen auf ihre Standardwerte vorbelegt. Die Verbindung der einzelnen Schichten des Emulatornetzes werden abschließend in der Funktion emulator_connect_all() hergestellt, die sich ebenfalls in der Datei emul.c befindet.

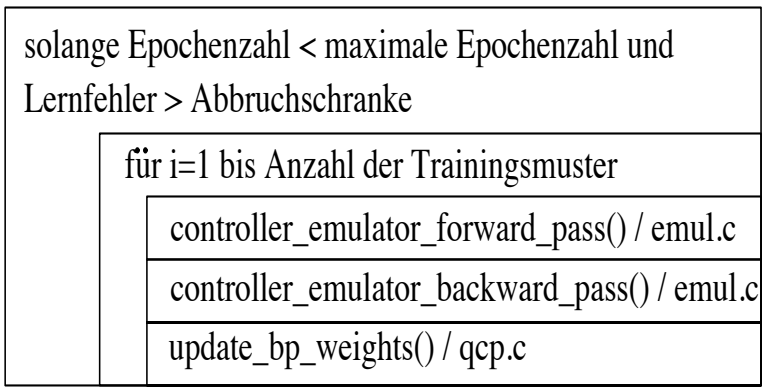

Abbildung 40: Struktur des Anlernvorganges des Reglers

Die Struktur des Anlernvorganges des Reglers zeigt Abbildung 40. Sie ist in der Funktion quickprop_main() in qcp.c zu finden. Je Epoche und Eingangsmuster werden zunächst die Eingangswerte des Lernmusters durch den Regler und den Emulator vorwärtsvermittelt. Dies geschieht durch die Funktion controller_emulator_forward_pass() in emul.c, deren genauere Struktur in Abbildung 41 dargestellt ist. Zunächst wird in der Funktion forward_pass()

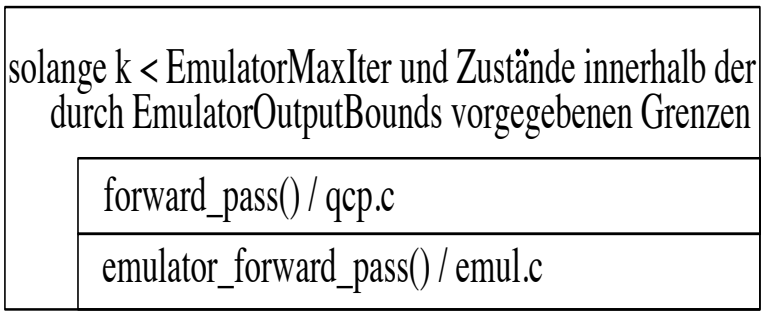

Abbildung 41: Struktur des Feedforward-Schrittes

in qcp.c das Stellsignal aus dem Eingangsmuster ermittelt. Anschließend wird das Stellsignal zusammen mit dem gleichen Eingangsmuster an emulator_forward_pass() in emul.c übergeben und daraus der neue Zustand berechnet. Bei diesem Vorgang der Vorwärtsvermittlung werden die Aktivierungen des Reglers und des Emulators bei jedem Schritt in der Matrix OutputArray gespeichert. Diese müssen vor jedem Backpropagation-Schritt geladen werden. Die zweite Möglichkeit, nur die Systemzustände zu speichern und die restlichen Aktivierungen vor jedem Backpropagation-Schritt durch einen Feedforward-Schritt über controller_emulator_forward_pass() zu bestimmen, wurde verworfen, da der zeitliche Aufwand im Vergleich zur Verringerung des Speicherplatzes unverhältnismäßig groß war. Der Vorgang der Vorwärtsvermittlung endet, wenn die maximal zulässige Anzahl an Schritten erreicht

ist, oder die Zustände ihre Definitionsbereiche verlassen.

Der Backpropagation-Schritt wird in Abbildung 40 durch die Funktion controller_emulator_backward_pass() abgearbeitet. Eine genauere Struktur dieser Funktion zeigt Abbildung 42. Zunächst wird für alle Ausgangszustände

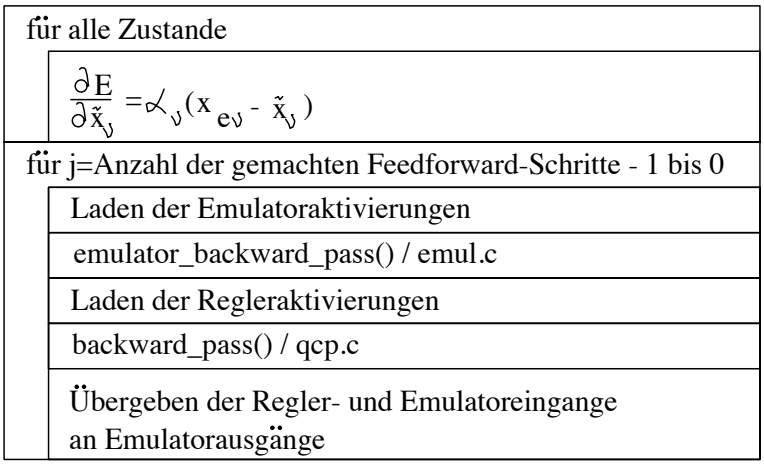

Abbildung 42: Struktur des Backpropagation-Schrittes

$\frac{\partial E}{\partial \tilde{x}_\nu}$ gemäß Gleichung (17) berechnet. Anschließend werden iterativ zunächst die Emulatoraktivierungen geladen und ein Backpropagation-Schritt durch das Emulatornetz vorgenommen. Daraufhin wird der gleiche Vorgang für den Regler durchgeführt und die entstehenden, partiellen Ableitungen des Fehlers E an den Regler- und Emulatoreingängen an die Emulatorausgänge weitergereicht. Dieser Vorgang wiederholt sich entsprechend der Anzahl der durchgeführten Feedforward-Schritte.

Nach Beendigung von controller_emulator_backward_pass() werden die Gewichte des Reglernetzes mit der Funktion update_bp_weights() in qcp.c gemäß den Beziehungen (8) und (9) verändert.

Der Lernalgorithmus bricht ab, wenn die maximale Anzahl der vorgegebenen Epochen erreicht ist, oder der Lernfehler gemäß Gleichung (17) unter eine festgelegte Abbruchschranke gesunken ist. In der Parameterdatei wird die maximale Anzahl der Epochen über `EmulatorMaxIter` und die Abbruchschranke des Lernfehlers über `ErrorThresHold` festgelegt.

6.4 Die Veränderungen der Gewichtsdatei

6.4.1 Allgemeine Bemerkungen

Das Verfahren nach Abschnitt 3.4.1 benötigt einen Emulator der Regelstrecke in Form eines neuronalen Netzes. Da dieser Emulator im allgemeinen auch zunächst angelernt werden muß, ist es naheliegend, auch hierfür den Netzwerksimulator *fast* zu verwenden. Damit die Information, die zum Anlernen des Emulators notwendig waren, nicht von Neuem wieder in die Parameterdatei zum Anlernen des Reglers aufgenommen werden muß, wird die Emulatorstruktur zusammen mit den angelernten Gewichten in einer Gewichtsdatei abgespeichert. Damit die bereits existierenden Routinen des Netzwerksimulators die Gewichtsdatei wie bisher abarbeiten können, sind die zusätzlichen Informationen in den Kommentarzeilen zu Beginn der Datei eingefügt worden. So kann die angelegte Gewichtsdatei sowohl für die neueren Funktionen zur Beschreibung der Emulatorstruktur, als auch für die alten Funktionen des Simulators verwendet werden.
Sind die Gewichte des Emulators bereits bekannt oder mit anderen Programmpaketen ermittelt worden, so können die Gewichte nachträglich in der von *fast* angelegten Gewichtsdatei modifiziert werden. Dies ist in dieser Arbeit beispielsweise für den Fall der linearen Regelstrecke durchgeführt worden.

Zur Erläuterung sind im folgenden die zusätzlichen Informationen, die in der Gewichtsdatei angelegt werden, beispielhaft für die Regelstrecke aus Abschnitt 4.2 angegeben worden.

6.4.2 Am Beispiel der linearen Strecke

Die zusätzlich in die Gewichtsdatei eingefügten Information für den Emulator nach Abschnitt 4.2 ist in Abbildung 43 dargestellt. Das erste Doppelkreuz zu

```
## NEmulatorInputs 3
## NEmulatorHidden1 0
## NEmulatorHidden2 0
## NEmulatorHidden3 0
## NEmulatorHidden4 0
## NEmulatorHidden5 0
## NEmulatorOutputs 2
## EmulatorUnitType Linear
## EmulatorOutputType Linear
## EmulatorUseBiasUnit False
```

Abbildung 43: Beispiel der Erweiterungen einer Parameterdatei

Beginn einer jeden Zeile wird von der Funktion emulator_get_param_data() in wio.c als ein Kommentarzeichen erkannt. Folgt darauf ein weiteres Kommentarzeichen, so wird die restliche Zeile als Anweisung zur Bestimmung der Emulatorstruktur interpretiert. Im Beispiel nach Bild 43 ist die Emulatorstruktur ein neuronales Netz mit drei Eingängen, keiner versteckten Schicht und zwei Ausgangsneuronen. Die Aktivierungsfunktionen aller Neuronen sind linear, das heißt das Summationsergebnis $s(\underline{x})$ aus Abbildung 2 liegt direkt am Ausgang eines jeden Neurons an. Weiterhin ist der BIAS-Term des Emulators abgeschaltet.

Die eben genannten Strukturmerkmale werden in jeder Gewichtsdatei abgespeichert und sind nach deren Einlesen in die Parameterdatei zum Anlernen des Reglers nicht mehr explizit anzugeben.

6.5 Optionen beim Aufruf von *fast*

Neue Optionen sind nicht hinzugefügt worden. Die Optionen beim Aufruf von *fast* zum Anlernen eines Reglers entsprechen der Beschreibung der Optionen in [1]. Ist beispielsweise die Datei `cont.net` eine Parameterdatei zum Anlernen eines Reglers, so kann der C-Code des Reglers über den Aufruf

```
fast -b cont.net
```

erzeugt werden. Der Name des erzeugten C-Codes des Reglers ist wie alle anderen von *fast* erzeugten C-Codes `blitz.c`, und er kann über die Datei `faster.c` an Matlab angebunden werden.

7 Verzeichnis der verwendeten Abkürzungen und Zeichen

M	Stellsignalbegrenzung
T	Abtastzeit bei Diskretisierung einer Strecke
$F, \tilde{F}$	zeitdiskrete, zeitkontinuierliche Abbildung der Zustände durch die Strecke
$f()$	Aktivierungsfunktion eines Neurons
k_{max}	maximale Anzahl an Schritten beim Entwurfsverfahren nach Abschnitt 3.4.1
$\tilde{A}, \tilde{b}$	zeitkontinuierliche Zustandsmatrizen
A, b	zeitdiskrete Zustandsmatrizen
$\hat{A}$	zeitdiskrete Systemmatrix, bei der bereits das Regelgesetz $u(k) = -\underline{k}^T \underline{x}(k)$ berücksichtigt wurde
$\underline{x}_0, \underline{x}_k$	Anfangszustand, Endzustand eines Regelvorganges
$\underline{x}_e$	Sollwert des Streckenzustandes
n	Ordnung der Regelstrecke
m	Anzahl der Lernmuster
$n()$	Abbildungsoperator eines neuronalen Netzes
o_ν	Aktivierung des ν-ten Neurons
$w_{\mu\nu}$	Verbindungsgewicht zwischen dem Ausgang des ν-ten und dem Eingang des μ-ten Neurons
$\underline{w}$	Gewichtsmatrix eines neuronalen Netzes

Literatur

[1] Arras M., Protzel P.:
Forwiss Artificial Neural Network Simulation Toolbox, User's Guide, Erlangen 1993

[2] Bronstein I. N. et al.:
Taschenbuch der Mathematik, Verlag Harri Deutsch, Thun 1987

[3] Fahlman S. E., Lebiere C.:
The Cascade-Correlation Learning Architecture, Thesis, Pittsburgh 1991

[4] Firsching P.:
Neuronale Netze in der Regelungstechnik, Institutsbericht 1/92 am Institut für Regelungstechnik, Universität Erlangen-Nürnberg, 1992

[5] Föllinger O.:
Optimierung dynamischer Systeme, Oldenbourg Verlag, München 1985

[6] Hippe P., Wurmthaler Ch.:
Zustandsregelung, Springer-Verlag, Berlin, Heidelberg 1985

[7] Isermann R.:
Digitale Regelsysteme, Band I, Springer-Verlag, Berlin 1988

[8] Kernighan B. W. et al.:
The C Programming Language, Prentice Hall, New Jersey 1988

[9] Kratzer K. P.:
Neuronale Netze, Carl Hanser Verlag, München 1991

[10] Kučera V.:
Analysis and Design of Discrete Linear Control Systems, Prentic-Hall, London 1991

[11] Kong S. und Kosko B.:
Adaptive Fuzzy Systems for Backing up a Truck-and-Trailer, IEEE Transactions on Neural Networks, Vol. 3, No. 2, März 1992

[12] Miller W. T. et al.:
Neural Networks for Control, The MIT-Press, Cambridge, Massachusetts 1992

[13] Nguyen D. und Widrow B.:
Neural Networks for Self-Learning Control Systems, Int. J. Control 1991, Vol. 54, No. 6, 1439-1451

[14] Ritter H. et al.:
Neuronale Netze, Addison-Wesley, München 1991

[15] Rumelhart D.E., McClelland J.:
Parallel Distributed Processing, The MIT-Press, Cambridge, Massachusetts, 1988

[16] Schlitt H.:
Regelungstechnik, Vogel Buchverlag, Würzburg 1988

[17] The MathWorks, Inc.:
Matlab for UNIX Workstations, User's & Reference Guide,
South Natick 1992

Programme

Die folgende Auflistung gibt eine Übersicht über die erstellten Programme. Sie gliedert sich in zwei Abschnitte.

Der erste Teil erklärt die Funktion der für die Beispiele benötigten Matlab-Programme und die dafür von *fast* benötigten und erzeugten Dateien.

Der zweite Teil gibt eine Auflistung der Dateien des Netzwerksimulators, die verändert wurden. Die in der Datei emul.c enthaltenen C-Module sind explizit angegeben.

TEIL 1: FUNKTIONSBESCHREIBUNG DER FUER DIE BEISPIELE BENOETIGTEN MATLAB-PROGRAMME UND DER DAFUER VON fast BENOETIGTEN UND ERZEUGTEN DATEIEN

1.) Programme zur Durchfuehrung der Deadbeat-Regelung einer linearen Strecke

Verzeichnis: ml/car1/deadbeat

Dateien:	Funktionsbeschreibung:
blitz.c	angelernte Gewichte des Reglers in Form eines C-Moduls
controllerdeadbeat.mexhp7	Datei zur Anbindung des Reglers an Matlab
deadbeat.m	Erzeugung der Trainings- und Testmuster zum Anlernen des Reglers unter Matlab
deadbeat.net	Parameterdatei zum Anlernen des Reglers mit fast
emdeadbeat.dat	Gewichtsdatei des Emulators
emuldeadbeat.m	mathematisches Modell des Fahrverhaltens des Autos in Form einer Zustandsgleichung
faster.c	C-Modul zur Erzeugung der Datei controllerdeadbeat.mexhp7
phasedeadbeat.m	graphische Darstellung eines geregelten Rueckfahrvorganges unter Matlab
testdeadbeat.dat	Datei der Testmuster zum Anlernen des Reglers
traindeadbeat.dat	Datei der Trainingsmuster zum Anlernen des Reglers
wdeadbeat.dat	angelernte Gewichte des Reglers in Textform

2.) Programme unter Beruecksichtigung einer Stellsignalbegrenzung

Verzeichnis: ml/car1/limited

Dateien:	Funktionsbeschreibung:
blitz.c	angelernte Gewichte des Reglers in Form eines C-Moduls
contlimit.mexhp7	Datei zur Anbindung des Reglers an Matlab
limited.m	Erzeugung der Trainings- und Testmuster zum Anlernen des Reglers unter Matlab
limited.net	Parameterdatei zum Anlernen des Reglers mit fast
emlimited.dat	Gewichtsdatei des Emulators
emullimit.m	mathematisches Modell des Fahrver-

	haltens des Autos in Form einer Zustandsgleichung
faster.c	C-Modul zur Erzeugung der Datei conlimit.mexhp7
phaselimit.m	graphische Darstellung eines geregelten Rueckfahrvorganges unter Matlab
testlimited.dat	Datei der Testmuster zum Anlernen des Reglers
trainlimited.dat	Datei der Trainingsmuster zum Anlernen des Reglers
wlimited.dat	angelernte Gewichte des Reglers in Textform

3.) Programme zur Regelung einer nichtlinearen Strecke

3.1.) Erzeugung des Streckenemulators

Verzeichnis: ml/car2/emul

Dateien:	Funktionsbeschreibung:
auto.m	Zustandsdarstellung der Bewegung des Autos
auto.net	Parameterdatei zum Anlernen der Strecke mit fast
emulauto.w	angelernte Gewichte der Strecke in Textform
genpattern.m	Erzeugung der Trainingsmuster zum Anlernen der Strecke unter Matlab
pattern.dat	Datei der Trainingsmuster zum Anlernen des Emulators

3.2.) Programme zur Generierung des Reglers in Lektionen

Verzeichnisse: ml/car2/lesson1
ml/car2/lesson2
ml/car2/lesson3
ml/car2/lesson4

Anmerkung: Durch die aehnliche Struktur der Dateien in den Verzeichnissen lesson1 ... lesson4 sind in der folgenden Tabelle die erstellten Programme nur einfach aufgefuehrt. Fur den Platzhalter X sind fuer die jeweilige Lektion die Zahlen 1 ... 4 einzusetzen

Dateien:	Funktionsbeschreibung:
autovor.m	Umkehrabbildung (Vorwaertsfahren) der Systemgleichung der Strecke
blitz.c	angelernte Gewichte des Reglers in Form eines C-Moduls
cont.net	Parameterdatei zum Anlernen des Reglers mit fast

emulauto.w	angelernte Gewichte der Strecke in Textform
lessonX.m	generieren der Trainingsmuster der X. Lektion
lessonXp.dat	Datei der Trainingsmuster zum Anlernen des Reglers
wcontX.dat	angelernte Gewichte des Reglers in Textform nach der X. Lektion
cont1.net	weitere Parameterdatei der 4. Lektion
wcont5.dat	angelernte Gewichte des Reglers in 4. Lektion mit cont1.net

3.3.) Programme zum Zeichnen von Rueckfahrvorgaengen

Verzeichnis: m1/car2

Anmerkung: Die Programme mit der gleichen Bezeichnung wie unter 3.1. und 3.2. sind identisch und werden nicht weiter beschrieben; X ist wieder als Platzhalter fuer die Zahlen 1...4 zu nehmen

Dateien:	Funktionsbeschreibung:
autobild.m	Zeichnet die Stellung des Fahrzeuges auf den Bildschirm bei vorgegebenem Systemzustand
faster.c	C-Modul zur Generierung von faster.sl
faster.sl	Datei zur Anbindung des Emulators an Matlab
fastercont.c	C-Modul zur Generierung von fastercont.sl; ist identisch mit faster.c
fastercont.sl	Datei zur Anbindung des Reglers nach 4. Lektion an Matlab
fastercontX.sl	Datei zur Anbindung des Reglers nach X. Lektion an Matlab
trace.m	vergleichender Rueckwaertsfahrvorgang von mathematischem Modell und Emulator der ungeregelten Strecke
menu.m	menuegesteuerter Rueckfahrvorgang; verwendete Routinen sind: regler.m = Reglereinstellung strecke.m = Streckeneinstellung ab.m = Anfangsbedingungen einstellen tastatur.m = Anfangsbedingungen von Tastatur einlesen datei.m = Anfangsbedingungen aus Datei einlesen back.m = Rueckfahrschritt durchfuehren

4.) Programme zur Regelung eines Sattelschleppers

4.1.) Erzeugung des Streckenemulators

Verzeichnis: ml/truck/emul

Dateien:	Funktionsbeschreibung:
emultruck.w	angelernte Gewichte der Strecke in Textform
genpattern.m	Erzeugung der Trainingsmuster zum Anlernen der Strecke unter Matlab
pattern.dat	Datei der Trainingsmuster zum Anlernen des Emulators
truck.m	Zustandsdarstellung der Bewegung des Sattelschleppers
truck.net	Parameterdatei zum Anlernen der Strecke mit fast

4.2.) Programme zur Generierung des Reglers in Lektionen

Verzeichnisse: ml/truck/lesson1
ml/truck/lesson2
ml/truck/lesson3
ml/truck/lesson4
ml/truck/lesson5
ml/truck/lesson6

Anmerkung: Durch die aehnliche Struktur der Dateien in den Verzeichnissen lesson1 ... lesson6 sind in der folgenden Tabelle die erstellten Programme nur einfach aufgefuehrt. Fur den Platzhalter X sind fuer die jeweilige Lektion die Zahlen 1 ... 6 einzusetzen

Dateien:	Funktionsbeschreibung:
blitz.c	angelernte Gewichte des Reglers in Form eines C-Moduls
cont.net	Parameterdatei zum Anlernen des Reglers mit fast
emultruck.w	angelernte Gewichte der Strecke in Textform
lessonX.m	generieren der Trainingsmuster der X. Lektion
lessonXp.dat	Datei der Trainingsmuster zum Anlernen des Reglers
truckvor.m	Umkehrabbildung (Vorwaertsfahren) der Systemgleichung der Strecke
wcontX.dat	angelernte Gewichte des Reglers in Textform nach der X. Lektion

4.3.) Programme zum Zeichnen von Rueckfahrvorgaengen

Verzeichnis: ml/truck

Anmerkung: Die Programme mit der gleichen Bezeichnung wie unter 4.1. und 4.2. sind identisch und werden nicht weiter beschrieben

Dateien:	Funktionsbeschreibung:
bild.m	Zeichnet die Stellung des Fahrzeuges auf den Bildschirm bei vorgegebenem Systemzustand
faster.c	C-Modul zur Generierung von faster.sl
faster.sl	Datei zur Anbindung des Emulators an Matlab
fastercont.c	C-Modul zur Generierung von fastercont.sl; ist identisch mit faster.c
fastercont.sl	Datei zur Anbindung des Reglers nach 6. Lektion an Matlab
fastercontX.sl fastercontX.mexhp7	Datei zur Anbindung des Reglers nach X. Lektion an Matlab
menu.m	menuegesteuerter Rueckfahrvorgang; verwendete Routinen sind: regler.m = Reglereinstellung strecke.m = Streckeneinstellung ab.m = Anfangsbedingungen einstellen tastatur.m = Anfangsbedingungen von Tastatur einlesen datei.m = Anfangsbedingungen aus Datei einlesen back.m = Rueckfahrschritt durchfuehren
trace.m	vergleichender Rueckwaertsfahrvorgang von mathematischem Modell und Emulator der ungeregelten Strecke

TEIL 2: 2.1. AUFLISTUNG DER DATEIEN DES NETZWERKSIMULATORS FAST,
DIE VERAENDERT WURDEN

Verzeichnis: arbeiten/fast

Dateien: cor.c, dio.c, int.c, qcp.c, wio.c, cgn.c
fst.h, par.h, str.h, var.h

2.2. AUFLISTUNG DER NEUEN MODULE IN DER DATEI EMUL.C

```
/*****************************************************************/
/*                                                               */
/* file:          emul.c                                         */
/*                                                               */
/* version:       1.0                                            */
/*                                                               */
/* author:        B. Frenzel                                     */
/*                                                               */
/* date:          Mai 20 1994                                    */
/*                                                               */
/* routine:                    description:                      */
/*---------------------------------------------------------------*/
/* build_emulator_prop_net()   allocates memory for weight matrix,   */
/*                             activations etc. for the emulator,    */
/*                             that is driven by the controller      */
/* emulator_connect_all()      connect all layers of the neural      */
/*                             emulator                              */
/* emulator_connect_layers()   subroutine of emulator_               */
/*                             connect_all()                         */
/* controller_emulator_        forward pass of the backpropa-        */
/*      forward_pass()         gation-algorithm through the          */
/*                             controller and emulator               */
/* emulator_forward_pass()     forward pass of the backpropa-        */
/*                             gation-algorithm through the emul.    */
/* controller_emulator_        backward pass of the backpropa-       */
/*      backward_pass()        gation-algorithm through the          */
/*                             controller and emulator               */
/* emulator_backward_pass()    backward pass of the backpropa-       */
/*                             gation-algorithm through the emul.    */
/* emulator_compute_score()    compute error bits and true error     */
/*                             of output for training patterns,      */
/*                             if desired                            */
/* emulator_compute_test_      compute error bits and true error     */
/*      score()                of output for test patterns, if       */
/*                             desired                               */
/* emulator_check_states()     check, if state variables of          */
/*                             of emulator exceed bounds             */
/*                                                               */
/* Copyright:                                                    */
/* Institut fuer Regelungstechnik der                            */
/* Universit t Erlangen-N rnberg                                 */
/* Cauerstr. 7, 91058 Erlangen, FRG, 1994                        */
/*                                                               */
/*****************************************************************/
#include "var.h"  /* include external declarations                 */
#include "str.h"  /* include for structure declarations            */
#include <math.h> /* include declarations for mathematical functions */


/* used functions in emul.c */
void build_emulator_prop_net(),emulator_connect_all(),
     emulator_connect_layers(),emulator_backward_pass(),
     emulator_forward_pass(),emulator_compute_score(),
     controller_emulator_backward_pass(),emulator_compute_test_score();
int  controller_emulator_forward_pass(),emulator_check_states();
```

```
/***** The function build_emulator_prop_net() allocates memory    *****/
/***** for the emulator                                           *****/
void build_emulator_prop_net()
{
     int i,hidden_count,k;

     if (NEmulatorHidden < 0)
          fatal_error0("NEmulatorHidden must be positive.\n");
     hidden_count = 0;
     for (i=0; i<HIDDEN_LAYERS; i++) {
          if (NEmulatorHiddenUnits[i]<0)
               fatal_error1("NEmulatorHidden%d must be positive.\n",i+1);
          hidden_count += NEmulatorHiddenUnits[i];
     }

     if (NEmulatorHidden > 0) {
          if (hidden_count > 0) {
               if (NEmulatorHidden != hidden_count)
                    fatal_error1("NEmulatorHidden must be removed when using NEmulatorHidden[:
                         HIDDEN_LAYERS);
          } else
               NEmulatorHiddenUnits[0] = NEmulatorHidden;
     } else          /* NEmulatorHidden == 0 */
          NEmulatorHidden = hidden_count;

     EmulatorFirstHidden  = NEmulatorInputs + 1;
     EmulatorFirstOutput  = EmulatorFirstHidden + NEmulatorHidden;
     EmulatorFirstUnit    = EmulatorFirstHidden;
     NEmulatorUnits       = EmulatorFirstOutput + NEmulatorOutputs;
     EmulatorErrorSums    = (double *)nmalloc(NEmulatorUnits * sizeof(double));
     EmulatorActFuncs     = (int *)nmalloc(NEmulatorUnits * sizeof(int));
     EmulatorOutputs      = (double *)nmalloc(NEmulatorUnits * sizeof(double));
     NEmulatorConnections = (int *)ncalloc(NEmulatorUnits, sizeof(int));

     /* build two dimensional arrays */
     EmulatorWeights = (double **)nmalloc(NEmulatorUnits * sizeof(double *));
     EmulatorConnections = (int **)nmalloc(NEmulatorUnits* sizeof(int *));
     EmulatorLinks = (double **)nmalloc((EmulatorMaxIter+1) * sizeof(double *));
     OutputArray = (double **)nmalloc((EmulatorMaxIter+1) * sizeof(double *));

     for (i=EmulatorFirstUnit; i<NEmulatorUnits; i++) {
          EmulatorWeights[i] = (double *)nmalloc(NEmulatorUnits * sizeof(double));
          EmulatorConnections[i] = (int *)ncalloc(NEmulatorUnits,sizeof(int));
          if (i<EmulatorFirstOutput)
               EmulatorActFuncs[i] = EmulatorUnitType;
          else
               EmulatorActFuncs[i] = EmulatorOutputType;
     }
     for (i=0; i<(EmulatorMaxIter+1); i++)  /* allocate Linkmatrix between emulator and contr
          EmulatorLinks[i] = (double *)nmalloc((NOutputs+NInputs) * sizeof(double));
     for (i=0; i<(EmulatorMaxIter+1); i++)
          OutputArray[i] = (double *)nmalloc((NUnits+NEmulatorUnits-2) * sizeof(double));

     /* the bias unit */
     if (EmulatorUseBiasUnit == TRUE)
          EmulatorOutputs[0]=1.0;

     /* allocate output bounds if not given in parameter file */
     if (EmulatorOutputBounds == NULL) {
          EmulatorOutputBounds = (double **)nmalloc(2*sizeof(double *));
          for (i=0; i<2; i++)
          EmulatorOutputBounds[i]=(double *)nmalloc(NEmulatorOutputs * sizeof(double));
          /* initialize output bounds */
          for (i=0; i<NEmulatorOutputs; i++) {
```

```
                EmulatorOutputBounds[0][i] = MIN_BOUND;
                EmulatorOutputBounds[1][i] = MAX_BOUND;
           }
      }

      /* allocate StatesDontCare if not given in parameter file */
      if (StatesDontCare == NULL)
           StatesDontCare = (int *)ncalloc(NEmulatorOutputs, sizeof(int));

      /* allocate EmulatorOutputScale, if not given in parameter file */
      if (EmulatorOutputScale == NULL) {
           EmulatorOutputScale = (double *)nmalloc(NEmulatorOutputs*sizeof(double));
           /* initialize output scale */
           for (i=0; i<NEmulatorOutputs; i++)
                EmulatorOutputScale[i]=1.;
      }

      /* allocate EmulatorErrorWeights if not given in parameter file */
      if (EmulatorErrorWeights == NULL) {
           EmulatorErrorWeights = (double *)nmalloc(NEmulatorOutputs*sizeof(double));
           /* initialize output scale */
           for (i=0; i<NEmulatorOutputs; i++)
                EmulatorErrorWeights[i]=1.;
      }
}
```

```
/***** emulator_connect_all() connects all layers for emulator *****/
void emulator_connect_all()
{
      int i,start1,end1,start2,end2;

      /* start at the input units */
      start1 = 1;
      end1   = NEmulatorInputs;
      start2 = EmulatorFirstHidden;

      /* connect all layers with positive number of units */
      /* except output layers                             */
      for (i=0; i<HIDDEN_LAYERS; i++)
           if (NEmulatorHiddenUnits[i] >0) {
                end2 = start2 + NEmulatorHiddenUnits[i] - 1;
                /* connect previous and following layer */
                emulator_connect_layers(start1,end1,start2,end2);
                start1 = start2;
                end1   = end2;
                start2 = end2+1;
           }
      /* connect last hidden and output layer */
      emulator_connect_layers(start1,end1,EmulatorFirstOutput,NEmulatorUnits-1);
}
```

```
/***** emulator_connect_layers() connects two layers *****/
void emulator_connect_layers(start1,end1,start2,end2)
int  start1, end1, start2, end2;
{
      int  i, j, k;
```

```
     if (ShowNetwork)
          fprintf(stdout, "Connect Layers of emulator: %d %d  %d %d\n",  start1, end1, start2,
     for (i=start2; i<=end2; i++) {
          /* add connection to BIAS unit */
          if ((NEmulatorConnections[i] == 0) && EmulatorUseBiasUnit) {
               NEmulatorConnections[i]   = 1;
               EmulatorConnections[i][0] = 0;
          }

          k = NEmulatorConnections[i];
          for (j=start1; j<=end1; j++) {
               NEmulatorConnections[i]++;
               if (NEmulatorConnections[i] > NEmulatorUnits)
                    fatal_error0("The number of allowable connections has been overrun.\n");
               EmulatorConnections[i][k] = j;
               k++;
          }
     }

}
```

```
/***** forward pass of the backpropagation-algorithm through the *****/
/***** controller and emulator                                    *****/
int controller_emulator_forward_pass(input,reset,pattern)
double    *input;
int       reset,
          pattern;
{
     int    i,j,k,EmulatorIter,EmulatorHalt;
     double *arrayp;

     /* initialize Variables */
     EmulatorIter = 0;
     EmulatorHalt = FALSE;
     arrayp       = OutputArray[0];
     for (i=0; i<NInputs; i++)
          arrayp[i] = input[i];

     /* main forward pass loop */
     while (EmulatorIter<EmulatorMaxIter && !EmulatorHalt) {
          /* forward pass through controller                      */
          forward_pass(arrayp,reset);
          /* store adaline outputs of controller in arrayp-Matrix */
          for (i=0; i<NUnits-1; i++)
                         arrayp[i] = Outputs[i+1];
          /* store outputs of controller in arrayp-Matrix         */
          for (i=0; i<NOutputs; i++)
               arrayp[i+NUnits-1] = Outputs[FirstOutput+i];
          /* store controller inputs as emulator inputs           */
          for (i=0; i<NEmulatorOutputs; i++)
               arrayp[NUnits-1+NOutputs+i] = arrayp[i];
          /* skip to emulator                                     */
          arrayp += (NUnits-1);
          /* forward_pass through emulator                        */
          emulator_forward_pass(arrayp);
          /* scale and passthrough emulator outputs               */
          for (i=0; i<NEmulatorOutputs; i++) {
               /* scale                                           */
               EmulatorOutputs[EmulatorFirstOutput+i] *=
               EmulatorOutputScale[i];
               /* passthrough                                     */
               EmulatorOutputs[EmulatorFirstOutput+i] +=
               (PassThrough*arrayp[NOutputs+i]);
          }
```

```
          /* store adaline outputs of emulator                    */
          for (i=0; i<NEmulatorUnits-1; i++)
               arrayp[i] = EmulatorOutputs[i+1];
          arrayp = OutputArray[EmulatorIter+1];
          /* store outputs of emulator                            */
          for (i=0; i<NEmulatorOutputs; i++)
               arrayp[i] = EmulatorOutputs[EmulatorFirstOutput+i];

          EmulatorHalt = emulator_check_states(input,
                              TrainingOutputs[pattern],arrayp);
          EmulatorIter++;
     }
     return(EmulatorIter);
}


/***** check, if outputs of emulator exceed bounds *****/
int emulator_check_states(input,goal,arrayp)
double  *input,*goal,*arrayp;
{
     int i;
     for (i=0; i<NEmulatorOutputs; i++) {
          if (!(StatesDontCare[i])) {
               if ((arrayp[i] <= EmulatorOutputBounds[0][i]) ||
                         (arrayp[i] >= EmulatorOutputBounds[1][i]))
                    return TRUE;
          }
     }
     return FALSE;
}




/***** forward pass through emulator net *****/
void emulator_forward_pass(input)
double *input;
{
     double    sum, *weight;
     int       i, j, *connect;


     /* Load in the input vector in emulator net */
     for (i=0; i<NEmulatorInputs; i++)
          EmulatorOutputs[i+1] = input[i];

     /* For each unit, collect the incoming activation */
     /* and pass it through                            */
     for (j=EmulatorFirstUnit; j<NEmulatorUnits; j++) {
          connect = EmulatorConnections[j];
          weight  = EmulatorWeights[j];
          sum     = 0.0;
          /* Calculate output values for Unit j */
          i = NEmulatorConnections[j];
          while (i > 0) {
               i--;
               sum += EmulatorOutputs[connect[i]] * weight[i];
          }
          EmulatorOutputs[j] = activation_function(EmulatorActFuncs[j], sum);
     }
}
```

```
/***** compute Errorbits and TrueError, if output desired *****/
void emulator_compute_score(goal)
double *goal;
{
     double diff;
     int  i;

     for (i=EmulatorFirstOutput; i<NEmulatorUnits; i++) {
          if (EmulatorErrorWeights[i-EmulatorFirstOutput]) {
               diff = goal[i-EmulatorFirstOutput] - EmulatorOutputs[i];
               TrueError += diff * diff;
               if (fabs(diff) > ScoreThreshold)
                    ErrorBits++;
          }
     }
}
```

```
/***** backward pass for controller emulator combination *****/
void controller_emulator_backward_pass(goal,EmulatorIter)
double *goal;
int    EmulatorIter;
{
     int i,j,k;

     /* set output error for backward pass */
     for (i=EmulatorFirstOutput; i<NEmulatorUnits; i++) {
#ifdef    HYPERERR
          EmulatorErrorSums[i] = hyperr(goal[i-EmulatorFirstOutput] -
                                        EmulatorOutputs[i]);
#else
          EmulatorErrorSums[i] = goal[i-EmulatorFirstOutput] -
                                        EmulatorOutputs[i];
#endif
          EmulatorErrorSums[i] = EmulatorErrorWeights[i-EmulatorFirstOutput]*
                                 EmulatorErrorSums[i];
     }

     for (j=EmulatorIter-1; j>=0; j--) {
          /* clear ErrorSums and EmulatorErrorSums */
          for (i=0; i<FirstOutput; i++)
               ErrorSums[i] = 0.0;
          for (i=0; i<EmulatorFirstOutput; i++)
               EmulatorErrorSums[i] = 0.0;
          /* load emulator adaline outputs */
          for (i=0; i<NEmulatorUnits-1; i++)
               EmulatorOutputs[i+1] = OutputArray[j][i+NUnits-1];
          /* scale emulator outputs */
          for (i=EmulatorFirstOutput; i<NEmulatorUnits; i++)
               EmulatorErrorSums[i] *= EmulatorOutputScale[i-EmulatorFirstOutput];
          /* backprop through emulator */
               emulator_backward_pass();
          /* passthrough emulator outputs */
```

```
          for (i=0; i<NEmulatorOutputs; i++) {
               if (PassThrough)
                    EmulatorErrorSums[NOutputs+1+i] +=
                    (EmulatorErrorSums[EmulatorFirstOutput+i]/
                         EmulatorOutputScale[i]);
          }
          /* load controller adaline outputs */
          for (i=0; i<NUnits-1; i++)
               Outputs[i+1] = OutputArray[j][i];
          /* load ErrorSums in controller outputs */
          for (i=0; i<NOutputs; i++)
               ErrorSums[FirstOutput+i] = EmulatorErrorSums[i+1];
          /* backprop through controller */
          backward_pass();
          /* store ErrorSums of emulator and controller in emulator outputs */
               for (i=0; i<NEmulatorOutputs; i++)
                    EmulatorErrorSums[EmulatorFirstOutput+i]=
                         EmulatorErrorSums[NOutputs+1+i]+
                              ErrorSums[i+1];
        }
}
```

```
/***** backward pass through emulator net *****/
void emulator_backward_pass()
{
     double    error, *weight;
     int       i, j, conn, *connect;

     for (j=NEmulatorUnits-1; j>=EmulatorFirstUnit; j--) {
          connect = EmulatorConnections[j];
          weight  = EmulatorWeights[j];
          error   = (activation_prime(EmulatorActFuncs[j], EmulatorOutputs[j]) +
                         SigmoidPrimeOffset) * EmulatorErrorSums[j];
          /* This should be a for loop for i from 0 to NConnections[j]-1, but */
          /* a lot of time is spent here, so here is the faster version.      */
          i = NEmulatorConnections[j];
          while (i > 0) {
               i--;
               conn = connect[i];
               EmulatorErrorSums[conn] += error * weight[i];
          }
     }
}
```

```
/***** compute error bits and true error of output for test patterns *****/
void emulator_compute_test_score()
{
     double    diff, *goal;
     int       i, j;

     /* clear test error */
     TestError = 0.0;

     for (i=0; i<NTestPatterns; i++) {
          /* pass through controller and emulator */
          controller_emulator_forward_pass(TestInputs[i],TestBreaks[i],i);
```

```
          goal = TestOutputs[i];
          for (j=EmulatorFirstOutput; j<NEmulatorUnits; j++) {
               if (EmulatorErrorWeights[j-EmulatorFirstOutput]) {
                    diff = goal[j-EmulatorFirstOutput] - EmulatorOutputs[j];
                    TestError += diff * diff;
               }
          }
     }
}
```