Dokumentation zu n++

Martin Riedmiller

23. Oktober 1997

Inhaltsverzeichnis

1 n++ - noch ein Simulator

Folgende Grundforderungen führten zur Entwicklung des Simulatorkerns **n++** für mehrschichtige neuronale Netztopologien:

- Gleichzeitige Simulation mehrerer Netze
- Einfache Erweiterbarkeit der Grundfunktionen
- Einfache Einbindung in Benutzerprogramme
- Trennung der Netzfunktionen und Benutzeranforderungen (z.B. Trainieren von Mustermengen)

Der Simulatorkern realisiert gemäß obiger Spezifikation im wesentlichen folgende Funktionen:

- Aufbau der Netztopologie
- Vorwärts propagieren eines Eingabemusters
- Rückwärts propagieren des Fehlervektors
- Änderung der Gewichte (lernen)
- Speichern/Laden des Netzes

2 Verzeichnisstruktur

Das *n++* Verzeichnis unterteilt sich in folgende Unterverzeichnisse:

src:	Quellcode *n++, PatternSet++, aux*
include:	Include Headerdatei für Anwenderprogramme
lib:	Objektdateien *n++, PatternSet++, aux*
demo_src:	Beispielprogramme *expl.c*
tools:	Werkzeuge
doc:	Dokumentation
examples:	Beispiele Netzbeschreibungen, Mustermengen

3 Der Simulatorkern n++.cc / n++.h

Der Simulatorkern ist als C++ Klasse implementiert.

3.1 Aufbau eines Netzes aus einem Anwendungsprogramm

3.1.1 Aufbau der Netzschichten

Die Netztopologie wird durch die Prozedur

create_layers(int layers, int layer_nodes[])

aufgebaut, wobei *layers* die Anzahl der Schichten des Netzes (inkl. Input/- und Outputlayer) angibt. Der Vektor layer_nodes beschreibt die Anzahl der Neuronen pro Schicht, wobei *layer_nodes[0]* die Anzahl der Eingabeneuronen, *layer_nodes[1]* die Anzahl der Neuronen in der ersten Hidden-layer, etc... angibt.

3.1.2 Aufbau der Verbindungsstruktur (Gewichte)

Die Prozedur

connect_layers()

baut eine Schicht-zu-Schicht Verbindungsstruktur auf (dies entspricht der typischen Verbindungsweise von Feed-forward Netzen). Zusätzlich erhält jedes Neuron ein Biasgewicht.

Die Prozedur

connect_shortcut()

baut neben den Schicht-zu-Schicht Verbindungen und den Biasgewichten noch schichtübergreifende Verbindungen auf ('Shortcut-Connections').

Mit der Prozedur

connect_units(int to_unit, int from_unit, FTYPE value)

kann gezielt eine Verbindung zu einem einzelnen Neuronen (*to_unit*) aufgebaut werden. Die Verbindung erhält das Gewicht *value*.

Die Prozedur

void init_weights(int mode, FTYPE range)

initialisiert die vorhandenen Netzgewichte und Biase mit Zufallswerten im Bereich $[-range, +range]$. Falls $mode = 1$ werden die Biase nicht initialisiert, sondern auf 0.0 gesetzt.

Mit

void set_seed(long seed_no)

kann gegebenenfalls vorher der Startpunkt des Zufallszahlengenerators festgelegt werden (um z.B. immer die gleiche Folge von Zufallsinitialisierungen zu erzeugen).

3.1.3 Setzen der Aktivierungsfunktionen

Jedem Neuron kann eine eigene Aktivierungsfunktion zugewiesen werden, die aus der gewichteten Summe der ankommenden Gewichte eine Aktivierung und damit die Ausgabe des Neurons berechnet. Die Funktion

set_unit_act_f(int unit_no,int act_id)

weist dem Neuron *unit_no* die Aktivierungsfunktion *act_id* zu. Im Moment stehen die Aktivierungsfunktionen *LOGISTIC (0)*, *SYMMETRIC (1)* und *IDENTIÄT (LINEAR) (2)* zur Verfügung.

Die Funktion

set_layer_act_f(int layer_no,int act_id)

weist einer ganzen Schicht *layer_no* die Aktivierungsfunktion *act_id* zu.

3.2 Aufbau eines Netzes mittels einer Netzbeschreibungssprache

n++ stellt einen komfortablen Mechanismus zum Aufbau einer Netztopologie mittels einer Beschreibungssprache zur Verfügung. Die durch *filename* bezeichnete Datei wird mittels der Prozedur

load_net(char filename[])

eingelesen. *n++* erkennt, ob es sich um die Beschreibung der Topologie und der Gewichte eines bereits trainierten Netzes handelt oder ob die Datei Kommandos zum Aufbau einer neuen Netzstruktur enthält. Damit können neue Netztopologien sehr einfach definiert werden. Die Netzbeschreibungssprache umfaßt im wesentlichen die Aufbau-Kommandos der 'C'-Schnittstelle:

topology: <*input*><*hidden 1*>... <*hidden n*><*output*> - Definition der Schichten und der Anzahl Neuronen eines Netzes (entspricht *create_layeres()*

connect_layers - Aufbau von Schicht-zu-Schicht Verbindungen

connect_shortcut - Aufbau von Schicht-zu-Schicht + Shortcut Verbindungen

set_unit_act_f <unit_no><act_id> - Setzen der Aktivierungsfunktion bei Neuron *unit_no*

set_layer_act_f <layer_no><act_id> - Setzen der Aktivierungsfunktion für eine ganze Schicht *layer_no*

init_weights <mode><range> - Initialisieren der Gewichte. *Mode = 1* bewirkt, daß Biasgewichte nicht zufällig initialisiert, sondern auf 0.0 gesetzt werden.

set_update_f <update_function><parameter 1>.. <parameter n> - Setzen der Gewichtsänderungsfunktion mit den entsprechenden Parametern.

scale_input <position><scale_mode><parameter 1><parameter 2> - Skalierung des durch *position* festgelegten Eingabewertes (Zählung beginnt bei 1!) unter Anwendung der Skalierungsart *scale_mode* mit den Parametern *parameter 1, parameter 2* (Skalierung s. 3.3).

scale_output <position><scale_mode><parameter 1><parameter 2> - Skalierung des durch *position* festgelegten Ausgabewertes (Zählung beginnt bei 1!) unter Anwendung der Skalierungsart *scale_mode* mit den Parametern *parameter 1, parameter 2* (Skalierung s. 3.3).

Beispiel (demo1.net):

```
topology: 2 4 3
connect_shortcut
set_layer_act_f 2 2
init_weights 0 .5
set_update_f 1 0.3 1.0

#scaling
input_scale 1 0 .5 2     # Scale first input unit, mode 0, parameter .5, 2
input_scale 4 2 .4 .6    # Scale fourth input unit, mode 2, parameter .4, .6
output_scale 2 1 .4 .4   # Scale fourth input unit, mode 2, parameter .4, .6
```

3.3 Transformation von Netzein- und ausgaben

3.3.1 Forward-Pass

Bei der Definition einer Netzstruktur (s. 3.2) besteht die Möglichkeit, einzelne Netzein- und ausgabewerte zu skalieren. Beim Aufruf der Prozedur *forward_pass* werden entsprechend der Spezifikation zuerst die Eingabewerte skaliert, die transformierten Werte durchs Netz propagiert, und die erzielte Netzausgabe gegebenfalls wiederum transformiert. Die Spezifikation der Transformation geschieht innerhalb einer Netzdefinition mit dem Schlüsselwort *scale_input* bzw. *scale_output*. Die Syntax ist für beide Schlüsselwörter dieselbe:

scale_input <position><scale_mode><parameter 1><parameter 2>

Hierbei gibt *position* die Position des zu skalierenden Werts im Eingabevektor an, **wobei die Zählung bei 1 beginnt!**. *scale_mode* gibt die Art der durchzuführenden Transformation und *parameter 1* bzw. *parameter 2* die Parameter an. Es stehen folgende Transformationen für die Skalierung des Eingabe-(Ausgabe-)werts $x \rightarrow x^*$ zur Verfügung:

- 0: Symmetrische Skalierung

$$x^* = \begin{cases} x * parameter1 & , \quad x > 0 \\ x * parameter2 & , \quad x \leq 0 \end{cases} \tag{1}$$

- 1: Lineare Skalierung

$$x^* = parameter1 * x + parameter2 \tag{2}$$

- 2: Binäre Skalierung $\rightarrow [0, 1]$

$$x^* = \begin{cases} 0 & , \quad x \leq parameter1 \\ 1 & , \quad x \geq parameter2 \end{cases} \tag{3}$$

- 3: Binäre symmetrische Skalierung $\rightarrow [-1, 1]$

$$x^* = \begin{cases} -1 & , \quad x \leq parameter1 \\ 1 & , \quad x \geq parameter2 \end{cases} \tag{4}$$

3.3.2 Backward-Pass

Beim Aufruf des Rückwärts-Propagieren (*backward_pass*) wird die Skalierung bei den Skalierungsmodi 0 (Symmetrische Skalierung) und 1 (Lineare Skalierung) berücksichtigt. Und zwar berechnet sich die Ableitung des skalierten Werts x^* nach dem unskalierten Wert x wie folgt:

- 0: Symmetrische Skalierung

$$\frac{dx^*}{x} = \begin{cases} parameter1 & , \quad x >= 0 \\ parameter2 & , \quad x < 0 \end{cases} \tag{5}$$

- 1: Lineare Skalierung

$$\frac{dx^*}{x} = parameter1 \tag{6}$$

Dabei wird sowohl die Skalierung der Eingabewerte als auch die Skalierung der Ausgabewerte berücksichtigt.

3.4 Interne Datenstruktur

Im Simulator wird das Netz als lineare Liste von Neuronen verwaltet. Neuron 0 bezeichnet das Bias Neuron mit konstanter Ausgabe 1. Neuron 1 bis i sind die Neuronen der Eingabeschicht, Neuron i+1 bis (i+h) sind die Neuronen der Hidden layer(s), Neuron (i+h+1) bis (i+h+o) bezeichnen die Neuronen der Ausgabeschicht (i = Anzahl der Eingabeneuronen, h = Anzahl der Hiddenneuronen, o = Anzahl der Ausgabeneuronen). Der Benutzer ist damit nur insofern konfrontiert, wenn einzelnen Neuronen im Netz eine bestimmte Aktivierungsfunktion zugewiesen werden soll, bzw. wenn zwei Neuronen des Netzes gezielt miteinander verbunden werden sollen.

Die Zählung der Schichten (*layers*) des Netzes beginnt bei *layer* 0, der Eingabeschicht. *layer* 1 bezeichnet also die 1. Hiddenlayer und so weiter.

3.5 Propagieren durchs Netz

Um das *Vorwärtspropagieren* eines Eingabevektors durch ein Netz zu bewirken, wird die Prozedur

*forward_pass(FTYPE *in_vec, FTYPE *out_vec)*

aufgerufen. Der Vektor *in_vec[]* enthält die Eingabewerte. Falls der Vektor größer ist als die Anzahl der Input-Units werden nur die ersten i Werte verwendet. Als Ergebnis des Prozeduraufrufs werden im o-stelligen Vektor *out_vec[]* die Ausgabewerte der Neuronen der Ausgabeschicht übergeben. Dabei

werden gegebenenfalls Skalierung der Eingabe- bzw. Ausgabewerte gemäß den Angaben des Benutzers vorgenommen.

Die Prozedur

*backward_pass(FTYPE *dedout, FTYPE *dedin)*

führt ein 'rückwärts propagieren' der Fehlerwerte der Ausgabeschicht und dabei das Berechnen der partiellen Ableitungen des Fehlers nach den Gewichten, $\frac{\partial E}{\partial w_{ij}}$, durch. Die eigentliche Durchführung der Propagierung wird von der aktuell ausgewählten Rückpropagierungsfunktion ausgeführt.

In der aktuellen Implementierung ist diese der herkömmliche Backpropagation-Algorithmus. Im Vektor *dedout[]* werden die Ableitungen der Ausgabeneuronen nach dem Fehler, $\frac{\partial E}{\partial o_i}$, übergeben (typischerweise gilt $\frac{\partial E}{\partial o_i} = -(t_i - o_i)$). Falls für ein Ausgabeneuron eine differenzierbare Skalierung angegeben ist, wird der Wert gemäß obiger Vorschrift differenziert (s. 3.3). Diese Werte werden durch das Netz propagiert, die partiellen Ableitungen berechnet und diese werden solange aufsummiert, bis die eigentliche Lernfunktion (s. *update_weights()*) aufgerufen wird. Als Besonderheit enthält der Vektor *dedin[]* die Ableitungen des Ausgabefehlers nach den Eingaben. Dies kann z.B. dazu benützt werden, um Fehler durch ein Netz durchzupropagieren (wird im Bereich 'Neuro Control' verwendet).

Die Prozedur

*backward_pass_light(FTYPE *dedout, FTYPE *dedin)*

berechnet **(nur) die Ableitungen nach der Eingabe, d.h. die Ableitung nach den Gewichten wird nicht berechnet. Dies ist dann nützlich, wenn man nur die Ableitung der Eingabe nach der Netzausgabe berechnen will, die (evtl. bislang berechneten) Gewichtsgradienten aber nicht verändern darf.**

3.6 Lernen der Gewichte - der 'Gewichts-update'

Die Prozedur

void update_weights()

ruft die aktuelle Lernfunktion zur Änderung der Gewichte auf. Je nach Häufigkeit des Aufrufs kann somit ein *learning by pattern* **(Aufruf nach jedem** *backward_pass()***), ein** *learning by block* **(Aufruf am Ende einer kompletten Mustermenge), oder eine Mischform davon erzielt werden.**

Die Update-Funktion berechnet die jeweilige Gewichtsänderung als Funktion des im *backward_pass* **berechneten (ggf. aufsummierten) Gradienten. Nach der Änderung wird der Gradient auf 0 gesetzt. Defaultmäßig ist die BP-Lernfunktion eingestellt.**

Hinweis: Die Prozedur *update_weights()* **wird ohne Parameter aufgerufen. Die Einstellung der Lernparameter erfolgt gleichzeitig mit der Wahl der Lernfunktion mittels**

*void set_update_f(int typ,float *params).*

Im Parametervektor *params[]* **werden die für die jeweilige Lernfunktion gültigen Parameter übergeben.**

3.6.1 Backpropagation (BP (0))

Auswahl mit

set_update_f(0,params) **oder** *set_update_f(BP,params).*

Parameter[0]: Lernrate

Parameter[1]: Momentum

Parameter[2]: Weight-decay

3.6.2 Rprop (RPROP (1))

Auswahl mit

set_update_f(1,params) **oder** *set_update_f(RPROP,params).*

Parameter[0]: $\triangle_0$

Parameter[1]: $\triangle_{max}$

Parameter[2]: Weight-decay

Achtung: Der weight-decay Parameter bei Rprop in der n++-Version unterscheidet sich *fundamental* **von der SNNS-Implementierung (beim SNNS gibt er den Exponent an und wird bei jedem Propagieren aufaddiert). In der n++-Version wird der weight-decay erst unmittelbar vor dem Gewichtsupdate angewendet. Umrechnung (ohne Gewähr) für SNNS- weight-decay Parameter α in n++-Parameter λ:**

$\lambda \approx 10^{-\alpha} * AnzahlMuster$

3.7 Sichern und Laden von Netzen

Das Speichern eines Netzes, d.h. die Topologie, Aktivierungsfunktionen und Gewichtswerte, wird von der Prozedur

save_net(char filename[])

ausgeführt, wobei *filename* **den Namen der Datei angibt. Das Laden eines Netzes von einer Datei geschieht analog mit**

load_net(char filename[]).

Mit

print_net()

kann die aktuelle Netztopologie auf *stdout* **ausgegeben werden.**

3.8 Öffentliche Parameter (Public variables)

Nach Aufbau der Netztopologie oder Einlesen eines Netzes aus einer Datei, steht folgende Topologiebeschreibung in Form eines *Records* **zur Verfügung:**

```
struct topo_typ{
  int layer_count;  /* number of layers */
  int in_count;     /* number of input units */
  int out_count;    /* number of output units */
  int hidden_count; /* number of hidden units */
  int unit_count;   /* total number of  units */
} topo_data;
```

Desweiteren werden zur Kommunikation mit dem Netz zwei Vektoren zur Verfügung gestellt, und zwar *in_vec*, **dessen Größe der Größe der Eingabeschicht entspricht, und** *out_vec*, **mit derselben Größe wie die Ausgabeschicht. Man kann sie beispielsweise sinnvoll zur Propagierung von Eingabevektoren einsetzen.**

Achtung: Die Vektoren *in_vec* **und** *out_vec* **dienen nur als Hilfestellung für den Benutzer; weder müssen sie eingesetzt werden, noch werden ihre Werte beim Vorwärts- /Rückwärtspropagieren automatisch gesetzt.**

3.9 Temporal Difference (TD) Learning

Das Temporal Difference Lernverfahren ist ein Verfahren zur Minimierung des Bewertungsfehlers aufeinanderfolgender Zustände in Sequenzen. Der Fehler, der zum Zeitpunkt t gemacht wird (*td_error(t)*), kann erst zum Zeitpunkt t+1 berechnet werden, wenn die Bewertung des Folgezustands t+1 bekannt ist. (Literatur: Sutton, 1988, *'Learning by Temporal Differences'*, Machine Learning, 1988). Damit ergibt sich folgendes Vorgehen:

Anfangszustand der Sequenz t_0

- **Initalisieren der Werte:** *TD_init_sequence()*
- **Berechnen der Bewertung:** *forward_pass(netin[],&Output(t_0))*
- **Rückpropagieren zur Berechnung von** $\frac{\partial O}{\partial w_{ij}(t_0)}$**:** *TD_backward_pass(0,λ)*

Zwischenzustände der Sequenz $t_1, .., t, .., t_{n-1}$

- **Berechnen der Bewertung:** *forward_pass(netin[],&Output(t))*
- **Berechnen des** $td\_error(t-1) := Output(t-1) - (r(t) + \gamma Output(t))$
- **Rückpropagieren:** *TD_backward_pass(td_error(t-1),λ)*
- **Ggf. Gewichtsänderung:** *update_weights()*
  Neuberechnung der Netzausgabe *forward_pass(netin[],&Output(t))*

Endzustand t_n

- **Berechnen der Bewertung:** *forward_pass(netin[],&Output(t_n)*
- **Berechnen des** $td\_error(t_{n-1}) := Output(t_{n-1}) - (r(t_n) + \gamma Output(t_n))$
- **Rückpropagieren:** *TD_backward_pass(td_error(t_{n-1}),λ*
- **Ggf. Gewichtsänderung:** *update_weights()*
  Neuberechnung der Netzausgabe *forward_pass(netin[],&Output(t))*
- **Berechnen des** $td\_error(t_n) := Output(t_n) - R(t_n)$
- **Rückpropagieren:** *TD_backward_pass(td_error(t_{n-1}),λ)*
- **Gewichtsänderung:** *update_weights()*

Besonderheiten

- **Vergessen von Sequenzen:** *clear_derivatives()* **löscht den summierten Gradienten seit dem letzten Gewichtsupdate. Sinn: Schlechte Sequenzen können vergessen werden.**
- **Betonen von Sequenzen:** *multiply_derivatives(factor)*. **Multipliziert den summierten Gradient mit dem angegebenen Faktor. Sinn: Betonen oder Abschwächen einzelner Sequenzen. Für** *factor = 0* **ergibt sich** *clear_derivatives()*

3.9.1 Implementierung des TD-Verfahrens

Beim Rückwärtspropagieren wird zum einen die Ableitung des Gewichts nach der aktuellen Ausgabe $Output(t)$, $\frac{\partial Output}{w_{ij}}(t)$, berechnet, und daraus der zeitlich 'verschmierte' Wert

$$\frac{\partial O}{w_{ij}}(t) = \lambda \frac{\partial O}{w_{ij}}(t-1) + \frac{\partial Output}{w_{ij}}(t) = \sum_{k=1}^{t} \lambda^{t-k} * \frac{\partial Output}{w_{ij}}(k)$$

Der für den Gewichtsupdate benutzte 'Gradient' $\frac{\partial E}{\partial w_{ij}}(t)$ berechnet sich wie folgt:

$$\frac{\partial E}{\partial w_{ij}}(t-1) = td\_error(t-1) * \frac{\partial O}{w_{ij}}(t-1)$$

Wie beim herkömmlichen Rückwärtspropagieren wird der Gradient solange aufsummiert, bis eine Gewichtsänderung mittels des Aufrufs *update_weights()* vorgenommen wird.

4 Benutzung des Simulatorkerns

Der Simulatorkern wird als Objektdatei zum Anwendungsprogramm dazugebunden (z.B.*g++ -o myprog myprog.o net.o -lm)*, die Header-Datei *net.h* per include im Programmtext eingebunden (*#include"net.h"*).

4.1 Beispiel (exp1.c) - einfaches dreischichtiges Netz

```
#include <stdio.h>
#include "n++.h"
//// bindet die n++ Headerdatei ein

#define INPUTS 2
//// 2 Eingabeneuronen

#define OUTPUTS 3
//// 3 Ausgabeneuronen

#define LAYERS 3
//// 3 Schichten (incl. Ein- u. Ausgabeschicht)

Net net1;
//// net1 ist vom Typ Net

int no_layers = LAYERS;
//// no_layers: Anzahl der Schichten

int layer_nodes[LAYERS]={INPUTS,4,OUTPUTS};
//// Vektor layer_nodes: Anzahl der Neuronen pro Schicht

float uparams[5];
//// 5-dimensionaler Parametervektor fuer Lernfunktion

FTYPE in_vec[INPUTS], out_vec[OUTPUTS];
//// Definition der Ein- u. Ausgabevektoren

int main( int argc, char *argv[] )
```

```
{
  int i;

  net1.create_layers(no_layers,layer_nodes);
//// erzeugen des Netzes

  net1.connect_layers();
//// Jedes Neuron wird mit allen Neuronen der benachbarten Schichten verbunden

  net1.init_weights(0,.5);
//// Die Gewichte mit Zufallswerten zwischen 0 und 0.5 initialisiert

  uparams[0] = 0.1;
  uparams[1] = 0.9;
//// Setzen der Lernparameter

  net1.set_update_f(BP,uparams);
//// waehlt Backpropagation als Lernfunktion aus

  for(i=0;i<INPUTS;i++)
    in_vec[i] = 0.5;
//// setzt alle Eingaben auf 0.5

  net1.forward_pass(in_vec,out_vec);
//// berechnet Ausgabe

  for(i=0;i<OUTPUTS;i++)
    printf("%f  ",out_vec[i]);
//// Ausgabe steht in out_vec[]

  printf("\n");
  printf("Number of input units: %d\n",net1.topo_data.in_count);
  printf("Number of hidden units: %d\n",net1.topo_data.hidden_count);
  printf("Number of output units: %d\n",net1.topo_data.out_count);
  printf("Total Number of units: %d\n",net1.topo_data.unit_count);
//// schreibt Ausgabe
}
```

5 Verwaltung von Mustermengen - PatternSet.cc / PatternSet.h

Zur Verwaltung von Trainings-/Testmustern steht die Klassen *PatternSet* zur Verfügung. Jedem Trainingsset kann eine eigene Instanz dieser Klasse zur Verfügung gestellt werden.

5.1 Laden einer Mustermenge

Die Prozedur

int load_pattern(char filename[])

lädt eine Trainingsmustermenge von der durch *filename* bezeichneten Datei. Falls ein Fehler beim Laden auftritt, wird ein entsprechender Fehlerwert zurueckgegeben, ansonsten 0. Das lesbare Fileformat ist dem SNNS-Patternformat ähnlich, es können auch SNNS-Dateien eingelesen werden. Kommentarzeilen beginnen mit '%'. Wichtig ist das Leerzeichen vor dem Doppelpunkt im Kopf des Patternfiles!

```
<beliebig viele Zeilen mit beliebigem Text bis zum ersten 'No.'>
No. of patterns     : <Anzahl der Muster>          /* ACHTUNG: Leerzeichen vor ':' */
No. of input units  : <Anzahl der Eingabneuronen>   /* ACHTUNG: Leerzeichen vor ':' */
No. of output units : <Anzahl der Ausgabeneuronen>  /* ACHTUNG: Leerzeichen vor ':' */

<beliebig viele Leer- oder Kommentarzeilen bis zur ersten Ziffer>

#Mustername
<beliebig viele Leer- oder Kommentarzeilen bis zur ersten Ziffer>
1.3 2.3 0 0 0 2.     % Eingabemuster; Keine Leerzeile zum Zielmuster!!!
0 1                  % Zielmuster
```

Die im Kopf angegebenen Grö ßen 'No. of input units' bestimmt die Anzahl der eingelesenen Werte des Eingabemusters. Falls die Zahl 'No. of input units' größer ist als die Anzahl der Werte pro Zeile, wird der Rest des Eingabe-Vektors mit Nullen aufgefüllt. Falls mehr Zahlen in der Zeile stehen als die Größe des Eingabe-Vektors, werden die überzähligen Ziffern ignoriert. Dasselbe gilt analog für die Anzahl der Ausgabe-Units.

Falls *No. of output units* auf 0 gesetzt ist, werden nur Eingabemuster gelesen.

Die Angabe der Anzahl vorhandener Muster (No. of patterns) in der Kopfzeile wird ignoriert. Sie ist nur aus kompatibilitätsgründen zum SNNS enthalten. Die Anzahl der Muster einer Mustermenge *pattern_count* (s.u) hängt allein von der in Musterdatei gefundenen Anzahl an Trainingsmustern ab. Diese Anzahl wird beim Einlesen der Musterdatei automatisch festgestellt.

5.2 Öffentliche Variablen

Folgende Variablen werden nach Einlesen der Musterdatei auf die entsprechenden Werte gesetzt:

int pattern_count: Anzahl der eingelesenen Muster

input_count: Anzahl der Eingabewerte pro Muster

target_count: Anzahl der Ausgabewerte pro Muster

Damit ist es zum Beispiel möglich, in Zusammenhang mit dem Neurokernel ein Netz in Abhängigkeit der eingelesenen Anzahl von Eingabeneuronen automatisch, d.h. zur Laufzeit zu generieren.

Über die Matrix input[0..*pattern_count*-1][0..*input_count*-1] kann auf ein bestimmtes Eingabemuster bzw. auf den Wert eines bestimmten Eingabeneurons für ein bestimmtes Eingabemuster zugegriffen werden. Analog gilt dies für die Matrix der Ziel-(Target)werte *target[][]*.

Die Prozedur

print_pattern()

gibt die eingelesenen Muster auf *stdout* aus.

5.3 Beispiel (expl2.c)

```
#include <stdio.h>
#include "n++.h"
//// bindet die n++ Headerdatei ein

#include "PatternSet.h"
//// bindet die PatternSet Headerdatei ein

#define LAYERS 3
//// 3 schichtiges Netz (incl. Ein- u. Ausgabeschicht)

#define HIDDEN_COUNT 5
//// 5 Hidden Units

int topology[LAYERS];
```

```
//// topology[]: Vektor, in dem die Anzahl der Neuronen der Schichten stehen.

float uparams[MAX_PARAMS]={0.0,0.0,0.0,0.0,0.0};
//// 5-dimensionaler Lernparametervektor

FTYPE *in_vec,*out_vec;
//// Ein- u. Ausgabevektoren

Net net1;
PatternSet pat1;
//// Netz net1, Muster pat1

void create_net(Net *net,int inputs, int hidden, int outputs)
//// erzeugt Netz
{
  topology[0] = inputs; topology[1] = hidden; topology[2] = outputs;
//// Anzahl der Neuronen in Input-,Hidden- und Output Unit

  net->create_layers(LAYERS,topology);
//// erzeugt Schichten

  net->connect_layers();
//// verbindet Schichten

  net->init_weights(0,.5);
//// initialisiert Gewichte mit Werten zwischen 0 und 0.5

  in_vec = new float [net->topo_data.in_count];
  out_vec = new float [net->topo_data.out_count];
}

float train(Net *net,PatternSet *pat)
//// trainiert Netz

{
  float error,tss;
  int p,i;

  tss = 0.0;
  for(p=0;p<pat->pattern_count;p++){
    net->forward_pass(pat->input[p],out_vec);
//// berechnet Ausgabe

    for(i=0;i<net->topo_data.out_count;i++){
      error = out_vec[i] = out_vec[i] - pat->target[p][i]; /* out_vec = dE/do = (o-t) */
//// Ausgabe = Ausgabe - Zielausgabe

      tss += error * error;
//// Fehler = Summe der quadrierten Abweichungen

    }
    net->backward_pass(out_vec,in_vec);
//// Fehler wird rueckpropagiert

  }
  net->update_weights();
//// Gewichte werden angepasst

  return(tss);
```

```
}

int main( int argc, char *argv[] )
{
  int n;
  int nepochs;

  pat1.load_pattern("n10");
//// laedt Muster

  create_net(&net1,pat1.input_count,HIDDEN_COUNT,pat1.target_count);
//// ruft Funktion auf (s.o.)

  printf("nepochs, uparams 0 1 2?\n");
  scanf("%d %f %f %f",&nepochs,&uparams[0],&uparams[1],&uparams[2]);
//// fragt Lernparameter und Anzahl der Lernzyklen ab

  net1.set_update_f(RPROP,uparams);
//// waehlt RPROP als Lernalgorithmus

  for(n=0;n<nepochs;n++)
    printf("%f\n",train(&net1,&pat1));
//// trainiert Netz nepochs-mal

  net1.save_net("expl2.net");
//// speichert Netz ab

}
```

6 Oft benötigte Hilfsfunktionen - aux.cc / aux.h

Die in *aux.cc* definierten Routinen sollen den Gebrauch von oftmals benötigten Funktionen, wie sie das Trainieren und Testen von Mustermengen erfordern, erleichtern.

Achtung: Die Headerdatei *aux.h* muß immer nach *n++.h* und *PatternSet.h* eingebunden werden.

6.1 Hilfsprozeduren

Die Prozedur

*void aux_trainAll(Net *net,PatternSet *pat,result_type *result)*

propagiert alle in (*pat* **geladenen Muster durch das Netz** *net*, **berechnet für jedes Muster den Fehler zwischen Ausgabe- und Zielwert, propagiert diesen zurück. Über den globalen Parameterrecord** *aux_params* **kann dabei der update-Modus by pattern (***aux_params.update_mode* **= ONLINE (1)) oder by epoch (... = OFFLINE (0)) ausgewählt werden. Das Ergebnis des Trainings wird in einem record vom Typ** *result_type* **(in** *aux.h***) zurückgegeben. Dieser hat zwei Komponenten, einmal den Wert** *tss*, **die Summe des quadrierten Fehlers über alle Ausgabeneuronen über alle Muster, und** *hamdis*, **die Anzahl der Fehlklassifikationen, ebenalls über alle Muster und Ausgabeneuronen. Mit dem Parameter** *aux_params.tolerance* **kann dabei die Toleranzgrenze für die noch korrekte Klassifikation angegeben werden (Fehler, falls** $|t_i - o_i| >$ **tolerance).**

Die Prozedur

*void aux_testAll(Net *net,PatternSet *pat,result_type *result)*

berechnet die Fehlerwerte wie obige Prozedur, führt aber weder eine Rückpropagierung noch einen Gewichtsupdate durch. Mit ihr kann z.B. der Fehler auf einer Testmenge festgestellt werden.

Die Prozedur

*void aux_testAllVerbose(Net *net,PatternSet *pat,result_type *result)*

gibt zusätzlich Eingabe-, Ausgabe- und Zielwerte aus.