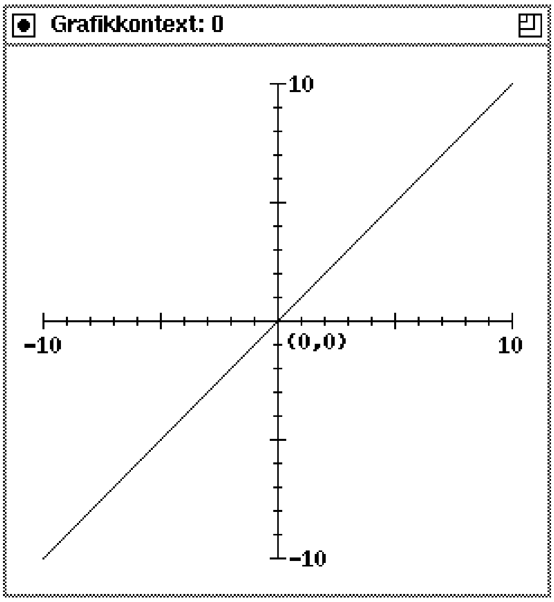

Abbildung 8.3: Die mit dem *plot*–Befehl erzeugte Identitätsfunktion

```
/* Programm zum Plotten der Identitaetsfunktion */
define int grafic_handle; /* Grafikhandle definieren */
open_grafic 300 , 300 to grafic_handle; /* Grafikfenster erzeugen */
/* setze jetzt das Weltkoordinatensystem: */
set_world -10.0 , 10.0 , -10.0 , 10.0 to grafic_handle;
/* zeichne Koordinatenachsen ein: */
axes -10.0 , 10.0 , -10.0 , 10.0 , 0.0 , 0.0 to grafic_handle;
/* zeichne nun die Identitaetsfunktion mit dem Plotbefehl */
define double delta_x = 20.0 / 260.0; /* Schrittweite in x-Richtung */
/* 260.0 ist die Breite des Weltkoordinatensystems in Pixel;
    20.0 ist die Breite des Weltkoordinatensystems in Weltkoordinaten */
define double x = -10.0; /* x-Startwert */
while( x < 10.0 )
{ /* Punkt muss gesetzt werden: */
  plot x , x to grafic_handle; /* setze Punkt */
  set x = x + delta_x; /* auf naechsten x-Wert setzen */
};
```

8.5.2 Linie ausgeben: line

Der *line*–Befehl zeichnet eine Gerade in ein Grafikfenster ein. Der Benutzer kann außerdem noch angeben, ob eine Strichlierung verwendet werden soll. Die Syntax des Kommandos ist:

line_command ::= *"line"* [**dotted_specification**] **x1** *","* **y1** *","* **x2** *","* **y2**
"to" **grafic_handle.**

x1	**::= double_expression.**
y1	**::= double_expression.**
x2	**::= double_expression.**
y2	**::= double_expression.**
grafic_handle	**::= int_expression.**

Bem.:

- **dotted_specification** ist die optionale Angabe einer Strichlierung (siehe Abschnitt 8.4).
- **x1** beschreibt im Weltkoordinatensystem die x-Position des Startpunktes.
- **y1** beschreibt im Weltkoordinatensystem die y-Position des Startpunktes.
- **x2** beschreibt im Weltkoordinatensystem die x-Position des Zielpunktes.
- **y2** beschreibt im Weltkoordinatensystem die y-Position des Zielpunktes.
- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.

Beispiele:
Die folgenden Eingaben im Ein/Ausgabefenster von ProSiNN erzeugen drei Linien in zwei Grafikfenstern:

```
>:define int grafic_handle_1
>:open_grafic 300 , 300 to grafic_handle_1
>:set_world -10.0 , 10.0 , -10.0 , 10.0 to grafic_handle_1
>:axes -10.0 , 10.0 , -10.0 , 10.0 , 0.0 , 0.0 to grafic_handle_1
>:line -10.0 , 5.0 , 10.0 , 0.0 to grafic_handle_1
>:line -10.0 , -10.0 , 10.0 , 50.0  to grafic_handle_1
>:define int grafic_handle_2
>:open_grafic 300 , 300 to grafic_handle_2
>:set_world -10.0 , 10.0 , -10.0 , 10.0 to grafic_handle_2
>:axes -10.0 , 10.0 , -10.0 , 10.0 , 0.0 , 0.0 to grafic_handle_2
>:line [ 10 , 5 , 2 , 7 ] -10.0 , -10.0 , 10.0 , 10.0 to grafic_handle_2
>:
```

Im linken Fenster der Abbildung 8.4 ist eine im sichtbaren Bereich verbleibende und eine aus diesem Bereich herausführende Linie zu sehen. Das zweite Fenster zeigt die Verwendung einer Strichlierung.

8.5.3 Texte ausgeben: text

Zur Textausgabe in einem Grafikfenster dient der *text*–Befehl. Seine Syntax lautet:

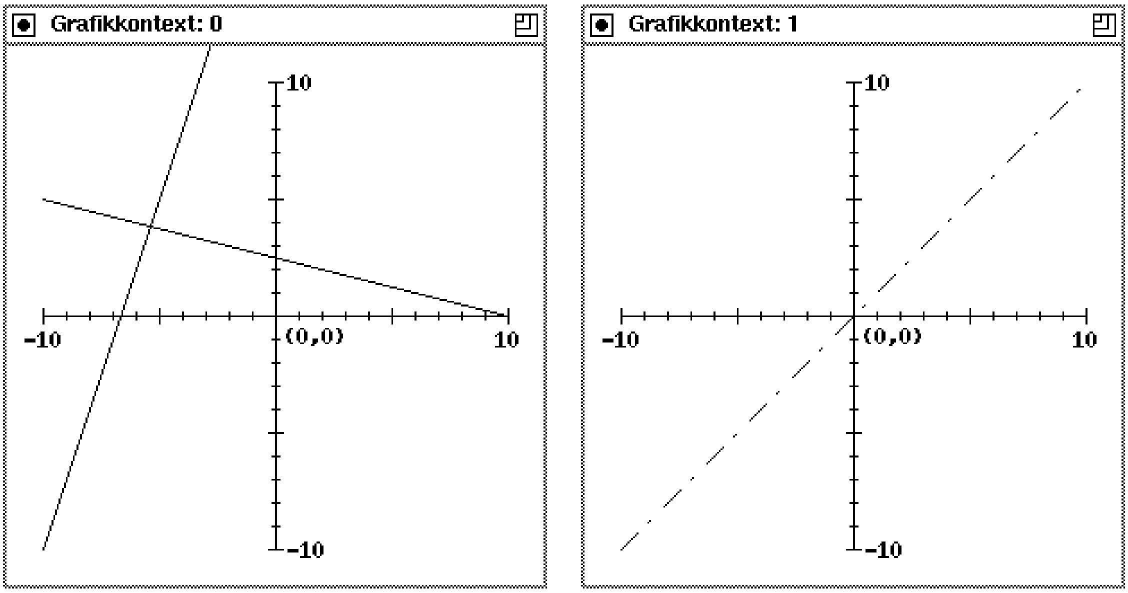

Abbildung 8.4: Mit dem *line*–Befehl erzeugte Geraden

text_command ::= *"text"* **string_val** *"at"* **x** *","* **y** *"to"* **grafic_handle.**
string_val ::= **"Variable oder Konstante vom Typ string".**
x ::= **double_expression.**
y ::= **double_expression.**
grafic_handle ::= **int_expression.**

Bem.:

- **string_val** ist der Ausgabetext. An der Position (**x** , **y**) befindet sich anschließend die linke obere Ecke des Textes.
- **x** beschreibt im Weltkoordinatensystem die linke Ecke des Textes.
- **y** beschreibt im Weltkoordinatensystem die obere Ecke des Textes.
- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.
- Man beachte, daß der Text sich in einem *unsichtbaren* Rechteck befindet. Der Bereich, den das Rechteck überlagert, wird gelöscht (siehe Beispiel).

Beispiel:
Die nachfolgenden Kommandos erzeugen vier Textausgaben in einem Grafikfenster:

```
>:define int grafic_handle
>:open_grafic 300 , 300 to grafic_handle
>:set_world -10.0 , 10.0 , -10.0 , 10.0 to grafic_handle
>:axes -10.0 , 10.0 , -10.0 , 10.0 , 0.0 , 0.0 to grafic_handle
```

```
>:text "(-5.0,-5.0)" at -5.0 , -5.0 to grafic_handle
>:text "Hallo" at -5.0 , 5.0 to grafic_handle
>:text "Hallo" at 5.0 , 5.0 to grafic_handle
>:text "Hallo" at 10.0 , 10.0 to grafic_handle
>:
```

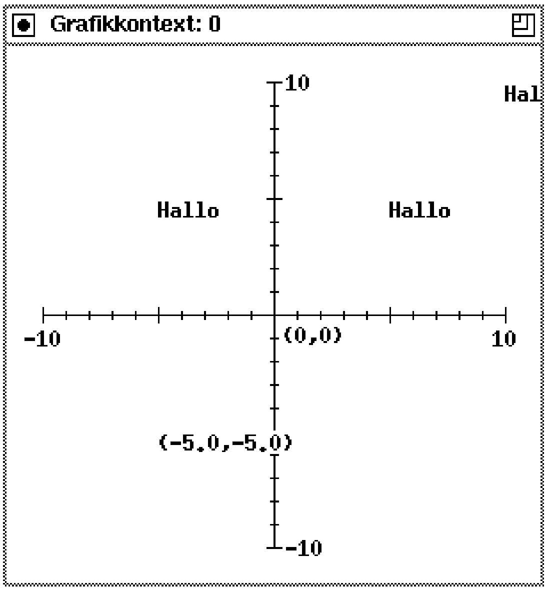

Abbildung 8.5: Textausgaben in einem Grafikfenster

8.6 Funktionen einzeichnen: plot_function

Zur grafischen Ausgabe von Funktionen dient der *plot_function*–Befehl. Die Syntax der Anweisung ist:

plot_function_command ::= *"plot_function"* [**dotted_specification**]
function_name *"to"* **grafic_handle.**
function_name ::= **"Name einer Integer– oder Doublefunktion".**
grafic_handle ::= **int_expression.**

Bem.:

- **dotted_specification** ist die optionale Angabe einer Strichlierung (siehe Abschnitt 8.4).
- **function_name** ist der Bezeichner einer Integer– oder Doublefunktion. Diese Funktion darf nur *ein* Argument (Funktionsvariable) haben!
- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.

- Die Funktionswerte werden über die gesamte Abszissenachse des Weltkoordinatensystems eingezeichnet.

Beispiel:
Bild 8.6 stellt zwei Fenster dar, in die Funktionen eingezeichnet worden sind. Dabei wurde für die Erzeugung des rechten Fensters eine Strichlierungsangabe verwendet.

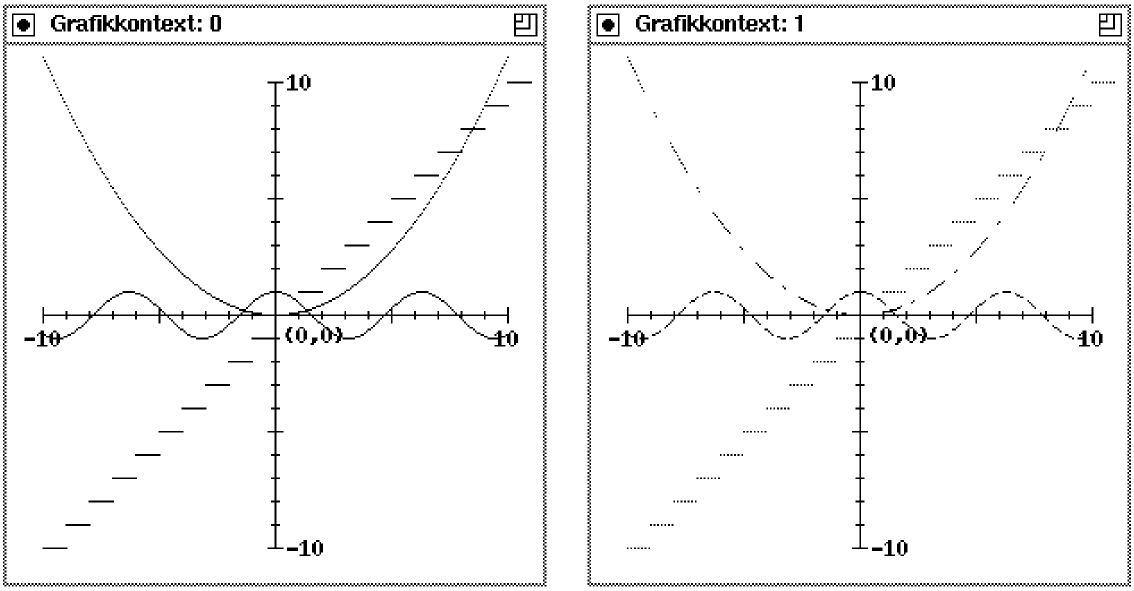

Abbildung 8.6: Mit dem *plot_function*–Befehl gezeichnete Funktionen

Die folgenden Kommandos generierten die beiden Grafikfenster:

```
>:define function int i(x) = x
>:define function double d(x) = (x / 3)^2
>:define int grafic_handle_1
>:open_grafic 300 , 300 to grafic_handle_1
>:set_world -10.0 , 10.0 , -10.0 , 10.0 to grafic_handle_1
>:axes -10.0 , 10.0 , -10.0 , 10.0 , 0.0 , 0.0 to grafic_handle_1
>:plot_function cos to grafic_handle_1
>:plot_function i to grafic_handle_1
>:plot_function d to grafic_handle_1
>:define int grafic_handle_2
>:open_grafic 300 , 300 to grafic_handle_2
>:set_world -10.0 , 10.0 , -10.0 , 10.0 to grafic_handle_2
>:axes -10.0 , 10.0 , -10.0 , 10.0 , 0.0 , 0.0 to grafic_handle_2
>:plot_function [ 5 , 2 , 5 , 2 ] cos to grafic_handle_2
>:plot_function [ 1 , 1 , 1 , 1 ] i to grafic_handle_2
>:plot_function [ 10 , 5 , 2 , 7 ] d to grafic_handle_2
>:
```

8.7 Inhalt eines Grafikfensters löschen: clear_grafic

Der *clear_grafic*–Befehl dient zum Löschen des Inhaltes eines Grafikfensters. Seine Syntax lautet:

clear_grafic_command ::= *"clear_grafic"* **grafic_handle.**

Bem.:

- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.
- Der gesamte Inhalt des Fensters geht verloren. Um eine erzeugte Grafik zu speichern, muß der Anwender auf *X Window Utilities* zurückgreifen.
- Das vorher gesetzte Weltkoordinatensystem bleibt erhalten.

Kapitel 9

Neuronale Netze bearbeiten

In diesem Kapitel werden die Kommandos vorgestellt, die ProSiNN zur Bearbeitung von Neuronalen Netzen anbietet. Dies sind u.a. Anweisungen, die Netze erzeugen und initialisieren, die Topologie ändern und ein Netz wieder freigeben. Weitere Merkmale sind Befehle zum Laden und Speichern von Neuronalen Netzen.

Nach der Einführung des Kommandos zum Erzeugen eines Neuronalen Netzes, wird der Befehl erläutert, der den globalen Netznamen setzt. Den Anschluß bilden die Anweisungen, die beim Initialisieren eines Netzes verwendet werden. Die beiden folgenden Abschnitte beschäftigen sich mit dem Speichern von Neuronalen Netzen. Ein Befehl bietet hier die Möglichkeit an, ein Netz in wieder einlesbarer Form abzuspeichern; die Alternative dazu ist das Speichern des Netzes als Teil eines *C*–Programms (die praktische Weiterverwertung eines Neuronalen Netzes ist damit gewährleistet). Der sechste Abschnitt ist dem Kommando gewidmet, das ein gespeichertes Netz wieder einliest. Im Anschluß werden drei Befehle zur Bearbeitung der Netzstruktur eingeführt. Es handelt sich hier um die Anweisungen, die eine Netzebene hinzufügen oder entfernen, sowie die Anzahl der Neuronen in einer Schicht ändern. Abschließend wird die Verwendung und das Setzen einzelner netzinterner Werte, sowie das Freigeben eines Netzes beschrieben.

9.1 Ein Netz neu erzeugen: layer

Zum Erzeugen eines Neuronalen Netzes[1] dient der *layer*–Befehl. Die Syntax des Kommandos lautet:

layer_command	::=	*“layer“* **number_layer** *“:“*
		layer_specification *“,“* **layer_specification**
		{ *“,“* **layer_specification** }.
number_layer	::=	**int_expression.**
layer_specification	::=	[**layer_name**] **number_units_in_layer.**
layer_name	::=	**„Bezeichner, der noch nicht definiert ist“.**
number_units_in_layer	::=	**int_expression.**

[1] Die von ProSiNN verwendeten Strukturen und Besonderheiten Neuronaler Netze werden in Abschnitt 3.2 beschrieben

Bem.:

- **number_layer** legt die Anzahl der Netzebenen fest, die zu generieren sind. Es muß gelten: **number_layer** ≥ 2.
- **layer_specification** ist die Spezifikation der schichtinternen Neuronenanzahl:
  - Für jede Netzebene muß eine solche Angabe vorliegen.
  - Die erste Spezifikation korrespondiert mit der Eingabeebene, die nächste Angabe mit der zweiten Schicht usw.
  - **layer_name** ist die optionale Angabe eines Bezeichners, der noch nicht im System definiert wurde. Ist dieser Parameter vorhanden, so wird eine Netzebenenvariable mit diesem Namen erzeugt. Sie bildet dann eine Referenz auf die Netzschicht, die diese Spezifikation beschreibt.
  - **number_units_in_layer** legt die Anzahl der zu erzeugenden Neuronen in dieser Ebene fest. Es muß gelten: **number_units_in_layer** ≥ 1.
- Alle netzinternen Werte (Schwellenwerte, Gewichte u.ä.) werden mit 0.0 initialisiert.
- Da im System nur *ein* Neuronales Netz vorhanden sein darf, wird der *layer*–Befehl mit einer Fehlermeldung abgebrochen, wenn schon ein Netz existierte. In diesem Fall muß das alte Netz z.B. mit dem *free_net*–Befehl freigegeben werden (siehe Abschnitt 9.9).

Beispiele:

- `layer 3: 4 , 3 , 2`

  Dieses Kommando erzeugt ein Neuronales Netz mit drei Ebenen: In der ersten Schicht befinden sich 4, in der mittleren 3 und in der letzten Ebene 2 Neuronen.

- `layer 4: l1 2 , 3 , 3 , l4 1`

  Dieser Befehl generiert ein Netz mit vier Ebenen. Für die erste Schicht werden 2 Neuronen und die Netzebenenvariable `l1` *erzeugt. Die beiden folgenden Ebenen erhalten jeweils 3 Neuronen. Die Ausgabeschicht ist anschließend mit der Layer–Variablen* `l4` *ansprechbar und beinhaltet ein Neuron.*

- `layer 10 - 8: 23 / 11 , 2 ^ 0`

  Mit dieser Anweisung wird ein zweischichtiges Neuronales Netz generiert, das 2 Eingabe– und ein Ausgabeneuron enthält.

9.2 Netznamen setzen: netname

Mit dem *netname*–Befehl kann der Benutzer eine globale Stringvariable setzen, die den Namen eines Neuronalen Netzes beinhaltet. Dieser Name wird mit beim Speichern des Netzes (in wieder einlesbarer Form) gesichert. Die Kommandosyntax ist:

netname_command ::= *“netname“* **string_val.**
string_val ::= **„Variable oder Konstante vom Typ string“.**

Bem.:

- **string_val** ist eine Variable oder Konstante vom Typ *string*, die den Netznamen enthält. Es darf auch der leere String verwendet werden.
- Mit diesem Befehl wird die globale Variable **net_name** gesetzt. Das Kommando darf auch dann verwendet werden, wenn im System kein Neuronales Netz existiert.
- Wird ein Neuronales Netz in wieder einlesbarer Form gespeichert, dann erscheint dieses Kommando immer in der gesicherten Datei vor den eigentlichen Netzdaten. Durch diesen Befehl wird beim Einlesen des Netzes die Variable **net_name** auf den Wert gesetzt, der beim Abspeichern aktuell war.

Beispiel:
Der folgende Befehl setzt die globale Variable **net_name***:*

```
netname "Dies wird der Wert der Variablen 'net_name'"
```

Anschließend beinhaltet die Variable **net_name** *die Zeichenkette:*
”`Dies wird der Wert der Variablen 'net_name'`”

9.3 Netz initialisieren

Erzeugt man ein Neuronales Netz neu, dann werden alle netzinternen Werte (Schwellenwerte, Gewichte u.ä.) mit 0.0 initialisiert. Für die Konvergenz des Trainingsverfahrens ist es ratsam, daß alle netzinternen Werte voneinander verschieden sind. Deshalb bietet ProSiNN die Möglichkeit an, diese Werte mit einem (Pseudo–) Zufallszahlengenerator neu zu initialisieren.

Im ersten Unterabschnitt wird erklärt, wie die Intervalle, aus denen die Zufallszahlen stammen, durch den Anwender gesetzt werden können. Anschließend wird der Befehl vorgestellt, der einen neuen Startwert für den Zufallszahlengenerator setzt. Den Abschluß bildet das Kommando, das das im System vorliegende Netz neu initialisiert.

9.3.1 Zufallswertebereich setzen

Der Benutzer kann mit dem *init_net*–Befehl (siehe Unterabschnitt 9.3.3) Pseudo–Zufallszahlen an die Schwellenwerte und Gewichte eines Netzes zuweisen. Der Bereich aus dem die Zufallszahlen stammen, ist durch den Anwender modifizierbar. Die folgenden globalen Variablen dienen zur Festlegung der Intervalle:

theta_min und **theta_max**:

> Diese Double–Variablen schränken den Bereich ein, aus dem die Initialisierungswerte für die Schwelle θ kommen. Sie dürfen vom Anwender beliebig gesetzt werden, solange **theta_max** größer oder gleich **theta_min** ist.
> Die Defaultwerte sind: **theta_min = -0.5** und **theta_max = 0.5**.

w_min und **w_max**:

> Diese Double–Variablen schränken den Bereich ein, aus dem die Initialisierungswerte für die Gewichte kommen. Sie dürfen vom Benutzer beliebig gesetzt werden, solange **w_max** größer oder gleich **w_min** ist.
> Die Defaultwerte sind: **w_min = -0.5** und **w_max = 0.5**.

Will der Anwender ein anderes Intervall setzen, so muß er den *set*–Befehl verwenden (siehe Abschnitt 5.2).

9.3.2 Zufallsgeneratorstartwert setzen: set_rand

Der Zufallsgenerator verwendet zur Erzeugung neuer Zahlen einen Startwert. Mit diesem Wert berechnet er dann neue Zufallszahlen (und ändert den Startwert für seinen nächsten Aufruf). Um beim Initialisieren der Netze vergleichbare Anfangsbedingungen zu gewährleisten, bietet ProSiNN mit dem *set_rand*–Befehl die Möglichkeit an, den Startwert zu setzen. Seine Syntax lautet:

set_rand_command ::= *“set_rand“* **start_val.**
start_val ::= **int_expression.**

Bem.:

- **start_val** ist der Startwert, der größer oder gleich 0 sein sollte. Negative Zahlen werden intern vom System als große positive Werte betrachtet.
- Nach dem Systemstart von ProSiNN wird beim ersten Initialisierungsbefehl der Startwert 1 verwendet. Bei jedem weiteren Initialisierungskommando wird ein neuer Startwert, der sich aus den vorhergehenden Berechnungen ergibt, benutzt.

9.3.3 Netz initialisieren: init_net

Zum Initialisieren eines Netzes muß der Benutzer den *init_net*–Befehl verwenden. Dabei wird vom System zur Berechnung der Zufallszahlen ein Generator benutzt, der auf einem *linearen Kongruenz–Algorithmus* basiert. Die Syntax dieses Kommandos ist:

init_net_command ::= *“init_net“*.

Bem.:

- Liegt kein Netz im System vor, so wird der Befehl mit einer Fehlermeldung abgebrochen.
- Die Schwellenwerte werden mit Zufallszahlen aus [**theta_min**,...,**theta_max**] und die Gewichte mit Werten aus dem Intervall [**w_min**,...,**w_max**] initialisiert.
- Die anderen netzinternen Werte, auf die mit den Neuronen– oder Gewichtspräfixen zugegriffen werden kann (siehe Unterabschnitt 4.3.2 und 4.3.3), initialisiert das System mit 0.0.
- Die beiden Integervariablen, die die aktuelle und die Gesamtanzahl der bisher ausgeführten Trainingszyklen beschreiben, werden mit 0 initialisiert. Anschließend haben also **act_iterations** und **sum_iterations** den Wert 0.

9.4 ProSiNN–Netz speichern

In diesem Abschnitt soll der Befehl vorgestellt werden, mit dem ein Neuronales Netz in wieder einlesbarer Form gespeichert werden kann. Dazu wird das Dateiformat, das eine einlesbare Netzdatei einhalten muß, beschrieben. Im Anschluß wird dann das Speicherkommando präsentiert.

9.4.1 Netz–Dateiformat

Um ein gespeichertes Netz wieder einlesen zu können, gibt ProSiNN ein festes Format, das nicht verletzt werden darf, vor. Eine Datei, die ein Neuronales Netz enthält, muß folgende Syntax einhalten:

net_file	::=	**statement_block**
		layer_command *“;“*
		“NET:“
		first_layer_block { **layer_block** } **last_layer_block.**
statement_block	::=	{ **statement** *“;“* }.
first_layer_block	::=	**unit_weights** { **unit_weights** }.
layer_block	::=	**unit_theta unit_weights**
		{ **unit_theta unit_weights** }.
last_layer_block	::=	**unit_theta** { **unit_theta** }.
unit_weights	::=	**weight_val** [**name**] *“;“* { **weight_val** [**name**] *“;“* }.
unit_theta	::=	**theta_val** [**name**] *“;“*.
weight_val	::=	**double_const.**
theta_val	::=	**double_const.**
name	::=	**„Bezeichner“.**

Bem.:

- **statement_block** ist eine Folge von ProSiNN–Kommandos, die jeweils mit einem Semikolon abgeschlossen sind. Der *layer*– und der *quit*–Befehl (siehe Abschnitt 13.7) dürfen hier nicht verwendet werden; ansonsten sind aber alle Anweisungen erlaubt.
- **layer_command** ist das Kommando zum Erzeugen eines Neuronalen Netzes (siehe Abschnitt 9.1).
- **first_layer_block** spezifiziert die Gewichte der ersten Ebene.

  Für jedes Neuron der ersten Schicht muß **unit_weights** stehen. **unit_weights** beinhaltet die Gewichte, die von dieser Zelle zu allen Neuronen der nächsten Ebene führen. Die Anzahl der Zellen in der zweiten Schicht bestimmt die Anzahl von **weight_val** [**name**] *“;“* im Nichtterminalsymbol **unit_weights**:

  – **weight_val** ist der Wert des entsprechenden Gewichtes. Hier muß eine Gleitkommakonstante stehen.

  – **name** ist ein optionaler (noch nicht existierender) Bezeichner, der eine Variable auf dieses Gewicht definiert.

- **layer_block** spezifiziert die Schwellenwerte und die Gewichte einer inneren Schicht. Für jede innere Ebene muß ein **layer_block** stehen. Dabei bestimmt die Anzahl der Zellen in dieser Ebene die Anzahl von **unit_theta unit_weights** im Nichtterminalsymbol **layer_block**:

  – **unit_theta** spezifiziert das entsprechende Neuron:

    * **theta_val** ist der Schwellenwert des Neurons. Hier muß eine Gleitkommakonstante stehen.
    * **name** ist ein optionaler (noch nicht existierender) Bezeichner, der eine Variable auf dieses Neuron definiert.

  – **unit_weights** beinhaltet die Gewichte, die von dieser Zelle zu allen Neuronen der nächsten Ebene führen. Die Anzahl der Zellen in der nächsten Schicht bestimmt die Anzahl von **weight_val** [**name**] *“;“* im Nichtterminalsymbol **unit_weights**:

    * **weight_val** ist der Wert des entsprechenden Gewichtes. Hier muß eine Gleitkommakonstante stehen.
    * **name** ist ein optionaler (noch nicht existierender) Bezeichner, der eine Variable auf dieses Gewicht definiert.

- **last_layer_block** spezifiziert die Schwellenwerte der letzten Schicht.

  Für jedes Neuron der letzten Ebene muß **unit_theta** stehen. **unit_theta** spezifiziert das entsprechende Neuron:

  – **theta_val** ist der Schwellenwert des Neurons. Hier muß eine Gleitkommakonstante stehen.

  – **name** ist ein optionaler (noch nicht existierender) Bezeichner, der eine Variable auf dieses Neuron definiert.

- In der Netzdatei dürfen beliebig geschachtelte Kommentare vorhanden sein.
- Nach den Netzdaten dürfen nur noch Kommentare folgen.

Beispiel:
Dieses Beispiel zeigt eine Datei, die ein Neuronales Netz mit drei Ebenen definiert. Jede Schicht enthält zwei Neuronen. Durch den layer*–Befehl wird eine Netzebenenvariable* `l_2`*, die die mittlere Schicht bezeichnet, erzeugt. Außerdem werden in der Netzstruktur eine Neuronenvariable* `u_2_2`*, die das 2. Neuron der mittleren Schicht repräsentiert, und eine Gewichtsvariable* `w_2_2_3_2`*, die einen Zugriff auf die Gewichtsverbindung, die von* `u_2_2` *zum letzten Ausgabeneuron verläuft, generiert. Die folgenden Zeilen zeigen die Netzdatei, die durch den* savenet*–Befehl (siehe unten) erzeugt wurde (die Angabe* „`Bias`“ *in den Kommentaren weist darauf hin, daß sich in dieser Zeile der Schwellenwert des entsprechenden Neurons befindet):*

```
netname "Beispielnetz";
layer 3 : 2 , l_2 2 , 2;
NET:

/* Unit 1,1: */
0.0000000; /* Weight to Unit 2,1 */
1.0000000; /* Weight to Unit 2,2 */

/* Unit 1,2: */
2.0000000; /* Weight to Unit 2,1 */
3.0000000; /* Weight to Unit 2,2 */

/* Unit 2,1: */
4.0000000; /* Bias */
5.0000000;/* Weight to Unit 3,1 */
6.0000000;/* Weight to Unit 3,2 */

/* Unit 2,2: */
7.0000000 u_2_2; /* Bias */
8.0000000;/* Weight to Unit 3,1 */
9.0000000 w_2_2_3_2; /* Weight to Unit 3,2 */

/* Unit 3,1: */
10.0000000; /* Bias */

/* Unit 3,2: */
11.0000000; /* Bias */
```

9.4.2 Netz speichern: savenet

Um ein Neuronales Netz in wieder einlesbarer Form zu speichern, muß der Benutzer den *savenet*–Befehl verwenden. Seine Syntax lautet:

savenet_command ::= *“savenet“* [**file_handle** | **file_name**].

Bem.:

- **file_handle** ist ein Dateihandle, das ein vom Anwender geöffnetes File spezifiziert. In dieser Datei wird das Neuronale Netz gespeichert.
- **file_name** ist der Pfad und Name der Datei, in die das Neuronale Netz gespeichert wird (siehe Unterabschnitt 7.2.1). Falls das File vorher schon vorhanden war, wird dessen ursprünglicher Inhalt gelöscht.
- Wird kein Handle und kein Dateiname angegeben, so benutzt ProSiNN das normale Ein/Ausgabefenster als Ausgabemedium (es wird also **stdout** als Dateideskriptor verwendet).
- Der Aufbau der gespeicherten Struktur entspricht der Syntax des Netz–Dateiformates (siehe letzten Unterabschnitt). Zu jedem Netzdatum wird ein Kommentar angegeben, der die entsprechende Strukturkomponente erläutert. Im System vorliegende Netzebenen–, Neuronen– und Gewichtsvariablen werden mit ausgegeben. Das Nichtterminalsymbol **statement_block** (siehe Seite 131) hat die nachfolgende Syntax:

  statement_block ::= **netname_command** *“;“* [**activation_command** *“;“*].

  activation_command (siehe Unterabschnitt 11.3.1) ist die ProSiNN–Anweisung, mit der der Benutzer eine eigene Aktivierungsfunktion setzen kann. Sie wird hier nur dann verwendet, wenn eine vom Anwender definierte Aktivierungsfunktion im System vorliegt.

Beispiel:
Im System sei ein dreischichtiges Neuronales Netz mit 2 Neuronen in der ersten und der zweiten Ebene, sowie einem Ausgabeneuron. Das Netz wurde solange zur Lösung des binären AND*–Operators trainiert, bis das quadratische Fehlermaß in der Größenordnung von* 10^{-12} *liegt. Mit dem Kommando*

```
savenet test
```

wird das Netz in der Datei `test` *des aktuellen Arbeitsverzeichnisses gespeichert:*

```
netname "Eine Loesung des AND-Problems";
layer 3 : 2 , 2 , 1;
NET:

/* Unit 1,1: */
-5.8428779; /* Weight to Unit 2,1 */
-6.5406577; /* Weight to Unit 2,2 */
```

```
/* Unit 1,2: */
-5.8438315; /* Weight to Unit 2,1 */
-6.5395857; /* Weight to Unit 2,2 */

/* Unit 2,1: */
8.1999683; /* Bias */
-14.092124;/* Weight to Unit 3,1 */

/* Unit 2,2: */
9.2644199; /* Bias */
-17.286658;/* Weight to Unit 3,1 */

/* Unit 3,1: */
14.607000; /* Bias */
```

9.5 Netz als C–Programm speichern

ProSiNN bietet die Möglichkeit ein Neuronales Netz als Teil eines *C*–Programms zu speichern. Für den Benutzer hat dies den Vorteil, daß trainierte Netze in eigenen Programmen verwendet werden können. Dazu stellt das System neben dem *save_c_net*–Befehl (siehe Unterabschnitt 9.5.3) zwei externe Dateien zur Verfügung. Bei der ersten handelt es sich um eine *include*–Datei, die von jedem *C*–Programm, das ein von ProSiNN erzeugtes Neuronales Netz anwenden will, importiert werden muß. Die zweite Datei beinhaltet die Propagierungsfunktion, die zum Berechnen einer Netzantwort benötigt wird. In den nachfolgenden Unterabschnitten werden diese Files und das Speicherkommando vorgestellt.

9.5.1 Die Include–Datei: `neural_net.h`

Der Inhalt der Datei `neural_net.h` ist im folgenden dargestellt:

```
/*******************************************************************/
/*  Include-Datei fuer C-Programme, die durch den Simulator erzeugte   */
/*              Neuronale Netze weiter verarbeiten wollen            */
/*******************************************************************/

/* Typ-Definition fuer ein Neuronales Netz,
   das in C verwendet werden soll:          */



typedef struct NEURAL_NET
{ unsigned int nr_layers,   /* Anzahl Ebenen des Neuronalen Netzes */
               *p_nr_units; /* Zeiger auf Array mit der
                               Anzahl Neuronen/Schicht  */
```

```
  double        *p_theta,    /* Zeiger auf Array von Schwellenwerten */
                *p_unit,     /* Zeiger auf Array von Aktivierungswerten */
                *p_weight;   /* Zeiger auf Array von Gewichten */
} NEURAL_NET;

/* Deklaration der Propagierungsfunktion: */

extern void propagate( /* double * ,
                          double * ,
                          NEURAL_NET *
                                      */ );

/* 1. Wert: Eingabearray (muss vom C-Programm mit Werten versorgt werden)
   2. Wert: Ausgabearray (enthaelt nachher das Ergebnis)
   3. Wert: Zeiger auf Netzdaten
*/
```

Hier wird zuerst ein neuer Datentyp `NEURAL_NET`, mit dem ein Neuronales Netz eindeutig festgelegt werden kann, definiert. ProSiNN erzeugt mit dem *save_c_net*–Befehl eine Datenstruktur dieses Typs (bzw. eine Funktion, die einen Zeiger auf diese Struktur liefert). Als letztes wird in dieser Datei noch die Propagierungsfunktion deklariert.

9.5.2 Die Auswertungs–Datei: `neural_net.c`

Der Inhalt der Datei `neural_net.c` ist im folgenden dargestellt:

```
/*********************************************************************/
/*    C-Programm, das die durch den Simulator erstellten Neuronalen   */
/*  Netze verarbeitet: Hier wird die Propagierungsfunktion definiert  */
/*********************************************************************/

/* Deklaration der Aktivierungsfunktion */

extern double activation( /* double */ );


/* Definition der Propagierungsfunktion: */


void propagate(in,out,net)

double *in,      /* Zeiger auf Netz-Eingabeearray */
       *out;     /* Zeiger auf Netz-Ausgabearray */
NEURAL_NET *net; /* Zeiger auf Netzstruktur */

/* Funktion, die die Ausgabewerte eines Neuronalen Netzes bei den
```

```
   Eingabedaten, auf die 'in' zeigt, berechnet und in das Ausgabe-
   array, auf das 'out' zeigt, speichert. Bei der Berechnung wird
   das Netz verwendet, auf dessen Daten 'net' zeigt.
*/

{ static unsigned int nr_layer, /* Anzahl Ebenen des Netzes */
                      i,j,l;    /* Zaehlvariablen */
  static double *p_w,           /* Zeiger auf Gewichte */
                *p_u,           /* Zeiger auf aktuelles Neuron */
                *p_temp_u,      /* Zeiger auf 1. Neuron
                                   der aktuellen Ebene  */
                *p_in,          /* Zeiger auf 1. Neuron der
                                   vorhergehenden Ebene     */
                *p_t;           /* Zeiger auf Schwellenwerte */

  p_w = net->p_weight; /* Zeiger auf 1. Gewicht */
  p_t = net->p_theta; /* Zeiger auf 1. Schwellenwert */
  if( (nr_layer = net->nr_layers) == 2 )
       p_u = out; /* aktuelles Neuron ist Ausgabeeinheit */
  else p_u = p_in = net->p_unit;
            /* aktuelles Neuron ist 1. Zelle der 2. Ebene */
  /* berechne nun die Aktivierungen der Zellen in der 2.
     Schicht (oder der Ausgabeeinheiten)                 */
  for( i = 0; i < *(net->p_nr_units + 1); i++,p_u++ )
  { /* ueber alle Neuronen der 2. Schicht, berechne Aktivierung */
    *p_u = *(p_t++); /* mit Schwellenwert initialisieren */
    for( j = 0; j < *(net->p_nr_units); j++ )
       /* ueber alle Eingabewerte */
       *p_u += *(p_w++) * *(in + j);
               /* Eingabe des Neurons i aufbauen */
    *p_u = activation(*p_u);
           /* Aktivierung des Neurons i berechnen */
  }
  /* Aktivierungen der Einheiten der 2. Ebene wurden berechnet */
  if( nr_layer == 2 ) return; /* fertig */
  /* hier gilt: Das Netz hat mindestens 3 Ebenen */
  p_temp_u = p_u; /* Zeiger auf 1. Neuron des 3. Schicht */
  /* berechne nun die Aktivierungen aller inneren Schichten */
  for( l = 3; l < nr_layer; l++,p_in = p_temp_u,p_temp_u = p_u )
  { /* ueber alle (ohne die letzte) Ebenen */
    for( i = 0; i < *(net->p_nr_units + l - 1); i++,p_u++ )
    { /* ueber alle Neuronen der l. Schicht, berechne Aktivierung */
      *p_u = *(p_t++); /* mit Schwellenwert initialisieren */
      for( j = 0; j < *(net->p_nr_units + l - 2); j++ )
         /* ueber alle Eingabewerte */
         *p_u += *(p_w++) * *(p_in + j);
              /* Eingabe des Neurons i aufbauen */
```

```
      *p_u = activation(*p_u);
             /* Aktivierung des Neurons i berechnen */
    }
    /* Aktivierungen der Einheiten der 1. Ebene wurden berechnet */
  }
  /* berechne nun die Aktivierungen der letzten Ebene */
  for( i = 0; i < *(net->p_nr_units + 1 - 1); i++ )
  { /* ueber alle Zellen der letzten Ebene: berechne Aktivierung */
    *(out + i) = *(p_t++); /* mit Schwellenwert initialisieren */
    for( j = 0; j < *(net->p_nr_units + 1 - 2); j++ )
       /* ueber alle Eingabewerte */
       *(out + i) += *(p_w++) * *(p_in + j);
                  /* Eingabe des Neurons i aufbauen */
    *(out + i) = activation(*(out + i));
                 /* Aktivierung des Neurons i berechnen */
  }
  /* Die Aktivierungen der Einheiten der letzten Ebene wurden nun
     berechnet und das  Ausgabearray wurde korrekt gesetzt        */
  return; /* fertig */
} /* Ende von 'propagate(...)' */
```

In dieser Datei wurde zuerst die Aktivierungsfunktion `activation` deklariert und anschließend die Propagierungsfunktion `propagate` definiert.

Bem.:

- **in** ist ein Eingabewert der Funktion `propagate`:
  Die Funktion `propagate` holt die Netzeingabewerte aus dem Array, auf das **in** zeigt. Diese Feldvariable muß vor dem Aufruf vom Programmierer gesetzt werden. Die Größe des Arrays richtet sich nach der Anzahl der Eingabezellen des Neuronalen Netzes. Der *C*–Programmierer ist für eine ausreichende Größe selbst verantwortlich!

- **out** ist ein Ausgabewert der Funktion `propagate`:
  Die Funktion `propagate` legt die Ausgabewerte in dem Array ab, auf das **out** zeigt. Die Größe des Arrays richtet sich nach der Anzahl der Ausgabezellen des Neuronalen Netzes. Der *C*–Programmierer ist für eine ausreichende Größe selbst verantwortlich!

- **net** ist ein Eingabewert der Funktion `propagate`:
  Der Programmierer muß zur Berechnung der Netzantwort einen Zeiger auf die Struktur des Neuronalen Netzes übergeben. Diesen Zeiger liefert die Funktion, die durch den *save_c_net*–Befehl erzeugt wird.

9.5.3 C–Programm speichern: save_c_net

Mit dem *save_c_net*–Kommando kann der Benutzer das im System vorliegende Netz zum Einfügen in einer *C*–Quelldatei speichern. Die Syntax des Befehls lautet:

save_c_net_command ::= *"save_c_net"* **[file_handle | file_name].**

Bem.:

- **file_handle** ist ein Dateihandle, das ein vom Anwender geöffnetes File spezifiziert. In dieser Datei wird das Neuronale Netz zum Einfügen in eine *C*–Quelldatei abgelegt.
- **file_name** ist der Pfad und Name der Datei, in die das Neuronale Netz zum Einfügen in eine *C*–Quelldatei gespeichert wird (siehe Unterabschnitt 7.2.1). Falls das File vorher schon vorhanden war, wird dessen ursprünglicher Inhalt gelöscht.
- Wird kein Handle und kein Dateiname angegeben, so benutzt ProSiNN das normale Ein/Ausgabefenster als Ausgabemedium (es wird also **stdout** als Dateideskriptor verwendet).
- Bei der von *save_c_net* ausgegebenen Struktur ist folgendes anzumerken:
  - ProSiNN erzeugt zuerst die Funktion `init_neural_net`, in der alle benötigten Ebenen, Neuronen und Gewichte definiert werden. Der Ausgabewert dieser Funktion ist ein Zeiger auf die Netzstruktur.

    Diese Funktion muß genau ein Mal im *C*–Programm aufgerufen werden, um den Zeiger auf die Netzstruktur zu erhalten.

    Wenn in einem *C*–Programm mehr als ein Neuronales Netz verwendet werden soll, müssen die weiteren `init_neural_net`–Funktionen umbenannt werden, um Namenskollisionen bei der Kompilierung zu vermeiden.
  - Als nächstes wird die Funktion `activation` ausgegeben, die die Aktivierungsfunktion definiert. Der Ausgabewert dieser Funktion ist der Aktivierungswert des übergebenen Eingabewertes.

    Definiert der Benutzer eine eigene Aktivierungsfunktion, die die **ln**– oder **log**–Funktion beinhaltet, dann müssen diese Namen geändert werden. In *C* wird für die **ln**–Funktion **log** und für **log log10** verwendet.

    Werden in einem *C*–Programm mehrere Neuronale Netze mit unterschiedlichen Aktivierungsfunktionen verwendet, so müssen alle weiteren `activation`–Funktionen umbenannt werden, um Namenskollisionen bei der Kompilierung zu vermeiden. Diese Modifikationen führen dann aber auch zu einer Änderung der Propagierungsfunktion `propagate`!

Beispiel:
Im System befindet sich das im Beispiel des Unterabschnitts 9.4.2 verwendete Netz zur Berechnung des binären AND*–Operators. Dieses Netz soll als Teil eines* C*–Programms in der Datei* `and.c` *gespeichert werden. Der nachfolgende Befehl führt die verlangte Aufgabe durch:*

```
save_c_net and.c
```

Nach Ausführung des Kommandos befindet sich in der Datei `and.c` *folgender Inhalt:*

```
NEURAL_NET *init_neural_net()
{
  static double weight[] =
  { -5.8428779,
    -5.8438315,
    -6.5406577,
    -6.5395857,
    -14.092124,
    -17.286658
  };

  static double unit[2];

  static double theta[] =
  { 8.1999683,
    9.2644199,
    14.607000
  };

  static unsigned int nr_units[] =
  {
    2,
    2,
    1
  };

  static NEURAL_NET con;

  con.nr_layers = 3;
  con.p_nr_units = &nr_units[0];
  con.p_theta = &theta[0];
  con.p_unit = &unit[0];
  con.p_weight = &weight[0];
  return(&con);
}

double activation(x_1)

double x_1;

{
  return(
  1.0 / (1.0 + exp(-x_1))
        );
}
```

Nun soll durch ein C–Programm jede binäre Eingabekombination des Neuronalen Netzes berechnet und anschließend ausgegeben werden. Das folgende Quellprogramm löst diese Aufgabe:

```
/* Programm zur Ausgabe aller Netzantworten bei der Eingabe
          aller binaeren Eingabekombinationen               */

#include <stdio.h>
#include <math.h>
#include "neural_net.h" /* deklariert alle Strukturen */
#include "neural_net.c" /* beinhaltet die Propagierungsfunktion */
#include "and.c"        /* beinhaltet das AND-Netz
                           und die Aktivierungsfunktion */

void main()
{ double    in[2], /* Eingabearray fuer das Netz */
              out; /* Ausgabespeicher fuer das Netz */
  NEURAL_NET *net; /* Zeiger auf die Netzstruktur */

  net = init_neural_net(); /*    Netz wird initialisiert und der
                              Zeiger auf die Struktur wird gesetzt */

  in[0] = in[1] = 0.0;     /* 1. Eingabekombination setzen */
  propagate(in,&out,net);  /* Netzantwort berechnen */
  /* Eingabewerte und Netzantwort ausgeben */
  printf("Eingabewerte: x = 0.0 , y = 0.0; Ausgabewert: %#.8lg\n",out);

  in[0] = 0.0; in[1] = 1.0; /* 2. Eingabekombination setzen */
  propagate(in,&out,net); /* Netzantwort berechnen */
  /* Eingabewerte und Netzantwort ausgeben */
  printf("Eingabewerte: x = 0.0 , y = 1.0; Ausgabewert: %#.8lg\n",out);

  in[0] = 1.0; in[1] = 0.0; /* 3. Eingabekombination setzen */
  propagate(in,&out,net); /* Netzantwort berechnen */
  /* Eingabewerte und Netzantwort ausgeben */
  printf("Eingabewerte: x = 1.0 , y = 0.0; Ausgabewert: %#.8lg\n",out);

  in[0] = 1.0; in[1] = 1.0; /* 4. Eingabekombination setzen */
  propagate(in,&out,net); /* Netzantwort berechnen */
  /* Eingabewerte und Netzantwort ausgeben */
  printf("Eingabewerte: x = 1.0 , y = 1.0; Ausgabewert: %#.8lg\n",out);
}
```

Nach der Kompilation und dem Start des Programms werden die folgenden Ausgaben erzeugt:

```
Eingabewerte: x = 0.0 , y = 0.0; Ausgabewert: 5.2299709e-08
```

```
Eingabewerte: x = 0.0 , y = 1.0; Ausgabewert: 5.1035601e-07
Eingabewerte: x = 1.0 , y = 0.0; Ausgabewert: 5.1036017e-07
Eingabewerte: x = 1.0 , y = 1.0; Ausgabewert: 0.99999900
```

9.6 ProSiNN–Netz laden: readnet

Um ein Neuronales Netz, das mit dem *savenet*–Befehl gespeichert (oder vom Benutzer mit Hilfe eines Editors eingegeben) wurde, wieder zu laden, muß der Anwender die *readnet*–Anweisung verwenden. Die Syntax des Kommandos ist:

readnet_command ::= *"readnet"* **file_name.**

Bem.:

- **file_name** ist der Pfad und Name der Datei, in der das zu ladende Neuronale Netz gespeichert ist.
- Der Aufbau der Netzdatei muß mit dem Dateiformat, das in Unterabschnitt 9.4.1 beschrieben wird, übereinstimmen.
- Existiert schon ein Neuronales Netz im System, dann bricht der Befehl mit einer Fehlermeldung ab. In diesem Fall muß das *alte* Netz z.B. mit dem *free_net*–Befehl entfernt werden (siehe Abschnitt 9.9).

Beispiel:
Das folgende Kommando lädt das Netz, das im Beispiel des Unterabschnitts 9.4.2 erzeugt wurde, wieder in das System ein:

```
readnet test
```

9.7 Netzstruktur ändern

In diesem Abschnitt werden die Kommandos vorgestellt, die die Topologie eines im System vorhandenen Netzes modifizieren. Es handelt sich hier um Befehle, die eine Netzebene hinzufügen oder entfernen, sowie die Anzahl der Neuronen in einer Schicht ändern.

9.7.1 Eine Netzebene hinzufügen: add_layer

Um eine Netzebene hinzuzufügen, muß der Benutzer den *add_layer*–Befehl verwenden. Die Kommandosyntax lautet:

add_layer_command ::= *"add_layer"*.

Bem.:

- Die hinzugefügte Ebene wird zur neuen Ausgabeschicht des Netzes.
- Die Anzahl der Zellen in der neuen (Ausgabe–) Ebene ist gleich der Anzahl der Ausgabeneuronen in der bisherigen letzten Schicht.
- Alle neu hinzugefügten Neuronen (bzw. deren Schwellenwerte) und Gewichte werden mit dem in Abschnitt 9.3 eingeführten Zufallszahlengenerator initialisiert.

Beispiel:
Im Speicher sei ein dreischichtiges Neuronales Netz, das in der Eingabeebene 3, in der mittleren 4 und in der Ausgabeschicht 2 Neuronen enthält. Der folgende Befehl erhöht die Anzahl der Ebenen um eins:

```
add_layer
```

Jetzt hat das Netz die nachfolgende Struktur:

Netzebene	*Anzahl Neuronen in dieser Ebene*
1	*3*
2	*4*
3	*2*
4	*2*

Ein weiteres add_layer–*Kommando würde die folgende Netzstopologie erzeugen:*

Netzebene	*Anzahl Neuronen in dieser Ebene*
1	*3*
2	*4*
3	*2*
4	*2*
5	*2*

9.7.2 Eine Netzebene entfernen: sub_layer

Um eine Netzebene zu entfernen, muß der Benutzer den *sub_layer*–Befehl verwenden. Die Syntax ist:

sub_layer_command ::= *"sub_layer"*.

Bem.:

- Das Kommando entfernt die Ausgabeschicht des Netzes, sofern mehr als zwei Ebenen existieren; ansonsten bricht der Befehl mit einer Fehlermeldung ab.

- Die Anzahl der Neuronen in der bisherigen vorletzten Schicht wird so modifiziert, das sie mit der Anzahl der bisher vorhandenen Ausgabezellen übereinstimmt. Ist eine Veränderung der Neuronenanzahl notwendig, dann werden neue Zellen hinzugefügt (die neuen Schwellenwerte und Gewichte werden mit dem in Abschnitt 9.3 eingeführten Zufallszahlengenerator initialisiert) oder vom Ende entfernt.

Beispiel:
Im System sei ein Netz, das folgende Strukturmerkmale aufweist:

Netzebene	*Anzahl Neuronen in dieser Ebene*
1	*3*
2	*4*
3	*2*
4	*1*

Der nachfolgende Befehl erniedrigt die Anzahl der Ebenen um eins:

```
sub_layer
```

Jetzt hat das Netz die folgende Topologie:

Netzebene	*Anzahl Neuronen in dieser Ebene*
1	*3*
2	*4*
3	*1*

Ein weiteres sub_layer–*Kommando führt zu der nachfolgenden Netzstruktur:*

Netzebene	*Anzahl Neuronen in dieser Ebene*
1	*3*
2	*1*

Der nächste sub_layer–*Befehl würde eine Fehlermeldung erzeugen.*

9.7.3 Neuronenanzahl ändern: alter_unit_number

Will der Benutzer die Anzahl der Neuronen in einer Netzebene ändern, so muß er den *alter_unit_number*–Befehl verwenden. Die Kommandosyntax lautet:

alter_unit_number_command ::= *“alter_unit_number“ “in“* **layer_nr**
“to“ **nr_units.**
layer_nr ::= **int_expression.**
nr_units ::= **int_expression.**

Bem.:

- **layer_nr** gibt die Nummer der Netzebene an, in der die Neuronenanzahl geändert werden soll. Es muß gelten: **layer_nr** $\in \{1, 2, \ldots,$ **nr_of_layers**$\}$. Die globale Variable **nr_of_layers** enthält die Anzahl der Netzschichten.
- **nr_units** legt die neue Anzahl der Neuronen in der angegebenen Netzebene fest. Es muß gelten: **nr_units** ≥ 1.
- Beschreibt **layer_nr** die erste oder die letzte Netzschicht, so dürfen keine Trainingsmuster geladen sein. In diesem Fall bricht das Kommando mit einer Fehlermeldung ab.
- Wenn im System nur Testmuster (und keine Trainingsmuster) vorhanden sind und **layer_nr** die erste oder die letzte Netzschicht beschreibt, dann werden bei der Befehlsausführung alle Testmuster freigegeben.
- Erhöht sich die Anzahl der Zellen in der angegebenen Schicht, so werden alle neu hinzugefügten Neuronen (bzw. deren Schwellenwerte) und Gewichte mit dem in Abschnitt 9.3 eingeführten Zufallszahlengenerator initialisiert.

Beispiel:
Im System sei ein dreischichtiges Neuronales Netz, das in der Eingabeebene 3, in der mittleren 4 und in der Ausgabeschicht 2 Neuronen enthält. Der folgende Befehl erhöht die Anzahl der Neuronen in der ersten Ebene auf 13:

```
alter_unit_number in 1 to 13
```

Das anschließende Kommando setzt die Zellenanzahl der mittleren Schicht auf 1:

```
alter_unit_number in 2 ^ 1 to 13 - 12
```

Die Anweisung

```
alter_unit_number in 3 to 2
```

verändert die Anzahl der Ausgabeneuronen nicht. Jetzt enthält das Netz 13 Eingabezellen, ein Neuron in der mittleren Schicht und 2 Ausgabeneuronen.

9.8 Netzwerte benutzen oder ändern

In diesem Abschnitt wird noch einmal wiederholt, wie man auf netzinterne Werte zugreift (siehe auch Abschnitt 4.3).

- Die Anzahl der Neuronen in einer Netzebene kann mit Hilfe des Präfixes **nr_units** erhalten werden:

nr_units_val ::= *"nr_units" ":"* **layer_val.**

layer_val spezifiziert die Nummer der Netzebene, dessen Neuronenanzahl der Ausdruck beschreibt. Hier darf eine Konstante oder Variable vom Typ *layer* verwendet werden. Es muß gelten: **layer_val** $\in \{1, 2, \ldots,$ **nr_of_layers**$\}$. Die globale Variable **nr_of_layers** enthält die Anzahl der Netzschichten.
Der angesprochene Wert ist vom Typ *int* und kann nicht verändert werden!

- Auf den aktuellen Schwellenwert eines Neurons kann mit dem Präfix **theta** zugegriffen werden:

  theta_val ::= *"theta" ":"* **unit_val.**

  unit_val spezifiziert eine Konstante oder Variable vom Typ *unit*; dabei müssen Konstanten die in Unterabschnitt 4.2.5 auf Seite 57 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **layer_const** und **unit_number** können auch Variablen vom Typ *layer* oder *int* sein. Es dürfen aber keine zusammengesetzten Ausdrücke für **layer_const** und **unit_number** verwendet werden!
  Der angesprochene Wert vom Typ *double* kann auch geändert werden (Schwellenwerte eines Eingabeneurons sind nicht angesprechbar, da sie real nicht existieren)!

- Mit Hilfe des Präfixes **o_pj** kann auf den aktuellen Ausgabewert (Aktivierungswert) eines Neurons zugegriffen werden:

  o_pj_val ::= *"o_pj" ":"* **unit_val.**

  unit_val spezifiziert eine Konstante oder Variable vom Typ *unit*; dabei müssen Konstanten die in Unterabschnitt 4.2.5 auf Seite 57 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **layer_const** und **unit_number** können auch Variablen vom Typ *layer* oder *int* sein. Es dürfen aber keine zusammengesetzten Ausdrücke für **layer_const** und **unit_number** verwendet werden!
  Der angesprochene Wert vom Typ *double* kann auch geändert werden (Ausgabewerte eines Eingabeneurons sind nicht angesprechbar; sie sind identisch mit den entsprechenden Netzeingabedaten)!

- Auf das Fehlersignal δ_{pj} eines Neurons kann mit dem Präfix **delta_pj** zugegriffen werden:

  delta_pj_val ::= *"delta_pj" ":"* **unit_val.**

  unit_val spezifiziert eine Konstante oder Variable vom Typ *unit*; dabei müssen Konstanten die in Unterabschnitt 4.2.5 auf Seite 57 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **layer_const** und **unit_number** können auch Variablen vom Typ *layer* oder *int* sein. Es dürfen aber keine zusammengesetzten Ausdrücke für **layer_const** und **unit_number** verwendet werden!
  Der angesprochene Wert vom Typ *double* kann auch geändert werden (Fehlersignale eines Eingabeneurons sind nicht angesprechbar)!

- Der Änderungswert $\Delta\theta_j$ eines Neurons ist mit Hilfe des Präfixes **delta_theta** zugreifbar:

delta_theta_val ::= *"delta_theta"* *":"* **unit_val.**

unit_val spezifiziert eine Konstante oder Variable vom Typ *unit*; dabei müssen Konstanten die in Unterabschnitt 4.2.5 auf Seite 57 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **layer_const** und **unit_number** können auch Variablen vom Typ *layer* oder *int* sein. Es dürfen aber keine zusammengesetzten Ausdrücke für **layer_const** und **unit_number** verwendet werden!
Der angesprochene Wert vom Typ *double* kann auch geändert werden (Änderungswerte eines Eingabeneurons sind nicht angesprechbar, da sie real nicht existieren)!

- Auf das aktuelle Gewicht einer Verbindung kann mit dem Präfix **w** zugegriffen werden:

  w_val ::= *"w"* *":"* **weight_val.**

  weight_val spezifiziert eine Konstante oder Variable vom Typ *weight*; dabei müssen Konstanten die in Unterabschnitt 4.2.6 auf Seite 58 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **unit_const**$_{start\_layer}$ (für die Startebene) und **unit_const**$_{destination\_layer}$ (für die Zielebene) können auch Variablen vom Typ *unit* sein. Anstatt eine *unit*–Variable zu benutzen, kann auch eine Form aus *layer*– und *int*–Angaben verwendet werden. Diese Komponenten dürfen aber nicht aus zusammengesetzten Ausdrücken aufgebaut sein!
  Der angesprochene Wert vom Typ *double* kann auch geändert werden!

- Mit Hilfe des Präfixes **delta_w_val** kann auf den Änderungswert Δw_{ij} eines Gewichtes zugegriffen werden:

  delta_w_val ::= *"delta_w"* *":"* **weight_val.**

  weight_val spezifiziert eine Konstante oder Variable vom Typ *weight*; dabei müssen Konstanten die in Unterabschnitt 4.2.6 auf Seite 58 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **unit_const**$_{start\_layer}$ (für die Startebene) und **unit_const**$_{destination\_layer}$ (für die Zielebene) können auch Variablen vom Typ *unit* sein. Anstatt eine *unit*–Variable zu benutzen, kann auch eine Form aus *layer*– und *int*–Angaben verwendet werden. Diese Komponenten dürfen aber nicht aus zusammengesetzten Ausdrücken aufgebaut sein!
  Der angesprochene Wert vom Typ *double* kann auch geändert werden!

Beispiele:
Im System sei ein dreischichtiges Neuronales Netz, das in der Eingabeebene 3, in der mittleren 4 und in der Ausgabeschicht 2 Neuronen enthält. Außerdem sei eine Layervariable `l`*, eine Unitvariable* `u` *und eine Gewichtsvariable* `weight` *vorhanden. Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt einige Zugriffsmöglichkeiten auf die vorliegende Netzstruktur:*

```
>: set l = 2
>:set u = l:3
>:set weight = 1:3:u
```

```
>:print nr_units:l
4
>:set theta:u = 2.0
>:print theta:u
2.0000000
>:set o_pj:l:3 = 1.0
>:print o_pj:u
1.0000000
>:set delta_pj:u = 3.0
>:print delta_pj:u
3.0000000
>:set delta_theta:l:3 = 4.0
>:print delta_theta:u
4.0000000
>:set w:weight = 5.0
>:print w:weight
5.0000000
>:set delta_w:1:3:l:3 = 6.0
>:print delta_w:weight
6.0000000
>:
```

9.9 Netz freigeben: free_net

Um ein Neuronales Netz wieder freizugeben, muß der Anwender den *free_net*–Befehl benutzen. Seine Syntax lautet:

free_net_command ::= *“free_net“*.

Bem.:

- Das Kommando gibt das Neuronale Netz und alle damit verbundenen Variablen wieder frei. D.h. jede Netzebenen–, Neuronen– und Gewichtsvariable, sowie alle Trainings– und Testmuster werden aus dem System entfernt.
- Die Variablen **act_iterations**, **sum_iterations**, **act_error** und **old_error** werden auf 0 bzw. 0.0 gesetzt. Außerdem wird **eta** an **act_eta** und **alpha** an **act_alpha** zugewiesen.

Kapitel 10

Muster bearbeiten

Zum Trainieren oder Berechnen der Ausgabewerte eines Neuronalen Netzes werden Muster benötigt, die die Ein– und Ausgabedaten enthalten. Ein Trainingsmuster (Lernmuster) ist eine Datenstruktur, die die Netzeingabe– und die gewünschten Netzausgabedaten beinhaltet (Testmuster bestehen dagegen nur aus Eingabewerten).

In diesem Kapitel werden die Befehle vorgestellt, die ProSiNN zur Verwaltung von Mustern zur Verfügung stellt. Es werden Kommandos eingeführt, die Trainings– und Testmuster einlesen, Musterdaten ausgeben und einzelne (oder alle) Muster wieder freigeben. Ein weiterer Abschnitt beschäftigt sich mit der Benutzung musterinterner Werte.

10.1 Trainingsmuster laden

Ein Trainingsmuster besteht aus einer Folge von Eingabedaten (deren Quantität korrespondiert mit der Anzahl der Eingabeneuronen) und den dazugehörigen Ausgabedaten (deren Quantität entspricht der Anzahl der Ausgabeneuronen) des Netzes. Eine Menge von beliebig vielen Trainingsmustern bestimmt die Lernaufgabe, die ein Neuronales Netz lösen soll.

Die Trainingsmuster müssen immer aus einer Datei eingelesen werden, wobei sie nur dann verwendet werden können, wenn auch ein Neuronales Netz im System existiert. Diese Strukturen sind i.allg. als unveränderbar zu betrachten (mit dem Menüsystem lassen sich allerdings auch musterinterne Werte verändern; siehe Abschnitt 15.3). Das Format, das eine Musterdatei aufweisen muß, wird im ersten Unterabschnitt beschrieben. Im Anschluß wird das Kommando eingeführt, das eine Trainingsmuster–Datei einliest.

10.1.1 Das Trainingsmuster–Dateiformat

Die Struktur einer Datei, die Trainingsmuster enthält, wird durch folgende Syntax beschrieben:

pattern_file	::=	**statement_block** *"PATTERN:"* **pattern { pattern }.**
statement_block	::=	**{ statement** *";"* **}.**
pattern	::=	**in_pattern out_pattern [name]** *";"*.
in_pattern	::=	**val { val }.**
out_pattern	::=	**val { val }.**
val	::=	**symbolic_val | double_val.**
symbolic_val	::=	*"0"* | *"1"* | *"2"* | *"3"* | *"4"* | *"5"* | *"6"* | *"7"* | *"8"* | *"9"* | *"a"*.
double_val	::=	**„Konstante oder Variable vom Typ Double".**
name	::=	**„Bezeichner".**

Bem.:

- **statement_block** ist eine Folge von ProSiNN–Kommandos, die jeweils mit einem Semikolon abgeschlossen werden müssen. Der *quit*–Befehl (siehe Abschnitt 13.7) darf hier nicht verwendet werden; ansonsten sind aber alle Anweisungen erlaubt.

- **pattern** beschreibt ein Trainingsmuster, das aus den Netzeingabedaten **in_pattern**, den Netzausgabedaten **out_pattern** und einem optionalen Bezeichner **name**, mit dem später auf das Muster zugegriffen werden kann, besteht:

  - **in_pattern** definiert die Netzeingabedaten für dieses Muster. Die Anzahl der Eingabeneuronen bestimmt die Quantität von **val** im Nichtterminalsymbol **in_pattern**.

  - **out_pattern** definiert die Netzausgabedaten für dieses Muster. Die Anzahl der Ausgabeneuronen bestimmt die Quantität von **val** im Nichtterminalsymbol **out_pattern**.

  - **name** ist die optionale Angabe eines Bezeichners, der noch nicht im System definiert wurde. Mit diesem Bezeichner kann das Muster anschließend angesprochen werden.

- **val** steht für ein Datum des Musters. Mit Hilfe der Variablen **floatpattern** wird entschieden, von welchem Typ jede **val**–Angabe sein muß:

  - Ist der Wert von **floatpattern** gleich **off**, so wird die symbolische Schreibweise **symbolic_val** für **val** verlangt.

    Jedes Symbol in **symbolic_val** steht für eine Transformationsvorschrift. Der Wert des Musterbestandteiles kann aus der folgenden Tabelle entnommen werden:

symbolic_val	Wert von **symbolic_val**
0	**pat_min**
1	$0.1 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
2	$0.2 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
3	$0.3 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
4	$0.4 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
5	$0.5 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
6	$0.6 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
7	$0.7 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
8	$0.8 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
9	$0.9 * (\mathbf{pat\_max} - \mathbf{pat\_min}) + \mathbf{pat\_min}$
a	**pat_max**

pat_min und **pat_max** sind die globalen Variablen, die das Intervall definieren, aus dem die Werte der symbolischen Muster stammen. **pat_min** beschreibt dabei die untere und **pat_max** die obere Grenze. Der Benutzer kann diesen Variablen beliebige Gleitkommazahlen zuweisen, solange **pat_max** immer größer oder gleich **pat_min** ist.
Die Vorgabewerte sind **pat_min** = 0.0 und **pat_max** = 1.0.

  – Ist der Wert von **floatpattern** gleich **on**, so wird für jedes **val** eine Konstante oder Variable vom Typ *double* erwartet.

- In der gesamten Netzdatei dürfen beliebig geschachtelte Kommentare vorhanden sein.

- Nach den Netzdaten dürfen nur noch Kommentare folgen.

Beispiele:
Die beiden folgenden Beispiele zeigen jeweils eine Musterdatei, die die Daten zum Lernen des binären AND*-Operators enthält. In jeder Datei wird das letzte Muster mit dem Bezeichner* `name` *definiert; mit diesem Namen kann das Muster anschließend angesprochen werden.*

- *Die erste Datei definiert die Muster in symbolischer Schreibweise:*

```
/*     Diese Datei definiert die Muster fuer das AND-Problem     */
/* Alle Daten werden hier in symbolischer Schreibweise angegeben */
set pat_min = 0.0; /* unteren Wertebereich setzen */
set pat_max = 1.0; /* oberen Wertebereich setzen */
set floatpattern = off; /* symbolischen Modus einschalten */
PATTERN:
00 0;      /* 1. Muster */
0a 0;      /* 2. Muster */
a0 0;      /* 3. Muster */
aa a name; /* 4. Muster */
```

- *In der nachfolgenden Datei sind alle Musterangaben Konstanten vom Typ* double*:*

```
/*   Diese Datei definiert die Muster fuer das AND-Problem    */
/* Alle Daten werden hier in Fliesskommadarstellung angegeben */
set floatpattern = on; /* Fliesskomma-Modus einschalten */
PATTERN:
0.0 0.0 0.0;      /* 1. Muster */
0.0 1.0 0.0;      /* 2. Muster */
1.0 0.0 0.0;      /* 3. Muster */
1.0 1.0 1.0 name; /* 4. Muster */
```

10.1.2 Trainingsmuster laden: readpattern

Um eine Trainingsmusterdatei einzulesen, muß der Anwender den *readpattern*–Befehl verwenden. Die Syntax des Kommandos ist:

readpattern_command ::= *"readpattern"* **file_name.**

Bem.:

- **file_name** ist der Pfad und Name der Datei, die die zu ladenden Muster enthält.
- Der Aufbau der Musterdatei muß mit dem Dateiformat, das im letzten Unterabschnitt beschrieben wurde, übereinstimmen.
- Neu eingelesene Muster werden immer zu den alten hinzugefügt. Muster müssen also explizit freigegeben werden, wenn sie nicht mehr verwendet werden sollen.
- Ist kein Neuronales Netz im System vorhanden, dann bricht der Befehl mit einer Fehlermeldung ab.

Beispiel:
Eine der beiden Musterdateien aus den Beispielen des letzten Unterabschnitts sei unter dem Namen test.pattern *im Sekundärspeicher des Rechners vorhanden. Der folgende Ausschnitt des Ein/Ausgabefensters zeigt die Erzeugung des benötigten Neuronalen Netzes und das anschließende Laden der Musterdatei:*

```
>:layer 3: 2 , 2 , 1
>:readpattern test.pattern
>:
```

10.2 Testmuster laden

Ein Testmuster besteht aus einer Folge von Eingabedaten, deren Quantität mit der Anzahl der Eingabeneuronen korrespondiert, und einem Bezeichner, mit dem auf das Muster später zugegriffen werden muß.

Die Testmuster müssen immer aus einer Datei eingelesen werden, wobei sie nur dann verwendet werden können, wenn auch ein Neuronales Netz im System existiert. Diese Strukturen sind i.allg. als unveränderbar zu betrachten (mit dem Menüsystem lassen sich allerdings auch musterinterne Werte verändern; siehe Abschnitt 15.3). Das Format, das eine Musterdatei aufweisen muß, wird im ersten Unterabschnitt beschrieben. Im Anschluß wird das Kommando eingeführt, das eine Testmuster–Datei einliest.

10.2.1 Das Testmuster–Dateiformat

Die Struktur einer Datei, die Testmuster enthält, wird durch die folgende Syntax beschrieben:

test_pattern_file	::=	**statement_block** *“TEST-PATTERN:“* **test_pattern** { **test_pattern** }.
statement_block	::=	{ **statement** *“;“* }.
test_pattern	::=	**in_pattern name** *“;“*.
in_pattern	::=	**val** { **val** }.
val	::=	**symbolic_val** | **double_val**.
symbolic_val	::=	*“0“* | *“1“* | *“2“* | *“3“* | *“4“* | *“5“* | *“6“* | *“7“* | *“8“* | *“9“* | *“a“*.
double_val	::=	**„Konstante oder Variable vom Typ Double“.**
name	::=	**„Bezeichner“.**

Bem.:

- **statement_block** ist eine Folge von ProSiNN–Kommandos, die jeweils mit einem Semikolon abgeschlossen werden müssen. Der *quit*–Befehl (siehe Abschnitt 13.7) darf hier nicht verwendet werden; ansonsten sind aber alle Anweisungen erlaubt.
- **test_pattern** beschreibt ein Testmuster, das aus den Netzeingabedaten **in_pattern** und dem Bezeichner **name** besteht, mit dem später auf das Muster zugegriffen werden muß:
  - **in_pattern** definiert die Netzeingabedaten für dieses Muster. Die Anzahl der Eingabeneuronen bestimmt die Quantität von **val** im Nichtterminalsymbol **in_pattern**.
  - **name** ist die Angabe eines Bezeichners, der noch nicht im System definiert wurde. Mit diesem Bezeichner muß das Muster angesprochen werden.
- **val** steht für ein Datum des Musters. Mit Hilfe der Variablen **floatpattern** wird entschieden, von welchem Typ jede **val**–Angabe sein muß:
  - Ist der Wert von **floatpattern** gleich **off**, so wird die symbolische Schreibweise **symbolic_val** für **val** verlangt.

    Jedes Symbol in **symbolic_val** steht für eine Transformationsvorschrift. Aus der folgenden Tabelle kann der Wert des Musterbestandteiles entnommen werden. Die dort verwendeten Bezeichner **pat_min** und **pat_max** sind die globalen

Variablen, die das Intervall definieren, aus dem die Werte der symbolischen Muster stammen. **pat_min** beschreibt dabei die untere und **pat_max** die obere Grenze. Der Benutzer kann diesen Variablen beliebige Gleitkommazahlen zuweisen, solange **pat_max** immer größer oder gleich **pat_min** ist.
Die Vorgabewerte sind **pat_min** = 0.0 und **pat_max** = 1.0.

symbolic_val	Wert von **symbolic_val**
0	**pat_min**
1	0.1 ∗ (**pat_max** − **pat_min**) + **pat_min**
2	0.2 ∗ (**pat_max** − **pat_min**) + **pat_min**
3	0.3 ∗ (**pat_max** − **pat_min**) + **pat_min**
4	0.4 ∗ (**pat_max** − **pat_min**) + **pat_min**
5	0.5 ∗ (**pat_max** − **pat_min**) + **pat_min**
6	0.6 ∗ (**pat_max** − **pat_min**) + **pat_min**
7	0.7 ∗ (**pat_max** − **pat_min**) + **pat_min**
8	0.8 ∗ (**pat_max** − **pat_min**) + **pat_min**
9	0.9 ∗ (**pat_max** − **pat_min**) + **pat_min**
a	**pat_max**

– Ist der Wert von **floatpattern** gleich **on**, so wird für jedes **val** eine Konstante oder Variable vom Typ *double* erwartet.

- In der gesamten Netzdatei dürfen beliebig geschachtelte Kommentare vorhanden sein.

- Nach den Netzdaten dürfen nur noch Kommentare folgen.

Beispiele:
Die beiden folgenden Dateien enthalten Testmuster, die die gleichen Eingabedaten für ein Netz enthalten, wie die in den Beispielen des Unterabschnitts 10.1.1:

- *Die erste Datei definiert die Testmuster in symbolischer Schreibweise:*

```
/* Diese Datei definiert Testmuster fuer ein Netz,
        das 2 Eingabeneuronen aufweist           */
/* Alle Daten werden hier in symbolischer Schreibweise angegeben */
set pat_min = 0.0; /* unteren Wertebereich setzen */
set pat_max = 1.0; /* oberen Wertebereich setzen */
set floatpattern = off; /* symbolischen Modus einschalten */
TEST-PATTERN:
00 test_1; /* 1. Testmuster */
0a test_2; /* 2. Testmuster */
a0 test_3; /* 3. Testmuster */
aa test_4; /* 4. Testmuster */
```

- *In der nachfolgenden Datei sind alle Musterangaben Konstanten vom Typ* double*:*

```
/* Diese Datei definiert Testmuster fuer ein Netz,
         das 2 Eingabeneuronen aufweist            */
/* Alle Daten werden hier in Fliesskommadarstellung angegeben */
set floatpattern = on; /* Fliesskomma-Modus einschalten */
TEST-PATTERN:
0.0 0.0 test_1; /* 1. Testmuster */
0.0 1.0 test_2; /* 2. Testmuster */
1.0 0.0 test_3; /* 3. Testmuster */
1.0 1.0 test_4; /* 4. Testmuster */
```

10.2.2 Testmuster laden: read_test_pattern

Um eine Testmusterdatei einzulesen, muß der Anwender den *read_test_pattern*–Befehl verwenden. Die Syntax des Kommandos ist:

read_test_pattern_command ::= *"read_test_pattern"* **file_name.**

Bem.:

- **file_name** ist der Pfad und Name der Datei, die die zu ladenden Testmuster enthält.
- Der Aufbau der Musterdatei muß mit dem Dateiformat, das im letzten Unterabschnitt beschrieben wurde, übereinstimmen.
- Neu eingelesene Muster werden immer zu den alten hinzugefügt. Muster müssen also explizit freigegeben werden, wenn sie nicht mehr verwendet werden sollen.
- Ist kein Neuronales Netz im System vorhanden, dann bricht der Befehl mit einer Fehlermeldung ab.

Beispiel:
Unter dem Namen `test.test_pattern` *sei eine der beiden Beispieldateien des letzten Unterabschnitts im Sekundärspeicher des Rechners vorhanden. Der folgende Befehl lädt die Testmusterdatei in das System, sofern ein Neuronales Netz mit zwei Eingabeneuronen existiert:*

```
read_test_pattern test.test_pattern
```

10.3 Muster ausgeben: print_pattern

Zur Ausgabe eines Musters kann der Benutzer den *print_pattern*–Befehl verwenden. Die Kommandosyntax lautet:

print_pattern_command	::=	*"print_pattern"* **pattern_specification** [**file_handle**].
pattern_specification	::=	**name** | **pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
file_handle	::=	**int_expression.**

Bem.:

- **pattern_specification** gibt das auszugebende Muster an:
  - Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation kann für ein Trainings– und für ein Testmuster verwendet werden.
  - Wird **pattern_index** benutzt, dann muß damit die Nummer des Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß:
    pattern_index $\in \{1, \ldots,$ **nr_pattern**$\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird das Muster in das Ein/Ausgabefenster geschrieben.
- Die Angaben eines Musters werden in der folgender Reihenfolge ausgegeben (siehe auch das daran anschließende Beispiel):
  - Mustertyp
  - Mustername, sofern er existiert
  - Musternummer, sofern es ein Trainingsmuster ist
  - Netzeingabedaten des Musters
  - Netzausgabe des Musters, sofern es sich um ein Trainingsmuster handelt

Beispiel:
Das folgende Beispiel zeigt einen Auschnitt aus dem Ein/Ausgabefenster, wobei im System die Trainings– und Testmuster vorlagen, die in den Beispielen der Unterabschnitte 10.1.1 und 10.2.1 verwendet worden sind:

```
>:print_pattern 4 stdout
Trainingsmuster
Mustername: name
Musternummer: 4
1. Eingabewert: 1.0000000
2. Eingabewert: 1.0000000
1. Ausgabezielwert: 1.0000000
```

```
>::print_pattern name
Trainingsmuster
Mustername: name
Musternummer: 4
1. Eingabewert: 1.0000000
2. Eingabewert: 1.0000000
1. Ausgabezielwert: 1.0000000

>:print_pattern test_3
Testmuster
Mustername: test_3
1. Eingabewert: 1.0000000
2. Eingabewert: 0.0000000
>:
```

10.4 Musterwerte benutzen

Will der Benutzer ein eigenes Lernverfahren programmieren, so muß er auf einzelne Elemente eines Musters zugreifen können. Aus diesem Grund sollen noch einmal die Mechanismen wiederholt werden, die beim Zugriff auf musterinterne Werte Verwendung finden (siehe auch Unterabschnitt 4.3.4).

- Mit dem Präfix **pattern_in** kann auf ein Element der Netzeingabedaten eines Test– oder Trainingsmusters zugegriffen werden:

pattern_in_val	::=	*"pattern_in"* *":"* **pattern_specification** *":"* **pattern_element.**
pattern_specification	::=	**name** | **pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
pattern_element	::=	**int_val.**

  - **pattern_specification** gibt das Muster an, auf dessen Struktur zugegriffen werden soll:
    * Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation kann für ein Trainings– und für ein Testmuster verwendet werden.
    * Wird **pattern_index** benutzt, dann muß damit die Nummer eines Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß: **pattern_index** $\in \{1, \ldots, \mathbf{nr\_pattern}\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.

  * **pattern_element** beschreibt das Eingabeelement des Musters. Es muß gelten: **pattern_element** $\in \{1, 2, \ldots,$ **nr_in_units**$\}$. **nr_in_units** ist die globale Variable, die die Anzahl der Eingabeneuronen enthält.

Der angesprochene Wert vom Typ *double* kann nicht geändert werden!

- Mit dem Präfix **pattern_out** kann auf ein Element der Netzausgabedaten eines Trainingsmusters zugegriffen werden:

pattern_out_val	::=	*"pattern_out"* *":"* **pattern_specification** *":"* **pattern_element.**
pattern_specification	::=	**name** | **pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
pattern_element	::=	**int_val.**

  - **pattern_specification** gibt das Muster an, auf dessen Struktur zugegriffen werden soll:
    * Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist.
    * Wird **pattern_index** benutzt, dann muß damit die Nummer eines Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß: **pattern_index** $\in \{1, \ldots,$ **nr_pattern**$\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.
  - **pattern_element** beschreibt das Ausgabeelement des Musters. Es muß gelten: **pattern_element** $\in \{1, 2, \ldots,$ **nr_out_units**$\}$. **nr_out_units** ist die globale Variable, die die Anzahl der Ausgabeneuronen enthält.

Der angesprochene Wert vom Typ *double* kann nicht geändert werden!

Beispiel:
Im System sollen die Trainings- und Testmuster vorliegen, die in den Beispielen der Unterabschnitte 10.1.1 und 10.2.1 verwendet worden sind. Das folgende Beispiel zeigt einen Auschnitt aus dem Ein/Ausgabefenster:

```
>:print pattern_in:4:1
1.0000000
>:print pattern_in:name:2
1.0000000
>:print pattern_out:4:1
1.0000000
>:
```

Hier wurden nur interne Werte des vierten Trainingsmusters ausgegeben. Man hätte auch das Testmuster test_4 *verwenden können; weil es aber keine Netzausgabedaten enthält, würde das letzte Kommando dann zu einer Fehlermeldung führen.*

10.5 Muster freigeben

In diesem Abschnitt sollen die beiden Befehle vorgestellt werden, die ein speziell angegebenes oder alle Muster wieder aus dem System entfernen.

10.5.1 Ein einzelnes Muster freigeben: free_pattern

Zur Freigabe eines bestimmten Musters muß der Anwender den *free_pattern*–Befehl verwenden. Die Befehlssyntax ist:

free_pattern_command	::= *"free_pattern"* **pattern_specification.**
pattern_specification	::= **name** | **pattern_index.**
name	::= **„Bezeichner eines Musters".**
pattern_index	::= **int_val.**

Bem.:

- **pattern_specification** gibt das freizugebende Muster an:
  - Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation kann für ein Trainings– und für ein Testmuster verwendet werden.
  - Wird **pattern_index** benutzt, dann muß damit die Nummer des Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß:
    pattern_index $\in \{1, \ldots,$ **nr_pattern**$\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.
- Wurde ein Trainingsmuster freigegeben, dann sind die Zugriffsschlüssel der nachfolgenden Muster (**pattern_index**) anschließend um eins kleiner (siehe Beispiel).
- Ist das Muster mit einem Bezeichner definiert worden, so existiert dieser nach Ausführung des Kommandos nicht mehr.

Beispiele:
Im System seien die Trainings– und Testmuster, die in den Beispielen der Unterabschnitte 10.1.1 und 10.2.1 definiert worden sind:

- *Das folgende Kommando entfernt das zweite Trainingsmuster:*

  ```
  free_pattern 2
  ```

  Beim Zugriff auf das Muster, das vor diesem Befehl mit der Musterspezifikation 3 *angesprochen werden konnte, muß jetzt* 2 *verwendet werden. Auf das ehemalige vierte Trainingsmuster muß anschließend mit* 3 *oder dem Bezeichner* name *zugegriffen werden.*

- `free_pattern name`

  Diese Anweisung eliminiert das ursprüngliche vierte Trainingsmuster.

- *Der nachfolgende Befehl beseitigt das Testmuster* test_2 *aus dem System:*

  ```
  free_pattern test_2
  ```

10.5.2 Alle Muster freigeben: free_all_pattern

Mit dem Kommando *free_all_pattern* kann der Benutzer alle Muster wieder freigeben. Die Syntax lautet:

free_all_pattern_command ::= *"free_all_pattern"*.

Bem.:

Alle Bezeichner, die vorher einem Muster zugeordnet waren, sind nach Ausführung dieses Befehls undefiniert.

Kapitel 11

Neuronale Netze trainieren

In diesem Kapitel werden die Befehle vorgestellt, die Neuronale Netze trainieren, die Aktivierungsfunktion ändern, sowie die Netzantwort oder den Fehler der Ausgabezellen ausgeben bzw. einer Variablen zuweisen.

Im ersten Abschnitt wird die Verwendung der Variablen erläutert, die für das Training Neuronaler Netze wichtig sind. Die Kommandos, die ein Netz mit allen oder einem bestimmten Muster trainieren, werden danach präsentiert. Anschließend folgen die Anweisungen, die eine andere Aktivierungsfunktion setzen, deren Ableitung ändern oder wieder auf die 'normale' (sigmoide) Aktivierungsfunktion umschalten. Im vierten Abschnitt werden die Befehle vorgestellt, die die Netzausgabewerte oder die Fehler der Ausgabeneuronen auf ein Medium ausgeben (bzw. einer Feldvariablen zuweisen). Den Abschluß bildet ein Kommando, das bei einer gewissen Kategorie von Rechensystemen das Lernverfahren beschleunigt.

11.1 Verwendung der zugehörigen Systemvariablen

In diesem Abschnitt werden noch einmal die Systemvariablen beschrieben, die beim Training Neuronaler Netze verwendet werden:

- Die nachfolgenden Variablen beschreiben netzinterne Strukturen, die beim Training Neuronaler Netze für das Fehlermaß und die Terminierung des Lernverfahrens verwendet werden:

  act_error und **old_error**:

  > Diese beiden Fließkommavariablen (Datentyp *double*) beschreiben das aktuelle Fehlermaß, das nach dem letzten Trainingszyklus vorliegt (durch **act_error**) und das vor diesem Trainigszyklus gültige Fehlermaß (durch **old_error**). Sie können beliebig gesetzt werden, wobei ihre Werte aber größer als *0.0* sein müssen. Die vorgegebenen Werte sind *0.0*.

error_limit:

Diese Gleitkommavariable vom Datentyp *double* enthält die untere Fehlermaßgrenze, bei der das Trainingsverfahren beendet wird (siehe auch die nachfolgende Variable). Ihr dürfen beliebige Werte größer als *0.0* zugewiesen werden. Der Vorgabewert ist *0.001*.

error_mode:

Die von ProSiNN zur Verfügung gestellten Lernbefehle verwenden immer das in Gleichung 2.1 und 2.2 dargestellte Fehlermaß zur Modifikation der Gewichte und Schwellenwerte.

Mit dieser Variablen vom Datentyp *int* wird dagegen die Modalität festgelegt, wie ProSiNN das Fehlermaß zur Terminierung des Trainingsverfahrens verwenden soll. Dabei wird der Fehler je nach Modus berechnet und mit der unteren Fehlerschranke **error_limit** verglichen. Wenn der errechnete Wert darunterliegt, wird das Verfahren beendet.

An **error_mode** dürfen nur die folgenden Konstanten zugewiesen werden (p beschreibt die Anzahl der Trainingsmuster **nr_pattern**, n stellt die Anzahl der Ausgabeneuronen **nr_out_units** dar, t_{ij} ist der verlangte Zielwert des Neurons j bei Eingabe des Musters i und o_{ij} repräsentiert den real erhaltenen Ausgabewert dieses Neurons. E beschreibt in den folgenden Gleichungen den Wert, der der Variablen **act_error** vom System zugewiesen wird):

quadratisch:

Dies ist der Defaultwert der Variablen **error_mode**, das quadratische Fehlermaß:

$$E = \sum_{i=1}^{p} \sum_{j=1}^{n} (t_{ij} - o_{ij})^2$$

betragsmaessig:

Bei diesem Fehlermaß wird die absolute Differenz als Summe über alle Neuronen und Muster gebildet:

$$E = \sum_{i=1}^{p} \sum_{j=1}^{n} |t_{ij} - o_{ij}|$$

mittel_betrag:

Hierbei handelt es sich um die auf ein Neuron und ein Muster gemittelte absolute Abweichung über alle Neuronen und Muster:

$$E = \frac{1}{p} \sum_{i=1}^{p} \frac{1}{n} \sum_{j=1}^{n} |t_{ij} - o_{ij}|$$

max_betrag:

Bei diesem Fehlermaß wird die maximale Abweichung eines Ausgabeneurons über alle Muster berechnet:

$$E = \max_{\substack{i \in \{1,\ldots,p\} \\ j \in \{1,\ldots,n\}}} \{|t_{ij} - o_{ij}|\}$$

- Die folgenden Variablen vom Datentyp *double* beschreiben die Lernrate η und das Lernmoment α:

  eta und **act_eta**:

  Diese Fließkommavariablen repräsentieren die Lernrate η. Dabei steht **eta** für die vom System (oder Anwender) vorgegebene Lernrate. **act_eta** beschreibt die aktuelle Lernrate (η wird durch Erfolge und Fehlschläge beim Netztraining variiert; siehe Abschnitt 2.3). Der Benutzer kann ihnen beliebige Werte zuweisen, sofern sie größer als *0.0* sind. Der von beiden benutzte Vorgabewert ist *0.2*.

  alpha und **act_alpha**:

  Mit diesen Gleitkommavariablen wird das Lernmoment α beschrieben. Dabei steht **alpha** für das vom System (oder Anwender) vorgegebene Lernmoment. **act_alpha** repräsentiert das aktuelle Lernmoment (α wird durch Erfolge und Fehlschläge beim Netztraining variiert; siehe Abschnitt 2.3). Der Benutzer kann ihnen beliebige Werte zuweisen, sofern sie größer als *0.0* sind. Der von beiden benutzte Vorgabewert ist *0.9*.

- Zur Unterstützung des in Abschnitt 2.3 vorgestellten modifizierten Backpropagationalgorithmus werden noch drei Variablen vom Datentyp *double* benötigt:

  fail_limit:

  Diese Fließkommavariable gibt die in Punkt (2.b) des Abschnitts 2.3 genannte Prozentangabe wieder. Wird ihr Wert mit 100% multipliziert, dann repräsentiert das Ergebnis den oberen Grenzwert der erlaubten Abweichung. Der Defaultwert beträgt *1.04* (mit 100% multipliziert ergibt sich 104%; wenn das aktuelle Fehlermaß mehr als 4% über dem Wert des vorhergehenden Zyklus liegt, sieht das Verfahren diesen Lernzyklus als Mißerfolg an). Der Anwender darf der Variablen beliebige Werte zuweisen, sofern sie größer als *1.0* sind.

  phi:

  Diese Gleitkommavariable beschreibt den Erfolgslernratenfaktor Φ. Auch ihr darf vom Benutzer ein beliebiger Wert zugewiesen werden, sofern er größer als *1.0* ist. Defaultmäßig wird *1.05* verwendet.

  beta:

  Bei dieser Fließkommavariablen handelt es sich um den Mißerfolgslernratenfaktor β. Auch sie darf beliebig geändert werden, sofern gilt:
  $0 <$ **beta** < 1. Der Defaultwert beträgt *0.7* .

- Die beiden folgenden Variablen vom Datentyp *int* beschreiben die Anzahl der bisher ausgeführten Trainingszyklen:

  act_iterations:

  > Sie beschreibt die Anzahl der beim *letzten* Trainingsbefehl ausgeführten Zyklen. Sie darf auf einen beliebigen ganzzahligen Wert größer *0* gesetzt werden. Allerdings wird sie vor Ausführung des nächsten Trainingskommandos wieder auf *0* gesetzt.

  sum_iterations:

  > Diese Variable beschreibt die Gesamtanzahl *aller* bisher ausgeführten Trainingszyklen. Sie darf auf einen beliebigen ganzzahligen Wert größer *0* gesetzt werden. Beim Systemstart wird sie mit *0* initialisiert.

11.2 Netz trainieren

In diesem Abschnitt werden die beiden wichtigsten Befehle zum Trainieren von Neuronalen Netzen vorgestellt.

Zuerst wird das Kommando präsentiert, das das Lernverfahren mit allen im System vorhandenen Trainingsmustern durchführt. Die Anweisung, die das Netz mit einem bestimmten Muster trainiert, wird abschließend dargestellt.

11.2.1 Trainiere Netz mit allen Mustern: cycle

Um ein Neuronales Netz mit allen im System vorhandenen Lernmustern zu trainieren, muß der Benutzer den *cycle*–Befehl verwenden. Seine Syntax lautet:

cycle_command ::= *"cycle"* **max_cycles.**
max_cycles ::= **int_expression.**

Bem.:

- **max_cycles** gibt die maximale Anzahl der auszuführenden Trainingszyklen an. Es muß gelten: **max_cycles** ≥ 1.
- Das Kommando verwendet das Lernverfahren, das in Abschnitt 2.3 eingeführt wurde.
- Das Netz wird solange trainiert, bis eine der folgenden Situationen eintritt:
  - Das Fehlermaß **act_error** (das auf die Art und Weise berechnet wird, die durch die Variable **error_mode** festgelegt ist) unterschreitet die vorgegebene Fehlermaßgrenze **error_limit**.

    Wenn diese Abbruchbedingung eintritt, ist das Lernziel erreicht worden!

  - Die Anzahl der bisher ausgeführten Trainingszyklen **act_iterations** überschreitet die maximale Anzahl der auszuführenden Zyklen **max_cycles**, die dem Kommando als Argument übergeben wurde.

    Tritt diese Abbruchbedingung ein, so ist das Lernziel nicht erreicht worden!
  - Der Benutzer bricht den Befehl mit der Tastenkombination `Control-C` im Ein/Ausgabefenster ab.

- Parallel zum Training kann das System eine beliebige Anzahl von Protokollen in Dateien, dem normalen Ein/Ausgabefenster und in Grafiken ausgeben. Die Protokolldefinitionen müssen vor Aufruf des *cycle*–Befehls definiert werden (siehe Kapitel 12).
- Der Befehl bricht mit einer Fehlermeldung ab, wenn im System kein Neuronales Netz oder kein Trainingsmuster existiert.
- Will der Anwender beim Netztraining einen anderen Fehlermaßmodus benutzen, dann empfiehlt sich folgende Strategie:

  Das Netz wird solange mit dem *quadratischen* Fehlermaßmodus trainiert, bis die vorgegebene Fehlermaßgrenze **error_limit** unterschritten ist. Im Anschluß wird dann der gewünschte Fehlermaßmodus gesetzt und das Netz erneut bis zur Lösung des Problems trainiert.

  Durch diese Vorgehensweise erzielt man i.allg. eine schnellere Konvergenz des Lernverfahrens!

Beispiele:
Die beiden folgenden Beispiele zeigen Ausschnitte aus dem Ein/Ausgabefenster:

- *Es soll ein Neuronales Netz erzeugt und trainiert werden, das das binäre* `AND`*–Problem löst. Das quadratische Fehlermaß soll im Anschluß den Wert* 0.001 *unterschreiten:*

```
>:layer 3: 2 , 2 , 1
>:init_net
>:readpattern test.pattern
>:cycle 1000
>:print act_iterations
122
>:print act_error
0.00096888376
>:
```

  Zuerst wurde ein Netz mit drei Ebenen generiert, das in der ersten und der mittleren Schicht 2 und in der Ausgabeebene ein Neuron enthält. Anschließend wurde das Netz initialisiert. Der folgende Befehl hat die Trainingsmusterdatei eingelesen, die in Unterabschnitt 10.1.1 definiert wurde. Der verwendete cycle*–Befehl, der das Neuronale Netz mit maximal 1000 Zyklen trainiert, wurde anschließend eingegeben. Die Ausgabe der aktuellen Anzahl der durchgeführten Zyklen und des Fehlermaßes zeigen, daß das Netz die vorgegebene Aufgabenstellung erfüllt.*

- *Nun soll das Netz solange trainiert werden, bis bei der Netzeingabe eines beliebigen Musters die maximale Abweichung des Ausgabeneurons zum vorgegebenen Zielwert 0.001 beträgt. Anschließend soll das Netz in wieder einlesbarer Form in der Datei* `and.loesung` *gespeichert werden:*

```
>:set error_mode = max_betrag
>:cycle 1000
>:print act_iterations
132
>:print act_error
0.00097559540
>:savenet and.loesung
>:
```

Nachdem der gewünschte Fehlermaßmodus gesetzt wurde, ist das Netz erneut trainiert worden. Die Ausgabe der aktuellen Anzahl der durchführten Zyklen und des Fehlermaßes zeigen, daß das Netz mit Erfolg gelernt hat. Nun ist die Abweichung bei der Eingabe eines Musters maximal 0.0009756*. Anschließend wurde das Netz in der gewünschten Datei gespeichert.*

11.2.2 Trainiere Netz mit einem Muster: cycle_pattern

Will der Benutzer ein Neuronales Netz nur mit einem bestimmten Muster trainieren, dann muß er das *cycle_pattern*–Kommando benutzen. Die Befehlssyntax ist:

cycle_command	::=	*"cycle_pattern"* **pattern_specification max_cycles.**
pattern_specification	::=	**name | pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
max_cycles	::=	**int_expression.**

Bem.:

- **pattern_specification** gibt das Muster an, mit dem das Netz trainiert werden soll:
  - Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation darf hier nur für ein Trainingsmuster verwendet werden.
  - Wird **pattern_index** benutzt, dann muß damit die Nummer des Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int* sein, für die gelten muß: **pattern_index** $\in \{1, \ldots,$ **nr_pattern**$\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.
- **max_cycles** gibt die maximale Anzahl der auszuführenden Trainingszyklen an. Es muß gelten: **max_cycles** ≥ 1.

- Das Kommando verwendet das Lernverfahren, das in Abschnitt 2.3 eingeführt wurde.
- Das Netz wird solange mit dem angegebenen Muster trainiert, bis eine der folgenden Situationen eintritt:
  - Das Fehlermaß **act_error** (das auf die Art und Weise berechnet wird, die durch die Variable **error_mode** festgelegt ist) unterschreitet die vorgegebene Fehlermaßgrenze **error_limit**.

    Wenn diese Abbruchbedingung eintritt, ist das Lernziel erreicht worden!
  - Die Anzahl der bisher ausgeführten Trainingszyklen **act_iterations** überschreitet die maximale Anzahl der auszuführenden Zyklen **max_cycles**, die dem Kommando als Argument übergeben wurde.

    Tritt diese Abbruchbedingung ein, so ist das Lernziel nicht erreicht worden!
  - Der Benutzer bricht den Befehl mit der Tastenkombination `Control-C` im Ein/Ausgabefenster ab.
- Parallel zum Training kann das System eine beliebige Anzahl von Protokollen in Dateien, dem normalen Ein/Ausgabefenster und in Grafiken ausgeben. Die Protokolldefinitionen müssen vor Aufruf des *cycle_pattern*–Befehls definiert werden (siehe Kapitel 12).
- Der Befehl bricht mit einer Fehlermeldung ab, wenn im System kein Neuronales Netz vorhanden ist oder das angegebene Trainingsmuster nicht existiert.
- Diese Anweisung bietet dem Anwender zwei Vorteile:
  - Der Benutzer kann das Netz an ein bestimmtes Muster anpassen. Dies ist evtl. wünschenswert, wenn das Netz Schwierigkeiten beim Lernen *eines* Musters hat.
  - Will der Anwender ein Lernverfahren verwenden, das von dem in Abschnitt 2.3 genannten Algorithmus differiert, so kann er diesen Befehl verwenden.

    Er ist mit diesem Kommando in der Lage, ein eigenes Programm zu schreiben, das jedes einzelne Muster trainiert. Dabei sollte man folgendes beachten (siehe auch das anschließende Beispiel):
    * **max_cycles** sollte einen Wert größer als 1 haben; ansonsten ist es möglich, das überhaupt kein Lernerfolg erzielt wird.
    * **act_eta** sollte vor jedem neuen Lernbefehl mit einem anderen Muster auf den Wert von **eta** gesetzt werden, da die Lernrate in Abhängigkeit des Trainingserfolges variiert wird.
    * **act_alpha** sollte vor jedem neuen Lernbefehl mit einem anderen Muster auf den Wert von **alpha** gesetzt werden, da das Lernmoment in Abhängigkeit des Trainingserfolges variiert wird.

Beispiel:
Das folgende Beispiel zeigt ein Programm, das einen anderen Backpropagation–Algorithmus zur Lösung des binären AND*–Problems verwendet. Dabei soll das quadratische Fehlermaß anschließend kleiner als* 0.01 *sein:*

```
/* Backpropagation-Verfahren zur
     Loesung des AND-Problems    */
layer 3: 2 , 2 , 1; /* erzeuge das benoetigte Netz */
init_net; /* initialisiere das Netz */
readpattern test.pattern; /* lese die Musterdatei ein */
define int i; /* definiere die Musterzaehlvariable */
define int sum; /* definiere die Variable, die die Zyklen zaehlt */
define double error = 1.0; /* enthaelt den quadratischen Fehler */
while( error >= 0.01 )
{ /* solange noch nicht die untere Fehlergrenze erreicht ist: */
  set i = 1; /* setze 'i' auf das erste Muster */
  set error = 0.0; /* setze Fehler wieder zurueck */
  set act_eta = 0.2; /* setze eta auf den gewuenschten Wert */
  set alpha = 0.9; /* setze alpha auf den gewuenschten Wert */
  while( i <= 4 )
  { /* ueber alle Muster */
    cycle_pattern i 10; /* trainiere Netz mit diesem Muster 10 mal */
    set i = i + 1; /* setze 'i' auf das naechste Muster */
    set error = error + act_error; /* setze Gesamtfehler zusammen */
  };
  print "Fehler: ";print error;println;
  set sum = sum + 1; /* wieder einen Zyklus verarbeitet */
};
print "Die Anzahl der Trainingszyklen ist ";
print sum; println;
print "Der quadratische Fehler ueber alle Muster ist ";
print error;
println;
/* fertig */
```

Dieses Programm sei in der Datei `test` *gespeichert. Der folgende Auszug aus dem normalen Ein/Ausgabefenster zeigt den Aufruf des Programms und dessen Ausgaben:*

```
>:load test
Fehler: 0.32665446
Fehler: 0.53696761
Fehler: 0.50270252
Fehler: 0.48213833
Fehler: 0.48266097
Fehler: 0.62349002
...................
Fehler: 0.010430975
Fehler: 0.010330472
Fehler: 0.010231707
Fehler: 0.010134637
Fehler: 0.010039222
Fehler: 0.0099454220
```

```
Die Anzahl der Trainingszyklen ist 161
Der quadratische Fehler ueber alle Muster ist 0.0099454220
>:
```

11.3 Aktivierungsfunktion ändern

In diesem Abschnitt werden die Befehle zur Bearbeitung der Aktivierungsfunktion eingeführt.

Zuerst wird die Anweisung vorgestellt, die eine beliebige Aktivierungsfunktion setzt. Im zweiten Abschnitt wird das Kommando präsentiert, das die Ableitung der neuen Aktivierungsfunktion ändert. Den Abschluß bildet der Befehl, der wieder auf die normale, sigmoide Aktivierungsfunktion umschaltet.

11.3.1 Neue Aktivierungsfunktion setzen: activation

Um im ProSiNN–System eine andere Aktivierungsfunktion zu verwenden, muß der Anwender das *activation*–Kommando benutzen. Die Befehlssyntax lautet:

activation_command	::= “*activation(*“ **var_name** “*)*“ “=“ **funct_expression.**
var_name	::= **„Bezeichner (Funktionsvariable)“.**
funct_expression	::= **term** { (“+“ | “–“) **term** }.
term	::= **factor** { (“*“ | “/“) **factor** }.
factor	::= **power** { „^ “ **power** }.
power	::= [“–“] **scalar.**
scalar	::= “*(*“ **funct_expression** “*)*“ | **function.**
function	::= “*sin(*“ **funct_expression** “*)*“ |
	“*cos(*“ **funct_expression** “*)*“ |
	“*tan(*“ **funct_expression** “*)*“ |
	“*asin(*“ **funct_expression** “*)*“ |
	“*acos(*“ **funct_expression** “*)*“ |
	“*atan(*“ **funct_expression** “*)*“ |
	“*sinh(*“ **funct_expression** “*)*“ |
	“*cosh(*“ **funct_expression** “*)*“ |
	“*tanh(*“ **funct_expression** “*)*“ |
	“*sqrt(*“ **funct_expression** “*)*“ |
	“*ln(*“ **funct_expression** “*)*“ |
	“*log(*“ **funct_expression** “*)*“ |
	var_or_const |
	own_function |
	var_name.
var_or_const	::= **„Eine Konstante oder Variable (int oder double)“**
own_function	::= **„Eine vom Benutzer definierte Funktion (vom Typ int oder double). Syntax: siehe Bemerkung.“**

Bem.:

- **var_name** ist das Funktionsargument. Dieser Name ist nur lokal bei diesem Befehl bekannt. D.h. es können beliebige — auch vordefinierte — Bezeichner verwendet werden; die ursprüngliche Bedeutung wird kurzfristig verdeckt!
- Bei der Definition können die beiden Datentypen *int* und *double* beliebig miteinander vermischt werden.
- Indem z.B. Operationen auf Konstanten zusammengefaßt werden, versucht ProSiNN **funct_expression** zu vereinfachen. Dabei werden aber Teilausdrücke nicht permutiert, so daß eine lokale und keine globale Vereinfachung stattfindet.
- Für **own_function** gilt die gleiche Syntax wie bei einer Funktionsauswertung (siehe Unterabschnitt 5.1.4); es darf aber zusätzlich noch das in der Definition vorkommende Funktionsargument verwendet werden!
- Das Kommando erzeugt die neue Aktivierungsfunktion und deren Ableitung:
  - Die Aktivierungsfunktion wird mit dem Bezeichner **activation_funct** als Funktion vom Typ *double* gespeichert.
  - Mit dem Bezeichner **der_activation_funct** wird die Ableitung der Aktivierungsfunktion gespeichert (vom Typ *double*).

  Beide Funktionen können vom Benutzer nicht durch den *free_var*– oder den *free*–Befehl (siehe Abschnitt 13.4.2) gelöscht werden!
  War der Bezeichner **activation_funct** oder **der_activation_funct** schon vor dem *activation*–Befehl im System definiert, dann wird die ursprünglich damit verbundene Variable gelöscht und mit der Aktivierungsfunktion bzw. der Ableitung neu definiert!
- Dieser Befehl ist immer ausführbar! Es ist also nicht nötig, vorher ein Neuronales Netz im System zu definieren.
- Man beachte, daß eine eigene Aktivierungsfunktion den Verlauf des Netztrainings beeinflußt. Zum einen ist die Rechengenauigkeit nicht so hoch wie die der vordefinierten sigmoiden Funktion und zum anderen wird bei der Berechnung mehr Zeit benötigt.

Beispiele:
Die folgenden Beispiele zeigen Ausschnitte aus dem Ein/Ausgabefenster:

- *Es soll eine lineare Aktivierungsfunktion (Identität) erzeugt werden. Im Anschluß ist die Funktion und deren Ableitung auszugegeben:*

```
>:activation( x ) = x
>:print_function activation_funct
activation_funct(x_1) = x_1
>:print_function der_activation_funct
der_activation_funct(x_1) = 1.00000
>:
```

- *Nun soll x^3 als Aktivierungsfunktion verwendet werden:*

```
>:activation( x ) = x ^ 3
>:print_function activation_funct
activation_funct(x_1) = x_1^3.00000
>:print_function der_activation_funct
der_activation_funct(x_1) = x_1^3.00000*1.00000/x_1*3.00000
>:
```

- *Das letzte Beispiel zeigt eine andere Möglichkeit, mit der wieder auf die sigmoide Aktivierungsfunktion umgeschaltet wird:*

```
>:activation( x ) = 1 / ( 1 + exp(-x) )
>:print_function activation_funct
activation_funct(x_1) = 1.00000/(1.00000+exp(-x_1))
>:print_function der_activation_funct
der_activation_funct(x_1) = --exp(-x_1)/(1.00000+exp(-x_1))^2.00000
>:
```

Will der Anwender wieder die sigmoide Aktivierungsfunktion benutzen, so sollte er den normal_activation*–Befehl (siehe Unterabschnitt 11.3.3) verwenden. Dann wird die besser (genauer und schneller) intern implementierte sigmoide Funktion benutzt!*

11.3.2 Ableitung der Aktivierungsfunktion ändern: derivation_of_activation_funct

Nach der Erzeugung einer neuen Aktivierungsfunktion kann die dazugehörige Ableitung mit dem *derivation_of_activation_funct*–Befehl geändert werden. Die entsprechende Kommandosyntax ist:

derivation_of_activation_funct_command	::=	*“derivation_of_activation_funct(“* **var_name** *“)“* *“=“* **funct_expression.**
var_name	::=	**„Bezeichner (Funktionsvariable)“.**
funct_expression	::=	**„siehe Bemerkungen“**

Bem.:

- **var_name** ist das Funktionsargument. Dieser Name ist nur lokal bei diesem Befehl bekannt. D.h. es können beliebige — auch vordefinierte — Bezeichner verwendet werden; die ursprüngliche Bedeutung wird kurzfristig verdeckt!

- Für **funct_expression** gilt dieselbe Syntax, die im letzten Unterabschnitt verwendet worden ist.

- Dieses Kommando setzt die gewünschte Aktivierungsfunktionsableitung in der Funktionsvariablen mit dem Bezeichner **der_activation_funct** (vom Typ *double*).
- Dieser Befehl ist nur ausführbar, wenn vorher mit der *activation*–Anweisung eine Aktivierungsfunktion definiert wurde!
- Dieses Kommando bietet folgende positive Aspekte für den Benutzer:
  - Durch den *activation*–Befehl können bei der Ableitung singuläre Stellen auftreten (die real gar nicht vorhanden sein müssen; siehe Beispiel). Mit dieser Anweisung lassen sich solche Definitionslücken entfernen!
  - Die Ableitung der Aktivierungsfunktion kann mit diesem Befehl evtl. so vereinfacht werden, daß eine Beschleunigung des Lernverfahrens erzielt wird (siehe Beispiel).
  - Der Anwender kann Experimente mit verschiedenen Ableitungen der Aktivierungsfunktion durchführen.

Beispiel:
In diesem Beispiel wird die Aktivierungsfunktion x^3 gesetzt und anschließend die dazugehörige Ableitung geändert:

```
>:activation( x ) = x ^ 3
>:print_function activation_funct
activation_funct(x_1) = x_1^3.00000
>:print_function der_activation_funct
der_activation_funct(x_1) = x_1^3.00000*1.00000/x_1*3.00000
>:derivation_of_activation_funct( x ) = 3 * x ^ 2
>:print_function der_activation_funct
der_activation_funct(x_1) = 3.0000000*x_1^2.0000000
>:
```

Die ursprüngliche Ableitung der Aktivierungsfunktion beinhaltet eine singuläre Stelle (beim Argument 0). Würde das System den Aktivierungswert an der Stelle 0 berechnen, dann wäre ProSiNN mit einer Fehlermeldung ausgestiegen! Durch das Ändern der Aktivierungsfunktionsableitung wird die singuläre Stelle behoben und ein Geschwindigkeitszuwachs beim Netztraining erzielt (weniger Berechnungen).

11.3.3 'Normale' Aktivierungsfunktion setzen: normal_activation

Will der Benutzer wieder die sigmoide Aktivierungsfunktion setzen, so sollte er den Befehl *normal_activation* verwenden. Die Kommandosyntax ist:

normal_activation_command ::= *"normal_activation"*.

Bem.:

- Der Anwender sollte auf keinen Fall den *activation*–Befehl benutzen, um die sigmoide Aktivierungsfunktion zu setzen! Durch das *normal_activation*–Kommando wird nämlich die besser intern implementierte (genauer und schneller) sigmoide Funktion verwendet.
- Diese Anweisung ist immer ausführbar!

11.4 Netzantworten verarbeiten

In diesem Abschnitt werden die Befehle behandelt, die die Netzantwort eines Musters oder den dazugehörigen Fehler verarbeiten.

Im ersten Unterabschnitt wird die Anweisung vorgestellt, die ein Eingabemuster und dessen Netzantwort auf ein Medium ausgibt. Daran schließen sich die Kommandos an, die den Fehler der Netzantwort ausgeben oder einer Feldvariablen zuweisen.

11.4.1 Netzantwort ausgeben: reply_to

Zur Ausgabe eines beliebigen Musters und dessen Netzantwort, muß der Anwender den *reply_to*–Befehl verwenden. Die Syntax des Kommandos lautet:

reply_to_command	::=	*"reply_to"* **pattern_specification**
		[file_handle].
pattern_specification	::=	**name | pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
file_handle	::=	**int_expression.**

Bem.:

- **pattern_specification** gibt das Muster an, dessen Netzantwort auszugeben ist:
  - Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation kann für ein Trainings– und für ein Testmuster verwendet werden.
  - Wird **pattern_index** benutzt, dann muß damit die Nummer des Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß:
    pattern_index $\in \{1, \ldots, \textbf{nr\_pattern}\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird die Netzantwort in das Ein/Ausgabefenster geschrieben.

- Die einzelnen Angaben werden in der folgender Reihenfolge ausgegeben (siehe auch das daran anschließende Beispiel):
  - Mustertyp
  - Mustername, sofern er existiert
  - Musternummer, sofern es ein Trainingsmuster ist
  - Netzeingabedaten des Musters
  - Netzantwort auf dieses Muster
- Der Befehl bricht mit einer Fehlermeldung ab, wenn im System kein Neuronales Netz vorhanden ist oder das angegebene Muster nicht existiert.

Beispiel:
Im System sei das Neuronale Netz des letzten Beispiels in Unterabschnitt 11.2.1 und die dort verwendeten Trainings– und Testmuster. Das folgende Beispiel zeigt einen Auschnitt aus dem Ein/Ausgabefenster, bei dem der Benutzer die Netzantworten des ersten und vierten Trainingsmusters, sowie des Testmusters `test_2` *ausgegeben hat:*

```
>:reply_to 1
Trainingsmuster
Musternummer: 1
1. Eingabewert: 0.0000000
2. Eingabewert: 0.0000000
Antwort des Netzes auf das Muster:
1. Ausgabewert: 5.2299709e-08

>:reply_to name stdout
Trainingsmuster
Mustername: name
Musternummer: 4
1. Eingabewert: 1.0000000
2. Eingabewert: 1.0000000
Antwort des Netzes auf das Muster:
1. Ausgabewert: 0.99999900

>:reply_to test_2
Testmuster
Mustername: test_2
1. Eingabewert: 0.0000000
2. Eingabewert: 1.0000000
Antwort des Netzes auf das Muster:
1. Ausgabewert: 5.1035601e-07

>:
```

11.4.2 Fehler der Netzantwort ausgeben: print_error_of

Um den absoluten Fehler der Ausgabeneuronen bei der Netzantwort eines Trainingsmusters auszugeben, muß der Benutzer die *print_error_of*-Anweisung verwenden. Die Kommandosyntax ist:

print_error_of_command	::=	*"print_error_of"* **pattern_specification** [**file_handle**].
pattern_specification	::=	**name** | **pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
file_handle	::=	**int_expression.**

Bem.:

- **pattern_specification** gibt das Trainingsmuster an, dessen Fehler bei der Netzantwort auszugeben ist:
  - Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation darf hier nur für ein Trainingsmuster verwendet werden.
  - Wird **pattern_index** benutzt, dann muß damit die Nummer des Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß: **pattern_index** $\in \{1, \ldots,$ **nr_pattern**$\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird der Fehler der Netzantwort in das Ein/Ausgabefenster geschrieben.
- Die einzelnen Angaben werden in der folgender Reihenfolge ausgegeben (siehe auch das daran anschließende Beispiel):
  - Mustertyp (hier: Trainingsmuster)
  - Mustername, sofern er existiert
  - Musternummer, sofern es ein Trainingsmuster ist
  - Netzeingabedaten des Musters
  - Fehler der Netzantwort
- Der Befehl bricht mit einer Fehlermeldung ab, wenn im System kein Neuronales Netz vorhanden ist oder das angegebene Muster nicht existiert.

Beispiel:
Im System sei das Neuronale Netz des letzten Beispiels in Unterabschnitt 11.2.1 und die dort verwendeten Trainingsmuster. Das folgende Beispiel zeigt einen Auschnitt aus dem Ein/Ausgabefenster, bei dem der Benutzer die Fehler der Netzantworten des ersten und vierten Trainingsmusters ausgegeben hat:

```
>:print_error_of 1
Trainingsmuster
Musternummer: 1
1. Eingabewert: 0.0000000
2. Eingabewert: 0.0000000
Absoluter Fehler fuer jedes Ausgabeneuron (mit Vorzeichen):
1. Fehlerwert: -5.2299709e-08

>:print_error_of name stdout
Trainingsmuster
Mustername: name
Musternummer: 4
1. Eingabewert: 1.0000000
2. Eingabewert: 1.0000000
Absoluter Fehler fuer jedes Ausgabeneuron (mit Vorzeichen):
1. Fehlerwert: 9.9937478e-07

>:
```

11.4.3 Fehler der Netzantwort einem Array zuweisen: error_of

Die absoluten Fehler der Ausgabeneuronen bei der Netzantwort eines Trainingsmusters lassen sich mit dem *error_of*-Befehl einer Feldvariablen vom Typ *double* zuweisen. Die Kommandosyntax lautet:

error_of_command	::=	*"error_of"* **pattern_specification array_name.**
pattern_specification	::=	**name** | **pattern_index.**
name	::=	**„Bezeichner eines Musters".**
pattern_index	::=	**int_val.**
array_name	::=	**„Bezeichner einer Double–Feldvariablen".**

Bem.:

- **pattern_specification** gibt das Trainingsmuster an, dessen Fehler bei der Netzantwort der Feldvariablen **array_name** zugewiesen werden soll:
  - Wird das Nichtterminalsymbol **name** zur Angabe des Musters verwendet, so muß dies der Bezeichner sein, der beim Einlesen des Musters definiert worden ist. Diese Spezifikation darf hier nur für ein Trainingsmuster verwendet werden.
  - Wird **pattern_index** benutzt, dann muß damit die Nummer des Trainingsmusters angegeben werden. **pattern_index** ist eine Konstante oder Variable (oder ein Feldelement) vom Datentyp *int*, für die gelten muß:
    pattern_index $\in \{1, \ldots, \mathbf{nr\_pattern}\}$. Dabei ist **nr_pattern** die globale Variable, die die Anzahl der Trainigsmuster enthält.

- **array_name** muß der Bezeichner einer Feldvariablen vom Typ *double* sein. Die Anzahl der Feldelemente muß größer oder gleich **nr_out_units** sein (**nr_out_units** enthält die Anzahl der Ausgabeneuronen).
- Der Befehl bricht mit einer Fehlermeldung ab, wenn im System kein Neuronales Netz vorhanden ist oder das angegebene Muster nicht existiert.

Beispiel:
Im System sei das Neuronale Netz des letzten Beispiels in Unterabschnitt 11.2.1 und die dort verwendeten Trainingsmuster. Das folgende Beispiel zeigt einen Auschnitt aus dem Ein/Ausgabefenster, bei dem der Benutzer die Fehler der Netzantworten des ersten und vierten Trainingsmusters an die Double–Feldvariable `da` *zugewiesen und anschließend ausgegeben hat:*

```
>:define array double da[1]
>:error_of 1 da
>:print da[0]
-5.2299709e-08
>:print_error_of 1
Trainingsmuster
Musternummer: 1
1. Eingabewert: 0.0000000
2. Eingabewert: 0.0000000
Absoluter Fehler fuer jedes Ausgabeneuron (mit Vorzeichen):
1. Fehlerwert: -5.2299709e-08


>:error_of name da
>:print da[0]
9.9937478e-07
>:print_error_of name
Trainingsmuster
Mustername: name
Musternummer: 4
1. Eingabewert: 1.0000000
2. Eingabewert: 1.0000000
Absoluter Fehler fuer jedes Ausgabeneuron (mit Vorzeichen):
1. Fehlerwert: 9.9937478e-07

>:
```

11.5 Schnelles Netztraining: fast_cycle

ProSiNN bietet dem Benutzer den *fast_cycle*–Befehl an, der mit einer *Festkommaarithmetik* Neuronale Netze trainiert. Bei einem Rechensystem, das nicht mit einem Co–Prozessor für Gleitkommaberechnungen ausgestattet ist, wird mit dieser Anweisung eine Beschleunigung des Lernverfahrens erzielt (durch schnellere Berechnungen). Die Syntax des Befehls ist:

fast_cycle_command ::= *"fast_cycle"* **max_cycles.**
max_cycles ::= **int_expression.**

Bem.:

- **max_cycles** gibt die maximale Anzahl der auszuführenden Trainingszyklen an. Es muß gelten: **max_cycles** ≥ 1.
- Das Kommando verwendet das Lernverfahren, das in Abschnitt 2.3 eingeführt wurde.
- Das Netz wird solange trainiert, bis eine der folgenden Situationen eintritt:
  - Das Fehlermaß **act_error** (das auf die Art und Weise berechnet wird, die durch die Variable **error_mode** festgelegt ist) unterschreitet die vorgegebene Fehlermaßgrenze **error_limit**.

    Wenn diese Abbruchbedingung eintritt, ist das Lernziel erreicht worden!
  - Die Anzahl der bisher ausgeführten Trainingszyklen **act_iterations** überschreitet die maximale Anzahl der auszuführenden Zyklen **max_cycles**, die dem Kommando als Argument übergeben wurde.

    Tritt diese Abbruchbedingung ein, so ist das Lernziel nicht erreicht worden!
  - Der Benutzer bricht den Befehl mit der Tastenkombination `Control-C` im Ein/Ausgabefenster ab.
- Parallel zur Befehlsausführung kann das System *eingeschränkte* Protokollausgaben (nur Textprotokolle; siehe nächstes Kapitel) vornehmen:

  Es wird immer nur das erste vom Anwender definierte Protokoll benutzt. Aus dieser Struktur wird der Zyklenabstand der einzelnen Ausgabezeilen und das Ausgabemedium entnommen. Jede Protokollzeile besteht aus der Angabe der Zyklusnummer und den Inhalten der globalen Variablen **act_error** und **act_eta** (siehe auch Unterabschnitt 12.2.1).
- Der Befehl bricht mit einer Fehlermeldung ab, wenn im System kein Neuronales Netz oder kein Trainingsmuster existiert.
- Dieses Kommando wird (wahrscheinlich) langsamer als der *cycle*–Befehl verarbeitet, wenn das zugrundeliegende Rechensystem mit einem Co–Prozessor für Gleitkommaberechnungen ausgestattet ist. Deshalb sollte man bei diesen Systemen auf die *fast_cycle*–Anweisung verzichten!
- Auf Grund der Festkommaarithmetik kann das System nur mit einer bestimmten Genauigkeit rechnen:

  Die Festkommarithmetik basiert auf einem 32 Bit großen Datenwort. Es werden 15 Bit als Vorkomma– und 16 Bit als Nachkommawert verwendet. D.h. die größte absolut darstellbare Zahl ist $2^{15} - 2^{-16}$ (32767.9999847) und die kleinste absolut darstellbare Zahl ist 2^{-16} (1.5258789e-05).

  Aus diesen Beschränkungen ergeben sich die folgenden Konventionen:

  - **error_limit** sollte nie kleiner als 0.001 gesetzt werden. Wird dies nicht eingehalten, dann kann das System in der Regel auf Grund der Rechengenauigkeit keine Lösung des Problems finden. Nach dem erfolgreichen Lernen läßt sich der *cycle*–Befehl zur Berechnung einer besseren Lösung verwenden.
  - **eta** und **alpha** sollten — weil sie intern als Multiplikand verwendet werden — nie kleiner als 0.001 gesetzt werden.
  - **fail_limit**, **phi** und **beta** sollten — weil sie intern als Multiplikand verwendet werden — nie kleiner als 0.001 gesetzt werden.
  - Das Netz sollte nur mit Werten initialisiert werden, die für das System im Rahmen der Genauigkeit noch auflösbar sind.
  - Die Muster sollten im Bereich der Rechengenauigkeit liegen.

- Will der Anwender beim Netztraining einen anderen Fehlermaßmodus benutzen, dann empfiehlt sich folgende Strategie:

  Das Netz wird solange mit dem *quadratischen* Fehlermaßmodus trainiert, bis die vorgegebene Fehlermaßgrenze **error_limit** unterschritten ist. Im Anschluß wird dann der gewünschte Fehlermaßmodus gesetzt und das Netz erneut bis zur Lösung des Problems trainiert.

  Durch diese Vorgehensweise erzielt man i.allg. eine schnellere Konvergenz des Lernverfahrens!

Beispiel:
Das nachfolgende Beispiel zeigt eine Möglichkeit, wie das binäre AND*–Problem mit dem* fast_cycle*–Befehl gelöst werden kann. Die im Beispiel des Unterabschnitts 10.1.1 eingeführten Trainingsmuster seien in der Datei* `and.pat` *gespeichert. Das Netz soll solange trainiert werden, bis das quadratische Fehlermaß* 0.001 *unterschreitet. Die folgenden Zeilen zeigen den entsprechenden Ausschnitt aus dem Ein/Ausgabefenster:*

```
>:layer 3: 2 , 2 , 1
>:init_net
>:readpattern and.pat
>:set error_limit = 0.001
>:set eta = 0.2
>:set alpha = 0.9
>:fast_cycle 1000
>:print act_iterations
147
>:print act_error
0.00099182129
>:
```

Nun soll das quadratische Fehlermaß des Netzes unter 0.000001 *gesenkt werden. Zur Problemlösung muß jetzt der* cycle*–Befehl verwendet werden. Im Anschluß ist das Netz in der Datei* `and.loesung` *zu speichern.*

```
>:set error_limit = 0.000001
>:cycle 1000
>:print act_iterations
133
>:print act_error
9.8492885e-07
>:savenet and.loesung
>:
```

Kapitel 12

Trainingsprotokolle

Um den Trainingsverlauf für den Benutzer transparent zu machen, bietet ProSiNN Protokolle an. Mit den entsprechenden Befehlen kann der Benutzer das System so einstellen, daß parallel zum Training ausgewählte Daten in eine Datei oder das Ein/Ausgabefenster geschrieben werden. Diese Daten lassen sich auch grafisch darstellen.

Im ersten Abschnitt wird die Anweisung vorgestellt, die Protokollausgaben in eine Datei oder das Ein/Ausgabefenster definiert, sowie das Kommando, das alle Protokolle wieder löscht. Daran schließen sich die Befehle an, die Grafikprotokolle erzeugen und wieder freigeben.

Bem.:
Jedes definierte Protokoll verlangsamt das Lernverfahren! Bei einem Problem mit unbekannter Komplexität ist es sinnvoll, *wenig* Protokolle zu definieren oder den Zyklenabstand der Protokollausgabezeilen *groß* zu wählen (bei Textprotokollen).

12.1 Textprotokolle

In diesem Abschnitt wird der Befehl eingeführt, der in einem beliebigen Zyklenabstand eine Protokollzeile ausgibt. Protokollierbare Elemente sind Zahlen (Integer oder Double), Felder, Funktionen und Zeichenketten. Im Anschluß wird das Kommando präsentiert, das alle Textprotokolle wieder löscht.

12.1.1 Textprotokoll erzeugen: protocol

Um ein Textprotokoll zu erzeugen, muß der Anwender den *protocol*–Befehl benutzen. Die Kommandosyntax lautet:

protocol_command	::=	*"protocol"* **file_handle cycle_distance** *":"* **protocol_parameter** { *","* **protocol_parameter** }.
file_handle	::=	**int_expression.**
cycle_distance	::=	**int_expression.**
protocol_parameter	::=	**int_val** | **double_val** | **array_val** | **funct_val** | **string_val.**

Bem.:

- Der Parameter **file_handle** gibt den Dateideskriptor an, der das Ausgabemedium für die Protokollzeilen repräsentiert. Wird das Handle **stdout** oder 0 verwendet, dann benutzt ProSiNN das Ein/Ausgabefenster als Medium.

- **cycle_distance** beschreibt den Zyklenabstand der Protokollzeilen. Eine Protokollzeile wird dann nach den folgenden Lernzyklen ausgegeben: n $*$ **cycle_distance**, wobei n $\in \{1, 2, 3, \ldots\}$.

- **protocol_parameter** definiert ein Element der Protokollzeile. Die Anzahl dieser Argumente ist beliebig (sofern sie größer als 1 ist). Die Parameter werden bei der Protokollierung in der Reihenfolge ihrer Definition ausgegeben:

  - **int_val** muß eine Konstante oder Variable vom Typ *int* sein.
  - **double_val** ist eine Konstante oder Variable vom Typ *double*.
  - **array_val** muß folgende Konvention einhalten:

    array_val ::= **array_name**“*[*“ **int_val** “*]*“.
    array_name ::= **„Bezeichner einer Feldvariablen“.**

    * **array_name** muß der Bezeichner einer Feldvariablen vom Typ *int* oder *double* sein.
    * **int_val** ist eine Konstante oder Variable vom Typ *int*. Dieser Wert muß bei der Auswertung des *protocol*–Befehls und während des Protokollierens ein gültiges Feldelement repräsentieren.

  - **funct_val** muß folgender Syntax genügen:

    funct_val ::= **funct_name**“*(*“ **funct_arguments** “*)*“.
    funct_name ::= **„Bezeichner einer Funktionsvariablen“.**
    funct_arguments ::= **val** { “,“ **val** }.
    val ::= **int_val** | **double_val.**

    * **funct_name** ist der Bezeichner einer Funktionsvariablen vom Typ *int* oder *double*.
    * **funct_arguments** setzt die Funktionsargumente. Für jedes Funktionsargument muß **val** stehen; dabei darf **val** eine Konstante oder Variable vom Typ *int* oder *double* sein.

  - **string_val** muß eine Konstante oder Variable vom Typ *string* sein.

- ProSiNN führt am Ende einer Protokollzeile keinen Zeilenumbruch durch. Aus diesem Grund sollte der letzte Parameter immer **EOL** sein.

- Wenn ein zu protokollierendes Element während des Trainings nicht mehr ansprechbar ist, bricht ProSiNN den Lernbefehl mit einer Fehlermeldung ab.

- Der Benutzer kann beliebig viele Textprotokolle generieren. Man beachte aber, jedes weitere Protokoll verlangsamt die Durchführung des Trainings!

- Protokollausgaben, die in eine Datei gelenkt werden, können i.allg. schneller verarbeitet werden als solche, die in das Ein/Ausgabefenster geschrieben werden.

- Textprotokolle können immer definiert werden! Es muß also kein Netz im System vorliegen.

Beispiele:

- *Das erste Beispiel definiert ein Textprotokoll, das in das Ein/Ausgabefenster nach jeweils 10 Zyklen eine Zeile ausgibt:*

  ```
  protocol 0 10: act_iterations , act_error , cos(act_error) , EOL
  ```

  In jeder Protokollzeile würde nun parallel zum Training die aktuelle Anzahl der ausgeführten Zyklen, das aktuelle Fehlermaß und dessen Kosinus (nur zur Demonstration eines Funktionsprotokollelementes) ausgegeben werden. Den Abschluß einer Zeile bildet die Stringkonstante, die einen Zeilenvorschub durchführt.

- *Nun soll das binäre* AND*–Problem gelöst werden. Parallel zum Lernverfahren ist die aktuelle Anzahl der Trainingszyklen und das dazugehörige Fehlermaß nach jeweils zehn Zyklen in die Datei* `protocol` *zu schreiben. Die im Beispiel des Abschnitts 10.1.1 eingeführten Trainingsmuster seien in der Datei* `and.pat` *gespeichert:*

  ```
  >:define int i
  >:open protocol i
  >:protocol i 10: act_iterations , act_error , EOL
  >:layer 3: 2 , 2 , 1
  >:init_net
  >:readpattern and.pat
  >:cycle 1000
  >:close i
  >:
  ```

  Nach Ausführung dieser Kommandos steht in der Datei `protocol` *folgendes:*

  ```
  10 0.75234247
  20 0.73331561
  30 0.71784070
  40 0.67732351
  50 0.55100598
  60 0.27711964
  70 0.069905777
  80 0.020826810
  90 0.0083386334
  100 0.0039210705
  110 0.0020143538
  120 0.0010905652
  ```

12.1.2 Alle Textprotokolle freigeben: free_protocol

Um *alle* Textprotokolle wieder freizugeben, muß der Benutzer das *free_protocol*–Kommando verwenden. Seine Syntax ist:

free_protocol ::= *"free_protocol"*.

Bem.:

Es werden alle Textprotolle (aber keine Grafikprotokolle) freigegeben.

12.2 Grafikprotokolle

Dieser Abschnitt beschäftigt sich mit dem Befehl, der ein Grafikprotokoll erzeugt, und dem Kommando, das *alle* Protokolle wieder freigibt. In ein Grafikfenster lassen sich beliebige Zahlen (Integer oder Double), Felder und Funktionen plotten.

12.2.1 Grafikprotokoll erzeugen: grafic_protocol

Zur Erzeugung eines Grafikprotokolls dient die *grafic_protocol*–Anweisung. Die Kommandosyntax lautet:

grafic_protocol_command ::= *"grafic_protocol"* **grafic_handle** *":"*
protocol_parameter { *","* **protocol_parameter** }.
grafic_handle ::= **int_expression.**
protocol_parameter ::= [**dotted_specification**]
(**int_val** | **double_val** | **array_val** | **funct_val**).

Bem.:

- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.
- **protocol_parameter** definiert ein Element, das in das angegebene Grafikfenster ausgegeben werden soll. Die Anzahl der **protocol_parameter** ist beliebig (sofern sie größer als 1 ist):
  - **dotted_specification** beschreibt die optionale Angabe einer Strichlierung (siehe Abschnitt 8.4), mit der dieser Parameter ausgegeben werden soll.
  - **int_val** muß eine Konstante oder Variable vom Typ *int* sein.
  - **double_val** ist eine Konstante oder Variable vom Typ *double*.
  - **array_val** muß folgende Konvention einhalten:

    array_val ::= **array_name** *"["* **int_val** *"]"*.
    array_name ::= **„Bezeichner einer Feldvariablen".**

  * **array_name** muß der Bezeichner einer Feldvariablen vom Typ *int* oder *double* sein.
  * **int_val** ist eine Konstante oder Variable vom Typ *int*. Dieser Wert muß bei der Auswertung des *grafic_protocol*–Befehls und während des Protokollierens ein gültiges Feldelement repräsentieren.

- **funct_val** muß folgender Syntax genügen:

  funct_val ::= **funct_name**"(" **funct_arguments** ")".
  funct_name ::= **„Bezeichner einer Funktionsvariablen"**.
  funct_arguments ::= **val** { "," **val** }.
  val ::= **int_val** | **double_val.**

  * **funct_name** ist der Bezeichner einer Funktionsvariablen vom Typ *int* oder *double*.
  * **funct_arguments** setzt die Funktionsargumente. Für jedes Funktionsargument muß **val** stehen; dabei darf **val** eine Konstante oder Variable vom Typ *int* oder *double* sein.

- Die Anzahl der Lernzyklen wird beim Protokollieren durch die Abszissenachse repräsentiert.
- Durch die Breite des Grafikfensters und des Weltkoordinatensystems (WKS) errechnet ProSiNN die Zyklenanzahl, nach der ein Element ausgegeben werden muß.
- Wird beim Ausgeben der rechte Rand des Weltkoordinatensystems erreicht, dann löscht ProSiNN den Fensterinhalt, zeichnet neue Koordinatenachsen ein (dabei wird der rechte Rand des WKS zum neuen linken Rand des WKS; die Breite des WKS bleibt erhalten; der untere Wert der Ordinate des WKS wird 0.0 und der obere Rand wird durch das letzte aktuelle Fehlermaß bestimmt) und fährt mit dem Lernverfahren und dem Protokollieren fort.
- Wenn ein zu protokollierendes Element während des Trainings nicht mehr ansprechbar ist, bricht ProSiNN den Lernbefehl mit einer Fehlermeldung ab.
- Der Benutzer kann beliebig viele Grafikprotokolle erzeugen. Man beachte aber, jedes weitere Protokoll verlangsamt die Durchführung des Trainings!
- Grafikprotokolle können immer definiert werden, sofern das angegebene Grafikfenster existiert! Es muß also kein Netz im System vorliegen.

Beispiel:
In diesem Beispiel wird das binäre AND*–Problem gelöst. Parallel zum Lernverfahren werden die Lernrate und das Fehlermaß grafisch protokolliert. Die im Beispiel des Abschnitts 10.1.1 eingeführten Trainingsmuster seien in der Datei* `and.pat` *gespeichert. Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt die dazu benötigten Kommandos.*

```
>:define int handle_1
>:define int handle_2
>:define int handle_3
>:open_grafic 300 , 300 to handle_1
```

```
>:open_grafic 300 , 300 to handle_2
>:open_grafic 300 , 300 to handle_3
>:set_world 0.0 , 150.0 , 0.0 , 5.0 to handle_1
>:set_world 0.0 , 150.0 , 0.0 , 1.0 to handle_2
>:set_world 0.0 , 150.0 , 0.0 , 0.1 to handle_3
>:axes 0.0 , 150.0 , 0.0 , 5.0 , 0.0 , 0.0 to handle_1
>:axes 0.0 , 150.0 , 0.0 , 1.0 , 0.0 , 0.0 to handle_2
>:axes 0.0 , 150.0 , 0.0 , 0.1 , 0.0 , 0.0 to handle_3
>:layer 3: 2 , 2 , 1
>:init_net
>:grafic_protocol handle_1: act_error , [ 5 , 5 , 5 , 5 ] act_eta
>:grafic_protocol handle_2: act_error , [ 5 , 5 , 5 , 5 ] act_eta
>:grafic_protocol handle_3: act_error , [ 5 , 5 , 5 , 5 ] act_eta
>:readpattern and.pat
>:cycle 1000
>:
```

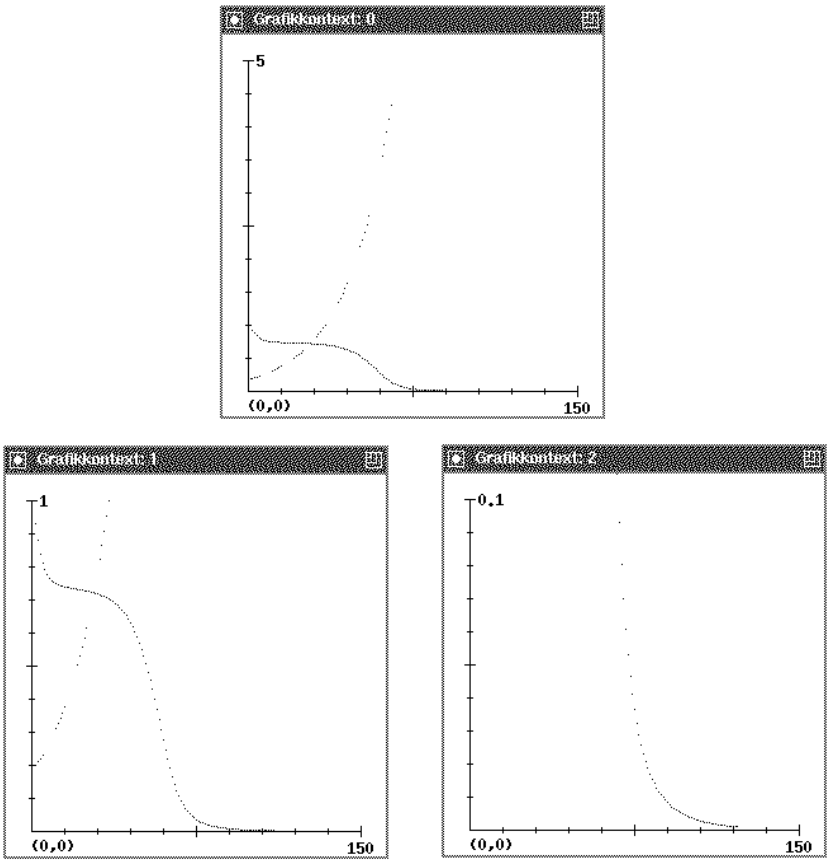

Abbildung 12.1: Mit dem *grafic_protocol*–Befehl erzeugte Grafikprotokolle

Zuerst wurden die Grafikhandles zur Generierung von drei Grafikfenstern definiert. Im Anschluß sind diese Fenster dann geöffnet worden. Die im folgenden ausgeführten Kommandos setzten die x-Achsen der Weltkoordinatensysteme auf den Bereich $[0.0, \ldots, 150.0]$

und die y-Achsen auf die Intervalle $[0.0, \ldots, 5.0]$, $[0.0, \ldots, 1.0]$ *und* $[0.0, \ldots, 0.1]$. *Danach wurden die Koordinatenachsen eingezeichnet. Anschließend wurde das Netz erzeugt und initialisiert. Die nachfolgenden Befehle definierten für jedes Grafikfenster ein Protokoll, das die aktuelle Lernrate (mit einer Strichlierung) und das Fehlermaß enthält. Die durch die cycle–Anweisung entstandenen Grafiken sind in Abbildung 12.1 dargestellt.*

12.2.2 Alle Grafikprotokolle freigeben: free_grafic_protocol

Um *alle* Grafikprotokolle wieder freizugeben, muß der Benutzer das *free_grafic_protocol*–Kommando verwenden. Seine Syntax ist:

free_grafic_protocol ::= *"free_grafic_protocol"*.

Bem.:

Es werden alle Grafikprotolle (aber keine Textprotokolle) freigegeben.

Kapitel 13

Systembefehle

In diesem Kapitel werden die Befehle vorgestellt, die ProSiNN zur Erlangung von Systeminformationen, sowie zur Verwaltung von Variablen und dem Arbeitsverzeichnis anbietet. Andere Anweisungen führen ein Betriebssystemkommando aus oder beenden ProSiNN (bzw. ein Simulationsprogramm).

Der erste Abschnitt beschäftigt sich mit den Befehlen, die Variableninformationen zur Verfügung stellen. Der Anwender kann sich die Bezeichner der Systemvariablen, der neu definierten oder aller Variablen ausgeben lassen. Daran schließen sich die Kommandos an, die Systeminformationen in eine Datei (oder das Ein/Ausgabefenster) schreiben. Der vierte Abschnitt ist der globalen Variablenverwaltung gewidmet. Hier werden Anweisungen eingeführt, die Systemvariablen mit Vorgabewerten setzen, vom Benutzer definierte Variablen wieder freigeben und evtl. zusätzlich das System neu initialisieren. Befehle, die das Arbeitsverzeichnis oder deren Inhalt ausgeben, sowie die Anweisung, die einen Verzeichniswechsel durchführt, schließen sich daran an. Der nachfolgende Abschnitt stellt das Kommando vor, das einen Betriebssystembefehl vom Anwender ausführt. Die Anweisung, die ein Programm oder ProSiNN beendet, wird abschließend präsentiert.

13.1 Variableninformationen

In diesem Abschnitt werden die Befehle vorgestellt, die Informationen über die im System vorliegenden Bezeichner liefern. In lexikographischer Reihenfolge werden die Bezeichner (und Informationen über sie) in eine Datei oder das Ein/Ausgabefenster geschrieben.

Im ersten Unterabschnitt wird das Kommando vorgestellt, das alle vom System vordefinierten Bezeichner behandelt. Daran schließt sich die Anweisung an, die nur Informationen über die vom Benutzer definierten Bezeichner ausgibt. Den Abschluß bildet der Befehl, der alle im System vorliegenden Bezeichner einbezieht.

13.1.1 Systemvariablen ausgeben: sys_names

Um Informationen über alle vordefinierten Bezeichner zu erhalten, muß der Anwender den *sys_names*–Befehl benutzen. Seine Syntax lautet:

sys_names_command ::= *"sys_names"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so werden die Informationen in das Ein/Ausgabefenster geschrieben.
- Alle vordefinierten Bezeichner werden in lexikographischer Reihenfolge ausgegeben. Dabei sind Zahlen lexikographisch kleiner als Großbuchstaben, die wiederum kleiner als Kleinbuchstaben sind!

Beispiel:
In diesem Beispiel sollen Informationen über alle vom System vordefinierten Bezeichner ausgegeben werden. Die folgenden Zeilen zeigen den entsprechenden (gekürzten) Ausschnitt des Ein/Ausgabefensters:

```
>:sys_names
EOL: String-Variable
acos: Double-Funktion
act_alpha: Aktuelles Lernmoment
act_error: Aktuelles Fehlermass
act_eta: Aktuelle Lernrate
act_iterations: Aktuelle Anzahl Lernzyklen
activation: System-Kommando
add_layer: System-Kommando
alias: System-Kommando
alpha: Lernmoment
.............................................
text: System-Kommando
theta: Unit-Praefix
theta_max: Schwellenwert-Maximum (beim Initialisieren)
theta_min: Schwellenwert-Minimum (beim Initialisieren)
time: System-Kommando
user_names: System-Kommando
w: Weight-Praefix
w_max: Gewichtsmaximum (beim Initialisieren)
w_min: Gewichtsminimum (beim Initialisieren)
while: System-Kommando

>:
```

13.1.2 Benutzervariablen ausgeben: user_names

Zur Ausgabe von Informationen über vom Benutzer definierte Bezeichner, muß das Kommando *user_names* verwendet werden. Die Syntax ist:

user_names_command ::= *"user_names"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so werden die Informationen in das Ein/Ausgabefenster geschrieben.
- Alle vom Benutzer definierten Bezeichner werden in lexikographischer Reihenfolge ausgegeben. Dabei sind Zahlen lexikographisch kleiner als Großbuchstaben, die wiederum kleiner als Kleinbuchstaben sind!

Beispiel:
In diesem Beispiel sollen Informationen über alle vom Benutzer definierten Variablen ausgegeben werden. Die folgenden Zeilen zeigen einen Ausschnitt des Ein/Ausgabefensters:

```
>:user_names stdout
>:define int i
>:define array double da[20]
>:user_names
da: Double-Array mit 20 Elementen
i: Integer-Variable

>:
```

Der erste user_names*-Befehl lieferte keine Ausgabe, da der Anwender bisher noch keine Variablen erzeugt hatte. Nach der Definition einer Integer- und einer Double-Feldvariablen, wurde das Kommando wiederholt. Diesmal wurden Informationen über die neu generierten Variablen ausgegeben.*

13.1.3 Alle definierten Bezeichner ausgeben: names

Um Informationen über alle im System vorliegenden Bezeichner zu erhalten, muß der Benutzer den *names*-Befehl verwenden. Die Kommandosyntax lautet:

names_command ::= *"names"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so werden die Informationen in das Ein/Ausgabefenster geschrieben.
- Alle im System vorliegenden Bezeichner werden in lexikographischer Reihenfolge ausgegeben. Dabei sind Zahlen lexikographisch kleiner als Großbuchstaben, die wiederum kleiner als Kleinbuchstaben sind!

13.2 Statusinformationen ausgeben: status

Will der Benutzer einen Systemstatusbericht in eine Datei oder in das Ein/Ausgabefenster ausgeben, dann muß er den *status*–Befehl verwenden. Die Syntax ist:

status_command ::= *“status“* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so werden die Systeminformationen in das Ein/Ausgabefenster geschrieben.
- Bei der Ausführung dieser Anweisung werden die Angaben in der folgenden Reihenfolge ausgegeben:
  - Netzname
  - Aktivierungsfunktion
  - Netzstruktur, sofern ein Netz im System existiert
  - Anzahl der Trainingsmuster, sofern welche im System existieren
  - Anzahl der Testmuster, sofern welche im System existieren
  - Anzahl der bisher ausgeführten Trainingszyklen
  - Anzahl der Textprotokolle
  - Anzahl der Grafikprotokolle
  - Fehlermaß-Modus (**error_mode**)
  - Untere Fehlerschranke (**error_limit**)
  - Aktueller Fehler (**act_error**)
  - Lernrate (**eta**)
  - Aktuelle Lernrate (**act_eta**)
  - Lernmoment (**alpha**)
  - Aktuelles Lernmoment (**act_alpha**)
  - Erfolgslernratenfaktor (**phi**)
  - Mißerfolgslernratenfaktor (**beta**)
  - Mißerfolgsfaktor (**fail_limit**)
  - Initialisierungsminimum für die Schwellenwerte
  - Initialisierungsmaximum für die Schwellenwerte
  - Initialisierungsminimum für die Gewichte
  - Initialisierungsmaximum für die Gewichte
  - Anzahl Nachkommastellen bei der Fließkommaausgabe

Beispiel:
Nach dem Start von ProSiNN sollen Informationen über das System ausgegeben werden. Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt die Durchführung dieser Aufgabenstellung:

```
>:status
*********************
* Netzstatusbericht *
*********************
Netzname:
*********************
activation(x) = 1.0 / ( 1.0 + exp(-x) )
*********************
Es ist kein Netz im Speicher vorhanden
*********************
Anzahl Textprotokolle: 0
Anzahl Grafikprotokolle: 0
*********************
Fehlermass-Modus (error_mode): Quadratische Abweichung
Untere Fehlerschranke (error_limit): 0.0010000000
Aktueller Fehler (act_error): 0.0000000
*********************
Lernrate (eta): 0.20000000
Aktuelle Lernrate (act_eta): 0.20000000
Lernmoment (alpha):0.90000000
Aktuelles Lernmoment (act_alpha): 0.90000000
Erfolgslernratenfaktor (phi): 1.0500000
Misserfolgslernratenfaktor (beta): 0.70000000
Misserfolgsfaktor (fail_limit): 1.0400000
*********************
Initialisierungsminimum fuer Schwellenwerte: -0.50000000
Initialisierungsmaximum fuer Schwellenwerte: 0.50000000
Initialisierungsminimum fuer Gewichte: -0.50000000
Initialisierungsmaximum fuer Gewichte: 0.50000000
*********************
Anzahl Nachkommastellen bei Fliesskommaausgabe: 8

>:
```

13.3 Zeit ausgeben: time

Soll die Systemzeit in eine Datei oder in das Ein/Ausgabefenster ausgegeben werden, so muß der Anwender den *time*–Befehl benutzen. Die Kommandosyntax lautet:

time_command ::= *"time"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Die Anweisung gibt das Datum und die Zeit aus.
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird die Zeit (und das Datum) in das Ein/Ausgabefenster geschrieben.
- Der Befehl führt keinen Zeilenumbruch am Ende der Ausgabe durch!

Beispiel:
Es soll das Datum und die aktuelle Systemzeit ausgegeben werden. Die folgenden Zeilen zeigen den entsprechenden Ausschnitt des Ein/Ausgabefensters:

```
>:time
Mon Oct 11 15:05:55 1993
>:
```

13.4 Globale Variablenverwaltung

In diesem Abschnitt werden drei Befehle vorgestellt, die die Variablen global beeinflußen. Zuerst wird das Kommando behandelt, das *bestimmte* Systemvariablen neu initialisiert. Daran schließt sich die Anweisung an, die den *größten* Teil der vom Benutzer definierten Strukturen wieder freigibt. Der dritte Befehl löscht *alle* vom Anwender generierten Objekte.

13.4.1 Systemvariablen neu initialisieren: init

Um *bestimmte* Systemvariablen neu zu initialisieren, muß das *init*–Kommando benutzt werden. Die Syntax ist:

init_command ::= *“init“*.

Bem.:

Der Befehl setzt die folgenden Variablen auf die angegebenen Werte:

- **precision** = 8
- **floatpattern** = **on**
- **pat_min** = 0.0 und **pat_max** = 1.0
- **theta_min** = −0.5 und **theta_max** = 0.5
- **w_min** = −0.5 und **w_max** = 0.5

- **act_error** = **old_error** = 0.0
- **error_limit** = 0.001
- **error_mode** = **quadratisch**
- **eta** = **act_eta** = 0.2
- **alpha** = **act_alpha** = 0.9
- **fail_limit** = 1.04
- **phi** = 1.05
- **beta** = 0.7
- **sum_iterations** = **act_iterations** = 0

13.4.2 Variablen freigeben: free

Will der Anwender *bestimmte* Variablen und Strukturen wieder freigeben (näheres siehe unter Bemerkungen), so sollte der *free*–Befehl benutzt werden. Dessen Syntax lautet:

free_command ::= *“free“*.

Bem.:

- Alle mit der *define*–Anweisung erzeugten Variablen werden wieder freigegeben. Dazu gehören Integer–, Double–, Feld–, Funktions–, String–, Layer–, Unit– und Weight–Variablen.
- Alle durch *alias* generierten Makros werden aus dem System entfernt.
- Ein Neuronales Netz bleibt im System erhalten!
- Testmuster werden gelöscht.
- Trainingsmuster bleiben erhalten; deren Bezeichner — sofern sie existieren — werden aber entfernt!
- Die Aktivierungsfunktion bleibt unverändert!
- Alle Dateien und Grafikfenster werden geschlossen.
- Alle Text– und Grafikprotokolle werden gelöscht.
- Nach Ausführung des Befehls existieren *keine* vom Benutzer definierten Bezeichner mehr im System (die Ausnahme bilden die beiden Bezeichner der Aktivierungsfunktion und deren Ableitung — sofern der Anwender eine neue Aktivierungsfunktion definiert hatte)!

13.4.3 Alles freigeben: free_all

Das *free_all*–Kommando gibt alle generierten Strukturen wieder frei und versetzt das System in den Zustand, der beim Start von ProSiNN vorlag. Die Syntax ist:

free_all_command ::= "*free_all*".

Bem.:

- Alle mit der *define*–Anweisung erzeugten Variablen werden wieder freigegeben. Dazu gehören Integer–, Double–, Feld–, Funktions–, String–, Layer–, Unit– und Weight–Variablen.
- Alle durch *alias* generierten Makros werden aus dem System entfernt.
- Ein existierendes Neuronales Netz wird eliminiert.
- Testmuster und Trainingsmuster werden gelöscht.
- Es wird wieder die sigmoide Aktivierungsfunktion gesetzt.
- Alle Dateien und Grafikfenster werden geschlossen.
- Alle Text– und Grafikprotokolle werden gelöscht.
- Die Variable **net_name** wird mit dem leeren String initialisiert.
- Der Befehl setzt die folgenden Variablen auf die anschließend angegebenen Werte:
  - **precision** = 8
  - **floatpattern** = **on**
  - **pat_min** = 0.0 und **pat_max** = 1.0
  - **theta_min** = -0.5 und **theta_max** = 0.5
  - **w_min** = -0.5 und **w_max** = 0.5
  - **act_error** = **old_error** = 0.0
  - **error_limit** = 0.001
  - **error_mode** = **quadratisch**
  - **eta** = **act_eta** = 0.2
  - **alpha** = **act_alpha** = 0.9
  - **fail_limit** = 1.04
  - **phi** = 1.05
  - **beta** = 0.7
  - **sum_iterations** = **act_iterations** = 0
- Nach Ausführung des Befehls existieren *keine* vom Benutzer definierten Bezeichner mehr im System!

13.5 Verzeichnisverwaltung

Zur Verwaltung des Arbeitsverzeichnisses stellt ProSiNN drei Kommandos zur Verfügung. Im ersten Unterabschnitt wird die Anweisung eingeführt, die den Inhalt des Arbeitsverzeichnisses ausgibt. Daran schließt sich der Befehl an, der den Arbeitsverzeichnispfad auf ein angegebenes Ausgabemedium schreibt. Den Abschluß bildet das Kommando, das das Arbeitsverzeichnis wechselt.

13.5.1 Arbeitsverzeichnisinhalt ausgeben: dir

Um den Inhalt des aktuellen Arbeitsverzeichnisses in eine Datei oder in das Ein/Ausgabefenster zu schreiben, muß man die *dir*–Anweisung benutzen. Die Kommandosyntax lautet:

dir_command ::= *"dir"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt dieses Angabe, so wird der Arbeitsverzeichnisinhalt in das Ein/Ausgabefenster geschrieben.
- Es werden alle vorhandenen Dateien in der Reihenfolge ihres Entstehens ausgegeben.

Beispiel:
Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt die Ausführung des dir*–Befehls, der den Inhalt des aktuellen Verzeichnisses ausgibt:*

```
>:dir
./
../
dipl.tex
kap1.tex
kap2.tex
kap3.tex
kap4.tex
kap5.tex
kap6.tex
kap7.tex
kap8.tex
kap9.tex
kap10.tex
kap11.tex
kap12.tex
kap13.tex
```

```
kap14.tex
kap15.tex
kap16.tex
kap17.tex
kap18.tex
kap19.tex
kap20.tex
kap21.tex
literatur.tex
appA.tex
appB.tex
appC.tex

>:
```

Es handelt sich hier um das Verzeichnis, das alle Quelldateien des vorliegenden Textes enthält.

13.5.2 Arbeitsverzeichnis ausgeben: pwd

Zur Ausgabe des aktuellen Arbeitsverzeichnispfades in eine Datei oder in das Ein/Ausgabefenster, muß der Benutzer den *pwd*–Befehl verwenden. Die Syntax ist:

pwd_command ::= *"pwd"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt dieses Angabe, so wird der Pfad des Arbeitsverzeichnisses in das Ein/Ausgabefenster geschrieben.

Beispiel:
Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt die Ausführung der pwd*–Anweisung, die im Anschluß nach dem letzten Beispiel eingegeben wurde:*

```
>:pwd
/a/bayes/u4/users/bayes/karsten/text
>:
```

13.5.3 Arbeitsverzeichnis wechseln: cd

Zum Wechseln des aktuellen Arbeitsverzeichnisses muß der Anwender das *cd*–Kommando benutzen. Die Syntax lautet:

cd_command ::= *"cd"* **path_name.**
path_name ::= **string_const** | **string_var** | **any_char_sequence.**
any_char_sequence ::= **char_const** { **char_const** }.

Bem.:

- **string_const** ist eine beliebige Stringkonstante (ohne innere Anführungszeichen), die den Pfad beschreibt.
- **string_var** ist eine Stringvariable, die den Pfad spezifiziert.
- Der Pfad kann auch durch eine beliebige Zeichenfolge **any_char_sequence**, die nicht in Anführungszeichen steht, angegeben werden. Diese Zeichenkette darf dann aber nicht mit einem systeminternen Bezeichner kollidieren. Außerdem dürfen in der Zeichenfolge keine Blanks, Tabulatoren und Zeilenendezeichen vorkommen; ansonsten betrachtet ProSiNN nur den ersten zusammenhängenden Teil als Pfadnamen.
- Bei der Angabe des Pfades darf ".." und "." verwendet werden. Dabei gibt ".." das Vaterverzeichnis und "." das aktuelle Verzeichnis an.
- Es müssen immer die vollständigen Pfadnamen angegeben werden.

Beispiele:
Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt einige Beispiele, bei denen das cd*-Kommando verwendet wurde.*

```
>:pwd
/a/bayes/u4/users/bayes/karsten/text
>:cd ..
>:pwd
/a/bayes/u4/users/bayes/karsten
>:cd ./text
>:pwd
/a/bayes/u4/users/bayes/karsten/text
>:cd /a/bayes/u4/users/bayes
>:pwd
/a/bayes/u4/users/bayes
>:cd karsten/text
>:pwd
/a/bayes/u4/users/bayes/karsten/text
>:cd ../../..
>:pwd
/a/bayes/u4/users
>:cd bayes/karsten/text
>:pwd
/a/bayes/u4/users/bayes/karsten/text
>:
```

13.6 Betriebssystemkommando ausführen: shell

Zum Aufruf des Betriebssystemkommandointerpreters muß der *shell*–Befehl benutzt werden. Seine Syntax ist:

shell_command ::= *"shell"* **command.**
command ::= **string_const** | **string_var** | **any_char_sequence.**
any_char_sequence ::= **char_const** { **char_const** }.

Bem.:

- Die Anweisung ruft den Standard–UNIX–Systemkommandointerpreter mit dem angegebenen Argument auf.
- **string_const** ist eine beliebige Stringkonstante (ohne innere Anführungszeichen), die das Kommando spezifiziert.
- **string_var** ist eine Stringvariable, die das Kommando enthält.
- Das Kommando kann durch eine beliebige Zeichenfolge **any_char_sequence**, die nicht in Anführungszeichen steht, angegeben werden. Diese Zeichenkette darf dann aber nicht mit einem systeminternen Bezeichner kollidieren. Außerdem dürfen in der Zeichenfolge keine Blanks, Tabulatoren und Zeilenendezeichen vorkommen; ansonsten betrachtet ProSiNN nur den ersten zusammenhängenden Teil als vollständigen Systembefehl. Wird diese Angabe verwendet, dann kann ein ausführbares Kommando nur aus einem Parameter bestehen.
- Wenn das angegebene Kommando Ausgaben vornimmt, dann werden die auszugebenden Daten in das Terminalfenster geschrieben, in dem das ProSiNN–System gestartet wurde.
- Mit diesem Befehl ist es z.B. möglich, weitere unabhängige ProSiNN–Programmsysteme zu erzeugen.

Beispiel:
Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt ein Beispiel, bei dem ein weiteres ProSiNN–Programmsystem als neuer Hintergrundprozeß durch das aktuelle System gestartet wurde:

```
>:shell "prosinn -p test &"
>:
```

In der Datei `test` *sei der im Beispiel des Abschnitts 3.3 vorhandene Ausgabebefehl gewesen. ProSiNN erzeugte nun ein weiteres Programmsystem, das die Datei* `test` *als Simulationsprogramm ausführte. Das zweite System hat die in Abbildung 3.4 dargestellte Gestalt.*

Da das neu generierte System mit der Maus plaziert werden muß, ist die interaktive Mitarbeit des Anwenders erforderlich. Wird dies nicht gewünscht, so sollte das Kommando folgendermaßen eingegeben werden:

```
>:shell "prosinn -p test -iconic &"
>:
```

13.7 Ein Programm oder ProSiNN beenden: quit

Zur Beendigung eines ProSiNN–Programms oder des gesamten Programmsystems dient der *quit*–Befehl. Die Kommandosyntax lautet:

quit_command ::= *"quit"*.

Bem.:

- Wird diese Anweisung im Ein/Ausgabefenster eingegeben, so beendet sich ProSiNN.
- Wird der Befehl in einem Simulationsprogramm ausgeführt, so wird dieses beendet. Weitere übergeordnete Programme bleiben unbeeinflußt.

Teil III

Die Menüoberfläche von ProSiNN

Kapitel 14

Das Menü „Netz bearbeiten“

Dieses Kapitel beschäftigt sich mit dem Menü *Netz bearbeiten*. Hier werden sieben Menüpunkte angeboten, die folgendes ermöglichen:

- Ein Netz kann mit beliebiger Topologie erzeugt werden
- Das Netz kann initialisiert werden:
  - Die Intervalle der Schwellenwerte sind veränderbar
  - Die Intervalle der Gewichte sind modifizierbar
  - Der Startwert des Zufallszahlengenerators kann geändert werden
- Ein Netz kann eingelesen werden
- Das Netz kann gespeichert werden als
  - wieder einlesbare Datei
  - C–Programm
- Die Netzstruktur kann geändert werden:
  - Es können Netzebenen hinzugefügt werden
  - Es können Netzschichten entfernt werden
  - Die Neuronenanzahl in einer Ebene kann geändert werden
- Netzwerte können modifiziert werden. Die folgenden Daten sind manipulierbar:
  - Aktivierungswert o_{pj} eines Neurons
  - Schwellenwert θ_j eines Neurons
  - Änderungswert $\Delta\theta_j$ eines Neurons
  - Fehlersignal δ_{pj} eines Neurons
  - Gewichtswert w_{ij}
  - Änderungswert Δw_{ij} eines Gewichtes

- Das Netz kann wieder freigegeben werden

Die Abbildung 14.1 zeigt das Menü in der Form, die der Benutzer nach dem Systemstart erhält:

Abbildung 14.1: Das Menü *Netz bearbeiten*

Die folgenden Abschnitte führen die einzelnen Menüpunkte ein.

14.1 Der Menüpunkt „Netz erzeugen“

Um ein Netz zu erzeugen, muß der Benutzer den Menüpunkt *Netz erzeugen* anwählen. Nach dessen Auswahl erscheint das in Bild 14.2 dargestellte Popup–Fenster:

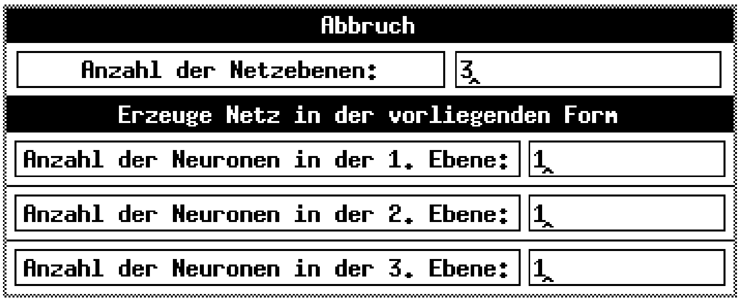

Abbildung 14.2: Das Popup–Fenster des Menüpunktes *Netz erzeugen*

Bem.:

- Durch Anklicken des Buttons Abbruch wird der Menüpunkt ohne Erzeugung eines Neuronalen Netzes beendet.
- Die Anzahl der Netzebenen läßt sich folgendermaßen ändern:
  - Den Mauszeiger in das Eingabefeld für die Netzebenen positionieren.
  - Die Netzebenenanzahl auf den gewünschten Wert korrigieren.
  - Durch Drücken der *Return*–Taste wird für jede Netzschicht ein Eingabefeld erzeugt.

Werden mehr als drei Ebenen angegeben, so erzeugt ProSiNN ein horizontales und ein vertikales Scrollfeld. Mit ihnen kann der Anwender das Eingabefeld für jede Netzschicht erreichen.

Die Anzahl der Netzebenen muß größer als 2 sein!

- Die Anzahl der Neuronen in einer Netzschicht wird folgendermaßen gesetzt:
  - Den Mauszeiger in das entsprechende Neuroneneingabefeld positionieren.
  - Die Neuronenanzahl auf den gewünschten Wert korrigieren.

  Die Anzahl der Neuronen in einer Netzebene muß größer als 1 sein!

- Hat der Benutzer die gewünschte Topologie eingegeben, dann kann das Netz durch Anklicken des Buttons Erzeuge Netz in der vorliegenden Form generiert werden.

14.2 Der Menüpunkt „Netz initialisieren“

Will der Anwender das Netz initialisieren, so muß er den Menüpunkt *Netz initialisieren* anwählen. Anschließend erscheint das Popup–Fenster, das in Abbildung 14.3 dargestellt ist:

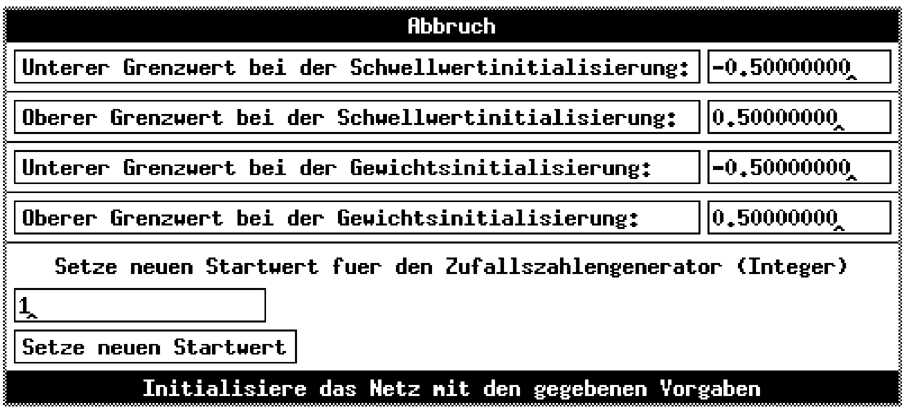

Abbildung 14.3: Das Popup–Fenster des Menüpunktes *Netz initialisieren*

Bem.:

- Durch Anklicken des Buttons Abbruch wird der Menüpunkt ohne Initialisierung des Netzes beendet.
- Die Intervalle beim Initialisieren der Schwellen– und Gewichtswerte lassen sich folgendermaßen ändern:
  - Den Mauszeiger in das Eingabefeld für den jeweiligen Grenzwert der Schwelle oder des Gewichtes positionieren.
  - Die dort angegebene Fließkommazahl auf den gewünschten Wert korrigieren.

Bei der Initialisierung müssen die unteren Grenzwerte kleiner (oder gleich) als die oberen Grenzwerte sein!

Nach der Initialisierung werden die angegebenen Intervalle in den globalen Systemvariablen **theta_min**, **theta_max**, **w_min** und **w_max** gespeichert!

- Der Benutzer kann den Startwert des Zufallszahlengenerators neu setzen. Dazu muß er folgendes ausführen:
  - Den Mauszeiger in das entsprechende Eingabefeld positionieren.
  - Den Startwert auf die gewünschte ganze Zahl korrigieren.
  - Durch Anklicken des Buttons Setze neuen Startwert wird der eingegebene Wert im folgenden verwendet.

  Diese Vorgehensweise ist identisch mit dem Ausführen des *set_rand*–Befehls in Unterabschnitt 9.3.2.

- Durch Anklicken des Buttons Initialisiere das Netz mit den gegebenen Vorgaben wird das Netz initialisiert.

14.3 Der Menüpunkt „Netz laden“

Soll ein Netz geladen werden, dann muß der Menüpunkt *Netz laden* angewählt werden. Nach dessen Auswahl erscheint die Dateiauswahlbox, die in Bild 14.4 dargestellt ist:

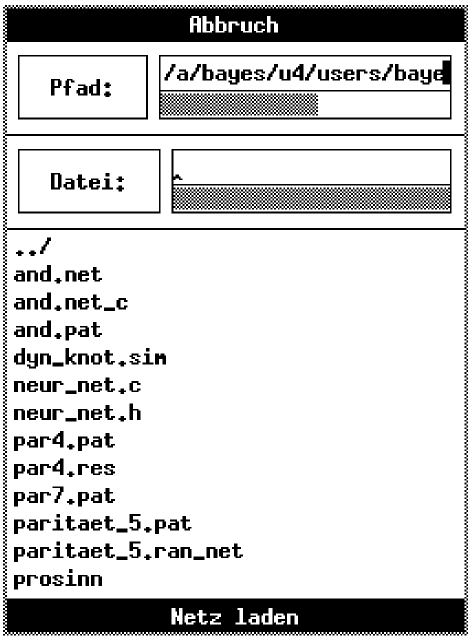

Abbildung 14.4: Die Dateiauswahlbox des Menüpunktes *Netz laden*

Bem.:

- Die Auswahl des Dateinamens (und des Pfades) und das Laden des Netzfiles ist in Unterabschnitt 3.4.2 beschrieben.
- Das Format der Datei, die das zu ladende Netz enthält, muß die in Unterabschnitt 9.4.1 eingeführten Konventionen einhalten.

14.4 Der Menüpunkt „Netz speichern“

Will der Anwender das im System vorliegende Netz in wieder einlesbarer Form (oder als C–Programm) speichern, so muß er den Menüpunkt *Netz speichern* anwählen. Anschließend erscheint das in Abbildung 14.5 dargestellte Popup–Fenster:

Abbildung 14.5: Die Auswahlbox des Menüpunktes *Netz speichern*

Bem.:

- Durch Anklicken des Buttons Abbruch wird der Menüpunkt ohne Speichern des Netzes beendet.
- Wird der Button Netz speichern gewählt, dann erscheint eine Dateiauswahlbox (siehe Unterabschnitt 3.4.2), die das Speichern des Neuronalen Netzes in wieder einlesbarer Form ermöglicht.

  Die Datei wird in dem Format gespeichert, das in Unterabschnitt 9.4.1 eingeführt wurde.
- Durch Anklicken des Buttons Netz als C–Programm speichern erscheint eine Dateiauswahlbox (siehe Unterabschnitt 3.4.2), die das Speichern des Neuronalen Netzes in dem gewünschten Format ermöglicht.

  Das Format der gespeicherten Datei ist mit dem identisch, das der *save_c_net*–Befehl liefert (siehe Unterabschnitt 9.5.3).

14.5 Der Menüpunkt „Netzstruktur ändern“

Mit Hilfe des Menüpunktes *Netzstruktur aendern* kann der Benutzer die Netztopologie modifizieren. Er hat die Möglichkeit, Netzebenen hinzuzufügen (oder zu entfernen) und die Neuronenanzahl in einer Schicht zu ändern. Nachdem der Menüpunkt angewählt wurde, erscheint die in Bild 14.6 dargestellte Auswahlbox als Popup–Fenster:

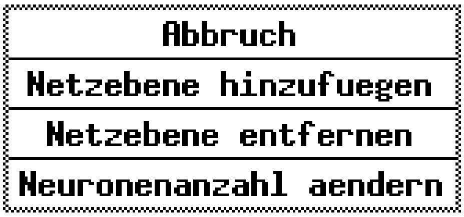

Abbildung 14.6: Die Auswahlbox des Menüpunktes *Netzstruktur aendern*

Bem.:

- Durch Anklicken des Buttons Abbruch wird der Menüpunkt ohne eine Topologieänderung beendet.
- Mit Hilfe des Buttons Netzebene hinzufuegen wird eine neue Netzschicht, die zur neuen Ausgabeebene wird, hinzugefügt.

  Dieser Unterpunkt ist identisch mit dem Ausführen des *add_layer*–Befehls (es gelten dieselben Bemerkungen, die in Unterabschnitt 9.7.1 gemacht worden sind).
- Durch Anklicken des Buttons Netzebene entfernen wird die bisherige Ausgabeschicht entfernt.

  Dieser Unterpunkt ist identisch mit dem Ausführen des *sub_layer*–Befehls (es gelten dieselben Bemerkungen, die in Unterabschnitt 9.7.2 gemacht worden sind).

  Enthält das Netz nur zwei Ebenen, dann erscheint dieser Button nicht in der Auswahlbox (da keine Netzschicht mehr entfernt werden darf).
- Durch Anwählen des Buttons Neuronenanzahl aendern wird ein weiteres Popup–Fenster erzeugt, das die in Abbildung 14.7 dargestellte Form aufweist:

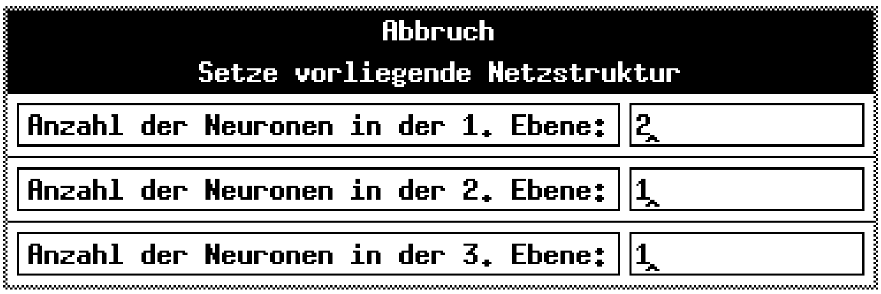

Abbildung 14.7: Die Auswahlbox des Menüunterpunktes *Neuronenanzahl aendern*

  – Durch Anklicken des Buttons Abbruch wird der Menüpunkt ohne Änderung der Neuronenzahl in einer Ebene beendet.
  – Wird der Button Setze vorliegende Netzstruktur ausgewählt, dann wird die aktuell dargestellte Neuronenanzahl jeder Schicht übernommen.
  – Die Anzahl der Neuronen in einer Ebene läßt sich folgendermaßen ändern: