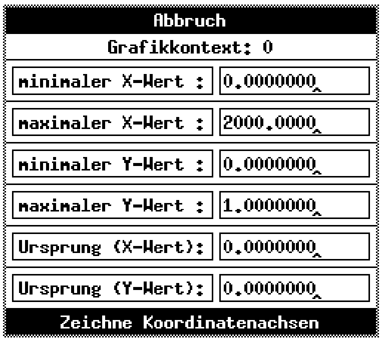


Abbildung 20.4: Das Fenster zum Zeichnen der Koordinatenachsen

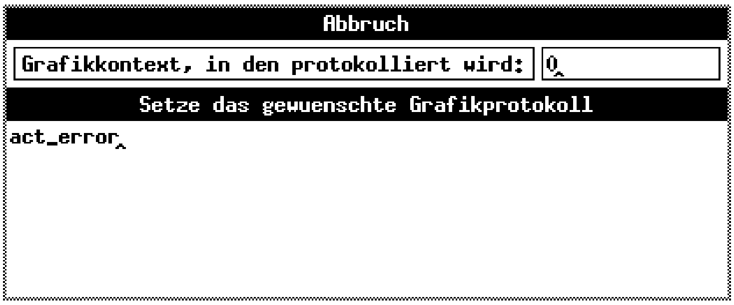


Abbildung 20.5: Das Popup–Fenster des Menüpunktes *Bearbeite Grafikprotokoll*

- Mit dem Button Setze das gewuenschte Grafikprotokoll wird das Protokoll definiert und das Popup–Fenster beendet.

- Jetzt ist das Netz bis maximal 10000 Zyklen zu trainieren.
  - In dem Menü *Netz trainieren* wird der Menüpunkt *Trainiere Netz* ausgewählt.
  - Nachdem die benötigte Eingabe vorgenommen worden ist, hat das erzeugte Popup–Fenster die in Abbildung 20.6 dargestellte Form:

Abbildung 20.6: Das Popup–Fenster des Menüpunktes *Trainiere Netz*

  - Mit dem Button Trainiere Netz wird das Lernverfahren (und die grafische Protokollierung) gestartet und das Popup–Fenster beendet.

- Das Bild 20.7 zeigt die Grafik, in die protokolliert wurde. Man erkennt, daß erst nach über 1200 Zyklen eine Verbesserung des Netzfehlers aufgetreten ist!

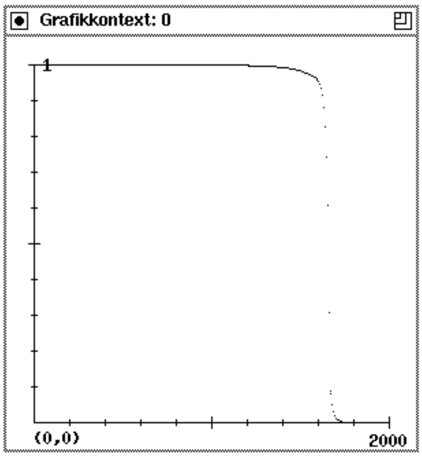


Abbildung 20.7: Das Grafikprotokollfenster des XOR–Netztrainings

- Um die Gesamtanzahl der Trainingszyklen zu erhalten, wird der Menüpunkt *Systemvariablen bearbeiten* im Menü *Systemwerte bearbeiten* angewählt. In Abbildung 20.8 ist das entsprechende Popup–Fenster dargestellt:

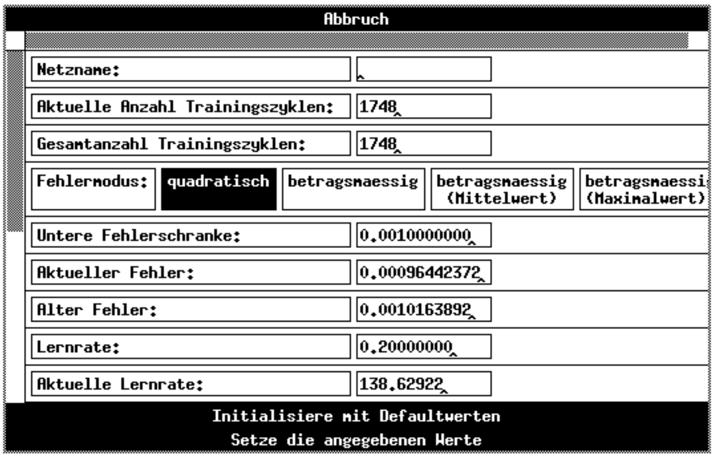


Abbildung 20.8: Die Systemvariablen nach dem XOR–Netztraining

Die Gesamtanzahl der ausgeführten Trainingszyklen beträgt 1748 — Rumelhart, Hinton und Williams (siehe [RUM86]) haben bei ihren Simulationen in nur zwei Fällen eine Konvergenz erhalten; dabei benötigten sie 6587 Zyklen bei einer Lernrate von $\eta = 0.25$!

- In Bild 20.9 ist das Neuronale Netz, das eine Lösung des XOR–Problems repräsentiert, dargestellt:

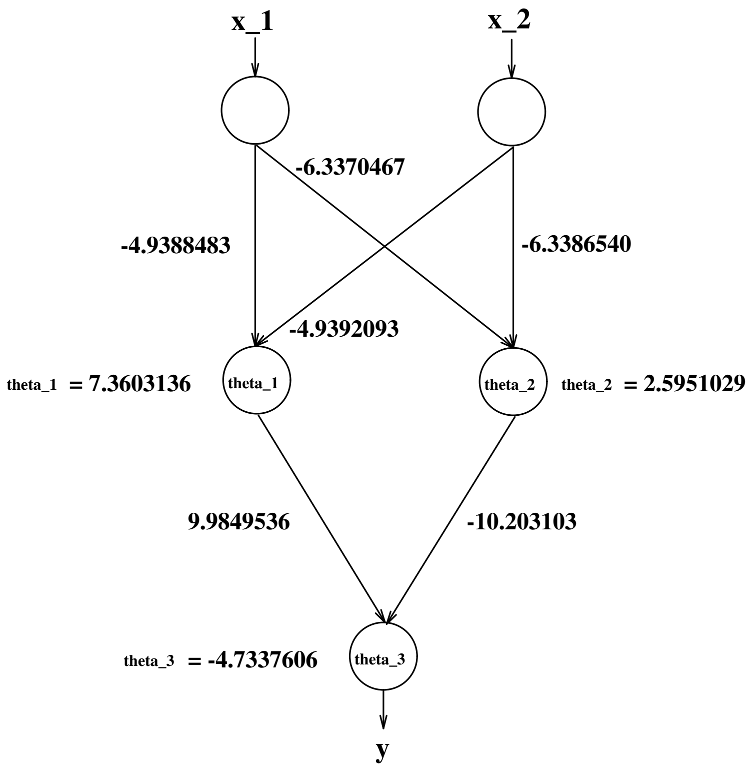


Abbildung 20.9: Die durch ProSiNN erzielte Lösung des XOR–Problems

20.2 Das Encoder–Problem

Ein Encoder hat die Aufgabe Werte zu verschlüsseln. Das hier zu lösende Problem soll zwei binäre Eingabewerte auf vier binäre Ausgabewerte abbilden. In der Tabelle 20.2 wird die Zuordnung der Eingangswerte zu der gewünschten Netzantwort dargestellt.

x_1	x_2	y_1	y_2	y_3	y_4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Tabelle 20.2: Das Encoder–Problem

Aufgabe:

Die in Tabelle 20.2 angegebene Abbildung soll ein Neuronales Netz erlernen. Dabei ist folgendes zu beachten:

- Das Netz soll zwei innere Ebenen enthalten. Die erste innere Schicht beinhalte eine Zelle und die nachfolgende Ebene vier Neuronen.

- Es ist ein grafisches Protokoll zu definieren, das das aktuelle Fehlermaß ausgibt.
- Das Netz soll anschließend in der Datei `encoder.net` gespeichert werden.

Lösung:

In der nachfolgenden Aufzählung wird beschrieben, wie die Aufgabenstellung gelöst wird.

- Die Datei `encoder.pat`, in der die Trainingsmuster gespeichert sind, enthalte die folgenden Daten:

```
set floatpattern = off;
set pat_min = 0.0;
set pat_max = 1.0;
PATTERN:
00 a000;
0a 0a00;
a0 00a0;
aa 000a;
```

- Als erstes soll nun die Netzstruktur erzeugt und initialisiert werden.

```
>:layer 4: 2 , 1 , 4 , 4
>:init_net
>:
```

- Als nächstes werden die Trainingsmuster eingelesen.

```
>:readpattern encoder.pat
>:
```

- Nun ist ein Grafikfenster zum Protokollieren zu eröffnen. Dazu wird eine Integervariable (mit dem Bezeichner `handle`) für den Grafikdeskriptor benötigt. Das zu generierende Weltkoordinatensystem und die Koordinatenachsen sollen in dem Intervall `X`: $[0.0, \ldots, 2500.0]$ und `Y`: $[0.0, \ldots, 2.0]$ liegen.

```
>:define int handle
>:open_grafic 300 , 300 to handle
>:set_world 0.0 , 2500.0 , 0.0 , 2.0 to handle
>:axes 0.0 , 2500.0 , 0.0 , 2.0 , 0.0 , 0.0 to handle
>:
```

- Jetzt wird ein Protokoll, das den aktuellen Netzfehler in die Grafik ausgibt, definiert.

```
>:grafic_protocol handle: act_error
>:
```

- Um das Netz bis maximal 5000 Zyklen zu trainieren, wird das nachfolgende Kommando eingegeben:

```
>:cycle 5000
>:
```

- In Bild 20.10 ist das Grafikfenster, in das protokolliert wurde, dargestellt:

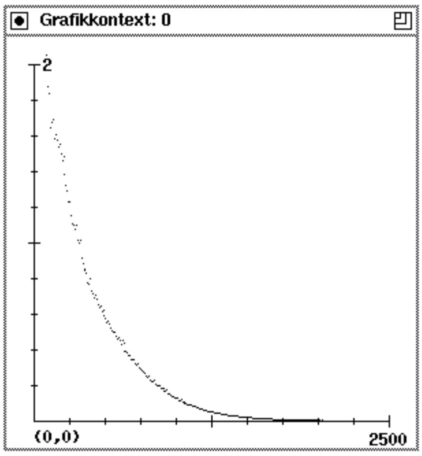


Abbildung 20.10: Das Grafikprotokollfenster des Encoder–Netztrainings

- Um die Gesamtanzahl der Trainingszyklen zu erhalten, wird im Menü *Systemwerte bearbeiten* der Menüpunkt *Systemvariablen bearbeiten* angewählt. In Abbildung 20.11 ist das entsprechende Popup–Fenster dargestellt:

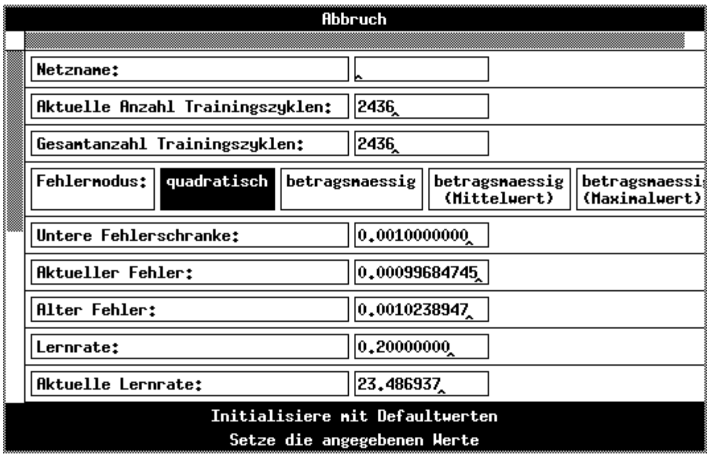


Abbildung 20.11: Die Systemvariablen nach dem Encoder–Netztraining

Die Gesamtanzahl der ausgeführten Trainingszyklen beträgt 2436 — bei einer Lernrate von $\eta = 0.05$ haben Rumelhart, Hinton und Williams (siehe [RUM86]) eine Konvergenz nach 5226 Zyklen erhalten!

- Nun soll das Netz in der Datei `encoder.net` gespeichert werden:

```
>:savenet encoder.net
>:
```

Anschließend enthält die Datei die folgenden Daten:

```
netname "";
layer 4 : 2 , 1 , 4 , 4;
NET:

/* Unit 1,1: */
5.2005285;  /* Weight to Unit 2,1 */

/* Unit 1,2: */
-6.7609663; /* Weight to Unit 2,1 */

/* Unit 2,1: */
0.64561106; /* Bias */
-6.8130099; /* Weight to Unit 3,1 */
12.326392;  /* Weight to Unit 3,2 */
-16.326651; /* Weight to Unit 3,3 */
12.618384;  /* Weight to Unit 3,4 */

/* Unit 3,1: */
3.7663920;  /* Bias */
-2.9531461; /* Weight to Unit 4,1 */
-1.1590170; /* Weight to Unit 4,2 */
-5.1157622; /* Weight to Unit 4,3 */
5.7969447;  /* Weight to Unit 4,4 */

/* Unit 3,2: */
-5.4510831; /* Bias */
9.9977007;  /* Weight to Unit 4,1 */
-6.0436715; /* Weight to Unit 4,2 */
0.76587700; /* Weight to Unit 4,3 */
-7.4650135; /* Weight to Unit 4,4 */

/* Unit 3,3: */
2.1187754;  /* Bias */
-8.6564410; /* Weight to Unit 4,1 */
10.810371;  /* Weight to Unit 4,2 */
-6.3612540; /* Weight to Unit 4,3 */
-12.982373; /* Weight to Unit 4,4 */

/* Unit 3,4: */
-10.410116; /* Bias */
-13.196621; /* Weight to Unit 4,1 */
-5.0450912; /* Weight to Unit 4,2 */
10.098048;  /* Weight to Unit 4,3 */
-9.8752731; /* Weight to Unit 4,4 */
```

```
/* Unit 4,1: */
-2.6289240; /* Bias */

/* Unit 4,2: */
-3.7528667; /* Bias */

/* Unit 4,3: */
-4.7580779; /* Bias */

/* Unit 4,4: */
1.2668911;  /* Bias */
```

20.3 Das T–C Problem

Das T–C Problem stellt eine geometrische Aufgabe dar. Ein Neuronales Netz soll nach dem Training in der Lage sein, die Buchstaben `T` und `C` voneinander unterscheiden zu können. Dabei bestehen die Eingabemuster aus den vier möglichen Rotationen der beiden Buchstaben. In Bild 20.12 sind alle Eingabemuster dargestellt:

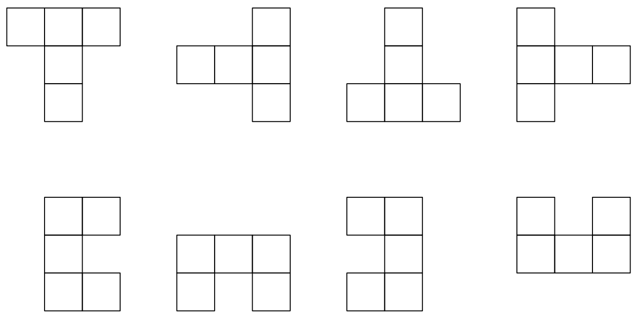


Abbildung 20.12: Die Muster des T–C Problems

Jeder Buchstabe besteht aus 5 Quadraten, die in einer 3x3 Quadrate großen Region angeordnet sind. Dabei unterscheidet sich ein `T`–Muster in der oberen Reihe nur in *einem* Quadrat zu dem darunterliegenden `C`–Muster!

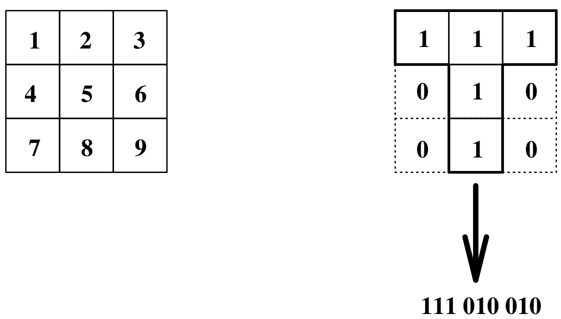


Abbildung 20.13: Die Definition der Netzeingabemuster

Zur Definition der Netzeingabewerte wird jedes Quadrat in der Region durchnummeriert (siehe das linke Teilbild in Abbildung 20.13). Ein Quadrat repräsentiert dabei ein Eingabeneuron des Netzes (dabei geben die Ziffern die Nummer der Eingabezelle an). Für jedes gesetzte Quadrat wird als Eingabewert 1 verwendet; ansonsten wird 0 benutzt. Die Definition des Netzeingabewertes für das erste `T`–Muster ist im rechten Teil des Bildes 20.13 dargestellt.

Die Antwort des Netzes bei der Vorlage des Buchstabens `T` soll 1 sein. Wenn ein `C` eingegeben wurde, hat das Netz mit einer 0 zu reagieren. In Tabelle 20.3 ist die Zuordnung aller Eingabewerte zu dem entsprechenden Netzausgabewert abgebildet:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	y
1	1	1	0	1	0	0	1	0	1
0	0	1	1	1	1	0	0	1	1
0	1	0	0	1	0	1	1	1	1
1	0	0	1	1	1	1	0	0	1
0	1	1	0	1	0	0	1	1	0
0	0	0	1	1	1	1	0	1	0
1	1	0	0	1	0	1	1	0	0
1	0	1	1	1	1	0	0	0	0

Tabelle 20.3: Das T–C Problem

Aufgabe:

Die in Tabelle 20.3 angegebene Abbildung soll ein Neuronales Netz erlernen. Dabei ist folgendes zu beachten:

- Es soll ein Netz mit drei Ebenen verwendet werden. Die innere Schicht enthält drei Neuronen.
- Es sind grafische Protokolle, die das aktuelle Fehlermaß ausgeben, zu definieren.
- Ein Textprotokoll, das nach jeweils 10 Trainingszyklen die aktuelle Anzahl der ausgeführten Zyklen und das aktuelle Fehlermaß in das Ein/Ausgabefenster schreibt, soll generiert werden.
- Nachdem das Netz die Aufgabe erlernt hat, ist der Fehlermaßmodus **max_betrag** zu setzen und das Training erneut durchzuführen.
- Das entstandene Netz soll in der Datei `tc.net` gespeichert werden.

Lösung:

In der nachfolgenden Aufzählung wird beschrieben, wie die Aufgabenstellung gelöst wird.

- Die Datei `tc.pat`, in der die Trainingsmuster gespeichert sind, enthalte die folgenden Daten:

```
set floatpattern = off;
set pat_min = 0.0;
set pat_max = 1.0;
PATTERN:
aaa0a00a0 a;
00aaaa00a a;
0a00a0aaa a;
a00aaaa00 a;
0aa0a00aa 0;
000aaaa0a 0;
aa00a0aa0 0;
a0aaaa000 0;
```

- Als erstes ist die Netzstruktur zu erzeugen und zu initialisieren.

```
>:layer 3: 9 , 3 , 1
>:init_net
>:
```

- Jetzt werden die Trainingsmuster eingelesen.

```
>:readpattern tc.pat
>:
```

- Nun sollen zwei Grafikfenster zum Protokollieren eröffnet werden. Dazu werden die Integervariablen `handle_1` und `handle_2` für die Grafikdeskriptoren definiert. Die zu generierenden Weltkoordinatensysteme und die Koordinatenachsen sollen in den Intervallen X_1: $[0.0, \ldots, 500.0]$ und Y_1 $[0.0, \ldots, 2.5]$, sowie in X_2: $[0.0, \ldots, 500.0]$ und Y_1: $[0.0, \ldots, 0.1]$ liegen.

```
>:define int handle_1
>:open_grafic 300 , 300 to handle_1
>:set_world 0.0 , 500.0 , 0.0 , 2.5 to handle_1
>:axes 0.0 , 500.0 , 0.0 , 2.5 , 0.0 , 0.0 to handle_1
>:define int handle_2
>:open_grafic 300 , 300 to handle_2
>:set_world 0.0 , 500.0 , 0.0 , 0.1 to handle_2
>:axes 0.0 , 500.0 , 0.0 , 0.1 , 0.0 , 0.0 to handle_2
>:
```

- Jetzt werden zwei Grafikprotokolle, die den aktuellen Netzfehler in die Grafik ausgeben, generiert.

```
>:grafic_protocol handle_1: act_error
>:grafic_protocol handle_2: act_error
>:
```

- Das Textprotokoll, das nach jeweils 10 Zyklen die aktuelle Anzahl der ausgeführten Trainingsdurchläufe und das aktuelle Fehlermaß in das Ein/Ausgabefenster schreibt, wird wie folgt definiert:

```
>:protocol stdout 10: sum_iterations , "Zyklen --> Fehler: " ,
  act_error , EOL
>:
```

- Nun wird das Netz bis maximal 500 Zyklen trainiert.

```
>:cycle 500
```

Parallel zum Training wurden die folgenden Zeilen in das Ein/Ausgabefenster geschrieben:

```
10 Zyklen --> Fehler: 2.0010364
20 Zyklen --> Fehler: 2.0006925
30 Zyklen --> Fehler: 2.0003134
40 Zyklen --> Fehler: 1.9998985
50 Zyklen --> Fehler: 1.9993692
60 Zyklen --> Fehler: 1.9979542
70 Zyklen --> Fehler: 1.9939462
80 Zyklen --> Fehler: 2.1433921
90 Zyklen --> Fehler: 2.0219008
100 Zyklen --> Fehler: 2.0126743
110 Zyklen --> Fehler: 1.9240008
120 Zyklen --> Fehler: 1.5694792
130 Zyklen --> Fehler: 1.3523653
140 Zyklen --> Fehler: 1.2324850
150 Zyklen --> Fehler: 1.0463722
160 Zyklen --> Fehler: 0.85263511
170 Zyklen --> Fehler: 0.61502633
180 Zyklen --> Fehler: 0.44287610
190 Zyklen --> Fehler: 0.23248404
200 Zyklen --> Fehler: 0.13708210
210 Zyklen --> Fehler: 0.078661012
220 Zyklen --> Fehler: 0.045020772
230 Zyklen --> Fehler: 0.025944335
240 Zyklen --> Fehler: 0.015093520
250 Zyklen --> Fehler: 0.0088632414
260 Zyklen --> Fehler: 0.0052473164
270 Zyklen --> Fehler: 0.0031274885
280 Zyklen --> Fehler: 0.0018739525
290 Zyklen --> Fehler: 0.0011274229

>:
```

- Nun wird der Fehlermaßmodus auf **max_betrag** gesetzt und das Netz erneut trainiert.

```
>:set error_mode = max_betrag
>:cycle 500
```

In das Ein/Ausgabefenster sind die folgenden Protokollzeilen geschrieben worden:

```
300 Zyklen --> Fehler: 0.013094046
310 Zyklen --> Fehler: 0.010650356
320 Zyklen --> Fehler: 0.0085370867
330 Zyklen --> Fehler: 0.0067750786
340 Zyklen --> Fehler: 0.0053413174
350 Zyklen --> Fehler: 0.0041933257
360 Zyklen --> Fehler: 0.0032841494
370 Zyklen --> Fehler: 0.0025693930
380 Zyklen --> Fehler: 0.0020099293
390 Zyklen --> Fehler: 0.0015728337
400 Zyklen --> Fehler: 0.0012314047

>:
```

- In Bild 20.14 sind die beiden Grafikfenster, in die protokolliert wurde, abgebildet:

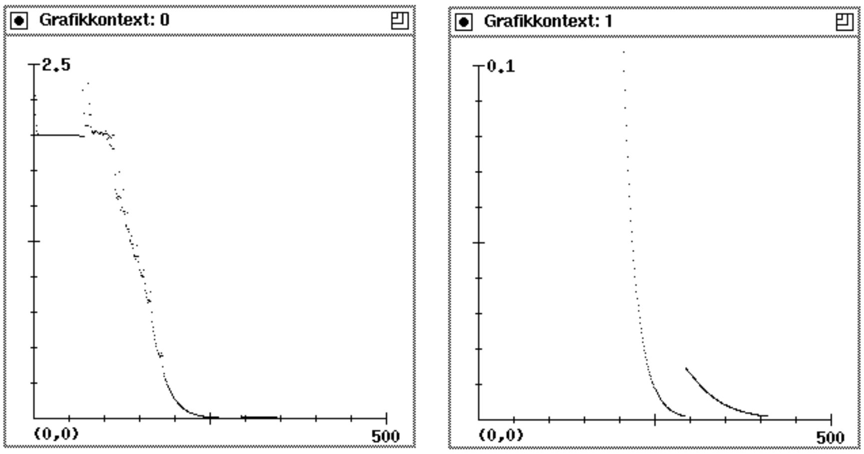


Abbildung 20.14: Die Grafikprotokolle des T–C Netztrainings

Die beiden in jeder Abbildung dargestellten Teilkurven sind durch den Wechsel des Fehlermaßmodus und dem erneuten Training entstanden!

- Um die Gesamtanzahl der Trainingszyklen zu erhalten, wird im Menü *Systemwerte bearbeiten* der Menüpunkt *Systemvariablen bearbeiten* angewählt. In Bild 20.15 ist das erzeugte Popup–Fenster dargestellt.

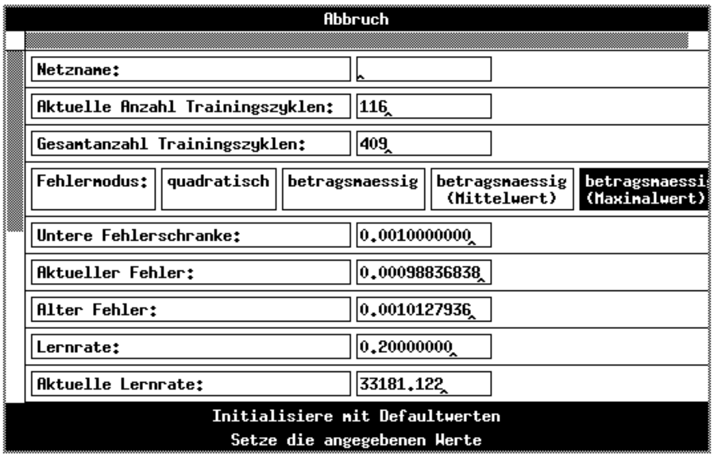


Abbildung 20.15: Die Systemvariablen nach dem T–C Netztraining

Die Gesamtanzahl der Trainingszyklen beträgt 409. Dabei wurden 293 Zyklen mit dem Fehlermaßmodus **quadratisch** und 116 mit dem Modus **max_betrag** ausgeführt. Rumelhart, Hinton und Williams (siehe [RUM86]) haben bei ihren Simulationen eine größere und komplexere Netztopologie verwendet; das dort verwendete Lernverfahren benötigte aber viel mehr Durchläufe um eine Konvergenz zu erzielen!

- Nun wird das Netz in der Datei `tc.net` gespeichert.

```
>:savenet tc.net
>:
```

Anschließend enthält die Datei die folgenden Daten:

```
netname "";
layer 3 : 9 , 3 , 1;
NET:

/* Unit 1,1: */
1.4543391;  /* Weight to Unit 2,1 */
-4.3564780; /* Weight to Unit 2,2 */
-5.0701905; /* Weight to Unit 2,3 */

/* Unit 1,2: */
2.5824050;  /* Weight to Unit 2,1 */
2.4218399;  /* Weight to Unit 2,2 */
-2.2999868; /* Weight to Unit 2,3 */

/* Unit 1,3: */
-1.2986461; /* Weight to Unit 2,1 */
-5.0181979; /* Weight to Unit 2,2 */
-4.1030332; /* Weight to Unit 2,3 */
```

```
/* Unit 1,4: */
-2.8123694; /* Weight to Unit 2,1 */
-2.4354286; /* Weight to Unit 2,2 */
1.6973157;  /* Weight to Unit 2,3 */

/* Unit 1,5: */
-0.11199280; /* Weight to Unit 2,1 */
-0.17704473; /* Weight to Unit 2,2 */
0.31938271;  /* Weight to Unit 2,3 */

/* Unit 1,6: */
-2.4016917; /* Weight to Unit 2,1 */
-1.7536455; /* Weight to Unit 2,2 */
1.9541417;  /* Weight to Unit 2,3 */

/* Unit 1,7: */
1.8787975;  /* Weight to Unit 2,1 */
4.9797500;  /* Weight to Unit 2,2 */
4.2567036;  /* Weight to Unit 2,3 */

/* Unit 1,8: */
2.1998034;  /* Weight to Unit 2,1 */
2.0996887;  /* Weight to Unit 2,2 */
-2.1570383; /* Weight to Unit 2,3 */

/* Unit 1,9: */
-1.4827866; /* Weight to Unit 2,1 */
4.4043084;  /* Weight to Unit 2,2 */
4.7345475;  /* Weight to Unit 2,3 */

/* Unit 2,1:  */
0.011935608; /* Bias */
14.420406;   /* Weight to Unit 3,1 */

/* Unit 2,2:  */
0.061642937; /* Bias */
-14.614194;  /* Weight to Unit 3,1 */

/* Unit 2,3: */
0.12134769; /* Bias */
14.476303;  /* Weight to Unit 3,1 */

/* Unit 3,1: */
-7.0902960; /* Bias */
```

Kapitel 21

Training mit dynamischer Neuronenerzeugung

Dieses Kapitel ist der praktischen Erprobung des in Abschnitt 2.4 eingeführten Lernverfahrens mit dynamischer Neuronenerzeugung gewidmet.

Die Topologie eines Neuronalen Netzes ist von dem zu lösenden Problem abhängig. Da die Anzahl der Ein– und Ausgabeneuronen für eine vorgegebene Aufgabenstellung feststeht, ist *nur* die Quantität der benötigten Zellen in den versteckten Schichten unbekannt. In Abschnitt 2.4 wurde ein Algorithmus vorgestellt, der immer dann ein weiteres Neuron in der mittleren Ebene eines dreischichtigen Netzes hinzufügt, wenn sich beim Training ein Fehlerplateau bildet.

Im ersten Abschnitt wird die zu untersuchende Aufgabenstellung präsentiert: Das Paritäts–Problem. Anschließend wird die Benutzerschnittstelle des Programms, das das in Abschnitt 2.4 eingeführte dynamische Lernverfahren verwendet, beschrieben und das Paritäts–Problem als Beispiel vorgeführt. Im letzen Abschnitt wird das verwendete Simulationsprogramm vorgestellt.

21.1 Die zu untersuchende Aufgabenstellung: Das Paritäts–Problem

Beim Paritäts–Problem muß das Netz bei Vorlage eines binären Eingabevektors als Ausgabe eine 1 liefern, wenn die Anzahl der Einsen in den Eingangsdaten ungerade ist; ansonsten muß mit einer 0 geantwortet werden. Aus den Untersuchungen von Rumelhart, Hinton und Williams (siehe [RUM86]) folgt, daß für Muster mit einer Bitlänge N mindestens N innere Neuronen in einem dreischichtigen Netz benötigt werden.

Hier soll das Problem für Eingabemuster der Bitlänge 4 untersucht werden. In Tabelle 21.1 ist die Zuordnung der Eingabewerte zu den Netzausgabewerten angegeben.

Das Neuronale Netz soll solange trainiert werden, bis das quadratische Fehlermaß 0.001 unterschreitet.

x_1	x_2	x_3	x_4	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Tabelle 21.1: Das Paritäts–Problem bei Eingabevektoren der Bitlänge 4

21.2 Die Programmschnittstelle zum Benutzer und die Lösung der Aufgabenstellung

In diesem Abschnitt wird die Schnittstelle des ProSiNN–Programms `abwdnc.prg`[1] zum Benutzer beschrieben. Im Anschluß wird die Anwendbarkeit anhand des Paritätsproblems gezeigt.

Benutzerschnittstelle:

Die folgende Aufzählung beschreibt die Eingaben, die das Programm vom Anwender erwartet, und die Ausgaben, die durchgeführt werden:

- **Ausgabe:**

```
Programm, das dynamisch Neuronen hinzufuegt, wenn sich das Fehlermass
nicht verbessert. Es wird ein Netz mit 3 Ebenen verwendet!
```

- **Eingabe:**

```
Geben Sie die Anzahl der Eingabeneuronen ein
?:
```

Die Anzahl der Eingabezellen des Neuronalen Netzes ist hier einzugeben.

[1] `abwdkc` steht für accelerated backpropagation with dynamic neuron creation

- **Eingabe:**

  ```
  Geben Sie die Startanzahl der Neuronen in der versteckten Schicht ein
  ?:
  ```

  Hier wird die Anzahl der Zellen in der mittleren Ebene verlangt, mit der das Netz erzeugt wird.

- **Eingabe:**

  ```
  Geben Sie die Anzahl der Ausgabeneuronen ein
  ?:
  ```

  Hier muß die Anzahl der Ausgabezellen des Neuronalen Netzes eingeben werden.

- **Eingabe:**

  ```
  Geben Sie die Anzahl der Neuronen ein, die im Misserfolgsfall
  hinzugefuegt werden
  ?:
  ```

  Dieser Wert gibt die Anzahl der Zellen an, die in der mittleren Schicht hinzugefügt werden, wenn Gleichung 21.1 (bzw. 2.21) erfüllt ist.

  Wird eine Zahl größer als 1 eingegeben, dann unterscheidet sich die Ausführung des Programms von dem Verfahren, das in Abschnitt 2.4 eingeführt wurde.

- **Eingabe:**

  ```
  Geben Sie die Fensterbreite w ein (z.B.: 1000)
  ?:
  ```

  Hier ist die Fenstergröße w, die in Gleichung 21.1 (bzw. 2.21) beschrieben wird, einzugegeben.

- **Eingabe:**

  ```
  Geben Sie das relative Fehlerverhaeltnis ein (z.B.: 0.05)
  ?:
  ```

  Hier wird die Schwelle Δ_T verlangt, die in der folgenden Gleichung enthalten ist (dabei beschreibt $E(t)$ das Fehlermaß zur Zeit t):

  $$\frac{E(t-w)-E(t)}{E(t_0)} < \Delta_T, \text{ wobei } \Delta_T > 0 \text{ und } t-w \geq t_0 \qquad (21.1)$$

- **Eingabe:**

  ```
  Geben Sie die untere Fehlermassgrenze ein (z.B.: 0.001)
  ?:
  ```

  Die untere Fehlermaßgrenze, bei der das Lernverfahren terminieren soll, wird hier angefordert.

- **Eingabe:**

  ```
  Geben Sie die Trainings-Musterdatei mit Pfad ein
  ?:
  ```

  Hier muß der Pfad und Name der Datei eingegeben werden, die die Trainingsmuster enthält.

- **Eingabe:**

  ```
  Wollen Sie ein Textprotokoll erzeugen lassen (j/n)?
  ?:
  ```

  Bei der Eingabe 'n' wird kein Textprotokoll erzeugt. Ist dagegen 'j' angegeben worden, so sind weitere Daten einzugeben:

  – `Geben Sie die Anzahl der Zyklen ein, nach denen protokolliert werden soll`
    `?:`
    Hier wird die Anzahl der Trainingszyklen verlangt, nach denen jeweils eine Protokollzeile ausgegeben wird.

  – `Wollen Sie das Protokoll in eine Datei schreiben lassen (j/n)?`
    `?:`
    Bei der Eingabe 'n' wird jede Protokollzeile in das Ein/Ausgabefenster ausgegeben. Bei 'j' muß der Anwender die Datei näher spezifizieren:

    * `Geben Sie die Protokolldatei mit Pfad ein`
      `?:`
      Hier wird der Pfad und Name der Datei angegeben, in die protokolliert werden soll.

  Wenn ein Textprotokoll generiert wurde, dann wird parallel zum Training die Anzahl der bisher ausgeführten Traingszyklen und das dazugehörige Fehlermaß ausgegeben.

- **Eingabe:**

  ```
  Wollen Sie ein Grafikprotokoll erzeugen lassen (j/n)?
  ?:
  ```

Bei der Eingabe 'n' wird kein Grafikprotokoll definiert. Ist dagegen 'j' angegeben worden, so wird anschließend ein Grafikfenster erzeugt; der Benutzer muß nun mit dem Mauszeiger das neu generierte Fenster plazieren.

Das Weltkoordinatensystem (WKS) des Grafikfensters ist wie folgt definiert:
Die X–Werte laufen von 0 bis zu der Fensterbreite w (dieses Intervall verschiebt sich um w, wenn der aktuelle Bereich durchlaufen worden ist); die Y–Werte befinden sich in dem Intervall von 0 bis zu dem Fehlermaß, das vorlag, als das WKS das letzte Mal verschoben wurde.

Bem.:

- Die folgenden Ausgaben werden vom Programm vorgenommen:
  - Immer wenn Neuronen hinzugefügt werden, wird im Ein/Ausgabefenster und in der Protokolldatei — sofern eine erzeugt wurde — die aktuelle Anzahl der Neuronen in der inneren Schicht ausgegeben.
  - Wurde ein Textprotokoll definiert, so wird die Anzahl der bisher ausgeführten Trainingszyklen und das dazugehörige Fehlermaß in die Protokolldatei — sofern eine definiert wurde — und in das Ein/Ausgabefenster geschrieben.
  - Parallel zum Training wird das aktuelle Fehlermaß in die erzeugte Grafik eingezeichnet — sofern ein Grafikprotokoll generiert wurde.
  - Wenn das Verfahren eine Lösung gefunden hat, wird die benötigte Anzahl der Neuronen in der mittleren Ebene und die Anzahl der Lernzyklen in die Protokolldatei — sofern eine erzeugt wurde — und in das Ein/Ausgabefenster ausgegeben.
- Alle nicht beschriebenen Systemvariablen behalten die Werte bei, die sie vor dem Programmstart enthielten.
- Nach der Terminierung des Programms, sind alle erzeugten Variablen und das Neuronale Netz noch im System enthalten. Nun kann z.B. das Netz mit einem anderen Fehlermaß oder einer kleineren Fehlermaßgrenze weiter trainiert werden.

Beispiel:
Jetzt soll das Paritätsproblem für Eingabemuster der Bitlänge 4 gelöst werden. Die folgenden Konventionen sind einzuhalten:

- Die Startanzahl der Neuronen in der mittleren Ebene ist mit 1 anzugeben.
- Im Fall des Mißerfolgs soll jeweils ein Neuron hinzugefügt werden.
- Für die Fensterbreite w ist 600 zu verwenden.
- Der Schwellenwert Δ_T soll 0.05 betragen.
- Für die untere Fehlermaßgrenze ist 0.001 einzugeben.

- In der Musterdatei `par4.pat` befinden sich die nachfolgenden Daten:

```
set floatpattern = off;
set pat_min = 0.0;
set pat_max = 1.0;
PATTERN:
0000 0;
000a a;
00a0 a;
00aa 0;
0a00 a;
0a0a 0;
0aa0 0;
0aaa a;
a000 a;
a00a 0;
a0a0 0;
a0aa a;
aa00 0;
aa0a a;
aaa0 a;
aaaa 0;
```

- Es ist ein Textprotokoll in das Ein/Ausgabefenster zu definieren. Eine Protokollzeile soll nach jeweils 100 Zyklen ausgegeben werden.
- Ein grafisches Protokoll soll nicht erzeugt werden (in Bild 21.1 ist der gesamte Trainingsverlauf abgebildet; bei einem hier generierten Grafikprotokoll würde nur der aktuelle Bereich über die Fensterbreite w dargestellt werden).
- Anschließend ist das Netz in der Datei `par4.net` zu speichern.

Die nachfolgenden Zeilen beschreiben die Eingaben, die der Benutzer vorgenommen hat, und die von ProSiNN ausgegebenen Daten:

```
>:load abwdnc.prg

Programm, das dynamisch Neuronen hinzufuegt, wenn sich das Fehlermass
nicht verbessert. Es wird ein Netz mit 3 Ebenen verwendet!

Geben Sie die Anzahl der Eingabeneuronen ein
?:4
Geben Sie die Startanzahl der Neuronen in der versteckten Schicht ein
?:1
Geben Sie die Anzahl der Ausgabeneuronen ein
?:1
Geben Sie die Anzahl der Neuronen ein, die im Misserfolgsfall
hinzugefuegt werden
```

```
?:1

Geben Sie die Fensterbreite w ein (z.B.: 1000)
?:600
Geben Sie das relative Fehlerverhaeltnis ein (z.B.: 0.05)
?:0.05
Geben Sie die untere Fehlermassgrenze ein (z.B.: 0.001)
?:0.001
Geben Sie die Trainings-Musterdatei mit Pfad ein
?:par4.pat
Wollen Sie ein Textprotokoll erzeugen lassen (j/n)?
?:j
Geben Sie die Anzahl Zyklen ein, nach denen protokolliert werden soll
?:100
Wollen Sie das Protokoll in eine Datei schreiben lassen (j/n)?
?:n
Wollen Sie ein Grafikprotokoll erzeugen lassen (j/n)?
?:n

Training mit 1 Zelle(n) in der 2. Ebene
100 Zyklen --> Fehler: 4.0598868
200 Zyklen --> Fehler: 4.0506323
300 Zyklen --> Fehler: 4.1058980
400 Zyklen --> Fehler: 4.0609778
500 Zyklen --> Fehler: 4.0492746
600 Zyklen --> Fehler: 4.0433440

Training mit 2 Zellen in der 2. Ebene
700 Zyklen --> Fehler: 4.0872246
800 Zyklen --> Fehler: 4.0416591
900 Zyklen --> Fehler: 4.0672079
1000 Zyklen --> Fehler: 4.1457407
1100 Zyklen --> Fehler: 4.1358572
1200 Zyklen --> Fehler: 4.1298799

Training mit 3 Zellen in der 2. Ebene
1300 Zyklen --> Fehler: 4.0836530
1400 Zyklen --> Fehler: 4.1211505
1500 Zyklen --> Fehler: 4.1052592
1600 Zyklen --> Fehler: 3.6233996
1700 Zyklen --> Fehler: 2.6528936
1800 Zyklen --> Fehler: 2.6464125
1900 Zyklen --> Fehler: 2.6482162
2000 Zyklen --> Fehler: 2.6003756
2100 Zyklen --> Fehler: 2.5940512
2200 Zyklen --> Fehler: 2.4131706
2300 Zyklen --> Fehler: 1.0302481
```

```
2400 Zyklen --> Fehler: 0.85996974
2500 Zyklen --> Fehler: 0.83165428
2600 Zyklen --> Fehler: 0.82032469
2700 Zyklen --> Fehler: 0.82207577
2800 Zyklen --> Fehler: 0.82564480
2900 Zyklen --> Fehler: 0.81086356
3000 Zyklen --> Fehler: 0.81019137

Training mit 4 Zellen in der 2. Ebene
3100 Zyklen --> Fehler: 0.54559353
3200 Zyklen --> Fehler: 0.17158649
3300 Zyklen --> Fehler: 0.064404523
3400 Zyklen --> Fehler: 0.016733352
3500 Zyklen --> Fehler: 0.0051034293
3600 Zyklen --> Fehler: 0.0015779329

Geschafft! Die Anzahl der Neuronen in der mittleren Ebene betraegt: 4
Es wurden insgesamt 3636 Lernzyklen benoetigt
>:savenet par4.net
>:
```

Rumelhart, Hinton und Williams (siehe [RUM86]) brauchten für die Lösung dieses Problems 2825 Zyklen bei einer Lernrate von $\eta = 0.5$. Das hier verwendete Verfahren erforderte nur 800 Zyklen mehr, um eine Lösung mit der minimal benötigten Netztopologie zu erhalten (in [RUM86] wurde aber nicht angegeben, welche untere Fehlermaßgrenze verwendet wurde).

In Abbildung 21.1 ist der gesamte Trainingsverlauf dargestellt:

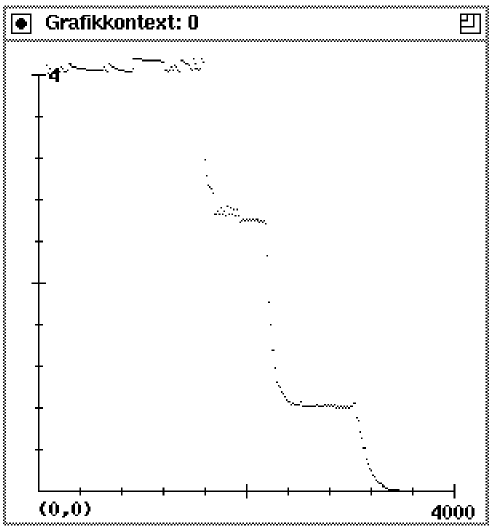


Abbildung 21.1: Der Trainingsverlauf beim Lösen des Paritätsproblems

Die Datei `par4.net`, die das Neuronale Netz enthält, hat jetzt den folgenden Inhalt:

```
netname "";
layer 3 : 4 , 4 , 1;
NET:

/* Unit 1,1: */
-6.5190572; /* Weight to Unit 2,1 */
-7.8331061; /* Weight to Unit 2,2 */
-7.5010938; /* Weight to Unit 2,3 */
3.2675980; /* Weight to Unit 2,4 */

/* Unit 1,2: */
-6.5214995; /* Weight to Unit 2,1 */
-7.8325285; /* Weight to Unit 2,2 */
-7.5032319; /* Weight to Unit 2,3 */
3.2677222; /* Weight to Unit 2,4 */

/* Unit 1,3: */
-6.5220215; /* Weight to Unit 2,1 */
-7.8336560; /* Weight to Unit 2,2 */
-7.4995377; /* Weight to Unit 2,3 */
3.2675077; /* Weight to Unit 2,4 */

/* Unit 1,4: */
-6.5165575; /* Weight to Unit 2,1 */
-7.8316700; /* Weight to Unit 2,2 */
-7.5042283; /* Weight to Unit 2,3 */
3.2677808; /* Weight to Unit 2,4 */

/* Unit 2,1: */
3.0584476; /* Bias */
-12.855525;/* Weight to Unit 3,1 */

/* Unit 2,2: */
10.871136; /* Bias */
13.480153;/* Weight to Unit 3,1 */

/* Unit 2,3: */
17.636575; /* Bias */
-12.666334;/* Weight to Unit 3,1 */

/* Unit 2,4: */
-11.608756; /* Bias */
-12.822137;/* Weight to Unit 3,1 */

/* Unit 3,1: */
6.3348546; /* Bias */
```

21.3 Das ProSiNN–Programm zum Training mit dynamischer Neuronenerzeugung

Das Programm, das ein Netz mit dynamischer Neuronenerzeugung trainiert, ist in der Datei `abwdnc.prg` gespeichert. Der Inhalt dieses Files ist:

```
/* Simulationsprogramm, das beim Netztraining die Anzahl der Neuronen in
   der mittleren Ebene dynamisch erhoeht, wenn es zur Ausbildung eines
   Plateaus beim Netzfehler kommt.
   Es wird immer ein Netz mit 3 Schichten erzeugt.
   Der Benutzer muss folgendes eingeben:
   - Die Anzahl der Eingabezellen
   - Die Startanzahl der Neuronen in der mittleren Ebene
   - Die Anzahl der Ausgabezellen
   - Die Anzahl der Neuronen, die im Misserfolgsfall hinzugefuegt werden
     sollen
   - Die Fensterbreite, nach der der Netzfehler zu untersuchen ist
   - Das relative Verhaeltnis zwischen altem und neuem Netzfehler
     (Schwellenwert)
   - Die untere Fehlermassgrenze
   - Die Trainingsmusterdatei
   - Abfrage, ob ein Textprotokoll erzeugt werden soll:
     + Wenn ja, dann wird nach der Anzahl der Zyklen gefragt, nach der
       eine Protokollzeile ausgegeben werden soll
     + Anschliessend wird gefragt, ob eine Protokolldatei generiert
       werden soll
   - Abfrage, ob ein Grafikprotokoll erzeugt werden soll
   Das Netz wird anschliessend solange trainiert, bis der Lernerfolg
   erzielt wird.
*/

/* Definiere alle benoetigten Variablen und initialisiere sie */
define int nr_1;       /* Startanzahl der Neuronen in der 1. Ebene */
define int start_nr_2; /* Startanzahl der Zellen in der 2. Schicht */
define int nr_3;       /* Startanzahl der Neuronen in der 3. Ebene */
define int nr_new_units;
                       /* Anzahl neu hinzukommender Zellen in der
                          mittleren Schicht                      */
define int window;     /* Fensterbreite, nach der getestet wird */
define double Delta_T; /* Verhaeltnis zwischen altem und neuem Netzfehler
                          (Schwellenwert) */
define string pattern_file;
                       /* Name der Trainingsmusterdatei */
define string flag;    /* zum Einlesen einer Entscheidung: (j/n) */
```

```
define int protocol_distance;
                       /* Anzahl Zyklen, nach der protokolliert
                          werden soll                          */
define string protocol_file;
                       /* Name der Protokolldatei */
define int protocol_file_handle = stdout;
                       /* Protokoll-Dateihandle;
                          Default: Ein/Ausgabefenster */
define int win_handle; /* Handle des zu oeffnenden Fensters */
define double max_x;   /* Breite des X-Bereichs im WKS */
define double E_t_minus_w;
                       /* Netzfehler vor den letzten w Trainingszyklen */
define double E_t_0;   /* Netzfehler, als Neuronen hinzugefuegt worden
                          sind                                         */


/* gebe eine Titelzeile aus */
println;
print "Programm, das dynamisch Neuronen hinzufuegt, wenn sich das";
print " Fehlermass"; println;
print "nicht verbessert. Es wird ein Netz mit 3 Ebenen verwendet!";
println; println;

/* lese alle benoetigten Benutzereingaben ein */
while( nr_1 < 1 )
{ /* lese die Anzahl der Eingabe-Zellen ein */
  print "Geben Sie die Anzahl der Eingabeneuronen ein";
  input nr_1;
};
while( start_nr_2 < 1 )
{ /* lese die Startanzahl der Neuronen in der mittleren Ebene ein */
  print "Geben Sie die Startanzahl der Neuronen in der versteckten";
  print " Schicht ein";
  input start_nr_2;
};
while( nr_3 < 1 )
{ /* lese die Anzahl der Ausgabe-Zellen */
  print "Geben Sie die Anzahl der Ausgabeneuronen ein";
  input nr_3;
};
while( nr_new_units < 1 )
{ /* lese die Anzahl der Zellen ein, die bei Misserfolg hinzugefuegt
     werden                                                          */
  print "Geben Sie die Anzahl der Neuronen ein, die ";
  print "im Misserfolgsfall hinzugefuegt werden";
  input nr_new_units;
};
```

```
while( window < 1 )
{ /* lese die Fensterbreite ein, nach der der Netzfehler zu untersuchen
     ist                                                                */
  print "Geben Sie die Fensterbreite w ein (z.B.: 1000)";
  input window;
};
while( Delta_T <= 0.0 )
{ /* lese das relative Verhaeltnis zwischem altem und neuem Fehler
     (Schwellenwert) ein                                           */
  print "Geben Sie das relative Fehlerverhaeltnis ein (z.B.: 0.05)";
  input Delta_T;
};
/* lese die untere Fehlermassgrenze zum Terminieren des Verfahrens ein */
print "Geben Sie die untere Fehlermassgrenze ein (z.B.: 0.001)";
input error_limit;
/* lese die Trainingsmusterdatei mit Pfad ein */
print "Geben Sie die Trainings-Musterdatei mit Pfad ein";
input pattern_file;
/* Frage jetzt an, ob ein Textprotokoll erzeugt werden soll */
while( flag = "" )
{ /* solange noch nicht "j" oder "n" eingegeben wurde, tue */
  print "Wollen Sie ein Textprotokoll erzeugen lassen (j/n)?";
  input flag;
  if( flag = "j" )
  { /* es soll ein Textprotokoll erzeugt werden */
    /* lese jetzt die Anzahl der Zyklen ein, nach denen protokolliert
       werden soll                                                   */
    while( protocol_distance < 1 )
    { /* lese Zyklendifferenz ein */
      print "Geben Sie die Anzahl der Zyklen ein, nach denen";
      print " protokolliert werden soll";
      input protocol_distance;
    };
    /* Frage jetzt an, ob das Protokoll in eine Datei geschrieben
       werden soll                                               */
    set flag = ""; /* initialisiere fuer die folgende Abfrage */
    while( flag = "" )
    { /* solange noch nicht "j" oder "n" eingegeben wurde, tue */
      print "Wollen Sie das Protokoll in eine Datei schreiben lassen";
      print " (j/n)?";
      input flag;
      if( flag = "j" )
      { /* hole Filenamen, oeffne die Datei und generiere
           das Textprotokoll                               */
        print "Geben Sie die Protokolldatei mit Pfad ein";
        input protocol_file;
        open protocol_file protocol_file_handle; /* oeffne die Datei */
```

```
        protocol protocol_file_handle protocol_distance: sum_iterations,
                                                          act_error,EOL;
      };
      else
      { /* enweder "n" oder nicht erlabte Zeichen */
        if( flag = "n" )
        { /* Textprotokoll soll in das Ein/Ausgabefenster
             gelenkt werden                              */
          protocol stdout protocol_distance: sum_iterations,
                                             "Zyklen --> Fehler: ",
                                             act_error,EOL;
        };
        else
        { /* nicht "j" und nicht "n" gelesen */
          set flag = "";
        };
      };
    };
  };
  else
  { if( flag = "n" )
    { /* es soll kein Textprotokoll erzeugt werden, tue nichts */
    };
    else
    { /* nicht "j" und nicht "n" gelesen */
      set flag = "";
    };
  };
};
/* nun wurde evtl. ein Textprotokoll erzeugt */
/* erzeuge jetzt das Netz und initialisiere es */
layer 3: nr_1 , start_nr_2 , nr_3;
init_net;
/* lese die Trainingsmusterdatei ein */
readpattern pattern_file;
/* fuehre nun einen Lernschritt (1 Zyklus) durch */
cycle 1; /* 'act_error' wird initialisiert */
/* Frage jetzt, ob ein Grafikprotokoll erzeugt werden soll */
set flag = ""; /* initialisiere fuer die folgende Abfrage */
while( flag = "" )
{ /* solange noch nicht "j" oder "n" eingegeben wurde, tue */
  print "Wollen Sie ein Grafikprotokoll erzeugen lassen (j/n)?";
  input flag;
  if( flag = "j" )
  { /* es soll ein Grafikprotokoll erzeugt werden */
    double_of(window) to max_x; /* setze den maximalen x-Bereich im WKS */
    open_grafic 300,300 to win_handle; /* erzeuge das Grafikfenster */
```

```
    set_world 0.0 , max_x , 0.0 , act_error to win_handle; /* setze WKS */
    axes 0.0 , max_x , 0.0 , act_error , 0.0 , 0.0 to win_handle;
              /* zeichne die Koordinatenachsen */
    grafic_protocol win_handle: act_error;
                    /* definiere das Grafikprotokoll */
  };
  else
  { if( flag = "n" )
    { /* es soll kein Grafikprotokoll erzeugt werden, tue nichts */
    };
    else
    { /* nicht "j" und nicht "n" gelesen */
      set flag = "";
    };
  };
};
/* merke nun den aktuellen Fehler */
set E_t_0 = act_error; /* merke den aktuellen Fehler */
set E_t_minus_w = act_error; /* merke den aktuellen Fehler */
/* gebe die Startanzahl der Zellen in der versteckten Schicht aus */
println protocol_file_handle;
print "Training mit " protocol_file_handle;
print nr_units:2 protocol_file_handle;
print " Zelle(n) in der 2. Ebene" protocol_file_handle;
println protocol_file_handle;
/* trainiere das vorliegende Netz max. einmal ueber die Fensterbreite */
cycle window; /* trainiere */
/* nun kommt die Schleife, die das Lernverfahren enthaelt */
while( act_error > error_limit )
{ /* Der aktuelle Fehler ist groesser als
     der untere Grenzwert: weiter lernen  */
  while( (E_t_minus_w - act_error) / E_t_0 < Delta_T )
  { /* Das relative Verhaeltnis ist zu gross ==> fuege Neuronen hinzu */
    set E_t_minus_w = act_error;
                     /* merke aktuellen Fehler als letzten Fehler */
    set E_t_0 = act_error;
               /* merke Fehler (als Knoten hinzugefuegt wurden) */
    alter_unit_number in 2 to nr_units:2 + nr_new_units;
         /* erhoehe die Anzahl der Neuronen in der mittleren Schicht */
    /* gebe nun eine Meldung aus */
    println protocol_file_handle;
    print "Training mit " protocol_file_handle;
    print nr_units:2 protocol_file_handle;
    print " Zellen in der 2. Ebene" protocol_file_handle;
    println protocol_file_handle;
    /* trainiere das vorliegende Netz maximal
       einmal ueber die Fensterbreite         */
```

```
    cycle window; /* trainiere */
    if( act_error < error_limit )
    { /* Die untere Fehlergrenze wurde erreicht ==> fertig! */
      /* gebe eine Abschlussmeldung aus */
      println protocol_file_handle;
      print protocol_file_handle;
      print "Geschafft! Die Anzahl der Neuronen in der"
            protocol_file_handle;
      print " mittleren Ebene betraegt: "
            protocol_file_handle;
      print nr_units:2 protocol_file_handle;
      println protocol_file_handle;
      print "Es wurden insgesamt " protocol_file_handle;
      print sum_iterations protocol_file_handle;
      print " Lernzyklen benoetigt" protocol_file_handle;
      println protocol_file_handle;
      close protocol_file_handle;
      quit;
    };
  };
  /* hier gilt: es muessen keine neuen Neuronen hinzugefuegt werden */
  set E_t_minus_w = act_error;
                    /* merke aktuellen Fehler als letzten Fehler */
  /* trainiere das vorliegende Netz max. einmal ueber die Fensterbreite */
  cycle window; /* trainiere */
};
/* hier gilt:  Die untere Fehlergrenze wurde erreicht ==> fertig! */
/* gebe eine Abschlussmeldung aus */
println protocol_file_handle;
print "Geschafft! Die Anzahl der Neuronen in der" protocol_file_handle;
print " mittleren Ebene betraegt: " protocol_file_handle;
print nr_units:2 protocol_file_handle;
println protocol_file_handle;
print "Es wurden insgesamt " protocol_file_handle;
print sum_iterations protocol_file_handle;
print " Lernzyklen benoetigt" protocol_file_handle;
println protocol_file_handle;
close protocol_file_handle;
quit;
```

Literaturverzeichnis

[AHO88] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Compiler–Bau
Teil 1 und Teil 2, Addison–Wesley 1988

[AND70] James A. Anderson
Two Models for Memory Organization using Interacting Traces
page 137–160, Mathematical Biosciences 8, 1970

[AND77] James A. Anderson
Neural Networks with Cognitive Implications
abgedruckt in:
D. LaBerge, S.J. Samuels (Eds.)
Basic Processes in Reading Perception and Comprehension
page 27–90, Erlbaum 1977

[ASH89] Timur Ash
Dynamic Node Creation in Backpropagation Networks
page 365–375, Connection Science, Volume 1, No.4, 1989

[FEL88] Wolf–Dietrich Fellner
Computer–Grafik
BI–Wissenschaftsverlag 1988

[GRO80] Stephen Grossberg
How does the brain build a cognitive code?
page 1–51, Psychological Review 87, 1980

[HEB49] D.O. Hebb
The Organization of Behavior
Wiley 1949

[HEC89] R. Hecht–Nielsen
Theory of the Backpropagation Neural Network
Proceedings of the International Joint Conference on Neural Networks I
page 593–611, 1989

[HOP82] John Hopfield
Neural Networks and physical systems with emergent collective computational abilities
Proceedings of the National Academy of Sciences 79
page 2554–2558, 1982

[HOR89] K. Hornik, M. Stinchcombe, H. White
Multilayer feedforward networks are universal approximators
Neural Networks 1, 1988

[JON89] Oliver Jones
Introduction to the X Window System
Prentice Hall 1989

[KEM88] C. Kemke
Der Neuere Konnektionismus: Ein Überblick
Seite 143–162, Informatik–Spektrum 11, Springer–Verlag 1988

[KER83] Brian W. Kernighan, Dennis M. Ritchie
Programmieren in C
Hanser Verlag 1983

[MAN90] Niall Mansfield
Das Benutzerhandbuch zum X Window-System
Addison–Wesley 1990

[MCC91] Joel McCormack, Paul Asente and Ralph R. Swick
X Toolkit Intrinsics — C Language Interface
X Window System: X Version 11, Release 5
MIT, Cambridge and DEC, Maynard 1991

[MIN69] Marvin Minsky, Seymour Papert
Perceptrons
MIT Press 1969

[MIR89] Gagan Mirchandani, Wei Cao
On Hidden Nodes for Neural Nets
page 661–664, IEEE Transactions on Circuits and Systems,
Volume 36, No. 5, 1989

[NAU90] Detlef Nauck
Zur Wissensrepräsentation und Wissensverarbeitung in künstlichen Neuronalen Netzen
Diplomarbeit, Institut für Betriebssysteme und Rechnerverbund,
Technische Universität Braunschweig 1990

[NEW86] William M. Newman, Robert F. Sproull
Grundzüge der interaktiven Computergrafik
McGraw–Hill 1986

[PET91] Chris D. Peterson
Athena Widget Set — C Language Interface
X Window System: X Version 11, Release 5
MIT X Consortium 1991

[PIT43] W. Pitts, W.S. McCulloch
A Logical Calculus of the Ideas Immament in Nervous Activity
page 115–133, Bulletin of Mathematical Biophysics 5, 1943
abgedruckt in:
Neurocomputing, Foundations of Research
herausgegeben von James A. Anderson und Edward Rosenfeld, MIT Press

[RIT91] Helge Ritter, Thomas Martinetz, Klaus Schulten
Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke
Addison–Wesley 1991

[ROJ93] Raúl Rojas
Theorie der neuronalen Netze: Eine systematische Einführung
Springer–Verlag 1993

[ROS58] Frank Rosenblatt
Two Theorems of Statistical Separability in the Perceptron
abgedruckt in:
Mechanisation of Thought Processes:
Proceedings of a Symposium held at the National Physical Laboratory, November 1958
page 421–456, Volume 1, HM Stationery 1959, Volume 1

[ROS62] Frank Rosenblatt
Principles of Neurodynamics
Spartan 1962

[RUM86] D.E. Rumelhart, J.L. McClelland and the PDP Research Group
Parallel Distributed Processing:
Explorations in the Microstructure of Cognition
Volume 1: Foundations
The MIT Press, Cambridge 1986

[RUM88] D.E. Rumelhart, J.L. McClelland
Explorations in Parallel Distributed Processing:
A Handbook of Models, Programs and Exercises
The MIT Press, Cambridge 1988

[VOG88] T.P.Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink and D.L. Alkon
Accelerating the Convergence of the Back–Propagation Method
page 257–263, Biological Cybernetics 59, Springer–Verlag 1988

[WID60] G. Widrow, M. E. Hoff
Adaptive Switching Circuits
Institute of Radio Engineers,
Western Electronic Show and Convention,
page 96 - 104, Convention Record 1960, Part 4

[YOU89] Douglas A. Young
X Window Systems: Programming and Applications with Xt
Prentice–Hall 1989

Teil V

Anhänge

Anhang A

Referenzliste der Schlüsselwörter und vordefinierten Bezeichner

Dieser Anhang enthält die alphabetisch sortierte Liste aller Schlüsselwörter und vordefinierten Bezeichner.
Jede Zeile der folgenden Tabelle besteht aus dem Namen, der Kurzbeschreibung und der Seitenangabe, auf der das Schlüsselwort oder der vordefinierte Bezeichner eingeführt wurde.

Name	**Kurzbeschreibung**	**Seite**
acos	`Gleitkommafunktion:` *Arcuskosinus*	68
act_alpha	`Gleitkommavariable:` *Aktuelles Lernmoment*	55
act_error	`Gleitkommavariable:` *Aktuelles Fehlermaß*	53
act_eta	`Gleitkommavariable:` *Aktuelle Lernrate*	55
act_iterations	`Integervariable:` *Aktuelle Anzahl ausgeführter Lernzyklen*	56
activation	`Kommando:` *Neue Aktivierungsfunktion setzen*	169
add_layer	`Kommando:` *Eine Netzebene hinzufügen*	142
alias	`Kommando:` *Makro definieren*	92
alpha	`Gleitkommavariable:` *Lernmoment*	55
alter_unit_number	`Kommando:` *Neuronenanzahl ändern*	144

Name	Kurzbeschreibung	Seite
asin	`Gleitkommafunktion:` *Arcussinus*	68
atan	`Gleitkommafunktion:` *Arcustangens*	68
axes	`Kommando:` *Koordinatenachsen zeichnen*	117
beta	`Gleitkommavariable:` *Mißerfolgslernratenfaktor*	55
betragsmaessig	`Integerkonstante:` *Zum Setzen des Fehlermaßmodus*	51, 54
cd	`Kommando:` *Arbeitsverzeichnis wechseln*	198
clear_grafic	`Kommando:` *Inhalt eines Grafikfensters löschen*	126
close	`Kommando:` *Datei schließen*	108
close_all_grafics	`Kommando:` *Alle Grafikfenster beenden*	116
close_grafic	`Kommando:` *Ein Grafikfenster beenden*	115
copy	`Kommando:` *Teilstring kopieren*	88
cos	`Gleitkommafunktion:` *Kosinus*	68
cosh	`Gleitkommafunktion:` *Kosinushyperbolikus*	68
cycle	`Kommando:` *Trainiere Netz mit allen Mustern*	164
cycle_pattern	`Kommando:` *Trainiere Netz mit einem Muster*	166
define	`Kommando:` *Variable definieren*	71
delete	`Kommando:` *Teilstring löschen*	89
delta_pj	`Neuronenpräfix:` *Zum Zugriff auf das Fehlersignal*	60
delta_theta	`Neuronenpräfix:` *Zum Zugriff auf den Änderungswert*	60
delta_w	`Gewichtspräfix:` *Zum Zugriff auf den Änderungswert*	62
derivation_of	`Kommando:` *Funktion ableiten*	86
derivation_of_activation_funct	`Kommando:` *Aktivierungsfunktionsableitung setzen*	171

Name	Kurzbeschreibung	Seite
dir	`Kommando:` *Arbeitsverzeichnisinhalt ausgeben*	197
double_of	`Kommando:` *Typumwandlung von int nach double*	85
e	`Gleitkommakonstante:` *Wert: 2.7182818284589998*	50
EOL	`Stringkonstante:` *Zeilenvorschub*	51
error_limit	`Gleitkommavariable:` *Untere Fehlermaßgrenze*	54
error_mode	`Gleitkommavariable:` *Fehlermaßmodus*	54
error_of	`Kommando:` *Netzantwortfehler einem Array zuweisen*	176
eta	`Gleitkommavariable:` *Lernrate*	55
exp	`Gleitkommafunktion:` *Exponentialfunktion*	68
fail_limit	`Gleitkommavariable:` *Mißerfolgsfaktor*	55
fast_cycle	`Kommando:` *Schnelles Netztraining*	177
floatpattern	`Integervariable:` *Einlesemodus für Muster*	52
free	`Kommando:` *Variablen freigeben*	195
free_all	`Kommando:` *Alles freigeben*	196
free_all_pattern	`Kommando:` *Alle Muster freigeben*	160
free_grafic_protocol	`Kommando:` *Alle Grafikprotokolle freigeben*	187
free_net	`Kommando:` *Netz freigeben*	148
free_pattern	`Kommando:` *Ein Muster freigeben*	159
free_protocol	`Kommando:` *Alle Textprotokolle freigeben*	184
free_var	`Kommando:` *Eine Variable freigeben*	93
grafic_protocol	`Kommando:` *Grafikprotokoll erzeugen*	184
if	`Kommando:` *Bedingte Verzweigung*	97

Name	Kurzbeschreibung	Seite
init	`Kommando:` *Systemvariablen initialisieren*	194
init_net	`Kommando:` *Netz initialisieren*	130
input	`Kommando:` *Wert über die Tastatur einlesen*	103
insert	`Kommando:` *Teilstring einfügen*	90
int_of	`Kommando:` *Typumwandlung von double nach int*	84
layer	`Kommando:` *Netz erzeugen*	127
length_of	`Kommando:` *Stringlänge bestimmen*	91
line	`Kommando:` *Linie in eine Grafik zeichnen*	121
ln	`Gleitkommafunktion:` *Natürlicher Logarithmus*	68
load	`Kommando:` *ProSiNN-Programm ausführen*	99
log	`Gleitkommafunktion:` *Dekadischer Logarithmus*	68
max_betrag	`Integerkonstante:` *Zum Setzen des Fehlermaßmodus*	51, 54
mittel_betrag	`Integerkonstante:` *Zum Setzen des Fehlermaßmodus*	51, 54
names	`Kommando:` *Alle definierten Bezeichner ausgeben*	191
net_name	`Stringvariable:` *Netzname*	52
netname	`Kommando:` *Netznamen setzen*	128
normal_activation	`Kommando:` *'Normale' Aktivierungsfunktion setzen*	172
nr_in_units	`Integervariable:` *Anzahl der Netzeingabeneuronen*	53
nr_of_layers	`Integervariable:` *Anzahl der Netzebenen*	53
nr_out_units	`Integervariable:` *Anzahl der Netzausgabeneuronen*	53
nr_pattern	`Integervariable:` *Anzahl der vorhandenen Trainingsmuster*	52
nr_units	`Netzebenenpräfix:` *Zum Zugriff auf die Neuronenanzahl*	59

Name	Kurzbeschreibung	Seite
pwd	`Kommando:` *Arbeitsverzeichnis ausgeben*	198
quadratisch	`Integerkonstante:` *Zum Setzen des Fehlermaßmodus*	51, 54
quit	`Kommando:` *Ein Programm oder ProSiNN beenden*	201
read_test_pattern	`Kommando:` *Testmuster einlesen*	155
readnet	`Kommando:` *ProSiNN–Netz laden*	142
readpattern	`Kommando:` *Trainingsmuster einlesen*	152
reply_to	`Kommando:` *Netzantwort ausgeben*	173
save_c_net	`Kommando:` *Netz als C–Programm speichern*	138
savenet	`Kommando:` *ProSiNN–Netz speichern*	133
set	`Kommando:` *Wert an eine Variable zuweisen*	79
set_rand	`Kommando:` *Startwert für Zufallszahlengenerator setzen*	130
set_world	`Kommando:` *Weltkoordinatensystem setzen*	116
shell	`Kommando:` *Betriebssystemkommando ausführen*	200
sin	`Gleitkommafunktion:` *Sinus*	67
sinh	`Gleitkommafunktion:` *Sinushyperbolikus*	68
sqrt	`Gleitkommafunktion:` *Quadratwurzel*	67
status	`Kommando:` *Statusinformationen ausgeben*	192
stdout	`Integerkonstante:` *Dateihandle für das Ein/Ausgabefenster*	51
sub_layer	`Kommando:` *Eine Netzebene entfernen*	143
sum_iterations	`Integervariable:` *Gesamtanzahl ausgeführter Lernzyklen*	56
sys_names	`Kommando:` *Systemvariablen ausgeben*	189
tan	`Gleitkommafunktion:` *Tangens*	68

Anhang B

Referenzliste der ProSiNN-Fehlermeldungen

Dieser Anhang enthält die numerisch sortierte Liste der von ProSiNN verwendeten Fehler. In der folgenden Aufzählung wird für jeden Fehler die Nummer und eine kurze Beschreibung angegeben.

Beim Auftreten eines Fehlers gibt ProSiNN die Nummer, eine Erläuterung und — wenn möglich — den Kontext mit aus. Entsteht der Fehler bei der Ausführung eines Programms, so wird außerdem noch der Programmname und die Zeile, in der der Fehler erkannt wurde, ausgegeben.

1. Diese *Fehlernummer* wird nur intern verwendet und kennzeichnet eine korrekte Befehlsausführung.

2. Diese *Fehlernummer* wird nur intern verwendet und kennzeichnet das Erreichen des Endes einer Struktur, in der die auszuwertende Befehlsfolge (oder die einzulesende Datei) enthalten ist.

3. Diese *Fehlernummer* wird nur intern verwendet und kennzeichnet, daß der **quit**–Befehl interpretiert wurde.

4. Der Benutzer hat über die Tastatur `Control-C` eingegeben. ProSiNN hat den letzten Befehl oder ein Programm beendet.

5. Es ist ein Folgefehler aufgetreten. Die Meldung des auslösenden Fehlers wurde aber schon ausgegeben.

6. Hier handelt es sich um einen Betriebssystemfehler. Ein von ProSiNN intern angeforderter Speicherbereich konnte nicht an das System zurückgegeben werden.

7. Der vorliegende Bezeichner sollte als Kommando ausgeführt werden. Er ist aber im System nicht definiert.

8. Der vorliegende Bezeichner sollte als Kommando ausgeführt werden. Er ist im System definiert, stellt aber kein ausführbares Kommando dar.

9. Es wurde ';' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
10. Der vorliegende Bezeichner enthält nicht erlaubte Zeichen.
11. Der vorliegende Bezeichner ist nicht im System definiert.
12. Es wurde versucht, ein vom System vordefinierten Bezeichner erneut zu definieren.
13. Es wurde ein Integer erwartet. Der vorliegende Bezeichner ist aber nicht als Integer definiert.
14. Dieser Fehler darf **nicht** auftreten. Wenn er trotzdem vorkommt, sind bei der Implementierung des ProSiNN–Systems Fehler gemacht worden!
15. Ein von ProSiNN vom Betriebssystem angeforderter Speicherbereich konnte nicht erhalten werden.
16. Es wurde ein Double erwartet. Der vorliegende Bezeichner ist aber nicht als Double definiert.
17. Die vorliegende Integerkonstante ist ungültig.
18. Die vorliegende Doublekonstante ist ungültig.
19. Es wurde ein String erwartet. Der vorliegende Bezeichner ist aber nicht als String definiert.
20. Die vorliegende Stringkonstante ist ungültig.
21. Das vorliegende Kommando ist unvollständig.
22. Es fehlen Parameter.
23. Es wurde eine Layerangabe erwartet. Der vorliegende Bezeichner ist aber nicht als Layer definiert.
24. Die vorliegende Layerkonstante ist ungültig.
25. Die vorliegende Layerangabe ist unkorrekt.
26. Es fehlen Parameter oder eine eingelesene Datei ist unvollständig.
27. Es wurde eine Unitangabe erwartet. Der vorliegende Bezeichner ist aber nicht als Unit definiert.
28. Es wurde eine Integerangabe erwartet, deren Wert größer oder gleich 0 ist.
29. Es wurde ':' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
30. Die vorliegende Unitangabe ist unkorrekt.
31. Es wurde eine Weightangabe erwartet. Der vorliegende Bezeichner ist aber nicht als Weight definiert.
32. Es wurde eine Unitangabe erwartet.

33. Die vorliegende Weightangabe ist unkorrekt.
34. Der angegebene Parameter ist falsch.
35. Es wurde ein Operator erwartet.
36. Die Indexangabe eines Feldelementes ist falsch.
37. Es wurde '[' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
38. Es wurde ']' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
39. Ein angegebenes Filehandle verläßt den erlaubten Bereich oder ist nicht belegt (d.h., es ist nicht mit einer zum Schreiben geöffneten Datei verbunden).
40. Die Schreiboperation in eine Datei ist nicht durchführbar.
41. Es wurde ein Filehandle erwartet.
42. Es wurde versucht, einer Konstanten einen neuen Wert zuzuweisen.
43. Die vorliegende Zuweisung ist nicht erlaubt.
44. Es wurde '=' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
45. Es wurde ein arithmetischer Operator ('+', '-', '*' oder '/') erwartet.
46. Es wurde versucht, einer Variablen einen Wert vom falschen Typ zuzuweisen.
47. Der angegebene Operator ist hier nicht erlaubt.
48. Der Variablentyp ist falsch.
49. Es wurde versucht, durch 0 zu dividieren.
50. Es wurde versucht, die untere Fehlermaßgrenze auf einen Wert kleiner oder gleich 0 zu setzen.
51. Es wurde versucht, auf interne Bestandteile eines Eingabeneurons zuzugreifen (sie existieren real aber nicht).
52. Es wurde versucht, auf interne Bestandteile eines Gewichtes zuzugreifen, dessen Quelle in der letzten Netzebene liegt (dieses Gewicht existiert aber nicht).
53. Es wurde '{' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
54. Es wurde '}' erwartet; dieses Zeichen konnte aber nicht gefunden werden.
55. Es fehlt ein Parameter.
56. ProSiNN versuchte vergeblich über das Betriebssystem die Länge einer Datei festzustellen.
57. Es liegt ein Fehler in der Kommentarstruktur vor.
58. Eine Stringkonstante ist unvollständig. Die Anführungszeichen (") fehlen.

59. Eine zum Schreiben geöffnete Datei konnte nicht geschlossen werden.

60. Es wurde versucht, einen im System vorliegenden Bezeichner erneut zu definieren.

61. Dieser Bezeichner darf nicht durch den *free_var*–Befehl freigegeben werden.

62. Bei der Ausführung des *shell*–Kommandos konnte auf den Systemkommando–Interpreter nicht zugegriffen werden.

63. Bei der Ausführung des *shell*–Kommandos entstand ein Betriebssystemfehler.

64. Beim Wechsel des Arbeitsverzeichnisses trat ein Fehler auf. Das angegebene Verzeichnis existiert nicht oder ist geschützt.

65. Das Arbeitsverzeichnis konnte nicht ausgegeben werden.

66. Es wurde '(' erwartet; dieses Zeichen konnte aber nicht gefunden werden.

67. Es wurde ')' erwartet; dieses Zeichen konnte aber nicht gefunden werden.

68. Es wurde eine Stringangabe erwartet.

69. Es wurde eine Doubleangabe erwartet.

70. Es wurde eine Integerangabe erwartet.

71. Es wurde eine Weightangabe erwartet.

72. Die vorliegenden Argumente dürfen nicht miteinander verglichen werden.

73. Das im System vorhandene Netz muß vor Ausführung des vorliegenden Befehls freigegeben werden.

74. Die einzulesende Netzdatei ist unvollständig.

75. Beim Einlesen der Netzdatei wurde festgestellt, daß einzelne Daten fehlen.

76. Beim Einlesen einer Netz– oder Musterdatei wurde der *quit*–Befehl gefunden. Diese Anweisung darf hier aber nicht stehen.

77. Beim Einlesen der Netzdatei wurde das *layer*–Kommando zum Erzeugen des Netzes nicht gefunden.

78. Beim Einlesen einer Netz– oder Musterdatei dürfen nach den eigentlichen Nutzinformationen keine weiteren Daten folgen (abgesehen von Kommentaren).

79. Es ist kein Neuronales Netz im System vorhanden.

80. Beim Einlesen einer Musterdatei wurde festgestellt, daß einzelne Musterdaten fehlen.

81. Die Bereichsgrenzen bei symbolischen Mustern sind falsch: **pat_min** $\geq$ **pat_max**.

82. Eine symbolische Musterangabe ist falsch.

83. Es wurde ',' erwartet; dieses Zeichen konnte aber nicht gefunden werden.

84. Es wurde eine Variable erwartet.
85. Ein angegebener Protokollparameter ist ungültig.
86. Das beim *protocol*–Kommando angegebene Dateihandle ist nicht mehr gültig.
87. Die Schreiboperation in eine Protokolldatei ist nicht durchführbar.
88. Die Grenzen beim Initialisieren der Gewichte sind falsch: **w_min** $\geq$ **w_max**.
89. Die Grenzen beim Initialisieren der Schwellenwerte sind falsch: **theta_min** $\geq$ **theta_max**.
90. Beim *cycle*–, *cycle_pattern*– oder *fast_cycle*–Kommando ist die maximale Anzahl der auszuführenden Trainingszyklen falsch. Es wurde ein Integer größer oder gleich 1 erwartet.
91. Es sind keine Trainingsmuster im System vorhanden.
92. Es ist ein Fehler beim Dateiöffnen aufgetreten.
93. Es wurde eine Integervariable erwartet.
94. Es existiert kein freies Dateihandle mehr.
95. Der Wert, der der Variablen für die untere Fehlermaßgrenze **error_limit** zugewiesen werden soll, ist nicht erlaubt.
96. Der Wert, der der Variablen für das Lernmoment **alpha** zugewiesen werden soll, ist nicht korrekt.
97. Der Wert, der der Variablen für den Erfolgslernratenfaktor **phi** zugewiesen werden soll, ist nicht erlaubt.
98. Der Wert, der der Variablen für die Lernrate **eta** zugewiesen werden soll, ist nicht korrekt.
99. Der Wert, der der Variablen für den Mißerfolgslernratenfaktor **beta** zugewiesen werden soll, ist nicht erlaubt.
100. Der Wert, der der Variablen für den Mißerfolgsfaktor **fail_limit** zugewiesen werden soll, ist nicht korrekt.
101. Der Wert, der der Variablen für den Fehlermaßmodus **error_mode** zugewiesen werden soll, ist nicht erlaubt.
102. Es wurde der Name und evtl. der Pfad einer Datei erwartet.
103. Beim Einlesen der Netzdatei wurde das Kontrollwort '`NET:`' erwartet.
104. Beim Einlesen einer Trainingsmusterdatei wurde das Kontrollwort '`PATTERN:`' erwartet.
105. Beim *protocol*–Kommando wurde eine Zyklendistanz (nach der jeweils eine Protokollzeile auszugeben ist) angegeben, die kleiner als 1 ist.

106. Es wurde die Zeichenfolge '`in`' erwartet.

107. Es wurde die Zeichenfolge '`to`' erwartet.

108. Die im System vorhandenen Muster müssen vor Ausführung des vorliegenden Befehls freigegeben werden.

109. Das vorliegende *sub_layer*–Kommando ist nicht erlaubt, da das Netz nur zwei Ebenen enthält.

110. Es wurde eine Musterangabe erwartet.

111. Die vorliegende Musterangabe ist falsch.

112. Es wurde ein Trainingsmuster erwartet.

113. Es wurde ein Double–Array erwartet.

114. Die Elementanzahl des angegebenen Arrays ist zu klein.

115. Beim Einlesen einer Testmusterdatei wurde das Kontrollwort '`TEST-PATTERN:`' erwartet.

116. Bei einer Funktion wurde eine Integer– oder Doubleangabe erwartet.

117. Das Argument beim Auswerten der Tangensfunktion ist ungültig.

118. Das Argument beim Auswerten der Arcussinusfunktion ist ungültig.

119. Das Argument beim Auswerten der Arcuskosinusfunktion ist ungültig.

120. Das Argument beim Auswerten der Quadratwurzelfunktion ist ungültig.

121. Das Argument beim Auswerten der natürlichen Logarithmusfunktion ist ungültig.

122. Das Argument beim Auswerten der dekadischen Logarithmusfunktion ist ungültig.

123. Das Argument beim Auswerten der Sinushyperbolikusfunktion ist ungültig.

124. Das Argument beim Auswerten der Kosinushyperbolikusfunktion ist ungültig.

125. Bei der Auswertung eines Ausdrucks wurde versucht, 0^0 zu berechnen.

126. Bei der Auswertung eines Integerausdrucks wurde der Integerwertebereich verlassen.

127. Es wurde der Bezeichner einer Funktion erwartet.

128. Es wurde eine Integerangabe größer oder gleich 1 erwartet, die ein Funktionsargument spezifiziert.

129. Eine selbst definierte Aktivierungsfunktion existiert nicht.

130. Das Argument beim Auswerten der Exponentialfunktion ist zu groß.

131. Beim Ausführen des *fast_cycle*–Kommandos muß im System die 'normale' sigmoide Aktivierungsfunktion vorliegen.

132. Es wurde die Zeichenfolge '`at`' erwartet.

133. Eine Positionsangabe in einem String ist falsch.

134. Die Anzahl der zu löschenden Zeichen in einem String muß größer als 0 sein.

135. Der Inhalt des Arbeitsverzeichnisses konnte nicht ausgegeben werden.

136. Beim Laden oder Speichern eines Files wurde in der Dateiauswahlbox kein Filename angegeben.

137. Die angegebene Anzahl der Nachkommastellen ist zu groß.

138. Das vorliegende Semikolon darf hier nicht stehen.

139. Das durch die Dateiauswahlbox zum Editieren ausgewählte File läßt sich nicht zum Schreiben öffnen.

140. Es wurde kein Variablenname angegeben (beim Erzeugen einer Variablen im Menüsystem).

141. Der angegebene Variablenname ist im System schon definiert (beim Erzeugen einer Variablen im Menüsystem).

142. Der angegebene Variablenname enthält ungültige Zeichen (beim Erzeugen einer Variablen im Menüsystem).

143. Die angegebene Anzahl der Arrayelemente ist falsch (beim Erzeugen einer Feldvariablen im Menüsystem).

144. Beim Generieren eines Grafikfensters wurde eine falsche Angabe für die Breite oder die Höhe angegeben.

145. Es existieren keine freien Grafikhandles mehr.

146. Es wurde ein Grafikhandle erwartet.

147. Das angegebene Grafikhandle verläßt den erlaubten Bereich oder ist nicht belegt (d.h., es ist nicht mit einem offenen Grafikfenster verbunden).

148. Bei einer Koordinatenangabe ist das Minimum größer oder gleich dem Maximum.

149. Die Funktion, die in eine Grafik eingezeichnet werden soll, darf nur ein Argument haben.

150. Das vorliegende Weltkoordinatensystem eines Grafikfensters ist für die Definition des Grafikprototokolls ungeeignet, da der rechte Rand des Weltkoordinatensystems kleiner als 1 ist.

Anhang C

Dokumentation und Installation des ProSiNN–Programmsystems

Dieser Anhang enthält die essentiellen Informationen, die für weitere Erweiterungen und die Installation von ProSiNN benötigt werden.

Nachdem im ersten Abschnitt das Ablaufschema des in der Programmiersprache `C` implementierten Systems vorgestellt worden ist, werden die wesentlichen in den Quelldateien verwendeten Datenstrukturen eingeführt. Abschließend wird die Installation von ProSiNN beschrieben.

C.1 Das Ablaufschema des Systems

In diesem Abschnitt wird der *grobe* Systemablauf dargestellt. In einzelnen Schritten wird erläutert, wie Eingaben vom Benutzer angenommen und verarbeitet werden. In Klammern werden die jeweils zuständigen Funktionen und die Quelldateien, in denen sich diese Routinen befinden, angegeben.

- **Initialisierung der grafischen Oberfläche und des übrigen Systems** (`main()` in `net.c`):
  Zuerst werden alle *Widgets* (das sind die grafischen Grundelemente, aus denen die Programmoberfläche aufgebaut ist; näheres dazu findet man in [JON89], [MAN90], [MCC91], [YOU89] und [PET91]) für die Menüoberfläche und das Ein/Ausgabefenster erzeugt und mit gültigen Werten versehen. Im Anschluß initialisiert das System die übrigen Datenstrukturen (`init()` in `interpre.c`). Danach wird überprüft, ob der Benutzer die `p`–Option beim Programmstart verwendet hat. In diesem Fall wird in das Ein/Ausgabefenster der entsprechende *load*–Befehl geschrieben und das Kommando anschließend ausgewertet (eine Befehlsinterpretierung im Ein/Ausgabefenster führt `CommandEval()` in `net.c` durch). Als letztes wird eine `Xt`–Funktion (dies ist eine Funktion des `X Toolkit`; siehe Abschnitt C.3.1) aufgerufen, die auf *X Events* wartet und vorliegende Ereignisse an den *Event Handler* weitergibt. Diese Bearbeitungsroutine gibt die Events an die zugehörigen Widgets weiter, die je nach Art der Ereignisse reagieren und Aktionen ausführen.

- **Berarbeitung eines Menüpunktes**:
Nachdem ein Menüpunkt (der durch ein Widget realisiert ist) angewählt wurde, wird durch den generierten Event eine mit dem Menüwidget verbundene Funktion aufgerufen (diese Funktion befindet sich in einem Quellfile, dessen Dateiname mit `'menu_'` beginnt). Diese Routine erzeugt in der Regel neue Widgets und generiert daraus ein Popup–Fenster, das Eingaben vom Anwender anfordert. Nachdem die Daten vom Benutzer übernommen und ausgewertet worden sind (und evtl. eine Fehlermeldung ausgegeben wurde), wird die mit dem Menüpunkt verbundene Semantik direkt realisiert oder es wird aus den Eingaben ein ProSiNN–Kommando zusammengesetzt und dem Interpreter zur Bearbeitung übergeben (siehe unten).

- **Auswertung eines im Ein/Ausgabefenster eingegebenen ProSiNN–Kommandos**:
Nachdem im Ein/Ausgabefenster die `Return`–Taste gedrückt wurde, wird ein Event an das entsprechende Widget weitergeleitet, das wiederum eine Funktion zum Auswerten aufruft (`CommandEval()` in `net.c`). Anhand der alten und der aktuellen Position im Ein/Ausgabefenster wird die vorliegende Kommandozeile extrahiert und in einer speziellen Struktur (vom Typ `C_STRING`; siehe Unterabschnitt C.2.1) abgelegt. Im Anschluß wird das Gebilde dem Interpreter zur Bearbeitung übergeben (siehe unten). Tritt bei der Auswertung ein Fehler auf, so wird eine Meldung ausgegeben. Als letztes werden die Positionen aktualisiert und die verarbeitete Befehlszeile wird in einem internen Speicher (für die zuletzt eingegebene Kommandozeile) vermerkt.

- **Auswertung einer Befehlssequenz durch den Interpreter** (`interpret()` in `interpre.c`):
Die Funktion, die den Interpreter realisiert, bearbeitet sequentiell ein ProSiNN–Kommando nach dem anderen. Zur Auswertung eines Befehls wird eine weitere Funktion (`eval_command()` in `interpre.c`) aufgerufen, die zuerst einen Bezeichner aus der entsprechenden Struktur liest (`name_fetch()` in `utility0.c`) und diesen Namen anschließend in den beiden Symboltabellen (siehe Unterabschnitt C.2.3) sucht (`find_h_unit()` in `interpre.c`). Ist das gefundene Symbolelement weder ein ausführbares Kommando noch ein selbst definiertes Makro, so wird ein Fehler zurückgegeben und die Auswertung beendet. Handelt es sich um ein vom Anwender definiertes Makro, dann wird der Makroersatztext dem Interpreter zur Auswertung übergeben. Liegt ein ProSiNN–Kommando vor, so wird die Adresse der dazugehörigen Funktion, die im Symbolelement gespeichert ist, aufgerufen. Diese Routine (die sich in einem Quellfile, dessen Dateiname mit `'funct_0'` beginnt, befindet) bearbeitet dann individuell den Rest des Kommandos. Beim Einlesen der weiteren Parameter des ProSiNN–Befehls wird die syntaktische Struktur überprüft und gegebenfalls ein Fehler zurückgegeben. Tritt beim Verarbeiten kein Fehler auf, so wird die mit dem ProSiNN–Kommando verbundene Sematik im System realisiert oder es werden weitere Hilfsfunktionen zu dessen Durchführung aufgerufen.

C.2 Die zentralen Datenstrukturen

In diesem Abschnitt werden die wichtigsten Datenstrukturen vorgestellt. Ihre Typdefinitionen befinden sich in der Datei `net.h`, die von allen Modulen importiert wird (mit Ausnahme der Quelldateien, in denen das Grafikwidget definiert wird: `GraficP.h`, `Grafic.h` und `Grafic.c`).

In den folgenden Unterabschnitten werden die Strukturen zur Kommando– und Programmauswertung, von Neuronalen Netzen, Symboltabellen (und deren Elemente) und dem ProSiNN–Kontrollblock vorgestellt.

C.2.1 Die Datenstruktur zur Auswertung von ProSiNN–Kommandos

Zur Auswertung von Interpreterbefehlen und ProSiNN–Programmen dient die folgende Datenstruktur:

```
typedef struct C_STRING
 { char *string;
   unsigned int diff;
   unsigned int line;
 } C_STRING;
```

Dabei werden die einzelnen Komponenten wie folgt verwendet:

- `string` ist ein Zeiger auf ein Array von Zeichen.

  In diesem Feld steht der auszuwertende Befehl oder ein Programm als Folge von ASCII–Zeichen. Wird ein Programm geladen, so werden alle überflüssigen Blanks, Tabulatoren und Kommentare entfernt. Die Zeilenendezeichen ('\n') bleiben zur Bestimmung der Zeilennummer im Fall eines Fehlers erhalten.

- `diff` repräsentiert das aktuelle auszuwertende Zeichen.

  Diese Differenz zur Startadresse des Arrays wird im Verlauf der Auswertung manipuliert.

- `line` repräsentiert die aktuelle zu bearbeitende Zeile.

  Wird ein Zeilenendezeichen gefunden, so wird diese Komponente um eins erhöht.

C.2.2 Die Datenstruktur eines Neuronalen Netzes

Im folgenden soll die Datenstruktur, die ein im Speicher enthaltenes Neuronales Netz aufweist, beschrieben werden. Dazu werden zuerst die einzelnen Bestandteile eines Netzes vorgestellt.

- Ein Neuron wird durch den Typ `UNIT` festgelegt:

```
typedef struct UNIT
 { double o_pj;
   double theta_akt;
   double delta_theta_akt;
   double theta_old;
   double deltatheta_old;
   double delta_pj;
   SYMBOL_POINTER *p_sym;
 } UNIT;
```

  - `o_pj` enthält den Ausgabewert (Aktivierungswert) o_{pj} des Neurons, nachdem ein Muster durch das Netz propagiert wurde.
  - `theta_akt` ist der aktuelle Schwellenwert θ_j des Neurons.
  - `delta_theta_akt` wird zum Aufbau des Änderungswertes $\Delta\theta_j$ der Schwelle in dem aktuellen Trainingszyklus verwendet.
  - `theta_old` ist der Schwellenwert, der vor einer Änderung gültig war.
  - `deltatheta_old` ist der Änderungswert $\Delta\theta_j$ der Schwelle, der vor einer Änderung gültig war.
  - `delta_pj` ist das Fehlersignal δ_{pj} des Neurons.
  - `p_sym` ist ein Zeiger auf ein Array von Symbolelementadressen (siehe unten). Hier sind all die Symbolelemente enthalten, die eine Neuronenvariable definieren. Existiert keine Variable zu diesem Neuron, so ist `p_sym` gleich `NULL`; ansonsten wird das Array mit dem `NULL`–Zeiger abgeschlossen.

- Ein Gewicht wird durch den Typ `WEIGHT` festgelegt:

```
typedef struct WEIGHT
 { double w_akt;
   double delta_w_akt;
   double w_old;
   double deltaw_old;
   SYMBOL_POINTER *p_sym;
 } WEIGHT;
```

  - `w_akt` enthält den aktuellen Gewichtswert w_{ij} einer Verbindung zwischen zwei Neuronen.
  - `delta_w_akt` wird zum Aufbau des Änderungswertes Δw_{ij} des Gewichtes in dem aktuellen Trainingszyklus verwendet.
  - `w_old` ist der Gewichtswert, der vor einer Änderung gültig war.
  - `deltaw_old` ist der Änderungswert Δw_{ij} des Gewichtes, der vor einer Änderung gültig war.

  - `p_sym` ist ein Zeiger auf ein Array von Symbolelementadressen (siehe unten). Hier sind all die Symbolelemente enthalten, die eine Gewichtsvariable definieren. Existiert keine Variable zu diesem Gewicht, so ist `p_sym` gleich `NULL`; ansonsten wird das Array mit dem `NULL`–Zeiger abgeschlossen.

- Eine Netzebene wird durch den Typ `LAYER` festgelegt:

```
typedef struct LAYER
 { UNIT *p_unit;
   WEIGHT *p_weight;
   unsigned int no_units;
   SYMBOL_POINTER *p_sym;
 } LAYER;
```

  - `p_unit` ist der Zeiger auf ein Array von Elementen des Typs `UNIT`. Hier sind nacheinander alle Neuronen dieser Ebene enthalten. Handelt es sich um die Struktur, die die Eingabeschicht beschreibt, so hat der Zeiger `p_sym` den Wert `NULL`.
  - `p_weight` ist der Zeiger auf ein Array von Elementen des Typs `WEIGHT`. Hier sind alle Gewichte enthalten, die von dieser Netzebene zur nächsten verlaufen (dabei ist die Reihenfolge wie folgt vorgegeben: Zuerst kommen alle Gewichte, die vom 1. Neuron der aktuellen Schicht zu allen Neuronen der folgenden Ebene verlaufen; danach kommen alle Gewichte, die vom 2. Neuron der aktuellen Schicht zu allen Neuronen der folgenden Ebene verlaufen; usw.). Handelt es sich um die letzte Netzschicht, so hat der Zeiger `p_sym` den Wert `NULL`.
  - `no_units` beinhaltet die Anzahl der Neuronen in dieser Netzebene.
  - `p_sym` ist ein Zeiger auf ein Array von Symbolelementadressen (siehe unten). Hier sind all die Symbolelemente enthalten, die eine Netzebenenvariable definieren. Existiert keine Variable zu dieser Netzebene, so ist `p_sym` gleich `NULL`; ansonsten wird das Array mit dem `NULL`–Zeiger abgeschlossen.

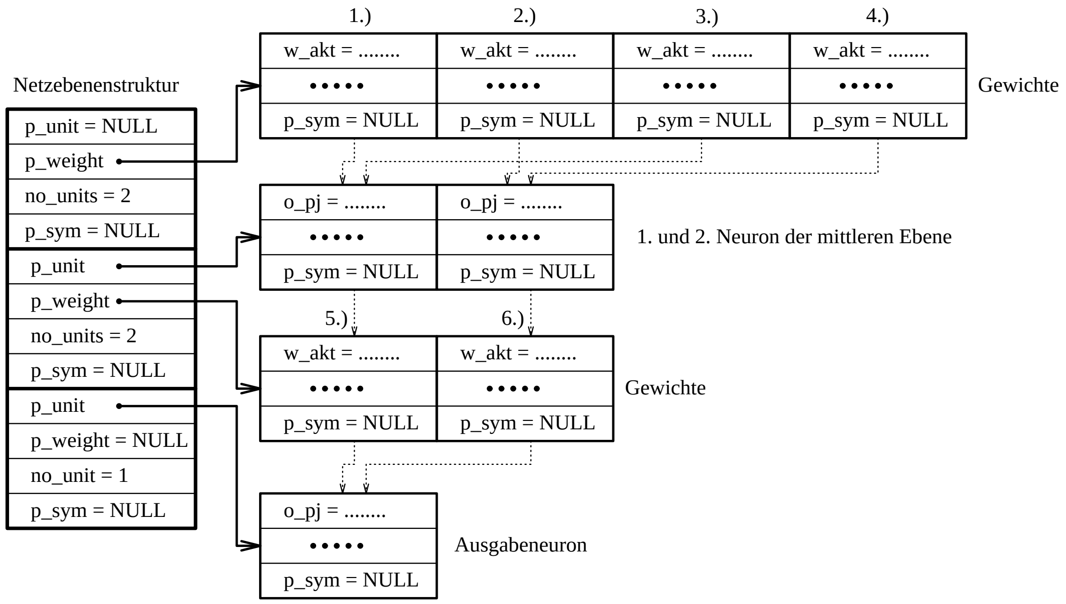
Ein Neuronales Netz wird durch ein Array von Elementen des Typs `LAYER` festgelegt, dessen einzelnen Komponenten mit *gültigen* Werten versehen sind. Um auf eine solche Struktur zugreifen zu können, müssen nur die Adresse des ersten `LAYER`–Elementes und die Anzahl der vorhandenen Netzebenen bekannt sein.

In Bild C.1 ist die Datenstruktur eines dreischichtigen Neuronalen Netzes angegeben.

Hier handelt es sich um ein Netz, das zwei Eingabeneuronen, zwei Zellen in der mittleren Ebene und ein Ausgabeneuron enthält.

- Die Strukturkomponenten `p_sym` sind alle auf `NULL` gesetzt. D.h. es existieren keine Variablen, die auf diese Struktur verweisen.

- Die Neuronen und Gewichtselemente sind nur auszugsweise dargestellt, da deren Inhalt nicht zur Erläuterung der Struktur benötigt wird. Die Netzebenenstruktur ist dagegen vollständig belegt.



1.) Gewicht vom 1. Eingabeneuron zum 1. Neuron der mittleren Schicht
2.) Gewicht vom 1. Eingabeneuron zum 2. Neuron der mittleren Schicht
3.) Gewicht vom 2. Eingabeneuron zum 1. Neuron der mittleren Schicht
4.) Gewicht vom 2. Eingabeneuron zum 2. Neuron der mittleren Schicht
5.) Gewicht vom 1. Neuron der mittleren Ebene zum Ausgabeneuron
6.) Gewicht vom 2. Neuron der mittleren Ebene zum Ausgabeneuron

Abbildung C.1: Die Datenstruktur eines dreischichtigen Netzes

- Die *dicken* Pfeile repräsentieren einen Zeiger auf das erste Element der Struktur.
- Die gestrichelten Pfeile deuten — nur zum besseren Verständnis — die Verbindungsstruktur der Gewichte an.
- Die nummerierten Angaben ('1.)', '2.)', ... '6.)') stellen jeweils ein Gewicht dar, dessen Quell– und Zielneuron im unteren Teil der Abbildung C.1 angegeben ist.

Für ein Neuronales Netz, das mit der Festkommaarithmetik trainiert werden soll, werden einige spezielle Datenstrukturen benötigt. Hierbei handelt es sich um angepaßte Konstruktionen für die Typen `PATTERN`, `UNIT`, `WEIGHT`, `LAYER` und `NET_MANAGER` (siehe Abschnitt C.2.4). Nähere Informationen findet man bei den entsprechenden Typdefinitionen in der Datei `net.h`.

C.2.3 Die Datenstruktur einer Symboltabelle und deren Elemente

Zur Realisierung von (vordefinierten) Konstanten, Variablen und Kommandos werden Elemente benötigt, die in einer Symboltabelle verwaltet werden (siehe u.a. [AHO88]). Ein Symbolelement ist wie folgt definiert:

```
typedef struct HASH_UNIT
 { char *name;
   char type;
   struct HASH_UNIT *next;
   union
   { int (*command)();
     P_TYPE p_val;
     L_TYPE l_val;
     U_TYPE u_val;
     W_TYPE w_val;
     A_I_TYPE a_i_val;
     A_D_TYPE a_d_val;
     I_F_TYPE i_f_val;
     D_F_TYPE d_f_val;
     C_STRING *com_string;
     double d_val;
     double *p_double;
     int i_val;
     int *p_int;
     unsigned int *p_u_int;
     char *p_string;
   } VAL;
 } HASH_UNIT, *SYMBOL_POINTER;
```

Die einzelnen Strukturkomponenten werden nun beschrieben.

- `name` ist ein Zeiger auf eine Zeichenkette.

  In dieser Zeichenfolge wird der Name des Symbolelementes (d.h. der Konstanten, Variablen u.ä.) gespeichert.

- `type` definiert den Typ des Symbolelementes.

  Hier werden Zeichen zur Spezifizierung des Typs verwendet. Alle existierenden Typen sind in `net.h` definiert; der Wert `'f'` steht z.B. für eine Integervariable.

- `next` ist ein Zeiger auf ein Symbolelement.

  In der Symboltabelle verweist dieser Zeiger auf das nächste Element in der linearen Liste (siehe unten).

- `VAL` realisiert eine variante Struktur. Je nach dem Typspezifikator `type` ist eine Komponente gültig. Die einzelnen Strukturelemente haben dabei die folgenden Bedeutungen:

  – `command` ist ein Zeiger auf eine `C`–Funktion.
    Hier wird die Adresse jeder Funktion, die ein ProSiNN–Kommando bearbeitet, gespeichert. Findet der Interpreter beim Auswerten eines Befehls– oder Programmtextes den Namen eines Symbolelmentes, das ein Kommando darstellt, so wird die Funktion, deren Adresse in `command` steht, aufgerufen. Dort wird dann der Befehl vollständig ausgewertet.

– Die Alternative `p_val` steht für eine Mustervariable. Ihr Typ ist wie folgt definiert:

```
typedef struct P_TYPE
 { PATTERN *p_pattern;
   char id;
 } P_TYPE;
```

  * `p_pattern` ist ein Zeiger auf den Kontrollblock eines Musters. Die Typdefinition eines solchen Blocks hat das Format:

    ```
    typedef struct PATTERN
     { double *p_pattern;
       struct HASH_UNIT *p_sym;
     } PATTERN;
    ```

    · `p_pattern` zeigt auf ein vollständiges Muster. Dort befinden sich zuerst alle Netzeingabewerte und anschließend — wenn es sich um ein Trainingsmuster handelt — folgen die Netzausgabewerte.

    · `p_sym` ist ein Zeiger, der die Rückwärtsverkettung auf das Symbolelement realisiert. Dies wird für das Löschen eines Musters benötigt, da dann auch das entsprechende Symbolelement entfernt werden muß. Für jedes Muster ist nur ein Bezeichner, der darauf verweist, erlaubt. Wenn kein Name für das Muster existiert, ist hier `NULL` gespeichert; ansonsten befindet sich in dieser Strukturkomponente die Adresse des dazugehörigen Symbolelementes.

  * `id` wird zur Unterscheidung von Trainings– und Testmustern verwendet. Die hier erlaubten Werte sind in `net.h` definiert.

– Die Alternative `l_val` wird bei einer Netzebenenvariablen benutzt. Ihr Typ ist folgendermaßen definiert:

```
typedef struct L_TYPE
 { struct LAYER *p_layer;
   unsigned int layer_nr;
 } L_TYPE;
```

  * `p_layer` ist ein Zeiger auf die Netzebene, die diese Variable spezifiziert.
  * `layer_nr` gibt die Nummer der Netzebene an.

– Die Alternative `u_val` wird bei einer Neuronenvariablen benutzt. Ihr Typ ist wie folgt definiert:

```
typedef struct U_TYPE
 { struct UNIT *p_unit;
   unsigned int layer_nr;
   unsigned int unit_nr;
 } U_TYPE;
```

  * `p_unit` ist ein Zeiger auf das Neuron, das diese Variable spezifiziert.
  * `layer_nr` gibt die Nummer der zu der Zelle gehörenden Netzebene an.
  * `unit_nr` repräsentiert die relative Nummer des Neurons in dieser Netzebene.

– Die Alternative w_val wird bei einer Gewichtsvariablen benutzt. Ihr Typ ist folgendermaßen definiert:

```
typedef struct W_TYPE
 { struct WEIGHT *p_weight;
   unsigned int layer_nr;
   unsigned int from_unit_nr;
   unsigned int to_unit_nr;
 } W_TYPE;
```

  * p_weight ist ein Zeiger auf das Gewicht, das diese Variable spezifiziert.
  * layer_nr gibt die Nummer der Quellebene des Gewichtes an (hier befindet sich das Neuron, von dem aus das Gewicht startet).
  * from_unit_nr repräsentiert die relative Nummer des Quellneurons in der Startebene.
  * to_unit_nr repräsentiert die relative Nummer des Zielneurons (in der Zielebene).

– Die Alternative a_i_val wird bei einer Integerfeldvariablen benutzt. Ihr Typ ist wie folgt definiert:

```
typedef struct A_I_TYPE
 { unsigned int last_nr;
   int *p_int;
 } A_I_TYPE;
```

  * last_nr ist der Index des letzten gültigen Arrayelementes.
  * p_int ist ein Zeiger auf das Array von Integerelementen (mit last_nr + 1 Elementen).

– Die Alternative a_d_val wird bei einer Doublefeldvariablen benutzt. Ihr Typ ist folgendermaßen definiert:

```
typedef struct A_D_TYPE
 { unsigned int last_nr;
   double *p_double;
 } A_I_TYPE;
```

  * last_nr ist der Index des letzten gültigen Arrayelementes.
  * p_double ist ein Zeiger auf das Array von Doubleelementen (mit last_nr + 1 Elementen).

– Die Alternative i_f_val wird bei einer Integerfunktionsvariablen benutzt. Ihr Typ ist wie folgt definiert:

```
typedef struct I_F_TYPE
 { unsigned int nr_var;
   int *p_to_data;
   int *p_stack;
   int *p_to_funct;
 } I_F_TYPE;
```

  * nr_var enthält die Anzahl der Funktionsargumente. Bei der Kosinusfunktion ist nr_var z.B. 1.

  * `p_to_data` ist der Zeiger auf ein Integerarray, das alle zur Auswertung benötigten Daten beinhaltet.
    Hier befindet sich der Platz für die Funktionsargumente. Anschließend folgen alle Konstanten dieser Funktion. Die Indexe der einzelnen Argumente und Konstanten werden in der Definition der Funktion (siehe `p_to_funct`) zur Referenz der entsprechenden Werte verwendet.
  * `p_to_stack` ist der Zeiger auf ein Integerarray, das als Laufzeitstack zum Auswerten der Funktion benötigt wird.
  * `p_to_funct` ist der Zeiger auf ein Integerarray, das die Funktionsdefinition in Postfixnotation enthält. Das Ende des Arrays wird durch eine spezielle Endzahl (-1) gekennzeichnet.
    Hier werden die Indexe der Funktionsargumente und Konstanten zur Referenz auf die entsprechenden Werte verwendet. Die (arithmetischen) Operatoren sind in der Datei `net.h` als negative Integerzahlen definiert und werden in diesem Array zur Auswertung benutzt.
- Die Alternative `d_f_val` wird bei einer Doublefunktionsvariablen benutzt. Ihr Typ ist folgendermaßen definiert:

```
typedef struct D_F_TYPE
 { unsigned int nr_var;
   double *p_to_data;
   double *p_stack;
   int *p_to_funct;
 } D_F_TYPE;
```

  * `nr_var` enthält die Anzahl der Funktionsargumente. Bei der Kosinusfunktion ist `nr_var` z.B. 1.
  * `p_to_data` ist der Zeiger auf ein Doublearray, das alle zur Auswertung benötigten Daten beinhaltet.
    Hier befindet sich der Platz für die Funktionsargumente. Anschließend folgen alle Konstanten dieser Funktion. Die Indexe der einzelnen Argumente und Konstanten werden in der Definition der Funktion (siehe `p_to_funct`) zur Referenz der entsprechenden Werte verwendet.
  * `p_to_stack` ist der Zeiger auf ein Doublearray, das als Laufzeitstack zum Auswerten der Funktion benötigt wird.
  * `p_to_funct` ist der Zeiger auf ein Integerarray, das die Funktionsdefinition in Postfixnotation enthält. Das Ende des Arrays wird durch eine spezielle Endzahl (-1) gekennzeichnet.
    Hier werden die Indexe der Funktionsargumente und Konstanten zur Referenz auf die entsprechenden Werte verwendet. Die (arithmetischen) Operatoren sind in der Datei `net.h` als negative Integerzahlen definiert und werden in diesem Array zur Auswertung benutzt.
- `com_string` ist der Zeiger auf eine Struktur vom Typ `C_STRING`, die ein durch den *alias*–Befehl erzeugten Makrotext enthält. Wird ein solches Symbolelement gefunden, dann wird der vorliegende Ersatztext wie eine *normale* Befehlsfolge mit dem Interpreter ausgewertet.
- `d_val` wird als Speicher verwendet, wenn der Anwender eine Doublevariable definiert.

- `p_double` ist ein Zeiger auf einen Doublewert. Er wird nur für vordefinierte Variablen benutzt. Bei der ProSiNN–Variablen **eta** wird hier z.B. die Adresse der dazugehörigen Strukturkomponente des ProSiNN–Kontrollblocks (siehe unten) gespeichert.
- `i_val` wird als Speicher verwendet, wenn der Anwender eine Integervariable definiert.
- `p_int` ist ein Zeiger auf einen Integerwert. Er wird nur für vordefinierte Variablen benutzt. Bei der ProSiNN–Variablen **nr_of_layers** wird hier z.B. die Adresse der dazugehörigen Strukturkomponente des ProSiNN–Kontrollblocks (siehe unten) gespeichert.
- `p_u_int` ist ein Zeiger auf einen positiven Integerwert *(unsigned)*. Er wird nur für vordefinierte Variablen benutzt. Bei der von ProSiNN zur Verfügung gestellten Variablen **act_iterations** wird hier z.B. die Adresse der dazugehörigen Strukturkomponente des ProSiNN–Kontrollblocks (siehe unten) gespeichert.
- `p_string` ist ein Zeiger auf eine Array von Zeichen. Diese Alternative wird bei Stringvariablen verwendet.

Eine Symboltabelle ist folgendermaßen definiert (ProSiNN verwaltet jeweils eine Tabelle für alle vordefinierten Bezeichner und Schlüsselwörter und eine für die vom Benutzer generierten Variablen):

```
typedef struct HASH
 { unsigned int no_units;
   SYMBOL_POINTER *p_hash;
 } HASH;
```

- `no_units` definiert die Größe der Symboltabelle.

  ProSiNN verwendet für beide Tabellen eine einheitliche Größe. Sie ist in der Datei `net.h` definiert und so gewählt worden, daß alle vordefinierten Schlüsselwörter und Bezeichner auf verschiedenen Tabellenplätzen liegen (der Zugriff erfolgt über eine *Hash*–Funktion).

- `p_hash` ist der Zeiger auf ein Array von `no_units` Elementen, die jeweils die Adresse eines Symbolelementes enthalten.

  Ist ein Tabellenplatz nicht belegt, so enthält `p_hash` den `NULL`–Zeiger. Ansonsten ist hier der Zeiger auf das erste Symbolelement gespeichert, das durch die *Hash*–Funktion auf diesen Index abgebildet wurde. Jedes weitere Element, das auf diesen Tabellenplatz gehört, befindet sich in einer verketteten Liste von Symbolelementen.

C.2.4 Die Datenstruktur des ProSiNN–Kontrollblocks

Alle vordefinierten und vom Benutzer generierten Variablen sind über den Kontrollblock, der das ganze System steuert, zugreifbar. Diese Datenstruktur ist die wichtigste interne Konstruktion und existiert nur einmal im gesamten System. Ihre Typdefinition lautet folgendermaßen:

```
typedef struct NET_MANAGER
  { SYMBOL_POINTER p_act;
    SYMBOL_POINTER p_der_act;
    OUTPUT *p_to_out;
    Widget *p_to_win;
    int os_error;
    unsigned int no_of_layers;
    LAYER *p_layer;
    unsigned int nr_in_units;
    unsigned int nr_out_units;
    unsigned int no_pattern;
    PATTERN *p_pattern;
    double *p_akt_pattern;
    double pat_min;
    double pat_max;
    unsigned int no_of_protocols;
    PROTOCOL *p_prot;
    unsigned int no_of_grafic_protocols;
    GRAFIC_PROTOCOL *p_gprot;
    unsigned int min_prot_dist;
    unsigned int max_iterations;
    int error_mode;
    double total_error;
    double alpha;
    double eta;
    double phi;
    double beta;
    double percent;
    double w_min;
    double w_max;
    double min_theta;
    double max_theta;
    unsigned int akt_iterations;
    unsigned int iterations;
    double akt_error;
    double old_error;
    double akt_alpha;
    double akt_eta;
    unsigned int precision;
    int floatpattern;
    HASH *user_hash;
    HASH *sys_hash;
 } NET_MANAGER;
```

Die einzelnen Komponenten werden im folgenden beschrieben.

- `p_act` ist nur dann ungleich `NULL`, wenn der Anwender eine neue Aktivierungsfunktion definiert hat.

  In diesem Fall befindet sich hier die Adresse des Symbolelementes, das die Aktivierungsfunktion in Postfixnotation enthält.

- `p_der_act` ist nur dann ungleich `NULL`, wenn der Anwender eine neue Aktivierungsfunktion definiert hat.

  In diesem Fall befindet sich hier die Adresse des Symbolelementes, das die Ableitung der Aktivierungsfunktion in Postfixnotation enthält.

- `p_to_out` ist ein Zeiger auf ein Array von Filepointern.

  ```
  typedef FILE *OUTPUT;
  ```

  Diese Feldelemente enthalten die Filepointer zu den erlaubten Dateihandles. Ist ein Filepointer nicht belegt, so steht dort ein `NULL`–Zeiger; ansonsten befindet sich in diesem Element der Filepointer einer zum Schreiben geöffneten Datei. Jeder Index in dieser Struktur korrespondiert mit dem vom System vergebenen Dateihandle.

  Die maximale Anzahl der zum Schreiben geöffneten Files kann in der Datei `net.h` verändert werden (siehe Unterabschnitt C.3.3).

- `p_to_win` ist ein Zeiger auf ein Array von Widgets (näheres dazu findet man in [JON89], [MAN90], [MCC91], [YOU89] und [PET91]).

  Diese Feldelemente enthalten die Widgets (die für vom Benutzer geöffnete Grafikfenster stehen) zu den erlaubten Grafikhandles. Ist ein Widget nicht belegt (d.h. das Fenster existiert nicht), so steht dort ein `NULL`–Zeiger; ansonsten befindet sich in diesem Element das Widget eines geöffneten Grafikfensters. Jeder Index in dieser Struktur korrespondiert mit dem vom System vergebenen Grafikhandle.

  Die maximale Anzahl der erzeugten Grafikfenster kann in der Datei `net.h` geändert werden (siehe Unterabbschnitt C.3.3).

- `os_error` dient zum Zwischenspeichern von Betriebssystemfehlern. Diese Komponente wird benötigt, da die Funktionen, die die Befehle auswerten, nur die ProSiNN–Fehlernummer zurückliefern.

- `no_of_layers` enthält die Anzahl der Netzebenen. Existiert kein Neuronales Netz im System, so ist hier 0 gespeichert.

- `p_layer` zeigt auf die erste Ebene (d.h. auf das erste `LAYER`–Element) eines im System enthaltenen Netzes. Existiert kein Neuronales Netz im System, so ist hier der `NULL`–Zeiger gespeichert.

- `nr_in_units` enthält die Anzahl der Netzeingabeneuronen. Existiert kein Neuronales Netz im System, so ist hier 0 gespeichert.

- `nr_out_units` enthält die Anzahl der Netzausgabeneuronen. Existiert kein Neuronales Netz im System, so ist hier 0 gespeichert.

- `no_pattern` enthält die Anzahl der Trainingsmuster im System. Existieren keine Trainingsmuster, so ist hier 0 gespeichert.
- `p_pattern` ist ein Zeiger auf den ersten Musterkontrollblock der im System geladenen Trainingsmuster.

  Es handelt sich hier um den Zeiger auf ein Array, dessen Elemente jeweils einen Kontrollblock vom Typ `PATTERN` repräsentieren (siehe oben). Existieren keine Trainingsmuster im System, so ist hier der `NULL`–Zeiger gespeichert.
- `p_akt_pattern` ist der Zeiger auf ein Doublearray, das das nächste zu trainierende Muster spezifiziert. Es handelt sich hier um denselben Wert, den die Komponente `p_pattern` des entsprechenden Musterkontrollblocks vom Typ `PATTERN` beinhaltet. Existieren keine Trainingsmuster im System, so ist hier der `NULL`–Zeiger gespeichert.
- `pat_min` und `pat_max` beinhalten die Intervallangabe, die zum Einlesen von symbolischen Mustern benötigt wird.
- `no_of_protocols` enthält die Anzahl der im System definierten Textprotokolle.
- `p_prot` ist der Zeiger auf ein Array von Protokollelementen. Ein Protokollelement beschreibt ein vom Anwender definiertes Textprotokoll und ist wie folgt definiert:

  ```
  typedef struct PROTOCOL
   { FILE *fp;
     char *prot_command;
     PROT_UNIT *p_prot;
     unsigned int number;
     unsigned int prot_dist;
   } PROTOCOL;
  ```

  Die einzelnen Komponenten werden nun beschrieben:

  - `fp` ist der Filepointer, der das Ausgabemedium des Textprotokolls spezifiziert.
  - `prot_command` ist der Zeiger auf ein Array von Zeichen, das den vollständigen Originaltext des ProSiNN–Kommandos *protocol* enthält (nach ':'). D.h. hier sind alle Protokollparameter im ASCII–Format gespeichert. Dies wird benötigt, um vor Beginn des Trainings feststellen zu können, ob noch alle Parameter gültig sind und um die Adressen der Protokollparameter in `p_prot` zu aktualisieren. Bei der Protokollierung von Arrayelementen und Funktionen wird jedesmal der Originaltext ausgewertet (siehe unten).
  - `p_prot` ist der Zeiger auf ein Array der zu protokollierenden Elemente. Jeder dieser Parameter ist dabei wie folgt definiert:

    ```
    typedef struct PROT_UNIT
     { char type;
       union
       { double *p_double,
                        d;
         int *p_int,
    ```

```
              i;
    char *p_string;
  } VAL;
} PROT_UNIT;
```

Die einzelnen Komponenten werden wie nachfolgend beschrieben verwendet:

* `type` definiert den Typ des auszugebenden Elementes (genau wie bei einem Symbolelement).
* Die folgenden Alternativen werden je nach Typ gesetzt:
  · `p_double` ist der Zeiger auf eine Doublevariable. Er wird für vordefinierte und vom Benutzer erzeugte Doublevariablen verwendet.
  · `d` ist der Speicher für einen Double. Hier wird der Wert einer Doublekonstanten gespeichert.
  · `p_int` ist der Zeiger auf eine Integervariable. Er wird für vordefinierte und vom Anwender erzeugte Integervariablen verwendet.
  · `i` ist der Speicher für einen Integer. Hier wird der Wert einer Integerkonstanten gespeichert.
  · `p_string` ist der Zeiger auf ein Array von Zeichen.
  Wenn `type` eine Konstante oder eine Variable vom Typ *string* definiert, dann ist hier die Adresse des Strings gespeichert.
  Bezeichnet `type` ein Integer– oder Doublearray, so wird hier die Startadresse des Parameters gespeichert. Diese Adresse zeigt auf den Beginn des Arrays im Originaltext, auf den `prot_command` verweist (siehe oben).
  Spezifiziert `type` eine Integer– oder Doublefunktion, dann wird hier die Startadresse des Parameters gespeichert. Diese Adresse zeigt auf den Beginn der Funktion im Originaltext, auf den der Zeiger `prot_command` verweist (siehe oben).

– `number` enthält die Anzahl der zu protokollierenden Elemente (d.h. der Arraygröße von `p_prot`).

– `prot_dist` enthält die Zyklendifferenz, nach der jeweils ein Protokollzeile auszugeben ist.

• `no_of_grafic_protocols` enthält die Anzahl der im System definierten Grafikprotokolle.

• `p_gprot` ist der Zeiger auf ein Array von Protokollelementen. Ein Protokollelement beschreibt ein vom Anwender definiertes Grafikprotokoll und ist wie folgt definiert:

```
typedef struct GRAFIC_PROTOCOL
 { Widget grafic;
   char *prot_command;
   G_PROT_UNIT *p_prot;
   unsigned int number;
   unsigned int prot_dist;
 } GRAFIC_PROTOCOL;
```

Die einzelnen Komponenten werden nun beschrieben:

– `grafic` ist das Widget, das das Grafikfenster spezifiziert, in das protokolliert werden soll.

– `prot_command` ist der Zeiger auf ein Array von Zeichen, das den vollständigen Originaltext des ProSiNN–Kommandos *grafic_protocol* enthält (nach ':'). D.h. hier sind alle Protokollparameter im ASCII–Format gespeichert. Dies wird benötigt, um vor Beginn des Trainings feststellen zu können, ob noch alle Parameter gültig sind und um die Adressen der Protokollparameter in `p_prot` zu aktualisieren. Bei der Protokollierung von Arrayelementen und Funktionen wird jedesmal der Originaltext ausgewertet (siehe unten).

– `p_prot` ist der Zeiger auf ein Array der zu protokollierenden Elemente. Jeder dieser Parameter ist dabei wie folgt definiert:

```
typedef struct G_PROT_UNIT
 { char type;
   int is_dotted;
   DOT_STRUCT dot;
   union
   { double *p_double,
                    d;
     int *p_int,
             i;
     char *p_string;
   } VAL;
 } G_PROT_UNIT;
```

Die einzelnen Komponenten werden wie nachfolgend beschrieben verwendet:

  * `type` definiert den Typ des auszugebenden Elementes (genau wie bei einem Symbolelement).
  * `is_dotted` gibt an, ob die grafische Protokollierung mit einer Strichlierung versehen werden soll. Ist der Wert ungleich 0, dann wird eine Strichlierung benutzt.
  * `dot` gibt die Strichlierungsstruktur an (die Typdefinition befindet sich in der Datei `Grafic.h`).
  * Die folgenden Alternativen werden je nach Typ in der Variante `VAL` gesetzt:
    · `p_double` ist der Zeiger auf eine Doublevariable. Er wird für vordefinierte und vom Benutzer erzeugte Doublevariablen verwendet.
    · `d` ist der Speicher für einen Double. Hier wird der Wert einer Doublekonstanten gespeichert.
    · `p_int` ist der Zeiger auf eine Integervariable. Er wird für vordefinierte und vom Benutzer erzeugte Integervariablen verwendet.
    · `i` ist der Speicher für einen Integer. Hier wird der Wert einer Integerkonstanten gespeichert.
    · `p_string` ist der Zeiger auf ein Array von Zeichen.

Bezeichnet `type` ein Integer– oder Doublearray, so wird hier die Startadresse des Parameters gespeichert. Diese Adresse zeigt auf den Beginn des Arrays im Originaltext, auf den `prot_command` verweist (siehe oben).
Spezifiziert `type` eine Integer– oder Doublefunktion, dann wird hier die Startadresse des Parameters gespeichert. Diese Adresse zeigt auf den Beginn der Funktion im Originaltext, auf den `prot_command` verweist (siehe oben).

  - `number` enthält die Anzahl der zu protokollierenden Elemente (d.h. der Arraygröße von `p_prot`).
  - `prot_dist` enthält die Zyklendifferenz, nach der jeweils ein Protokollelement in die Grafik auszugeben ist. Dieser Wert wird von ProSiNN so berechnet, das in horizontaler Richtung jeder vorhandene Grafikpunkt verwendet wird.

- `min_prot_dist` beinhaltet die minimale Zyklendifferenz aller Protokolle, nach denen evtl. jeweils protokolliert werden muß. Dieser Wert wird durch Berechnung des größten gemeinsamen Teilers aller Protokollzyklendifferenzen erhalten.
- `max_iterations` wird bei Beginn jedes Netztrainings die maximale Anzahl der auszuführenden Trainingszyklen zugewiesen.
- `error_mode` enthält den Fehlermaßmodus.
- `total_error` enthält die untere Fehlermaßgrenze, bei der das Training eines Neuronalen Netzes beendet werden soll.
- `alpha` enthält das vorgegebene Lernmoment.
- `eta` enthält die vorgegebene Lernrate.
- `phi` enthält den Erfolgslernratenfaktor.
- `beta` enthält den Mißerfolgslernratenfaktor.
- `percent` enthält den Mißerfolgsfaktor.
- `w_min` und `w_max` beinhalten die Intervallangabe, die beim Initialisieren der Gewichte verwendet wird.
- `min_theta` und `max_theta` enthalten die Intervallangabe, die beim Initialisieren der Schwellenwerte verwendet wird.
- `akt_iterations` enthält die Anzahl der beim letzten (oder dem gerade aktuellen) Trainingsbefehl ausgeführten Lernzyklen.
- `iterations` enthält die Gesamtanzahl der bisher ausgeführten Lernzyklen.
- `act_error` enthält das Fehlermaß, das nach dem letzten Trainingszyklus vorlag.
- `old_error` enthält das Fehlermaß, das vor dem letzten Trainingszyklus vorlag.
- `akt_alpha` enthält das aktuelle Lernmoment.

- `akt_eta` enthält die aktuelle Lernrate.
- `precision` enthält die Anzahl der Nachkommastellen, die bei der Ausgabe einer Fließkommazahl verwendet werden.
- `floatpattern` gibt an, ob beim Einlesen von Mustern die symbolische Schreibweise erwartet wird. Ist der Wert gleich 0, so müssen symbolische Muster benutzt werden.
- `user_hash` ist der Zeiger auf die Benutzersymboltabelle.

  In ihr werden alle vom Anwender erzeugten Variablen gespeichert.
- `sys_hash` ist der Zeiger auf die Systemsymboltabelle.

  Sie enthält alle vordefinierten Bezeichner und Schlüsselwörter. Diese Symboltabelle wird nicht verändert.

C.3 Die Systeminstallation

In diesem Abschnitt werden die für die Systeminstallation wichtigen Informationen aufgeführt. Dazu werden die Rechnerumgebung, die Quelldateien, evtl. notwendige Änderungen und die Erzeugung von ProSiNN beschrieben.

C.3.1 Die der Entwicklung zugrundeliegende Rechnerumgebung

- Das ProSiNN–Programmsystem Version 2.0 wurde auf `Sun`–Workstations mit `SPARC`–Architektur implementiert und getestet.
- Das zugrundeliegende Betriebssystem `SunOS Release 4.1.x` stammt aus der Familie der `BSD-Unix`–Implementierungen; es realisiert die Funktionalität des `BSD 4.3`.
- Notwendig für die Implementierung von ProSiNN ist die grafische Benutzeroberfläche `X`. Implementiert und getestet wurde das System auf der vom `MIT X Consortium`[1] vertriebenen Version `X11 Release 5`. Von dieser Implementierung werden u.a. `Xlib` (es realisiert das `C`–Interface zu `X`), das `X Toolkit` (hier werden `C`–Funktionen zum Generieren und Benutzen von *Widgets* zur Verfügung gestellt) und das `Athena Widget Set` (das komplette *Widgets* zur Realisierung einer Benutzeroberfläche anbietet) benötigt.

C.3.2 Die Quelldateien des Systems

In diesem Abschnitt werden die Quelldateien des Systems mit einer kurzen Beschreibung vorgestellt:

- **Makefile**:

  Die Kontrolldatei, die das Systemkommando *make* zur Generierung des Programmsystems benötigt (das vollständige File ist in Unterabschnitt C.3.3 aufgelistet).

[1] `MIT X Consortium`: Massachusetts Institute of Technology `X Consortium`

- **net.h**:

  Diese *Include*–Datei wird von den anderen Quellfiles importiert. Sie enthält die *include*–Befehle für die benutzten Widgets, die Konstanten und Fehlernummern, die Typdefinitionen der Datenstrukturen und die Deklarationen der implementierten Funktionen.

- **Grafic.h**:

  Definiert die öffentliche *Include*–Datei zur Verwaltung von Grafikfenstern. Hier wird ein neues Widget zur Verfügung gestellt, das grafische Ausgaben in ein Fenster ermöglicht.

- **GraficP.h**:

  Definiert die private *Include*–Datei zur Verwaltung von Grafikfenstern. Hier wird ein neues Widget zur Verfügung gestellt, das grafische Ausgaben in ein Fenster erlaubt.

- **Grafic.c**:

  Enthält die Funktionen, die zur Verwaltung von Grafikfenstern benötigt werden. Hier wird ein neues Widget zur Verfügung gestellt, das grafische Ausgaben in ein Fenster ermöglicht.

- **net.c**:

  Enthält die Funktionen, die das System hochfahren und Eingaben vom Benutzer auswerten.

- **interpre.c**:

  Hier werden die Symboltabellen definiert und initialisiert. Außerdem sind Funktionen zur Interpretierung von ProSiNN–Befehlen vorhanden.

- **menu_1.c**:

  Enthält Funktionen zur Bearbeitung des Menüs *Netz bearbeiten* und zur Erzeugung einer Dateiauswahlbox.

- **menu_2.c**:

  Enthält Funktionen zur Bearbeitung des Menüs *Muster bearbeiten.*

- **menu_3.c**:

  Enthält Funktionen zur Bearbeitung des Menüs *Netz trainieren.*

- **menu_4.c**:

  Enthält Funktionen zur Bearbeitung des Menüs *Systemwerte bearbeiten.*

- **menu_5.c**:

  Enthält Funktionen zur Bearbeitung des Menüs *Benutzerwerte bearbeiten.*

- **menu_6.c**:

  Enthält Funktionen zur Bearbeitung des Menüs *Programm beenden.*

- **utility0.c**:

  Enthält Routinen, die aus der intern vorliegenden Befehlsstruktur Parameter einlesen.

- **utility1.c**:

  Enthält Routinen, die aus der intern vorliegenden Befehlsstruktur Parameter einlesen.

- **utility2.c**:

  Enthält Funktionen, die Ausgaben des Systems durchführen.

- **utility3.c**:

  Enthält Funktionen, die intern verwendete Referenzen updaten oder löschen.

- **utility4.c**:

  Enthält Funktionen, die temporäre Berechnungen für Interpreterbefehle ausführen und Routinen, die im Fehlerfall für einen stabilen Systemzustand sorgen.

- **utility5.c**:

  Enthält die Funktion, die Fehlermeldungen in das Ein/Ausgabefenster schreibt.

- **utility6.c**:

  Enthält Routinen, die vom Anwender definierte Funktionen in ein Grafikfenster ausgeben und die beim Netztraining das grafische Protokollieren durchführen.

- **utility7.c**:

  Enthält Routinen, die Funktionen in Postfixnotation im Speicher ablegen und diese Strukturen vereinfachen oder ableiten.

- **utility8.c**:

  Enthält Funktionen, die das in Abschnitt 2.3 eingeführte modifizierte Backpropagation–Verfahren realisieren.

- **utility9.c**:

  Enthält Funktionen, die das in Abschnitt 2.3 eingeführte modifizierte Backpropagation–Verfahren mit einer Festkommaarithmetik realisieren.

- **funct_01.c**:

  Enthält Funktionen, die die Interpreterbefehle zur Bearbeitung von Variablen zur Verfügung stellen (siehe Kapitel 5).

- **funct_02.c**:

  Enthält Funktionen, die die Interpreterbefehle zur Bearbeitung von Kontrollstrukturen und ProSiNN–Programmen zur Verfügung stellen (siehe Kapitel 6).

- **funct_03.c**:

  Enthält Funktionen, die die Interpreterbefehle zur Ein– und Ausgabeverwaltung zur Verfügung stellen (siehe Kapitel 7).

- **funct_04.c**:

  Enthält Funktionen, die die Interpreterbefehle zum Verwalten von Grafiken zur Verfügung stellen (siehe Kapitel 8).

- **funct_05.c**:

  Enthält Funktionen, die die Interpreterbefehle zur Bearbeitung von Neuronalen Netzen zur Verfügung stellen (siehe Kapitel 9).

- **funct_06.c**:

  Enthält Funktionen, die die Interpreterbefehle zur Bearbeitung von Mustern zur Verfügung stellen (siehe Kapitel 10).

- **funct_07.c**:

  Enthält Funktionen, die die Interpreterbefehle zum Training von Neuronalen Netzen zur Verfügung stellen (siehe Kapitel 11).

- **funct_08.c**:

  Enthält Funktionen, die die Interpreterbefehle zum Verwalten von Trainingsprotokollen zur Verfügung stellen (siehe Kapitel 12).

- **funct_09.c**:

  Enthält Funktionen, die das System betreffende Interpreterbefehle zur Verfügung stellen (siehe Kapitel 13).

C.3.3 Die Generierung des ProSiNN–Systems

In diesem Abschnitt wird beschrieben, wie das ProSiNN–System installiert wird und welche Änderungen dazu evtl. notwendig sind.

- Zur Generierung des Systems müssen die im letzten Unterabschnitt vorgestellten Dateien in *einem* Verzeichnis abgelegt werden.

- Notwendige oder gewünschte Änderungen in den Quelldateien:

  – Für die Auswertung von Ausdrücken oder bei Typumwandlungen, die intern von ProSiNN durchgeführt werden, ist die Realisierung der Datentypen `int` und `double` des `C`–Compilers *bedingt* wichtig.

    Die Implementierung von ProSiNN basiert auf einer Datenbreite von 32 Bits für `int`– und 64 Bits für `double`–Werte. Wenn die Datenbreite größer ist, wird bei der Auswertung von Ausdrücken oder bei einer Typumwandlung mitunter ein Überlauf des Datenbereichs gemeldet, der real nicht aufgetreten ist.

    Um dies zu vermeiden, sollten in der *Include*–Datei `net.h` die folgenden Konstanten angepaßt werden:

    ∗ `#define MAX_INT 2147483647`:
      Diese Konstante beschreibt den maximal erlaubten positiven Integerwert.

  * `#define MIN_INT -2147483648`:
    Diese Konstante beschreibt den minimal erlaubten negativen Integerwert.
  * `#define MAX_HYP 712.0`:
    Diese Konstante beschreibt das maximal erlaubte Argument der hyperbolischen Funktionen.
  * `#define MAX_EXP 709.0`:
    Diese Konstante beschreibt das maximal erlaubte Argument der Exponentialfunktion.
- Für den ProSiNN–Befehl *fast_cycle* wird eine Festkommaarithmetik verwendet. Sie basiert auf einer Größe von 32 Bit des `C`–Datentyps `long` und darauf, daß höherwertige Bytes im Speicher vor niederwertigen liegen (ist dies nicht der Fall, so sollte das Kommando nicht verwendet oder der Programmquelltext muß entsprechend modifiziert werden). Wenn dieser Typ eine andere Datenbreite aufweist, dann sind folgende Konstanten in der Datei `net.h` neu zu definieren (n sei die halbe Datenbreite eines `long`–Wertes; beim vorliegenden System ist n = 16):
  * `#define f_exp_max 681391L`:
    Dies stellt das maximale Argument der Exponentialfunktion in Festkommaarithmetik dar.
    Die Konstante muß auf den nach unten gerundeten Wert des Ausdrucks $\ln(2^{n-1})2^n$ gesetzt werden.
  * `#define f_x_max 0x7FFFFFFFL`:
    Dies stellt die maximale positive Zahl in Festkommaarithmetik dar.
    Die Konstante muß auf den Wert $0x7F \ldots L$ gesetzt werden. Dabei muß an Stelle der Punkte $\frac{n}{2} - 2$ mal ein `F` eingetragen werden.
  * `#define d_max 32767.9999847`:
    Dies stellt die maximale positive Festkommazahl als Gleitkommawert dar.
    Die Konstante muß auf den Wert $2^{n-1} - 2^{-n}$ gesetzt werden.
  * `#define d_neg_max -32767.9999847`:
    Dies stellt die minimale negative Festkommazahl als Gleitkommawert dar.
    Die Konstante muß auf den Wert $-(2^{n-1} - 2^{-n})$ gesetzt werden.
  * `#define f_x_neg_max 0x80000000L`:
    Dies stellt den minimalen negativen Wert in Festkommaarithmetik dar.
    Die Konstante muß auf den Wert $0x80 \ldots L$ gesetzt werden. Dabei muß an Stelle der Punkte $\frac{n}{2} - 2$ mal eine `0` eingetragen werden.
  * `#define d_min 1.52587890625e-5`:
    Dies stellt die minimale positive Festkommazahl als Gleitkommawert dar.
    Die Konstante muß auf den Wert 2^{-n} gesetzt werden.
  * `#define _dividend 0x80000000L`:
    Dies stellt den Dividenden zur Berechnung des reziproken Wertes in Festkommaarithmetik dar.
    Die Konstante muß auf den Wert $0x80 \ldots L$ gesetzt werden. Dabei muß an Stelle der Punkte $\frac{n}{2} - 2$ mal eine `0` eingetragen werden.
  * `#define pre_mask 0xFFFF0000L`:
    Dies stellt eine Maske dar, die zur Ausblendung der n niederwertigen Bits verwendet wird.

Die Konstante muß auf den Wert $0xF\ldots0\ldots L$ gesetzt werden. Dabei muß an Stelle der ersten Punkte $\frac{n}{4}-1$ mal ein `F` und an Stelle der zweiten Punkte $\frac{n}{4}-1$ mal eine `0` eingetragen werden.

* **`#define post_mask 0xFFFFL`:**
  Dies stellt eine Maske dar, die zur Ausblendung der n höherwertigen Bits verwendet wird.
  Die Konstante muß auf den Wert $0xF\ldots L$ gesetzt werden. Dabei muß an Stelle der Punkte $\frac{n}{4}-1$ mal ein `F` eingetragen werden.
* **`#define f_eins 0x10000L`:**
  Dies stellt die Festkommazahl 1 dar.
  Die Konstante muß auf den Wert $0x10\ldots L$ gesetzt werden. Dabei muß an Stelle der Punkte $\frac{n}{4}-1$ mal eine `0` eingetragen werden.
* **`#define bin_point_digits 16`:**
  Dies stellt die Hälfte der Datenbreite eines `long`–Wertes dar.
  Die Konstante muß auf den Wert n gesetzt werden.
* **`#define d_f_factor 65536.0`:**
  Dies stellt den Umrechnungsfaktor in Gleitkommadarstellung dar.
  Die Konstante muß auf den Wert 2^n gesetzt werden.

In der Datei `utility9.c` wird die `C`–Funktion `f_act` zur Berechnung der Aktivierungsfunktion verwendet. Hier werden einige statische Variablen definiert, die zur Berechnung des Aktivierungswertes mittels einer Potenzreihe verwendet werden. Diese Angaben müssen bei einer anderen — als der hier vorgegebenen — Datenbreite eines `long`–Wertes geändert werden. Die nachfolgenden Zeilen zeigen einen Ausschnitt der entsprechenden Variablendefinitionen:

```
static unsigned long int a[] =
       /* Vorfaktoren der Potenzreihe: bei einigen
          Werten wurde auf- oder abgerundet        */
       { 65536L, /* a_0: Festpunktwert von 1/0! */
         65536L, /* a_1: Festpunktwert von 1/1! */
         32768L, /* a_2: Festpunktwert von 1/2! */
         ............
         13L,    /* a_7: Festpunktwert von 1/7! */
         2L      /* a_8: Festpunktwert von 1/8! */
       };

static unsigned long int f_pre_exp[] =
       /* Festpunktwerte: Vorkommastellen von e ^ 0,1,...,10 */
       { 1L,     /* Vorkommawert von e ^ 0 */
         2L,     /* Vorkommawert von e ^ 1 */
         7L,     /* Vorkommawert von e ^ 2 */
         ............
         8103L,  /* Vorkommawert von e ^ 9 */
         22026L  /* Vorkommawert von e ^ 10 */
       };
```

```
static unsigned long int f_post_exp[] =
       /* Festpunktwerte: Nachkommastellen von e ^ 0,1,...,10
          bei einigen wurde auf- oder abgerundet              */
       { 0L,     /* Nachkommawert von e ^ 0 */
         47073L, /* Nachkommawert von e ^ 1 */
         25497L, /* Nachkommawert von e ^ 2 */
         ............
         5500L,  /* Nachkommawert von e ^ 9 */
         30526L  /* Nachkommawert von e ^ 10 */
       };
```

Im folgenden sei n wieder die halbe Datenbreite eines `long`–Wertes; beim vorliegenden System ist n = 16:

- ∗ Die Werte für das Array `a[]` werden wie folgt berechnet:
  Berechne für ein `a_i`: $\frac{1}{i!}2^n$. Das (gerundete) Ergebnis wird als Wert an der entsprechenden Stelle eingetragen.
- ∗ Die Werte für das Array `f_pre_exp[]` werden folgendermaßen berechnet:
  Berechne für jedes `e ^ i` (siehe im Kommentar) e^i. Von dieser Zahl wird der Vorkommawert an der entsprechenden Stelle eingetragen.
- ∗ Die Werte für das Array `f_post_exp[]` werden wie folgt berechnet:
  Berechne für jedes `e ^ i` (siehe im Kommentar) e^i. Von dieser Zahl wird der Vorkommawert abgezogen und der entstandene Wert mit 2^n multipliziert. Diese Zahl wird dann (gerundet) an der entsprechenden Stelle eingetragen.

Wenn die Genauigkeit der Aktivierungsfunktion zu klein sein sollte, müssen weitere Konstanten in den Feldvariablen hinzugefügt werden. Allerdings muß dann auch im dazugehörigen Programmtext die Auswertung der Potenzreihe so modifiziert werden, daß auch die neuen Faktoren der Reihe berücksichtigt werden!

– Will der Benutzer die maximale Anzahl der zum Schreiben geöffneten Dateien verändern, so muß er in der *Include*–Datei `net.h` die folgende Zeile seinen Wünschen anpassen:

```
#define MAX_O_FILES 10
```

– Soll die maximale Anzahl der geöffneten Grafikfenster verändert werden, so muß in der *Include*–Datei `net.h` die nachfolgende Zeile angepaßt werden:

```
#define MAX_O_WINDOWS 10
```

– Wenn die maximale Größe eines Grafikfensters geändert werden soll, müssen die folgenden Zeilen in der *Include*–Datei `net.h` angepaßt werden:

```
#define _G_MAX_WIDTH 2000
#define _G_MAX_HEIGHT 2000
```

Die erste Zeile definiert die maximale Grafikbreite und die zweite die entsprechende Höhe.

- Nun muß evtl. die für das *make*–Systemkommando zuständige Datei `Makefile` korrigiert werden. Der Inhalt dieses Files ist:

```
# Makefile for 'ProSiNN version 2.0':
# Copyright (C) 1993 Uwe Karsten

# where to find X inclues and libs:

XINCPATH = -I/usr/local/X11/include
XLIBPATH = -L/usr/local/X11/lib

# compiler and linker options:

CFLAGS = -O $(XINCPATH)
# CFLAGS = -g $(XINCPATH)

LDFLAGS = $(XLIBPATH)
# LDFLAGS = -g $(XLIBPATH)

# library options:

XLIBS =  -lXaw -lXmu -lXt -lXext -lX11
LIBS = -lm

# macros:

PRG  = prosinn
OBJS = Grafic.o \
net.o interpre.o \
utility0.o utility1.o utility2.o utility3.o \
utility4.o utility5.o utility6.o utility7.o \
utility8.o utility9.o \
menu_1.o menu_2.o menu_3.o menu_4.o  menu_5.o menu_6.o \
funct_01.o funct_02.o funct_03.o funct_04.o \
funct_05.o funct_06.o funct_07.o funct_08.o \
funct_09.o

# suffix rules for making 'prosinn':

$(PRG) : $(OBJS)
$(CC) $(LDFLAGS) -o $(PRG) $(OBJS) $(XLIBS) $(LIBS)

$(OBJS) : net.h
```

– Der Suchpfad für die `X11`–*Include*–Dateien muß evtl. angepaßt werden. Das Makro `XINCPATH` beschreibt diesen Pfad. Hier werden die *Include*–Dateien in dem Verzeichnis `/usr/local/X11/include` gesucht.

– Der Suchpfad für die `X11`–Bibliothek muß evtl. geändert werden. Das Makro `XLIBPATH` beschreibt diesen Pfad. Hier werden die Files in dem Verzeichnis `/usr/local/X11/lib` gesucht.

- Soll nicht der vorgegebene Systemcompiler verwendet werden, so muß das Makro `CC` neu definiert werden. Wird in der ersten Zeile der Text

  ```
  CC = gcc
  ```

  eingefügt, so wird der `GNU-C`–Compiler verwendet (die Installation von ProSiNN wurde aber nicht mit diesem Compiler getestet).
- Will der Anwender einen Fehler im Programm suchen oder den Systemablauf verfolgen, so sollte er die auskommentierte Fassung des Makros `CFLAGS` verwenden. Anschließend kann ein Debugger zur Unterstützung benutzt werden.

- Um das ProSiNN–Programmsystem zu generieren, muß der Anwender den Systembefehl

  ```
  make
  ```

  oder

  ```
  make -f Makefile
  ```

  als Kommando eingeben. Im Anschluß *sollten* die Quelldateien compiliert und zu einem Programm zusammengebunden werden.

  Nun kann das System mit dem Kommando `prosinn` und den entsprechenden Optionen (siehe Kapitel 3) gestartet werden.