ProSiNN — Entwurf und Implementierung einer programmierbaren Simulationsumgebung für Neuronale Netze

Uwe Karsten

26. Oktober 1993

Technische Universität Braunschweig
Institut für Rechnerverbund und Betriebssysteme

Diplomarbeit

ProSiNN — Entwurf und Implementierung einer programmierbaren Simulationsumgebung für Neuronale Netze

von
cand. inform. Uwe Karsten

Aufgabenstellung und Betreuung:
Prof. Dr. R. Kruse und Dipl. Inform. D. Nauck

Braunschweig, den 26. Oktober 1993

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, den 26. Oktober 1993

Kurzfassung

Im Bereich der Künstlichen Intelligenz sind in den letzten Jahren Neuronale Netze in das Interesse der Forschung getreten. In Anlehnung an menschliche Gehirnstrukturen bestehen diese Systeme aus einfachen, parallel arbeitenden Elementen, die über gewichtete Verbindungen Informationen austauschen. Im Rahmen der vorliegenden Arbeit wurde die grafische Simulationsumgebung **ProSiNN** (**Pro**grammierbarer **Si**mulator für Künstliche **N**euronale **N**etze) entwickelt. Auf der Basis des mehrschichtigen Perceptrons können mit diesem Werkzeug Neuronale Netze generiert, trainiert und analysiert werden. Dazu enthält das System eine grafische Oberfläche zur komfortablen Bedienung und eine universelle Programmiersprache, die mit Fähigkeiten zur Bearbeitung von Neuronalen Netzen ausgestattet ist.

Abstract

In recent years scientific research has focussed on neural networks within the field of Artifical Intelligence. According to a simplified model of humain brain structures these systems consist of simple, parallel working elements which exchange information across weighted connections. Within the scope of this thesis the graphical simulation environment **ProSiNN** (**Pro**grammable **Si**mulator for Artifical **N**eural **N**etworks) has been developed. With the help of this toolkit neural networks can be generated, trained and analyzed on the basis of a multilayer perceptron. The system offers a graphical user interface for comfortable use and an universal programming language capable of handling neural nets.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufbau der Arbeit 2
1.2 Darstellung der Syntaxregeln 5

I Neuronale Netze und ProSiNN **9**

2 Künstliche Neuronale Netze **11**
2.1 Einführung in die Theorie Neuronaler Netze 11
2.2 Das mehrschichtige Perceptron 16
2.2.1 Die elementaren Neuronen 16
2.2.2 Die Topologie des Netzes 18
2.2.3 Die Lernregel: Das Backpropagation–Verfahren 20
2.3 Das modifizierte Backpropagation–Verfahren 24
2.4 Training mit dynamischer Neuronenerzeugung 26

3 Einführung in das Programmsystem ProSiNN **31**
3.1 Das ProSiNN–System im Überblick 31
3.2 Neuronale Netze im ProSiNN–System 33
3.3 Start des Systems 35
3.4 Die grafische Oberfläche von ProSiNN 37
3.4.1 Der Kommandozeileninterpreter 37
3.4.2 Das Menüsystem 41

II Die Programmiersprache ProSiNN **47**

4 Datentypen, Operatoren und Ausdrücke **49**
4.1 Bezeichner und Schlüsselwörter 49
4.2 Datentypen und Konstanten 56
4.2.1 Konstanten vom Datentyp *int* 56
4.2.2 Konstanten vom Datentyp *double* 56
4.2.3 Konstanten vom Datentyp *string* 57
4.2.4 Netzebenenkonstanten vom Datentyp *layer* 57
4.2.5 Neuronenkonstanten vom Datentyp *unit* 57
4.2.6 Gewichtskonstanten vom Datentyp *weight* 58
4.2.7 Musterkonstanten vom Datentyp *pattern* 59

4.3 Präfixe für interne Werte . . . 59
4.3.1 Der Netzebenenpräfix: **nr_units** . . . 59
4.3.2 Die Neuronenpräfixe: **theta** , **o_pj** , **delta_pj** , **delta_theta** . . . 60
4.3.3 Die Gewichtspräfixe: **w** , **delta_w** . . . 61
4.3.4 Die Musterpräfixe: **pattern_in** , **pattern_out** . . . 62
4.4 Arithmetische Operatoren und vordefinierte Funktionen . . . 64
4.4.1 Addition und Subtraktion . . . 64
4.4.2 Multiplikation und Division . . . 66
4.4.3 Potenzen . . . 66
4.4.4 Vordefinierte mathematische Funktionen . . . 67
4.5 Mathematische Ausdrücke . . . 69

5 Variablen und deren Bearbeitung 71
5.1 Variablendefinition: **define** . . . 71
5.1.1 Ganzzahlige Variablen: **int** . . . 72
5.1.2 Gleitkomma–Variablen: **double** . . . 73
5.1.3 Felder: **array int** und **array double** . . . 73
5.1.4 Funktionen: **function int** und **function double** . . . 74
5.1.5 String–Variablen: **string** . . . 76
5.1.6 Netzebenen–Variablen: **layer** . . . 77
5.1.7 Neuronen–Variablen: **unit** . . . 78
5.1.8 Gewichts–Variablen: **weight** . . . 78
5.2 Neuen Wert an eine Variable zuweisen: **set** . . . 79
5.2.1 Variablen vom Typ: **int** , **double** , **array int** und **array double** . 80
5.2.2 Variablen vom Typ: **string** . . . 81
5.2.3 Variablen vom Typ: **layer** . . . 81
5.2.4 Variablen vom Typ: **unit** . . . 82
5.2.5 Variablen vom Typ: **weight** . . . 83
5.3 Typumwandlungen . . . 84
5.3.1 **double** → **int** : **int_of** . . . 84
5.3.2 **int** → **double** : **double_of** . . . 85
5.4 Funktionsvariablen bearbeiten . . . 86
5.4.1 Funktion ändern . . . 86
5.4.2 Funktion ableiten: **derivation_of** . . . 86
5.5 Stringbefehle . . . 87
5.5.1 Strings konkatenieren . . . 88
5.5.2 Teilstrings kopieren: **copy** . . . 88
5.5.3 Teilstrings löschen: **delete** . . . 89
5.5.4 Teilstrings einfügen: **insert** . . . 90
5.5.5 Position eines Teilstrings bestimmen: **position_of** . . . 90
5.5.6 Stringlänge bestimmen: **length_of** . . . 91
5.6 Kommandomakros definieren: **alias** . . . 92
5.7 Eine Variable freigeben: **free_var** . . . 93

6 Kontrollstrukturen und ProSiNN–Programme **95**
6.1 Kontrollstrukturen 95
6.1.1 Vergleichsoperatoren 95
6.1.2 Die bedingte Verzweigung: **if** 97
6.1.3 Die Schleife: **while** 98
6.2 ProSiNN–Programme ausführen: **load** 99

7 Ein– und Ausgabebefehle **103**
7.1 Werte einlesen: **input** 103
7.2 Dateiverwaltung 105
7.2.1 Dateinamen und Dateihandles 106
7.2.2 Datei zum Schreiben öffnen: **open** 107
7.2.3 Datei schließen: **close** 108
7.3 Werte ausgeben 108
7.3.1 Standardausgabebefehl: **print** 108
7.3.2 Zeilenvorschub ausführen: **println** 110
7.3.3 Funktion ausgeben: **print_function** 111

8 Grafikbefehle **113**
8.1 Grafikfenster bearbeiten 113
8.1.1 Grafikhandles 113
8.1.2 Grafikfenster erzeugen: **open_grafic** 114
8.1.3 Ein Grafikfenster beenden: **close_grafic** 115
8.1.4 Alle Grafikfenster beenden: **close_all_grafics** 116
8.2 Weltkoordinatensystem setzen: **set_world** 116
8.3 Koordinatenachsen einzeichnen: **axes** 117
8.4 Strichlierungsangaben 118
8.5 Zeichenbefehle 120
8.5.1 Punkt ausgeben: **plot** 120
8.5.2 Linie ausgeben: **line** 121
8.5.3 Texte ausgeben: **text** 122
8.6 Funktionen einzeichnen: **plot_function** 124
8.7 Inhalt eines Grafikfensters löschen: **clear_grafic** 126

9 Neuronale Netze bearbeiten **127**
9.1 Ein Netz neu erzeugen: **layer** 127
9.2 Netznamen setzen: **netname** 128
9.3 Netz initialisieren 129
9.3.1 Zufallswertebereich setzen 129
9.3.2 Zufallsgeneratorstartwert setzen: **set_rand** 130
9.3.3 Netz initialisieren: **init_net** 130
9.4 ProSiNN–Netz speichern 131
9.4.1 Netz–Dateiformat 131
9.4.2 Netz speichern: **savenet** 133
9.5 Netz als C–Programm speichern 135
9.5.1 Die Include–Datei: `neural_net.h` 135
9.5.2 Die Auswertungs–Datei: `neural_net.c` 136

9.5.3 C–Programm speichern: **save_c_net** . . . 138
9.6 ProSiNN–Netz laden: **readnet** . . . 142
9.7 Netzstruktur ändern . . . 142
9.7.1 Eine Netzebene hinzufügen: **add_layer** . . . 142
9.7.2 Eine Netzebene entfernen: **sub_layer** . . . 143
9.7.3 Neuronenanzahl ändern: **alter_unit_number** . . . 144
9.8 Netzwerte benutzen oder ändern . . . 145
9.9 Netz freigeben: **free_net** . . . 148

10 Muster bearbeiten **149**
10.1 Trainingsmuster laden . . . 149
10.1.1 Das Trainingsmuster–Dateiformat . . . 149
10.1.2 Trainingsmuster laden: **readpattern** . . . 152
10.2 Testmuster laden . . . 152
10.2.1 Das Testmuster–Dateiformat . . . 153
10.2.2 Testmuster laden: **read_test_pattern** . . . 155
10.3 Muster ausgeben: **print_pattern** . . . 155
10.4 Musterwerte benutzen . . . 157
10.5 Muster freigeben . . . 159
10.5.1 Ein einzelnes Muster freigeben: **free_pattern** . . . 159
10.5.2 Alle Muster freigeben: **free_all_pattern** . . . 160

11 Neuronale Netze trainieren **161**
11.1 Verwendung der zugehörigen Systemvariablen . . . 161
11.2 Netz trainieren . . . 164
11.2.1 Trainiere Netz mit allen Mustern: **cycle** . . . 164
11.2.2 Trainiere Netz mit einem Muster: **cycle_pattern** . . . 166
11.3 Aktivierungsfunktion ändern . . . 169
11.3.1 Neue Aktivierungsfunktion setzen: **activation** . . . 169
11.3.2 Ableitung der Aktivierungsfunktion ändern . . . 171
11.3.3 'Normale' Aktivierungsfunktion setzen: **normal_activation** . . . 172
11.4 Netzantworten verarbeiten . . . 173
11.4.1 Netzantwort ausgeben: **reply_to** . . . 173
11.4.2 Fehler der Netzantwort ausgeben: **print_error_of** . . . 175
11.4.3 Fehler der Netzantwort einem Array zuweisen: **error_of** . . . 176
11.5 Schnelles Netztraining: **fast_cycle** . . . 177

12 Trainingsprotokolle **181**
12.1 Textprotokolle . . . 181
12.1.1 Textprotokoll erzeugen: **protocol** . . . 181
12.1.2 Alle Textprotokolle freigeben: **free_protocol** . . . 184
12.2 Grafikprotokolle . . . 184
12.2.1 Grafikprotokoll erzeugen: **grafic_protocol** . . . 184
12.2.2 Alle Grafikprotokolle freigeben: **free_grafic_protocol** . . . 187

13 Systembefehle **189**
13.1 Variableninformationen 189
13.1.1 Systemvariablen ausgeben: **sys_names** 189
13.1.2 Benutzervariablen ausgeben: **user_names** 190
13.1.3 Alle definierten Bezeichner ausgeben: **names** 191
13.2 Statusinformationen ausgeben: **status** 192
13.3 Zeit ausgeben: **time** 193
13.4 Globale Variablenverwaltung 194
13.4.1 Systemvariablen neu initialisieren: **init** 194
13.4.2 Variablen freigeben: **free** 195
13.4.3 Alles freigeben: **free_all** 196
13.5 Verzeichnisverwaltung 197
13.5.1 Arbeitsverzeichnisinhalt ausgeben: **dir** 197
13.5.2 Arbeitsverzeichnis ausgeben: **pwd** 198
13.5.3 Arbeitsverzeichnis wechseln: **cd** 198
13.6 Betriebssystemkommando ausführen: **shell** 200
13.7 Ein Programm oder ProSiNN beenden: **quit** 201

III Die Menüoberfläche von ProSiNN **203**

14 Das Menü „Netz bearbeiten“ 205
14.1 Der Menüpunkt „**Netz erzeugen**“ 206
14.2 Der Menüpunkt „**Netz initialisieren**“ 207
14.3 Der Menüpunkt „**Netz laden**“ 208
14.4 Der Menüpunkt „**Netz speichern**“ 209
14.5 Der Menüpunkt „**Netzstruktur ändern**“ 209
14.6 Der Menüpunkt „**Netzwerte bearbeiten**“ 211
14.7 Der Menüpunkt „**Netz freigeben**“ 213

15 Das Menü „Muster bearbeiten“ **215**
15.1 Der Menüpunkt „**Trainingsmuster laden**“ 216
15.2 Der Menüpunkt „**Testmuster laden**“ 216
15.3 Der Menüpunkt „**Muster bearbeiten**“ 216
15.4 Der Menüpunkt „**Alle Muster freigeben**“ 218

16 Das Menü „Netz trainieren“ 221
16.1 Der Menüpunkt „**Trainiere Netz (1000 Zyklen)**“ 222
16.2 Der Menüpunkt „**Trainiere Netz**“ 223
16.3 Der Menüpunkt „**Trainiere Netz mit einem Muster**“ 224
16.4 Der Menüpunkt „**Propagiere Muster durch das Netz**“ 225
16.5 Der Menüpunkt „**Bearbeite Standardprotokoll**“ 227
16.6 Der Menüpunkt „**Bearbeite Grafikprotokoll**“ 229

17 Das Menü „Systemwerte bearbeiten“ 233
17.1 Der Menüpunkt **„Systemvariablen bearbeiten“** 234
17.2 Der Menüpunkt **„Aktivierungsfunktion definieren“** 236
17.3 Der Menüpunkt **„Ableitung der Aktivierungsfunktion setzen“** 238
17.4 Der Menüpunkt **„Normale Aktivierungsfunktion setzen“** 240
17.5 Der Menüpunkt **„Ausgabefenster speichern“** 240
17.6 Der Menüpunkt **„Ausgabefenster loeschen“** 241
17.7 Der Menüpunkt **„Alles freigeben“** 241

18 Das Menü „Benutzerwerte bearbeiten“ 243
18.1 Der Menüpunkt **„Datei editieren“** 244
18.2 Der Menüpunkt **„Programm laden und ausfuehren“** 246
18.3 Der Menüpunkt **„Variablen definieren“** 247
18.3.1 Der Menüunterpunkt **„Integer/Doubles definieren“** 247
18.3.2 Der Menüunterpunkt **„Zeichenketten definieren“** 248
18.3.3 Der Menüunterpunkt **„Felder definieren“** 249
18.3.4 Der Menüunterpunkt **„Funktionen definieren“** 250
18.3.5 Der Menüunterpunkt **„Benutzerkommandos definieren“** 253
18.3.6 Der Menüunterpunkt **„Netzvariablen definieren“** 254
18.4 Der Menüpunkt **„Variablen bearbeiten“** 255
18.4.1 **„Integer/Doubles bearbeiten“** 256
18.4.2 **„Zeichenketten bearbeiten“** 257
18.4.3 **„Felder bearbeiten“** 258
18.4.4 **„Funktionen bearbeiten“** 260
18.4.5 **„Benutzerkommandos bearbeiten“** 264
18.4.6 **„Netzvariablen bearbeiten“** 265
18.5 Der Menüpunkt **„Grafikfenster bearbeiten“** 266
18.5.1 **„Grafikfenster loeschen“** 269
18.5.2 **„Inhalt des Grafikfensters loeschen“** 269
18.5.3 **„Weltkoordinatensystem setzen“** 269
18.5.4 **„Koordinatenachsen ausgeben“** 270
18.5.5 **„Linie ausgeben“** 271
18.5.6 **„Text ausgeben“** 273
18.6 Der Menüpunkt **„Alle Variablen freigeben“** 274

19 Die Menüs „ProSiNN Info“ und **„Programm beenden“** 275
19.1 Der Menübutton **„ProSiNN Info“** 275
19.2 Der Menüpunkt **„Programm beenden“** 276

IV Anwendungsbeispiele **277**

20 Beispiele für das Training Neuronaler Netze **279**
20.1 Das XOR–Problem 280
20.2 Das Encoder–Problem 285
20.3 Das T–C Problem 289

21 Training mit dynamischer Neuronenerzeugung **297**
21.1 Die zu untersuchende Aufgabenstellung: Das Paritäts–Problem 297
21.2 Die Programmschnittstelle und die Lösung der Aufgabenstellung 298
21.3 Das ProSiNN–Programm mit dynamischer Neuronenerzeugung 306

Literaturverzeichnis **313**

V Anhänge **317**

A Referenzliste der vordefinierten Bezeichner **319**

B Referenzliste der ProSiNN-Fehlermeldungen **327**

C Installation des ProSiNN–Programmsystems **335**
C.1 Das Ablaufschema des Systems 335
C.2 Die zentralen Datenstrukturen 337
C.2.1 Die Datenstruktur zur Auswertung von ProSiNN–Kommandos . . . 337
C.2.2 Die Datenstruktur eines Neuronalen Netzes 337
C.2.3 Die Datenstruktur einer Symboltabelle und deren Elemente 340
C.2.4 Die Datenstruktur des ProSiNN–Kontrollblocks 345
C.3 Die Systeminstallation 352
C.3.1 Die der Entwicklung zugrundeliegende Rechnerumgebung 352
C.3.2 Die Quelldateien des Systems 352
C.3.3 Die Generierung des ProSiNN–Systems 355

Kapitel 1

Einleitung

Innerhalb des Forschungsgebietes der *Künstlichen Intelligenz (Artifical Intelligence)* beschäftigt man sich u.a. mit Methoden, die ein Rechensystem in die Lage versetzen, *intelligentes* Verhalten zu simulieren. Zu den Anwendungsmöglichkeiten gehören z.B. das Erkennen und Verarbeiten von akustischen und visuellen Informationen, die Steuerung von Maschinen und Robotern, Expertensysteme (sie wenden das spezialisierte Wissen von Experten zur Lösung von Aufgaben an, die nicht auf einer fundamentierten Theorie basieren müssen) und die Modellierung kognitiver Prozesse.

Ein in den letzten Jahren an Interesse zunehmender Forschungszweig ist das Gebiet der *Künstlichen Neuronalen Netze (Konnektionistische Systeme, Parallele Gedächnismodelle, Parallel Distributed Processing)*. Ein solches System besteht in Anlehnung an das menschliche Nervensystem aus einer Vielzahl von einfachen, parallel arbeitenden Prozessoren, die über gewichtete Verbindungen Informationen austauschen. Dabei kann eine Verbindung die auf ihnen fließenden Signale verstärken, hemmen oder invertieren. Ein Prozessor verarbeitet seine Eingangsinformationen mit Hilfe weniger Operationen und einem relativ kleinen Speicher zu einem Ausgangssignal. Diese Ausgabe wird dann an andere Einheiten des Netzes weitergegeben. So bildet ein Künstliches Neuronales Netz eine Folge von Eingabedaten auf eine Folge von Ausgabedaten ab. Die Topologie eines Netzes und die Verbindungsgewichte bestimmen dabei im wesentlichen das *Wissen* eines Neuronalen Netzes. Um ein bestimmtes Ausgabeverhalten zu realisieren, werden für die unterschiedlichen Modelle eine Reihe von Lernverfahren angeboten, die die Verbindungsgewichte an eine vorgegebene Aufgabenstellung anpassen. Neuronale Netze zeigen Eigenschaften der Verallgemeinerung und die Fähigkeit zu assoziieren. Sie sind fehlertolerant und können komplexe Informationen bewerten.

Das dieser Arbeit zugrundeliegende Netzwerkmodell ist das *mehrschichtige Perceptron*[1]: Es handelt sich dabei um einen Netztyp, der durch die schichtartige Anordnung der Einheiten und den streng vorwärts gerichteten Signalfluß charakterisiert ist. Die gewichteten Eingangsignale einer Einheit werden aufsummiert und einer — in der Regel — *nichtlinearen Aktivierungsfunktion* zugeführt, deren Ergebnis als *Aktivierungswert* der Zelle zu den Einheiten der nachfolgenden Schichten weitergeleitet wird.

[1] Der Terminus *Perceptron* leitet sich von dem englischen Wort *perception* für Wahrnehmung ab

Das gewünschte Ausgabeverhalten wird durch eine Ansammlung von *Mustern* festgelegt. Jedes Muster enthält die Daten, die eine Eingabe und die dazugehörige Ausgabe des Netzes spezifizieren. Zur Lösung einer vorgegebenen Aufgabe wird eine *Kostenfunktion* durch Annäherung des Ist–Verhaltens der Einheiten an das Soll–Verhalten minimiert. Mit einer systematischen Korrektur der Verbindungsgewichte wird diese Optimierung erzielt.

Um Künstliche Neuronale Netze auf der Basis des mehrschichtigen Perceptrons generieren, trainieren und untersuchen zu können, wurde die programmierbare grafische Simulationsumgebung **ProSiNN** (**Pro***grammierbarer* **Si***mulator für Künstliche* **N***euronale* **N***etze*) entwickelt. Das implementierte System bietet eine universelle Programmiersprache, die mit Fähigkeiten zur Erzeugung und dem Training von Neuronalen Netzen ausgestattet ist. Mit der Unterstützung eines Kommandozeileninterpreters kann der Benutzer alle existierenden Befehle direkt eingeben und ausführen lassen. Außerdem wird eine Menüoberfläche bereitgestellt, die die komfortable Bearbeitung von Neuronalen Netzen und von ProSiNN–Simulationsprogrammen gestattet.

Mit ProSiNN wird dem Anwender ein Werkzeug zur Verfügung gestellt, das insbesondere zur Synthese und Analyse von Künstlichen Neuronalen Netzen entwickelt wurde. Dabei sind die Netze mit Variationen des Backpropagation–Algorithmus trainierbar. Der Benutzer kann hier aber ein beliebiges Lernverfahren einsetzen, da jede netzinterne Struktur durch die Programmiersprache ansprechbar und modifizierbar ist. Aber auch andere Aufgaben, die dieses Gebiet nicht betreffen, sind mit ProSiNN lösbar. Das durch das System vorgegebene Anwendungsspektrum ist im weiteren nur durch eine möglicherweise zu zeitaufwendige Interpretation der auszuführenden Programme und der Kreativität des Benutzers eingegrenzt.

In den beiden folgenden Abschnitten wird die Gliederung der Arbeit und die Darstellung der Syntaxregeln, die u.a. die Grammatik der von ProSiNN angebotenen Programmiersprache beschreibt, dargestellt.

1.1 Aufbau der Arbeit

Das vorliegende Dokument ist in fünf Teile untergliedert: Nach einer Einführung in die Theorie der Neuronalen Netze und des Programmsystems ProSiNN wird im zweiten Teil die realisierte Programmiersprache beschrieben. Anschließend wird die Benutzung der Menüoberfläche dargestellt. Beispiele, die einen Einstieg in die Anwendung liefern und die die Möglichkeiten der fortgeschrittenen Programmierung zeigen, sind der Gegenstand des vierten Teils. In den danach folgenden Anhängen werden alle vordefinierten Bezeichner und Fehlermeldungen des Systems aufgelistet, sowie die Installation von ProSiNN geschildert.

Die Theorie der Künstlichen Neuronalen Netze und eine Einführung in das Programmsystem ProSiNN stehen im Mittelpunkt des ersten Teils dieser Arbeit.

Im zweiten Kapitel wird nach einem historischen Rückblick über die Entwicklung von Künstlichen Neuronalen Netzen und deren allgemeine Grundlagen das mehrschichtige Perceptron vorgestellt. Im weiteren wird das von ProSiNN verwendete Lernverfahren — der modifizierte *Backpropagation–Algorithmus* — und eine Methode, die während des Trainings die Netztopologie bis zum Erzielen einer Lösung verändert, präsentiert.

Die wichtigsten Merkmale und das von ProSiNN verwendete Netzmodell werden im folgenden Kapitel beschrieben. Anschließend wird der Start des Systems und der Umgang mit der Menüoberfläche und dem Kommandozeileninterpreter, der eine interaktive Eingabe der von ProSiNN unterstützten Befehle erlaubt, dargestellt.

Der zweite Teil dieses Dokumentes enthält die Beschreibung der von ProSiNN unterstützten Programmiersprache.

Im vierten Kapitel werden alle von ProSiNN angebotenen Datentypen, die die Grundlage der verwendeten Variablen bilden, sowie die Schlüsselwörter und die vordefinierten Bezeichner vorgestellt. Die arithmetischen Operatoren, die mathematischen Ausdrücke und Funktionen runden dieses Kapitel ab.

Das folgende Kapitel beschäftigt sich mit der Erzeugung von Variablen und der Veränderung ihrer Inhalte. Typumwandlungen zwischen Variablen, die ganze Zahlen und Gleitkommawerte enthalten, sowie die Bearbeitung von Zeichenketten, Funktionen und die Generierung einer beliebigen symbolischen Ableitung stellen weitere Schwerpunkte dar. Zum Schluß wird die Definition eines Makros, das eine Kommandosequenz beinhaltet, und die Freigabe einzelner Variablen beschrieben.

Das sechste Kapitel führt die von ProSiNN unterstützten Kontrollstrukturen, die eine bedingte Verzweigung und eine wiederholte Ausführung von Programmsegmenten erlauben, ein. Der syntaktische Aufbau eines Simulationsprogramms und der Befehl, der ein solches ausführt, werden abschließend präsentiert.

Die von ProSiNN angebotenen Ein– und Ausgabeanweisungen bilden den Inhalt des folgenden Kapitels. Dazu wird ein Kommando vorgestellt, das vom Anwender am Bildschirm interaktiv Eingaben anfordert. Daran schließt sich die Darstellung der Dateiverwaltung an, die das Schreiben beliebiger Daten in eine Datei ermöglicht. Die wichtigsten Befehle, die beliebige Konstanten, Variablen und Funktionen ausgeben, werden anschließend eingeführt.

Im achten Kapitel werden die Kommandos zur Generierung von grafischen Ausgaben beschrieben. Dazu werden die Befehle zur Verwaltung von Grafikfenstern, zum Setzen eines Weltkoordinatensystems und zum Einzeichnen von Koordinatenachsen, Punkten, Linien, Texte und Funktionen präsentiert.

Die Bearbeitung eines Neuronalen Netzes ist der Gegenstand des neunten Kapitels. Hier werden Anweisungen eingeführt, die ein Neuronales Netz erzeugen, initialisieren, laden und in wieder einlesbarer Form oder als C–Programm speichern. Anschließend werden Kommandos zur Änderung der Netztopologie, d.h. dem Hinzufügen oder Entfernen von Netzschichten bzw. Einheiten, und zum Freigeben des im System vorliegenden Netzes vorgestellt.

Die Musterverwaltung wird im folgenden Kapitel erörtert. Es werden Befehle zum Laden der Muster, die eine Lernaufgabe spezifizieren oder die nur zur Berechnung der dazugehörigen Netzantwort verwendet werden sollen, dargestellt. Weitere Anweisungen schreiben einzelne Muster auf ein Ausgabemedium oder geben sie wieder frei.

Das elfte Kapitel ist dem Training von Neuronalen Netzen gewidmet. Es werden Kommandos eingeführt, die ein vorliegendes Netz mit einzelnen oder allen Mustern trainieren, die benutzte Aktivierungsfunktion verändern und die Netzantwort bzw. den entstandenen Fehler ausgeben oder einer Variablen zuweisen. Besitzt das verwendete Rechensystem keinen Co–Prozessor für Fließkommaberechnungen, so bietet der anschließend präsentierte Befehl die Möglichkeit, das Netztraining mit einer schnellen Festkommaarithmetik durchzuführen.

Der Gegenstand des zwölften Kapitels ist die Verwaltung von Protokollen, die das Training durch parallele Ausgabe von beliebigen Werten für den Anwender transparent machen. Es werden Kommandos zu deren Definition vorgestellt, die Daten für den Benutzer aufbereiten, indem sie sie auf ein Medium ausgeben oder in ein Grafikfenster einzeichnen. Zum Schluß werden die Kommandos beschrieben, die die Definitionen der Protokolle wieder zurücknehmen.

Das folgende Kapitel beschäftigt sich mit den Befehlen, die Informationen über das System liefern oder zu dessen Verwaltung dienen. Dazu werden Anweisungen eingeführt, die vordefinierte oder vom Benutzer generierte Konstanten, Variablen und Kommandos ausgeben, einen Systemstatusbericht erzeugen, die aktuelle Zeit liefern, das System neu initialisieren und bestimmte oder alle Variablen wieder freigeben. Weitere Befehle betreffen die Verwaltung des Arbeitsverzeichnisses, den Aufruf eines Kommandointerpreters des Betriebssystems und die Beendigung eines Simulationsprogramms oder der gesamten ProSiNN–Umgebung.

Der dritte Teil dieser Arbeit beschreibt die Menüoberfläche von ProSiNN.

Im vierzehnten Kapitel wird die Benutzung des Menüs *Netz bearbeiten* geschildert. Hier werden Menüpunkte angeboten, die das Erzeugen, Initialisieren, Laden, Speichern und Freigeben eines Netzes ermöglichen. Interne Werte eines Neuronalen Netzes können betrachtet und geändert werden. Auch die Topologie eines vorliegenden Netzes läßt sich modifizieren.

Daran schließt sich ein Kapitel an, das das Menü *Muster bearbeiten* erläutert. Die implementierten Menüpunkte können Muster — die zum Training oder zur Berechnung der Netzantwort verwendet werden sollen — laden, bearbeiten und wieder freigeben.

Das Menü *Netz trainieren* ist der Gegenstand des sechzehnten Kapitels. Die vorhandenen Menüpunkte erlauben das Netztraining mit einzelnen oder allen Mustern, die Berechnung der Netzantwort, sowie die Generierung und das Freigeben von Protokollen, die parallel zum Training beliebige Ausgaben in das ProSiNN–Hauptfenster oder in ein Grafikfenster durchführen.

Das Menü *Systemwerte bearbeiten* wird im folgenden Kapitel erörtert. Hier werden Menüpunkte bereitgestellt, die das Anzeigen und Modifizieren von vordefinierten Variablen, die Änderung der Aktivierungsfunktion und das Löschen der vom Benutzer erzeugten Strukturen ermöglichen. Außerdem läßt sich das Hauptfenster von ProSiNN, dessen Inhalt alle generierten Ein– und Ausgaben enthält, in einer Datei speichern.

Im achtzehnten Kapitel wird das Menü *Benutzerwerte bearbeiten* vorgestellt. Die angebotenen Menüpunkte erlauben das Editieren von Dateien, das Ausführen eines Simulationsprogramms, sowie die Generierung und die Bearbeitung von Variablen und Grafikfenstern.

Das folgende Kapitel beschreibt das Menü *ProSiNN Info*, das Informationen über das System und den Autor liefert, und das Menü *Programm beenden*, das die Simulationsumgebung beendet.

Im vierten Teil des vorliegenden Dokumentes werden Anwendungsbeispiele präsentiert.

Das zwanzigste Kapitel enthält drei Aufgabenstellungen, deren Lösbarkeit mit dem Programmsystem ProSiNN untersucht werden soll. Bei der Lösung dieser Probleme wird u.a. auch die Benutzung der Menüoberfläche und des Kommandozeileninterpreters dargestellt.

Im letzten Kapitel wird ein Simulationsprogramm vorgestellt, das die in Kapitel zwei eingeführte Methode, die während des Trainings die Netztopologie bis zum Erzielen einer Lösung verändert, realisiert. Die Anwendbarkeit dieses Verfahrens wird anhand einer Problemstellung gezeigt.

Der fünfte Teil dieser Arbeit enthält die Anhänge.

Im Anhang A werden alle Schlüsselwörter und vordefinierten Bezeichner alphabetisch aufgelistet. Zu jedem Element wird eine kurze Beschreibung und eine Seitenangabe, auf der das Schlüsselwort oder der vordefinierte Bezeichner eingeführt wurde, angegeben.

Die numerisch sortierte Liste aller von ProSiNN verwendeten Fehlernummern und die dazugehörigen Erklärungen werden in dem folgenden Anhang präsentiert.

In Anhang C wird die Dokumentation und die Installation des ProSiNN–Programmsystems geschildert. Nachdem das grundsätzliche Ablaufschema des Systems erläutert wurde, werden die zentralen Datenstrukturen vorgestellt. Die der Entwicklung zugrundeliegende Rechnerumgebung, die Quelldateien und die Generierung des Programmsystems werden abschließend beschrieben.

1.2 Darstellung der Syntaxregeln

Zur Beschreibung von syntaktischen Strukturen wird die *Erweiterte Backus–Naur–Form (EBNF)* verwendet. Dazu werden folgende Konventionen getroffen:

- Ein *Terminalsymbol* steht für eine in der Struktur vorhandene Zeichenfolge und wird in kursiver Schrift und Anführungszeichen angegeben:

  "int" bezeichnet z.B. die Zeichenfolge "`int`", die an dieser Stelle eingesetzt werden muß (die Anführungszeichen gehören nicht zum Terminalsymbol).

- Ein *Nichtterminalsymbol* repräsentiert eine syntaktische Einheit, die an dieser Stelle definiert wird oder einzufügen ist. Diese Elemente werden durch eine fette Schriftart gekennzeichnet:

  ident steht z.B. für ein Nichtterminalsymbol mit dem entsprechenden Namen.

- Ein Nichtterminalsymbol wird durch eine Definitionszeile festgelegt. Dabei kennzeichnet der Rumpf, der durch "::=" eingeleitet und mit einem Punkt abgeschlossen wird, den Inhalt der syntaktischen Einheit.

  Zur Spezifizierung des Rumpfes existieren weitere *Metazeichen*, die im folgenden eingeführt werden:

  - Eine Folge von Nichtterminalsymbolen und/oder Terminalsymbolen kennzeichnet die Konkatenation der syntaktischen Elemente. Befindet sich zwischen zwei angegebenen Elementen mindestens ein Leerzeichen, so darf in der entsprechenden Struktur eine beliebige Anzahl von Tabulatoren, Leer– und Zeilenendezeichen stehen.

    In dem folgenden Beispiel wird das Nichtterminalsymbol **expression** als Konkatenation des Nichtterminalsymbols **term_1**, des Terminalsymbols "+" und des Nichtterminalsymbols **term_2** definiert:

    expression ::= term_1 "+" term_2.

  - Um zwischen Alternativen auswählen zu können wird das *Metazeichen* | verwendet:

    add_operator ::= "+" | "–".

    Hier wird das Nichtterminalsymbol **add_operator** durch das Terminalsymbol "+" oder "-" definiert.

  - Gruppierungen einzelner syntaktischer Elemente werden durch runde Klammern dargestellt.

    In dem folgenden Beispiel wird das Nichtterminalsymbol **expression** als Konkatenation des Nichtterminalsymbols **term_1**, des Terminalsymbols "+" oder "-" und des Nichtterminalsymbols **term_2** definiert:

    expression ::= term_1 ("+" | "+") term_2.

  - Zur Angabe von optionalen Elementen werden rechteckige Klammern benutzt:

    integer ::= ["+" | "–"] decimal_const.

    Das Nichtterminalsymbol **integer** wird hier durch das Nichtterminalsymbol **decimal_const** oder als die Konkatenation des Terminalsymbols "+" oder "-" mit dem Nichtterminalsymbol **decimal_const** definiert.

  - Innerhalb von geschweiften Klammern können Elemente angegeben werden, die beliebig oft wiederholt werden dürfen. Die eingeschlossene Symbolfolge kann aber auch weggelassen werden.

    Die folgende Zeile definiert das Nichtterminalsymbol **decimal_const** als Konkatenation des Nichtterminalsymbols **decimal_digit** mit einer beliebigen Anzahl desselben Symbols:

    decimal_const ::= decimal_digit { decimal_digit }.

  - Um ein Nichtterminalsymbol in der natürlichen Sprache beschreiben zu können, wird der Informationstext in fetter Schrift und Anführungszeichen dargestellt:

    object_name ::= „Bezeichner, der ein Objekt spezifiziert“.

Beispiel:
Die Syntax einer beliebigen ganzen Zahl, die hier durch das Nichtterminalsymbol **integer** repräsentiert wird, könnte folgendermaßen definiert werden:

integer ::= [*"+"* | *"-"*] **decimal_const.**
decimal_const ::= **decimal_digit** { **decimal_digit** }.
decimal_digit ::= *"0"* | *"1"* | *"2"* | *"3 "* | *"4"* | *"5"* | *"6"* | *"7 "* | *"8"* | *"9"*.

Teil I

Neuronale Netze und ProSiNN

Kapitel 2

Künstliche Neuronale Netze

Im praktischen Anwendungsbereich der Künstlichen Intelligenz sind bis vor einigen Jahren nur Systeme eingesetzt worden, die mit Hilfe von imperativen oder deduktiven Algorithmen Aufgaben lösten. Dazu mußten deren theoretische Grundlagen bekannt sein bzw. die Beziehungen der einzelnen *Wissensfakten* mit zu entwickelnden Regeln aufeinander abgestimmt werden. *Künstliche Neuronale Netze*[1] basieren auf problemunabhängigen Modellen und bieten darüber hinaus neue Einsatzmöglichkeiten. Systeme werden nicht mehr programmiert, sondern lernen eine Aufgabe selbständig: Ihr *Wissen* erwerben sie an Hand von Erfolgen und Fehlschlägen, die durch Interaktion mit der Umwelt entstehen. Dabei wird ein den biologischen Gehirnstrukturen nachempfundenes Modell von einem Computer simuliert oder direkt durch Schaltkreise realisiert.

In diesem Kapitel werden die theoretischen Grundlagen von Neuronalen Netzwerken eingeführt: Der erste Abschnitt beschreibt das biologische Gehirn als dessen Quelle und gibt einen historischen Überblick über die Entwicklung auf diesem Gebiet. Anschließend wird das in ProSiNN verwendete Netzmodell, das *mehrschichtige Perceptron*, und dessen Lernregel, das *Backpropagation*–Verfahren, vorgestellt. ProSiNN benutzt — sofern der Anwender keinen eigenen Algorithmus programmiert — ein modifiziertes Lernverfahren, das der Gegenstand des dritten Abschnitts ist. Abschließend wird eine Methode präsentiert, die solange die Topologie eines Neuronalen Netzes verändert, bis eine Lösung der vorgegebenen Aufgabenstellung gefunden wird.

2.1 Einführung in die Theorie Neuronaler Netze

Neuronale Netze haben — wie der Name schon sagt — mit der Vernetzung von Neuronen zu tun. Ein Neuron ist eine Konstruktion, die den biologischen Gehirnzellen nachgebildet worden ist. In Anlehnung an das menschliche Nervensystem werden Informationen durch die Interaktion der Neuronen verarbeitet. Die Kommunikation erfolgt dabei über die Verbindungen zwischen den Neuronen.

Für das Verständnis von Neuronalen Netzen ist es sinnvoll, sich mit deren Quelle zu beschäftigen, den biologischen Gehirnstrukturen:

[1] Im folgenden wird der Terminus *Neuronale Netze* für *Künstliche Neuronale Netze* verwendet

Das menschliche Gehirn enthält mehr als 10 Milliarden Nervenzellen (Neuronen), die Informationen verarbeiten und übertragen können. Eine solche Zelle besteht aus einem Zellkörper und einem darin befindlichen Zellkern (siehe Abbildung 2.1):

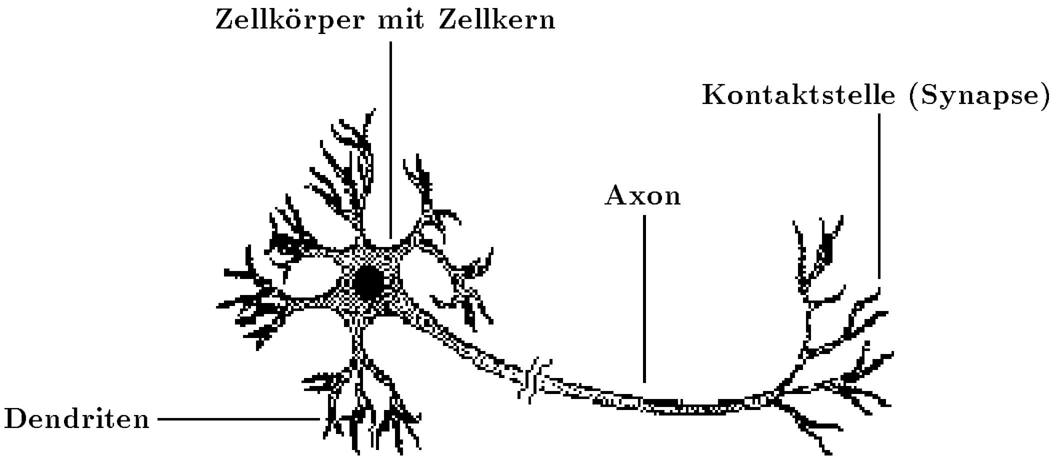


Abbildung 2.1: Das biologische Neuron

Die Eingabesignale erhält ein Neuron über feine Fasern (*Dendriten*), deren Enden den Kontakt zu anderen Zellen herstellen. Die Ausgangsinformation wird über Ausläufer der Zellmembran (*Axon*) zu anderen Neuronen übertragen. Das Axon fächert sich baumartig auf, um das Signal an die Eingänge der nachgeschalteten Neuronen weiterzuleiten. Die Kontaktstellen zwischen den Neuronen werden *Synapsen* genannt (im Mittel ist jedes Neuron mit 100 bis 10000 Neuronen verbunden). Diese Verbindungen sind nicht statisch: Sie können innerhalb des Lebenszyklus wieder gelöst werden, aber auch neue Kontakte können entstehen. Auch die Übertragungseigenschaften der Synapsen sind einem veränderlichen Prozeß unterworfen.

Ein Neuron, das an seinen Eingängen Signale (die durch elektrische Erregungsreize realisiert werden) empfängt, summiert diese Informationen auf. Dabei werden die einzelnen Signale durch die Synapsen unterschiedlich gewichtet: Eine Synapse kann den Informationsfluß hemmen oder ihn fördern. Wenn die Summe der gewichteten Eingangssignale einen Schwellenwert überschreiten, wird über das Axon ein elektrischer Impuls als Ausgangssignal an die nachfolgenden Neuronen übertragen. Diese verarbeiten dann in der gleichen Art und Weise die ankommenden Informationen weiter.

Die einzelnen Vorgänge — soweit sie bekannt sind — laufen folgendermaßen ab:

In der Zellmembran befinden sich Ionenpumpen, die Ladungsträger durch das Membran transportieren. Liegen keine Eingangssignale vor, dann besteht zwischen beiden Seiten der Membran eine elektrische Spannung, das *Ruhepotential.* Wird durch eingehende Informationen ein Schwellenwert überschritten, so öffnen sich Ionenkanäle in der Zellmembran. Dadurch entsteht ein *Aktionspotential* (die Membranspannung ändert für ca. 1 ms ihr Vorzeichen). Aus der Nachbarschaft fließen kleine Ströme zum Ausgleichen der Spannungsdifferenz herbei. Dadurch breitet sich das Aktionspotential auf dem Axon aus und erzeugt in den Synapsen der nachfolgenden Neuronen biochemische Vorgänge.

Durch das Eintreffen des Aktionspotentials — dessen Nutzsignal durch die Frequenz und Phase charakterisiert ist — an einer Synapse wird ein chemischer Stoff freigesetzt, der in den schmalen Übergang zwischen der Sender– und Empfängerzelle eintritt (es handelt sich also um keine direkte elektrische Kopplung). Diese Substanz bewirkt eine Wechselwirkung mit Molekülen auf der Membran des Zielneurons. Dadurch wird für kurze Zeit das Membranpotential beeinflußt und das Empfängerneuron kann evtl. ein eigenes Aktionspotential bilden. Neben dieser *erregenden* Synapse, die zur Bildung eines Aktionspotentials positiv beiträgt, gibt es auch noch *hemmende* Synapsen, die die Membranspannung negativ beeinflussen und somit die Ausbildung eines Aktionspotentials erschweren.

Abstrakt gesehen ist ein biologisches Neuron ein Prozessor, der mehrere Eingangssignale zu einem Ausgabesignal verarbeitet.

Das biologische Gehirn unterscheidet sich deutlich von einem herkömmlichen Computer: Es speichert seine Informationen nicht in klar lokalisierbarer Form (also in exakt adressierbaren Speicherzellen), sondern über die gesamte Gehirnstruktur verteilt (ähnlich einem Hologramm). Die Änderungen der Verbindungsstruktur und der Übertragungseigenschaften der Synapsen geben den Lernprozess im täglichen Leben wieder. Das Gehirn stellt somit ein stark parallel arbeitendes System dar, das außerdem den Vorteil aufweist, auch bei einem Ausfall von bestimmten Teilbereichen noch ungestört arbeiten zu können.

Die Tabelle 2.1 zeigt im Überblick einige Eigenschaften des menschlichen Gehirns und eines konventionellen seriellen Computers:

Eigenschaft	Gehirn	Computer
Parallelität	hoch	niedrig
Speicherzugriff	global	lokal
Selbstorganisation	ja	nein
Präzision	niedrig	hoch
Fehlertoleranz	hoch	niedrig
Fähigkeit zur Generalisierung	gut	schlecht

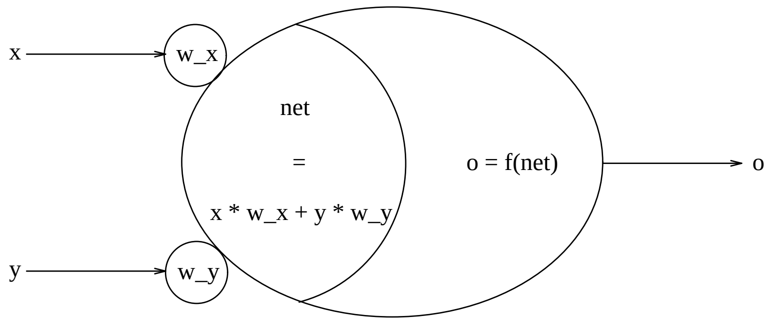
Tabelle 2.1: Vergleich einiger Eigenschaften des menschlichen Gehirns und eines Computers

Die positiven Eigenschaften des menschlichen Gehirns legen den Wunsch nahe, ein mathematisches Modell zu entwickeln, das vergleichbare Vorteile aufweist.

Um ein biologisches Neuron annähernd zu modellieren, müssen die Aktivierungen und die synaptischen Verbindungen simuliert werden. Dabei dürfen nur Rechenoperationen und lokale Variablen verwendet werden.

1943 gelang es McCulloch und Pitts eine einfache Zelle nach dem biologischen Vorbild zu modellieren [PIT43]. Dieses künstliche Neuron kann z.B. binäre Funktionen realisieren.

In Bild 2.2 ist diese Zelle dargestellt:



$$f(\text{net}) = \begin{cases} 1 & , \text{ falls net} > \text{theta} \\ 0 & , \text{ sonst} \end{cases}$$

Binäres Gatter	w_x	w_y	theta
NAND	-1	-1	-1.5
NOR	-1	-1	-0.5

Abbildung 2.2: Das McCulloch–Pitts–Neuron

Die Eingangssignale stammen aus der zweielementigen Menge $\{0, 1\}$. x und y werden mit den synaptischen Gewichten w_x und w_y multipliziert und zu einem einzigen Wert zusammenaddiert (diese Abbildung heißt *Propagierungsfunktion*). Anschließend wird das Ergebnis einer Aktivierungsfunktion zugeführt: Übersteigt die Summe einen bestimmten Schwellenwert (die abhängig von der zu realisierenden binären Funktion ist), so antwortet das Neuron mit einer 1; ansonsten wird 0 ausgegeben.

Da mit dem McCulloch–Pitts–Neuron jede binäre Grundoperation erzeugbar ist (mit NOR– und NAND–Gattern sind alle anderen Operatoren generierbar), kann mit diesem Modelltyp die gesamte heutige Digitaltechnik erfaßt werden.

Eine allgemeine Lernregel, die die synaptischen Gewichte und den Schwellenwert an eine beliebige Aufgabe anpaßt, konnten McCulloch und Pitts aber nicht herleiten.

Weitere Fortschritte entstanden erst, als D. Hebb 1949 für die Anpassung der synaptischen Gewichte forderte [HEB49]: Eine verstärkte Kopplung (die den erregenden Charakter der biologischen Synapsen repräsentiert) zwischen einem Sende– und einem Empfangsneuron entsteht nur dann, wenn beide Zellen aktiviert sind. In der Praxis bedeutet diese Annahme, daß die synaptischen Verbindungsgewichte proportional zu dem Produkt der vorliegenden Aktivierungen modifiziert werden. Mit dieser Überlegung konnte aber noch kein explizit trainierbares Modell entwickelt werden. Allerdings sollte seine Hypothese die Grundlage vieler Lernverfahren werden.

1960 entwickelte G. Widrow und M. E. Hoff ein neues Netzwerkmodell, das *Adaline–Netzwerk* [WID60]. Diese Konstruktion wurde zum ersten Mal auch für praktische Probleme eingesetzt (als adaptiver Filter, der die Echoeffekte bei Ferngesprächen in Telefonleitungen unterdrückt).

1962 entwarf Frank Rosenblatt ein lernfähiges Modell, das *Perceptron* (siehe [ROS58] und

[ROS62]). Diese Generation von Neuronalen Netzen kann einfache Muster klassifizieren und ist darüber hinaus fehlertolerant. Das Modell verwendet eine Neuronenschicht, die die Eingangsinformationen mit den synaptischen Gewichten multipliziert und über eine Schwellwertlogik die binäre Entscheidung der Klassenzugehörigkeit eines Eingabemusters ausgibt. Durch einen iterativen Lernvorgang werden die Gewichte der Neuronen modifiziert. Mit diesem Verfahren lassen sich allerdings nur linear separable Muster erlernen (siehe dazu die Bemerkung in Abschnitt 2.4 auf Seite 27). Theoretisch konnte sogar bewiesen werden, daß nach endlich vielen Lernschritten all die Muster klassifiziert werden, für die eine Lösung überhaupt existiert *(Perceptron Convergence Theorem)*.

1969 veröffentlichten Minsky und Papert ihre Ergebnisse, die sie bei der Verallgemeinerung des Perceptron–Modells auf mehrschichtige Netze erhielten [MIN69]. Sie zeigten, daß das mehrschichtige Perceptron prinzipielle Schwächen aufweist und stellten damit fälschlicherweise seine Limitierung fest. Auf Grund dieser falschen Aussagen wurde das Perceptron–Modell in den Hintergrund des Forschungsinteresses gedrängt. Erst in den 80er Jahren wurden diese Fehleinschätzungen korrigiert.

In den folgenden Jahren konstruierte James A. Anderson zwei wichtige Netzwerkmodelle, die auf der Hebb'schen Regel basieren. 1970 wurde der *lineare Assoziator* vorgestellt, der allerdings nur linear unabhängige Muster fehlerfrei lernen kann [AND70]. Eine Weiterentwicklung ist das 1977 vorgestellte *Brain–State–in–a–Box*–Modell [AND77]. Hier dürfen im Gegensatz zum linearen Assoziator alle Neuronen beliebig miteinander verbunden werden (Rückkopplungen sind somit auch erlaubt).

Stephen Grossberg entwickelte ab Mitte der 70er Jahre eine neuartige Theorie, die sich stärker an der tatsächlichen Funktionsweise des Gehirns orientierte — die *Adaptive Resonance Theory*. Die darauf basierenden Netzwerke arbeiten mit Rückkopplungen, einem Kurz– und Langzeitgedächnis, sowie einem speziellen Aufmerksamkeitsmechanismus, der das Lernen kontrolliert (siehe u.a. [GRO80]).

1982 wurde von John Hopfield ein thermodynamisches Modell vorgestellt, das Parallelen zur theoretischen Physik herstellt [HOP82]. Neuartig ist auch die Generierung der Netzantwort: Ein Neuron berechnet nur dann einen Ausgabewert, wenn es durch einen Algorithmus deterministisch oder probabilistisch ausgewählt wurde. Eine Weiterentwicklung stellt die *Boltzmann–Maschine* dar: Sie wurde von Hinton und Sejnowski 1983 entwickelt und setzt Methoden aus der Theorie der Kristallzüchtung ein [RUM86].

Ein Lernverfahren für das mehrschichtige Percepron, das auch linear nicht separable Muster klassifizieren kann, wurde erstmals 1985 von Rummelhart, Hinton und Williams präsentiert (siehe [RUM86] und Abschnitt 2.2).

Einen Übersichtsartikel zum Thema Neuronale Netze bietet [KEM88]. Weitergehende Informationen findet man in [RUM86], [NAU90] und [ROJ93]. Zur praktischen Erprobung empfiehlt sich [RUM88], da hier auf Disketten Simulationsprogramme für die verschiedenen Modelle mitgeliefert werden.

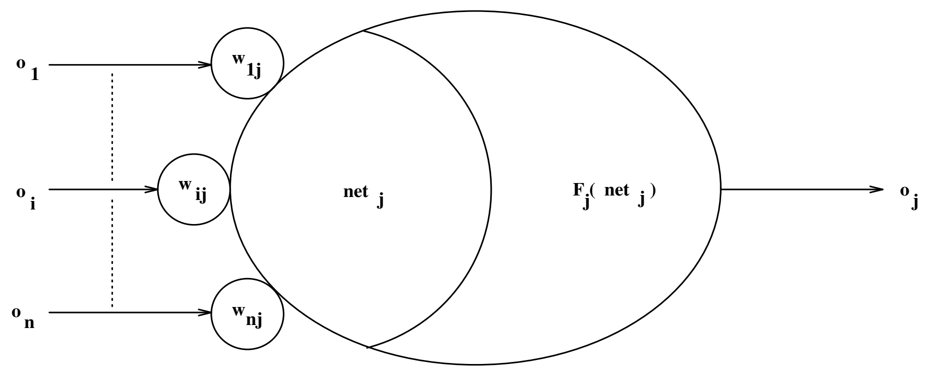
2.2 Das mehrschichtige Perceptron

In diesem Abschnitt soll das mehrschichtige Perceptron, auf dessen Grundlage ProSiNN Neuronale Netze trainiert, eingeführt werden. Ein Neuronales Netz besteht aus drei Komponenten:

- Die elementaren Neuronen
- Die Topologie des Netzes (d.h. der Verbindungsstruktur der Neuronen)
- Die Lernregel, die das Netz an die zu lösende Aufgabe anpaßt

Die ersten beiden Bestandteile werden in den folgenden Unterabschnitten in Beziehung zum mehrschichtigen Perceptron und der hier verwendeten Lernregel vorgestellt. Das Lernverfahren selbst wird anschließend hergeleitet.

2.2.1 Die elementaren Neuronen



$o_1, \ldots, o_n$	:	Eingangswerte (in der Regel von den vorgeschalteten Neuronen)
w_{ij}	:	Synaptisches Gewicht zwischen Neuron i und Neuron j
net_j	:	Propagierungsfunktion des Neurons j
$F_j(net_j)$	:	Aktivierungsfunktion des Neurons j
o_j	:	Ausgabewert des Neurons j

Abbildung 2.3: Das Perceptron–Neuron

Im Bild 2.3 ist ein Neuron eines mehrschichtigen Perceptrons dargestellt. Diese Zelle arbeitet nach folgendem Schema:

- Durch die *Propagierungsfunktion* net_j werden die Eingangssignale des Neurons j auf einen Wert abgebildet (bei den biologischen Neuronen entspricht dies der Übertragung über die synaptischen Kopplung der Zellen).

- Das so entstandene Signal wird der *Aktivierungsfunktion* F_j zugeführt, die den Ausgabewert des Neurons berechnet (das biologische Äquivalent ist die Aktionspotentialbildung).

Die von Rumelhart, Hinton und Williams entwickelte Lernregel für das mehrschichtige Perceptron verlangt, daß alle Neuronen die gleiche Propagierungsfunktion net_j benutzen. Sie wird definiert als:

$$net_j = \sum_i w_{ij} o_i$$

Mathematisch gesehen wird das Skalarprodukt des Eingabevektors mit dem Vektor der synaptischen Gewichte gebildet.

Die Abbildung F_j bestimmt den Aktivierungswert des Neurons j, der gleichzeitig auch der Ausgabewert ist (in anderen Modellen existiert mitunter noch eine zusätzliche Funktion, die den Aktivierungswert — also den aktuellen Zustand der Zelle — auf einen Ausgabewert abbildet). Bild 2.4 zeigt gebräuchliche Aktivierungsfunktionen:

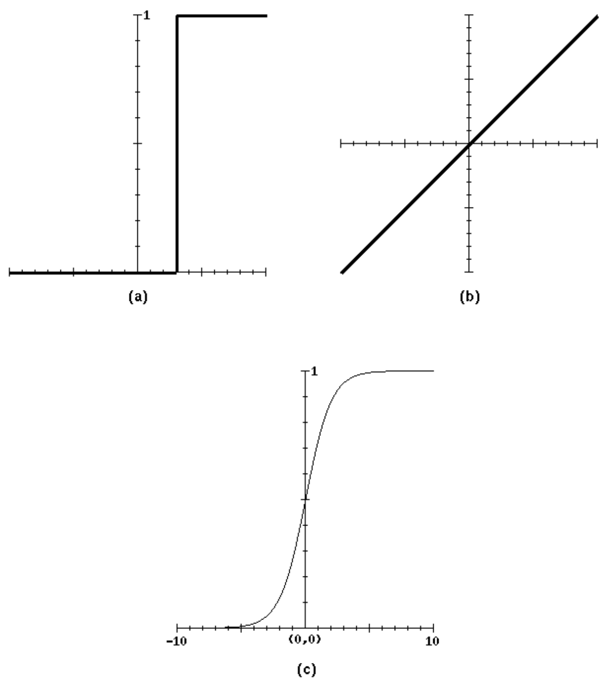


Abbildung 2.4: Verschiedene Aktivierungsfunktionen: (a) Schwellenwertfunktion, (b) lineare und (c) sigmoide, logistische Aktivierungsfunktion

Zur Modellierung der biologischen Nervenzelle kann eine *Schwellenwertfunktion* benutzt werden: Sie ist aber für die später eingeführte Lernregel ungeeignet, da eine differenzierbare Aktivierungsfunktion benötigt wird. Eine weitere einfache Alternative ist die *lineare Funktion*: Mit ihr können aber nur die Probleme bewältigt werden, die auch das einschichtige Perceptron lösen kann (es wird nur ein Matrixprodukt gebildet). Die gebräuchlichste

Aktivierungsfunktion ist die in Bild 2.4 (c) dargestellte *sigmoide (logistische)* Abbildung, die auch ProSiNN defaultmäßig verwendet.

Je nach Motivation des Modells (d.h. wie sich das Neuron verhalten soll) ist eine geeignete Funktion zu wählen. Eine weitere Möglichkeit bestände darin, für jedes Neuron eine eigene Aktivierungsfunktion zu definieren (diese Möglichkeit wird aber von ProSiNN nicht unterstützt).

Durch die Definition der Propagierungsfunktion und der Wahl einer in Abbildung 2.4 dargestellten Aktivierungsfunktion ergeben sich folgende Beziehungen in bezug auf die synaptische Kopplung zweier Neuronen:
Ist das synaptische Gewicht w_{ij} positiv, so reagiert das Neuron j *erregend* auf die Zelle i. Ist w_{ij} dagegen 0, so entspricht dies einer gekappten Verbindung. Eine *hemmende* Reaktion auf das Neuron i entsteht durch einen negativen Wert des Gewichtes w_{ij}.

2.2.2 Die Topologie des Netzes

Das Verhalten von Neuronalen Netzen wird wesentlich durch die Netzwerkarchitektur, die die Verbindungen der Neuronen untereinander repräsentiert, bestimmt. Diese Struktur kann als gerichteter Graph, dessen Kanten bewertet sind, aufgefaßt werden.

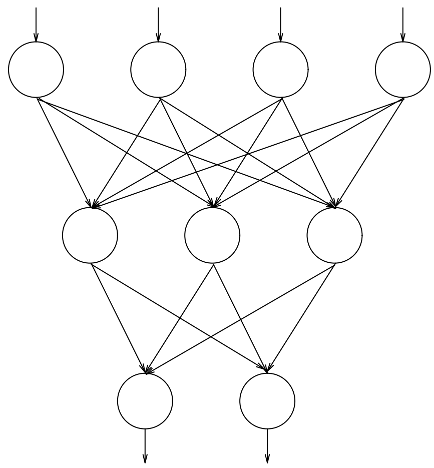
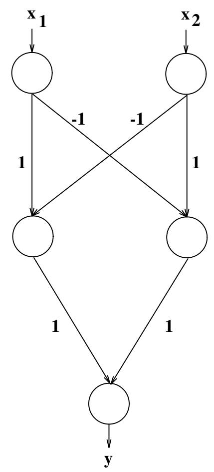


Abbildung 2.5: Eine mögliche Struktur eines Neuronalen Netzes

Bild 2.5 zeigt eine mögliche Verbindungsstruktur: Ein Kreis kennzeichnet hier ein Neuron und eine Kante die Verknüpfung zweier Zellen. Dabei repräsentiert die Richtung einer Kante den Fluß der Informationen und deren Bewertung das synaptische Gewicht zwischen den verbundenen Neuronen. Das mehrschichtige Perceptron hat eine hierarchisch angelegte Struktur, in der das Netz in mehrere Ebenen eingeteilt ist. Die Netzantwort auf ein Eingabemuster wird folgendermaßen generiert: Die Neuronen in der *Eingabeschicht* werden mit den Werten des Musters versorgt. Die von diesen Zellen erzeugten Ausgaben werden an die nächste *innere Ebene (hidden layer)* weitergeleitet. Von dieser Schicht werden erneut die Neuronenausgabewerte gebildet und an die nächste Ebene übergeben. So gelangen die Signale Schicht für Schicht durch das Netz (die Anzahl der inneren Schichten ist beliebig). Nachdem das Muster durch das gesamte Netz propagiert wurde, wird die Netzantwort von den Zellen der *Ausgabeebene* zur Verfügung gestellt. Eine solche Netzstruktur läßt sich als *Black Box* auffassen, die einen Eingabevektor auf einen Ausgabevektor abbildet.

Beim mehrschichtigen Perceptron besteht in der Regel eine Verbindung zwischen jedem Neuron einer Ebene und jeder Zelle der nächsten Schicht. Es handelt sich dann um eine vollständige Vernetzung zwischen zwei Ebenen. Das Lernverfahren von Rumelhart, Hinton und Williams erlaubt auch Verbindungen zwischen Neuronen, die nicht in aufeinanderfolgenden Schichten liegen. Dabei dürfen die Signale nur von der Eingangsebene in Richtung der Ausgangsebene transportiert werden (es sind also keine Rückkopplungen erlaubt): Man spricht hier von *vorwärtsgetriebenen* oder *feed–forward* Netzen. Das Überspringen von einzelnen Schichten wird hier aber nicht weiter betrachtet, da sie keine generelle Leistungssteigerung bewirken und ProSiNN nur eine vollständige Vernetzung zwischen zwei benachbarten Ebenen erlaubt.



x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Abbildung 2.6: Eine Lösung des XOR–Problems mit einem mehrschichtigen Perceptron. Als Aktivierungsfunktion wird eine Schwellenwertfunktion verwendet, die immer dann 1 liefert, wenn ihre Eingabe größer 0 ist

Bild 2.6 zeigt als Beispiel ein mehrschichtiges Perceptron, das die Exklusiv–Oder–Funktion realisiert. Die Eingangswerte des Netzes werden von den Neuronen der ersten Schicht unverändert weitergegeben. Jede Eingabe der folgenden Zellen wird mit dem synaptischen Gewicht der Verbindung multipliziert und anschließend zu einem Wert zusammenaddiert (es handelt sich hier um die im letzten Unterabschnitt definierte Propagierungsfunktion). Ist das Ergebnis größer als 0, so gibt das entsprechende Neuron eine 1 aus; ansonsten wird 0 weitergegeben.

Das Percepton von Rosenblatt konnte diese Aufgabe nicht lösen, da es sich hier um linear nicht separable Muster handelt. Das in Abbildung 2.6 dargestellte Netz läßt sich aber nicht mit dem im nächsten Unterabschnitt hergeleiteten Lernverfahren erzielen, da dafür eine differenzierbare Aktivierungsfunktion benötigt wird. Die Generierung einer durch ProSiNN erhaltenen Lösung wird im Abschnitt 20.1 vorgestellt.

2.2.3 Die Lernregel: Das Backpropagation–Verfahren

Die von Rumelhart, Hinton und Williams 1985 entwickelte Lernregel [RUM86] basiert auf einer mathematischen Optimierungsaufgabe, bei der die synaptischen Gewichte iterativ solange modifiziert werden, bis der Netzausgabefehler aller Muster minimal (bzw. genügend klein) ist. In der Literatur findet man das Verfahren auch unter den Stichworten *Verallgemeinerte Delta–Regel* und *(Error–) Backpropagation.*

Die Grundidee des Optimierungsverfahrens beruht auf folgendem Schema:

Zuerst werden für ein Eingabemuster die Ausgangswerte des Neuronalen Netzes berechnet. Bei jedem Neuron der Ausgabeschicht wird der errechnete Wert mit dem Zielwert verglichen. Aus der Differenz der beiden wird ein Fehlersignal ermittelt. Mit diesem Wert werden die synaptischen Gewichte des jeweiligen Neurons derart modifiziert, das der Ausgabefehler des Neurons minimiert wird. Von der Ausgangsebene bis zur Eingangsebene werden so Schritt für Schritt (für jedes Neuron) die Fehlersignale zurückgerechnet und die entsprechenden Gewichte korrigiert. Dieses Verfahren wird dann für alle restlichen Muster wiederholt und solange fortgesetzt, bis der Gesamtfehler über alle Muster einen genügend kleinen Wert erreicht hat.

Als Fehlermaß E_p für ein Muster p wird die Summe der quadratischen Abweichungen zwischen erhaltenem und vorgegebenem Wert aller Ausgabeneuronen verwendet:

$$E_p = \frac{1}{2} \sum_j (t_{pj} - o_{pj})^2 \tag{2.1}$$

Hierbei ist t_{pj} der verlangte Zielwert des Ausgabeneurons j bei Eingabe des Musters p. o_{pj} beschreibt den tatsächlich erhaltenen Wert, den das Neuron j bei der Eingabe des Musters p ausgibt. Für das Gesamtsystem ergibt sich das Fehlermaß E als die Summe der Fehler über alle Muster:

$$E = \sum_p E_p \tag{2.2}$$

Das Ziel ist die synaptischen Gewichte w_{ij} so zu modifizieren, das E_p bzw. E minimal wird. Das Verfahren mit dem dies gelöst werden soll, ist die *Methode des steilsten Abstiegs (Gradientenverfahren)*:

Betrachtet man z.B. eine Funktion zweier Variablen, so kann man sich diese als ein Gebirge über den beiden Koordinatenachsen vorstellen. Um von einem Startpunkt aus zu einer lokal tiefsten Position des Gebirges zu kommen, wählt man die *brauchbarste* Richtung aus. Dies ist offensichtlich der Weg, der am *steilsten bergab* führt. In diese Richtung *geht* man solange, wie der Weg noch bergab führt. Diese Position wird dann der neue Startpunkt.

Betrachtet man formal eine differenzierbare Funktion $F(\vec{x})$ mehrerer Veränderlicher: Um diese Funktion zu minimieren, benötigt man dessen partielle Ableitungen. Die Richtung des steilsten Gefälles $\vec{g}$ in einem Punkt $\vec{x}_1$ gibt der negative Gradient der Funktion F an: $\vec{g} = -gradF(\vec{x}_1)$. Als neuer Startwert $\vec{x}_2$ wird dann $\vec{x}_2 = \vec{x}_1 + \eta\vec{g}$ verwendet; die Schrittweite η ist dabei geeignet zu wählen. Wenn der Gradient $\vec{g}$ an einer Stelle $\vec{x}_k$ gleich dem $\vec{0}$–Vektor ist, wird das Verfahren beendet (es liegt ein lokales Minimum vor).

Wie man leicht erkennen kann, ist die geeignete Wahl von η entscheidend für die Konvergenz des Verfahrens. Ein weiterer Kritikpunkt betrifft das erhaltene Minimum: Bei dem iterativen Gradientenverfahren besteht die Möglichkeit in einem lokalen Minima *hängenzubleiben*, obwohl ein globales Minima existiert.

Zur Herleitung des Backpropagation–Verfahrens wird die *Methode des steilsten Abstiegs* angewendet: Hierbei ist das Fehlermaß E_p über seinen Variablen w_{ij} zu minimieren, wobei w_{ij} das synaptische Gewicht der Verbindung von Neuron i zu Neuron j repräsentiert. Gleichung 2.3 beschreibt den iterativen Vorgang der Gewichtsmodifizierung für zwei aufeinanderfolgende Zeitschritte n und $n+1$ und dem Muster p als Eingabe (dabei bezeichnen die Zeitschritte die beim Lernen aufeinanderfolgenden Präsentationen des Musters p und die daraus resultierenden temporären Änderungen der Gewichte):

$$\boxed{w_{ij}(n+1) = w_{ij}(n) + \Delta_p w_{ij}(n)} \tag{2.3}$$

$\Delta_p w_{ij}$ ergibt sich durch das Gradientenverfahren zu (die Zeitschritte werden im folgenden wegen der Übersichtlichkeit nicht mehr angegeben):

$$\Delta_p w_{ij} = \eta \left(-\frac{\partial E_p}{\partial w_{ij}} \right) \tag{2.4}$$

Die *Lernrate* η entspricht der Lernschrittgröße. $\left(-\frac{\partial E_p}{\partial w_{ij}}\right)$ ist der negative Gradient des Fehlermaßes E_p: Dies ist die Richtung des steilsten Gefälles, die zur Modifikation des Gewichtes w_{ij} bei Eingabe des Musters p verwendet werden muß.

Zur Erinnerung sei noch einmal die vollständige Definition der Propagierungsfunktion *net* und der Ausgabefunktion o angegeben:

$$net_{pj} = \sum_i w_{ij} o_{pi} \tag{2.5}$$

net_{pj} ist die Propagierungsfunktion, die den Eingabewert, den das Neuron j bei Vorlage des Musters p erhält, berechnet.

$$o_{pj} = F_j(net_{pj}) \tag{2.6}$$

o_{pj} ist der Ausgabewert, den das Neuron j bei Netzeingebe des Musters p erzeugt. F_j ist die Aktivierungsfunktion der Zelle j.

Der in Gleichung 2.4 aufgeführte Term $\left(-\frac{\partial E_p}{\partial w_{ij}}\right)$ kann durch die Kettenregel als ein Produkt zweier Ableitungen geschrieben werden:

$$-\frac{\partial E_p}{\partial w_{ij}} = -\frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ij}} \tag{2.7}$$

Der erste Faktor beschreibt dabei die Änderung des Fehlermaßes E_p bei einer Variation des Propagierungswertes net_{pj}. Der andere charakterisiert dagegen die Veränderung der

Eingabe net_{pj} für eine Einheit j, die durch eine Modifikation des Gewichtes w_{ij} erzeugt wird.

Die Ableitung $\frac{\partial net_{pj}}{\partial w_{ij}}$ läßt sich leicht aus der Gleichung 2.5 bestimmen:

$$\frac{\partial net_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{kj} o_{pk} = o_{pi} \tag{2.8}$$

Der negative Faktor $-\frac{\partial E_p}{\partial net_{pj}}$ wird auch als das Fehlersignal δ_{pj} des Neurons j bei Eingabe des Musters p bezeichnet:

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} \tag{2.9}$$

Das Fehlersignal δ_{pj} läßt sich mit Hilfe der Kettenregel als Produkt zweier Faktoren schreiben:

$$\delta_{pj} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}} \tag{2.10}$$

Die erste Ableitung beschreibt die Veränderung des Fehlersmaßes bei einer Modifikation der Ausgabe des Neurons j. Der andere Faktor charakterisiert die Änderung der Ausgabe bei einer Variation der Neuroneneingabe.

Für den zweiten Term gilt nach Gleichung 2.6:

$$\frac{\partial o_{pj}}{\partial net_{pj}} = F'_j(net_{pj}) \tag{2.11}$$

Um den ersten Faktor der Gleichung 2.10 zu berechnen, muß eine Fallunterscheidung durchgeführt werden. Dabei wird zwischen einem Neuron der Ausgabeschicht und einer inneren Zelle unterschieden.

Zuerst sei das Neuron j ein Element der Ausgabeebene. Nach Definition des Fehlermaßes in Gleichung 2.1 kann diese Ableitung zu folgender Formel berechnet werden:

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) \tag{2.12}$$

Damit ergibt sich im Fall eines Ausgabeneurons das Fehlersignal δ_{pj} durch Einsetzen von 2.11 und 2.12 in Gleichung 2.10 zu:

$$\delta_{pj} = (t_{pj} - o_{pj}) F'_j(net_{pj}) \tag{2.13}$$

Im zweiten Fall sei nun das betrachtete Neuron eine Zelle einer inneren Ebene. Die Fehlermaßvariation $\frac{\partial E_p}{\partial o_{pj}}$ der Einheit j ergibt sich mit Hilfe der Kettenregel zu einer Summe der Fehlermaßänderungen der nachfolgenden Neuronen (also aller Zellen, die über eine Verbindung Eingaben der Einheit j bekommen):

$$\frac{\partial E_p}{\partial o_{pj}} = \sum_k \left(\frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pj}} \right)$$

$$
\begin{aligned}
&= \sum_k \left(\frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ik} o_{pi} \right) \\
&= \sum_k \left(\frac{\partial E_p}{\partial net_{pk}} w_{jk} \right) \\
&= -\sum_k \delta_{pk} w_{jk}
\end{aligned}
$$

Zusammengefaßt ergibt sich also für ein Neuron der inneren Ebene:

$$\frac{\partial E_p}{\partial o_{pj}} = -\sum_k \delta_{pk} w_{jk} \tag{2.14}$$

Setzt man nun 2.11 und 2.14 in Gleichung 2.10 ein, so erhält man für das Fehlersignal δ_{pj} eines inneren Neurons:

$$\delta_{pj} = F_j'(net_{pj}) \sum_k \delta_{pk} w_{jk} \tag{2.15}$$

Durch die beiden Gleichungen 2.13 und 2.15 ergibt sich ein rekursives Verfahren, mit dem die Fehlersignale δ_{pj} aller Neuronen berechnet werden können.

Nun sind alle benötigten Gleichungen für das Lernverfahren hergeleitet worden. Die zentralen Gleichungen seien in der folgenden Formelsammlung noch einmal dargestellt:

$$\boxed{\begin{aligned}
\Delta_p w_{ij} &= \eta \delta_{pj} o_{pi} \\
\delta_{pj} &= \begin{cases} (t_{pj} - o_{pj}) F_j'(net_{pj}) & , \text{ für ein Ausgabeneuron} \\ F_j'(net_{pj}) \sum_k \delta_{pk} w_{jk} & , \text{ für ein inneres Neuron} \end{cases}
\end{aligned}} \tag{2.16}$$

Zur Realisierung eines Gradientenabstiegs in E dürfen die Gewichtsänderungen erst dann vorgenommen werden, wenn alle Muster dem Netz präsentiert worden sind. Da hier die Gewichte nach jedem Muster modifiziert werden, wird von der Methode des steilsten Abstiegs abgewichen. Um dies zu kompensieren, muß die Lernrate η genügend klein gehalten werden. In diesem Fall wird der Gradientenabstieg gut approximiert und das Verfahren kann eine Lösung finden. Ein kleines η verringert allerdings die Lerngeschwindigkeit; wird η dagegen zu groß gewählt, so können beim Lernen Oszillationen auftreten (Fehlermaß E schwingt).

Um die Lernrate trotzdem vergrößern zu können (und damit die Lerngeschwindigkeit zu erhöhen), wird $\Delta_p w_{ij}$ in Gleichung 2.16 um einen Summanden erweitert. Dieser zweite Term beschreibt die Auswirkungen, die vergangene Gewichtsänderungen zur aktuellen Gewichtsmodifikation haben sollen: Damit werden hochfrequente Schwankungen von E herausgefiltert.

$$\Delta_p w_{ij}(n) = \eta \delta_{pj} o_{pi} + \alpha \Delta_p w_{ij}(n-1) \tag{2.17}$$

Hier steht $n-1$ für die letzte und n für die aktuelle Gewichtsänderung. Das sogenannte *Lernmoment* α ist eine Konstante, die darüber entscheidet, in welchem Maß vergangene Gewichtsänderungen mit einbezogen werden sollen.

Das i.allg. verwendete Backpropagationverfahren ergibt sich damit zu:

$$\boxed{\begin{array}{rcl} \Delta_p w_{ij}(n) & = & \eta\delta_{pj}o_{pi} + \alpha\Delta_p w_{ij}(n-1) \\ \delta_{pj} & = & \left\{ \begin{array}{ll} (t_{pj} - o_{pj})F'_j(net_{pj}) & , \text{ für ein Ausgabeneuron} \\ F'_j(net_{pj})\sum_k \delta_{pk}w_{jk} & , \text{ für ein inneres Neuron} \end{array} \right. \end{array}} \tag{2.18}$$

Die Lernregel wird in der Praxis folgendermaßen angewendet:

- Der Benutzer muß vor Beginn des Trainings die Parameter η und α festlegen. Danach gliedert sich das Training für jedes Muster in zwei Phasen:
  - Zuerst wird den Neuronen der Eingabeschicht ein Eingabemuster präsentiert. Diese Zellen berechnen die dazugehörigen Aktivierungswerte und geben sie an die nächste Ebene weiter. So gelangen die generierten Werte von Schicht zu Schicht, bis die Ausgabewerte des Netzes erzeugt wurden.
  - Im zweiten Schritt wird das Fehlersignal δ_{pj} der Ausgabeneuronen mit dem zum Eingabemuster gehörenden Zielmuster und der Gleichung 2.13 berechnet. Mit diesen Werten werden die neuen Gewichtsänderungen $\Delta_p w_{ij}$ bestimmt (die dazugehörigen Gewichte werden aber noch nicht modifiziert). Nun werden die Fehlersignale der davorliegenden Ebene (vor der Ausgabeschicht) mit der Formel 2.15 ermittelt. Dies wird von Ebene zu Ebene solange durchgeführt, bis man an der Eingabeschicht angelangt ist. Durch diese Vorgehensweise wird das Fehlersignal von der Ausgabe– bis zur Eingabeschicht zurückpropagiert. Anschließend werden die Gewichtsmodifikationen mit der Gleichung 2.3 durchgeführt.
- Diese zwei Phasen werden für alle Muster solange wiederholt, bis das Gesamtfehlermaß E eine vorgegebene Grenze unterschreitet.

2.3 Das modifizierte Backpropagation–Verfahren

In diesem Abschnitt soll die Lernregel für das mehrschichtige Perceptron vorgestellt werden, auf deren Grundlage ProSiNN Neuronale Netze trainiert. Dieses Verfahren wurde von T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink und D.L. Alkon 1988 entwickelt, um die Konvergenz der verallgemeinerten Delta–Regel zu beschleunigen [VOG88]. Dabei basierten ihre Überlegungen auf praktischen Erfahrungen mit anderen nicht–linearen Optimierungsverfahren.

Die Motivation für eine Weiterentwicklung ergibt sich aus den Nachteilen, die der konventionelle Backpropagation–Algorithmus enthält:

- Da nach jeder Musterpräsentation (und der entsprechenden Fehlerpropagierung) die Gewichte modifiziert werden, liegt real kein Gradientenabstieg vor. Nur durch eine geeignete Wahl von η kann die Methode des steilsten Abstiegs approximiert werden.

- Ein Änderungsschritt ($\Delta_p w_{ij}$ berechnen und die entsprechenden Gewichte modifizieren), der das Fehlermaß E in bezug auf *ein bestimmtes* Muster minimiert, wird i.allg. kein Neuronales Netzwerk erzeugen, das den Fehler auch für alle anderen noch zu lernenden Muster reduziert: Eine durch ein bestimmtes Muster hervorgerufene Gewichtsmodifizierung kann den Optimierungspfad im Fehlermaßraum in eine falsche Richtung lenken. Daraus resultiert eine Erhöhung der Anzahl der Trainingsdurchläufe für eine Konvergenz des Verfahrens (sofern eine Lösung überhaupt gefunden wird).

- Die Konvergenz des Backpropagation–Verfahrens hängt von der Wahl einer geeigneten Lernrate η ab: Ein für ein Problem gewähltes η muß bei einer anderen Trainingsaufgabe nicht zur Konvergenz und damit zum Lernerfolg führen.

  Die geeignete Lernrate η und das gesamte Verfahren sind abhängig von der Gestalt des Fehlermaßes E, das man sich als hyperdimensionalen Raum, der über den Gewichten w_{ij} aufgetragen ist, vorstellen kann. In diesem Raum wird ein Weg zum (globalen) Minimum gesucht. Wenn das Lernverfahren auf seinem Optimierungsweg z.B. an einem steilen V–förmigen Tal ankommt, kann folgendes passieren:
  Ist die Lernrate η zu groß gewählt worden, besteht die Möglichkeit, daß der Optimierungsweg über das Tal *hinausspringt* und das Minimum nicht erkannt wird (oder es kommt zu Oszillationen auf den Talrändern: Der Weg *springt* von einer Talseite auf die andere usw.). Auf der anderen Seite wird bei einem zu kleinen η der Lernerfolg lange auf sich warten lassen.

  Man muß also eine geeignete Lernrate η finden, die den beiden Extremfällen entgegenwirkt. In der Praxis sollte η gerade so groß gewählt werden, daß keine Oszillationen auftreten. Ein Problem bleibt aber, daß ein gewähltes η zu einem Zeitpunkt optimal sein kann, während es sich später vielleicht negativ auf die Konvergenz auswirkt.

Um die Mängel des konventionellen Backpropagation–Verfahrens auszuschalten, werden die folgenden zwei Änderungen durchgeführt:

1. Die Gewichtsmodifikationen werden erst dann ausgeführt, wenn jedes Muster dem Neuronalen Netz präsentiert worden ist:

   Nachdem ein Muster durch das Netz propagiert wurde, werden mit der verallgemeinerten Delta–Regel die Gewichtskorrekturen $\Delta_p w_{ij}$ berechnet. Mit diesen Änderungen werden die Gewichte aber erst dann modifiziert, wenn für jedes Muster $\Delta_p w_{ij}$ errechnet wurde. Über alle Muster und für jedes Gewicht wird die Änderung $\Delta_p w_{ij}$ aufsummiert, wobei p die Anzahl der Summanden (d.h. der Muster) ist. Nach der Summation werden die Gewichtsmodifikationen ausgeführt. D.h., alle Gewichte werden erst dann geändert, wenn die Gesamtrichtung des Optimierungsweges feststeht. In den folgenden Formeln ist das vollständige modifizierte Verfahren noch einmal angegeben:

$$w_{ij}(m+1) = w_{ij}(m) + \Delta w_{ij}(m) \tag{2.19}$$

$$\Delta w_{ij}(m) = \eta \sum_p (\delta_{pj} o_{pi}) + \alpha \Delta w_{ij}(m-1) \tag{2.20}$$

Hierbei beschreibt p die Anzahl der zu lernenden Muster und m repräsentiert die Anzahl der Durchläufe über alle Muster, die im folgenden als ein Lernschritt oder ein *Zyklus* bezeichnet wird.

2. Die Lernrate η wird in Abhängigkeit davon variiert, ob durch einen Lernzyklus das Fehlermaß verkleinert oder vergrößert wurde:

   (a) Bei einer Verringerung des Fehlermaßes zum vorhergehenden Wert wird die Lernrate η mit einem Faktor $\Phi > 1$ multipliziert (die Lernschritte werden vergrößert). Dieses neue η ist dann im nächsten Zyklus gültig. Im Anschluß werden die berechneten Gewichtsänderungen ausgeführt.

   (b) Bei einer Vergrößerung des Fehlermaßes um ein paar Prozent zum vorhergehenden Wert (typisch sind ca. 1–5%), wird die Lernrate η mit einem Faktor $0 < \beta < 1$ multipliziert (die Lernschritte werden verkleinert). Alle Gewichtsänderungen, die vor diesem Zyklus durchgeführt worden sind, werden wieder rückgängig gemacht. Außerdem wird das Lernmoment α auf 0 gesetzt. Im Anschluß wird der gesamte Zyklus noch einmal durchlaufen. Wenn ein erfolgreicher Lernzyklus stattgefunden hat, wird α wieder auf seinen Ursprungswert gesetzt.

   Bem.:
   Wenn die über den Gewichten aufgetragene Oberfläche des Fehlermaßes relativ glatt und gleichförmig verläuft, hilft die Einbeziehung der letzen Gewichtsänderung über das Lernmoment α der Konvergenz des Verfahrens. Wenn aber ein Lernzyklus das Fehlermaß erhöht, dann verlangt die Minimierung dieses Maßes eine Änderung des Optimierungsweges. Dabei würde die vorhergehende Gewichtsänderung mehr irreführend als nützlich sein. Durch Setzen von α auf 0 gehen diese *Erinnerungen* verloren.

   (c) Wenn nach einem Lernzyklus das Fehlermaß gleich geblieben ist oder sich nur wenig verschlechtert hat (unterhalb der Prozentangabe, die in (b) definiert wurde), werden alle neu berechneten Gewichtsänderungen gültig gemacht. Die Lernrate η und das Lernmoment α behalten ihre bisherigen Werte.

Vergleiche zwischen der verallgemeinerten Delta–Regel und dem modifizierten Backpropagation–Verfahren, die in [VOG88] vorgenommen worden sind, versprechen i.allg. eine deutliche Reduzierung der Lernzyklen.

2.4 Training mit dynamischer Neuronenerzeugung

In diesem Abschnitt soll ein Verfahren vorgestellt werden, mit dem es möglich ist, die Topologie eines Netzes beim Training dynamisch zu verändern. Motiviert wird dies zum einen durch die Wichtigkeit der Netzarchitektur beim Lösen einer Lernaufgabe und zum anderen durch die Möglichkeit ein interpretierbares Programm durch das ProSiNN–System ausführen zu lassen (Kapitel 21 enthält eine mögliche Aufgabenstellung und das Simulationsprogramm, das das dynamische Verfahren zur Verfügung stellt).

Durch das Backpropagation–Verfahren werden nur die synaptischen Gewichte modifiziert, während die Topologie aber unverändert bleibt. Betrachtet man das biologische Gehirn als Vorbild der Lernregel, so widerspricht dies seiner Vorgehensweise: Die Gehirnstrukturen werden teilweise im Laufe der Zeit geändert (Verbindungen werden gekappt und neue erzeugt). Ein anderer Aspekt betrifft die Topologie in bezug auf die zu bewältigende Lernaufgabe. Hierbei wäre ein mathematisches Verfahren wünschenswert, das eine minimale Netzwerkstruktur zu einer vorgegebenen Aufgabenstellung liefert. Ein solches Verfahren existiert aber nicht.

Bem.:

Es gibt allerdings theoretische Untersuchungen, die wenigstens eine Einschätzung der Topologie liefern:

Betrachtet man z.B. ein mehrschichtiges Perceptron, das neben der Ein– und Ausgabeschicht noch eine innere Ebene enthält, so gibt es die folgende Vermutung:
Ist T die Anzahl der zu trainierenden Muster, so sollte die Anzahl der Neuronen in der mittleren Schicht H in der Ordnung von $\log_2 T$ und $(T-1)$ liegen.

In [MIR89] leiten die Autoren ein Theorem her, das die maximale Anzahl der linear separablen Regionen im Eingaberaum berechnet:
Es sei d die Anzahl der Neuronen in der Eingangsschicht und H die Anzahl der Neuronen in der mittleren Ebene. M sei die maximale Anzahl der linear separablen Regionen im Eingaberaum (der d–dimensionale Eingangsraum ist in M linear separable Räume einteilbar, wenn M disjunkte Regionen existieren, deren Grenzen den Raum in Hyperebenen zerlegen). Dann gilt:

$$M(H,d) = \sum_{k=0}^{d} \binom{H}{k}, \text{ wobei } \binom{H}{k} = 0, \text{ wenn } H < k$$

Für die Praxis bringt dieses Theorem i.allg. keinen Nutzen, da zur Bestimmung der Quantität der inneren Neuronen die Anzahl der linear separablen Räume bekannt sein müßte: Diese Größe kann aber durch keine geschlossene Formel ermittelt werden.

Wird die Topologie eines Neuronalen Netzes zu klein gewählt, so kann die vorgegebene Lernaufgabe nicht mit genügender Genauigkeit gelöst werden. Ist die Netzstruktur dagegen zu groß, leidet die Generalisierungfähigkeit des Neuronalen Netzes darunter. Deshalb sollte ein Netzwerk gerade groß genug sein, um die Lernaufgabe bewältigen zu können, und klein genug, um die Abstraktionsfähigkeit nicht zu verringern.

Es gibt grundsätzlich zwei Verfahren, die die Netzwerktopologie beim Lernen dynamisch verändern:

1. Als Initialisierungsnetz wird eine Struktur gewählt, die so groß ist, daß die Lernaufgabe bewältigt werden kann. Anschließend wird die Topologie solange reduziert, bis eine minimale Struktur gefunden wird: Dabei entfernt man z.B. Verbindungen, deren synaptische Gewichte nahezu 0 sind (und wenn möglich auch die entsprechenden Neuronen). Nach diesem Schritt wird das Netz wieder trainiert und es werden evtl. weitere Verbindungen entfernt usw.

2. Bei der zweiten Alternative wird zu Beginn eine kleine Netztopologie gewählt. Im Verlauf des Trainings werden solange neue Neuronen und deren Verbindungen hinzugefügt, bis die Lernaufgabe bewältigt werden kann. An dieses Verfahren gibt es grundsätzlich zwei Annäherungen:

   (a) Nachdem mindestens ein Neuron hinzugefügt wurde, werden nur die neuen Zellen trainiert. Der Rest des Netzes wird praktisch *eingefroren*.

   (b) Nachdem die Netzstruktur vergrößert wurde, wird das gesamte vorliegende Netz trainiert.

Das erste Verfahren hat den grundsätzlichen Nachteil, daß beim Trainieren einer großen Netzstruktur viel Speicherplatz und Rechenzeit verbraucht wird.

(2.a) bietet den Vorteil des geringen Trainingsaufwandes, da das Lernverfahren nur neu hinzukommende Neuronen bearbeitet. Es kann i.allg. aber nicht die bestmögliche Lösung erzielt werden (sofern überhaupt eine gefunden wird), da der Optimierungsweg beim Durchlaufen des Gewichtsraumes auf bestimmte Pfade eingeschränkt ist (der größte Teil des Raumes ist mit seinen aktuellen Werten eingefroren).

Motiviert durch die genannten Nachteile, entwickelte T. Ash 1989 eine Methode, die auf der in (2.b) genannten Idee beruht [ASH89]. Bei dem sogenannten*DNC*–Verfahren (**D***ynamic* **N***ode* **C***reation*) werden in den inneren Ebenen neue Neuronen hinzugefügt. Die zugrundeliegende hierarchische Netzwerkarchitektur besteht aus drei Ebenen. Dies stellt aber keine Einschränkung dar: Alle Aufgaben, die Netze mit größerer Ebenenanzahl lösen können, kann auch das dreischichtige Perceptron mit beliebiger Genauigkeit und einer endlichen Anzahl von Neuronen in der mittleren Ebene bewältigen ([HEC89] und [HOR89]). Netzwerke mit mehr als drei Schichten sind in der Praxis aber oft kleiner und konvergieren damit schneller.

Eine neue Zelle wird dann hinzugefügt, wenn der Verlauf des Fehlermaßes E zu *glatt* wird (d.h. die Konvergenz des Verfahrens wird langsamer).

In Abbildung 2.7 ist ein möglicher Verlauf des Fehlermaßes angegeben: Nachdem zur Zeit t_0 ein Neuron hinzugefügt worden ist, wird das Verhältnis der Fehlermaßdifferenz, die sich durch die Zeitpunkte $t - w$ und t ergibt (w bestimmt dabei die Größe des verwendeten Fensters), mit dem Fehlermaß, daß bei t_0 vorlag, gebildet. Liegt dieser Wert unter einer vorgegebenen Schranke Δ_T, so wird ein weiteres Neuron in die innere Ebene eingefügt. Formal wird das Netz um eine Zelle erweitert, wenn gilt:

$$\frac{E(t-w) - E(t)}{E(t_0)} < \Delta_T, \text{ wobei } \Delta_T > 0 \text{ und } t - w \geq t_0 \tag{2.21}$$

Wenn das berechnete Verhältnis über Δ_T liegt, wird kein neues Neuron hinzugefügt; $t - w$ wird dann auf t gesetzt und das Verfahren wird wiederholt. Sinkt E unter die vorgegebene Fehlermaßschranke, so wird der Algorithmus beendet.

Für die Leistungsfähigkeit dieser dynamischen Methode ist eine geeignete Wahl von Δ_T und w entscheidend: Fällt z.B. die Kurve von E relativ flach (Backpropagation liefert oft über lange Zeit stagnierende Fehlermaße), so wird bei einem zu kleinen w ein neues Neuron hinzugefügt, das für die Lösung vielleicht gar nicht gebraucht worden wäre.

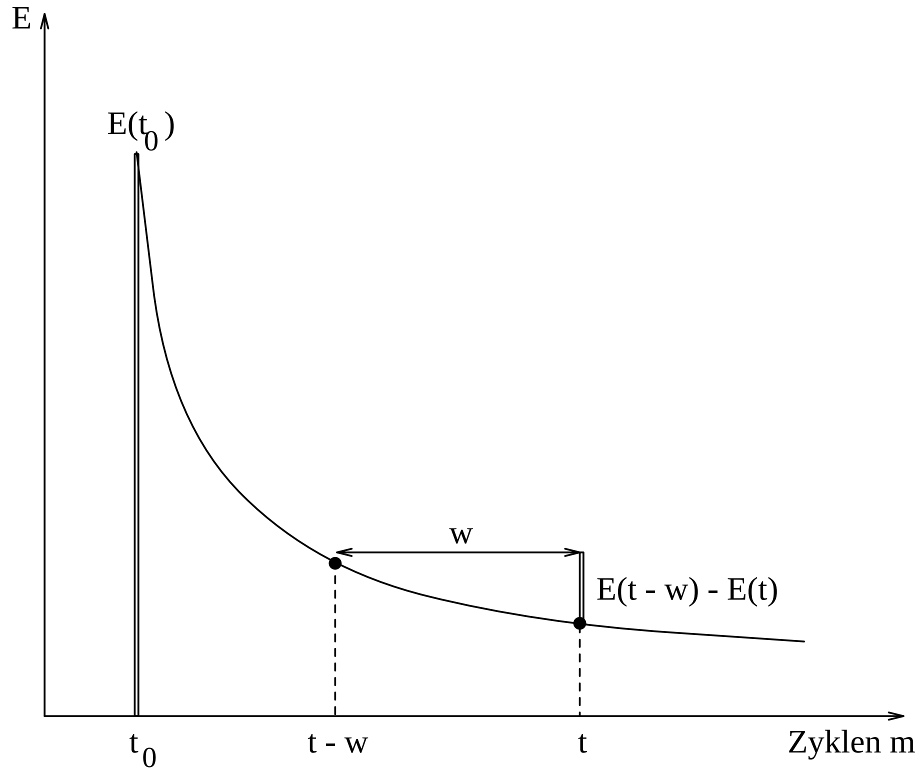


Abbildung 2.7: Das Fehlermaß eines möglichen dynamischen Trainingsverlaufes

Wird beim Lösen einer Aufgabe eine relativ große Netztopologie erwartet, so kann die Startanzahl der Neuronen in der mittleren Ebene auch auf einen höheren Wert gesetzt werden; eine weitere Möglichkeit besteht darin, daß anstatt einer mehrere Zellen hinzufügt werden.

Das reine Backpropagation–Verfahren ist i.allg. schneller als die DNC–Methode, dafür liefert der DNC–Algorithmus aber zu einem unbekannten Problem zusätzlich noch die benötigte Netzarchitektur — sofern überhaupt eine Lösung gefunden werden kann.

Kapitel 3

Einführung in das Programmsystem ProSiNN

In diesem Kapitel werden die wesentlichen Grundlagen, die der Anwender zum Bedienen des Systems benötigt, erläutert. Einleitend dazu werden einige zentrale Merkmale des Programmsystems vorgestellt. Anschließend wird auf das Modell, das ProSiNN für Neuronale Netze verwendet, eingegangen. Der Systemstart stellt dann die erste Verbindung zur Praxis her. Den Abschluß dieses Kapitels bildet die Beschreibung des Umgangs mit der grafischen Oberfläche von ProSiNN.

3.1 Das ProSiNN–System im Überblick

Mit ProSiNN wird ein Werkzeug zur Verfügung gestellt, das insbesondere zur Erzeugung Neuronaler Netze, zu dessem Training und zur Untersuchung der Netzstrukturen entwickelt wurde. Der Benutzer kann auf eine Vielzahl von Möglichkeiten zurückgreifen. Neben einer komfortablen grafischen Oberfläche bietet das System eine Programmiersprache (die im folgenden auch als **ProSiNN** bezeichnet wird), die durch einen Interpreter bearbeitet wird. Mit dieser universellen Sprache können vielfältige Programmieraufgaben gelöst werden.

Das System stellt dem Anwender zwei Schnittstellen zur Verfügung:

- Ein Kommandozeileninterpreter bildet die erste Grundlage des Systems. Mit dessen Hilfe kann der Benutzer sämtliche Anweisungen der Programmiersprache ProSiNN ausführen. Die eingegebenen Befehle und die vorgenommenen Ausgaben werden in einem eigenen Teilfenster protokolliert. Dadurch können alle vorgenommenen Bearbeitungsschritte immer überblickt werden.

- Zur leichteren Bedienung wird eine Menüoberfläche zur Verfügung gestellt. Fast alle Interpreterkommandos lassen sich so bequem über die Maussteuerung ausführen. Darüber hinaus erlaubt das Menüsystem weitergehende Möglichkeiten (z.B. Aufruf eines Editors).

Im folgenden werden nun die wichtigsten Merkmale des ProSiNN–Systems im Überblick dargestellt.

- ProSiNN bietet vielfältige Möglichkeiten zur Bearbeitung von Neuronalen Netzen:
  - Ein Netz kann mit beliebiger Ebenenanzahl erzeugt werden. In jeder dieser Schichten ist die Anzahl der Neuronen nur von der Größe des vorhandenen Primärspeichers abhängig.
  - Es lassen sich dynamisch Netzebenen entfernen und hinzufügen. Auch die Anzahl der Neuronen in einer Schicht ist durch den Benutzer nachträglich veränderbar.
  - Netzinterne Werte (synaptische Gewichte u.ä.) sind beliebig manipulierbar. Dies kann bequem über das Menüsystem oder durch einen Interpreterbefehl durchgeführt werden.
  - Neuronale Netze können auf zwei Arten gespeichert werden. Zum einen in wieder einlesbarer (und kommentierter) Form und zum anderen als C–Programm. Die erste Alternative bietet den Vorteil, daß Netzstrukturen archiviert und weiter bearbeitet werden können. Die zweite Form ist für den praktischen Einsatz eines bereits vollständig trainierten Netzes gedacht.
- Alle zum Trainieren und Testen des Netzes notwendigen Muster müssen aus einer Datei eingelesen werden (mit Hilfe des Menüsystems können die Muster auch modifiziert werden). Es existieren zwei Musterarten: Das Trainings– und das Testmuster. Die Trainingsmuster repräsentieren die Aufgabe, die ein Neuronales Netz lösen soll. Mit einem Testmuster kann dagegen nur die Netzantwort eines Eingabemusters berechnet werden.
- Zum Erlernen einer Aufgabenstellung existieren einige Features, die das Training erleichtern und für den Anwender transparent machen:
  - Beim Netztraining können entweder alle oder auch nur ein bestimmtes Muster als Lernaufgabe verwendet werden. Mit der zweiten Alternative ist es möglich, ein Netz besser an ein spezielles Muster anzupassen oder über die Programmiersprache ProSiNN ein Programm zu entwickeln, das einen alternativen Backpropagationalgorithmus benutzt (im Gegensatz zum verwendeten Standardverfahren).
  - In Abhängigkeit der Trainingszyklen sind beliebige Parameter protokollierbar. Diese Werte lassen sich auch in Dateien umdirigieren und können im Anschluß zu einer Trainingsanalyse verwendet werden.
Eine weitere Möglichkeit besteht in der grafischen Ausgabe dieser Werte. So kann der Trainingsverlauf optisch *auf einen Blick* verfolgt werden.
  - Die Aktivierungsfunktion ist durch den Benutzer manipulierbar. Damit kann im Einzelfall eine Konvergenzbeschleunigung oder eine bessere Lösung erzielt werden.
- ProSiNN liefert eine Anzahl verschiedener Variablentypen, die die Programmiersprache für den Anwender universal macht. Dazu gehören Variablen, die ganze Zahlen

und Gleitkommawerte (sowie Felder auf diesen Typen), Funktionen, Zeichenketten, Befehlsmakros und netzinterne Werte beschreiben. Für diese Typen existieren eine Reihe von Kommandos, die das weitere Verarbeiten der Variablen ermöglichen (z.B. die symbolische Ableitung einer Funktion erzeugen).

- Der Anwender kann mit bestimmten Befehlen (oder dem Menüsystem) grafische Ausgaben generieren. Dazu gehören die Erzeugung und das Löschen von Grafikfenstern, sowie Kommandos, die Achsenkreuze, Linien, Texte, Punkte oder ganze Funktionen in eine Grafik einzeichnen.

- Ein eigener Editor vereinfacht die Generierung von Netz–, Muster– und Programmdateien.

Die hier genannten Punkte sollten einen kleinen Einblick in die vielfältigen Möglichkeiten des Programmsystems geben. ProSiNN bietet darüber hinaus noch eine Reihe weiterer Charaktere, die jeweils beim entsprechenden Kontext eingeführt und erläutert werden.

3.2 Neuronale Netze im ProSiNN–System

ProSiNN verwendet ein Modell für Neuronale Netze, das sich in einigen Punkten von dem in Abschnitt 2.2 dargestellten mehrschichtigen Perceptron unterscheidet. Diese Differenzen werden im folgenden beschrieben.

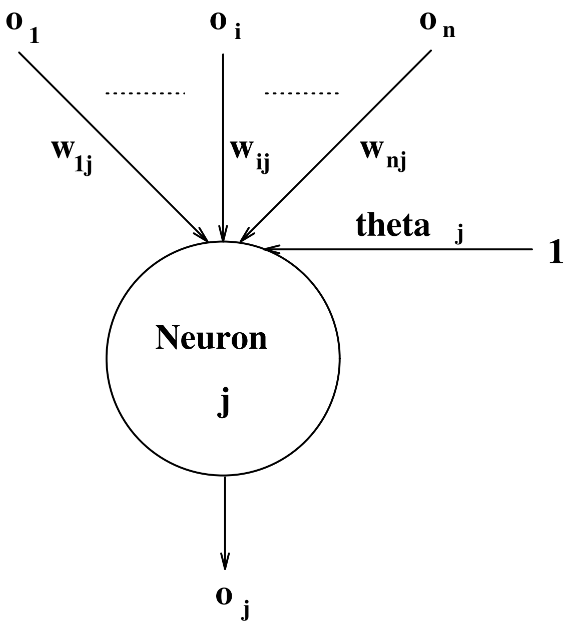


Abbildung 3.1: Das ProSiNN–Neuron

- ProSiNN benutzt ein leicht modifiziertes Neuron. Ein Modell dieser Zelle ist in Abbildung 3.1 angegeben. Zusätzlich zu den Eingabewerten, enthält das ProSiNN–Neuron noch einen Schwellenwert θ_j (in Bild 3.1 als `thetaj` bezeichnet), der die

Aktivierungsfunktion um θ_j auf der Abszissenachse verschiebt. Beim Training wird dieser Verschiebungswert genau wie ein Gewicht *erlernt*. Formal ergibt sich damit die Propagierungsfunktion zu:

$$net_j = \left(\sum_{i=1}^{n} w_{ij} o_i \right) + \theta_j \tag{3.1}$$

Der Schwellenwert θ_j ist normalerweise ein Bestandteil der Zelle j. Er ist in Abbildung 3.1 außerhalb der Zelle eingezeichnet worden, damit die folgende Erläuterung besser nachvollziehbar ist: Beim Backpropagationverfahren wird die Schwelle als ein zusätzliches Gewicht betrachtet, dessen vorgeschaltetes Neuron immer *aktiv* ist (als *aktiv* wird hier ein Neuron bezeichnet, das eine 1 als Ausgabe liefert). Dadurch können Schwellenwertmodifikationen genau wie Gewichtsänderungen berechnet werden [RUM86]. Mit der Festlegung $w_{n+1,j} = \theta_j$ und $o_{m+1} = 1$ läßt sich diese Aussage leicht überprüfen:

$$net_j = \sum_{i=1}^{n+1} w_{ij} o_i = \left(\sum_{i=1}^{n} w_{ij} o_i \right) + \theta_j$$

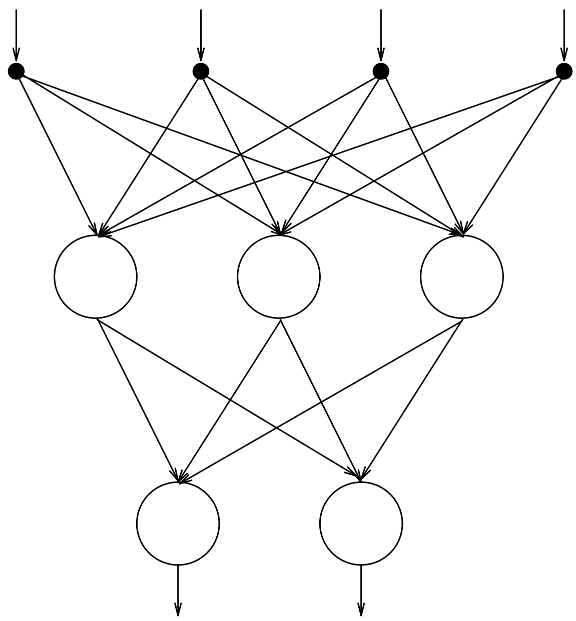


Abbildung 3.2: Ein ProSiNN–Netz mit drei Ebenen

- In Bild 3.2 ist eine mögliche Netztopologie dargestellt (im ProSiNN–System wird dieses Netz als dreischichtig bezeichnet). Die Eingaben aller Neuronen der ersten Ebene (sie sind hier als *schwarz gefüllte Kreise* eingezeichnet worden) werden ohne Änderung wieder ausgegeben. Durch die zweite Schicht gelangen die Werte dann wie in Abschnitt 2.2 beschrieben. Die Netzeingabewerte werden mit den Gewichten multipliziert und zusammenaddiert. Im Anschluß wird die Schwelle dazuaddiert (siehe Gleichung 3.1). So gelangen die Eingangsinformationen Ebene für Ebene durch das Netz.

Dieser Modelltyp verlangt mindestens zwei Netzschichten. In diesem Fall existiert *real* nur eine Ebene mit Neuronen, die die Eingabewerte modifizieren.

Es sei noch einmal erwähnt, daß ProSiNN nur eine vollständige Vernetzung zwischen zwei Schichten erlaubt. D.h. von jedem Neuron einer Ebene geht eine Verbindung zu jeder Zelle der nächsten Schicht.

- Als Aktivierungsfunktion benutzt ProSiNN die in Abbildung 2.4(c) dargestellte sigmoide (logistische) Funktion. Formal wird sie durch die folgende Gleichung beschrieben:

  $$o_j = \frac{1}{1 + \exp(-net_j)} \tag{3.2}$$

  Damit ergibt sich der Ausgabewert für das Neuron j zu:

  $$o_j = \frac{1}{1 + \exp\left(-\left\{\sum_{i=1}^{n} w_{ij} o_i + \theta_j\right\}\right)} \tag{3.3}$$

  Bem.:

  - Man beachte, daß diese Aktivierungsfunktion nur Ausgabewerte in dem Intervall $(0, \ldots, 1)$ liefert.
  - Die in Gleichung 3.2 angegebene Funktion ist die Vorgabe, die ProSiNN verwendet. Der Benutzer kann aber eine andere Aktivierungsfunktion definieren (siehe Abschnitt 11.3).

Die hier nicht erwähnten Einzelheiten des ProSiNN–Modells entsprechen denen, die in Abschnitt 2.2 und 2.3 gemacht worden sind.

3.3 Start des Systems

In diesem Abschnitt soll der Befehl vorgestellt werden, der das Programmsystem ProSiNN startet. Die Syntax des Kommandos lautet:

“prosinn“ [*“-p“* **filename**] [**xoptions**]

Der Aufruf des Befehls `prosinn` startet das System und erzeugt ein neues Fenster, das in Bild 3.3 dargestellt ist.

Bem.:

- Mit der Option `-p` **filename** kann der Anwender ein Simulationsprogramm angeben, das ProSiNN nach dem Start ausführen soll. **filename** muß den Pfad und Dateinamen des Programms enthalten.

  Beispiel:
  Die Datei `test`*, die sich in dem selben Verzeichnis wie das ProSiNN–System befindet, enthalte folgenden Text:*

  ```
  print "Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden";
  ```

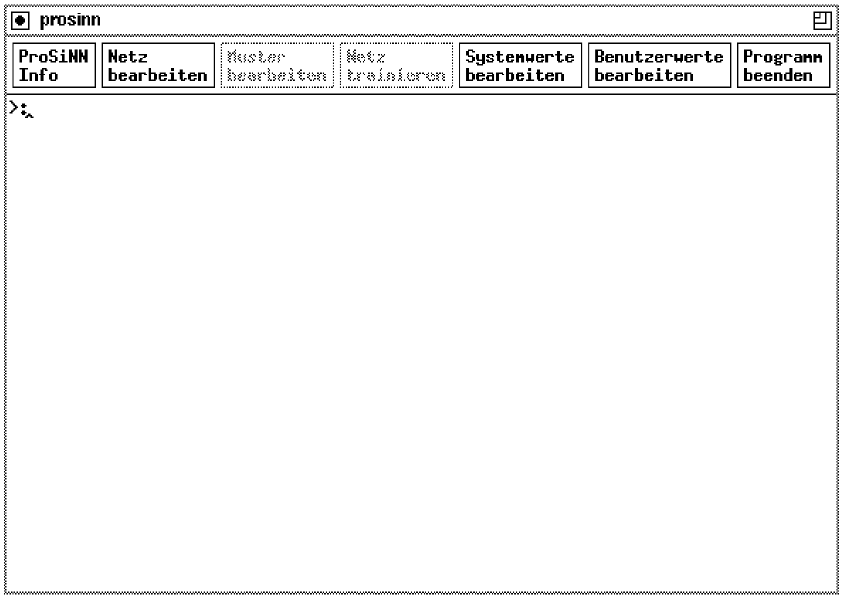


Abbildung 3.3: ProSiNN nach dem Systemstart

Dieser Befehl schreibt in das Fenster die Zeichenkette, die in den Anführungsstrichen eingeschlossen ist (näheres zum `print`*–Befehl siehe in Abschnitt 7.3).*

Nach Aufruf der Befehlsfolge `prosinn -p test` *wird das System gestartet und das Programm ausgeführt. In Abbildung 3.4 ist das dadurch entstandene Fenster dargestellt (die Zeile* >`:load test` *steht für den Befehl, der das Programm* `test` *lädt; siehe Abschnitt 6.2).*

  - Ein Vorteil dieser Option liegt in einem zeitlich verschobenen Systemstart. Wenn ein Anwender z.B. ein Neuronales Netz trainieren will und dies aufgrund der benötigten Rechenzeit tagsüber nicht möglich ist, so kann er das System nachts starten und dann die Aufgabe lösen lassen. Er muß dafür nur alle erforderlichen ProSiNN–Befehle in eine Datei schreiben und das System zu einem bestimmten Zeitpunkt durch das Betriebssystem starten lassen. Ohne die `p`–Option ist dies nicht möglich, da ProSiNN nach dem Start über die Maussteuerung und die Tastatur Eingaben erwartet.
  - Eine andere Möglichkeit ist die Ausführung einer Programmdatei, die Kommandos zur Definition von Vorgabewerten und Befehlsmakros enthält.

- Mit der zusätzlichen Angabe **xoptions** können alle Kommando–Optionen angegeben werden, die das *X Window System* unterstützt. Hier lassen sich u.a. Geometrievorgaben, Farben, Zeichensätze oder andere Ressourcen für das ProSiNN–System setzen (näheres dazu findet man in [JON89] und [YOU89]; eine allgemeine Benutzereinführung in das *X Window System* wird in [MAN90] gegeben).

- Das Programmsystem ist nicht gegen *äußere* Tastatureingaben wie `Control-C` geschützt. Wird diese Tastenkombination in dem Terminalfenster erzeugt, von dem aus das System gestartet wurde, so beendet sich ProSiNN sofort. Deshalb ist es

Abbildung 3.4: Der ProSiNN–Start mit der Option `p`

ratsam, ProSiNN immer als Hintergrundprozeß zu starten. In diesem Fall kann es zu keinem unbeabsichtigten Programmabbruch mehr kommen.

3.4 Die grafische Oberfläche von ProSiNN

In diesem Abschnitt werden die relevanten Aspekte der Anwenderschnittstellen vorgestellt. Dazu wird der Umgang mit dem Ein– und Ausgabefenster, in dem die Befehle für den Kommandozeileninterpreter eingegeben werden, und der Menüoberfläche beschrieben. In Abbildung 3.5 sind u.a. die beiden Schnittstellen dargestellt.

Die Menüoberfläche besteht aus sieben Menüs, deren Benutzung in Unterabschnitt 3.4.2 beschrieben wird. Zunächst wird der Umgang mit dem Kommandozeileninterpreter und dessen Eingabefeld vorgestellt.

3.4.1 Der Kommandozeileninterpreter

Der in Bild 3.5 *kariert* dargestellte Fensterausschnitt bildet den *normalen* Ein– und Ausgabebereich von ProSiNN.

- Befindet sich der Mauszeiger innerhalb dieses Gebietes, das im folgenden als *(normales) Ein/Ausgabefenster* bezeichnet wird, so kann der Benutzer mit Hilfe der Tastatur Eingaben an das System weiterreichen. Eingaben sind aber nur möglich, wenn das Prompt '>:' und der Cursor '‸' vorhanden sind. Wird die `Return`–Taste gedrückt, dann übernimmt der Interpreter die Eingabe und wertet sie aus.

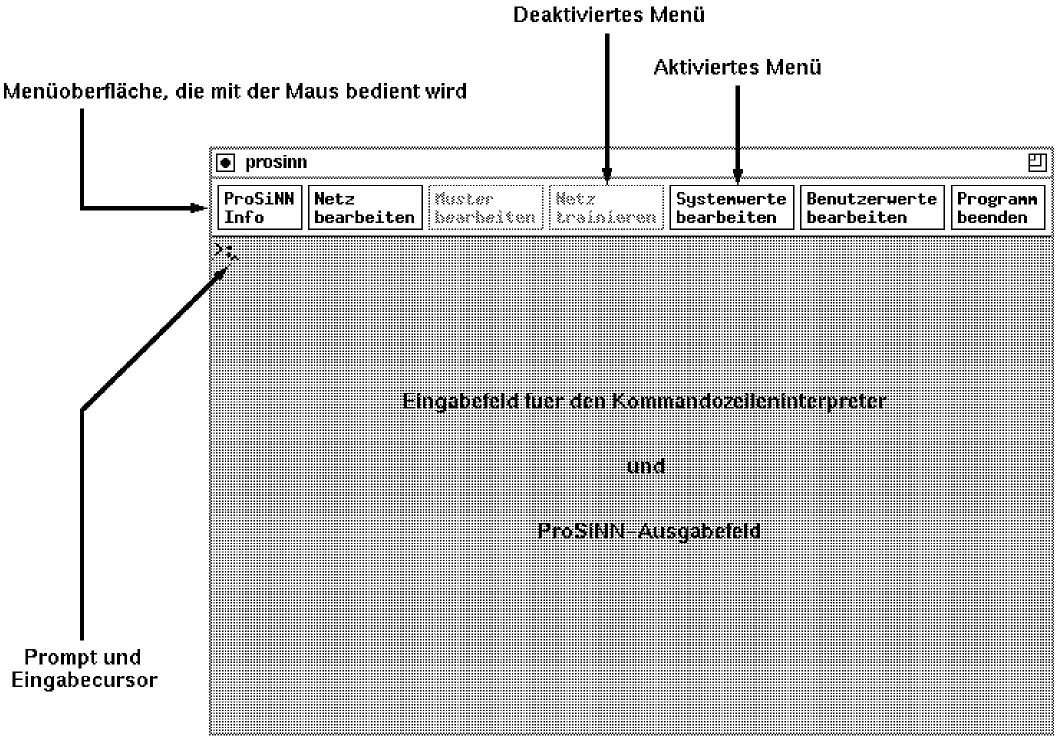


Abbildung 3.5: Die Bedienungsoberfläche von ProSiNN

Beispiel:
Um den Ausgabebefehl von Seite 35 direkt auszuführen, gibt man folgendes ein:

```
print "Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden"
```

Nach dem Drücken der `Return`*–Taste wird der Befehl ausgewertet. Nun steht im Ein/Ausgabefenster folgender Text:*

```
>:print "Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden"
Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden
>:
```

- Mit einem Semikolon kann der letzte eingegebene Befehl wiederholt werden.

Beispiel:
Nachdem das Semikolon als nächste Eingabezeile eingegeben wurde, steht im normalen Ein/Ausgabefenster der nachfolgende Text:

```
>:print "Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden"
Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden
>:;
Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden
>:
```

- Das Semikolon muß zum Trennen mehrerer Befehle in einer Eingabezeile verwendet werden. Am Ende der Kommandozeile darf aber kein Semikolon stehen!

  Beispiel:
  Zur Weiterführung des Beispiels soll nun der gleiche Text noch einmal ausgegeben werden. Jetzt wird das ursprüngliche Kommando aber in zwei Befehle zerlegt, die jeweils einen Teil des Textes ausgeben.

```
>:print "Hallo!"; print
" Ich bin von einem ProSiNN-Programm erzeugt worden"
Hallo! Ich bin von einem ProSiNN-Programm erzeugt worden
>:
```

- Entsteht bei der Ausführung eines Kommandos ein Fehler, so erzeugt das System eine Meldung. ProSiNN gibt dann die Fehlernummer und eine nähere Erläuterung aus. Sollte der Fehler in einem Simulationsprogramm aufgetreten sein, wird außerdem noch der Name der ausgeführten Datei und die Zeilennummer, in der der Fehler erkannt wurde, hinzugefügt.

  Beispiel:
  Wenn der Benutzer ein unbekanntes Kommando eingibt, erscheint z.B. die folgende Fehlermeldung:

```
>:prind "Hallo!"

Fehler 7:
Kommando unbekannt: "prind"

>:
```

  Man achte hier auf die fehlerhafte Schreibweise des `print`*–Befehls.*

- Im folgenden werden die wichtigsten Texteditierbefehle (Tastenkombinationen) und die Maussteuerung beschrieben. Die angegebenen Editierbefehle gelten für alle Eingabefelder von ProSiNN. Auch die über die Maus gesteuerten Scrollkommandos werden im gesamten System auf die gleiche Art und Weise benutzt.

  – Innerhalb des Ein/Ausgabefensters können Zeichenfolgen nur hinzugefügt werden. Um eine eingegebene Zeile zu korrigieren, müssen mit der `Delete`–Taste alle Zeichen rückwärts bis zur Fehlerstelle gelöscht werden.

  – Paßt der vorhandene Text nicht mehr in das Ein/Ausgabefenster, so erscheint links ein Scrollfeld. In Abbildung 3.6 ist dieser Sachverhalt dargestellt.

    Der Scrollbalken repräsentiert die relative Lage innerhalb des Textes. Um den angezeigten Bereich zu verschieben, muß der Mauszeiger auf das Scrollfeld positioniert werden. Mit den Maustasten kann der dargestellte Bereich folgendermaßen geändert werden:

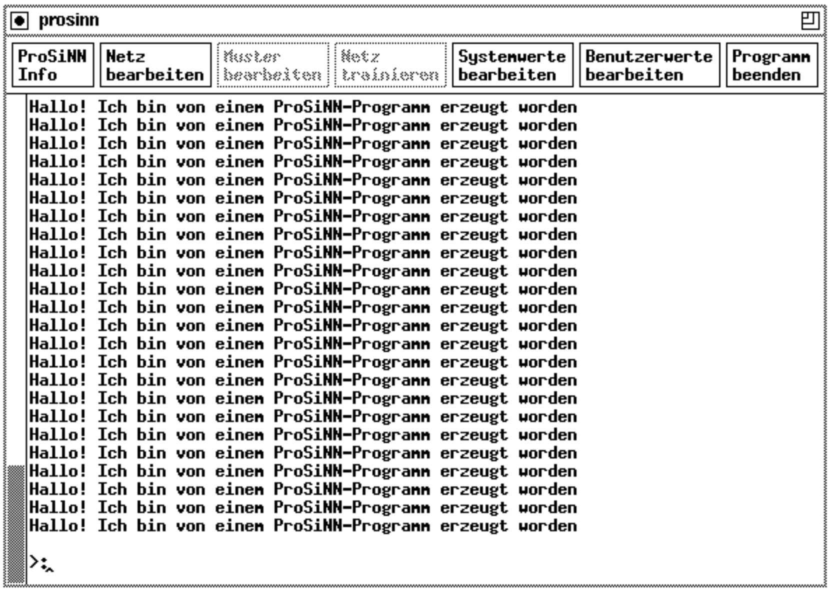


Abbildung 3.6: Das Scrollfeld des Ein/Ausgabefensters

Button 1 : Text wird nach oben bewegt (Richtung Textende)
Button 2 : Text wird zu der relativen Lage des Mauszeigers im Scrollfeld bewegt
Button 3 : Text wird nach unten bewegt (Richtung Textanfang)

– Zur Textbearbeitung existieren eine Reihe von Tastenkombinationen, die für alle Eingabefelder des ProSiNN–Systems gelten. Innerhalb des Ein/Ausgabefensters können einige allerdings nur eingeschränkt verwendet werden (da nur vom Ende des Textes gelöscht und eingefügt werden darf). Die folgenden Kombinationen bestehen aus zwei oder drei Tasten, die gleichzeitig gedrückt werden müssen (**Ctrl** repräsentiert die `Control`–Taste):

Ctrl-a : zum Zeilenanfang gehen
Ctrl-b : ein Zeichen zurücksetzen
Ctrl-d : nächstes Zeichen löschen
Ctrl-e : zum Zeilenende gehen
Ctrl-f : ein Zeichen vorwärts gehen
Ctrl-h : vorhergehendes Zeichen löschen
Ctrl-j : erzeuge Zeilenende und Absatz
Ctrl-k : lösche den Rest der Zeile
Ctrl-l : zentriere Text zum Cursor
Ctrl-m : erzeuge Zeilenende
Ctrl-n : gehe zur nächsten Zeile
Ctrl-p : gehe zur vorhergehenden Zeile
Ctrl-r : suche/ersetze rückwärts
Ctrl-s : suche/ersetze vorwärts
Ctrl-t : tausche die Zeichen vor und nach dem Cursor aus

Ctrl-u	: Die nächste Taste(nkombination) wird 4 mal ausgeführt
Ctrl-v	: eine Seite vorwärts scrollen
Ctrl-w	: lösche selektierten Bereich
Ctrl-y	: letzten Löschbefehl eines selektierten Bereiches zurücknehmen
Ctrl-z	: eine Zeile nach oben scrollen
Meta-b	: gehe ein Wort zurück
Meta-f	: gehe ein Wort vorwärts
Meta-i	: füge Datei ein
Meta-k	: lösche bis Ende des Absatzes
Meta-q	: formatiere Absatz
Meta-v	: eine Seite rückwärts scrollen
Meta-y	: füge selektierten Bereich ein
Meta-z	: eine Zeile nach unten scrollen
Meta-d	: lösche nächstes Wort
Meta-h	: lösche vorhergehendes Wort
Meta-<	: scrolle zum Textanfang
Meta->	: scrolle zum Textende
Meta-]	: scrolle zum nächsten Absatz
Meta-[	: scrolle zum vorhergehenden Absatz
Meta-Delete	: lösche vorhergehendes Wort
Meta-Shift-Delete	: lösche vorhergehendes Wort (mit Ctrl-y wieder korrigierbar)

Übersicht: Die erlaubten Tastenkombinationen in Eingabefeldern

– Mit der Maus lassen sich Textbereiche selektieren und einfügen. Die Maustasten haben dabei die folgenden Bedeutungen:

Button 1 drücken	: Selektion beginnt
Button 1 und Maus wird bewegt	: Selektion wird mitbewegt
Button 1 loslassen	: selektierter Bereich wird intern gespeichert
Button 2 drücken	: Selektion wird eingefügt
Button 3 drücken	: Änderung der Selektion beginnt
Button 3 und Maus wird bewegt	: Selektion wird geändert
Button 3 loslassen	: Änderung der Selektion wird beendet

3.4.2 Das Menüsystem

In diesem Unterabschnitt sollen die wesentlichen *Umgangsformen*, die zum Bedienen des Menüsystems erforderlich sind, beschrieben werden.

Das ProSiNN–Menüsystem besteht aus sieben verschiedenen Menüs. Jedes Menü kann aktiv oder inaktiv sein (siehe Abbildung 3.5). Inaktive Menüs ergeben sich durch den Systemzustand. In Bild 3.5 ist das Menü *Muster bearbeiten* inaktiv, da noch kein Neuronales Netz geladen oder erzeugt worden ist und kein Muster ohne das dazugehörige Netz vorhanden sein darf! Da in diesem Fall kein Netz trainiert werden kann, ist auch das Menü *Netz trainieren* inaktiv. Im Menüsystem werden diese Aktivierungen durch ProSiNN selbständig verwaltet.

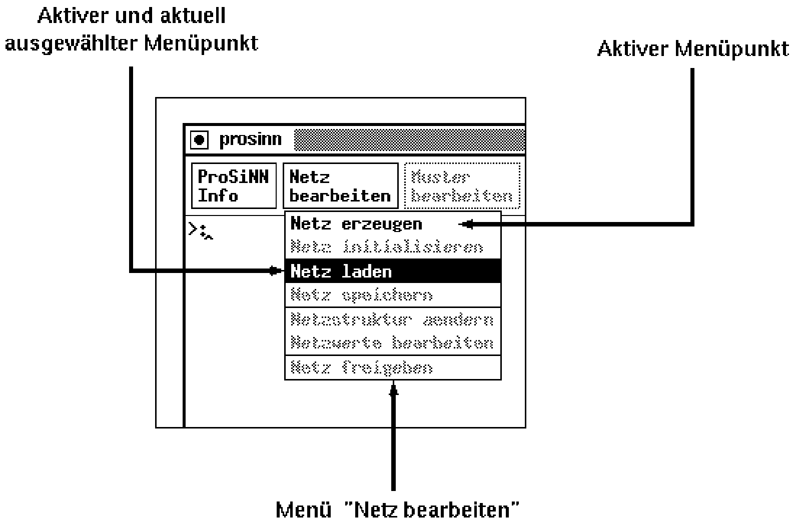


Abbildung 3.7: Das Menü *Netz bearbeiten*

Abbildung 3.7 zeigt einen Auschnitt des Bildschirms, der gemacht wurde, als das Menü *Netz bearbeiten* nach dem Systemstart angeklickt wurde. In der anschließend erzeugten Menübox sind nur zwei Menüpunkte aktiv: *Netz erzeugen* und *Netz laden*. Nur diese beiden Möglichkeiten sind zur Zeit für den Benutzer zugänglich. Um einen Punkt auszuwählen, muß der Mausbutton gedrückt bleiben; durch vertikale Bewegung der Maus wird derjenige Menüpunkt invers dargestellt, auf dem sich der Mauszeiger gerade befindet (in der Abbildung ist dies *Netz laden*). Wenn der Anwender jetzt die gedrückte Maustaste losläßt, wird die mit dem Menüpunkt verbundene Semantik ausgeführt.

Das Bild 3.8 zeigt das System, nachdem der Menüpunkt *Netz laden* ausgewählt worden ist.

Das generierte Popup–Fenster dient zum Laden einer Datei, die die Definition eines Neuronalen Netzes enthält. ProSiNN erzeugt immer diese Dateiauswahlbox, wenn der Benutzer mit dem Menüsystems ein File auswählen muß.

In einem Popup–Fenster werden Auswahlbuttons in der Regel *schwarz (invertiert)* dargestellt. Hier wird z.B. durch Anklicken des Buttons $\boxed{\text{Abbruch}}$ der Menüpunkt ohne Laden eines Netzes beendet.

In dem Verzeichnispfadeingabefeld wird der Pfad des aktuellen Arbeitsverzeichnisses angegeben. Er kann dadurch geändert werden, daß der Anwender den Mauszeiger in dieses Feld setzt und über die Tastatur den Pfad korrigiert. Im Anschluß wird durch Drücken der `Return`–Taste das neue Verzeichnis und dessen Inhalt angezeigt. Über das Teilfeld, das