Alexander Sinsel, Heidelberg, 2001

Protokoll zum Informatik−Praktikum

Themenbereich: Neuronale Netze

(1) Gegenstand des Praktikums

Ein mit einem genetischen Lernalgorithmus ausgestattetes, künstliches, neuronales Netzwerk soll anhand von geeigneten Lerngegenständen getestet werden. Die Menge der in diesem Zusammenhang sinnvollen Lernobjekte ist durch Art und Anzahl der Eingabeneuronen, durch die Grenzen realisierbarer Netzwerktopologien und die Geschwindigkeit des Netzwerkes stark eingeschränkt, stellt man vernünftige Erwartungen an den Lernerfolg und dessen zeitlichen Rahmen. Die Netzwerktopologie muß den Anforderungen des Lerngegenstandes genügen, die Eingabemuster müssen den Eingabeneuronen entsprechend codierbar sein und auch die Darstellbarkeit der gewünschten Ergebnisse in den Ausgangsneuronen ist zu gewährleisten. Der Lernerfolg ist durch den Quotienten der Anzahl richtig erkannter aus einer Anzahl vorgelegter Eingabemuster in Form einer **Fitness** quantifizierbar.

(2) Beschreibung des Netzwerks

Das vorliegende Netzwerk hat als Hardware−Realisierung einen Geschwindigkeitsvorteil gegenüber Softwaresimulationen neuronaler Netze. Dafür ist die Anzahl der Neuronen fest vorgegeben. Es handelt sich um 64 **digitale Eingangsneuronen,** die mit 64 **digitalen Ausgangsneuronen** über **analoge Synapsen** vollständig vernetzt sind. 32 der 64 Ausgangsneuronen lassen sich auf die unbelegten Eingangsneuronen zurück koppeln, so daß mehrschichtige Netzwerktopologien erreicht werden können. Die folgende Tabelle zeigt, welche Ausgangsneuronen auf welche Eingangsneuronen (als sogenanntes **feedback**) zurück geführt werden können:

Ausgangsneuron	64	63	62	...	51	50	49	1	2	3	4	....	14	15	16
Eingangsneuron	2	4	6	....	28	30	32	34	36	38	40	....	60	62	64
Feedbackschalter	1	2	3		14	15	16	17	18	19	20		30	31	32

(**Tabelle 2.1**)

Ob ein Eingangsneuron eine externe Eingabe oder aber ein vom entsprechenden Ausgangsneuron zurückgeleitetes Ausgabesignal erhält, ist durch Setzen des jeweiligen Feedbackschalters zu steuern. Das bedeutet, daß bei einem Netzwerk mit n (0 <= n <= 32) Neuronen in den Zwischenschichten (hidden layers) nur noch eine maximale Anzahl von 64 − n Eingabeneuronen zur Verfügung steht. So gestaltet sich beispielsweise ein 3−schichtiges Netzwerk mit jeweils 4 Neuronen in der Eingabe− und Zwischenschicht mittels der Eingangsneuronen 1, 3, 5, 7 für die Eingabeschicht und der Ausgangsneuronen 61, 62, 63, 64 als Zwischenschicht unter Rückkopplung auf die Eingangsneuronen 2, 4, 6, 8, durch Aktivieren der ersten 4 Feedbackschalter.

Die Steuerung und Einstellung des Netzwerks erfolgt über das Programm EVOQT. Im Register *Network* befindet sich das Textfeld *Feedback* in welchem sich die 32 Feedbackschalter in Form einer 32 Bit Binärzahl einstellen lassen. Den übrigen 32 Eingabeneuronen ist statt eines Feedbackschalters ein Combineschalter vorgesetzt. Setzt man die Eingabeneuronen auf **combine**, so wird ihnen statt eines Eingabesignals das invertierte Signal des nachfolgenden Eingabeneurons zugewiesen. Dies hat den Zweck, elektrische Asymmetrien, in dem Falle hinsichtlich der Kapazitaet, zwischen den Eingabesignalen 0 und 1 zu vermeiden, indem diese in die kapazitiv gleichwertigen Signale 10 für 1, bzw. 01 für 0 umgewandelt werden. Das bedeutet, daß man kein Eingabeneuron mehr zur Verfügung hat, wenn man alle 32 feedback Bits und alle 32 combine Bits setzt. Da nur insgesamt 64 Eingabeneuronen zur Verfügung stehen, wird man bei mehrschichtiger Netzwerktopologie und 32 oder mehr Eingabebits auf den Combinemodus verzichten müssen. Ein zu untersuchender Aspekt ist der Einfluß des Combinemodus auf den Lernprozeß (siehe Teil 5!)

Im Register *Evolution* findet man neben der Einstellung für den evolutionären Lernalgorithmus unter dem Begriff *mapping* eine Einstellungsmöglichkeit für *first* und *last neuron*. In diesem Intervall müssen die als feedback zurückgeleiteten Ausgangsneuronen der Zwischenschicht erfaßt sein, als auch noch eine entsprechende Anzahl weiterer Ausgabeneuronen für das Ergebnis. Wollte man in dem oben genannten Beispiel die Ausgabe auf ein Neuron beschränken, so müßte man also *first neuron* = 60 und *last neuron* = 64 einstellen, wobei dann Ausgangsneuron 60 zur Ausgabe und 61 bis 64 als Zwischenschicht verwendet würden. (Vielleicht sollte das *mapping* in das Register *Network* verschoben werden, da es sich auf die Netzwerktopologie bezieht und nicht auf den genetischen Lernalgorithmus)

(3) Der genetische Lernalgorithmus

(a) Lernen in der Theorie der neuronalen Netze

In der Theorie der neuronalen Netze stellt sich der Begriff des Lernens als das Auffinden richtiger synaptischer Gewichtungen in einem neuronalen Netz dar. Ein Netzwerk hat ein Problem erlernt, wenn die Gewichtungen zwischen den einzelnen Neuronen so beschaffen sind, daß zu jeder an den Eingangsneuronen angelegten Eingabe die Ausgangsneuronen das gewünschte Signal liefern. Voraussetzung ist, daß die **Netzwerktopologie**, also die Anzahl der Neuronen und die synaptische Verbindungsstruktur zwischen diesen, ein Erlernen überhaupt prinzipiell ermöglicht. Ein neuronales Netz hat beispielsweise das logische ODER bzw. UND erlernt, wenn die Eingaben (grau) eine Ausgabe entsprechend der folgenden Tabellen bewirken:

OR	0	1
0	0	1
1	1	1

(**Tabelle 3.1**)

AND	0	1
0	0	0
1	0	1

(**Tabelle 3.2**)

Die Realisierung der Oder−Funktion ist mit der primitivsten Netzwerktopologie bereits möglich, mit dem sogenannten **Perzeptron**, welches aus n Eingabeneuronen besteht, die über n Synapsen zu einem Ausgabeneuron führen. Im Falle der logischen Funktionen, wie hier des ODERs wäre n=2. Das Perzeptron funktioniert nach dem Prinzip "Integrate and fire", d.h. das Ausgabeneuron summiert die ankommenden Signale auf und vergleicht diese Summe mit seinem Schwellenwert. Erreicht diese Summe den Schwellenwert, so sendet das Ausgangsneuron des Perzeptrons ein Signal, liegt die Summe unterhalb des Schwellenwertes sendet es kein Signal. Da das Ausgabeneuron lediglich ein bestimmtes Signal sendet oder nicht, ist dies ein digitales Neuron. Die Eingabeneuronen sind ebenfalls digital, gemäß dem Wertebereich beider logischer Variablen (0 für falsch, 1 für wahr). Gewichte und Schwellenwert sind hingegen beliebige kontinuierliche Werte, entsprechend den analogen Synapsen.

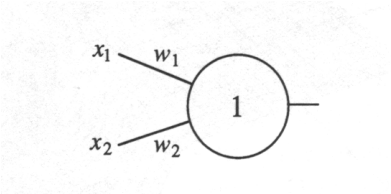
Die Bedingung im Perzeptron für FIRE (oder wahr) ist:

(Synapsengewicht1 * Eingabe1) + (Synapsengewicht2 * Eingabe2) >= Schwellenwert

Habe das Perzeptron den Schwellenwert 1, so ist die Gewichtung 2 für beide Synapsen eine Lösung der ODER−Funktion:

(2 * Eingabe1) + (2 * Eingabe2) >= 1 (Bedingung für wahr)

Die niedrigste Gewichtung im Lösungsraum für die ODER−Funktion ist jeweils 1 für beide Synapsen. Gewichtungen im Intervall von [0,5 ; 1) wären hingegen eine Lösung für die logische UND−Funktion (Tabelle 3.2). Die entscheidenden Variablen bei dem Lernproblem sind die synaptischen Gewichte − die Schwellenwerte lassen sich prinzipiell alle auf 1 normieren, indem man Schwellenwert und die Gewichte der ankommenden Synapsen durch den Schwellenwert teilt. Die Anzahl der aufzusuchenden Gewichte nennt man die **Dimension des Lernproblems**. Sie ist in beiden Fällen 2. In der vorliegenden Implementierung können die Gewichte sowohl positiv als auch negativ sein



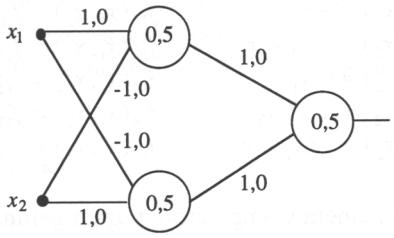
(**Abbildung 3.1**: Das Perzeptron)[1]

Es lassen sich jedoch durchaus nicht alle der 16 Boolschen Funktionen von 2 Variablen mit einem Perzeptron realisieren. Für die XOR−Funktion und ihre Negation finden sich keine Gewichtungen. Diese erfordern eine komplexere Netzwerkarchitektur. Sie sind mit keinem zweischichtigen Netzwerk zu realisieren.

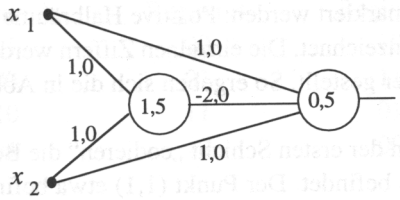
Definition 3.1:

Eine **Schichtenarchitektur** liegt vor, wenn die Menge N von Neuronen in j Schichten N_1, N_2, .., Nj aufgeteilt werden kann, und zwar so, daß die Eingabestellen nur an die Schicht N_1 und die Ausgabestellen nur an die Schicht Nj angeschlossen sind. Außerdem dürfen die Synapsen ausschließlich Neuronen in aufeinander folgenden Schichten verbinden.[2]

Die folgenden Abbildungen stellen zwei Netze für die XOR−Funktion dar, ein dreischichtiges, bei dem die Dimension des Lernproblems 6 beträgt (Abbildung 3.2) und ein Netz ohne Schichtenarchitektur, bei dem die Dimension des Lernproblems auf 5 reduziert ist (Abbildung 3.3).



(**Abbildung 3.2**)[3]



(**Abbildung 3.3**)[4]

1 Aus Rojas − „Theorie der neuronalen Netze“; 4. korregierter Nachdruck; Springerverlag: Abbildung 3.8
2 Rojas: Seite 122
3 Rojas: Abbildung 6.3
4 Rojas: Abbildung 6.5

Definition 3.2 Zwei Mengen A und B von Punkten in einem n-dimensionalen Raum sind *linear trennbar*, falls $n+1$ reelle Zahlen $w_1,...,w_{n+1}$ existieren, so daß für jeden Punkt $(x_1,...,x_n) \in A$ gilt

$$\sum_{i=1,n} w_i x_i \geq w_{n+1}$$

und für jeden Punkt $(x_1,...,x_n) \in B$

$$\sum_{i=1,n} w_i x_i < w_{n+1}.$$

Definition 3.3 Eine binäre Funktion $f: R^n \rightarrow \{0,1\}$ heißt linear trennbar, falls die Menge der Punkte, für die $f(x_1,...,x_n) = 1$ gilt, von der Menge der Punkte, für die $f(x_1,...,x_n) = 0$ gilt, linear getrennt werden kann.

[5]

Es sind außer der XOR−Funktion und ihrer Negation alle anderen binären Funktionen mit zwei Variablen linear trennbar und damit durch ein Perzeptron darstellbar. Ist lineare Trennbarkeit für eine Funktion nicht gegeben, so wird die erforderliche **Netzkapazität** (Anzahl der vom Netz darstellbaren Funktionen) und damit die Dimension des Lernproblems größer.
Betrachtet man die **Fehlerfunktion**, welche den Fehler in Abhängigkeit von den Gewichtungen angibt, so wird deutlich, warum lineare Trennbarkeit das Lernproblem erheblich vereinfacht. Der **Fehler** ist der reziproke Wert der Fitness, d.h. die Anzahl richtiger pro Anzahl aller Ausgabebits über die Zahl aller denkbaren Eingabemuster summiert. (Bei digitalen Ausgabeneuronen muß der Fehler in der Fehlerfunktion nicht gewichtet werden − da zwischen wahr und falsch keine Zwischenwerte liegen, ist jeder auftauchende Fehler mit 1 zu gewichten.) Die Fehlerfunktion eines Lernproblems der Dimension n entspricht einer n−dimensionalen Funktionslandschaft. Das Lernproblem besteht in dem Aufsuchen des absoluten Minimums der Fehlerfunktion. Bei linearer Trennbarkeit ist jedes lokale Minimum ein absolutes. Damit vereinfacht sich das Lernproblem erheblich.

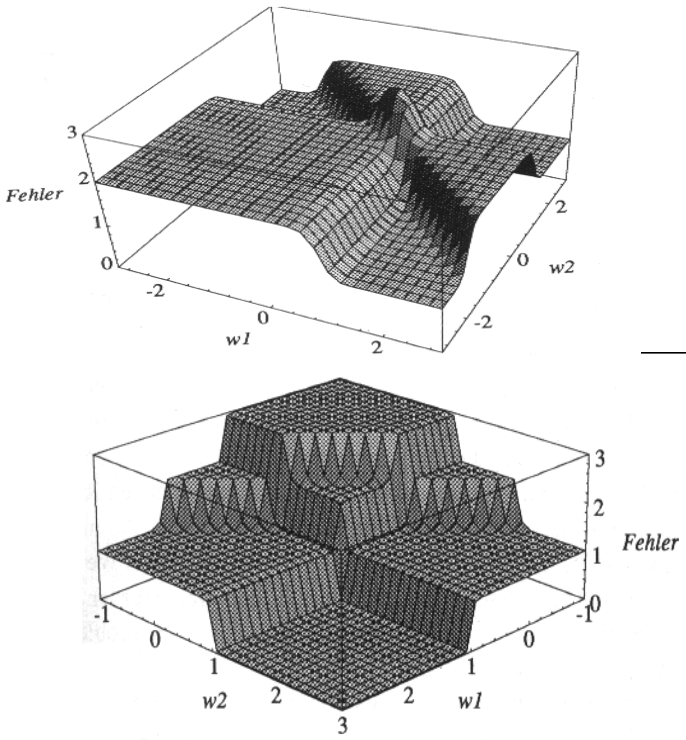


Abbildung 3.4 (oben): Fehlerfläche der XOR−Funktion
Abbildung 3.5 (unten): Fehlerfläche der OR−Funktion[6]

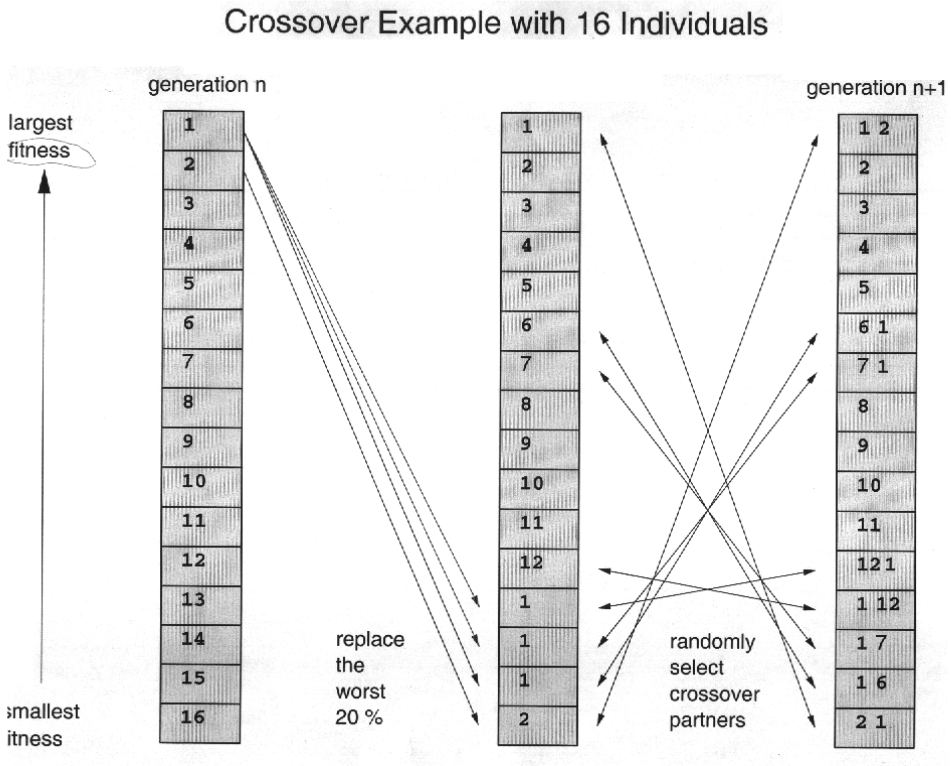
5Rojas: Seite 78/79
6Rojas: Abbildung 3.10 und 3.9

(b) Beschreibung des genetischen Algorithmus

Es gibt verschiedene Algorithmen zum Auffinden der günstigsten Netzwerkgewichtungen, d.h. zur Minimierung der Fehlerfunktion. Mit am bekanntesten ist Backpropagation, eine Art Gradienten−Abstiegsverfahren. Ein grundsätzliches Problem hierbei ist, daß solches sich in all seinen Modifikationen mehr oder weniger in lokalen Minima verläuft, die nicht unbedingt dem kleinsten Fehler entsprechen müssen (siehe oben!). Zudem machen digitale Ausgabeneuronen die Fehlerfunktion zu einer Stufenfunktion, bei der sich der Gradient nicht bilden läßt.

Der hier verwendete, sogenannte **genetische Algorithmus** bedarf keiner Gradientenbildung und ist von daher dem Netzwerk angemessen. Außerdem ist die Gefahr, in einem ungünstigen lokalen Minimum der Fehlerfunktion zu enden, bei diesem Algorithmus geringer. Das Prinzip ist folgendes:

Der Gewichtswert einer einzelnen Synapse wird durch ein **Gen** repräsentiert. Alle Gene, die sich auf ein Neuron beziehen, werden zu einem **Chromosom** zusammengefaßt. So umfaßt das Genom eines **Individuums** im vorliegenden Netzwerk 64 Chromosomen und beinhaltet eine gesamte Netzwerkgewichtung. Eine **Evolution** wird mit einer zufälligen Anfangspopulation gestartet. Im Register *Evolution* in EVOQT wird im Textfeld *population* die Größe einer Population, also die Anzahl der Individuen pro Generation festgelegt. Nun wird jedes Individuum in die Gewichtematrix des Netzwerkes geladen und die Testmuster an der Eingabeschicht angelegt. Dabei wird für jedes Individuum die Fitness berechnet, indem die Signale an den Ausgabeneuronen mit der erwünschten Ausgabe verglichen werden (**überwachtes Lernen**). Pro übereinstimmendes Bit, also für jedes richtige Ausgabeneuron, wird die Fitness um 1 erhöht. Anschließend wird die Population nach der Fitness sortiert. Das Wesentliche besteht nun in der Anwendung der beiden genetischen Operatoren: Kreuzung (**Crossover**) und **Mutation**. Die schlechtesten 25% einer Population werden zu gleichen Hälfte durch das beste Individuum und die 12,5% besten Individuen ersetzt und anschließend mit einer weiteren Anzahl von zufällig gewählten 25% aller Individuen gekreuzt. Die restlichen 50% der Population bleibt dabei unverändert. Die Kreuzung erfolgt, indem für jeweils zwei entsprechende (sich auf das gleiche Neuron beziehende) Chromosomen beider Individuen ab einem zufällig gewählten Punkt alle Gene miteinander vertauscht werden. Auf diese neue Population wird daraufhin der Mutationsoperator angewandt Dieser verändert jedes Gen mit der gleichen Wahrscheinlichkeit. Diese Wahrscheinlichkeit kann man im Textfeld *mutation* einstellen. Wenn ein Gen verändert wird, so ist der neue Gewichtswert wieder ein zufälliger.



(**Abbildung 3.6**)[7]

7 Graphik von Johannes Schemmel

Da der genetische Algorithmus zugleich an vielen Stellen nach dem globalen Minimum der Fehlerfunktion (dem Maximum der Fitness) sucht, ist er für Fehlerfunktionen mit vielen lokalen, suboptimalen Minima − also für Lernprobleme, welche in mehreren zweidimensionalen Hyperebenen nicht linear trennbar sind − besonders geeignet. Bei einer anderen Implementierung (z.B. Softwareimplementierung) koennte man auch an verschiedenen zufällig gewählten Ausgangspunkten eine Backpropagation starten, diese ließe sich jedoch nicht parallelisieren. In der Parallelisierbarkeit liegt die eigentliche Stärke des genetischen Algorithmus. (Wobei jedoch eine zeitliche Begrenzung durch die Taktung des Netzwerkes und dessen Ansteuerung gegeben ist.)

(4) Codierung der Lerndaten und Kapazität der Netzwerktopologie

Durch die begrenzte Anzahl der Eingabeneuronen auf maximal 64, sind die in Frage kommenden Lerngegenstände von vornherein eingeschränkt. Grundsätzlich müssen die vom Lernproblem erforderten Testmuster binär und maximal mit 64 Bit codierbar sein. Das bedeutet, daß der zu erlernende Gegenstand eine höchstens 64−stellige Funktion auf dem Definitionsbereich {0,1} sein kann. Für die Erkennung graphischer Muster, welche ein typisches Einsatzgebiet neuronaler Netze darstellt bedeutet das konkreter, daß diese Muster als höchstens 64−Pixel Schwarz−Weiß−Graphik ohne Grauabstufungen vorliegen müssen. Zudem bestehen zu jedem Lernproblem Mindestanforderungen an die Netzwerktopologie. Wie bereits erläutert, läßt sich die UND−Funktion mit einem Perzeptron, d.h. mit einem zweischichtigen Netzwerk aus zwei Eingabe− und einem Ausgabeneuron darstellen. Die XOR−Funktion erfordert jedoch mindestens ein weites Neuron oder eine weitere Schicht (siehe Abbildungen 3.2 und 3.3!). Mit jedem Neuron in der Zwischenschicht reduzieren sich die zur Verfügung stehenden Eingabeneuronen um 1.

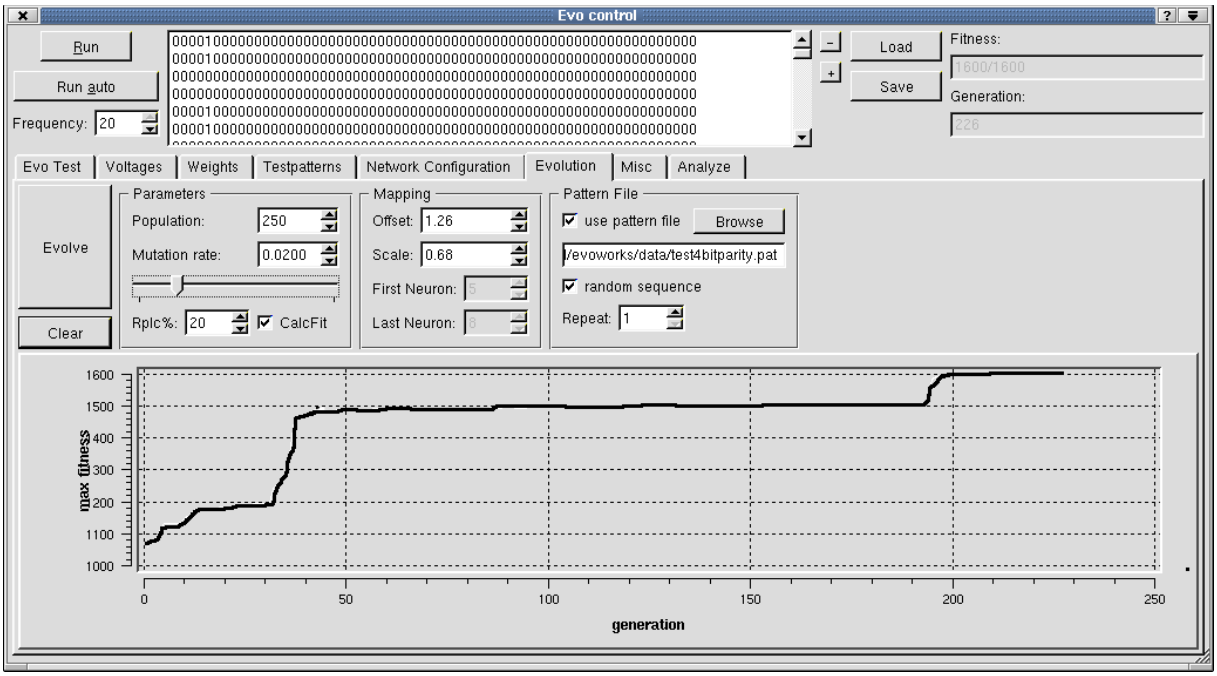
Zu untersuchen wäre, ob in bestimmten Fällen die Lerngeschwindigkeit steigt, wenn man zusätzliche Neuronen in die Zwischenschicht setzt und damit die Netzwerkkapazität über die minimal nötige hinaus erhöht. Dies verändert die interne Codierung des Netzwerkes. So kann die 2 Bit Zahl 3 zum einen als Anzahl von Einzen im Strichmuster (1111) mit 4 Neuronen (4 Bit Codierung) oder als 2 Bit Binärzahl (11) mit nur 2 Neuronen gespeichert werden. Im ersten Falle unterschiede sich die Zahl 3 von der 1 in ¾ der Ausgabeneuronen, im zweiten Falle nur um ½. Betrifft die Codierung die Ausgabeneuronen, so würde für ein Individuum mit der Ausgabe 1 statt des Sollwertes 3 der relative Fehler im ersten Falle zu ¾, im zweiten nur zu ½ berechnet werden, was sich auf den Lernalgorithmus auswirkt. Bei der zweiten Codierung würde der Lernalgorithmus dem Wert 1 und dem Wert 2 den gleichen Fehler zuordnen, unbeachtet der Tatsache, daß die mathematische Differenz zur 3 eine unterschiedliche ist. Es ist zu beachten, daß oftmals, aber keineswegs immer ist die mathematische Differenz vom Sollwert eine Aussage über den Fehler trifft. Seien beispielsweise 4 verschiedene graphische Muster zu erkennen, so kann das mit 1 bezeichnete Muster dem mit 3 bezeichneten ähnlicher sein, als das mit 2 bezeichnete Muster. All solche Aspekte sind bei der **Codierung der Sollwerte** zu beachten.

(5) Paritätsberechnung

Das Paritätsproblem stellt eine besondere Herausforderung für einen Lernalgorithmus dar. Insofern es die mehrdimensionale Verallgemeinerung der XOR−Funktion ist, handelt es sich nämlich um eine in jeder zweidimensionalen Hyperebene linear untrennbare Funktion. Dementsprechend reich ist die Fehlerfunktion an suboptimalen Minima. Darum ist das Paritätsproblem ein geeignetes, um die Fähigkeiten des Lernalgorithmus zu testen. Bei einer 4 Bit Zahl ist die Parität in gleichem Maße von der Anzahl der Einzen, als auch von der Anzahl der Nullen abhängig. Die Ziffern 1 und 0 sind also völlig gleichwertig. Darum soll bei der Paritätsberechnung von 4 Bit Binärzahlen untersucht werden, wie sich der Combinemodus auf den Lernerfolg auswirkt. Zum anderen wäre interessant zu untersuchen, wie sich zusätzliche Neuronen in der Zwischenschicht auf den Lernprozeß auswirken. Weiter könnte man die Grenzen des Netzwerkes überprüfen, indem man untersucht, ob die Parität bei 6 Bit , 8 Bit usw. ebenfalls noch berechenbar ist.

Ergebnisse:

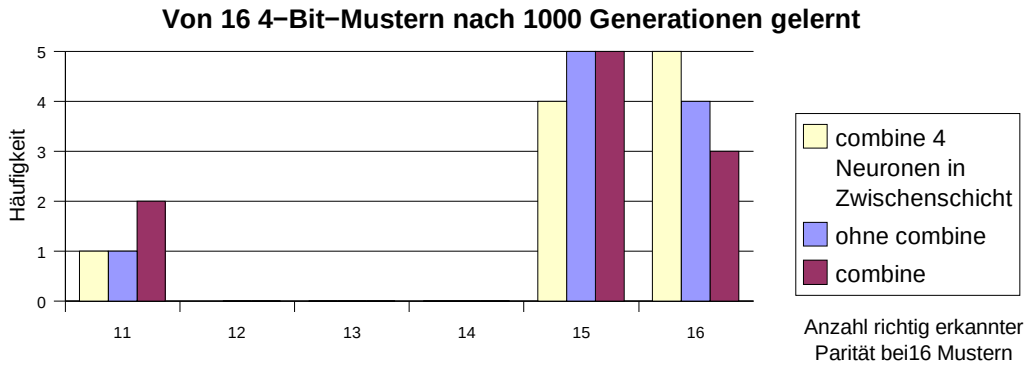
Zunächst galt es, für die Paritätsberechnung von 4−Bit−Zahlen die optimale Mutationsrate zu finden. Eine zu hohe Mutationsrate zerstört die bereits richtig gefundenen Gewichte immer wieder aufs Neue. Eine zu geringe Mutationsrate bewirkt, daß der Lernalgorithmus aus dem erstgefundenen lokalen Fehlerminimum nicht mehr herausfindet. Eine Mutationsrate von 1,6 Prozent erschien mir nach mehreren Variationen am geeignetsten. Für die Größe der Population wählte ich 250 Individuen. Die Netzwerktopologie beinhaltete eine Zwischenschicht mit 3 Neuronen.



(**Abbildung 5.1**: Ein typischer Evolutionsverlauf)

Anzahl der Generationen für 15 richtig erkannte Paritäten von 16	***Anzahl der Generationen für 16 richtig erkannte Paritäten von 16***	***Abbruch bei Generation***	***Combine***	***Neuronen in der Zwischen−schicht***
200	−	2000	Ja	3
49	60	60	Ja	3
2180	2200	2200	Ja	3
−	−	1400	Ja	3
250	582	582	Ja	3
460	−	2000	Ja	3
60	198	198	Nein	3
220	−	7000	Nein	3
120	−	3000	Nein	3
120	439	439	Nein	3
80	−	1000	Ja	4
40	150	150	Ja	4
200	−	1000	Ja	4
35	−	1000	Ja	4
20	45	45	Ja	4
19	42	42	Ja	4
900	1900	2000	Ja	4
30	450	450	Ja	4
−	−	1000	Ja	4
22	45	45	Ja	2

(**Abbildung 5.2**: Ergebnisse von 20 Evolutionen)

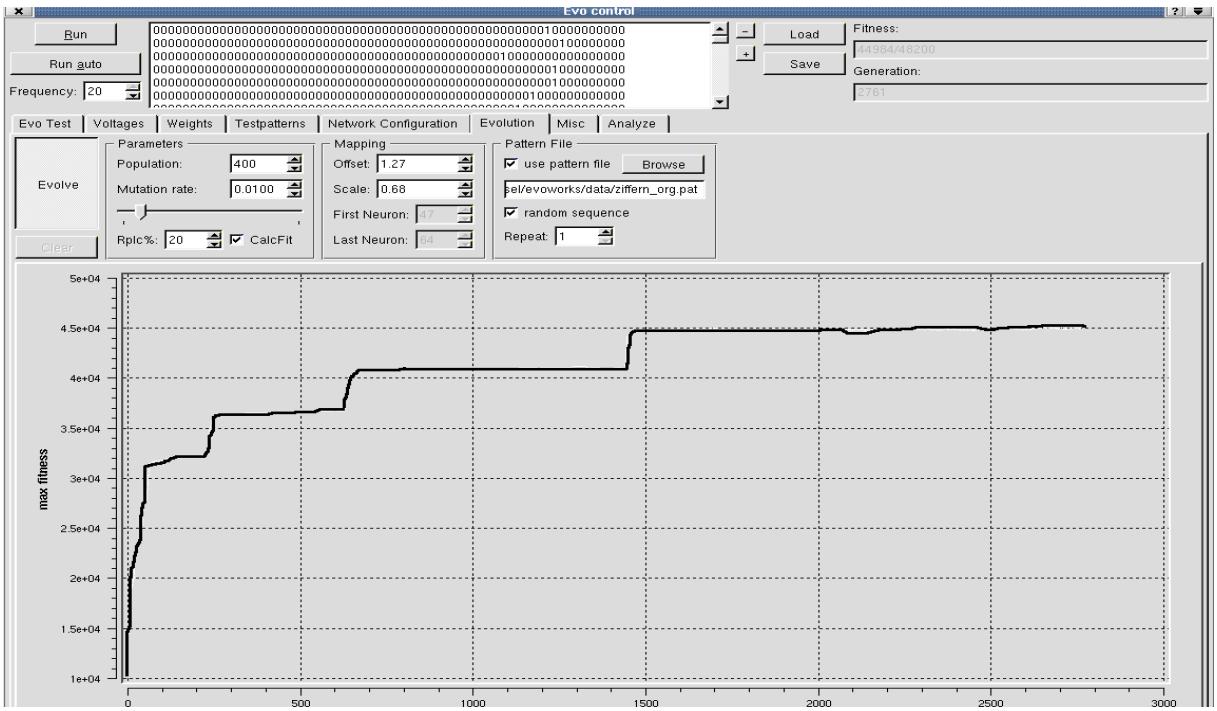


(**Diagramm 5.1**: Veranschaulichung des Lernerfolgs nach 1000 Generationen von jeweils 10 Evolutionen im Combinemodus und 10 Evolutionen ohne combine bei 3 Neuronen in der Zwischenschicht, sowie 4 Neuronen in der Zwischenschicht mit combine)

Bei 8−Bit−Binärzahlen scheiterte eine 100−prozentige Erkennung der Parität innerhalb von 7000 Generationen in mehreren Versuchen über Nacht. Jedoch gelang Johannes ein Lernerfolg von 100 Prozent mit 6−Bit−Binärzahlen.

(6) Identifikation handschriftlicher Ziffern

Da Mustererkennung sehr komplizierter Algorithmen bedarf, ist sie ein typisches Einsatzgebiet neuronaler Netze. So konnte mich Felix' Vorschlag, handschriftliche Ziffern als Testmuster vorzugeben, schnell begeistern, insbesondere wegen der vielen denkbaren Anwendungsbereiche. Die Umsetzung erfolgte mit einer Netzwerktopologie von 56 Eingabeneuronen und 8 Neuronen in der Zwischenschicht. Zunächst wählte ich die ersten 56 Eingangsneuronen für die Eingabe der Testmuster. Die handschriftlichen Ziffern wurden eingescannt und anschließend in Graustufen skaliert. Ich schrieb Thorsten Mauchers Programm VDT dahingehen um, daß man aus der Bilddatei mit den handschriftlichen Ziffern, diese einzeln ausschneiden kann und sie als Schwarz−Weiß−Eingabemuster mit 7 mal 8 Pixel binär codiert in eine für EVOQT lesbare Patterndatei geschrieben werden. Die zu erlernende Ziffer codierte ich zunächst als 4Bit−Binärzahl für 4 Ausgabeneuronen. Bei diesem ersten Versuch führten mehrere Evolutionen zu keinem Lernerfolg. Ich codierte daraufhin die Ausgabe als 10−Bit−Strichmuster mit 9 Nullen und an der der Ziffer entsprechenden Stelle eine 1. Daraufhin ergab sich eine teilweise erfolgreiche Evolution.



(**Abbildung 6.1**: Ergebnis vom 27.04.01)

Im Folgenden ist eine typische Ausgabe des Netzwerkes bei der Eingabe der Ziffermuster 1, 2, 3, .. bis 0 zu sehen.

Eingabemuster: entspricht Ziffer:

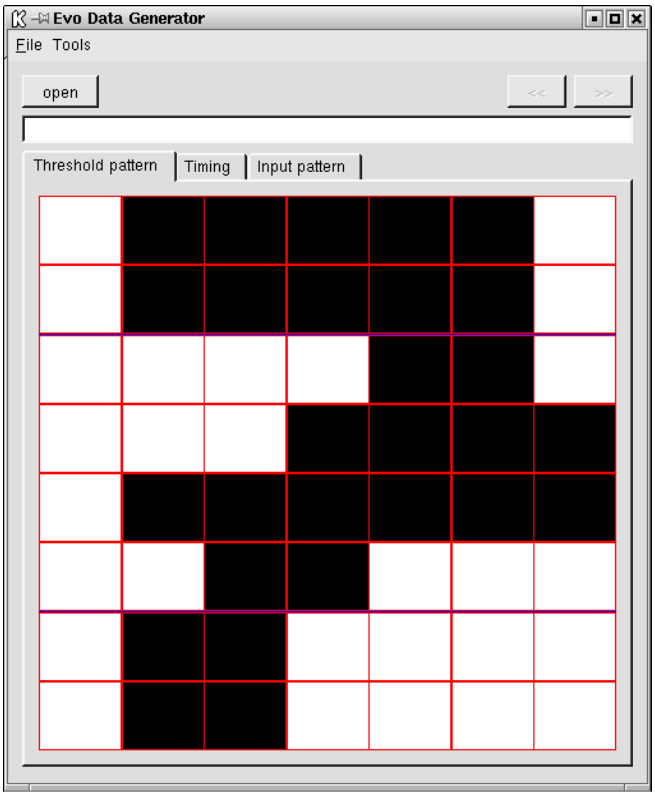
```
0000001010000000001100000110000111000111100110110000011000001100 <--1
0000000000000000111000110110000001000001100001100000111100001100 <--2
0000001010101000111111011011001011000000110000001000111100111110 <--3
0000001000000000010000011001001101100011110000010000001000000000 <--4
0000000000000000011111001000000110000011100000011000111000000000 <--5
0000001000000000011100011000001100000111000010110001111000011000 <--6
0000000000000000111110000001000001100111110000110000010000000000 <--7
0000101010100000011110001011000111000111000010100001011000111100 <--8
0000101010000000001001000101100011110000001000000100111110011100 <--9
0000000000000000011000011110001001100100011011111000011000000000 <--10
```

Ausgabe des Netzwerkes :

```
00000000000000000000000000000000000000000000000011000000001111111
00000000000000000000000000000000000000000000000011000010001111111
00000000000000000000000000000000000000000000000011100000001111110
00000000000000000000000000000000000000000000000011010000001111111
00000000000000000000000000000000000000000000000011001000001111111
00000000000000000000000000000000000000000000000011000100001111111
00000000000000000000000000000000000000000000000011010010001111111
00000000000000000000000000000000000000000000000011000000001111111
00000000000000000000000000000000000000000000000011000000001111111
00000000000000000000000000000000000000000000000011010000001111111
```

```
      Neuronen für die Ausgabe  der Stellen:  1234567890
```

Erkennbar hat das Netzwerk die ersten 7 Ziffer erlernt, jedoch sind die Ausgabeneuronen für die Ziffern 1 und 2 in jedem Fall aktiviert. Es ist unwahrscheinlich, daß dieser systematische Fehler durch den Lernalgorithmus bedingt ist. Naheliegender ist die Vermutung, daß die Ursache hierfür im Netzwerk selbst oder in dessen Ansteuerung liegen muß. Trotz dieses Fehlers ist der Lernerfolg erstaunlich, da auf 7 mal 8 Pixel reduzierte Ziffern nicht sehr leicht zu identifizieren sind, wie folgendes Beispiel der Ziffer 7 veranschaulichen soll:



(**Abbildung 6.2**: Schwarzweißbild einer handgeschriebenen Ziffer 7 reduziert auf 7 mal 8 Pixel)

(7) Persönliches Resümee

Wie bei allen Dingen, die sich in der Testphase befinden, muß man mit unerwarteten Hinternissen rechnen. So gab es doch immer wieder Probleme mit dem Netzwerk, oder der Netzwerkkarte, wie der Sofware, welche die Arbeit aufhielten. In diesem Zusammenhang sind der wiederkehrende Wackelkontakt der provisorischen Karte zu nennen, als auch manche Eigenarten im Programm EVOQT, das zum Teil von Felix während meines Praktikums in einigen Aspekten abgeändert wurde, was wiederum die Lernmuster mit den Eingabemustern in EVOQT inkonsistent machte, so daß dort Platzhalter für feetback−bits hinzugefügt werden mußten.
Auch beim Programmieren des Datengenerators ergaben sich für mich Schwierigkeiten. Zum einen dadurch, daß mir QT nicht vertraut ist, und ich insbesondere mit der Umwandlung von QT−Datenstrukturen in C++− Datenstrukturen große Schwierigkeiten hatte. Weil ich zudem in der objektorientierten Programmierweise ohnehin nicht allzu erfahren bin, fiel es mir schwer die Übergabe solcher Datenstrukturen ohne das Zurückgreifen auf Zeiger zu bewältigen. Die Gesamtheit vieler derartiger Hindernisse machte diese ursprünglich als Ferienpraktikum gedachte Angelegenheit dann etwas langwierig. Trotz dieser Schwierigkeiten ging von meiner anfangs geradezu mystifizierenden Begeisterung für neuronale Netze − von denen ich zunächst noch nicht einmal wußte, was sie sind − nichts verloren. Im Gegenteil: So mühsam auch der Weg zu einer technischen Nutzung im größeren Rahmen sein mag, er scheint sich so lohnen. Vor allem bin ich nach der Lektüre der „Theorie der neuronalen Netze“ von Rojas in meinem Glauben bestärkt, daß diese interdisziplinäre Wissenschaft auch einen erkenntnistheoretischen Gewinn bringen kann. Kennzeichnend bei der Frage der Codierung der Lerndaten war das intuitive Vorgehen. Bei der Mustererkennung, wie im Falle der handgeschriebenen Ziffern, kapituliert ein analytisches Vorgehen bereits vor der Komplexität des Problems. Einen Algorithmus zu finden, scheint fast unmöglich. Doch die einfache Frage, „wie − also bei welcher Codierung − würde ich selbst eine Systematik am ehesten erkennen ?“, führte zu den besten Ergebnissen. Diese tatsächliche Ähnlichkeit des Netzwerkes zu meiner eigenen Denkweise finde ich außerordentlich faszinierend.
Mag es auch unüblich sein, so möchte ich hierbei dennoch abschließend erwähnen, daß sich die Arbeitsatmosphäre in Hilfsbereitschaft und Freundlichkeit der Anwesenden auszeichnet, und mir das Praktikum sehr viel Freude bereitet hat.

Anhang zu Evodatagen

In der Menüleiste findet man unter Tools die Auswahl Graphic. Dies ist der Programmteil, mit dem man in zuvor gewählter Auflösung Schwarzweißmuster aus einer Bilddatei ausschneiden kann und diese in einer Patterndatei für EVOQT speichern kann. Die Auflösung wird in der Datei in Form von drei Kommentarzeilen gespeichert. Im Feld Value kann man das zu dem graphischen Muster gewünschte Ausgabemuster angeben. Die Codierung des Ausgabemusters findet man in der Methode „**void wgCGraficInputImpl::slotSave()**“ Mit Painter kann man selbst Muster zeichnen und sich den Binärcode anschauen, um dieses gegebenfalls mit Copy und Paste EVOQT vorzulegen. Mit dem Button OPEN kann man eine Patterndatei öffnen [**void wgCDeviceImpl::slot_openpatfile()**]und durch die Muster browsen [**void wgCDeviceImpl::slot_nextpat() und void wgCDeviceImpl::slot_prevpat()**].Diese werden graphisch dargestellt.[**void wgCManualInputControlImpl::set_bincode()**]