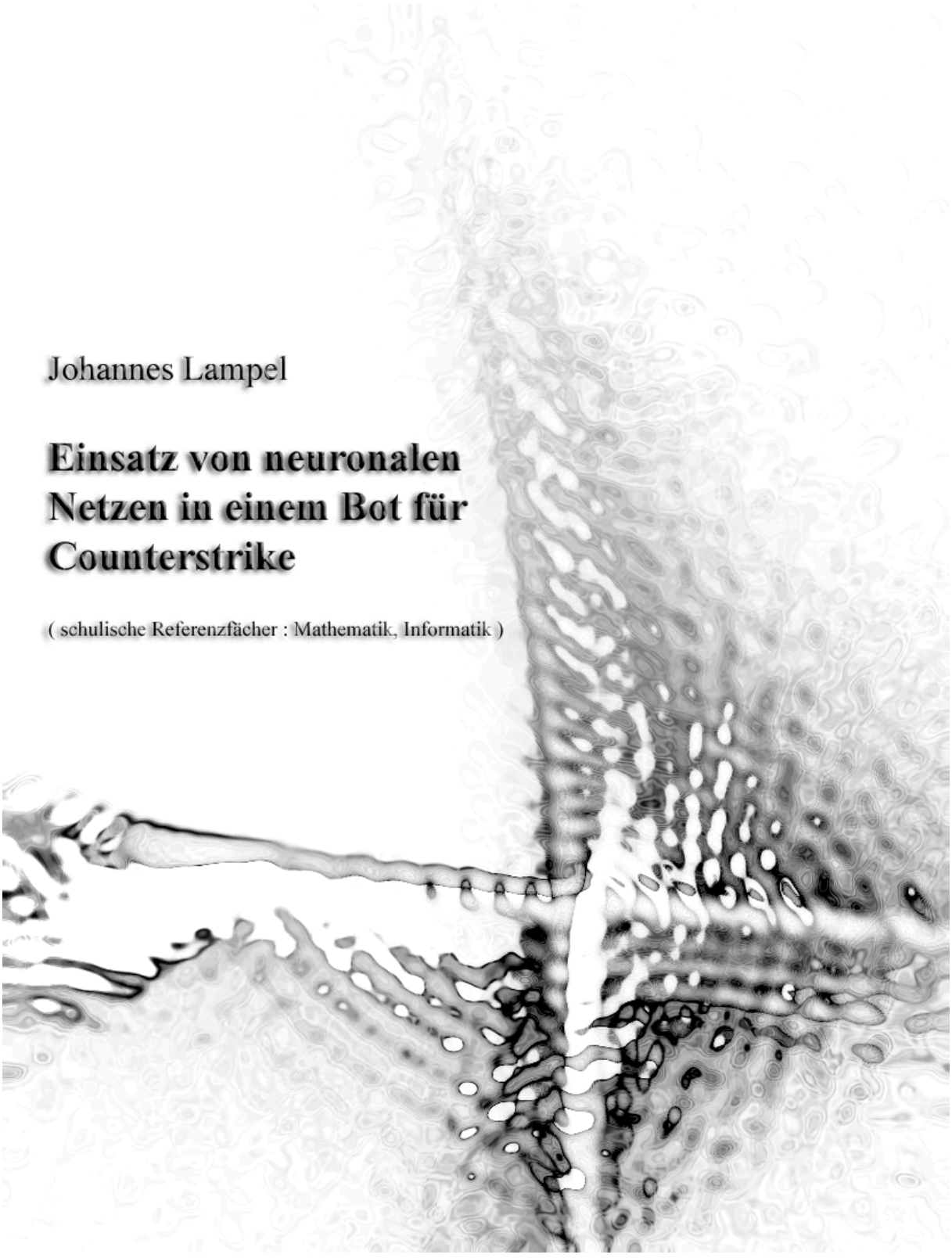
Johannes Lampel

Einsatz von neuronalen Netzen in einem Bot für Counterstrike

(schulische Referenzfächer : Mathematik, Informatik)



Schule :

Friedrich Schiller-Gymnasium Preetz / Holst.

Am Ihlsol 10-12

24211 Preetz

Name des Schülers :

Johannes Lampel

Thema der besonderen Lernleistung :

Einsatz von Neuronalen Netzen in einem Bot für Counterstrike

Betreuende Lehrkraft :

Frau Dr. E. Rabe

Termin der endgültigen Festlegung des Themas :	Februar 2001
Termin der endgültigen Formulierung des Themas :	14. Nov. 2001
Abgabetermin :	16. Nov. 2001
Bewertung :	

_____

Unterschrift des Schülers

_____

Unterschrift der Fachkraft

Info zum Titelbild: Das Bild ist entstanden, indem alle Gewichte der Neuronen einer SOM mit einem 2 dimensionalen Eingabevektor während der Lernphase aufgezeichnet wurden. Die SOM sollte Eingaben lernen, die gleichmäßig über ein quadratisches Gebiet verteilt waren. Die Kreuzform entstand durch einen topologischen Defekt. Das Bild wurde im Nachhinein noch mit einem Bildbearbeitungsprogramm verändert. Das Originalbild findet sich am Ende der Arbeit.

Johannes Lampel

Einsatz von Neuronalen Netzen in einem Bot für Counterstrike

Besondere Lernleistung mit den Referenzfächern Informatik, Mathematik

Inhalt

INHALT 4

1. EINLEITUNG 6

1.1 Was sind Bots? 6

2. THEORIE NEURONALER NETZE 9

2.1 Prinzipieller Aufbau biologischer neuronaler Netze 9

2.2 Prinzipieller Aufbau künstlicher neuronaler Netze 10
2.2.1 Interne Funktionen eines Neurons 12
2.2.2 Funktionen des Netzes 13
2.2.3 Aufbau eines Netzes 13

2.3 Lernen 15
2.3.1 Überwachtes Lernen (Supervised Learning) 15
2.3.2 Bestärkendes Lernen (Reinforcement Learning) 16
2.3.3 Unüberwachtes Lernen (Unsupervised Learning) 17

2.4 Feedforward-Netze 18
2.4.1 Hebbsche Lernregel 18
2.4.2 Delta Regel 18
2.4.3 Backpropagation 19
2.4.3.1. Herleitung der Backpropagation Regel 19
2.4.3.2. Momentum-Term 24
2.4.3.3. Flat Spot Elimination 24

2.5 SOM (Self organizing map) 26
2.5.1 Lernverfahren 27

3. PRINZIPIELLER AUFBAU DER SIMULATOREN 31

3.1. Simulator für Feedforward-Netze (LSNNSI) 31
3.1.1 Aufbau 32

3.2. Simulator für Self-Organizing-Maps (LSSOMSim) 35
3.2.1 Aufbau 35

4. PRAKTISCHER EINSATZ DER NEURONALEN NETZE IM BOT 36

4.1 Aufgaben des neuronalen Netzes 42

4.2 Gewinnung der Trainingsdaten 44

4.3 Technische Einzelheiten bei der Implementierung des Bots 52
4.3.1 Einbindung in Halflife 52

ANHANG A: LITERATUR 55

ANHANG B : QUELLTEXTE 56

B.1 Quelltext des (Feedforward-) NN-Simulators 56
NeuralNet.h 56
LinkBaseDef.h 57
LinkBase.h 58
ListeDef.h 59
Liste.h 61
ListeElemDef.h 67
ListeElem.h 68
NeuralNetBaseDef.h 69
NeuralNetBase.h 71
NeuralNetBPropDef.h 74
NeuralNetBProp.h 77
NeuronAktDef.h 88
NeuronAkt.h 89
NeuronBaseDef.h 90
NeuronBase.h 91
WeightBaseDef.h 92
WeightBase.h 93

B.2 Quelltext des SOM-Simulators (komplett) 94
CqSOM2d.h 94
CqSOM2d.cpp 96
fNBorKernel.h 101
fNBorKernel.cpp 102
GlobalCqSOMDefs.h 103
nSOMErr.h 104
nVec.h 105
nVec.cpp 107
nVecErr.h 111
SOM.h 112
SOMPattern.h 113
SOMPattern.cpp 114
SOMPatternElem.h 116
SOMPatternElem.cpp 117
SOMPatternErr.h 118

B.3 Relevante Teile des Bot Quelltextes 119
CBotBase.h 119
BotNNDefs.h 126
CBotCS_combat.cpp 131

ANHANG C : ERKLÄRUNG 137

ANHANG D: VERBREITUNG DES BOTS 138

1. Einleitung

Der folgende Text behandelt zuerst die Grundlagen der Simulation künstlicher neuronaler Netze, um dann anschließend auf eine Anwendung von neuronalen Netzen, die Steuerung von einem Bot, zu kommen. Es werden nur Feedforward-Netze und **S**elf **O**rganizing **M**aps (SOM) behandelt, da nur sie beim Bot zum Einsatz kommen.

Der theoretische Abschnitt beschäftigt sich mit der Motivation bestimmte Prozesse zu simulieren und geht kurz auf die biologischen Vorbilder ein. Danach folgt eine Beschreibung des Aufbaus neuronaler Netze inklusive ihrer Komponenten. Nach einer kurzen Diskussion verschiedener Ansätze des Lernens folgt die Beschreibung verschiedener Lernverfahren für Feedforward-Netze, wobei der Schwerpunkt bei Backpropagation und seinen verschiedenen Abänderungen liegt. Als Beispiel für ein unüberwachtes Lernverfahren werden die selbstorganisierenden Karten von Kohonen beschrieben.

Der 3. Teil geht auf die Realisierung der Simulation von neuronalen Netzen auf Computern nach. Meine Simulatoren für Feedforward-Netze und SOMs werden einzeln vorgestellt, weil sie auf verschiedenen Strukturen basieren. Es werden die wichtigsten Datenstrukturen und ihre Funktionen vorgestellt. Gegen Ende des 3. Teils gehe ich noch einmal kurz auf die Kompromisse ein, die ich machen musste, um vernünftige Geschwindigkeiten zu erreichen. Alle Quelltexte sind in C/C++ geschrieben.

Im 4. Teil wird dann der praktische Einsatz neuronaler Netze beschrieben. Neben der Aufbereitung der Daten, die der Bot aus seiner ‚Umgebung' bezieht und der nachfolgenden Nutzung des neuronalen Netzes wird die Gewinnung von geeigneten Trainingsdaten erläutert, wie auch technische Einzelheiten.

Anhang B beinhaltet die Quelltexte der Simulatoren und NN-relevante Teile des Botquelltextes.

(Anmerkung: Seit dem Beginn meiner Arbeiten an meinem Bot ist mittlerweile eine zweite und dritte Version der HPB-Template erschienen. Mein Bot basiert auf der Version1 und ich beziehe mich in diesem Text, wenn ich vom Templatecode spreche auf diese Version.)

1.1 Was sind Bots?

Bots sind computergesteuerte Mitspieler in Computerspielen, die eine Mehrspieler-Unterstützung besitzen. Bots werden programmiert, damit es den Anschein hat, als ob der

Spieler mit menschlichen Spielern spielt. Dies ist sonst nur auf LAN-Partys[1] oder im Internet möglich. Ich beziehe mich in diesem Text, wenn ich von Bots spreche auf Bots für Counterstrike (CS), eine Erweiterung von Halflife (HL). HL wurde von [Valve] programmiert.

Hier hat sich mittlerweile eine größere Gemeinde von Programmierern gebildet, die Bots auf der Basis von HL programmieren. Als Grund für diese Beliebtheit ist zum einen die freie Verfügbarkeit des Halflife Quellcodes als auch die große Beliebtheit von Counterstrike als Mehrspieler-Spiel im Internet und auf LAN-Partys zu nennen. Halflife von somit auch Counterstrike sind sogenannte First-Person-Shooter, d.h. man spielt aus der Sicht einer Spielfigur in einer in diesem Fall realistischen 3D-Welt und muss Gegner mit Waffengewalt töten oder Missionsziele erreichen. Bei Counterstrike spielen 2 Gruppen, die Terroristen und die Counter-Terroristen gegeneinander, wobei die Counter-Terroristen Geiseln befreien, Bomben entschärfen müssen. Die Terroristen versuchen natürlich mit allen Mitteln dies zu verhindern. Durch diese Aufgaben ist ein wenig taktisches Vorgehen empfehlenswert, weshalb diese Bots auch schwierig zu programmieren sind.

Gerade nach den Ereignissen am 11. September stellt sich einem natürlich schon die Frage, ob es richtig ist, ein brutales Spiel, das zudem einen solchen Hintergrund hat, zu unterstützen, indem man es selber erweitert. CS ist in diesem Zusammenhang besonders zu nennen, da es den Anspruch erhebt, z.B. von der Wahl der Waffen realistisch zu wirken, einige Karten, die standardmäßig mit dabei sind, eindeutig in einer arabischen Umgebung spielen und es sogar ein Spielermodel bei den Terroristen gibt, das einen arabischen Kämpfer darstellt. Dem gegenüber sind die Spielermodels der Counter-Terroristen nach dem Vorbild der SAS, der GSG-9 und den Navy Seals nachempfunden. Entscheidend für die Wahl des Spiels war damals für mich jedoch, dass die Programmierschnittstelle offen ist, es einige Quellen über diese gibt, und das ich mich somit nicht um die Simulation der ‚Außenwelt' kümmern musste. Dies hatte bei vorhergehenden Projekten immer eine Menge Arbeit bedeutet. Außerdem spielte ich damals gerne CS auf LANs und war der Meinung, dass ich selbst auch einen Bot schreiben könne, der mindestens so gut wie die damaligen Bots war, auf die ich, wenn ich CS spielen wollte, angewiesen war.

[1] LAN – Partys sind Veranstaltungen, zu denen Menschen, die Computer spielen wollen, ihre Rechner mitbringen, sie vernetzen und somit gegen Gegner oder mit Mitspielern, die von Menschen gesteuert werden, spielen können. Das Spielen mit menschlichen Mitspielern ist oft reizvoller als das Spiel gegen gängige Computergegner, da diese oft nur wenige oder keine Taktiken besitzen, die Mitspieler jedoch auf den Einsatz von wiederkehrenden Taktiken reagieren.

Viele der aktuellen Bots, auch meiner, basieren auf einem Template-Quellcode von [botman], der keine KI-Elemente enthält, aber die nötigen Funktionen zur Verbindung mit dem eigentlichen Spiel beinhaltet. Auf die technischen Realisation gehe ich in Kapitel 4 ein.

2. Theorie neuronaler Netze

Bevor man sich mit künstlichen neuronalen Netzen auseinandersetzt, ist es interessant das biologische Vorbild näher zu betrachten. Deshalb beschäftigt sich dieses Kapitel zuerst mit den biologischen Ursprüngen künstlicher neuronaler Netze, danach dann mit dem Aufbau und den Lernverfahren für diese Netze.
Die Vorteile künstlicher neuronaler Netze liegen in der Fehlertoleranz und der von Natur aus gegebenen Parallelität, die sie für Berechnungen auf Parallelrechnern prädestiniert.

2.1 Prinzipieller Aufbau biologischer neuronaler Netze

In biologischen Systemen übernehmen Nervenzellen die Aufgabe der Kommunikation von Zentralnervensystem und Muskeln bzw. Sinnesorganen. Der Mensch besitzt ca. 100 Milliarden Nervenzellen, wobei sich die größte Anzahl im Gehirn befindet, die Zweitgrößte im Bauchraum.
Die Nervenzellen bestehen meist aus einem Zellkörper, einem Axon und Dendriten. (Ein Spezialfall stellen die Unipolaren Nervenzellen dar, die nur ein Axon, jedoch keine Dendriten besitzen) Mit Hilfe der Dendriten, dünnen, röhrenförmigen und meist stark verästelten Fortsätzen, nimmt die Nervenzelle Reize auf. Über das Axon werden diese Impulse weitergegeben. Das Ende des Axons ist meist mehr oder weniger stark verästelt. An diesem Teil des Axons haben sich die Synapsen gebildet. Die Synapsen wandeln die elektrischen Impulse des Axons in chemische Signale um, die von anderen Nervenzellen wieder durch ihre Dendriten aufgenommen werden können. Aber auch die Bildung von Synapsen zwischen Dendriten oder Axonen kommt vor, wobei der Großteil aber Axone und Dendriten verbindet. Die meisten Zellen besitzen zwischen 2.000 und 10.000 Synapsen, in Extremfällen auch bis zu 150.000 Synapsen. (nach [ZellSNN97])

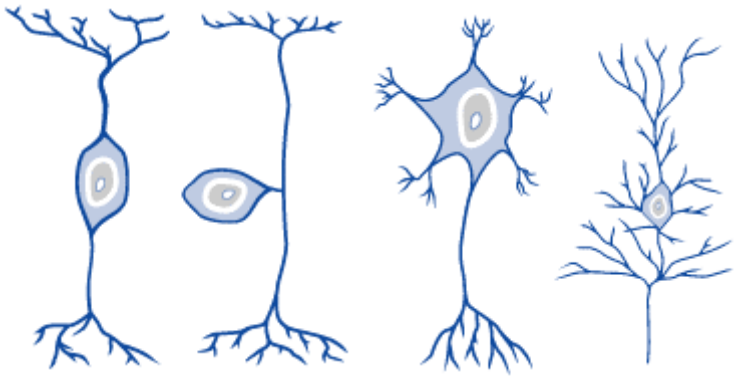


Abbildung 1 : Verschiedene Arten von Neuronen. Von links nach rechts: 1. Bipolare Neuronen, um Neuronen untereinander zu verbinden; 2. Unipolare Neuronen, die sensorische Daten übermitteln; 3. Multipolares Neuron, z.B. bei der motorischen Signalweiterleitung zu finden; 4. Ein pyramidales Neuron, das bspw. im Gehirn zu finden ist.

Bei der Betrachtung des biologischen Vorbilds der Theorien zur Simulation künstlicher Neuronaler Netze ist zu bemerken, dass es hier zwei verschiedene Ansätze gibt. T. Kohonen nennt sie in seinem Buch ([Kohonen]) „scholars“ (engl. für Gelehrte) und „inventors“ (engl. für Erfinder). Die ‚Gelehrten’ gehen mit dem wissenschaftlichen Vorsatz an das Problem heran, dass jede Theorie durch Experimente gestützt werden können muss. Dies führt auf dem Bereich der künstlichen Neuronalen Netze dazu, dass sehr viele Parameter in die Formeln zur Berechnung der Neuronen mit einfließen müssen. Auf der anderen Seite gibt es die ‚Erfinder’, die diesen Anspruch nicht haben und ein Modell mehr oder weniger erfinden und herleiten und damit evtl. auch einige Phänomene simulieren können, obwohl sie nicht wie die ‚Gelehrten’ dem wissenschaftlichen Anspruch der Nachprüfbarkeit beim ursprünglichen Vorbild entsprechen.

2.2 Prinzipieller Aufbau künstlicher neuronaler Netze

Künstliche Neuronen besitzen wie ihre biologischen Vorbilder einen oder mehrere ‚Eingänge’ und einen ‚Ausgang’, zwischen denen eine meist einfache Verarbeitung des Impulse nach einfachen Regeln erfolgt.
Eine wichtige Grundlage ist, dass jeder ‚Eingang’ eines Neurons gewichtet ist, d.h. jedem ‚Eingang’ ist ein bestimmter Wert zugeordnet, der seine ‚Wichtigkeit’ bei der Verarbeitung der Daten darstellt. Dieser Wert der Verbindung von Neuron i zu Neuron j wird mit $\mathbf{w_{ij}}$, die Ausgabe des Neurons i mit $\mathbf{o_i}$ und die gesamte Eingabe eines Neurons mit $\mathbf{net_i}$ bezeichnet.
Die Berechnung der einzelnen Aktivierungen der Neuronen nach dem Anlegen einer Eingabe nennt man Propagierung.

Als Gewichte werden normalerweise sowohl die Werte für die Stärke der Verbindungen w_{ij} als auch die zusätzlich auftretenden Schwellenwerte θ_i bezeichnet.

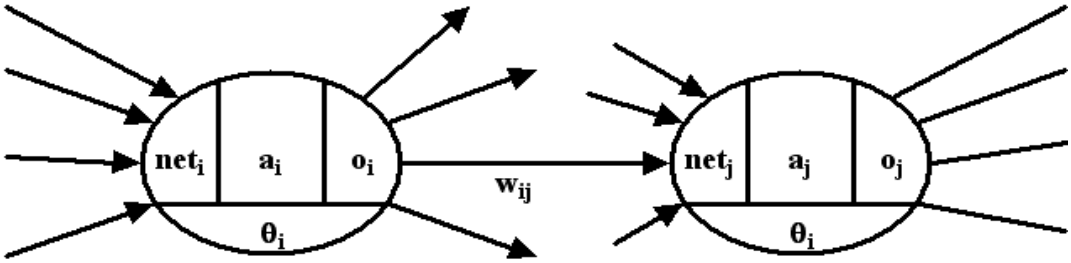


Abbildung 2 : Variablen der Neuronen

In Zusammenhang mit dem biologischen Vorbild sind hier viele Vereinfachungen zu nennen. Die chemischen Prozesse innerhalb der Neuronen wie bspw. der Natrium- und Kaliumionenaustausch werden nicht explizit simuliert und auch die Nachrichtenübertragung an sich wird meist vereinfacht : In der Natur kann man die Aktivität eines Neurons an dessen Frequenz mit der es feuert erkennen, während die Neuronen hier oft kontinuierliche oder auch binäre Werte benutzen. Weiterhin werden die lokalen Nachbarschaftsbeziehungen der Neuronen untereinander, die in der Natur u.a. durch die Diffusion von Neurotransmittern im umliegenden Gewebe stattfindet, in den meisten Modellen nicht simuliert.
Zusammenfassend kann man sagen, dass die Neuronen künstlicher neuronaler Netze meistens sehr stark vereinfacht sind und nur in Einzelfällen halbwegs realistisch simuliert werden, was man sich dann mit dem Nachteil der sehr aufwändigen Berechnung der Netze erkauft.

2.2.1 Interne Funktionen eines Neurons

Der Aktivierungszustand $\mathbf{a_i}$ eines Neurons ist eine interne Variable, die sich aus der Netzeingabe $\mathbf{net_i}$ und dem Schwellenwert $\boldsymbol{\theta_i}$ mit Hilfe der Aktivierungsfunktion berechnen lässt.

$$a_i(t+1) = f_{act}(net_i(t), \theta_i) \qquad \textbf{Formel 2.1}$$

Als Aktivierungsfunktion werden hauptsächlich 3 verschiedene Funktionen benutzt (nach [ZellSNN97])

- Die binäre Aktivierung

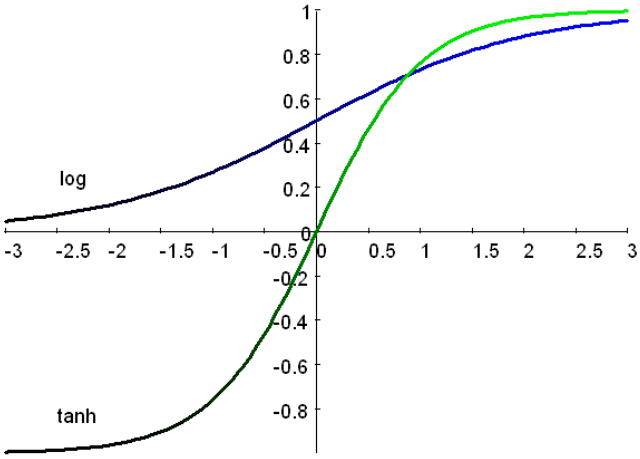
$$o_i = \begin{cases} 1 \text{ falls net}_i \geq \theta_i \\ 0 \end{cases}$$

- Sigmoide Aktivierungsfunktion
  - Die Logistische Aktivierungsfunktion

    $f_{\log}(x) = \frac{1}{1+e^{-(x-\theta_i)}}$ bzw.

    mit Temperatur-Parameter T

    $f_{\log}(x) = \frac{1}{1+e^{-(x-\theta_i)/T}}$



  - Tangens hyperbolicus

    $f_{\tanh}(x) = \tanh(x - \theta_i)$

    Diese Aktivierungsfunktion wird gerne in Zusammenhang mit dem Lernverfahren Backpropagation verwendet, da hier auch ein inaktives Neuron ($\mathbf{o_i} = -1$) im Gegensatz zur logistischen Aktivierungsfunktion nicht 0 ist und somit eine in beide Richtungen Gewichtsänderung möglich ist.

In einigen Fällen wird aus simulationstechnischen Gründen der Schwellenwert durch ein sogenanntes ‚on-Neuron' ersetzt. Dieses besondere Neuron ist immer aktiviert und durch seine gewichteten und durch das Lernen veränderbaren Verbindungen zu jedem Neuron ein Ersatz für die expliziten, internen Schwellenwerte. Bei der Simulation müssen die Schwellenwerte nicht extra berechnet werden, sondern können zusammen mit den anderen Gewichten verändert werden. Allerdings steht diesem Vorteil ein großer Nachteil gegenüber: Bei der Visualisierung der Netze werden diese leicht unübersichtlich.

Auch für die Berechnung der Ausgabe kann es eine eigene Funktion geben, die allerdings oft mit der Aktivierungsfunktion zusammengefasst wird.

$$o_j = f_{out}(a_j)$$ **Formel 2.2**

2.2.2 Funktionen des Netzes

Die Propagierungsfunktion bezeichnet die Vorschrift, mit der die Netzeingabe weiterverarbeitet wird

$$net_j(t) = \sum_i o_i(t) w_{ij}$$ **Formel 2.3**

Die Lernregel ist ein Algorithmus, der bestimmt, wie das neuronale Netz lernt. Hier gibt es mehrere Ansätze, die teilweise recht verschieden sind

2.2.3 Aufbau eines Netzes

Die Neuronen in einem künstlichen neuronalen Netz sind meistens schichtweise angeordnet. Diese künstlichen Netze besitzen eine Eingabe und eine Ausgabeschicht. Um die Fähigkeiten der Netze zu erhöhen, werden oft noch verdeckte Schichten dazwischen eingefügt. Bei der häufigsten Form, den feedforward-Netzen sind die Ebenen meistens vollständig in einer Richtung verbunden, d.h. jedes Neuron der 1. Schicht ist jeweils mit allen Neuronen der 2. Schicht verbunden. Ein Netz mit einer Eingabeschicht, einer verdeckten Schicht und einer Ausgabeschicht wird 2-stufig genannt, da es 2 Schichten trainierbarer Gewichte besitzt.

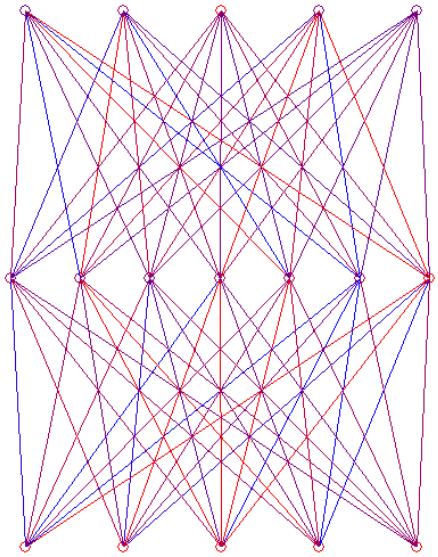


Abbildung 3 : Ein vollständig ebenenweise verbundenes 2-stufiges 5-7-5 Feedforward–Netz

Direkte Verbindungen zwischen der Eingabeschicht und der Ausgabeschicht, also über die verdeckte Schicht hinweg, heißen *shortcut connections* und sind meist ohne größere Änderungen der Simulatorarchitektur hinzuzufügen. Auch rekurrente Netze sind möglich, die jedoch größeren Aufwand in Bezug auf den Algorithmus erfordern und, für ein Ergebnis oft mehrere Propagationen benötigen und teilweise anfangen zu schwingen. Man unterscheidet vier Arten rekurrenter Netze:

- Direkte Rückkopplung
  Durch die direkte Rückkopplung können sich die Neuronen selbst hemmen bzw. aktivieren und somit kleine Eingangssignale in bestimmten Bereichen verstärken.
- Indirekte Rückkopplung
  Verbindungen von höheren Schichten zu niedrigeren können beispielsweise zur Aufmerksamkeitssteuerung eingesetzt werden. Hierbei wird die ursprüngliche Eingabe verarbeitet und mit einer Ausgabe assoziiert, die wieder dazu benutzt werden kann, bestimmte Eigenschaften der Eingabe besonders zu beachten.
- Rückkopplung innerhalb einer Schicht. (lateral feedback)
  Die Rückkopplung innerhalb einer Schicht wird oft dann eingesetzt, wenn nur ein Neuron in der gesamten Schicht aktiv sein soll. Um das zu erreichen, hemmen alle Neuronen alle anderen Neuronen und versuchen sich selbst noch weiter zu aktivieren. Das Neuron mit der ursprünglich höchsten Aktivierung bleibt am Ende als Gewinner, während alle anderen Neuronen nicht mehr aktiviert sind. Deshalb heißen solche Netztopologien auch winner-takes-all Netzwerke.

- Vollständig verbundene Netze
  Bei vollständig verbundenen Netzen ist jedes Neuron mit jedem Neuron verbunden. Bekanntester Vertreter dieser Art sind die Hopfield-Netze, die bei Optimierungsproblemen wie dem Travelling-Salesman-Problem eingesetzt werden können.

(nach [ZellSNN97])

2.3 Lernen

Die Lernfähigkeit neuronaler Netze ist eine der speziellen Eigenschaften neuronaler Netze. Sie benötigen dafür eine Eingabe und meistens auch eine Ausgabe, mit deren Hilfe sie ihre internen Zustände wie Schwellenwerte und Gewichte verändern, um der „erwünschten" Ausgabe möglichst nahe zu kommen. Wenn sie einmal einen Trainingsdatensatz gelernt haben, sind sie auch in der Lage unbekannte Eingaben so zu verarbeiten meist und ein passendes Ergebnis zu erzeugen.
Das Lernen kann bei neuronalen Netzen durch Veränderung der Gewichte, also der Gewichtungen der Verbindungen und der Schwellenwerte, das Erstellen oder Löschen von Verbindungen oder Neuronen erfolgen. In der Praxis werden fast immer nur die Gewichte verändert.
Man unterscheidet 3 Arten des Lernens:

2.3.1 Überwachtes Lernen (Supervised Learning)

Beim überwachten Lernen werden dem Netz von außen sowohl Eingabe, als auch die zugehörige gewünschte Ausgabe präsentiert. Die Aufgabe des Lernalgorithmus besteht darin, die Gewichte und Schwellenwerte so zu verändern, dass nach mehrmaliger Präsentation der Muster das Netz in der Lage ist, zu einer Eingabe die passende Ausgabe zu liefern. Nach dem Lernprozess ist das Netz dann auch in der Lage für neue, unbekannte Eingabevektoren eine geeignete Ausgabe zu finden. Diese Eigenschaft, die Generalisierungsfähigkeit, ist eine wichtige Eigenschaft neuronaler Netze, da ihnen meist nicht alle möglichen Eingaben mit den entsprechenden Ausgaben von vornherein präsentiert werden können.
Ein Nachteil des Überwachten Lernens ist, dass es biologisch nicht plausibel ist, weil in biologischen Systemen keine Institution existiert, die Eingaben zusammen mit der gewünschten Ausgabe liefert.
Beispiel für den Ablauf eines überwachten Lernprozesses:

- Präsentation der Eingabe durch entsprechende Aktivierung der Neuronen der Eingabeschicht
- Propagierung des Netzes, sodass eine Ausgabe erzeugt wird
- Vergleich der Ausgabe mit der gewünschten Ausgabe
- Bei Netzen mit mehreren Schichten: Rückwärtspropagierung der Fehler der Ausgabeschicht zu Neuronen in verdeckten Schichten.
- Änderung der Gewichte, um den Fehler zu minimieren

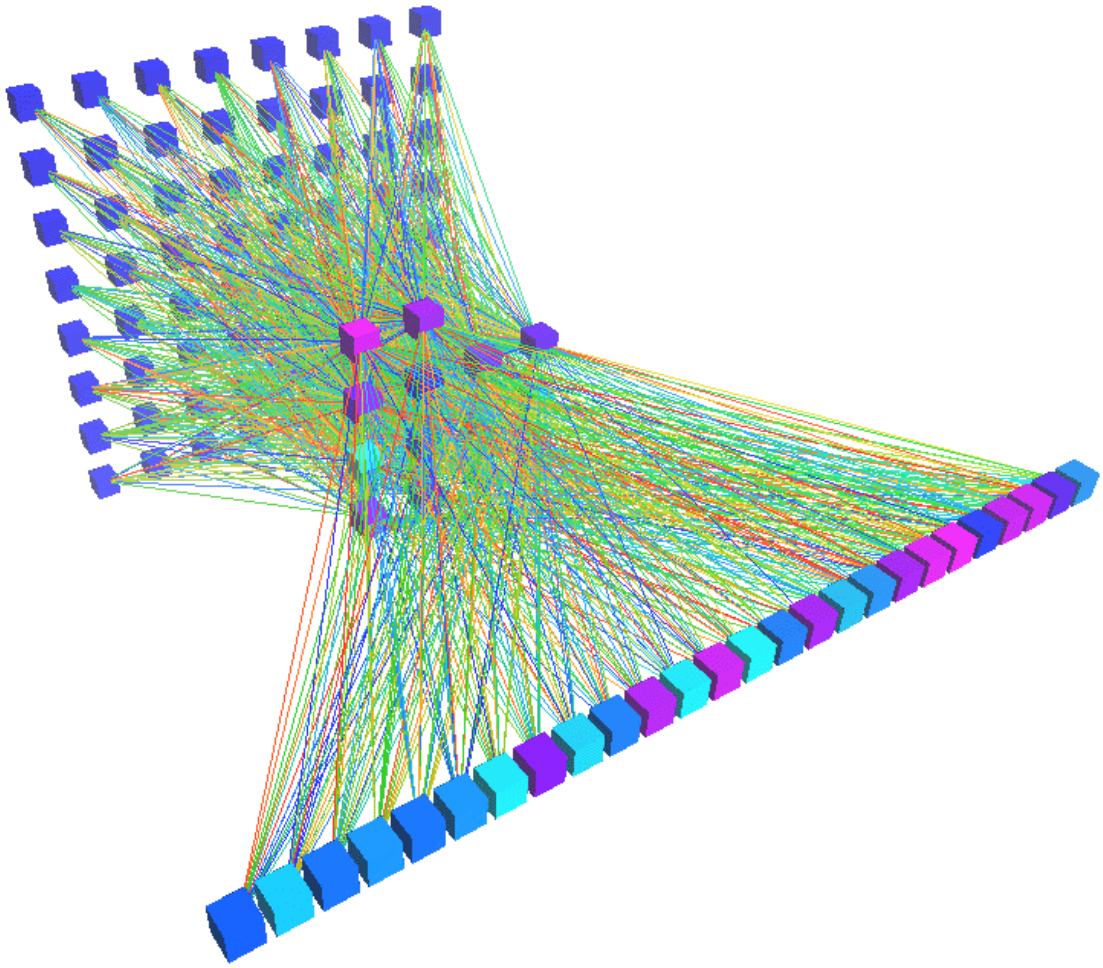


Abbildung 4 : Ein 64-14-26 feedforward Netz, das ich mit Hilfe des überwachten Lernverfahrens Backpropagation zur Erkennung von großen Druckbuchstaben verwendet habe. (die Farben der Verbindungen stellen das Gewicht, die der Neuronen deren Schwellenwert dar)

2.3.2 Bestärkendes Lernen (Reinforcement Learning)

Bei diesem Lernverfahren wird dem Netz zusammen mit der Eingabe nicht der komplette Ausgabevektor, sondern nur wenige Informationen, richtig / falsch oder auch der Grad des Fehlers, präsentiert. Dieses Verfahren ist auch biologisch plausibler als überwachtes Lernen, da Vorgänge wie ‚Erfolg' oder ‚Misserfolg' auch in der Natur zu finden sind. Allgemein ist

dieses Verfahren jedoch sehr zeitaufwändig, da dem Netz nur wenige Informationen zur Verfügung stehen.

2.3.3 Unüberwachtes Lernen (Unsupervised Learning)

Beim Unüberwachten Lernen werden dem Netz eine Trainingsmenge von Eingabevektoren präsentiert, die es ohne direkte Vorgaben von außen in Gruppen einteilt. Die bekanntesten Netze, die nach einem Unüberwachten Lernverfahren arbeiten, sind die Selbstorganisierenden Karten (**S**elf **O**rganizing **M**aps). Bei SOM beeinflussen aktive Neuronen ihre räumlich nahen Nachbarneuronen und bewirken damit, dass sich ähnliche Muster an bestimmten Stellen ‚sammeln', Gruppen bilden. Den SOMs ähnliche Strukturen wurden im visuellen Kortex von Säugetieren gefunden, sodass diese Modelle auch biologisch plausibel sind. Die für dieses Lernverfahren nötige Suche nach dem am stärksten aktivierten Neuron wird allerdings bei der Simulation fast immer durch eine einfache Maximumsuche realisiert, da ein rekurrentes Netz, das die gleiche Aufgabe erfüllen soll, immer mehr Zeit benötigt, einen stabilen Punkt zu erreichen, bzw. keinen findet, also ständig oszilliert.

Beispiel für den Ablauf eines unüberwachten Lernprozesses am Beispiel von SOM:

- Präsentation der Eingabe durch entsprechende Aktivierung der Eingabeneuronen
- Propagierung
- Suche nach dem am stärksten aktivierten Neuron (winner-takes-all)
- Änderung der Gewichte anhand der Entfernung vom zu aktualisierenden Neuron zum Gewinnerneuron

(nach [ZellSNN97])

2.4 Feedforward-Netze

Als Feedforward-Netze werden neuronale Netze bezeichnet, bei denen die Eingabe nur in Richtung der Ausgabeschicht propagiert wird.
η ist in den folgenden Fällen als Lernrate definiert, also eine Variable, die die Geschwindigkeit mit der die Gewichte verändert werden, angibt. Wird η zu klein gewählt, dauert das Training unnötig lange und es findet in lokalen Minima einen stabilen Punkt. Wird die Lernrate jedoch zu groß gewählt, kann es vorkommen, dass der Lernvorgang nicht optimal verläuft.
Die Hebb- und Deltaregeln gelten nur für einstufige Netze.

2.4.1 Hebbsche Lernregel

Die Hebbsche Lernregel ist die Grundlage für viele andere Lernregeln. Sie besagt, dass wenn zwei verbundene Neuronen stark aktiviert sind, das Gewicht der Verbindung erhöht werden soll.

$$\Delta w_{ij} = \eta o_i a_j \qquad \textbf{Formel 2.4}$$

Diese Lernregel wird meistens bei binären Aktivierungsfunktionen benutzt, wobei oft jedoch –1 und 1 als Aktivierungen verwendet werden, um auch eine Verminderung der Beträge der Gewichte zu ermöglichen.

2.4.2 Delta Regel

Bei dieser Lernregel ist die Veränderung proportional zur Differenz δ_i von propagierter Ausgabe o_i(Ausgaben der Neuronen der Ausgabeschicht) von gewünschter Ausgabe t_i. Dies ist eine Erweiterung der Hebbschen Lernregel, wobei auch die propagierte Ausgabe bei der Änderung beachtet wird.

$$\Delta w_{ij} = \eta o_i (t_j - o_j) = \eta o_i \delta_j \qquad \textbf{Formel 2.5}$$

δ_j ist hier der Fehler des Ausgabeneurons j.

Die Delta-Regel ist außerdem ein Sonderfall, der bei 1-stufigen Netzen auftritt, von dem nun folgenden Lernverfahren Backpropagation.

2.4.3 Backpropagation

Das Lernverfahren Backpropagation (BP) ist eine Verallgemeinerung der Delta-Regel für mehrstufige Netze. Hierbei wird der Fehler, der in der Ausgabeschicht direkt berechnet werden kann, zurückpropagiert, d.h. es werden alle Fehler der Neuronen, mit denen das betreffende Neuron in einer höheren Schicht verbunden ist, unter Beachtung der Gewichte der entsprechenden Verbindungen, addiert. Da die Aktivierungsfunktion bei der Vorwärtspropagierung und somit auch bei der Rückwärtspropagierung eine Rolle spielt, muss auch sie mit in die Rechnung einbezogen werden.

Backpropagation :

$$\Delta w_{ij} = \eta o_i \delta_j \qquad \textbf{Formel 2.6}$$

$$\delta_j = \begin{cases} f_{act}'(net_j)(t_j - o_j) \text{ wenn j eine Ausgabezelle ist} \\ f_{act}'(net_j)\sum_k (\delta_k w_{jk}) \text{ wenn j eine verdeckte Zelle ist} \end{cases} \qquad \textbf{Formel 2.7}$$

2.4.3.1. Herleitung der Backpropagation Regel

Backpropagation ist ein sogenanntes Gradientenabstiegsverfahren. Wenn man sich die Fehlerfläche eines neuronalen Netzes bildlich vorstellt, so sucht dieses Verfahren stets den kürzesten Weg ins Tal. Dies tut es indem es sich entgegen der Steigung an dieser Stelle bewegt. Die Fehlerfläche eines Netzes ist aber nicht 2 dimensional, sondern hat so viele Dimensionen wie es Gewichte gibt. (In Anlehnung an [ZellSNN97])

Die Gesamtfehlerfunktion ist wie folgt definiert :

$$E = \sum_p E_p \qquad \textbf{Formel 2.8}$$

Hierbei ist p der Index für die einzelnen Trainingsmuster. Für einzelne Muster lautet sie :

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \qquad \textbf{Formel 2.9}$$

Bei der Gesamtfehlerfunktion des Netzes kann man davon ausgehen, dass man eine feste Mustermenge hat. Damit ist der Funktionswert von E nur noch von den Gewichten abhängig, da nur sie in die Berechnung von o_i eingehen. Das ½ vereinfacht das weitere Vorgehen, da dieses sich nach dem Differenzieren herauskürzt und es egal ist, ob man nun E oder ½ E

minimiert. Weiterhin ist es aus dem gleichen Grund unerheblich ob man den euklidischen Abstand oder dessen Quadrat minimiert.
Da man das Minimum finden will, muss man die Gewichte entgegen dem Gradienten verändern.

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \qquad \textbf{Formel 2.10}$$

Hier begegnet uns wieder η, die Lernrate, die angibt, wie schnell die Gewichte verändert werden sollen. Nun muss die Ableitung der Fehlerfunktion berechnet werden. Zuerst werden wir die Herleitung der Delta Regel betrachten, die einen Sonderfall von Backpropagation darstellt. In diesem Fall gilt :

$$net_{pj} = \sum_i o_{pi} w_{ij} \qquad \textbf{Formel 2.11}$$

$$o_j = net_j \qquad \textbf{Formel 2.12}$$

Die Änderung der Gewichte über alle Muster ist dann :

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \sum_p \frac{\partial E_p}{\partial w_{ij}} \qquad \textbf{Formel 2.13}$$

Mit der Kettenregel folgt :

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ij}} \qquad \textbf{Formel 2.14}$$

Aus der Gleichung für die Fehlerfunktion einzelner Muster folgt :

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj} \qquad \textbf{Formel 2.15}$$

Nun muss noch o_{pj} nach w_{ij} abgeleitet werden :

$$\frac{\partial o_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_i o_{pi} w_{ij} = o_{pi} \qquad \textbf{Formel 2.16}$$

Zusammengefasst lautet oberes dann :

$$\frac{\partial E_p}{\partial w_{ij}} = -o_{pi}\delta_{pj}$$ **Formel 2.17**

Insgesamt ergibt sich somit für die Delta Regel :

$$\Delta w_{ij} = \eta \sum_p o_{pi}\delta_{pj}$$ **Formel 2.18**

Man erkennt dass es sich hierbei um ein offline oder batch-Trainingsverfahren handelt. Ein gravierender Nachteil dieses Verfahrens ist, dass man extra Datenelemente benötigt, die die aufsummierte Änderung speichert. Ein weiterer Nachteil ist, dass man von vornherein alle Trainingsdaten benötigt. Aus diesen Gründen verwendet man in der Praxis oft die Online-Variante, d.h. eine Variante die die Gewichte jeweils nach der Präsentation eines Musters ändert.

$$\Delta_p w_{ij} = \eta o_{pi}\delta_{pj}$$ **Formel 2.19**

Backpropagation stellt nun eine Art Erweiterung der Delta Regel für neuronale Netze mit mehreren trainierbaren Schichten von Gewichten dar.

In in dieser Weise aufgebauten Netzen ist die Berechnung des Fehlers δ nicht mehr so einfach wie im Falle der einschichtigen neuronalen Netze. Man hat keine ‚Sollwerte' für die Neuronen der verdeckten Schicht(en). Aus dieser Tatsache resultiert, dass der Fehler der Ausgabeneuronen zurückpropagiert werden muss um diesen fehlenden Wert zu bekommen (womit auch dann der Name des Lernverfahrens geklärt ist).

Zusammenfassend noch einmal die Propagierungsregeln, auf die später wieder zurückgegriffen werden muss :

$$net_{pj} = \sum_i o_{pi} w_{ij}$$ **Formel 2.20**

$$o_i = f_{act}(net_{pi})$$ **Formel 2.21**

Analog zu der Herleitung der Delta Regel gilt wieder :

$$\Delta w_{ij} = -\eta \sum_p \frac{\partial E_p}{\partial w_{ij}}$$ **Formel 2.22**

Auch hier kann man den abzuleitenden Term mit Hilfe der Kettenregel erst einmal zerlegen :

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$ **Formel 2.23**

Der zweite Term ist dann (mit **2.20**) :

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_i o_{pi} w_{ij} = o_{pi}$$ **Formel 2.24**

Und der Fehler eines Neurons wird auch wie bei der Delta Regel definiert (**2.15**):

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}}$$ **Formel 2.25**

Wenn man jetzt die Formeln (**2.22, 2.23, 2.24, 2.25**) zusammenfasst, erhält man

$$\Delta w_{ij} = \eta \sum_p o_{pi} \delta_{pj}$$ **Formel 2.26**

oder in der online Version :

$$\Delta_p w_{ij} = \eta o_{pi} \delta_{pj}$$ **Formel 2.27**

wie bei dem oben hergeleiteten Lernverfahren, mit dem Unterschied, dass δ aus den oben genannten Gründen komplizierter zu berechnen ist. Und auch hier hilft die Kettenregel wieder weiter :

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}}$$ **Formel 2.28**

Aus Gleichung (**2.21**) folgt :

$$\frac{\partial o_{pj}}{\partial net_{pj}} = \frac{\partial}{\partial net_{pj}} f_{act}(net_{pj}) = f'_{act}(net_{pj})$$ **Formel 2.29**

Nun muss man bei den Neuronen zwischen verdeckten Zellen und Ausgabeneuronen unterscheiden. Im ersten Fall gilt wieder das Gleiche wie bei der Delta Regel :

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj})$$ **Formel 2.30**

Im zweiten Fall kann man den Term mit Hilfe der Gleichungen (**2.20, 2.25**) lösen :

$$-\frac{\partial E_p}{\partial o_{pj}} = -\sum_k \frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial o_{pj}} = \sum_k (\delta_{pk} \frac{\partial}{\partial o_{pj}} \sum_i o_{pi} w_{ik}) = \sum_k \delta_{pk} w_{jk}$$ **Formel 2.31**

k ist in dieser Gleichung ein Index für die Nachfolgezellen von j (i;j;k). In Worten bedeutet dies, dass der Fehler eines Neurons größer ist, wenn die Verbindungen zu den Neuronen mit den größten Fehlern in der nächsthöheren Schicht stark sind. Dies ist auch logisch, da eine fehlerbehaftete Ausgabe dieses Neurons eine größere Auswirkung auf die Ausgabe hat, sofern das Gewicht der Verbindung groß ist.

Quickpropagation ist ein Lernverfahren, das ähnlich wie Backpropagation funktioniert. Hierbei wird mit Hilfe zweier vorhergehender Werte wird die Fehlerfläche mit Parabeln angenähert. In einigen Fällen kann dieses Verfahren das Training um das zehnfache verkürzen. Es kann aber auch vorkommen, dass es schlechter konvergiert.

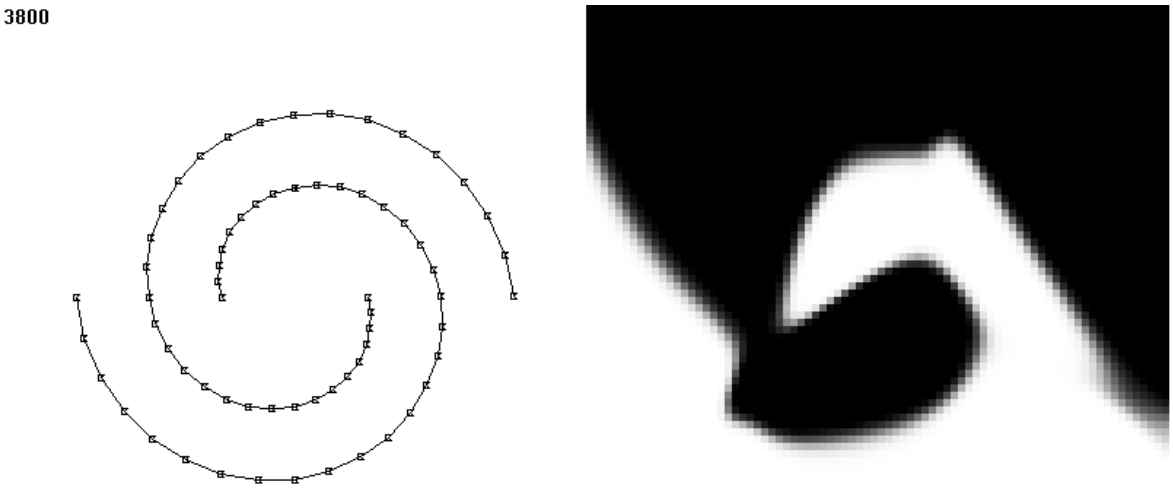


Abbildung 5 : Das 2 Spiralen Problem, ein beliebtes Benchmarkverfahren um die Leistungsfähigkeit eines Lernalgorithmus zu testen. Ein neuronales Netz, das aus zwei Eingabe-, einer unbestimmten Anzahl Schichten verdeckter Neuronen und einem Ausgabeneuron aufgebaut ist, soll anhand der Position eines Punktes, die als Eingabe vorgelegt wird, entscheiden, zu welcher der beiden Spiralen dieser Punkt gehört, durch die schwarzen und weißen Punkte auf der rechten Seite symbolisiert.

2.4.3.2. Momentum-Term

Für einige Trainingsmuster sind Abänderungen teilweise vorteilhaft. Eine solche Erweiterung ist der Momentum-Term. Dieser Term beachtet die letzte Gewichtsänderung bei der neuen Berechnung. Er bewirkt eine Beschleunigung der Gewichtsänderungen auf weiten Plateaus der Fehlerfläche und bremst in zerklüfteten Gebieten ab, da hier die ‚alten' Gewichtsänderungen oft ein anderes Vorzeichen haben.

Gängige Werte für α liegen zwischen 0.5 und 0.9, wobei dieser Wert jedoch auf die einzelnen Trainingsmuster angepasst werden.

$$\Delta w_{ij}(t+1) = \eta o_i \delta_j + \alpha \Delta w_{ij}(t) \qquad \textbf{Formel 2.32}$$

2.4.3.3. Flat Spot Elimination

Ein Problem ist, dass sich Neuronen bei den oben beschriebenen Verfahren in den extremen Aktivierungsbereichen nur langsam ändern können. Die Ursache liegt bei der Ableitung der Aktivierungsfunktion, die bei der Fehlerberechnung benutzt wird. In diesen Bereichen ist die hat die Ableitungsfunktion einen Wert nahe 0. Um dieses Verhalten zu verbessern kann man zu jedem Wert der Ableitung einen festen Wert (0.1 genügt oft schon) addieren. Auch eine feste Zuweisung verbessert teilweise die Konvergenz des Netzes, die variable Variante ist jedoch meist schneller.

In eigenen Versuchen mit einem 10-5-10 De/Encoder konnte ich eine 40% Senkung der zu trainierenden Epochen mit dieser Variante gegenüber der unveränderter Ableitung erreichen.

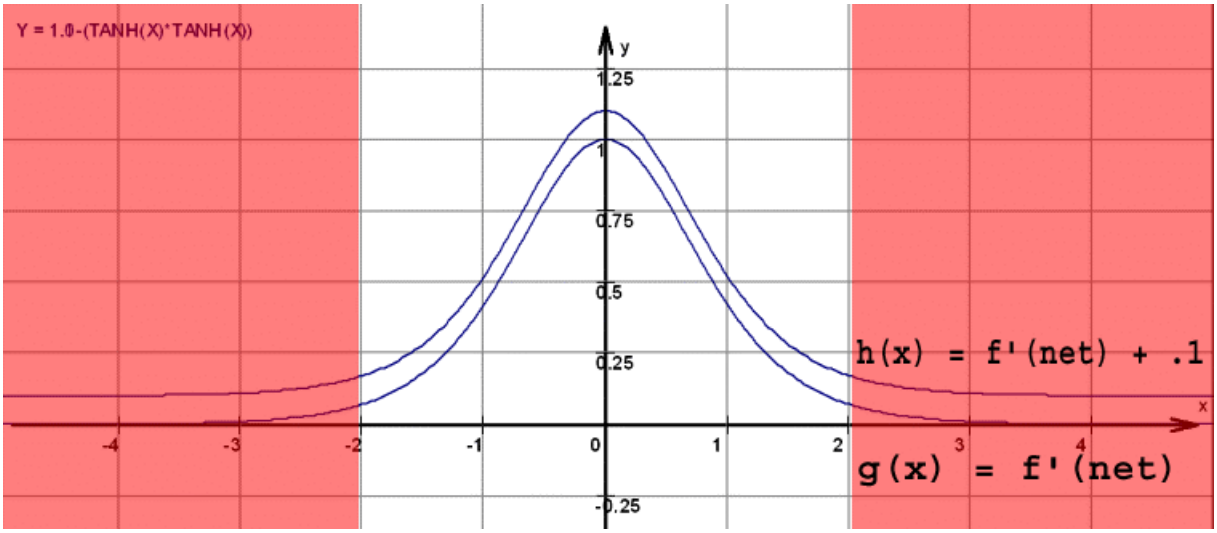


Abbildung 6 : Funktionsgraph der Ableitung der Aktivierungsfunktion ohne (g(x)) und mit Flat Spot Elimination (h(x)). In den roten Bereichen, den extremen Aktivierungsbereichen, wird g(x) sehr klein und somit werden auch die Veränderungen während des Lernvorgangs sehr klein, da $f_{act}(net_j)$ als Faktor bei der Berechnung der Fehler der einzelnen Neuronen vorkommt. Somit ist es für ein Neuron, das sich in einem solchen Bereich befindet, diesen nur langsam wieder zu verlassen. Wenn man zu dieser Funktion immer einen festen Wert hinzuzählt oder auch diese auf einen festen Wert setzt, sind überall ausreichende Gewichtsänderungen möglich. (h(x))

2.5 SOM (Self organizing map)

Selbstorganisierende Karten sind aus einer Eingabeschicht und einer Schicht aktiver Neuronen aufgebaut und werden mit einem unüberwachten Lernverfahren trainiert. Die Neuronen besitzen untereinander Nachbarschaftsbeziehungen, und sie sind in einem n-dimensionalen Gitter angeordnet. Um die Visualisierung zu vereinfachen, werden meist rechteckige oder teilweise auch hexagonale 1,2 oder 3 - dimensionale Gitter benutzt. [Kohonen]

Die Neuronen der aktiven Schicht besitzen jeweils einen Gewichtsvektor W_j, der so viele Dimensionen hat, wie der Eingabevektor E.

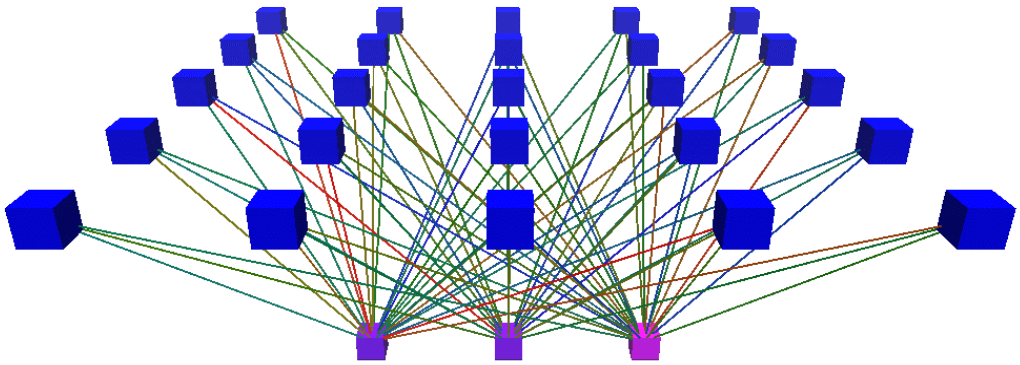


Abbildung 7 : Prinzipieller Aufbau einer SOM. Unten sind die 3 Eingabeneuronen, oben die 25 Neuronen, die die eigentliche selbstorganisierende Karte darstellen. In der Natur besitzen ähnlich aufgebaute Netze noch laterale rekurrente Verbindungen, in Simulatoren, werden diese jedoch oft durch eine Maximumsuche ersetzt. Auch die Eingabeneuronen existieren nicht immer explizit in den Simulatorstrukturen, da die eigentlichen Bestandteile des Netzes die, in diesem Fall, oberen Neuronen sind.

2.5.1 Lernverfahren

Das Lernen läuft so ab, dass dem Netz zuerst ein Trainingsmuster präsentiert wird. Danach folgt eine Art Propagierung, allerdings keine Berechnung einzelner Aktivierungen aus Schwellenwert und Netzeingabe, sondern eine Suche nach dem kleinsten Unterschied zwischen Eingabe und Gewichtsvektor eines Neurons. Zum Vergleich werden meist die euklidische Norm oder das Skalarprodukt bei normalisierten Vektoren verwendet. Danach werden die Gewichtsvektoren zum Eingabevektor hin verändert, wobei mit steigendem Abstand die Stärke der Veränderungen abnimmt.

$$W_j(t+1) = W_j(t) + \eta(t) h_{cj}(t)[E(t) - W_j(t)] \qquad \textbf{Formel 2.33}$$

$$\lim_{t \to \infty} \eta(t) = 0 \qquad \textbf{Formel 2.34}$$

$$h_{cj}(t) = h(z,t) = h'(z,d) \qquad \textbf{Formel 2.35}$$

$$\lim_{t \to \infty} d(t) = 0 \qquad \textbf{Formel 2.36}$$

η(t) ist die zeitlich veränderliche Lernrate. Normalerweise verwendet man eine monoton fallende Funktion, aber auch mit festen Werten habe ich brauchbare Ergebnisse erzielt. $\mathbf{h_{cj}}(t)$ ist die Distanzfunktion. (neighbourhood kernel) **c** in diesem Fall das Gewinnerneuron, **j** das aktuelle Neuron und **z** die Distanz zwischen beiden. $\mathbf{h_{cj}(t)}$ ist auch eine monoton fallende Funktion. Bei ihrer Berechnung wird der interne Wert **d(t)**, der Distanzparameter, der den Radius darstellt in dem die Gewichte der Neuronen verändert werden, mit verwendet. **d** ist am Anfang des Training oft so groß, dass die Gewichte aller Neuronen ausreichend stark verändert werden. [Kohonen] schlägt sowohl für **η(t)** als auch für $\mathbf{h_{cj}(t)}$ lineare Funktionen vor. Ich habe auch gute Erfahrungen mit Exponentialfunktionen gemacht, habe allerdings beobachtet, dass man dann mehr Epochen benötigt um das Netz gut trainiert zu haben.

Beispiele für gängige Nachbarschaftsfunktionen mit[0,1]:

$$h_{gauss1}(z,d) = e^{-(z/d)^2} \qquad \textbf{Formel 2.37}$$

$$h_{cone}(z,d) = \begin{cases} 1 - \frac{z}{d} \text{ falls z < d} \\ 0 \text{ sonst} \end{cases}$$ **Formel 2.38**

$$h_{\cos}(z,d) = \begin{cases} \cos\left(\frac{z}{d}\frac{\pi}{2}\right) \text{ falls z < d} \\ 0 \text{ sonst} \end{cases}$$ **Formel 2.39**

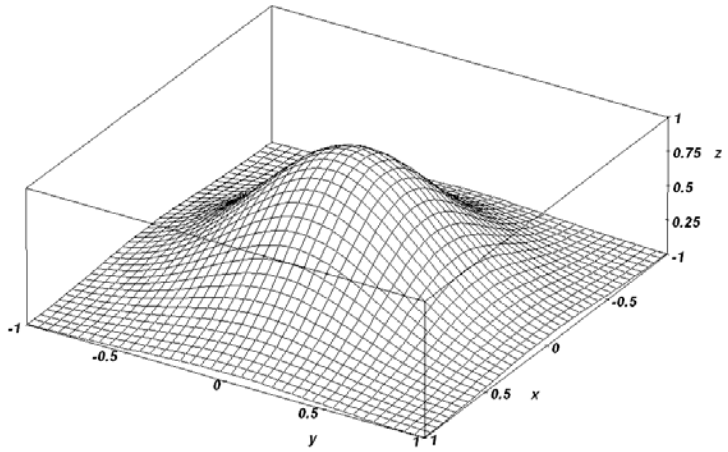


Abbildung 8 : Die Nachbarschaftsfunktion h_{gauss1}

Beispiel für eine Nachbarschaftsfunktion mit [-1,1]:

$$h_{gauss2}(z,d) = \left(1 - \left(\frac{z}{d}\right)^2\right) e^{-(z/d)^2}$$ **Formel 2.40**

Man könnte denken, dass eine solche Karte sich schneller entfaltet, da alle weit entfernten Neuronen ‚abgestoßen' werden. Dieser Fall tritt jedoch nicht ein, sondern die Karte dehnt sich so stark aus, dass keine vernünftigen Klassifikationen mehr erreicht werden können. Auch mit sog. Mexican-Hat Funktionen, also zur y-Achse symmetrische Funktionen, die mit größeren x-Werten erst von 1 auf einen negativen Wert absinken um sich dann der y-Achse anzunähern, habe ich solche Effekte beobachtet.

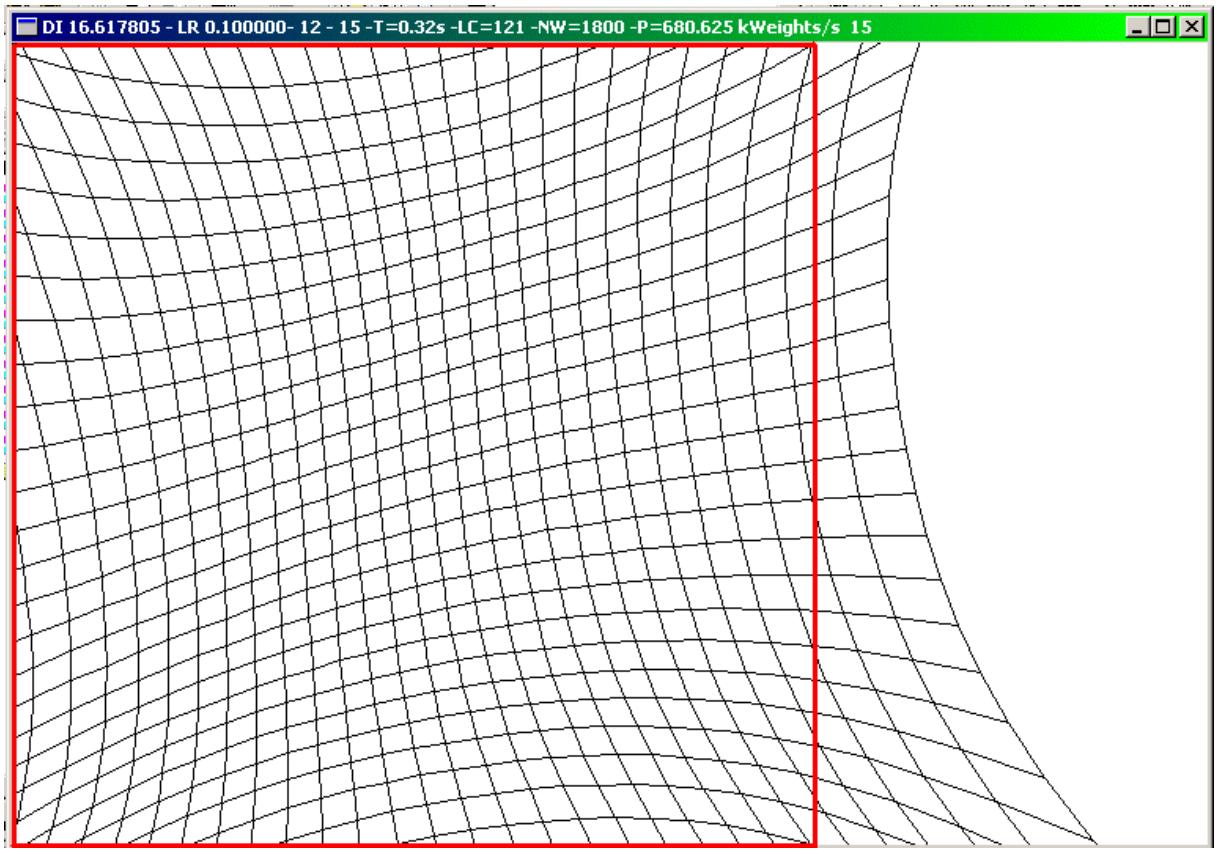


Abbildung 9 : Eine ‚explodierende' Karte nach 15 Epochen, die h_{gauss2} benutzte. Die roten Linien zeigen die Grenzen des Eingaberaums. Nach weiteren Epochen waren nur noch wenige Neuronen sichtbar.

Beispiel einer Entfaltung einer Karte:

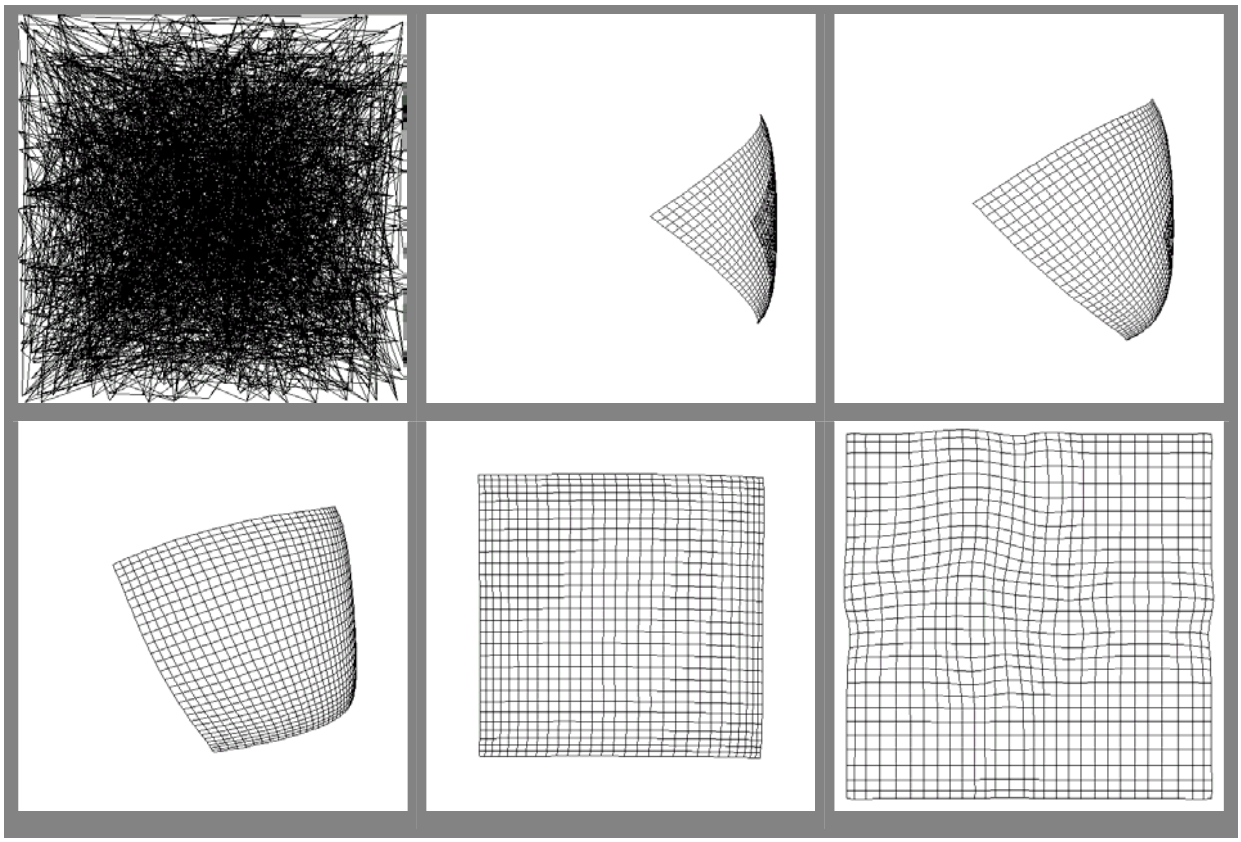


Abbildung 10 : Entfaltung einer Karte (S_x = 30; S_y=30) nach dem Training von 0,5,10,20,40,69 Epochen ($d(t) = 0{,}75S_x - 0.3t$; $\eta(t) = 0{,}1 * 0{,}995^t$) eines das Quadrat gleichmäßig abdeckenden Datensatz. (Epoche -> 1 Präsentation aller zu lernender Datensätze)

Wenn der Parameter d zu klein initialisiert wird, oder zu schnell absinkt, kann es vorkommen, dass sich weiter entfernte Neuronen nicht mehr beeinflussen können und somit nur Teilbereiche korrekt klassifiziert werden können. Diesen Effekt nennt man topologischen Defekt.

Wenn die Lernrate nicht schnell genug abfällt oder konstant ist, kann es vorkommen, das einzelne Neuronen sich noch kurz vor Ende des Trainings zu den ihrem Gewichtsvektor nahe liegenden Trainingsmustern in diese Richtung verändern. Man erhält dann kein ‚glattes' Bild, sondern einige Neuronen scheinen aus ihren Reihen gezogen worden zu sein.

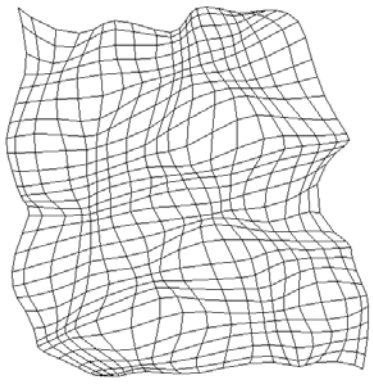


Abbildung 11 : Endform eines Netzes, das mit einer konstanten Lernrate trainiert wurde und die oben genannten Merkmale zeigt.

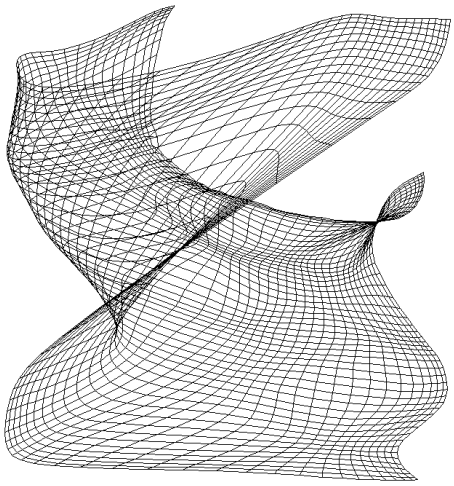


Abbildung 12 : Ein topologischer Defekt (bzw. eigentlich sogar zwei Defekte)

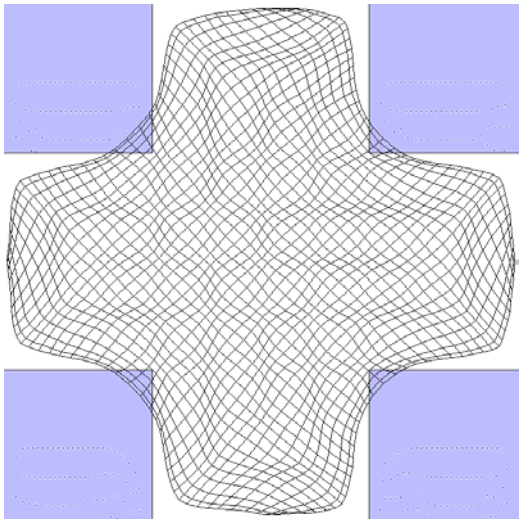


Abbildung 13 : Eine Karte, die einen kreuzförmigen Eingabebereich abbildet. Sie hat keine topologischen Defekte.

3. Prinzipieller Aufbau der Simulatoren

Bei den beiden nachfolgend vorgestellten Simulatoren handelt es sich um 2 vollkommen getrennte Programme, die auch verschieden aufgebaut sind.

3.1. Simulator für Feedforward-Netze (LSNNSI)

Dieser Simulator simuliert Feedforward-Netze, die mit Backpropagation oder seinen Modifikationen Momentum und Flat Spot Elimination trainiert werden. Auch Time Delay NN (TDNN) werden unterstützt. Der Simulator ist so ausgelegt, dass auch während des Trainings ohne Probleme Neuronen und Verbindungen hinzugefügt werden können. Neben den bereits existierenden Aktivierungsfunktionen können auch eigene eingebunden werden.

Alle mit diesem Simulator gewonnen Daten können in Dateien gespeichert werden. Normalerweise ist dies ein Binärformat, Trainingsdaten können aber auch in einem für Menschen lesbaren Format gespeichert werden.

Es gibt eine einfache im Aufbau befindliche beschreibende Sprache, wobei allerdings bis jetzt Trainingsmuster in Dateien vorliegen müssen und nicht innerhalb der Sprache generiert werden können.

3.1.1 Aufbau

Tabelle 1 : Hauptklassen des Simulators für Feedforward-Netze

Klasse	Bedeutung / Funktion	Dateien
NeuronBase	Dieser Datentyp beinhaltet alle grundlegenden Variablen eines Neuronen in einem Feedforward-Netz. Neben den Variablen für die Aktivierung, dem Schwellenwert, dem zurückpropagierten Fehler, der Netzeingabe, der Position im Netz und Zeiger auf die Aktivierungsfunktion und ihre Ableitung gibt es je eine Liste mit Zeigern auf die mit diesem Neuron verbundenen Neuronen und Indices für die dazugehörigen Gewichte. Es gibt eine Liste für die Neuronen, von denen Signale kommen und zu denen Signale ‚gesendet' werden.	NeuronBase.h
NeuronBProp	Dieser Datentyp ist vom Datentyp NeuronBase abgeleitet. Er erweitert ihn um die Propagierungsfunktion und die Funktion zur Berechnung des Fehlers δ.	NeuralNetBProp.h NeuralNetBPropDef.h
LinkBase	Eine Datenstruktur, die einen Zeiger auf ein Neuron und einen Index für das dazugehörige Gewicht in der Liste der Gewichte enthält.	LinkBase.h LinkBaseDef.h
WeightBase	Diese Klasse besitzt einen Zahlenwert für das Gewicht und einen Zähler für die Anzahl der Benutzungen in einem Netz, um gekoppelte Gewichte zu ermöglichen.	WeightBase.h WeightBaseDef.h
WeightBProp	Dieser Datentyp ist von WeightBase abgeleitet, aber hat keine Funktionen oder Variablen, die ihn erweitern	NeuralNetBProp.h NeuralNetBPropDef.h
NeuralNetBase	Basisklasse für alle in diesem Simulator verwendeten Typen neuronaler Netze. Sie liefert Versionsinformationen und definiert mit Hilfe rein virtueller Funktionen, welche Funktionen in einer	NeuralNetBase.h NeuralNetBaseDef.h

	abgeleiteten Klasse implementiert werden **müssen**. Zudem enthält sie Funktionen zur Berechnung der Gesamtfehler des Netzes.	
NeuralNetBProp	Klasse, die ein vollständiges NN beinhaltet. Ein Feld enthält die Listen der Neuronen auf jeweils einer Schicht. Außerdem sind alle Gewichte der Verbindungen in einer Liste gespeichert.	NeuralNetBProp.h NeuralNetBPropDef.h

(Alle relevanten Klassen besitzen Funktionen, die das Speichern der Daten in Dateien unterstützen)

Die Klassen NeuralNetBPropM für Backpropagation mit Momentum-Term und NeuralNetTDNNBPropM für Time Delay Neural Networks mit Momentum-Term sind größtenteils mit NeuralNetBProp identisch. Der gesamte Simulator ist in C++ geschrieben, jedoch habe ich nicht ganz streng objektorientiert, was man daran sieht, dass NeuralNetBPropM nicht von NeuralNetBProp abgeleitet ist. Als ich den Simulator geschrieben habe, habe ich diesen Aufbau aus Performancegründen gewählt.(Dies wäre sicherlich durch Zeiger die auf erweiternde Datenelemente zeigen oder durch Anwendung von Polymorphismus zu lösen gewesen, jedoch hatte ich zu dem Zeitpunkt noch nicht die nötige Erfahrung.) Ein großer Teil der Arbeit an dem Simulator bestand darin, die Geschwindigkeit zu optimieren. Mit Hilfe vieler Veränderungen war es dann letztendlich möglich, die Geschwindigkeit von ungefähr einem kCUPS[2] auf zwei MCUPS zu steigern.

Die Gewichte der Verbindungen des neuronalen Netzes werden in einer Liste gespeichert, die Verbindungen besitzen nur einen Gewichtsindex. Die Gewichte als oft benutzte Daten liegen so zusammen und können schnell erreicht werden.

Am Anfang der Entwicklung des Simulators benutzte ich noch einfach verkette Listen. Diese Listen hatten jedoch bei einer größeren Länge den Nachteil, dass sie extrem langsam wurden. Daraufhin habe ich eine Listenklasse entwickelt, die auf verketteten Feldern basiert. Dieser Ansatz hat zwar den Nachteil, dass oft viel Speicher reserviert wird, dafür aber umso schneller ist. Man kann die Größe der Felder in der Liste anpassen, sodass man ein Mittelmaß zwischen Speichernutzung und Geschwindigkeit finden kann. Eine Festlegung der Schichtgrößen der Netze von vornherein, bzw. bei Anfang des Trainings wollte ich nicht vornehmen um auch während des Trainings das einfache Hinzufügen von Neuronen und

[2] CUPS : Connection Updates Per Second : Anzahl der Verbindungen die pro Sekunde beim Lernvorgang verändert werden können.

Verbindungen zu ermöglichen (zur einfachen Realisierung von Cascade-Correlation, einem Lernverfahren, dass auf dem Verändern von Gewichten bei gleichzeitigem Hinzufügen von Neuronen basiert). Als Feldgrößen bieten sich 2er Potenzen an, da diese von den meisten Compilern als einfache Verschiebeoperationen übersetzt werden, die wesentlich schneller als Divisionen oder Multiplikationen und Verschiebungen auszuführen sind.

Tabelle 2 : Beispiele für die Berechnung der Nummer des Feldes bei gegebenem Index des zu suchenden Objekts bei verschiedenen Feldgrößen

Berechnung mit Hilfe des Verschiebeoperators SAR und SHL, der 3 Zyklen benötigt (Feldgröße $2^{10} = 1024$)	Berechnung mit dem mit Verschiebe, Additions- und Multiplikationsbefehlen (Feldgröße 875)

`mov esi, DWORD PTR [ebx+8]`
`cdq`
`and edx, 1023`
`push edi`
`add eax, edx`
`mov edi, ecx`
`sar eax, 10`
`mov edx, eax`
`shl edx, 10`
`sub edi, edx`
	`mov eax, 628292359`
	`dec ecx`
	`push ebp`
	`imul ecx`
	`sar edx, 7`
	`mov eax, edx`
	`push esi`
	`shr eax, 31`
	`add edx, eax`
	`mov esi, DWORD PTR [ebx+8]`
	`push edi`
	`mov edi, ecx`
	`lea eax, DWORD PTR [edx*8]`
	`sub eax, edx`
	`lea eax, DWORD PTR [eax+eax*4]`
	`lea eax, DWORD PTR [eax+eax*4]`
	`lea eax, DWORD PTR [eax+eax*4]`
	`sub edi, eax`

3.2. Simulator für Self-Organizing-Maps (LSSOMSim)

3.2.1 Aufbau

Tabelle 3 : Hauptklassen des SOM Simulators

Klasse	Bedeutung / Funktion	Dateien
nVec	Dieser Datentyp ermöglicht das Rechnen mit n-dimensionalen Vektoren, wie sie bei der Simulation von SOMs benötigt werden. Diese Klasse beinhaltet Funktionen zur Addition, Subtraktion, Normalisierung und Streckung. Außerdem wurde ein spezieller Mechanismus implementiert, der es erlaubt, auf Datensätze zuzugreifen, ohne sie zeitaufwändig kopieren zu müssen.	nVec.h nVec.cpp nVecErr.h
qSOM2d	Diese Klasse ist der Datentyp, der die Daten und Funktionen zur Simulation Self Organizing Maps bereitstellt. Die Schnittstelle wurde auf möglichst wenige Funktionen beschränkt. Durch sie können die Lernrate, der Distanzparameter, das aktuelle Gewinnerneuron und einzelne Gewichtsvektoren abgefragt bzw. festgesetzt werden.	CqSOM2d.h CqSOM2d.cpp NSOMErr.h
SOMPattern	Diese Klasse verwaltet Trainingsdaten für SOMs. Sie können manipuliert und als Datei gespeichert werden.	SOMPattern.h SOMPattern.cpp

(Alle relevanten Klassen besitzen Funktionen, die das Speichern der Daten in Dateien unterstützen)

Die Klasse qSOM2d beinhaltet ein Feld Zeiger auf die Gewichtsvektoren vom Typ nVec. Die Klasse SOMPattern greift über einen internen Zeiger, der einmal zur Laufzeit gesetzt werden muss, auf die Klasse qSOM2d zu, um bspw. eine Epoche zu trainieren. Die Trainingsdaten werden in der Klasse SOMPatternElem gespeichert, die selbst von nVec abgeleitet ist. Während des Trainings werden zusätzlich die Kategorisierungsfehler in einer Variable innerhalb von SOMPatternElem gespeichert, um im Nachhinein eine Aussage über das Lernverhalten der Karte in Bezug auf bestimmte Datensätze machen zu können.

qSOM2d besitzt als Besonderheit zwei Funktionen zur Kategorisierung von Eingabedaten : qSOM2d::Categorize und qSOM2d::CategorizeFast. Die erste Funktion geht alle Neuronen durch, berechnet ihren Abstand zum Gewinnerneuron und passt die Gewichte entsprechend an. Die zweite Funktionen zur Kategorisierung berechnet nur die Neuronen, die sich innerhalb eines Quadrates, dessen Seitenlänge doppelt so groß ist wie der Distanzparameter, und dessen Mittelpunkt im Gewinnerneuron liegt, befinden. Diese Funktion ist bei den Nachbarschaftsfunktionen gut anzuwenden, bei denen der Funktionswert 0 ist, wenn die Distanz größer als der Distanzparameter ist. Bei der sonst gut konvergierenden Nachbarschaftsfunktion f_{gauss1} ist diese beschleunigte Funktion meiner Erfahrung nach nicht ohne eine Verschlechterung der Kategorisierung anzuwenden.
In den oben genannten Funktionen ist noch eine technische Besonderheit zu beobachten : Die Neuberechnung der Gewichte erfolgt nicht mit einer lokalen Kopie der Daten des zu verändernden Gewichts, sondern direkt im Speicher. Dies wird dadurch erreicht, dass die entsprechende Instanz von nVec nicht selber Speicher allokiert, sondern nur einen Zeiger auf die Position der Daten im Speicher und eine entsprechende interne Variable setzt, um später nicht zu versuchen diesen Speicher im Destruktor wieder freizugeben.

4. Praktischer Einsatz der Neuronalen Netze im Bot

Das neuronale Netz des Bots wird (bis jetzt) offline, d.h. außerhalb des Spiels, trainiert. Ein online-Training wäre zwar rein zeitlich möglich, jedoch müssen die Trainingsmuster so sortiert werden, dass bestimmte Muster, die zwar selten auftreten, aber dennoch wichtig sind, gut genug gelernt werden können. Diese Selektierung würde Zeit in Anspruch nehmen und auch eine automatische Generierung der gewünschten Ausgaben ist problematisch : Welchen Zweck sollte dieser Vorgang haben? Die Ausgaben würden entweder nach bestimmten Kriterien ausgewählt werden, könnten also auch von Anfang an offline gelernt werden, oder das NN soll sich an die Art des Spielens der Gegner anpassen. Um Letzteres zu ermöglichen müssten auch Muster gefunden werden, die dem Bot Vorteile bringen könnten. Diese neuartigen, ‚kreativen' Trainingsmuster könnten vielleicht mit Hilfe eines evolutionären Algorithmus, also mit Hilfe von Mutationen und Kreuzungen meist erfolgreicher Bots, gewonnen werden. Ein evolutionärer Algorithmus benötigt auch immer eine sogenannte Fitnessfunktion, die die Fähigkeiten des Bots bewertet. Als Möglichkeit wäre anzunehmen, dass der Wert der Fitnessfunktion von der Anzahl der Frags[3] und der Anzahl der eigenen Tode abhängt. Ein solcher Ansatz ist aber bei Counterstrike nicht aussagefähig, weil dort zum

[3] FRAG : Punkt den ein Spieler bei der Tötung eines Gegners bekommt

einen das Erreichen des Missionsziels hinzukommt, zum anderen auch keine einfache Bewertung der Frags aufgrund der vielen verschiedenen Eigenschaften der verschiedenen Waffen möglich ist. Am ehesten wäre eine Bewertung aufgrund des Ergebnisses mehrerer Runden zu machen um Kreuzungen zu steuern. Da eine Runde bis zu 5 Minuten dauert, ist ein solcher Ansatz nicht praktikabel, weil oft mehr als 50 Generationen entwickelt werden müssen, bis eine merkliche Verbesserung der Eigenschaften des Agents zu beobachten ist. [PfScheier99]

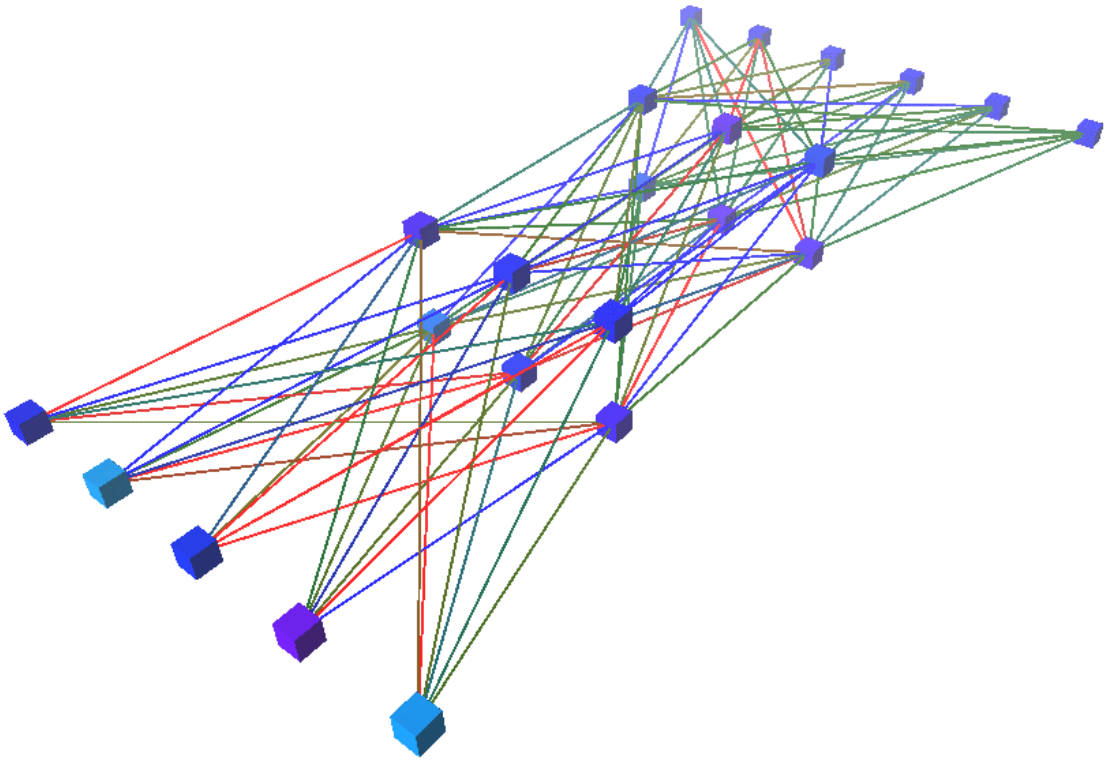


Abbildung 14 : Kampf-NN des Bots (6-6-6-5)

Aufgrund der oben genannten Schwierigkeiten habe ich mich bei diesem Bot für ein einfaches NN für die Kampfentscheidungen entschieden, dass offline trainiert wird. Zurzeit hat das Netz eine Größe von 6-6-6-5 und wird mit dem unmodifizierten Lernverfahren Backpropagation trainiert. Das Netz zur Vermeidung von Kollisionen ist ein 3-3-1 Feedforward-Netz.

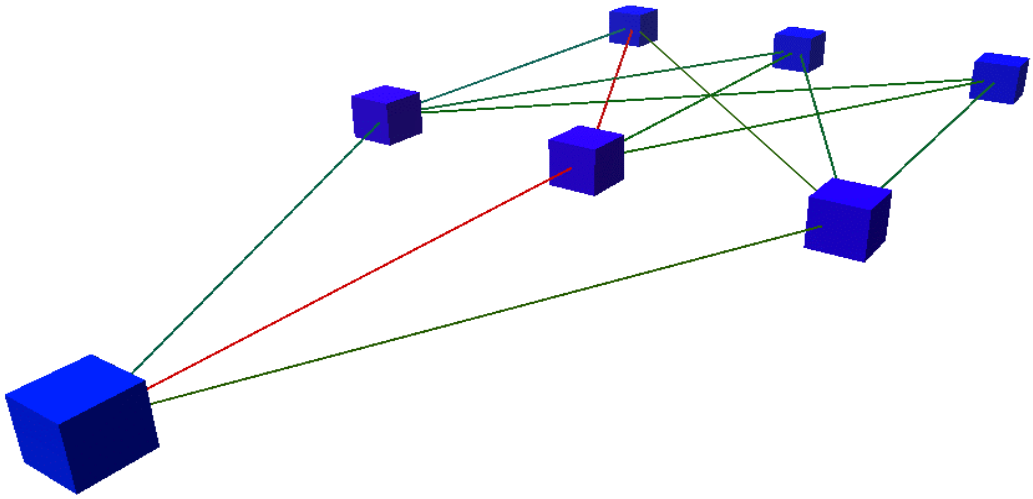


Abbildung 15 : Kollisions- NN des Bots (3-3-1)

Abbildung 16 : Die 3 'Fühler' des Bots, die die Daten für das Kollisionsnetz sammeln. (Hier sind auch die ‚Fühler' aus vorherigen Frames sichtbar.)

Zeitweise habe ich versucht, das neuronale Netz während des Spiels mit Daten aus dem Spiel zu trainieren. Wenn ein Bot bspw. gegen einen Gegner gewonnen hatte oder mit dem Leben davongekommen war, so trainierte ich das Netz mit den Eingabe und Ausgabedaten der letzten maximal 2 Sekunden, wobei die Ausgabedaten verstärkt wurden, d.h. wenn eine Ausgabe den Wert 0.8 hatte, so wurde sie auf 1.0 gesetzt. Für diesen Zweck verwendete ich

ein größeres Netz (5-10-10-5), um zu ermöglichen, dass das Netz mehr verschiedene Situationen lernen kann. Wenn ich das mit den offline gewonnenen Trainingsdaten trainierte Netz als Ausgangsnetz verwendete, so kam es vor, dass die Bots anfingen sehr defensiv zu spielen – sie wurden ja immer ‚belohnt', wenn sie überlebten – oder es kam auch vor, dass die Aktivierung der Ausgabeneuronen immer nahe Null lag, und diese auch nicht mehr stark von der Aktivierung der Eingabeneuronen abhing, obwohl die Verbindungsgewichte nicht alle gleich 0 waren, sondern die Signale sich aufhuben. Dies war vor allem der Fall, wenn das Netz bei dem Tod eines Bots dessen Trainingsdaten verlernen sollte. (Dem Netz wurden die Eingabedaten und die inversen Ausgabedaten zum Lernen repräsentiert. Als Lernverfahren fand Backpropagation Anwendung, was eigentlich ja nicht für diesen Zweck gedacht war). Auch mit einem zufälligen Netz war auch nur eine Konvergenz der Ausgabedaten gegen 0 zu beobachten.

Die einzelnen Verhaltensweisen des Bots stehen in der nachfolgend dargestellten Beziehung :

Hide : Verhaltensweise, die bewirkt, dass sich der Bot versteckt. Die Entscheidung über diesen Sachverhalt trifft das allgemeine Kampf-NN. (*CBotBase.cpp : CBotBase::SearchHidingPlace(edict_t *)* bzw. *CBotCS_combat.cpp / CBotDOD_combat.cpp : CBotCS::Fight(void))*

Combat : In Kampfsituationen werden bis auf die Ausnahme des Versteckens alle Bewegungen durch das Kampf-NN gesteuert. *(CBotCS_combat.cpp / CBotDOD_combat.cpp : CBotCS::Fight (void))*

AvoidTK : Dieser Teil ist dafür verantwortlich, das versehentliche Angreifen von Teammitglieder während Kämpfen zu vermeiden. *(CBotDOD_combat.cpp / CbotCS_combat.cpp : CBotCS::FireWeapon(void))*

SpEnt : (für ‚special Entity') Dieser Teil bewirkt, dass der Bot versucht bestimmte Gegenstände, die für ihn interessant sind, zu erreichen. Dies sind bspw. Geiseln oder auch eine frei herumliegende Bombe. *(CBotCS_navigate.cpp / CBotDOD_navigate.cpp : CBotCS::GoToSpEnt (void) / CBotCS::GoToSpEnt (void))*

Bored : Wenn die Bots lange Zeit nichts 'zu tun' hatten, fangen sie für kurze Zeit an, bspw. verschiedene Dinge aus Langeweile zu zerschießen. *(CBotCS_navigate.cpp / CBotDOD_navigate.cpp : CBotCS::Bored(void))*

Navigate : Das normale Verhalten eines Bots, also das Navigieren und das 'Erforschen' einer Karte, wird durch diesen Komplex gesteuert. *(CBotCS_navigate.cpp / CBotDOD_navigate.cpp : CBotCS::HeadTowardWaypoint (void)* und *waypoint.cpp)*

Wander : Der Bot läuft einfach solange geradeaus, bis er auf ein Hindernis stößt. Dort ändert er seine Richtung längs der Richtung des betreffenden Gegenstands. *(CBotCS.cpp / CBotDOD.cpp : CBotCS::Think(void))*

AvoidCollision : Dieser Teil ist immer aktiv, sofern er nicht explizit durch CBotBase. f_dont_avoid abgeschaltet wurde, was nur in seltenen Situationen wie bspw. bei der Annäherung an einen Scharfschützenposten passiert. *(CBotBase.cpp : CBotBase ::AvoidCollision(void))*

AvoidFall : Dieser Teil verhindert, dass der Bot zu tief fällt und sich verletzt. Dafür wird die Bewegungsrichtung des Bots festgestellt, die Tiefe in einer gewissen Distanz vor dem Bot gemessen und gegebenenfalls die Bewegungsrichtung umgekehrt. Dieser Teil kann durch CBotBase.f_ignore_fall unterdrückt werden. *(CBotCS.cpp / CBotDOD.cpp : CBotCS::Think(void))*

Des Weiteren basieren die Entscheidungen, wie bspw. der aktuelle Zielort des Weges auf den Daten, die im ‚Gedächtnis' des Bots gespeichert werden. Der Bot besitzt 3 ‚Gedächtnisse' :

- Das Kurzzeitgedächtnis, das Positionen, Bewegungen usw. der zuletzt gesehenen Gegner speichert, um auf dieser Basis Vorraussagen zu treffen, wo sich der Gegner befinden könnte. Diese Daten werden maximal 20 Sekunden gespeichert und werden somit nur in der direkten Kampfsituation und direkt danach benutzt.

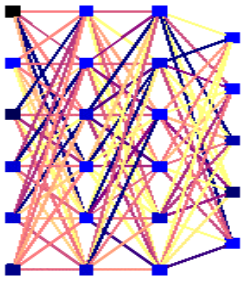
- Das Langzeitgedächtnis speichert bspw. die Orte, an denen Gegner besiegt, Die Zeiten, zu denen die Bombenplätze überprüft wurden oder wo und wann man zuletzt getötet wurde. Dieses Gedächtnis ist nicht zeitlich, sondern in der Anzahl der Ereignisse, die es speichert begrenzt. Diese Anzahl liegt z. Z. bei 10. Diese Daten können maximal so lange bestehen, wie der betreffende Bot am Spiel teilnimmt.
- Das letzte System, das einige Entscheidungsprozesse beeinflusst, sammelt Daten über Kämpfe und deren Ausgang bezogen auf das Waypointsystem. Wenn ein Spieler an einer bestimmten Stelle getötet wird, wird diese Information auf den nächstgelegenen Waypoint bezogen und in einer geeigneten Struktur gespeichert. Diese Daten werden in einer Datei gespeichert und werden somit mit jedem Spiel erweitert.

(Die entsprechenden Datenstrukturen sind in *bot_wpstat.cpp / bot_wpstat.h, stmem.cpp / stmem.h und CBotBase.h zu finden (CSTMem, LTMEM, CWPStat)*)

4.1 Aufgaben des neuronalen Netzes

Das neuronale Netz ist im Spiel für die Bewegungen des Bots in Kampfsituationen und zur Vermeidung von Kollisionen zuständig.

Tabelle 4 : Eingaben und Ausgaben des Kampf-NN des Bots (siehe auch BotNNDefs.h, nntrain.pta)

Eingaben		**Ausgaben**
‚Gesundheit' des Bots		Springen
Entfernung des Gegners		Ducken
Beschr. der Waffe des G.		Verstecken
Beschr. der akt. Waffe		Links / Rechts
Menge der akt. Munition		Laufen / Gehen
Momentane Situation		

Alle Eingabewerte sind im Bereich [-1,1]. Binäre Ausgaben wie bspw. Ducken, Springen oder Verstecken werden als aktiv angesehen, wenn ein bestimmter Schwellenwert überschritten wird. Dieser liegt im Moment bei 0.5 für Springen bzw. Ducken und bei 0.75 in Bezug auf Verstecken.(siehe CBotCS_combat.cpp Zeile 79)

Der Gesundheitswert liegt wie in vielen anderen Computerspielen als Prozentwert vor. Die Entfernung[4] des Gegners wird so in den Bereich von -1-1 eingepasst, dass die Auflösung für den Nahbereich größer ist. (ConvertDistance-bot.cpp-Distanz < 200 -> Distanz / 200-1; Distanz > 200 ∧ Distanz < 1000 -> (Distanz –200) / 800; sonst 1) Die Beschreibung der Waffe sowohl des Bots selbst als auch des Gegners basiert auf fest definierten Werte (siehe NNWeapon.cpp), die die Reichweite einer Waffe bzw. deren Genauigkeit bezogen auf die Entfernung repräsentiert. Das Messer hat in diesem Zusammenhang beispielsweise den Wert –1, während das stärkste Scharfschützengewehr den Wert 1 hat. Die Eingabe der Munitionsmenge gibt den prozentualen Anteil der Munition im Magazin zu seiner Gesamtgröße an.

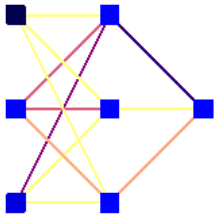
Die momentane Situation des Bots setzt sich aus der Anzahl der Gegner, der Anzahl der Mitkämpfer auf der eigenen Seite in einem Kampf, der allgemeinen Einstellung, d.h. ob der Bot immer eher defensiv oder aggressiv spielt und der momentanen Aggressivität bzw. Defensivität, die durch verschiedene Ereignisse verändert werden kann, zusammen.

[4] In HL entsprechen 75 Einheiten der Höhe eines Spielers

Das Kollisionsnetz wird nur aktiv wenn einer der ‚Fühler' ein Signal liefert. Da diese nur 75 Einheiten lang sind, ist dieses Netz meist inaktiv solange sich der Bot im freien Gelände bewegt. In engen Gängen sind sie dagegen fast immer aktiv.

Tabelle 5: Ein- und Ausgaben des Kollisionsnetzes. α ist z. Zt. 35° (siehe auch BotNNDefs.h, CollNNTrain.pta)

Eingabe	**Ausgabe**
Entfernung-α° links Entfernung Mitte Entfernung α° rechts	Bewegung : - links (<-.5) - rechts (>.5)



… bis zum nächsten Objekt

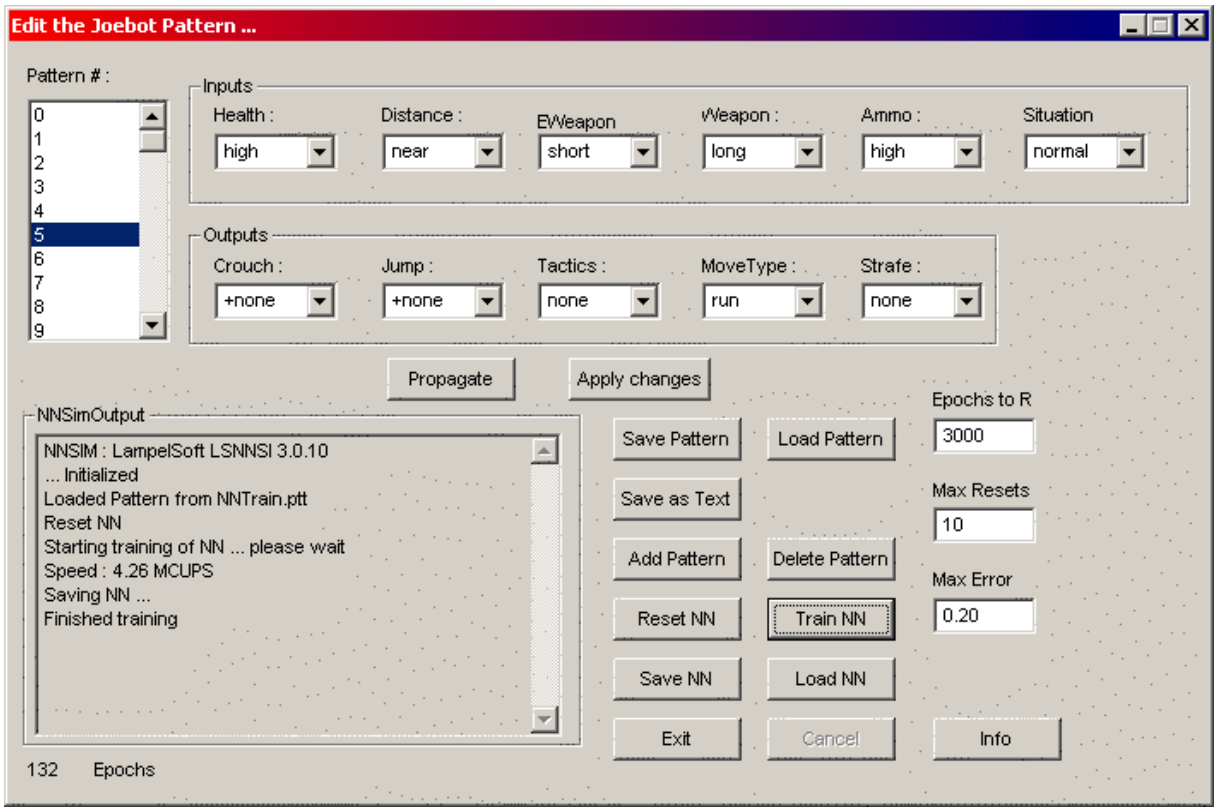


Abbildung 17 : GUI zur Erstellung der Trainingsdaten. Es wurde erstellt um die Trainingsdaten übersichtlicher verändern können. Sonst musste dies in NNTrain.cpp in reinem ASCII-Text erreicht werden.

4.2 Gewinnung der Trainingsdaten

Die Auswahl der Trainingsdaten muss sehr sorgfältig erfolgen, um möglichst alle Situationen abzudecken, bzw. genügend verschiedene Muster dem Neuronalen Netz bereitzustellen, um eine möglichst sinnvolle Interpolation der Ausgabedaten zu erreichen.
Ich bin so vorgegangen, dass ich zuerst versucht habe mir möglichst viele, aber dennoch verschiedene Situationen auszudenken. Diese Muster wurden dann dem neuronalen Netz des Bots beigebracht und der Bot wurde dann getestet um evtl. Schwachstellen zu entdecken. Da dieses Verfahren jedoch sehr zeitaufwändig ist, da nach kurzer Zeit die Liste der Trainingsmuster immer länger wird, suchte ich mir eine andere Methode: Ich sammelte zuerst alle Eingaben des Kampf-NNs während eines Spiels. Diese Muster gab ich einer selbstorganisierenden Karte zum Kategorisieren. Nach der fertigen Kategorisierung projizierte ich die Muster, die ich zum Trainieren des neuronalen Netzes benutzt habe auf die Karte. Nun kann man die Verteilung der Gewichte und die Trainingsdaten vergleichen und evtl. Muster finden, die das Netz wenig kennt, also der Abstand zwischen Trainingsdaten und Eingabe aus dem Spiel möglichst groß ist.

Die nachfolgenden 5 Bilder stellen einzelnen Komponenten der Gewichtsvektoren der Neuronen der SOM dar. (Diese Daten wurden zu einem Zeitpunkt gewonnen, zu der das Netzt des Bots noch 5 Eingabeneuronen hatte) Jeder Bildpunkt ist an der Stelle angeordnet, an der auch das dazugehörige Neuron in der SOM angesiedelt ist. Das Diffenzbild ist dadurch entstanden, dass ich für jeden Gewichtsvektor der Karte das am nächsten gelegene Trainingsmuster gesucht habe und den Abstand berechnet habe. (Betrag des Diffenzvektors)

Die weißen Werte in dieser Tabelle sind jeweils die höchsten. Weiß entspricht somit einem Wert nahe 1, schwarz einem Wert nahe –1. Beim Differenzbild gilt: Weiß entspricht einem Abstand der Gewichtsvektoren von 1,97; schwarz 0; Die schwarzen Punkte repräsentieren die manuell gefundenen Trainingsmuster für das Kampf-Netz. Diese Muster wurden dem SOM als Eingabe präsentiert, woraufhin es das Gewinnerneuron ermitteln musste. An der Stelle des Gewinnerneurons wurde nun der schwarze Kreis gesetzt.
Die nachfolgenden Bilder stammen von einer SOM, die 90*100 Neuronen groß war, mit 12.000 Trainingsmustern aus einem Counterstrike-Spiel trainiert wurde. Weiterhin benutzte ich :

$n(t) = 0.1$
$d(t) = 0.9 * \text{X-Größe} * 0.99^t$
$f_{NK}=f_{gauss1}$

Das Training wurde bei d < 1 abgebrochen (~ 438 Epochen) und benötigte auf einem PIII 500Mhz ungefähr 31h. (Erst später habe ich mit einer linear fallenden Distanzfunktion bessere Erfahrungen gemacht und habe diese auch erst dann verwendet.)

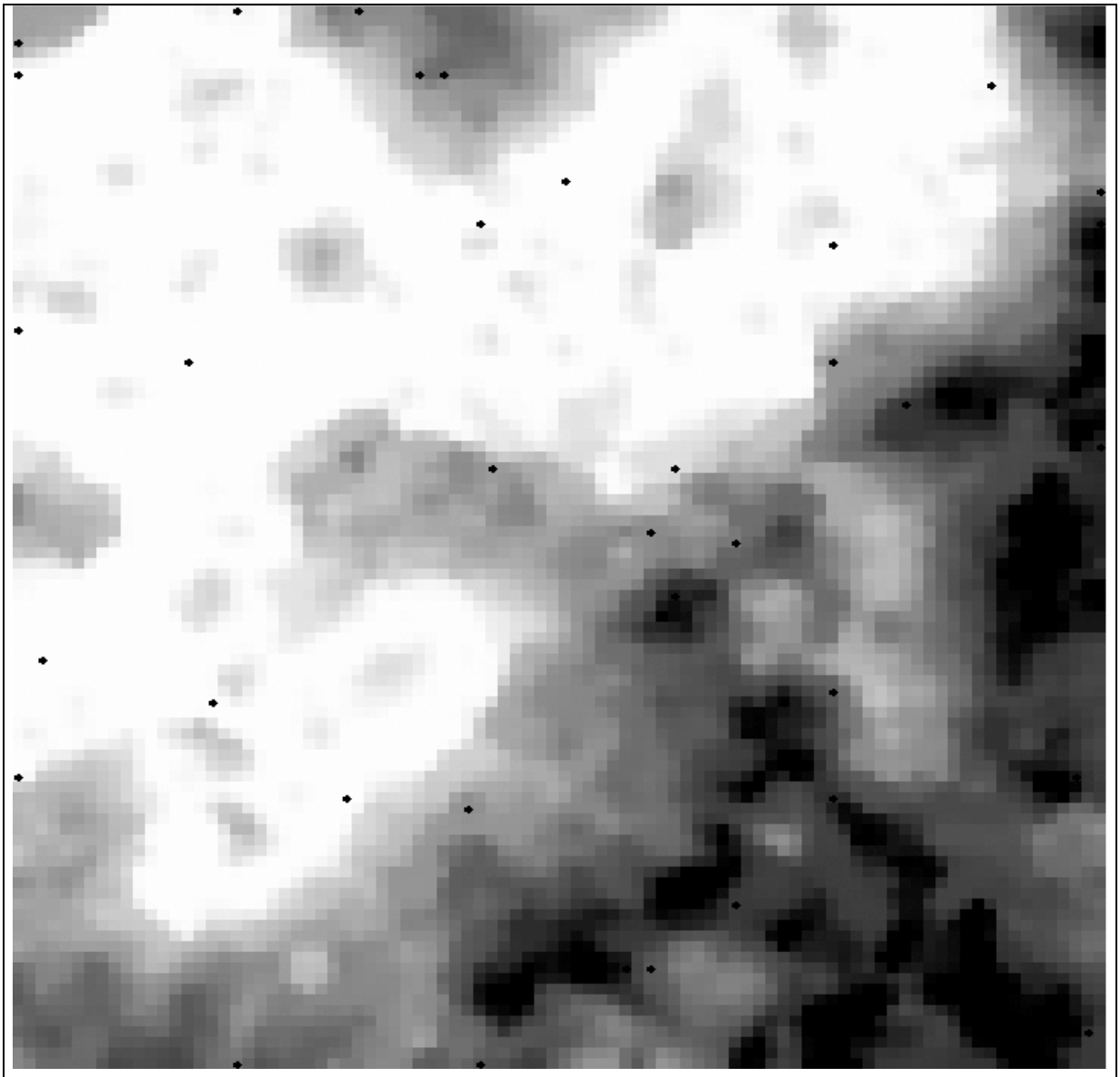


Abbildung 18 : (IHealth) Gesundheits- Komponente der Gewichtsvektoren der Neuronen

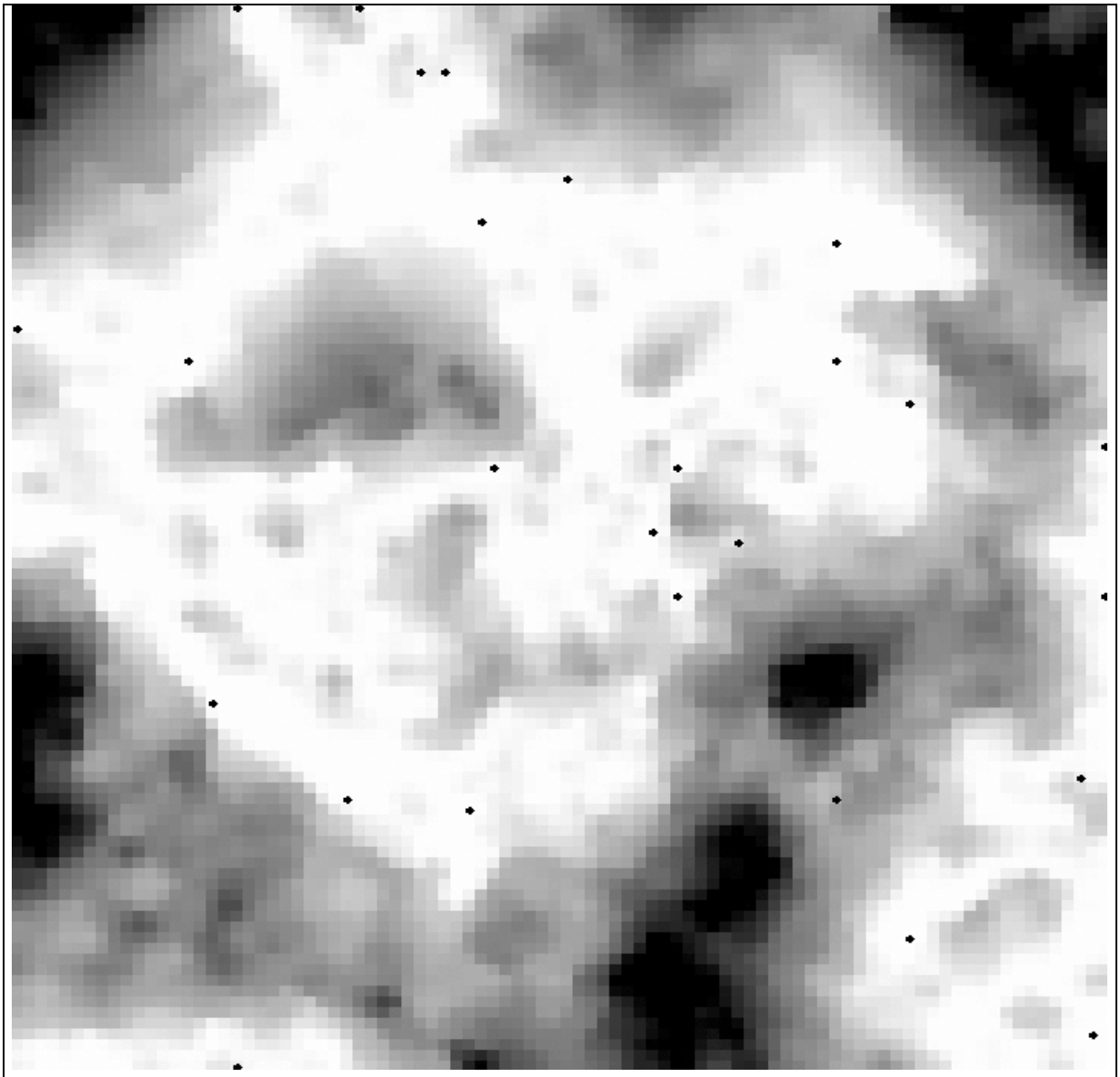


Abbildung 19 : (IDistance) 'Distanz zum Gegner'-Komponente

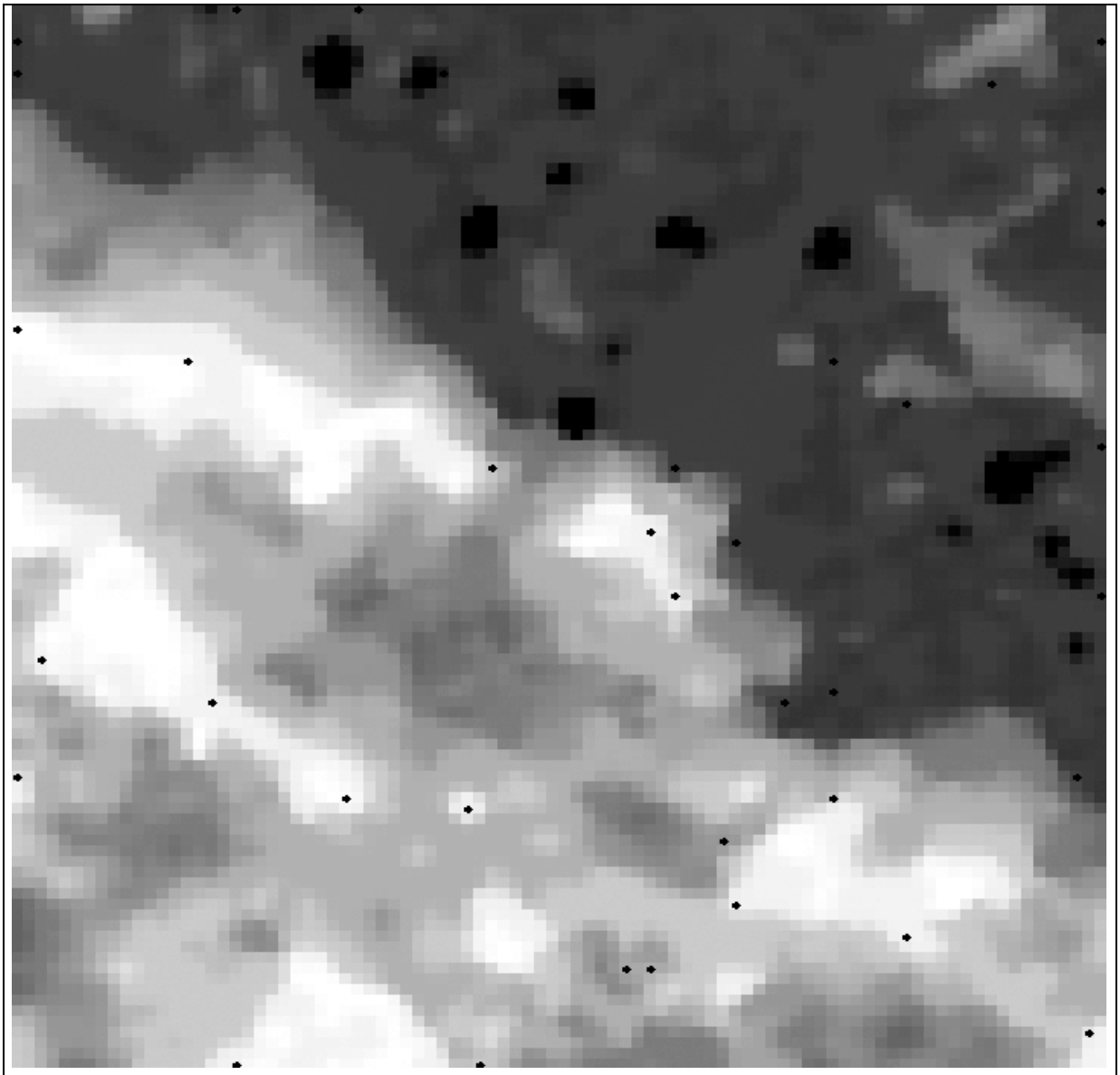


Abbildung 20 : (IEWeapon) 'Waffe des Gegners'-Komponente

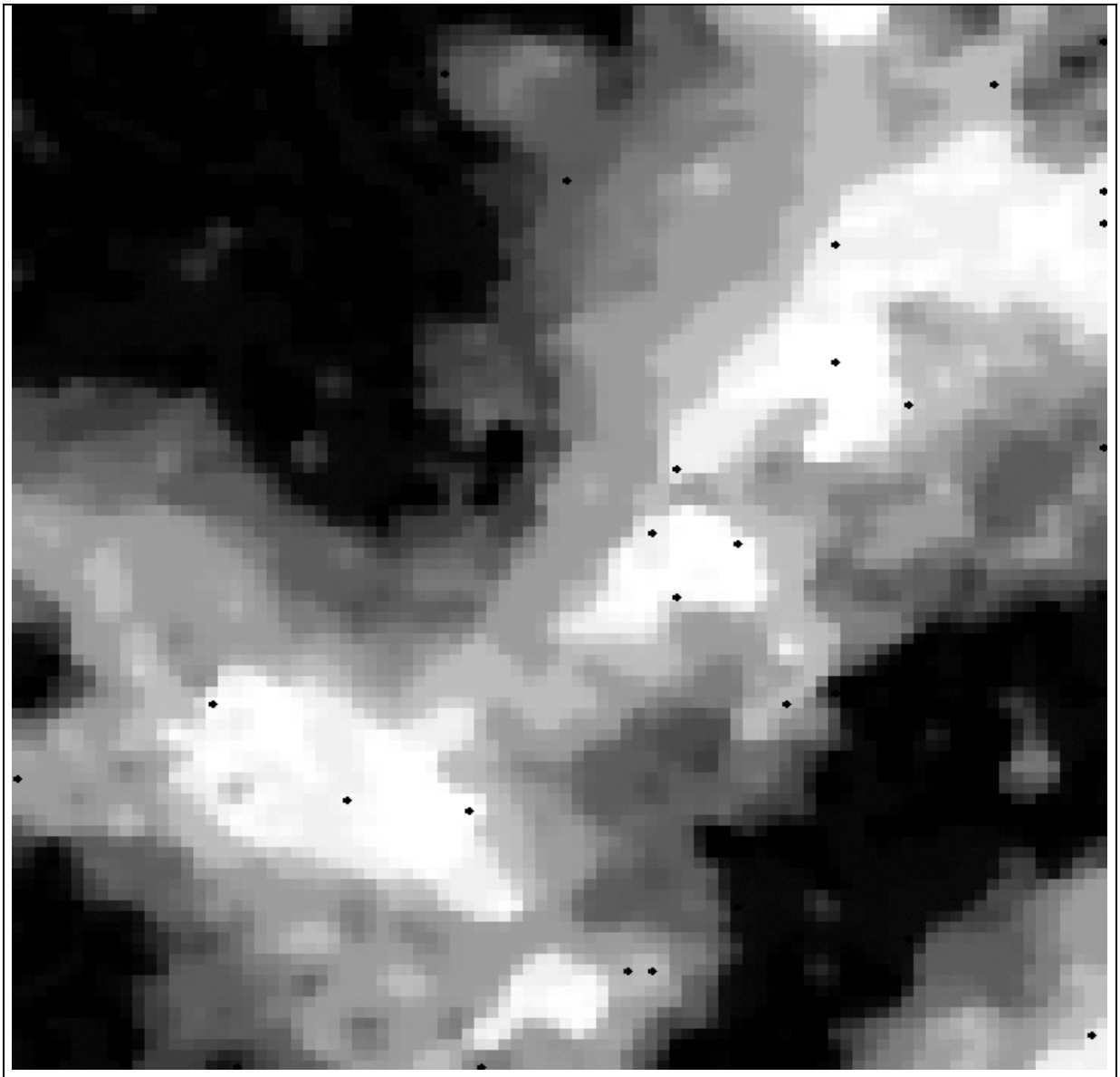


Abbildung 21 : (IWeapon) 'Waffe des Bots'-Komponente

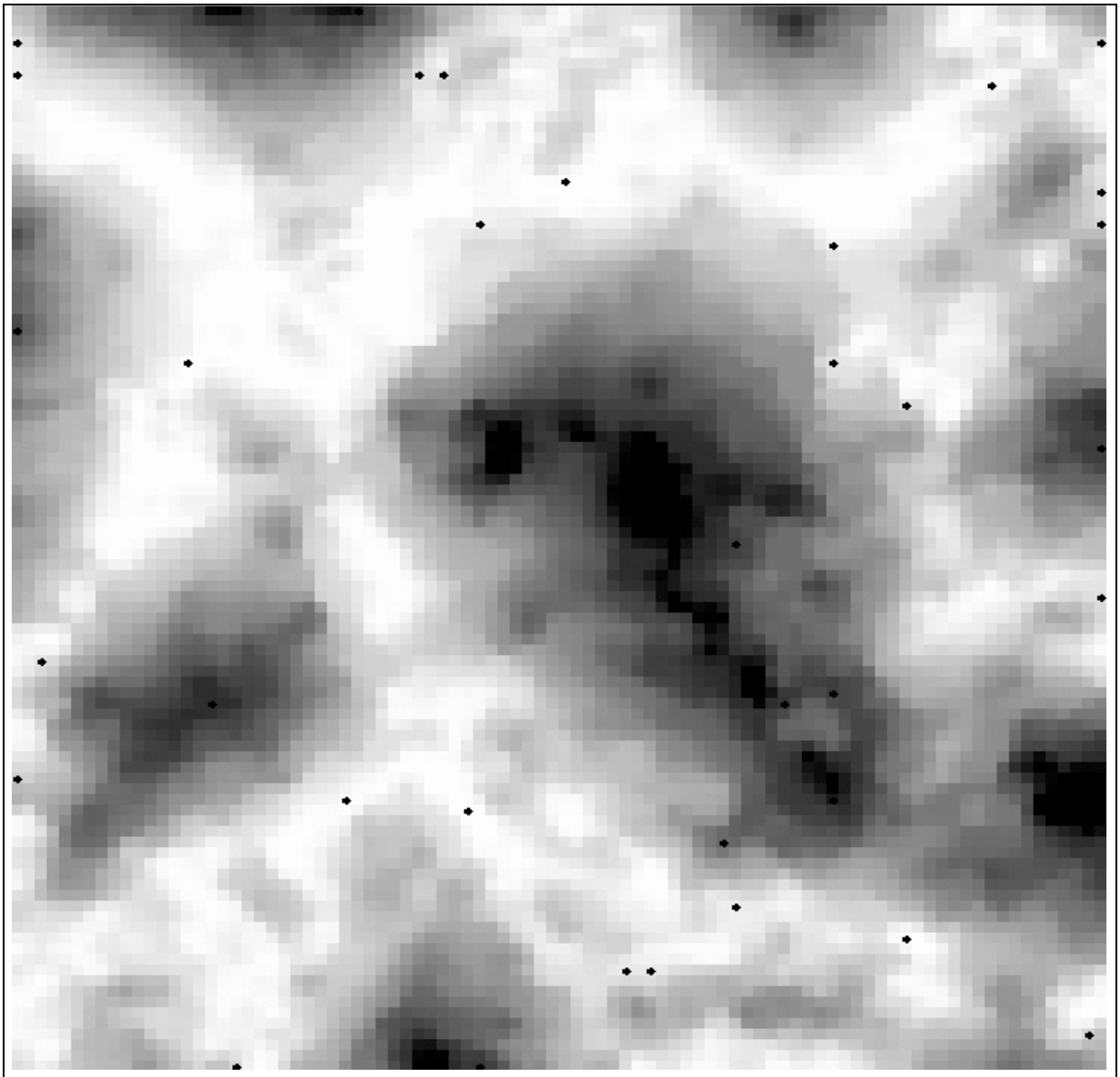


Abbildung 22 : (IAmmo) 'aktuelle Munition des Bots'-Komponente

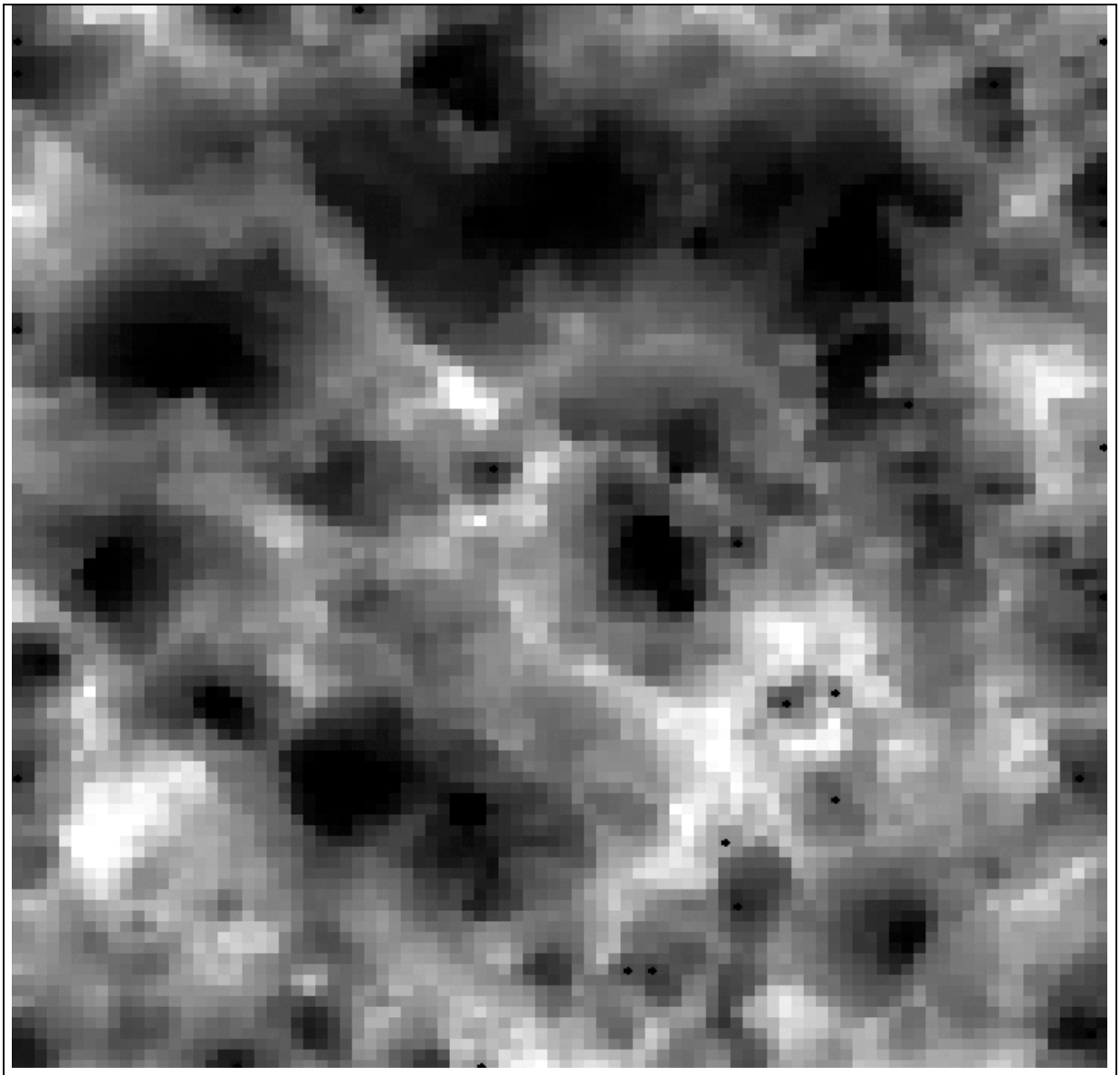


Abbildung 23 : Bild der Differenzen der Gewichtsvektoren zu dem jeweils nächsten Trainingsmuster

In Abbildung 11-15 kann man erkennen, dass der Lernalgorithmus der SOMs ähnliche Trainingsmuster an ähnlichen Stellen abbildet : Die schwarzen bzw. weiße Gebiete hängen zusammen, es gibt keine Bildung von mehreren stark weißen ‚Inseln' in einem dunklen Bereich.

Bei dem Differenzbild kann man einige weiße Bereiche erkennen. Diese Bereiche hoher Diskrepanz zwischen Trainingsdaten und aufgenommenen Daten aus einem Spiel müssen näher untersucht werden. Im Visualisierungsprogramm kann man sich die Gewichtsvektoren der einzelnen Neuronen anzeigen lassen und so die genauen Werte erhalten. Es ist aber nicht zwingend nötig diese Muster neu hinzuzufügen, da es einige Muster gibt, für die das Netz gut generalisiert, also selber eine gut passende Ausgabe findet.

4.3 Technische Einzelheiten bei der Implementierung des Bots

4.3.1 Einbindung in Halflife

Der kompilierte Code des Bots wird in einer DLL[5] gespeichert. Halflife ist so aufgebaut, dass einfach Erweiterungen (sog. Mods) hinzugefügt werden können. Dies wird dadurch erreicht, dass die HL.exe (die ausführbare Datei des Spiels) , die die Funktionen bspw. der 3D-Engine[6] und der Netzwerkfunktionen bereitstellt, eine sogenannte ‚client'-DLL aufruft, die den spielspezifischen Code enthält. Die Dokumentation der HL-API ist bei [Valve] in Form des HL-SDKs, des kompletten Quellcodes des Spiels Halflife, zu erhalten.

Der Botcode wird auf besondere Art und Weise eingebunden: Er selbst stellt keine vollständige Erweiterung dar, wohl aber eine Erweiterung für eine Erweiterung von HL, muss also wieder mit der eigentlichen DLL des Spiels verbunden sein. Alle Funktionen, die nicht den Bot betreffen werden direkt ‚durchgelinkt', d.h. es werden die Funktionsaufrufe der entsprechenden Funktionen direkt an die, im Falle von CS, mp.dll weitergegeben. Einige Funktionen werden erweitert, z.B. um neue Konsolenbefehle[7] hinzuzufügen.

Die beiden DLLs kommunizieren weiterhin über eine Nachrichtensystem, um z.B. Informationen an die Engine zu geben, die sie auf dem HUD[8] darstellen soll. Dieses Nachrichtensystem wird auch verwendet, wenn mehrere Spieler über eine Netzwerkverbindung zusammen spielen. Diese Meldungen werden abgefangen wenn sie an die Bots gerichtet sind, da die Bots kein gerendertes Bild bekommen, sondern sich Informationen über bestimmte Funktionen oder über diese Meldungen beschaffen müssen. Fast alle aktuellen Bots, wie auch meiner, basieren auf den Funktionen, die [botman] zur Kommunikation zwischen HL.exe, der ‚client'-DLL und der Bot-Dll geschrieben hat. Der HPB-Templatecode enthält nur die Funktionen die hierfür nötig sind, er enthält keine weitergehenden Funktionen.

---

[5] DLL : Dynamic Link Library : eine Datei, die ausführbaren Code bereitstellt. DLLs werden dann verwendet wenn mehrere Programme bestimmte Funktionen benötigen. In diesen Fällen kann dann die DLL geladen und deren Funktionen genutzt werden. Weiterhin ist es möglich nur die Schnittstelle zu veröffentlichen und den eigentlichen Code in kompilierter Form in der DLL weiterzugeben, damit der Quelltext nicht zwangsläufig mit veröffentlicht werden muss.

[6] Engine : Teil des Spiels, dass die Funktionen zur graphischen Darstellung bereitstellt

[7] Konsolenbefehle : HL besitzt eine Art Kommandozeile, die Konsole genannt wird. Sie ermöglicht eine Veränderung von spielinternen Variablen wie bspw. einige Optimierungsoptionen oder auch den Aufruf der zur Administration eines Servers benötigten Befehle.

[8] HUD : head up display – Bei vielen Spielen werden wichtige Informationen wie bspw. der aktuelle Gesundheitsstatus über das gerenderte Bild gelegt.

Bei jedem Frame[9] wird eine bestimmte Funktion (`void StartFrame( void )`) aufgerufen, die Verwaltungsaufgaben und den Aufruf der Hauptfunktion des Bots (`void CbotBase::Think( void )`) übernimmt. Somit werden bei jedem Frame alle Parameter überprüft und evtl. Variablen wie Blickrichtung, Geschwindigkeit usw. verändert. Da die Framerate normalerweise bei 40-100 fps (frames per second) liegt, war eine möglichst gute Optimierung der Funktion nötig. Aus diesem Grund war auch eine möglichst hohe Leistungsfähigkeit des NN-Simulators nützlich. Einschränkend muss allerdings gesagt werden, dass die Situationen, in denen die NNs eingesetzt werden, höchstens 10% aller Frames ausmachen. Außerdem habe ich während der Botentwicklung den neuronale Netze-Simulator nicht weiter optimiert, dies hatte ich schon aus anderen Motivationsgründen vorher getan.

Die grobe Navigation wird von einem Waypointsystem übernommen. Für dieses System müssen einmal einige Wege in einer Karte eingetragen werden. Um einen Weg zu finden muss eine bestimmte Eigenschaft eines Wegpunktes angegeben werden. Dies kann eine Positionsangabe, oder auch eine andere Eigenschaft, wie der Ort der Geiseln, sein. Daraufhin werden nun Möglichkeiten zu diesem Punkt zu kommen berechnet und aufgrund anzugebender Entscheidungskriterien ausgewählt. Früher kam zur Berechnung der Wege Backtracking zum Einsatz, ein rekursives Verfahren, dass alle Möglichkeiten ausprobiert. Im Moment verwende ich als Basis den Floyd/Warshall Algorithmus [fewalgo] , der durch ein dem alten System ähnlichen Verfahren erweitert wird. Die Umstellung war nötig, da es nicht möglich war, den Backtracking-Algorithmus auf mehr als 100-200 Waypoints anzuwenden, da dieser Vorgang teilweise recht lange dauerte. (>0.03 s)

Solange dem Bot keine Gegner oder interessante Gegenstände begegnen orientiert er sich nur an diesem System, was natürlich eine gewisse Inflexibilität mit sich bringt, die allerdings mit einer größeren Anzahl von Wegpunkten annähernd ausgeglichen werden kann.

Kollision werden durch die oben beschriebene Kombination von einem NN und 3 ‚Fühlern' realisiert. Die HL-Engine ermöglicht das Verfolgen von Geraden durch bestimmte Ortsvektoren, wodurch man eine Unterbrechung dieser Gerade etwa durch eine Wand feststellen kann.

Die Daten die dem Bot während eines Kampfes zur Verfügung stehen, erhält er zum Teil aus den Meldungen an die Engine (Gesundheit, Munition, Waffentyp usw.) bzw. durch den Aufruf von Funktionen der Engine. Es kann mit Hilfe dieser Funktionen direkt ein Zeiger auf eine `edict_t` Struktur eines Mitspielers erhalten werden, die auch die Position der

[9] Ein Frame ist ein gerendertes Bild

Gegner/Mitspieler enthält. Damit das Spiel meines Bots jedoch allerdings menschlich wirkt, simuliert er auch einen Sichtbereich und eine Reaktionszeit, die allerdings manuell hinzuzufügen waren. Wenn das Ziel ein möglichst guter Bot wäre, könnten alle Bots sofort die Positionen aller Gegner feststellen und sich gemäß diesen Informationen verhalten.
In einigen wenigen Situationen machen die Bots dies sogar, da Abschätzen von weiteren Bewegungen schwer zu programmieren ist und somit die Spielstärke der Bots noch ein wenig steigern kann. Diese wird durch ein System kontrolliert, mit dem man die Fähigkeiten der Bots einzeln konfigurieren kann (*CSkill.h*).

Anhang A: Literatur

[aiindex] http://www.cs.berkeley.edu/~russell/ai.html-Ein Index für einige AI Seiten

[aiMIT] http://ai.mit.edu - Die Ansätze von R. Brooks und seinen Mitarbeitern

[BFMalaka96] H. Braun, J. Feulner, R. Malaka; Praktikum Neuronale Netze; 1997, Springer

[botman] http://planethalflife.com/botman

[botf1] http://dynamic.gamespy.com/~botman/yabb/YaBB.pl - Bot Forum

[fewalgo] http://wannabe.guru.org/alg/node1.html - Einige nützliche Algorithmen

[gameai] http://www.gameai.com – Allg. Informationen zu AI in Spielen

[gamedev] http://www.gamedev.net – Eine sehr informative Seite

[joebot] http://www.botepidemic.com/joebot (Aktuelle Version des Bots)

[Kohonen] Teuvo Kohonen, Self-Organizing Maps, 3. Auflage, Springer 2001, Berlin, Heidelberg

[Mataric] http://www-robotics.usc.edu/~agents/

[nnbot] http://planethalflife.com/nnbot - ähnlicher Ansatz, leider nicht mehr weiterentwickelt

[PfScheier99] Rolf Pfeifer, Christian Scheier; Understanding Intelligence; Cambridge, 1999, The MIT Press
http://www.ifi.unizh.ch/groups/ailab/people/lambri/mitbook/start.html

[Valve] http://www.valvesoftware.com Der Hersteller von HL

[ZellSNN97] A. Zell; Simulation neuronaler Netze; Oldenbourg Verlag, 1997, München

Anhang B : Quelltexte

Der Umfang der gesamten Botquelltexte beträgt ungefähr 1,1 MB, weshalb hier nur einige Ausschnitte aus den Simulatoren und dem eigentlichen Botquelltext abgedruckt sind.

B.1 Quelltext des (Feedforward-) NN-Simulators

Hier sind jetzt nur die wichtigsten Teile des Simulators abgedruckt. Der Teil des Simulators für BP + Momentum und TDNNs wurden nicht abgedruckt, da sie nur geringe Abweichungen der BProp Klasse darstellen.

NeuralNet.h

```
#ifndef __NEURALNET_H
#define __NEURALNET_H
#include <assert.h>

// Includen der verschiedenen Arten von NNs
#include "NeuralNetDef.h"
#include "Impl/ErrorClasses.h"
#include "Impl/Liste.h"
#include "Impl/Queue.h"
#include "Impl/NeuralNetBProp.h"
#include "Impl/NeuralNetBPropM.h"
#include "Impl/NeuralNetTDNNBPropM.h"
#include "Impl/NeuralNetBPropBatch.h"
#include "Impl/Pattern.h"
#include "Impl/NBasic.h"

#endif//__NEURALNET_H
```

LinkBaseDef.h

```
#ifndef __LINKBASEDEF_H
#define __LINKBASEDEF_H

// written by Johannes Lampel / LampelSoft '99
// This class contains pointer to neurons, on which the nn is based. LptrNt and LptrNf in
class
// NeuronBase use them.
//
// Nptr    : Pointer to connected Neuron
// Wptr    : pointer to related Weight ( can be found in NeuralNet... :: LWeights )
// lWeight : index of related weight in NeuralNet ...::LWeights
//
// comment : The pointer Wptr has only be added to make things faster. Otherwise
Liste::operator[]
// would have to be called. The Weight are in a list because the 'from' and 'to' part of a
// connection are stored and the value of this weight has to be the same. (This gave problems
// with dtors)
class WeightBase;
class NeuronBase;

class DllExport LinkBase
{
        friend class NeuronBase;
public:
        LinkBase();
        LinkBase(LinkBase &);
        virtual ~LinkBase();
        const LinkBase & operator=(const LinkBase&);
        // it's not needed, but Liste required it
        int Load(FILE *);
        int Save(FILE *);
//protected:
        NeuronBase     *Nptr;
        WeightBase     *Wptr;
        long           lWeight;
};

#endif//__LINKBASEDEF_H
```

LinkBase.h

```
#ifndef __LINKBASE_H
#define __LINKBASE_H

#include "../Defs/LinkBaseDef.h"

LinkBase :: LinkBase()
{
	Nptr = 0;
	Wptr = 0;
	lWeight = 0;
}

LinkBase :: LinkBase(LinkBase &LParam)
{
	Nptr = LParam.Nptr;
	Wptr = LParam.Wptr;
	lWeight = LParam.lWeight;
}

LinkBase :: ~LinkBase(){
	Nptr = 0;
	Wptr = 0;
	lWeight = 0;
}

const LinkBase & LinkBase :: operator =(const LinkBase &LParam)
{
	Nptr = LParam.Nptr;
	Wptr = LParam.Wptr;
	lWeight = LParam.lWeight;
	return ((const LinkBase &)(*this));
}

int LinkBase :: Save(FILE *fhd){
	printf("int LinkBase :: Save(FILE *fhd) isnt impl");
	return 1;
}

int LinkBase :: Load(FILE *fhd){
	printf("int LinkBase :: load(FILE *fhd) isnt impl");
	return 1;
}

#endif//__LINKBASE_H
```

ListeDef.h

```
#ifndef __LISTEDEF_H
#define __LISTEDEF_H

// written by Johannes Lampel / LampelSoft '99
//
// LampelSoft High - Speed - List by Johannes Lampel
// (c) Johannes Lampel / LampelSoft
//
// written : 11. September 1999
//
// PLEASE UPDATE OPERATOR LISTE <TYPE> =
//
// This library was written for high-performance applications and wastes more memory the
higher
// _FANZ is initialized. If ya've enought memory set _FANZ to the largest list ya exspect.
// BUT PLEASE USE POWERS OF 2 !! This increases the performance a lot,because many compilers
are
// optimizing divisions with this kinds of numbers to a rotate - assembler - instruction (SAR
etc.)
// Without optimizing the compiler uses a IDIV - instruction, which uses up to 22 cycles, SAR
// uses only 3 cycles (on a Pentium).
//

#include "ListeelemDef.h"
#include <stdio.h>

template <class TYPE> class Liste
{
       friend class NeuralNetBProp;
       friend class NeuralNetBPropM;
       friend class NeuralNetTDNNBPropM;
public:
                     Liste                 ();
                     //Liste               (const Liste <TYPE> &);
                     ~Liste                ();
       inline int    DelElem               (long);                       // Element löschen
       inline void   DelAllData            (void);
       //Alle Elemente einer Liste lö.
       inline TYPE   *AddElem              (const TYPE&);                //Element hinzufügen
       inline int    ChangeElem            (const long,const TYPE&);     //Element verändern
       inline int    InsertElem            (long,const TYPE&);   //Element einfügen
       inline long   GetAnz (void);
       //Anzahl der Elemente
       inline TYPE   *GetElem      (long)const;                                //Gibt
Element zurück
                     int           LSave(FILE *);
              //Speichert Liste
                     int           LLoad(FILE *);
              //Lädt Liste

       inline const Liste <TYPE>    &operator =    (const Liste <TYPE> &);      //Liste
kopieren
       inline const Liste <TYPE>    &operator +=   (const TYPE &);
       //Element anfügen
       inline const Liste <TYPE>    &operator +=   (Liste <TYPE> &);            //Liste
anfügen
       inline TYPE                  *operator []   (long);                      //Gibt
Element zurück

protected:
       long lanz;
                     //Anzahl der Elemente(Listeelem
       long lganz;
                     //Anzahl der einzelnen Elemente TYPE
       Listeelem <TYPE>      *fElem,
       //Zeiger auf erstes
                                       *lElem;
                     //Zeiger auf zweites Element
};

class NeuronBase;
class NeuronBProp;
class NeuronBPropM;
class WeightBProp;
class WeightBPropM;
class LinkBase;
class PatternElem;
```

```
class VarNeuralNetBProp;
class VarNeuralNetBPropM;
class VarInteger;
class VarFloat;
class VarPattern;

template class DllExport Liste <NeuronBase>;
template class DllExport Liste <NeuronBProp>;
template class DllExport Liste <NeuronBPropM>;
template class DllExport Liste <WeightBProp>;
template class DllExport Liste <WeightBPropM>;
template class DllExport Liste <LinkBase>;
template class DllExport Liste <PatternElem>;
template class DllExport Liste <VarNeuralNetBProp>;
template class DllExport Liste <VarNeuralNetBPropM>;
template class DllExport Liste <VarInteger>;
template class DllExport Liste <VarFloat>;
template class DllExport Liste <VarPattern>;

#endif//__LISTEDEF_H
```

Liste.h

```
#ifndef __Liste_H
#define __Liste_H

#include <stdio.h>

#include "../Defs/ListeDef.h"

const long _FANZ=256;
#include "Listeelem.h"

template <class TYPE>
Liste <TYPE> :: Liste ()
{
	lanz = lganz = 0;
	fElem = lElem = 0;
}

template <class TYPE>
Liste <TYPE> :: ~Liste ()                                    //Löscht alle Elemente in der Liste
(wie DelAllData)
{
	DelAllData();
}

template <class TYPE>                                                    //Fügt Element an
inline TYPE * Liste <TYPE> :: AddElem (const TYPE &tparam)
{
	if(lanz)                                             //Prüfen,ob überhaupt schon
instanzen von Listeelem bestehen
	{
		Listeelem <TYPE> *LEakt;
		if(lanz==1)                                  // wenn ja, testen ob nur eine
besteht
		{
			if(fElem -> laktf < _FANZ)           //Wenn das hinzuzufügende
Element noch in die alte ListeelemInstanz passt
			{
				fElem -> Data[fElem->laktf] = tparam;
				fElem -> laktf ++;
				lganz++;
				return (&(fElem->Data[fElem->laktf-1]));
			}
			else                                                     //Sonst neue
Instanz anlegen
			{
				LEakt = new Listeelem <TYPE>;
				lanz++;                                          //Zähler
inkrementieren
				LEakt -> prevelem = fElem;           //Zeiger setzen
				fElem -> nextelem = LEakt;
				lElem = LEakt;
				LEakt -> Data[0] = tparam;
				LEakt -> laktf ++;
				lganz++;
				return (&(LEakt->Data[0]));
	//Zähler inkrementieren
			}
		}
		else                                                     // wenn mehrere
Elemente (Listeelem) existieren
		{
			LEakt = lElem;

			if(LEakt -> laktf < _FANZ)           // Wenn dort noch platz ist
			{
				LEakt -> Data[LEakt->laktf] = tparam;
				LEakt -> laktf ++;
				lganz++;
				return (&(LEakt->Data[LEakt->laktf-1]));
			}
			else                                                     // sonst neue
Listeelem Instanz anfügen
			{
				Listeelem <TYPE> *LEaktn = new Listeelem <TYPE> ;
				lanz++;
				LEaktn -> prevelem = LEakt;
```

```
                                LEakt -> nextelem = LEaktn;
                                lElem = LEaktn;
                                LEaktn -> Data[0] = tparam;
                                LEaktn -> laktf ++;
                                lganz++;
                                return (&(LEaktn->Data[0]));
                        }
                }
        }
        else    //wenn keine Listeelem Instanzen existieren
        {
                fElem = lElem = new Listeelem <TYPE> ;
                lanz++;
                fElem -> Data[0] = tparam;
                fElem ->laktf ++;
                lganz++;
                return (&(fElem->Data[0]));
        }
}

template <class TYPE>                                                   // Löscht alle
Elemente
inline void Liste<TYPE> :: DelAllData(void)
{
        register Listeelem<TYPE>     *aktelem,
                                                        *aprevelem;

        if(!fElem)
                return;
        aktelem=fElem;
        while(aktelem->nextelem)
                aktelem=aktelem->nextelem;
        while(1)
        {
                aprevelem=aktelem->prevelem;
                delete aktelem;
                if(!aprevelem)
                        break;
                aktelem=aprevelem;
        }
        fElem = 0;
        lElem = 0;
        lanz = 0;
        lganz = 0;
        return;
}

template <class TYPE>
inline int Liste <TYPE> :: ChangeElem(const long lchaanz,const TYPE &tparam)
{
        *GetElem(lchaanz)=tparam;
        return(lchaanz);
}

template <class TYPE>
inline int Liste <TYPE> :: DelElem(long lchaanz)
{
        lchaanz--;
        Listeelem<TYPE>       *LAktLE,
                                *LOldLE,
                                *LStaLE;

        long lschl;
        long lLEpos;                                                    // (wahrscheinlich)
Das zu löschende Element
        long lfnum = lchaanz / _FANZ;
        long lenum = lchaanz - (lfnum * _FANZ);

        LAktLE = fElem;

        if(lchaanz<=lganz)
        {
                if(lanz==1)
                {
                        lLEpos = lenum;
                        LAktLE = fElem;
                }
                else
```

```
		{
			long lschl = 0;

			while(lschl != lfnum)
			{
				lschl++;
				LAktLE = LAktLE -> nextelem;
			}

			assert (LAktLE!=0);
			if(LAktLE->laktf >= lenum)
			{
				lLEpos = lenum;
			}
			else
				return 0;
		}
	}
	else
		return 0;

	LStaLE = LAktLE;
	//cerr<<"DELELEM  lchaanz:"<<lchaanz<<"; lganz:"<<lganz<<endl;
	for(lschl=lchaanz;lschl<lganz-1;lschl++)
	{
		if(lLEpos<=_FANZ)
		{
			LAktLE -> Data [lLEpos] = LAktLE -> Data [lLEpos+1];
			lLEpos ++;
		}
		else
		{
			LOldLE = LAktLE;
			LAktLE = LAktLE -> nextelem;
			LOldLE -> Data [_FANZ-1] = LAktLE -> Data[0];
			lLEpos = 1;
		}
	}
	if(LAktLE ->laktf == 1)
	{
		lElem = LAktLE -> prevelem;
		lanz--;
		delete LAktLE;
	}
	else
	{
		(LStaLE -> laktf) --;
	}
	lganz --;
	return 1;
}

template <class TYPE>
inline long Liste <TYPE> :: GetAnz(void)
{
	return(lganz);
}

template <class TYPE>
inline TYPE *Liste <TYPE> :: GetElem(long lchaanz) const
{
	lchaanz--;
	if(lchaanz<=lganz)
	{
		long lfnum = lchaanz / _FANZ;
		long lenum = lchaanz - (lfnum * _FANZ);

		if(lanz==1)
		{
			return (&(fElem->Data[lenum]));
		}
		else
		{
			Listeelem <TYPE> *LEakt = fElem;
			long lschl = 0;

			while(lschl != lfnum)
			{
```

```
                        lschl++;
                        LEakt = LEakt -> nextelem;
                };
                if(LEakt->laktf >= lenum)
                        return (&(LEakt->Data[lenum]));
                else
                        return 0;
                /*if(lfnum < lhanz)
                {
                        Listeelem <TYPE> *LEakt = fElem;
                        long lschl = 0;

                        while(lschl != lfnum)
                        {
                                lschl++;
                                LEakt = LEakt -> nextelem;
                        };
                        if(LEakt->laktf >= lenum)
                                return (&(LEakt->Data[lenum]));
                        else
                                return 0;
                }
                else
                {
                        Listeelem <TYPE> *LEakt = lElem;
                        long lschl = 0;

                        while(lschl != lfnum)
                        {
                                lschl--;
                                LEakt = LEakt -> prevelem;
                        };
                        if(LEakt->laktf >= lenum)
                                return (&(LEakt->Data[lenum]));
                        else
                                return 0;
                }*/
        }
    }
    else
        return 0;
}

template <class TYPE>
inline int Liste <TYPE> :: InsertElem(long lchaanz,const TYPE &tparam)
{
    lchaanz--;
    Listeelem<TYPE>     *LAktLE,
                        *LOldLE,
                        *LNewLE,
                        *LAktLEF;

    long lschl;
    long lLEpos,
         lLEPosF;

    LAktLE = fElem;

    if(lchaanz<=lganz)
    {
        long lfnum = lchaanz / _FANZ;
        long lenum = lchaanz - (lfnum * _FANZ);

        if(lanz==1)
        {
            lLEpos = lenum;
            LAktLE = fElem;
        }
        else
        {
            LAktLE = fElem;
            long lschl = 0;

            while(lschl != lfnum)
            {
                lschl++;
                LAktLE = LAktLE -> nextelem;
            }
```

```
                        if(LAktLE->laktf >= lenum)
                        {
                                lLEpos = lenum;
                        }
                        else
                                return 0;
                }
        }
        else
                return 0;

        lLEPosF = lLEpos;
        LAktLEF = LAktLE;
        for(lschl=lchaanz;lschl<lganz;lschl++)
        {
                if(lLEpos+1<_FANZ)
                {
                        LAktLE -> Data [lLEpos] = LAktLE -> Data [lLEpos+1];
                        lLEpos ++;
                }
                else
                {
                        LOldLE = LAktLE;
                        LAktLE = LAktLE -> nextelem;
                        if(!LAktLE)
                        {
                                LNewLE = new Listeelem <TYPE>;
                                lanz++;
                                lElem -> nextelem = LNewLE;
                                LNewLE -> prevelem = lElem;
                                lElem = LNewLE;

                                LAktLE = LNewLE;
                        }

                        LAktLE -> Data [0] = LOldLE -> Data [_FANZ];
                        lLEpos = 1;
                }
        }

        LAktLEF -> Data [lLEPosF] = tparam;

        if(LAktLE ->laktf == _FANZ)
        {
                LNewLE = new Listeelem <TYPE>;
                lanz++;
                lElem -> nextelem = LNewLE;
                LNewLE -> prevelem = lElem;
                lElem = LNewLE;
        }
        else
        {
                (LAktLE -> laktf) ++;
        }
        lganz ++;
        return 1;
}

template <class TYPE>
inline const Liste <TYPE> & Liste <TYPE> :: operator = (const Liste <TYPE> &param)
{
        register long lllanz;
        TYPE    tmpType;

        DelAllData();

        for(lllanz=param.lganz;lllanz;lllanz--)
        {
                //memcpy((void*)(&tmpType),(void*)param.GetElem(lllanz),sizeof(TYPE));
                tmpType = *param.GetElem(lllanz);
                //AddElem(*tmpType.object);
                AddElem(tmpType);
        }
        return((const Liste <TYPE> &)(*this));
}

template <class TYPE>
```

```
inline const Liste <TYPE> & Liste <TYPE> :: operator+= (const TYPE &param)
{
	AddElem (param);
	return((const Liste <TYPE> &)(*this));
}

template <class TYPE>
inline const Liste <TYPE> & Liste <TYPE> :: operator+=(Liste <TYPE> &param)
{
	TYPE typeelem;
	register long lllanz;

	for(lllanz=param.lganz;lllanz;lllanz--)
	{
		typeelem= *param.GetElem(lllanz);
		AddElem(typeelem);
	}
	return((const Liste <TYPE> &)(*this));
}

template <class TYPE>
inline TYPE *Liste <TYPE> :: operator[](long anz)
{
	if(anz>lganz) return(0);

	return(GetElem(anz));
}

template <class TYPE>
int Liste <TYPE> :: LSave (FILE *fhd)
{
	long lschl;
	fwrite(&lganz,sizeof(long),1,fhd);

	for(lschl=1;lschl<=lganz;lschl++)
	{
		GetElem(lschl)->Save(fhd);
	}
	return 1;
}

template <class TYPE>
int Liste <TYPE> :: LLoad (FILE *fhd)
{
	long lschl;
	TYPE temp;

	DelAllData();

	fread(&lschl,sizeof(long),1,fhd);

	for(;lschl>0;lschl--)
	{
		temp.Load(fhd);
		AddElem(temp);
	}
	return 1;
}

#endif // __Liste_H
```

ListeElemDef.h

```
#ifndef __LISTEELEMDEF_H
#define __LISTEELEMDEF_H

//written by Johannes Lampel / LampelSoft '99
//
// LampelSoft High - Speed - List by Johannes Lampel
// (c) Johannes Lampel / LampelSoft
//
// written : 11. September 1999

template <class TYPE> class Liste;
template <class TYPE> class Listeelem;

template <class TYPE>
class DllExport Listeelem
{
	friend class NeuralNetBProp;
	friend class NeuralNetBPropM;
	friend class NeuralNetTDNNBPropM;
public:
	friend class Liste <TYPE> ;

	inline Listeelem();
	inline Listeelem(const Listeelem <TYPE> &);
	inline ~Listeelem();
	inline const Listeelem <TYPE> & operator=(const Listeelem <TYPE> &);

private:
	Listeelem <TYPE>	*nextelem,
					*prevelem;
	long				laktf;
	TYPE				*Data;
};

#endif//__LSITEELEMDEF_H
```

ListeElem.h

```
#ifndef __ListeELEM_H
#define __ListeELEM_H

#include <memory.h>

template <class TYPE> class Liste;
template <class TYPE> class Listeelem;

template <class TYPE>
inline Listeelem <TYPE> :: Listeelem()
{
	nextelem = prevelem = NULL;
	laktf = 0;
	Data = new(TYPE[_FANZ]);
}

template <class TYPE>
inline Listeelem <TYPE> :: ~Listeelem()
{
	if(Data)
		delete [] Data;
}

template <class TYPE>
inline Listeelem <TYPE> :: Listeelem(const Listeelem <TYPE> &param)
{
	nextelem = param.nextelem;
	prevelem = param.prevelem;
	laktf    = param.laktf;

	memcpy(Data,param.Data,sizeof(Listeelem <TYPE>)*laktf);
}

template <class TYPE>
inline const Listeelem <TYPE> &Listeelem <TYPE> :: operator=(const Listeelem <TYPE> &param)
{
	nextelem = param.nextelem;
	prevelem = param.prevelem;
	laktf    = param.laktf;

	memcpy(Data,param.Data,sizeof(Listeelem <TYPE>)*laktf);

	return((const Listeelem &)(*this));
}

#endif // __ListeELEM_H
```

NeuralNetBaseDef.h

```
#ifndef __NEURALNETBASEDEF_H
#define __NEURALNETBASEDEF_H

//Neuronale Netze Simulator written by Johannes Lampel / LampelSoft '99
//written : 18. Nov 1999
//
// base class for every NN, which is based on BProp or QProp
//
// NeuralNetBase    : Allocs mem for information about the current version and inits pointer
//                    ,learnrate and the pointer to the teaching input
// ~NeuralNetBase   : frees eventually alloced mem
// GetVersion       : returns pointer to information about version
// SetLRate         : Sets learnrate
//
// Data elements
// szVersion        : char * pointer to information about version
// dpTInput         : double * pointer to teaching input
// dLRate           : learnrate
// iType            : type of net -> definitions of constants in NeuralNet.h

#include "NeuronBaseDef.h"

#define __MAXNNVERSIONLENGTH 50                          // don't change this, cause then
the old files would be incompatible

class  DllExport NeuralNetBase
{
       friend class NeuronBProp;
       friend class NeuronBPropBatch;
       friend class NeuronBPropM;
       friend class NeuronTDNNBPropM;
public:
       NeuralNetBase();
       virtual ~NeuralNetBase();
       char *GetVersion(void);
       double GetOutputError(void);
       double GetOutputErrorMax(void);
       double GetOutputErrorMin(void);
       long GetMaxOutput(void);
       long GetMaxOutput(long,long);
       void SetLRate(double);

       // Virtual functions
       virtual void EndEpoch(void);
       virtual void Crossing(NeuralNetBase &,double,bool);
       virtual int BackPropagate(void);

       //'pure' virtual functions
       virtual int AddNeuron(int)                                  = 0;
       virtual int AddNeurons(int,long)                    = 0;
       virtual int CleanNet(void)                                  = 0;
       virtual int ConnectFullLayer(int,int)               = 0;
       virtual int ConnectNeurons(int,long,int,long)       = 0;
       virtual void GetOutput(double *)                    = 0;
       virtual long GetLayerNum(int)                       = 0;
       virtual int GetLayerNum(void)                       = 0;
       virtual double GetLRate(void)                       = 0;
       virtual long GetOutputNum(void)                             = 0;
       virtual double GetOutput(int,long)                  = 0;
       virtual long GetInputNum(void)                              = 0;
       virtual long GetConnectionNum(void)                 = 0;
       virtual long GetUWeightNum(void)                    = 0;   // number of updated
weights
       virtual void InitConnections(double,double)         = 0;
       virtual inline int IsNHidden(int)                   = 0;
       virtual void Propagate(void)                        = 0;
       virtual void SetInput(double *)                             = 0;
       virtual void SetTeachingInput(double *)             = 0;
       virtual int Save(char *)                                    = 0;
       virtual int Load(char *)                                    = 0;

       long NumProp,
              NumBProp;
       int iType;

protected:
```

```
        char szVersion[__MAXNNVERSIONLENGTH];
        double *dpTInput;
        double dLRate;
};

#endif//__NEURALNETBASEDEF_H
```

NeuralNetBase.h

```
#ifndef __NEURALNETBASE_H
#define __NEURALNETBASE_H

#include "Liste.h"
#include "../Defs/NeuralNetBaseDef.h"
#include "NeuronBase.h"

#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <memory.h>

char *NNType[5]={"No Information","BackProp","BackPropM","BackPropB","BackPropMTDNN"};

NeuralNetBase:: NeuralNetBase(){
	char szbuffer[]="LampelSoft Neuronale Netze-Simulator V 2.2";	// String which
could be read by calling NeuralNetBase::GetVersion

	strcpy(szVersion,szbuffer);

	srand(time(NULL));
		// Inititalization of randomgenerator

	dpTInput=0;
	dLRate=.1;
	NumProp = NumBProp = 0;
	iType = NN_NOINFORMATION;
}

NeuralNetBase :: ~NeuralNetBase(){
}

void NeuralNetBase :: EndEpoch(void){
}

void NeuralNetBase :: Crossing(NeuralNetBase &Param,double dPoint,bool bFlag){		//
function has to be written for each class - couldn't be generalized
	try{
		throw FunctionNotImpl();
	}
	catch(FunctionNotImpl &E){
		cerr << "void NeuralNetBase :: Crossing(NeuralNetBase &,double,bool) ---
Function isn't implementated for this type of net (yet)" << endl;
		throw(E);
	}
}

int NeuralNetBase :: BackPropagate(void){
	try{
		throw FunctionNotImpl();

		return 1;
	}
	catch(FunctionNotImpl &E){
		cerr << "int NeuralNetBase :: BackPropagate(void) --- Function isn't
implementated for this type of net (yet)" << endl;
		throw(E);
	}
}

double NeuralNetBase :: GetOutputError(void){			//Gibt den den
durchschnittlichen Ausgabefehler der Ausgabeneuronen des Netzes zurück
	long OutputNum = GetOutputNum();
	if(dpTInput && OutputNum){
		double dError=0;
		double *dOutput;
		GetOutput(dOutput = new double[OutputNum]);
		for(long lschl=0;lschl<OutputNum;lschl++){
			dError += fabs(dpTInput[lschl]-dOutput[lschl]);
		}
		delete [] dOutput;
		return dError/(double)(OutputNum);
	}
	else
		return 0;
}
```

```
double NeuralNetBase :: GetOutputErrorMax(void){          //Gibt den den größten
Ausgabefehler der Ausgabeneuronen des Netzes zurück
	long OutputNum = GetOutputNum();
	if(dpTInput && OutputNum){
		double dError=0;
		double dtemp;
		double *dOutput;
		GetOutput(dOutput = new double[OutputNum]);
		for(long lschl=0;lschl<OutputNum;lschl++){
			dtemp = fabs(dpTInput[lschl]-dOutput[lschl]);
			dError=dError>dtemp?dError:dtemp;
		}
		delete [] dOutput;
		return dError;
	}
	else
		return 0;
}

double NeuralNetBase :: GetOutputErrorMin(void){                  //Gibt den den kleinsten
Ausgabefehler der Ausgabeneuronen des Netzes zurück
	long OutputNum = GetOutputNum();
	if(dpTInput && OutputNum){
		double dError=1000;
		double dtemp;
		double *dOutput;
		GetOutput(dOutput = new double[OutputNum]);
		for(long lschl=0;lschl<OutputNum;lschl++){
			dtemp = fabs(dpTInput[lschl]-dOutput[lschl]);
			dError=dError>dtemp?dtemp:dError;
		}
		delete [] dOutput;
		return dError;
	}
	else
		return 0;
}

char * NeuralNetBase :: GetVersion(void){
	return (szVersion);
}

void NeuralNetBase :: SetLRate(double dParam){
	dLRate = dParam;
}

long NeuralNetBase :: GetMaxOutput(void){
	long OutputNum = GetOutputNum();
	if(dpTInput && OutputNum){
		long lMax=0;
		double dMax = -100;
		double *dOutput;
		GetOutput(dOutput = new double[OutputNum]);
		for(long lschl=0;lschl<OutputNum;lschl++){
			if(dOutput[lschl]>dMax){
				dMax=dOutput[lschl];
				lMax=lschl;
			}
		}
		delete dOutput;
		return lMax;
	}
	else
		return -1;
}

long NeuralNetBase :: GetMaxOutput(long llB,long luB){
	long OutputNum = GetOutputNum();
	if(dpTInput && OutputNum&&
		llB < OutputNum && luB+1 < OutputNum){
		long lMax=0;
		double dMax = -100;
		double *dOutput;
		GetOutput(dOutput = new double[OutputNum]);
		for(long lschl=llB;lschl<=luB;lschl++){
			if(dOutput[lschl]>dMax){
				dMax=dOutput[lschl];
```

```
                                lMax=lschl;
                        }
                }
                delete dOutput;
                return lMax;
        }
        else
                return -1;
}

#endif//__NEURALNETBASE_H
```

NeuralNetBPropDef.h

```
#ifndef __NEURALNETBPROPDEF_H
#define __NEURALNETBPROPDEF_H

// NEURALNETBPROP :: SCELETIZE AND NEURALNETBPROP : SCELETIZER ARE NOT WORKING CORECT

// written by Johannes Lampel / LampelSoft '00
//
// Die Klasse NeuralNetBProp ist eine öffentliche Klasse:
//
// Funktionen :
// NeuralNetBProp       : Allokiert Speicher für die Listen, die die einzelnen Neuronen der
jeweiligen
//                                        Layer aufnehmen sollen. Außerdem wird die im
1Param übergebene Zahl der Layer
//                                        in der insternen Variable iLayerNum abgespeichert.
// ~NeuralNetBProp      : Gibt das Feld mit Listen wieder frei.
// AddNeuron            : Fügt Neuron auf dem 1Param Layer hinzu.
// AddNeurons           : Fügt 2Param Neuronen auf 1Param Layer ein.
// BackPropagate        : Propagiert das Netz zurück, wobei jedoch dpTInput !=0 sein muss, da
//                                        sonst die Funktion abgebrochen wird. BackPropagate
verwendet sogenanntes
//                                        Vanilla - BProp , dass heißt es wurden keine den
Lernprozess verschnellernden
//                                        Faktoren, wie bspw. der Momentum - Term
implementiert.
// CleanNet           : Sucht nach Gewichten in der Netz - globalen Gewichtsliste und löscht
Gewicht, die nicht
//                      benötigt werden. Dies kommt beispielsweise dann vor, wenn man mit
//                      NeuralNetBProp :: Sceletize Gewicht mit einem kleinem absoluten
Gewichtswert
//                      gelöscht hat. In der Funktion NeuralNetBProp :: Sceletize werden nur
die
//                      Zeiger auf Gewichtsinstanzen gelöscht, nicht jedoch die Instanzen
selbst.
// ConnectFullLayer   : Verbindet die Layer 1Param und 2Param vollständig
//                                        (Bis jetzt noch kein Test ob Verbindungen doppelt)
// ConnectNeurons       : Verbindet das 2Parm Neuron auf 1Param Layer mit dem 4Param Neuronauf
3Param
//                                        Layer.
//                                        (Bis jetzt noch kein Test, ob Verbindungen
doppelt)
// GetHOutput           : Gibt Nummer der Neurons mit dem höchsten Output auf dem Output -
Layer
//                                        zurück.
// GetLayerNum(int)   : Gibt Numahl der Neuronen auf 1Param Layer zurück
// GetLayerNum(void)  : Gibt Numahl der Layer zurück. (iLayerNum)
// GetOutput(int,long): Gibt Output des 2Param Neurons auf Layer 1Param zurück
// GetOutput(double *): Schreibt die Outputs der Neuronen des Ausgabelayers in das Feld
1Param.
// GetInputNum          : Gibt Anzahl der Neuronen auf dem Input - Layer zurück.
// GetConnectionNum   : Gibt die Anzahl der Verbindungen zurück
// InitConnections      : Inititalisiert die Verbindungen mit zufälligen Werten zwischen
1Param
//                                        und 2Param.
// IsNHidden            : Gibt 1 zurück,wenn das 2Param Neuron auf Layer 1Param kein
OuputNeuron ist
// Propagate            : Propagiert das Netz
// Sceletize          : Löscht alle Gewichte, deren Absolutwert unter 1Param liegt. (Siehe
auch CleanNet)
// SetInput             : Setzt die Eingabewerte eines Netzes, damit es danach propagiert
werden kann
//                                        Der 1Param muss auf ein Feld zeigen, welches
mindestens so viele double -
//                                        Elemente hat, wie es Eingabeneuronen gibt, da es
sonst einen Ausnahmefehler
//                                        gibt.
// SetTeachingInput   : Setzt TeachingInput fest, also das Ergebnis ,dem sich
BProp(BackPropagation)
//                                        annähern soll. Hierbei gilt das gleich in Bezug
auf die Größe des Feldes,
//                                        wie bei NeuralNetBProp :: SetInput, nur auf die
Anzahl der Ausgabeneuronen
//                                        bezogen.
// Save                         : Speichert das Netz in die 1Param - Datei ab
// Load                         : Lädt Netz aus 1Param - Datei
```

```
//                      Sofern das Netz der Datei kein NeuralNetBProp - Netz ist (oder
entsprechend)
//                      wird es in den Netz - Typ der Instanz konvertiert, der die Methode
'Load'
//                      angehört. Die Variable NeuralNetBase :: iType bekommt den
entsprechenden
//                      Wert, der dem geladenem Netz entspricht. Die Definitionen für die
einzelnen
//                      Netztypen finden sich in NeuralNet.h
// UpdateNet            : Aktualisiert die Positionsangaben innerhalb der Neuronen. Korrekte
Angaben
//                                      werden für NeuralNetBProp :: Load bzw Save
benötigt.
// TestNetErrors      : Testet das Netz auf Fehler. Treten Fehler auf, so werden entsprechende
//                      Fehlermeldungen auf -- STDERR -- ausgegeben.
//
// DatenElemente :
// LWeights             : Liste aller Gewichte des Netzes.
// LLayerN                      : Feld von Listen mit Neuronen, wobei jede Liste einen Layer
beinhaltet.
// iLayerNum            : Anzahl der Layer , d.h. Größe des LLayerN - Feldes.
//
// ********************************************************************************
// Funktionen von NeuronBProp
//
// CalcOutput           : Berechnet aktuelle Ausgabe des Neurons aus den Ausgaben der
vorhergehenden
//                                      Neuronen,den entspechden Gewichten,der
Aktivierungsfunktion und des
//                                      Schwellenwertes des Neurons.
// CalcDelta            : Berechnet den Propagierungsfehler des Neurons

#include "NeuralNetBaseDef.h"
#include "NeuronBaseDef.h"
#include "WeightBaseDef.h"

class DllExport WeightBProp : public WeightBase
{
public:
protected:
};

class DllExport NeuronBProp : public NeuronBase
{
public:
       void CalcOutput(void);
       void CalcDelta(void);
protected:
};

class DllExport NeuralNetBProp : public NeuralNetBase
{
       friend class NeuronBProp;
public:
       void test(void);
       NeuralNetBProp(int);
       NeuralNetBProp();
       virtual ~NeuralNetBProp();
       int AddNeuron(int);
       int AddNeurons(int,long);
       int BackPropagate(void);
       int CleanNet(void);
       int ConnectFullLayer(int,int);
       int ConnectNeurons(int,long,int,long);
       int ConnectNeurons(int,long,int,long,WeightBProp &);
       void Crossing(NeuralNetBProp &,double,bool);
       void GetOutput(double *);
       long GetLayerNum(int);
       int GetLayerNum(void);
       double GetLRate(void);
       NeuronBProp *GetNeuron(int,long);
       long GetOutputNum(void);
       double GetOutput(int,long);
       long GetInputNum(void);
       long GetConnectionNum(void);
       long GetUWeightNum(void);
       void InitConnections(double,double);
       inline int IsNHidden(int);
```

```
        void Propagate(void);
        void SetInput(double *);
        void SetLayerNum(const int);
        void SetTeachingInput(double *);
        long Sceletize(double);
        void TestNetErrors(void);
        int Save(char *);
        int Load(char *);
        virtual const NeuralNetBProp &operator=(const NeuralNetBProp &);

//protected:
        void UpdateNet(void);

        Liste <WeightBProp> LWeights;
        Liste <NeuronBProp> *LLayerN;
        int iLayerNum;
};

#endif//__NEURALNETBPROPDEF_H
```

NeuralNetBProp.h

```
#ifndef __NEURALNETBPROP_H
#define __NEURALNETBPROP_H

#include <iostream.h>
#include <stdlib.h>
#include <math.h>

#include "NeuralNetBase.h"
#include "../Defs/NeuralNetBPropDef.h"

void NeuronBProp :: CalcOutput(void)        //Output of all Neurons is being calculated
{
        long lschl;
        LinkBase *lnk;

        Net=.0;

        for(lschl = LptrNf.GetAnz();lschl;lschl--){
                lnk = LptrNf[lschl];
                assert (lnk!=0);
                Net+=lnk->Nptr->Out * lnk->Wptr->dWeight;
        }

        Out = fAct(Net-Bias);
}

void NeuronBProp :: CalcDelta(void)         // calculating Delta of each Neuron
{
        double  dTemp;
        assert(NN!=0);
        if(((NeuralNetBProp*)(NN))->IsNHidden(iNLayer)){
                LinkBase       *LptrNext;
        //wenn das Neuron nicht auf dem AusgabeLayer liegt
                long    lschl;
                double dW;

                dTemp = 0;

                for(lschl=LptrNt.GetAnz();lschl;lschl--){
                        LptrNext = LptrNt[lschl];
                        assert (LptrNext!=0);
                        dW = LptrNext->Wptr->dWeight;
                        dTemp += (LptrNext -> Nptr -> Delta * dW);
                }
                Delta  = dTemp * fActS(Net);
                return /*(Delta  = dTemp * fActS(Net))*/;
        }
        else
        {
                Delta = fActS(Net) * ( NN->dpTInput[lNNeuron-1] - Out); //Neuron liegt auf dem
AusgabeLayer
                return /*(NptrAkt ->Delta)*/;
        }
}

NeuralNetBProp :: NeuralNetBProp(int iNum)
{
        //iNum<=0?iNum=1:iNum;                              // Ein Netz besitz immer mindestens
einen Layer, da es sonst Probleme bei der Deallokierung gibt
        LLayerN = new Liste<NeuronBProp>[iNum];
        iLayerNum = iNum;
        iType = NN_BACKPROP;
}

NeuralNetBProp :: NeuralNetBProp()
{
        iLayerNum = 0;
        LLayerN = 0;
}

NeuralNetBProp :: ~NeuralNetBProp()
{
        if(iLayerNum)
                delete [] LLayerN;
        iLayerNum = 0;
}
```

```
const NeuralNetBProp &NeuralNetBProp :: operator=(const NeuralNetBProp &Param)
{
	for(int inum = 0;inum < iLayerNum ; inum++){
		LLayerN[inum].DelAllData();
	}
	iLayerNum = Param.iLayerNum;
	LLayerN = new Liste<NeuronBProp>[iLayerNum];
	for(int ischl=0;ischl<iLayerNum;ischl++){
		LLayerN[ischl] = Param.LLayerN[ischl];
	}
	LWeights = Param.LWeights;

	// NeuralNetBase - variables
	dLRate = Param.dLRate;
	iType = Param.iType;
	NumBProp = Param.NumBProp;
	NumProp = Param.NumProp;

	if(dpTInput)
		delete [] dpTInput;
	dpTInput = 0;

	return ((NeuralNetBProp &)(*this));
}

int NeuralNetBProp :: AddNeuron(int iLayer)
{
	if(iLayer<=iLayerNum){
		NeuronBProp Neuron;
		Neuron.NN = this;
		LLayerN[iLayer].AddElem(Neuron);
		UpdateNet();
		return 1;
	}
	else
		return 0;
}

int NeuralNetBProp :: AddNeurons(int iLayer,long lNeurons)
{
	if(iLayer<=iLayerNum){
		long lschl;
		NeuronBProp Neuron;
		Neuron.NN = this;
		for(lschl=0;lschl<lNeurons;lschl++){
			LLayerN[iLayer].AddElem(Neuron);
		}
		UpdateNet();
		return 1;
	}
	else
		return 0;
}

int NeuralNetBProp :: BackPropagate(void)
{
	if(dpTInput){
		if(iLayerNum!=1){
			int    ischl;
			long   lschl,
			       lfschl;
			NeuronBProp *NptrAkt;
			LinkBase      *LptrAkt;
			WeightBProp *WptrAkt;
			double ddiff;

			NumBProp ++;
			for(ischl = iLayerNum-1;ischl>0;ischl--){
				for(lschl = LLayerN[ischl].GetAnz();lschl;lschl--){
					NptrAkt = LLayerN[ischl][lschl];
	// Berechnet Delta der einzelnen Neuronen
					NptrAkt -> CalcDelta();
					NptrAkt -> Bias -= dLRate * NptrAkt->Delta; // Ändert die
Schwellenwerte
				}
			}
```

```
                        for(ischl =iLayerNum-1;ischl>=0;ischl--){
                                for(lschl = LLayerN[ischl].GetAnz();lschl;lschl--){
                                        NptrAkt = LLayerN[ischl][lschl];

                                        for(lfschl = NptrAkt->LptrNt.GetAnz();lfschl;lfschl--){
                                                LptrAkt = NptrAkt->LptrNt[lfschl];
                                                WptrAkt = (WeightBProp*)LptrAkt -> Wptr;

                                                ddiff = dLRate * NptrAkt -> Out * LptrAkt->Nptr-
>Delta;

                                                WptrAkt -> dWeight += ddiff;
        // ändert Gewichte
                                        }
                                }
                        }
                        return 1;
                }
                else
                        return 0;
        }
        else
                return 0;
}

int NeuralNetBProp :: CleanNet(void)                                    // Beseitigt nicht benötigte
Gewichte aus der Liste der Gewichte LWeights
{
        long lschl,lNschl,lWschl;
        int ischl;
        NeuronBProp *NptrAkt;

        for(lschl=1;lschl<=LWeights.GetAnz();lschl++){              //Alle Gewicht durchgehen
                if(LWeights[lschl]->luse <= 0){                                         //Wenn ein
Gewicht nicht benutzt wird
                        for(ischl = 0;ischl<iLayerNum;ischl++){
        //Alle Neuronen durchgehen (Layer)
                                for(lNschl = 1;lNschl<=LLayerN[ischl].GetAnz();lNschl++){
        //Alle Neuronen durchgehen (Neuronen / Layer)
                                        NptrAkt = (NeuronBProp*)LLayerN[ischl][lNschl];

                                        for(lWschl=1;lWschl<=NptrAkt->LptrNt.GetAnz();lWschl++){
                                                if (NptrAkt->LptrNt[lWschl]->lWeight > lschl){
        // Evtl. Variable mit Nummer des Gewichtes dekrementieren
                                                        NptrAkt->LptrNt[lWschl]->lWeight--;
                                                        NptrAkt->LptrNt[lWschl]->Wptr =
LWeights[NptrAkt->LptrNt[lWschl]->lWeight];// und den Zeiger auf das Gewicht updaten
                                                }
                                        }
                                }
                        }
                        LWeights.DelElem(lschl);
                }
        }
        return 1;
}

int NeuralNetBProp :: ConnectFullLayer(int ifLayer,int isLayer)
{
        if(ifLayer<=iLayerNum&&isLayer<=iLayerNum){
                long lfschl,lsschl;
                for(lfschl=LLayerN[ifLayer].GetAnz();lfschl;lfschl--){
                        for(lsschl=LLayerN[isLayer].GetAnz();lsschl;lsschl--){
                                ConnectNeurons(ifLayer,lfschl,isLayer,lsschl);
                        }
                }
        }
        UpdateNet();
        return 1;
}

int NeuralNetBProp :: ConnectNeurons(int ifLayer,long lfNeuron,int isLayer,long lsNeuron)
{
        LinkBase        Lit,
                Lif;
        WeightBProp Wtmp;

        if(ifLayer<=iLayerNum&&isLayer<=iLayerNum){
```

```
                if(lfNeuron<=LLayerN[ifLayer].GetAnz()&&lsNeuron<=LLayerN[isLayer].GetAnz()){
                        //cout << lfNeuron << " : " << lsNeuron <<endl;
                        Lit.Nptr = LLayerN[isLayer][lsNeuron];
                        Lif.Nptr = LLayerN[ifLayer][lfNeuron];
                        Lit.Wptr=Lif.Wptr=LWeights.AddElem(Wtmp);
                        Lit.Wptr->luse ++;
                        assert(Lit.Wptr!=0);
                        Lit.lWeight=Lif.lWeight=LWeights.GetAnz();
                        //Lit.Wptr=Lif.Wptr = LWeights[Lit.lWeight];

                        LLayerN[ifLayer][lfNeuron]->LptrNt.AddElem(Lit);
                        LLayerN[isLayer][lsNeuron]->LptrNf.AddElem(Lif);
                        //Lit.Nptr=Lif.Nptr=0;
                        //LLayerN[ifLayer][lfNeuron]-
>ConnectNeuronTo((NeuronBase*)(LLayerN)[isLayer][lsNeuron]);
                }
        }
        return 1;
}

int NeuralNetBProp :: ConnectNeurons(int ifLayer,long lfNeuron,int isLayer,long
lsNeuron,WeightBProp &WParam)
{
        LinkBase        Lit,
                Lif;

        if(ifLayer<=iLayerNum&&isLayer<=iLayerNum){
                if(lfNeuron<=LLayerN[ifLayer].GetAnz()&&lsNeuron<=LLayerN[isLayer].GetAnz()){
                        //cout << lfNeuron << " : " << lsNeuron <<endl;
                        Lit.Nptr = LLayerN[isLayer][lsNeuron];
                        Lif.Nptr = LLayerN[ifLayer][lfNeuron];
                        Lit.Wptr=Lif.Wptr = LWeights.AddElem(WParam);
                        Lit.Wptr->luse ++;
                        assert(Lit.Wptr!=0);
                        Lit.lWeight=Lif.lWeight=LWeights.GetAnz();

                        LLayerN[ifLayer][lfNeuron]->LptrNt.AddElem(Lit);
                        LLayerN[isLayer][lsNeuron]->LptrNf.AddElem(Lif);
                        //Lit.Nptr=Lif.Nptr=0;
                        //LLayerN[ifLayer][lfNeuron]-
>ConnectNeuronTo((NeuronBase*)(LLayerN)[isLayer][lsNeuron]);
                }
        }
        return 1;
}

void NeuralNetBProp :: Crossing (NeuralNetBProp &NNParam,double dPoint,bool bFlag){
        // for evol. alg
        // bflag = 0    ->    0 to break is copied
        // bflag = 1    ->    break to end is copied
        long lWAnz = LWeights.GetAnz();
        if(dPoint < 1.0 && NNParam.LWeights.GetAnz() == lWAnz){
                long lBorder = long(dPoint * (double)(lWAnz));
                long lschl;

                if(bFlag){
                        for(lschl = lBorder;lschl < lWAnz;lschl++){
                                *LWeights[lschl] = *NNParam.LWeights[lschl];
                        }
                }
                else{
                        for(lschl = 0;lschl < lBorder;lschl++){
                                *LWeights[lschl] = *NNParam.LWeights[lschl];
                        }
                }
        }
}

void NeuralNetBProp :: GetOutput(double *dptr)
{
        for(long lschl = LLayerN[iLayerNum-1].GetAnz();lschl;lschl--)
                dptr[lschl-1] = LLayerN[iLayerNum-1][lschl]->Out;
}

long NeuralNetBProp :: GetLayerNum(int iLayer)
{
        if(iLayer<=iLayerNum){
                return LLayerN[iLayer].GetAnz();
```

```
	}
	else
		return 0;
}

int NeuralNetBProp :: GetLayerNum(void)
{
	return iLayerNum;
}

double NeuralNetBProp :: GetLRate(void)
{
	return dLRate;
}

NeuronBProp *NeuralNetBProp :: GetNeuron(int i,long l)
{
	return LLayerN[i][l];
}


long NeuralNetBProp :: GetConnectionNum(void)	//Geht alle Neuronen Layerweise durch und
zählt jeweils ihre LptrNto - Numahlen fest und addiert sie
{
	long	lschl,
		ltemp=0;
	int	inum;

	for(inum=0;inum<iLayerNum;inum++)
		for(lschl=LLayerN[inum].GetAnz();lschl;lschl--){
			ltemp += LLayerN[inum][lschl]->LptrNt.GetAnz();			//
Weights of neurons
			//ltemp++;
				// don't add Bias of each Neuron cause this function should only
count the connections between neurons, otherwise the load and save functions would not run
properly
		}

	return ltemp;
}

long NeuralNetBProp :: GetUWeightNum(void)  //Geht alle Neuronen Layerweise durch und zählt
jeweils ihre LptrNto - Numahlen fest und addiert sie
{
	long	lschl,
		ltemp=0;
	int	inum;

	for(inum=0;inum<iLayerNum;inum++)
		for(lschl=LLayerN[inum].GetAnz();lschl;lschl--){
			ltemp += LLayerN[inum][lschl]->LptrNt.GetAnz();			//
Weights of neurons
			ltemp++;
				// neuron bias
		}

	return ltemp;
}

long NeuralNetBProp :: GetOutputNum(void)
{
	return LLayerN[iLayerNum-1].GetAnz();
}

double NeuralNetBProp :: GetOutput(int iParam,long lParam)
{
	return LLayerN[iParam][lParam]->Out;
}

long NeuralNetBProp :: GetInputNum(void)
{
	return LLayerN[0].GetAnz();
}

void NeuralNetBProp :: InitConnections(double dmin,double dmax)
{
/*for(long lschl=LWeights.GetAnz();lschl;lschl--)
LWeights[lschl]->dWeight=(double)rand()/RAND_MAX*(dmax-dmin)+dmin;
```

```
	*/
	Listeelem <WeightBProp> *LEakt;
	long lschl;
	LEakt = LWeights.fElem;
	if(LEakt==0)
		return;
	do{
		for(lschl=0;lschl<LEakt->laktf;lschl++){
			LEakt->Data[lschl].dWeight = (double)rand()/RAND_MAX*(dmax-dmin)+dmin;
		}
		for(long llschl=0;llschl < iLayerNum;llschl++){
			for(long lnschl=1;lnschl <= LLayerN[llschl].GetAnz();lnschl++){
				LLayerN[llschl][lnschl]->Bias = (double)rand()/RAND_MAX*(dmax-
dmin)+dmin;
			}
		}
	}while((LEakt=LEakt->nextelem));

}

int NeuralNetBProp :: IsNHidden(int iLayer)
{
	if(iLayer==iLayerNum-1)
		return 0;
	else
		return 1;
}

void NeuralNetBProp :: UpdateNet(void)
{
	long lschl;
	int ischl;

	for(ischl=0;ischl<iLayerNum;ischl++){                          // Setting internal
NeuronVariables of postion
		for(lschl=1;lschl<=LLayerN[ischl].GetAnz();lschl++){
			LLayerN[ischl][lschl]->iNLayer = ischl;
			LLayerN[ischl][lschl]->lNNeuron =lschl;
		}
	}
}

void NeuralNetBProp :: Propagate(void)
{
	int iLschl;
	long lNNumpL;
	NumProp++;
	for(iLschl=1;iLschl<iLayerNum;iLschl++)
		for(lNNumpL=LLayerN[iLschl].GetAnz();lNNumpL;lNNumpL--)
			LLayerN[iLschl][lNNumpL]->CalcOutput();
}

void NeuralNetBProp :: SetInput(double *dptr)
{
	for(long lschl=1;lschl<=GetInputNum();lschl++)
		LLayerN[0][lschl]->Out = dptr[lschl-1];
}

void NeuralNetBProp :: SetLayerNum(const int iParam)
{
	for(int inum = 0;inum < iLayerNum ; inum++){
		LLayerN[inum].DelAllData();
	}
	delete [] LLayerN;

	LLayerN = new Liste<NeuronBProp>[iParam];
	iLayerNum = iParam;
	iType = NN_BACKPROP;
}

void NeuralNetBProp :: SetTeachingInput(double *dptr)
{
	if(dpTInput)
		delete [] dpTInput;

	dpTInput = new double[GetOutputNum()];
	memcpy(dpTInput,dptr,GetOutputNum()*sizeof(double));
}
```

```
long NeuralNetBProp :: Sceletize (double drad)
{
	cout << "DON'T USE long NeuralNetBProp :: Sceletize (double drad) - it's causing a lot
of page faults later !!" << endl;
	long lanz=0;
	int ischl;
	long	lschl,
		lLschl,
		lNschl,
		ltemp;

	bool flag;
	NeuronBProp *NptrAkt;

	for(ischl=0;ischl<iLayerNum;ischl++)			//Alle Neuronen durchgehen
		for(lschl=1;lschl<=LLayerN[ischl].GetAnz();lschl++){
#ifdef _DEBUG
			if(lschl>LLayerN[ischl].GetAnz()){
				cerr <<"KACKE"<<endl;
			}
#endif
			for(lLschl = 1;lLschl<=LLayerN[ischl][lschl]-
>LptrNt.GetAnz();lLschl++){//Alle Links des Neurons durchgehen
				if(lLschl>LLayerN[ischl][lschl]->LptrNt.GetAnz()){
					cerr <<"SCHEISSE -
"<<ischl<<":"<<lschl<<":"<<lLschl<<":"<<LLayerN[ischl][lschl]->LptrNt.GetAnz()<<endl;
					continue;
				}
				assert(LLayerN[ischl][lschl]!=0);
				assert(LLayerN[ischl][lschl]->LptrNt[lLschl]!=0);
				assert(LLayerN[ischl][lschl]->LptrNt[lLschl]->Wptr!=0);

				if (fabs(LLayerN[ischl][lschl]->LptrNt[lLschl]->Wptr-
>dWeight)<drad){
					ltemp = LLayerN[ischl][lschl]->LptrNt[lLschl]->lWeight;
					NptrAkt = (NeuronBProp*)LLayerN[ischl][lschl]-
>LptrNt[lLschl]->Nptr;
					flag = 0;
					for(lNschl=1;lNschl<=NptrAkt->LptrNf.GetAnz();lNschl++){
						if(NptrAkt->LptrNf[lNschl]->lWeight == ltemp){
							flag = 1;
							break;
						}
					}
					assert(flag!=0);
					LLayerN[ischl][lschl]->LptrNt.DelElem(lLschl);
					NptrAkt->LptrNf.DelElem(lNschl);
					//LWeights.DelElem(ltemp);
					LWeights[ltemp]->luse--;
					lanz++;
					lLschl--;
					lNschl--;
					//TestNetErrors();
				}
			}
		}
		return lanz;
}

int NeuralNetBProp :: Save(char * szname)
{
	int	inum;
	long	lConNum,
		lschlN,
		lschlto;
	FILE	*fhd;
	NeuronBProp *fNeuron,
		*sNeuron;
	Liste <NeuronBProp> *LaktN;

	UpdateNet();
	try{
		if(!(fhd=fopen(szname,"wb")))
			throw FileOpenError();
		iType = NN_BACKPROP;

		fwrite(szVersion,sizeof(char),__MAXNNVERSIONLENGTH,fhd);
```

```
                fwrite (&iType,sizeof(int),1,fhd);
                fwrite(&NumProp,sizeof(long),1,fhd);
                fwrite(&NumBProp,sizeof(long),1,fhd);
                fwrite(&dLRate,sizeof(double),1,fhd);
                fwrite(&iLayerNum,sizeof(int),1,fhd);

                for(inum=0;inum<iLayerNum;inum++){
                        LaktN=&LLayerN[inum];
                        LaktN->LSave(fhd);
                }

                lConNum = GetConnectionNum();

                fwrite(&lConNum,sizeof(long),1,fhd);

                for(inum=0;inum < iLayerNum;inum++)
                        if(LLayerN[inum].GetAnz())
                                for(lschlN=1;lschlN<=LLayerN[inum].GetAnz();lschlN++){
                                        fNeuron = LLayerN[inum][lschlN];
                                        if(fNeuron->LptrNt.GetAnz())
                                                for(lschlto = 1;lschlto <= fNeuron-
>LptrNt.GetAnz();lschlto++){
                                                        sNeuron = (NeuronBProp*)fNeuron-
>LptrNt[lschlto]->Nptr;

                                                        fwrite(&fNeuron-
>iNLayer,sizeof(int),1,fhd);
                                                        fwrite(&fNeuron-
>lNNeuron,sizeof(long),1,fhd);

                                                        fwrite(&sNeuron-
>iNLayer,sizeof(int),1,fhd);
                                                        fwrite(&sNeuron-
>lNNeuron,sizeof(long),1,fhd);

                                                        fwrite(&LWeights[fNeuron->LptrNt[lschlto]-
>lWeight]->dWeight,sizeof(double),1,fhd);
                                                }
                                }
                                fclose(fhd);
                                return 1;
        }
        catch(FileOpenError){
                cerr << "Error opening file for writing: " << szname<<endl;
                throw FileOpenError();
        }
        catch(...){
                cerr << "UNKNOWN ERROR IN " << __FILE__ << __LINE__ << __DATE__ <<endl;
                throw;
        }
}

int NeuralNetBProp :: Load(char * szname)
{
        FILE    *fhd;
        int     inum;
        int     ifLayer,
                isLayer,
                ischl;
        long    lfNNum,
                lsNNum,
                lConNum,
                lschl;
        WeightBProp CWeight;
        double dtemp;
        Liste <NeuronTDNNBPropM> *TempLLayerN;

        try{
                if(!(fhd = fopen(szname,"rb")))
                        throw FileOpenError();
                rewind(fhd);
                if(iLayerNum)
                        delete [] LLayerN;
                iLayerNum = 0;

                fread(szVersion,sizeof(char),__MAXNNVERSIONLENGTH,fhd);
                fread(&iType,sizeof(int),1,fhd);
                fread(&NumProp,sizeof(long),1,fhd);
```

```
		fread(&NumBProp,sizeof(long),1,fhd);
		switch(iType){
		case NN_BACKPROP:
			fread(&dLRate,sizeof(double),1,fhd);
			fread(&iLayerNum,sizeof(int),1,fhd);

			LLayerN = new Liste <NeuronBProp>[iLayerNum];

			for(inum=0;inum<iLayerNum;inum++){
				if(!((&LLayerN[inum])->LLoad(fhd)))
					throw FileBadDataError();
			}

			UpdateNet();
			fread(&lConNum,sizeof(long),1,fhd);
			for(lschl=0;lschl<lConNum;lschl++){
				fread(&ifLayer,sizeof(int),1,fhd);
				fread(&lfNNum,sizeof(long),1,fhd);

				fread(&isLayer,sizeof(int),1,fhd);
				fread(&lsNNum,sizeof(long),1,fhd);

				fread(&dtemp,sizeof(double),1,fhd);
				CWeight.dWeight = dtemp;

				ConnectNeurons(ifLayer,lfNNum,isLayer,lsNNum,CWeight);
			}
			break;

		case NN_TDNNBACKPROPM:
			fread(&dLRate,sizeof(double),1,fhd);
			fread(&dtemp,sizeof(double),1,fhd);
			fread(&iLayerNum,sizeof(int),1,fhd);

			LLayerN = new Liste <NeuronBProp>[iLayerNum];
			TempLLayerN = new Liste <NeuronTDNNBPropM>[iLayerNum];

			for(inum=0;inum<iLayerNum;inum++){
				if(!((&TempLLayerN[inum])->LLoad(fhd)))     // in TDNN Netz
laden
					throw FileBadDataError();
			}
			// vom TDNN - Netz in anderes Netz konvertieren
			for(ischl=0;ischl < iLayerNum;ischl++){
				for(lschl=1;lschl<=TempLLayerN[ischl].GetAnz();lschl++){
					AddNeuron(ischl);
					LLayerN[ischl][LLayerN[ischl].GetAnz()]->Bias =
TempLLayerN[ischl][lschl]->Bias;
				}
			}
			UpdateNet();
			fread(&lConNum,sizeof(long),1,fhd);
			for(lschl=0;lschl<lConNum;lschl++){
				fread(&ifLayer,sizeof(int),1,fhd);
				fread(&lfNNum,sizeof(long),1,fhd);

				fread(&isLayer,sizeof(int),1,fhd);
				fread(&lsNNum,sizeof(long),1,fhd);

				fread(&dtemp,sizeof(double),1,fhd);
				CWeight.dWeight = dtemp;
				fread(&dtemp,sizeof(double),1,fhd);                //

				ConnectNeurons(ifLayer,lfNNum,isLayer,lsNNum,CWeight);
			}
			break;

		case NN_BACKPROPM:
			fread(&dLRate,sizeof(double),1,fhd);
			fread(&dtemp,sizeof(double),1,fhd);
			fread(&iLayerNum,sizeof(int),1,fhd);

			LLayerN = new Liste <NeuronBProp>[iLayerNum];

			for(inum=0;inum<iLayerNum;inum++){
				if(!((&LLayerN[inum])->LLoad(fhd)))
					throw FileBadDataError();
			}
```

```
                        UpdateNet();
                        fread(&lConNum,sizeof(long),1,fhd);
                        for(lschl=0;lschl<lConNum;lschl++){
                                fread(&ifLayer,sizeof(int),1,fhd);
                                fread(&lfNNum,sizeof(long),1,fhd);

                                fread(&isLayer,sizeof(int),1,fhd);
                                fread(&lsNNum,sizeof(long),1,fhd);

                                fread(&dtemp,sizeof(double),1,fhd);
                                CWeight.dWeight = dtemp;
                                fread(&dtemp,sizeof(double),1,fhd);                //

                                ConnectNeurons(ifLayer,lfNNum,isLayer,lsNNum,CWeight);
                        }
                        break;

                default:
                        throw FileUnknownTypeError();
                        return 0;
                }
                fclose (fhd);

                for(inum=0;inum<iLayerNum;inum++)
        //Allen Neuronen die Adresse ihres NN geben
                {
                        for(lschl=1;lschl<=LLayerN[inum].GetAnz();lschl++){
                                LLayerN[inum][lschl]->NN = this;
                        }
                }
                return 1;
        }
        catch(FileOpenError){
                cerr << "Error opening file for reading: " << szname<<endl;;
                throw FileOpenError();
        }
        catch(FileUnknownTypeError){
                cerr << "Unknown type of NN in " << szname<<endl;
                throw FileUnknownTypeError();
        }
        catch(FileBadDataError){
                cerr << "Error while reading file "<< szname <<" . Version - conflicts ???"
<<endl;
                throw FileBadDataError();
        }
        catch(...){
                cerr << "UNKNOWN ERROR IN " << __FILE__ << __LINE__ << __DATE__ <<endl;
                throw;
        }
}

void NeuralNetBProp :: test(void)
{
        for(int inum=0;inum<iLayerNum;inum++){
        //Allen Neuronen die Adresse ihres NN geben
                for(long lschl=0;lschl<=LLayerN[inum].GetAnz();lschl++){
                        cout << LLayerN[inum][lschl]->NN<<":";
                }
        }
}

void NeuralNetBProp :: TestNetErrors(void)                                          //
SELBSTKLÄRENDER NAME
{
        long lschl,lLschl;
        int ischl;
        for (ischl=0;ischl<iLayerNum;ischl++){
                for(lschl=1;lschl<=LLayerN[ischl].GetAnz();lschl++){
                /*Testing routines for neurons
                        */
                        if(fabs(LLayerN[ischl][lschl]->Bias)>100000.0){
                                cerr << "Bias == extrem; ischl:"<<ischl<<";
lschl:"<<lschl<<endl;
                        }

                        for(lLschl = 1;lLschl<=LLayerN[ischl][lschl]->LptrNt.GetAnz();lLschl++){
                        /*Testing routines for Link lists
```

```
                                */
                                if(LLayerN[ischl][lschl]->LptrNt[lLschl]->Wptr==0){
                                        cerr << "LptrNt :: Wptr == 0; ischl:"<<ischl<<";
lschl:"<<lschl<<"; lLschl:"<<lLschl<<"; Link.lWeight:"<<LLayerN[ischl][lschl]->LptrNt[lLschl]-
>lWeight<<"; Link.Nptr:"<<LLayerN[ischl][lschl]->LptrNt[lLschl]->Nptr<<endl;
                                }
                                if(LLayerN[ischl][lschl]->LptrNt[lLschl]->lWeight==0){
                                        cerr << "LptrNt :: lWeight == 0; ischl:"<<ischl<<";
lschl:"<<lschl<<"; lLschl:"<<lLschl<<"; Link.lWeight:"<<LLayerN[ischl][lschl]->LptrNt[lLschl]-
>lWeight<<"; Link.Nptr:"<<LLayerN[ischl][lschl]->LptrNt[lLschl]->Nptr<<endl;
                                }
                                if(LLayerN[ischl][lschl]->LptrNt[lLschl]->Nptr==0){
                                        cerr << "LptrNt :: Nptr == 0; ischl:"<<ischl<<";
lschl:"<<lschl<<"; lLschl:"<<lLschl<<"; Link.lWeight:"<<LLayerN[ischl][lschl]->LptrNt[lLschl]-
>lWeight<<"; Link.Nptr:"<<LLayerN[ischl][lschl]->LptrNt[lLschl]->Nptr<<endl;
                                }
                        }
                        for(lLschl = 1;lLschl<=LLayerN[ischl][lschl]->LptrNf.GetAnz();lLschl++){
                        /*Testing routines for Link lists
                                */
                                if(LLayerN[ischl][lschl]->LptrNf[lLschl]->Wptr==0){
                                        cerr << "LptrNf :: aWptr == 0; ischl:"<<ischl<<";
lschl:"<<lschl<<"; lLschl:"<<lLschl<<"; Link.lWeight:"<<LLayerN[ischl][lschl]->LptrNf[lLschl]-
>lWeight<<"; Link.Nptr:"<<LLayerN[ischl][lschl]->LptrNf[lLschl]->Nptr<<endl;
                                }
                                if(LLayerN[ischl][lschl]->LptrNf[lLschl]->lWeight==0){
                                        cerr << "LptrNf :: lWeight == 0; ischl:"<<ischl<<";
lschl:"<<lschl<<"; lLschl:"<<lLschl<<"; Link.lWeight:"<<LLayerN[ischl][lschl]->LptrNf[lLschl]-
>lWeight<<"; Link.Nptr:"<<LLayerN[ischl][lschl]->LptrNf[lLschl]->Nptr<<endl;
                                }
                                if(LLayerN[ischl][lschl]->LptrNf[lLschl]->Nptr==0){
                                        cerr << "LptrNf :: Nptr == 0; ischl:"<<ischl<<";
lschl:"<<lschl<<"; lLschl:"<<lLschl<<"; Link.lWeight:"<<LLayerN[ischl][lschl]->LptrNf[lLschl]-
>lWeight<<"; Link.Nptr:"<<LLayerN[ischl][lschl]->LptrNf[lLschl]->Nptr<<endl;
                                }
                        }
                }
        }
        cout << "sucessfully tested" << endl;
}

#endif//__NEURALNETBPROP_H
```

NeuronAktDef.h

```
#ifndef __NEURONAKTDEF_H
#define __NEURONAKTDEF_H

extern const int NEURONACTDEFAULT    = 1;

extern const int NEURONACTTANH              = 1;   // Params für Neuron :: SetActFunc(int);
extern const int NEURONACTLOG       = 2;   // für die einzelnen Sorten der
Aktivierungsfunktionen

// Forward -Def's
double fNActtanh(const double);             // Tangens hyperbolicus als Aktivierungsfunktion
double fNActtanhS(const double);     // und seine Ableitung
double fNActlog(const double);              // Die logistische Aktivierungsfunktion
double fNActlogS(const double);             // und ihre Ableitung

#endif
```

NeuronAkt.h

```
#ifndef __NEURONAKT_H
#define __NEURONAKT_H

//written by Johannes Lampel / LampelSoft '00
// written 12. Nov 1999

#include <math.h>

// Functions
double fNActtanh(const double dParam)
{
	return (tanh(dParam));
}

double fNActtanhS(const double dParam)
{
	return (1.1-(tanh(dParam)*tanh(dParam)));
}

double fNActlog(const double dParam)
{
	return (1/(1+exp(dParam)));
}

double fNActlogS(const double dParam)
{
	return (fNActlog(dParam)*(1-fNActlog(dParam)));
}

#endif//__NEURONAKT_H
```

NeuronBaseDef.h

```
#ifndef __NEURONBASEDEF_H
#define __NEURONBASEDEF_H

//NeuronBase.h written by Johannes Lampel / LampelSoft '00
//written : 18. Nov 1999

#include <stdio.h>
#include "ListeDef.h"
#include "NeuralNetBaseDef.h"
#include "LinkBaseDef.h"

class NeuralNetBase;

class  DllExport NeuronBase
{
public:
       NeuronBase();
       virtual ~NeuronBase();
       const NeuronBase &operator = (const NeuronBase &);
       //int ConnectNeuronTo(NeuronBase *);
       //virtual void CalcOutput(void);
       double (*fAct) (const double);
       double (*fActS)(const double);
       virtual int Save(FILE *fhd);
       virtual int Load(FILE *fhd);
//protected:
       NeuralNetBase         *NN;                  // the nn they belong to
       double                       Bias,                 // Bias
                                    Out,                  // Output
                                    Net,                  // Input from net
                                    Delta;                // delta - error
backpropagated throught net
       int                          iNLayer;              // position of this neuron -
Layer [0;n]
       long                         lNNeuron;             // posision of this neuron -
neuron[1;n] on Layer iNLayer
       Liste <LinkBase>      LptrNf,               // connections from
                                    LptrNt;               // connections to
};

#endif//__NEURONBASEDEF_H
```

NeuronBase.h

```
#ifndef __NEURONBASE_H
#define __NEURONBASE_H

//#include "../Liste.h"
#include "WeightBase.h"
#include "LinkBase.h"
#include "NeuronAkt.h"

NeuronBase :: NeuronBase()
{
	Bias   = .0;
	Out            = .0;
	Net            = .0;

	fAct = fNActtanh;
	fActS= fNActtanhS;
}

NeuronBase :: ~NeuronBase()
{
	LptrNf.DelAllData();
	LptrNt.DelAllData();
}

const NeuronBase &NeuronBase :: operator = (const NeuronBase &NParam)
{
	NN = NParam.NN;
	Bias = NParam.Bias;
	Out = NParam.Out;
	iNLayer = NParam.iNLayer;
	lNNeuron = NParam.lNNeuron;
	LptrNf = NParam.LptrNf;
	LptrNt = NParam.LptrNt;

	return ((const NeuronBase &)(*this));
}

/*int NeuronBase :: ConnectNeuronTo(NeuronBase *NParam)
{
	LinkBase       Lit,
			Lif;

	Lit.Nptr = NParam;
	Lif.Nptr = this;

	LptrNt.AddElem(Lit);
	NParam->LptrNf.AddElem(Lif);

	return 1;
}*/

int NeuronBase :: Save(FILE *fhd)
{
	fwrite(&Bias,sizeof(double),1,fhd);
	return 1;
}

int NeuronBase :: Load(FILE *fhd)
{
	fread(&Bias,sizeof(double),1,fhd);
	return 1;
}

#endif//__NEURONBASE_H
```

WeightBaseDef.h

```
#ifndef __WEIGHTBASEDEF_H
#define __WEIGHTBASEDEF_H

//WeightBase.h written by Johannes Lampel / LampelSoft '99
//written : 18. Nov 1999

class  DllExport WeightBase
{
public:
       WeightBase();
       virtual ~WeightBase();
       virtual const WeightBase &operator = (const WeightBase &);
       //protected:
       double dWeight;
       long luse;
       // it's not needed, but Liste required it
       int Save(FILE *fhd);
       int Load(FILE *fhd);
};

#endif//__WEIGHTBASEDEF_H
```

WeightBase.h

```
#ifndef __WeightBase_H
#define __WeightBase_H

#include <stdio.h>
//#include "Liste.h"

WeightBase :: WeightBase()
{
	dWeight = 0;
	luse = 0;
}

WeightBase :: ~WeightBase()
{
}

const WeightBase &WeightBase :: operator = (const WeightBase &WParam)
{
	dWeight = WParam.dWeight;
	luse = WParam.luse;

	return ((const WeightBase &)(*this));
}

int WeightBase :: Save(FILE *fhd){
	printf("WeightBase :: Save(FILE *fhd) isn't impl");
	return 1;
}

int WeightBase :: Load(FILE *fhd){
	printf("WeightBase :: Load(FILE *fhd) isn't impl");
	return 1;
}

#endif//__WeightBase_H
```

B.2 Quelltext des SOM-Simulators (komplett)

CqSOM2d.h

```
#ifndef __CQSOM2D_H
#define __CQSOM2D_H

/**************************************************************************
 *                                                                        *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   any later version.                                                    *
 *                                                                        *
 **************************************************************************/

// class for 2d SOMs
// by Johannes Lampel / LampelSoft
// (c) Johannes Lampel / LampelSoft
// written Sept 13th 2000

#include "GlobalCqSOMDefs.h"
#include "nVec.h"
#include "nSOMErr.h"
#include "fNBorKernel.h"

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <iostream.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <float.h>

#define _SOM_ID 0x20000927                                  // ID for som - files

class DllExport qSOM2d{
public:
       qSOM2d();                                                                  // ctor
       ~qSOM2d();                                                                 // dtor

       int SetSize(long,long,long);                         // set size of map (Dim of
Input,x,y)     -> x=y isn't too useful
       int InitWeights(InitType);                                  // ...
       int SetInput(double *);                                        // copies input into
local class storage
       double Categorize(void);                                       // cats input
       double Categorize(double*);                                 // like the two above
       double CategorizeFast(void);                                // cats faster cause it
neglects some (!) neurons
       double CategorizeFast(double*);                             // ...
       void GetWinner(long &,long &);                              // finds 'nearest' neuron
       double *GetNWeights(long,long);                             // return the weight matrix
of one neuron
       double GetDistP(void);                                      // return stiffness param
       void SetDistP(double);                                      // ..
       double GetLRate(void);                                      // return lRate
       void SetLRate(double);                                      // ..
       void SetNKFunction(double (*name)(const double&,const double&));  // sets the pointer
to the NK functions, which is set to fNKgauss1 by default
       void *GetNKFunction(void);                                  // ..
       void Save(const char *);                                    // savin' data to file
       void Load(const char *);                                    // loadin' data from file to
mem
protected:
       long lXSize,lYSize,lNNeurons;
       long lRow;                                                                 // lXSize *
lNNeurons     (just performance)
       double *dp3Weights;                                                 // field of weights
       double *dp1Input;                                                   // input to be
categorized or tested
       double dLearnrate;                                                  // ...
```

```
        double dDistanceP;                                              // stiffness param
        char *szVersion;                                                // Just some
information about this simulator
        double (*fNK)(const double&,const double&); // pointer to NK function
        nVec    *pVDiff;                                                // diff (updated by
calling cat) between Input and Winner
private:
};

#endif  __CQSOM2D_H
```

CqSOM2d.cpp

```
#include "CqSOM2d.h"

qSOM2d :: qSOM2d(){
	char szTemp[] = "LampelSoft SOM Simulator 0.2 by Johannes Lampel\n";
	szVersion = new char[strlen(szTemp)+1];
	strcpy(szVersion,szTemp);

	dp3Weights     = 0;
	dp1Input       = 0;
	pVDiff         = 0;
	dLearnrate     = 0.0;
	dDistanceP     = 0.0;

	// init pointer to Neightborhoodkernel function
	fNK = fNKgauss1;

	// INIT RAND
	srand(time(0));
}

qSOM2d :: ~qSOM2d(){
	if(dp3Weights) delete [] dp3Weights;
	if(dp1Input)   delete [] dp1Input;
	if(szVersion)  delete [] szVersion;
	if(pVDiff)             delete pVDiff;

	dp3Weights     = 0;
	dp1Input       = 0;
	szVersion      = 0;
	pVDiff         = 0;
}

int qSOM2d :: SetSize(long lINeurons,long lpXSize,long lpYSize){
	lNNeurons      = lINeurons;
	lXSize         = lpXSize;
	lYSize         = lpYSize;
	lRow           = lXSize * lNNeurons;

	// alloc mem for weights - 3dim
	// usage : weigth := weights[ly * lXSize * lNNeurons + lx * lNNeurons + lN]
	// lXSize * lNNeurons := lRow

	if(dp3Weights) delete [] dp3Weights;
	dp3Weights = new double[lNNeurons * lXSize * lYSize];

	// alloc mem for input neurons

	if(dp1Input) delete [] dp1Input;
	dp1Input = new double [lNNeurons];

	// Alloc space for VDiff - Diff between input and winner vector

	if(pVDiff) delete pVDiff;
	pVDiff = new nVec(lNNeurons);

	return 1;
}

int qSOM2d :: InitWeights(InitType IType){
	long lschl;
	switch (IType){
	case Init_Random:
		for(lschl=0;lschl < lNNeurons * lXSize * lYSize;lschl++){
			dp3Weights[lschl] = double(rand()) / RAND_MAX;
		}
		return 0;
	default:
		cerr << "\"What's that\" - InitWeights"<<endl;
		return 1;
	}
}

int qSOM2d :: SetInput(double *dptr){
	memcpy(dp1Input,dptr,sizeof(double) * lNNeurons);
```

```
	return 0;
}

void qSOM2d :: GetWinner(long &lxMin,long &lyMin){		// GetNearest Neuron to
Input
	try{
		if(dp1Input){
			// set vars to 0
			// this has to be done, 'cause sometimes ( or only with optimized
compilin' with MS C++6.0 he cannot find the nearest Neuron !!!??) this function isn't able to
find minima and would otherwise cause a page fault
			lyMin = lxMin = 0;

			long lxschl,lyschl;
			double dMin = DBL_MAX;// dist of best neuron
			double dLength;
			long lsP;

			nVec VInput (lNNeurons);
			nVec VComp     (lNNeurons);
			nVec VDiff     (lNNeurons);

			VInput.SetMemVec(dp1Input);		// don't copy data - only
set pointer

			for(lxschl = 0;lxschl < lXSize;lxschl++){
				lsP = lxschl * lNNeurons;
				for(lyschl = 0;lyschl < lYSize;lyschl++){
					VComp.GetVec(&dp3Weights[lyschl * lRow + lsP]);
					VComp -= VInput;
					dLength = VComp.SQuadr();

					if(dLength < dMin){
						dMin = dLength;
						lxMin = lxschl;
						lyMin = lyschl;
					}
				}
			}
		}
		else{
			throw NoInput();
		}
	}
	catch(NoInput NI){
		cerr << "No Input specified" << endl;
		throw NI;
	}
}

double qSOM2d :: Categorize(double *dpParam){
	SetInput(dpParam);
	return Categorize();
}

double qSOM2d :: Categorize(void){		// learning function
	long lxWinner,lyWinner;
	long lxschl,lyschl;
	double ddistance;
	long lDx,lDy;
	nVec   VIn(lNNeurons);			// input vector
	nVec   VNeuron(lNNeurons);		// vector of current neuron
	nVec   VUpdated(lNNeurons);  // updated vector
	nVec   VTemp(lNNeurons);		// temp
	double *dpCVec;

	GetWinner(lxWinner,lyWinner);		// getting
winner neuron
	VIn.GetVec(dp1Input);		//
get input to vector

	pVDiff->GetVec(GetNWeights(lxWinner,lyWinner));
	*pVDiff -= VIn;

	for(lxschl=0;lxschl < lXSize;lxschl++){
		for(lyschl=0;lyschl < lYSize;lyschl++){
```

```
                        // get current weights

                        dpCVec = &dp3Weights[lyschl * lRow + lxschl * lNNeurons];
                        // get distance to winner
                        lDx = lxschl - lxWinner;
                        lDy = lyschl - lyWinner;
                        ddistance = sqrt(lDx * lDx + lDy * lDy);
                        // get weights from current neuron
                        VNeuron.SetMemVec(dpCVec);
                        // update weights

                        VTemp = VIn;
                        VTemp -= VNeuron;
                        VTemp *= (dLearnrate * fNK(ddistance,dDistanceP));
                        VNeuron += VTemp;
                        // don't have to write updated vector back to mem cause it was calced in
mem (VNeuron.SetMemVec)
                }
        }
        return pVDiff->Length();
}

double qSOM2d :: CategorizeFast(double *dpParam){
        SetInput(dpParam);
        return CategorizeFast();
}

double qSOM2d :: CategorizeFast(void){
/* fast learning function -> not suitable for gauss1 + 2  -> only for cone,cyl and cos because
for the gauss' the result of them isn't 0 when z > d
This function neglects the neurons outside a square defined by the winner neuron as middle and
the min radius by the 'dDistanceP' The distance or stiffness param
*/

        long lxWinner,lyWinner;
        long lxschl,lyschl;
        double ddistance;
        long lDx,lDy;
        nVec    VIn(lNNeurons);                     // input vector
        nVec    VNeuron(lNNeurons);           // vector of current neuron
        nVec    VUpdated(lNNeurons);  // updated vector
        nVec    VTemp(lNNeurons);             // temp
        double  *dpCVec;
        long lxstart,lxend,lystart,lyend;

        GetWinner(lxWinner,lyWinner);                                                   // getting
winner neuron
        VIn.GetVec(dp1Input);                                                                   //
get input to vector

        pVDiff->GetVec(GetNWeights(lxWinner,lyWinner));
        *pVDiff -= VIn;

        // calc vars for only changing weights in neurons which are in the radius of dDistanceP
        if(dDistanceP < 1.0){                                   // no other neurons are affected
                lxstart = lxWinner;
                lxend = lxstart + 1;
                lystart = lyWinner;
                lyend = lystart + 1;
        }
        else{                                                                   // otherwise test
...
                lxstart = long(lxWinner - dDistanceP - 1);
                lxend  = long(lxWinner + dDistanceP + 1);
                lystart = long(lyWinner - dDistanceP - 1);
                lyend  = long(lyWinner + dDistanceP + 1);
                if(lxstart < 1){                                        // if outside of the borders
                        lxstart = 0;
                }
                if(lystart < 1){
                        lystart = 0;
                }
                if(lxend > lXSize){
                        lxend = lXSize;
                }
                if(lyend > lYSize){
```

```
                        lyend = lYSize;
                }
        }

        for(lxschl=lxstart;lxschl < lxend;lxschl++){
                for(lyschl=lystart;lyschl < lyend;lyschl++){
                        // get current weights
                        dpCVec = &dp3Weights[lyschl * lRow + lxschl * lNNeurons];
                        // get distance to winner
                        lDx = lxschl - lxWinner;
                        lDy = lyschl - lyWinner;
                        ddistance = sqrt(lDx * lDx + lDy * lDy);
                        // get weights from current neuron
                        VNeuron.SetMemVec(dpCVec);
                        // update weights
                        VTemp = VIn;
                        VTemp -= VNeuron;
                        VTemp *= (dLearnrate * fNK(ddistance,dDistanceP));
                        VNeuron += VTemp;
                        // don't have to write updated vector back to mem cause it was calced in
mem (VNeuron.SetMemVec)
                }
        }
        return pVDiff->Length();
}

double *qSOM2d :: GetNWeights(long lx,long ly){
        try{
                if(lx < 0 ||ly < 0)
                        throw FuckinParam();
                return &dp3Weights[ly * lRow + lx * lNNeurons];
        }
        catch(FuckinParam FP){
                cerr << "Got some fuckin' params in qSOM2d :: GetNWeights"<<endl;
                throw FP;
        }
}

double qSOM2d :: GetDistP(void){
        return dDistanceP;
}

void qSOM2d :: SetDistP(double dParam){
        dDistanceP = dParam;
}

double qSOM2d :: GetLRate(void){
        return dLearnrate;
}

void qSOM2d :: SetLRate(double dParam){
        dLearnrate = dParam;
}

void qSOM2d :: Save(const char *szFileName){
        FILE *fhd;
        long lCoding = _SOM_ID;
        try{
                if((fhd = fopen(szFileName,"wb"))){
                        fwrite(&lCoding,sizeof(long),1,fhd);               // writing id for
somfiles
                        fwrite(&lXSize,sizeof(long),1,fhd);
                        fwrite(&lYSize,sizeof(long),1,fhd);
                        fwrite(&lNNeurons,sizeof(long),1,fhd);
                        fwrite(&lRow,sizeof(long),1,fhd);
                        fwrite(&dLearnrate,sizeof(double),1,fhd);
                        fwrite(&dDistanceP,sizeof(double),1,fhd);
                        fwrite(dp3Weights,sizeof(double),lXSize * lYSize * lNNeurons,fhd);
                        fclose(fhd);
                }
                else
                {
                        throw FileSaveError();
                }
        }
        catch(FileSaveError FSE){
```

```
                cerr << "Error opening file : " << szFileName << endl;
                throw FSE;
        }
}


void qSOM2d :: Load (const char *szFileName){
        FILE *fhd;
        long lCoding;
        try{
                if((fhd=fopen(szFileName,"rb"))){
                        if(dp3Weights) delete [] dp3Weights;
                        if(dp1Input)   delete [] dp1Input;

                        fread(&lCoding,sizeof(long),1,fhd);                                     //
read coding and verify
                        if(lCoding != _SOM_ID)
                                throw UnknownFileFormat();
                        fread(&lXSize,sizeof(long),1,fhd);                                      //
just as save - vice versa
                        fread(&lYSize,sizeof(long),1,fhd);
                        fread(&lNNeurons,sizeof(long),1,fhd);
                        fread(&lRow,sizeof(long),1,fhd);

                        fread(&dLearnrate,sizeof(double),1,fhd);
                        fread(&dDistanceP,sizeof(double),1,fhd);

                        dp3Weights     = new double [lXSize * lYSize * lNNeurons];
                        dp1Input       = new double [lNNeurons];

                        fread(dp3Weights,sizeof(double),lXSize * lYSize * lNNeurons,fhd);
                        memset(dp1Input,0,sizeof(double) * lNNeurons);
                        fclose(fhd);
                }
                else
                {
                        throw FileLoadError();
                }
        }
        catch(FileLoadError FLE){
                cerr << "Error opening file : " << szFileName << endl;
                throw FLE;
        }
        catch(UnknownFileFormat UFF){
                cerr << "Unknown file format" << endl;
                throw UFF;
        }
}

void qSOM2d :: SetNKFunction(double (*fNKParam)(const double&,const double&)){
        fNK = fNKParam;
}

void *qSOM2d :: GetNKFunction(void){
        return ((void *)(fNK));
}
```

fNBorKernel.h

```
#ifndef __fNBorKernel_H
#define __fNBorKernel_H

double fNKcos         (const double&,const double&);
double fNKgauss1      (const double&,const double&);
double fNKgauss2      (const double&,const double&);
double fNKcyl         (const double&,const double&);
double fNKcone        (const double&,const double&);

#endif  __fNBorKernel_H
```

fNBorKernel.cpp

```
// Neighbourhood kernel functions for SOMs
// by Johannes Lampel / LampelSoft
// (c) Johannes Lampel / LampelSoft
// written Sept 13th 2000

// z -> Distance of neurons
// d -> size of neighbourhood kernel
#include <math.h>
#include "GlobalCqSOMDefs.h"

#define pi 3.14159265358979

double DllExport fNKcos(const double &z,const double &d){
	if(z < d)
		return (cos( z/d * pi/2.0));
	else
		return 0;
}

double DllExport fNKgauss1(const double &z,const double &d){
	return exp(-((z*z)/(d*d)));
	// don't use pow(?,2),'cause it's to time expensive
}

double DllExport fNKgauss2(const double &z,const double &d){
	return (1-((z*z)/(d*d)))*fNKgauss1(z,d);
}

double DllExport fNKcyl(const double &z,const double &d){
	if(z < d)
		return 1.0;
	else
		return 0.0;
}

double DllExport fNKcone(const double &z,const double &d){
	if(z < d)
		return 1.0-(z/d);
	else
		return 0.0;
}
```

GlobalCqSOMDefs.h

```
#ifndef __GlobalCqSOMDefs_H
#define __GlobalCqSOMDefs_H

// written Sept 13th 2000

typedef long InitType;
typedef long dim_c;

enum eInitTypes{
        Init_Random,
        Init_Regular
};

// def for easy use as dll

#ifdef __DLL
#define DllExport __declspec( dllexport )
#else
#define DllExport
#endif

#endif
```

nSOMErr.h

```
#ifndef __SOMERR_H
#define __SOMERR_H

// written Sept 13th 2000

class NoInput{};                    // In qSOM2d :: Categorize : No input has been specified
yet
class FileSaveError{};        // Error while saving
class FileLoadError{};        // Error while loading
class UnknownFileFormat{};    // ID of SOM file hasn't been right
class FuckinParam{};          // a function was called with irr params

#endif
```

nVec.h

```
#ifndef __NVEC_H
#define __NVEC_H

/*************************************************************************
 *                                                                       *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   any later version.                                                  *
 *                                                                       *
 *************************************************************************/

// n-dim vectorclass
// by Johannes Lampel / LampelSoft
// (c)  copyright Johannes Lampel
// written Sept 13th 2000

#include "GlobalCqSOMDefs.h"
#include "nVec.h"
#include "nVecErr.h"
#include <stdio.h>
#include <math.h>
#include <memory.h>
#include <iostream.h>

enum nVecTypes{
       nVT_copy=0,
       nVT_mem
};

class nVec;

nVec operator *(double,nVec&);

class DllExport nVec{
       friend nVec operator *(double,nVec&);
       friend class SOMPattern;
public:
       nVec();
       nVec(const dim_c);
       nVec(const dim_c,const double * const);
       nVec(const nVec &);
       virtual ~nVec();

       double  Length(void);                                          // ...
       int            Normalize(void);                                       // ...
       int            GetVec(double *);                                      // copy data
from field into this class
       int            SetMemVec(double *);                                   // set ddpDim to
Param and iType to nVT_mem - be careful, you havn't a copy of this data, your changing the
original data
       int            ConvMemtoCopy(void);                                   // convert a
Mem - typed instance of this class to a instance, which contains a copy of this data items -
alloc mem and copy
       int            SetVec(double *);                                      // copy data
from class to field
       double  SQuadr(void);                                          // return sum of all
quadrs of comps
       int            SetNumDim(const dim_c);

       int            Load(FILE *);
       int            Save(FILE *);

       virtual const nVec & operator = (const nVec &);     // ...
       virtual const nVec & operator +=(const nVec &);     // ...
       virtual const nVec & operator -=(const nVec &);     // ...
       virtual const nVec & operator *=(const double);     // ...
       virtual const nVec & operator /=(const double);     // ...

       virtual nVec          operator +(const nVec &);     // ...
       virtual nVec          operator -(const nVec &);     // ...
       virtual nVec          operator *(const double);     // ...
       virtual nVec          operator /(const double);     // ...

       virtual double &      operator [](long);                  // return value of
[dimension] component
```

```
protected:
        double *ppdDim;                                                 //
data items
        long lCDim;                                                     //
number of dimensions
        int iType;                                                      //
copy data or set pointer to data
private:
};

#endif  __NVEC_H
```

nVec.cpp

```
#include "nVec.h"

nVec :: nVec(){
        lCDim = 0;
        iType = -1;
        ppdDim = 0;
}

nVec :: nVec(const dim_c dcDim){
        ppdDim = new double[dcDim];
        lCDim = dcDim;
        iType = nVT_copy;                                                    // copy data by
default
}

nVec :: nVec(const dim_c dcDim,const double * const dptr){
        ppdDim = new double[dcDim];
        lCDim = dcDim;
        memcpy(ppdDim,dptr,sizeof(double) * lCDim);
        iType = nVT_copy;                                                    // copy data by
default
}

nVec :: nVec(const nVec &npVec){
        ppdDim = 0;
        this->operator = (npVec);
        iType = nVT_copy;                                                    // copy data by
default
}

nVec :: ~nVec(){
        if(!iType && ppdDim) delete [] ppdDim;
        lCDim = 0;
        ppdDim = 0;
}

double nVec :: Length(void){
        double dQSum = 0,dTemp;
        // sum of quadr
        long lschl;
        for(lschl = 0;lschl < lCDim;lschl ++){
                dTemp = ppdDim[lschl];
                dQSum += dTemp * dTemp;
        }
        return sqrt(dQSum);
}

int nVec :: Normalize(void){
        double dLength = Length();
        if(dLength==0.0){
                dLength=1.0;
        }
        long lschl;
        for(lschl=0;lschl < lCDim;lschl++){
                ppdDim[lschl] /= dLength;
        }
        return 1;
}

const nVec & nVec :: operator = (const nVec & NParam){
        try{
                if(!iType){
                        if(ppdDim) delete []ppdDim;
                        ppdDim = new double[NParam.lCDim];
                        lCDim = NParam.lCDim;
                        memcpy(ppdDim,NParam.ppdDim,sizeof(double)*lCDim);
                }
                else{
                        if(lCDim != NParam.lCDim){
                                throw IncompatibleVectors();
                        }
                        memcpy(ppdDim,NParam.ppdDim,sizeof(double)*lCDim);
                }
                return ((const nVec &)(*this));
        }
        catch(IncompatibleVectors IV){
```

```
		cerr << " 'IncompatibleVectors' thrown by = " << lCDim << " + " << NParam.lCDim
<<" : "<< &NParam <<endl;
		throw (IV);
	}
}

const nVec & nVec :: operator +=(const nVec & NParam){
	try{
		if(lCDim != NParam.lCDim){
			throw IncompatibleVectors();
		}
		long lschl;
		for(lschl=0;lschl<lCDim;lschl++){
			ppdDim[lschl] += NParam.ppdDim[lschl];
		}
	}
	catch(IncompatibleVectors IV){
		cerr << " 'IncompatibleVectors' thrown by += " << lCDim << " + " <<
NParam.lCDim <<" : "<< &NParam <<endl;
		throw(IV);
	}
	return ((const nVec &)(*this));
}

const nVec & nVec :: operator -=(const nVec & NParam){
	try{
		if(lCDim != NParam.lCDim){
			throw IncompatibleVectors();
		}
		long lschl;
		for(lschl=0;lschl < lCDim;lschl++){
			ppdDim[lschl] -= NParam.ppdDim[lschl];
		}
	}
	catch(IncompatibleVectors IV){
		cerr << " 'IncompatibleVectors' thrown by -= " << lCDim << " + " <<
NParam.lCDim <<" : "<< &NParam <<endl;
		throw(IV);
	}
	return ((const nVec &)(*this));
}

const nVec & nVec :: operator *=(const double dParam){
	long lschl;
	for(lschl=0;lschl<lCDim;lschl++){
		ppdDim[lschl] *= dParam;
	}
	return ((const nVec &)(*this));
}

const nVec & nVec :: operator /=(const double dParam){
	long lschl;
	for(lschl=0;lschl<lCDim;lschl++){
		ppdDim[lschl] /= dParam;
	}
	return ((const nVec &)(*this));
}

nVec nVec :: operator +(const nVec &NParam){
	try{
		if(lCDim != NParam.lCDim){
			throw IncompatibleVectors();
		}
			nVec TmpVec(lCDim);
			TmpVec = *this;
			TmpVec+=NParam;
			return TmpVec;
	}
	catch(IncompatibleVectors IV){
		cerr << " 'IncompatibleVectors' thrown by + " << lCDim << " + " << NParam.lCDim
<<" : "<< &NParam <<endl;
		throw(IV);
	}
	return 0;			// error if this is executed
}

nVec nVec :: operator -(const nVec &NParam){
	try{
```

```
                if(lCDim != NParam.lCDim){
                        throw IncompatibleVectors();
                }
                nVec TmpVec(lCDim);
                TmpVec = *this;
                TmpVec-=NParam;
                return TmpVec;
        }
        catch(IncompatibleVectors IV){
                cerr << " 'IncompatibleVectors' thrown by - " << lCDim << " + " << NParam.lCDim
<< " : "<< &NParam <<endl;
                throw(IV);
        }
        return 0;                       // error if this is executed
}

nVec nVec :: operator *(const double dParam){
        nVec TmpVec(lCDim);
        TmpVec = *this;
        TmpVec*=dParam;
        return TmpVec;
}

nVec nVec :: operator /(const double dParam){
        nVec TmpVec(lCDim);
        TmpVec = *this;
        TmpVec/=dParam;
        return TmpVec;
}

double & nVec :: operator [] (long lIndex){
        return ((double &)(ppdDim[lIndex]));
}

double nVec :: SQuadr(void){
        double dtemp=0;
        double dx;
        long lschl;
        for(lschl=0;lschl < lCDim;lschl++){
                dx = ppdDim[lschl];
                dtemp += (dx * dx);
        }
        return dtemp;
}

int nVec :: GetVec(double *dptr){
        /*long lschl;
        for(lschl=0;lschl < lCDim;lschl ++){
                ppdDim[lschl] = dptr[lschl];
        }*/
        memcpy(ppdDim,dptr,sizeof(double) * lCDim);
        return 1;
}

int nVec :: SetVec(double *dpDest){
        long lschl;
        for(lschl=0;lschl < lCDim;lschl++){
                dpDest[lschl] = ppdDim[lschl];
        }
        return 1;
}

int nVec :: SetMemVec(double *dpParam){
        // setting ppdDim to mempos and iType to nVT_mem (case del of old alloc mem)
        if(!iType){
                if(ppdDim) delete [] ppdDim;
        }
        ppdDim = dpParam;
        iType = nVT_mem;
        return 1;
}

int nVec :: ConvMemtoCopy(void){
        double *dpTmp;
        dpTmp = new double[lCDim];
        memcpy(dpTmp,ppdDim,sizeof(double) * lCDim);
        ppdDim = dpTmp;
        iType = nVT_copy;
```

```
        return 1;
}

int nVec :: SetNumDim(const dim_c dcDim){
        if(!iType && !ppdDim) delete [] ppdDim;
        lCDim = dcDim;
        ppdDim = new double[dcDim];
        return 1;
}

int nVec :: Save(FILE *fhd){
        return fwrite(ppdDim,sizeof(double),lCDim,fhd);
}

int nVec :: Load(FILE *fhd){
        if(ppdDim) return fread(ppdDim,sizeof(double),lCDim,fhd);
        return false;
}

// global functions

nVec operator *(double dParam,nVec &nParam){
        return (nParam.operator *(dParam));
}
```

nVecErr.h

```
#ifndef __NVECERR_H
#define __NVECERR_H

// written Sept 13th 2000

class IncompatibleVectors{};         // Two vectors have different number of dimensions
class NoPointInit{};                 // Pointer to data hasn't been inited - todo

#endif  __NVECERR_H
```

SOM.h

```
#ifndef __SOM_H
#define __SOM_H

#include "CqSOM2d.h"
#include "SOMPattern.h"

#endif  __SOM_H
```

SOMPattern.h

```
#ifndef __SOMPATTERN_H
#define __SOMPATTERN_H

// SOMPattern.h by Johannes Lampel /LampelSoft
// (c) Johannes Lampel / LampelSoft
// written Sep 27th 2000

/************************************************************************
 *                                                                      *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   any later version.                                                 *
 *                                                                      *
 ************************************************************************/

#include "GlobalCqSOMDefs.h"
#include "nVec.h"
#include "SOMPatternElem.h"
#include "CqSOM2d.h"
#include "SOMPatternErr.h"

class DllExport SOMPattern{
public:
       SOMPattern(const dim_c);                                          // ctor
       ~SOMPattern();                                                           //
dtor
       int SetMaxPatternNum(const long);                                 // max Number of
Pattern - not yet able to copy data from old array to new and no dynamic alloc
       long AddPattern(const double *);                                  // ...
       long AddPattern(const nVec &);                                    // ... (->
calling AddPattern(const double *)
       long GetNumP(void);
       int SetSOM(qSOM2d *);                                             // setting
som to be trained
       int TeachEpoch(const double&,const double&);       // Teaching a full number of
pattern and storing the errors ( return values of qSOM2d :: Categorize ) in a pattern related
variable
       int TeachEpochFast(const double &,const double &);  // same as above, only using
qSOM2d::CategorizeFast

       int Load(const char *);
       int Load(FILE *fhd);
       int Save(const char *);
       int Save(FILE *fhd);
protected:
       SOMPatternElem *pFPattern;                                        // Field of
PatternElem instances with static size
       long lPDimNum;                                                           //
Number of dimensions of every Pattern
       long lPNum;
       // Number of pattern
       long lActPPos;                                                           //
Actual position of last elem ( lActPPos - 1 )

       qSOM2d * SActSOM;                                                        //
som to be trained
private:
};

#endif  __SOMPATTERN_H
```

SOMPattern.cpp

```
#include "SOMPattern.h"

SOMPattern :: SOMPattern (const dim_c dcDim){
	pFPattern = 0;
	lPDimNum = dcDim;
	lActPPos = 0;
}

SOMPattern :: ~SOMPattern (){
	if(pFPattern) delete [] pFPattern;
}

int SOMPattern :: SetMaxPatternNum(const long lNum){
	long lschl;
	lPNum = lNum;

	if(pFPattern) delete [] pFPattern;

	pFPattern = new SOMPatternElem[lNum];
	for(lschl=0;lschl < lNum;lschl++){
		pFPattern[lschl].SetNumDim(lPDimNum);
	}
	return 1;
}

long SOMPattern :: AddPattern(const nVec &VParam){
	try{
		if(VParam.lCDim != lPDimNum)
			throw IncompatibleVectors();

		return AddPattern(VParam.ppdDim);
	}
	catch(IncompatibleVectors IV){
		cerr << "IncompatibleVectors thrown in SOMPattern :: AddPattern" << endl;
		throw IV;
	}
}

long SOMPattern :: AddPattern(const double *dpFParam){
	try{
		if(!pFPattern)
			throw SetMaxPatternwasntcalled();

		if(lActPPos >= lPNum)
			return -1;

		pFPattern[lActPPos].iType = nVT_copy;
		pFPattern[lActPPos].lCDim = lPDimNum;
		memcpy(pFPattern[lActPPos].ppdDim,dpFParam,sizeof(double) * lPDimNum);
		pFPattern[lActPPos].dLastError = 0.0;

		lActPPos ++;

		return lActPPos;
	}
	catch(SetMaxPatternwasntcalled SM){
		cout << "SetMaxPatternwasntcalled was thrown by SOMPattern :: AddPattern" <<
endl;
		throw SM;
	}
}

int SOMPattern :: SetSOM (qSOM2d *pSOM){
	SActSOM = pSOM;
	return 1;
}

int SOMPattern :: TeachEpoch(const double &dLearn,const double &dDist){
	long lschl;
	SActSOM -> SetDistP(dDist);
	SActSOM -> SetLRate(dLearn);
	for(lschl=0;lschl < lActPPos;lschl ++){
		pFPattern[lschl].dLastError = SActSOM->Categorize(pFPattern[lschl].ppdDim);
	}
	return 1;
}
```

```
int SOMPattern :: TeachEpochFast(const double &dLearn,const double &dDist){
	long lschl;
	SActSOM -> SetDistP(dDist);
	SActSOM -> SetLRate(dLearn);
	for(lschl=0;lschl < lActPPos;lschl ++){
		pFPattern[lschl].dLastError = SActSOM->CategorizeFast(pFPattern[lschl].ppdDim);
	}
	return 1;
}

int SOMPattern :: Load (const char *szName){
	FILE *fhd;
	if(fhd = fopen(szName,"rb")){
		Load(fhd);
		fclose(fhd);
		return true;
	}
	else{
		// failed
		cerr << "couldn't open file in SOMPattern::load" << endl;
		fclose(fhd);
		return false;
	}
}

int SOMPattern :: Save (const char *szName){
	FILE *fhd;
	if(fhd = fopen(szName,"wb")){
		Save(fhd);
		fclose(fhd);
		return true;
	}
	else{
		// failed
		cerr << "couldn't open file in SOMPattern::Save" << endl;
		fclose(fhd);
		return false;
	}
}

int SOMPattern :: Save(FILE *fhd){
	long lschl;

	fwrite(&lActPPos,sizeof(long),1,fhd);
	fwrite(&lPDimNum,sizeof(long),1,fhd);

	for(lschl=0;lschl < lActPPos;lschl++){
		pFPattern[lschl].Save(fhd);
	}
	return true;
}

int SOMPattern :: Load (FILE *fhd){
	long lschl;

	if(pFPattern) delete [] pFPattern;

	fread(&lPNum,sizeof(long),1,fhd);
	lActPPos = lPNum;
	pFPattern = new SOMPatternElem[lPNum];
	fread(&lPDimNum,sizeof(long),1,fhd);
	for(lschl=0;lschl < lPNum;lschl++){
		pFPattern[lschl].SetNumDim(lPDimNum);
		pFPattern[lschl].Load(fhd);
	}
	return true;
}

long SOMPattern :: GetNumP(void){
	return lActPPos;
}
```

SOMPatternElem.h

```
#ifndef __SOMPATTERNELEM_H
#define __SOMPATTERNELEM_H

// SOMPatternElem.h by Johannes Lampel / LampelSoft
// (c) Johannes Lampel / LampelSoft
// written Sep 27th 2000

/*************************************************************************
 *                                                                       *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   any later version.                                                  *
 *                                                                       *
 *************************************************************************/

#include "nVec.h"

class DllExport SOMPatternElem:protected nVec{
       friend class SOMPattern;
public:
       SOMPatternElem();                    // ctor
       ~SOMPatternElem();                   // dtor
protected:
       double dLastError;                   // just the last error, which has been calced last
m::cat
private:
};

#endif  __SOMPATTERNELEM_H
```

SOMPatternElem.cpp

```
#include "SOMPatternElem.h"

SOMPatternElem :: SOMPatternElem(){
       dLastError = 0.0;
}

SOMPatternElem :: ~SOMPatternElem(){
}
```

SOMPatternErr.h

```
#ifndef __SOMPATTERNERR_H
#define __SOMPATTERNERR_H

class SetMaxPatternwasntcalled{};

#endif
```

B.3 Relevante Teile des Bot Quelltextes

CBotBase.h

```
#ifndef __CBOTBASE_H
#define __CBOTBASE_H

#include "bot_globaldefs.h"

#include "extdll.h"
#include "util.h"
#include "cbase.h"
#include "weaponinfo.h"
#include "limits.h"

#include "waypoint.h"

#include "neuralnetdef.h"
#include "som.h"
#include "NNWeapon.h"
#include "CRadio.h"
#include "CSkill.h"
#include "CSTMem.h"
#include "CLTMem.h"
#include "CState.h"
#include "CTask.h"
#include "CPersonality.h"
#include "bot_names.h"
#include "bot_wpdir.h"
#include "CSDecals.h"

#include "globalvars.h"

// stuff for Win32 vs. Linux builds

#ifndef __linux__

typedef int (FAR *GETENTITYAPI)(DLL_FUNCTIONS *, int);
typedef int (FAR *GETNEWDLLFUNCTIONS)(NEW_DLL_FUNCTIONS *, int *);
typedef void (DLLEXPORT *GIVEFNPTRSTODLL)(enginefuncs_t *, globalvars_t *);
typedef void (FAR *LINK_ENTITY_FUNC)(entvars_t *);

#else

#include <dlfcn.h>
#define GetProcAddress dlsym

typedef int BOOL;

typedef int (*GETENTITYAPI)(DLL_FUNCTIONS *, int);
typedef int (*GETNEWDLLFUNCTIONS)(NEW_DLL_FUNCTIONS *, int *);
typedef void (*GIVEFNPTRSTODLL)(enginefuncs_t *, globalvars_t *);
typedef void (*LINK_ENTITY_FUNC)(entvars_t *);

#endif

edict_t *CREATE_FAKE_CLIENT( const char *netname );
char *GET_INFOBUFFER( edict_t *e );
char *GET_INFO_KEY_VALUE( char *infobuffer, char *key );
void SET_CLIENT_KEY_VALUE( int clientIndex, char *infobuffer,char *key, char *value );

void FakeClientCommand(edict_t *pBot, char *arg1, char *arg2, char *arg3);

class FEItem{
public:
        FEItem();
        ~FEItem();

        edict_t *pEdict;

        float fLastSeen;             // last seen time -> time when bot reacted on enemy
        float fRecogn;               // time when bot was visible, but that doesn't mean that
bot recognizes it as an enemy
        float fTime2seeit;           // time it takes to recognize the enemy
        bool bBehindSG;
};

class GrenadeOrder{
```

```
public :
	GrenadeOrder();
	~GrenadeOrder();
	void Attack(Vector &, long);

	long lState;
	long lTypeoG;
	Vector VAim;
};

class CBotBase{
public:
	// public functions
	CBotBase();
	~CBotBase();

	virtual void Think(void);
	virtual void Think5(void);
	virtual void Fight(void);
	bool PauseShoot(void);
	void VATurnTo(const Vector &);
	void InstantTurn(void);
	bool FireWeapon( Vector &v_enemy);

	virtual void HandleMenu(void) = 0;
	virtual bool HandleOrders(void);
	virtual void AdaptAvoid(void);
	virtual void UpdateSkill(void);
	static void MakeName(char *,const char*,int,float);
	virtual void SpawnInit(void) = 0;
	virtual void Init(void) = 0;
	virtual long WeaponModel2ID(const char *);

	virtual bool AvoidStuck(void);
	virtual bool AvoidCollision(void);
	virtual bool CanJumpUp(void);
	virtual void FixIdealYaw(void);
	virtual void FixIdealPitch(void);
	virtual void TestOnButtonWay(edict_t **);
	virtual bool BoolTestOnButtonWay(int,int);

	virtual float ChangePitch( void );
	virtual float ChangeBodyPitch( void );
	virtual float ChangeBodyYaw( void );
	virtual float ChangeYaw( void );

	virtual bool HeadToward(Vector &);
	virtual bool InitCamp(void);
	virtual bool SearchForIL(void);
	virtual bool GoToSpEnt( void );
	virtual bool Bored(void);
	virtual bool HeadTowardWaypoint( void );
	virtual void OnLadder( void );
	virtual void AdjustLadder( void );
	virtual bool DecideOnWay(void);

	virtual edict_t *FindEnemy(void);
	virtual double ConvertDistance(const double);
	virtual bool GotoHidingPlace(void);
	virtual bool SearchHidingPlace(edict_t *);
	virtual long Change2Weapon(const long);
	virtual long ChangeToLWeapon(void);
	virtual void ShootAtEnemy( void );
	virtual Vector BodyTarget( edict_t *pBotEnemy);

	static long WeaponClass2ID(const char *);
	virtual long IsPrimaryWeapon(long lbit)=0;
	virtual long IsSecondaryWeapon(long lbit)=0;
	virtual long IsShotgunWeapon(long lbit)=0;
	virtual long IsSubMWeapon(long lbit)=0;
	virtual long IsRifleWeapon(long lbit)=0;
	virtual long IsSniperWeapon(long lbit)=0;
	virtual long IsGrenadeWeapon(long lbit)=0;
	virtual long IsCWeaponPrimary(void);
	virtual long IsCWeaponSecondary(void);
	virtual long IsCWeaponShotgun(void);
	virtual long IsCWeaponSubM(void);
	virtual long IsCWeaponRifle(void);
```

```
        virtual long IsCWeaponSniper(void);
        virtual long IsCWeaponGrenade(void);
        virtual long HasPrimary(void);
        virtual long HasSecondary(void);
        virtual long HasShotgun(void);
        virtual long HasSubM(void);
        virtual long HasRifle(void);
        virtual long HasSniper(void);
        virtual long HasGrenade(void);

        void ResetWPlanning(void);

        // public data
        edict_t *pEdict;

        FEItem FFE[32];

        bool need_to_initialize;
        char name[BOT_NAME_LEN+1];
        char skin[BOT_SKIN_LEN+1];    // useless in cs-> defined by model
        int not_started;
        int start_action;

        // TheFatal - START
        int msecnum;
        float msecdel;
        float msecval;
        // TheFatal - END
        float f_Delay;                // delay before first shoot
        float f_PauseShoot;           // time to pause shooting - Used by PauseShoot based on
Data from NNWeapon.cpp
        float f_DenyWChange;  // set it to a certain time when u don't want any weapon changes
        float f_TKPunish;
        float f_StrCamp;              // last time of strategic camping
        //bool bFight;                // flag, which is set to true, while fighting
        int iOnLadder;                // flag for ladder handling -> 1 means up, -1 down and 0
no ladder
        edict_t *pELadder;            // point to entity of current ladder - no ladder -> 0

        long lButton;                 // temp storage for pressed buttons
        // things from pev in CBasePlayer...
        int bot_team;
        int bot_teamnm;               // coherent with UTIL_GETTEAM ...
        int bot_class;
        int bot_health;
        int bot_armor;
        int bot_weapons;  // bit map of weapons the bot is carrying
        int bot_skill;

        bool bFlashlight;

        float fViewSpeedYaw;          // speed of changing view yaw
        float fViewSpeedPitch;        // speed of changing view pitch
        float fIdealAnglePitch;
        float fIdealAngleYaw;
        float fAngleSpeedPitch;
        float fAngleSpeedYaw;

        float f_ReactionTime;

        float yaw_degrees,pitch_degrees;
        float old_yaw_degrees,old_pitch_degrees;
        CRadio bot_IRadioM;
        //bot_nn_log sbNNLog;

        long lTypeOWW;                        // type of wanted way -> safe, 'kill-extensive',
no special stuff etc .... WW_*
        int iGoal;                                    // goal running to
        int iFarGoal;                         // goal where bot should arrive in some time
        int iWantedDiv;                               // wanted division of route

        edict_t *pNEn;                        // nearest enemy
        edict_t *pNTeam;                      // nearest teammember to enemy

        edict_t *pBotEnemy;                   // point to en edict
        edict_t *pLastGlobalKill;
        float f_SearchEnemy;
        edict_t *pBotOldEnemy;        // point to old en edict
```

```
	long lFightingAgainst;		// diff of bot teams fighting against
	Vector v_Offset;				// 'conscious' offset to locations bot's
shooting @ - angles
	float f_AMomentum;			// Aiming speed ( 1.0-value of momentum )
	float f_ASpeed;				// aiming speed (c)
	float f_AimHead;			// 0-1 ... to make better and defined HEADSHOTS
	float f_InDistance;			// max distance at which an enemy can be seen
instantly
	float f_TPD;				// time in milliseconds to view 1000 units further
	long lEnWeapon;				// weapon of enemy
	CSTMem STMem;				// short term memory
	CLTMem LTMem;				// long term memory

	CState Action;				// current action of bot
	edict_t *pButton2Use;		// only with BA_GOBUTTON
	int iButtonAGoal;			// the goal after the button is pressed
	//CState Order;				// current order to bot, i.e. fulfilling
radio command etc.
	CTask Task;
	CState Ordered;				// what the bot ordered

	float f_RWKnife;

	float f_IOrigD;				// distance to last most intersting origin
of an wp
	float f_LTSFIP;				// Last time searching for an interesting
place
	float f_LookTo;				// time to look a a location specified by
VLookTo, while running to VRunningTo
	float f_GoBack;				// normally the bot wouldn't look really
back , but u can force it to to this with this var.
	GrenadeOrder GOrder;
	Vector VRunningTo;			// ..
	Vector VLookTo;				// ..

	Vector VSuspEn;				// bot hears so there
	float f_SuspRem;			// time where bot remarked en ..
	float f_CheckSpEnt;
	//bool bSawSuspLoc;			// has susp loc been seen ?
	//bool bPlantDefBomb;		// bot's planting or defusing bomb
	edict_t *killer_edict;		// the guy who killed u ...
	edict_t *pEKilled;			// the guy u killed ...
	edict_t *pELVictim;			// the last guy u killed
	Vector LastEnemyOrigin;		// ...
	Vector VHidingPlace;		// where bot has decided to hide from current enemy
	float f_UsedRadio;			// last time using radio
	float f_shootbox;			// shoot through everything @ enemy
	bool bSeeThru;				// when shootbox, does the bot know the exact
position of enemy ?
	//float f_GotoHidingPlace;	// set to last time, where bot should try to reach
VHidingPlace
	int iJumpOC;				// the frames till a bot jumps over sth are stored
here
	float f_Hide;				// Bot hides this time ...
	float f_HidingTime;			// bot should hide this time when hidden
	//edict_t *pHidefrom;		// this var contains a pointer to the enemy the
bot is hiding from, when f_Hide is > gpGlobals->time

	float f_max_speed;			// max speed
	float f_strafe;				// strafing speed
	float prev_speed;			// speed at last iteration
	Vector v_prev_origin;		// ...
	float f_ducktill;			// ducktill this time
	float f_lastWP;				// time ya visited the last WP
	float f_ResetWP;			// last time the i_CurrWP was resetted
	float f_noWP;				// to to visit no WP
	float f_dont_check_stuck;
	float f_bot_stuck;			// last time bot was stuck
	long lStuckC;				// counter of getting stuck the last time
	float f_noStuck;			// time of last 'free' running
	float f_Camp;				// time, till bot is snipering
	float f_noCamp;				// don't stay a sniper points ( to get
there away )
	float f_CaLooktoWP;			// time of looking to one wp
	WAY	WLookTo;				// wp to look to;
		// snipering/camping
	short sLookToChange;		// direction of change			//
snipering/camping
```

```
	float f_Pause;				// pausing bot
	float f_blinded_time;		// flashbang
	float f_round_time;				// length of one round
	//float f_start_round;		// time of roundstart
	float f_end_use_ladder_time;		// end using ladder
	float f_start_use_ladder_time;			// start using ladder
	float f_use_button_time;

	float f_dont_avoid;
	float f_avoid_strafe;
	int i_avoidstrafe;				// value to strafe while avoiding obstacle
	float f_gotohostage;		// time to go to hostage after recognizing it
	float f_Bored;					// time, when bot was bored last time
	int i_BorA;						// action of this bored sequence
	float f_BoredTill;				// duration of 'bored period'
	float f_NNNUpdate;				// time after which an nn update is possible (
NextNN )
	double dCombatNNIn    [_NNBInput];   // last input of combat nn
	double dCombatNNOut   [_NNBOutput];  // last output of combat nn
	double dCollNNIn      [3];
	double dCollNNOut     [1];
	float f_Aggressivity;		// current aggressivity of bot. this value is added to
the assessement of the situation
	double d_Manner;				// is the bot aggressive of defensive in general ?
	long lPWeapon;					// Preferred weapon
	char szPWeapon[20];				// string of pweapon

	// stuff for replaying movement data
	ADV_WPRecMove *pWPAMPlay;
	float fStartADVM;
	bool bReplay;

	float f_move_speed;				// current wanted move speed
	float moved_distance;		// length of v_diff vector (distance bot moved)

	int i1C;						// every sec ( counting down every 5th and
if 0, set up to 5 )
	Vector v_diff_angles;		// ...
	float f_LastFight;				// time of last fight
	float f_IgnoreFall;				// ignore falling down -> at ladder p.ex.
	int i_FOV;						// current fov ... that obv not the same
like pEdict->v.fov

	float fRightDist,fMiddleDist,fLeftDist;		// 'fuehler' des Bots

	// 'not frontal' collision stuff
	//float f_old_direction;
	long l_cState;				// first (0) jump to come over obstacle - ifthis wasn't
sucessful and next time ya still stuckin', change to the direction of the obstacle (wall, ...)

	bool bRSInit;				// is set to 1 at beginning of a round

	// waypoint stuff
	int i_CurrWP;				// current wp bot is aiming to
	int f_D2C;
	int i_LastWP;				// last wp

	bot_current_weapon_t current_weapon;  // one current weapon for each bot
	int m_rgAmmo[MAX_AMMO_SLOTS];  // total ammo amounts (1 array for each bot)

	enum{
		_MAXLKW = 4
	};
	int iCLKW;
	int FLKW[_MAXLKW];

	CPersonality Personality;
protected:
private:
};

#endif __CBOTBASE_H
```

CBotCS.h

```
#ifndef __CBOTCS_H
#define __CBOTCS_H

#include "CBotBase.h"
#include "bot_chat.h"

class CBotCS : public CBotBase{
public:
bool DistanceSight(void);

	CBotCS();
	~CBotCS();

	virtual void Think(void);
	virtual void Fight(void);

	virtual void HandleMenu(void);
	virtual bool HandleOrders(void);
	virtual void SpawnInit(void);
	virtual void Init(void);
	virtual bool BuyWeapon(void);

	virtual bool ReactOnSound(void);
	virtual void ReactOnRadio(void);

	virtual long IsPrimaryWeapon(long lbit);
	virtual long IsSecondaryWeapon(long lbit);
	virtual long IsShotgunWeapon(long lbit);
	virtual long IsSubMWeapon(long lbit);
	virtual long IsRifleWeapon(long lbit);
	virtual long IsSniperWeapon(long lbit);
	virtual long IsGrenadeWeapon(long lbit);
	virtual long WeaponModel2ID(const char *);

	virtual Vector BodyTarget( edict_t *pBotEnemy);

	virtual edict_t *FindEnemy(void);
	bool IsBehindSG(edict_t *pPlayer);
	virtual bool FireWeapon( Vector &v_enemy );

	virtual bool HasAlreadyHostage(const edict_t *pHostage);

	virtual bool GoToSpEnt( void );
	virtual bool HeadTowardSpEnt(void);
	virtual void Goto(Vector &VGoto);
	virtual bool Bored(void);
	virtual bool HeadTowardWaypoint( void );
	virtual void FindRoute(int iNearWP);
	virtual bool DecideOnWay(void);
	void CheckTasksOnReachability(int);

	// data
	float f_buytime;                    // last time of buying
	int bot_money;
	bool bot_vip;                       // is bot VIP ?
	bool bGlockBurst;

	edict_t *pHostages[_MAXHOSTAGES];           // pointer to entities of used hostages
	int i_UsedHostages;                 // number of touched hostages
	CChat Chat;
protected:
private:
};

void BotBuy_CS_WEAPON_P228(CBotBase *pBot);
void BotBuy_CS_WEAPON_SCOUT(CBotBase *pBot);
void BotBuy_CS_WEAPON_HEGRENADE(CBotBase *pBot);
void BotBuy_CS_WEAPON_XM1014(CBotBase *pBot);
void BotBuy_CS_WEAPON_MAC10(CBotBase *pBot);
void BotBuy_CS_WEAPON_AUG(CBotBase *pBot);
void BotBuy_CS_WEAPON_SMOKEGRENADE(CBotBase *pBot);
void BotBuy_CS_WEAPON_ELITE(CBotBase *pBot);
void BotBuy_CS_WEAPON_FIVESEVEN(CBotBase *pBot);
void BotBuy_CS_WEAPON_UMP45(CBotBase *pBot);
void BotBuy_CS_WEAPON_SG550(CBotBase *pBot);
void BotBuy_CS_WEAPON_USP(CBotBase *pBot);
```

```
void BotBuy_CS_WEAPON_GLOCK18(CBotBase *pBot);
void BotBuy_CS_WEAPON_AWP(CBotBase *pBot);
void BotBuy_CS_WEAPON_MP5NAVY(CBotBase *pBot);
void BotBuy_CS_WEAPON_M249(CBotBase *pBot);
void BotBuy_CS_WEAPON_M3(CBotBase *pBot);
void BotBuy_CS_WEAPON_M4A1(CBotBase *pBot);
void BotBuy_CS_WEAPON_TMP(CBotBase *pBot);
void BotBuy_CS_WEAPON_G3SG1(CBotBase *pBot);
void BotBuy_CS_WEAPON_FLASHBANG(CBotBase *pBot);
void BotBuy_CS_WEAPON_DEAGLE(CBotBase *pBot);
void BotBuy_CS_WEAPON_SG552(CBotBase *pBot);
void BotBuy_CS_WEAPON_AK47(CBotBase *pBot);
void BotBuy_CS_WEAPON_P90(CBotBase *pBot);
void BotBuy_PAmmo(CBotBase *pBot);
void BotBuy_SAmmo(CBotBase *pBot);
void BotBuy_CS_WEAPON_HE(CBotBase *pBot);
void BotBuy_CS_WEAPON_FL(CBotBase *pBot);
void BotBuy_CS_WEAPON_SG(CBotBase *pBot);

#endif  __CBOTCS_H
```

BotNNDefs.h

```
#ifndef __BOTNNDEFS_H
#define __BOTNNDEFS_H

#include "../dlls/bot_weapons.h"

// def values for every weapon

// low range  -> -1
// high range ->  1 (-> awp);

const double dr_weapon_ak47                =   0;         // 4 1
const double dr_weapon_aug                 =  .6;         // 4 4
const double dr_weapon_awp                 = 1.0;         // 4 6
const double dr_weapon_c4                  =- .9;         //
const double dr_weapon_deagle        =  .0;         // 1 3
const double dr_weapon_elite         =  .1;         // 1 5 t
const double dr_weapon_flashbang     =- .5;         //
const double dr_weapon_g3sg1         =  .9;         // 4 7
const double dr_weapon_glock18             = -.5;         // 1 2 t
const double dr_weapon_hegrenade     =- .5;         //
const double dr_weapon_knife         =-1.0;
const double dr_weapon_m249                =- .3;         // 5 1
const double dr_weapon_m3                  =- .2;         // 2 1
const double dr_weapon_m4a1                =  .6;         // 4 3
const double dr_weapon_mac10         =- .1;         // 3 4 t
const double dr_weapon_mp5navy             =  .2;         // 3 1
const double dr_weapon_p228                = -.2;         // 1 4
const double dr_weapon_p90                 =  .1;         // 3 3
const double dr_weapon_tmp                 =  .3;         // 3 2
const double dr_weapon_usp                 = -.6;         // 1 1 ct
const double dr_weapon_scout         =  .9;         // 4 5
const double dr_weapon_sg552         =  .4;         // 4 2
const double dr_weapon_smokegrenade  =- .5;         //
const double dr_weapon_xm1014        =- .4;         // 2 2
const double dr_weapon_fiveseven     = -.2;         // 1 5 ct
const double dr_weapon_sg550         =  .9;         // 4 8
const double dr_weapon_ump45         =  .4;         // 3 5

// indices for layers of nn

enum Inputs{
       IHealth = 0,          // health of bot
       IDistance,                  // distance to en
       //IEnMove,                  // Movements of En
       //IPositionH,               // pos of en - left _ right
       IEWeapon,                   // pos of en - top _ bottom
       IWeapon,                    // special value for each weapon - range
       IAmmo,                      // amount of ammo in used clip
       ISituation,                 // situation, i.e. how many enemies are fighting against
how many allies ?
       IEND
};

const long _NNBInput = IEND;

enum Health{
       H_Low         = -1,
       H_Mid         = 0,
       H_High        = 1
};

enum Distance{
       D_Near        = -1,
       D_Mid         = 0,
       D_Far         = 1
};

enum EWeapon{
       EW_Low        = -1,
       EW_Mid        = 0,
       EW_Long       = 1
};

enum Weapon{
       W_Low         = -1,
       W_Mid         = 0,
```

```
	W_Long		= 1
};

enum AmmoClip{
	A_Low		= -1,
	A_Mid		= 0,
	A_High		= 1
};

enum Situations{
	S_Bad = -1,
	S_Normal =0,
	S_Good = 1
};

enum Outputs{
	ODuck = 0,			// Duck
	OJump,				// Jump
	OHide,				// run backward - forward
	OMoveType,			// Run - Walk
	OStrafe,			// strafe left - right
	OEND
};

const long _NNBOutput = OEND;

enum Duck{
	D_NoDuck	= -1,
	D_Duck		= 1
};

enum Jump{
	J_NoJump	= -1,
	J_Jump		= 1
};

enum Moves{
	M_Back		= -1,
	M_None		= 0,
	M_Forward	= 1
};

enum MoveType{
	T_None = -1,
	T_Walk = 0,
	T_Run  = 1
};

enum Strafe{		// strafe
	S_Left=-1,
	S_None=0,
	S_Right=1
};

enum Hide{
	H_Hide = 1,
	H_None = 0
};

// defs for collision NN

enum ICollII{
	ICI_Left = 0,
	ICI_Middle,
	ICI_Right
};

enum OCollLoc{
	OC_Left = -1,
	OC_Middle = 0,
	OC_Right = 1,
	OC_LEFT = -1,
	OC_MIDDLE = 0,
	OC_RIGHT = 1
};
```

#endif __BOTNNDEFS_H

Ausschnitt aus CBotBase (CBotBase::AvoidCollision)

```
bool CBotBase :: AvoidCollision(void){
	if(iOnLadder != 0 				// don't avoid collisions on ladders ;-)
		|| bReplay 				// no collsion avoidance when replaying
		|| f_dont_avoid > gpGlobals->time)   // general don't avoid stuff
		return false;

	Vector VStart, 				// start for all traces
		VEndLeft, 			// left aim
		VEndMiddle, 			// straight ahead
		VEndRight; 			// right aim
	//char buffer[100];
	double dOAngle = pEdict->v.angles.y * M_PI / 180.0,
		dAngle;
	//float fleft,fmiddle,fright;	// length of traces without fraction
	TraceResult tr; 				// temp storage of results

	if(g_b5th){ 				// just check that every .2 s
		nVec nVCollIn(3);

		nVCollIn.SetMemVec(dCollNNIn); 		// set mem vec to nninput

		VStart = pEdict->v.origin + pEdict->v.view_ofs;     // position of bot's eyes

		// make every vector to the same height  - calc the ends of the bot's 'feelers'
		VEndLeft.z = VStart.z;
		VEndMiddle.z = VStart.z;
		VEndRight.z = VStart.z;

		VEndMiddle.x = _CAL * cos(dOAngle) + VStart.x;
		VEndMiddle.y = _CAL * sin(dOAngle) + VStart.y;

		dAngle = dOAngle + _CAA * M_PI / 180.0;

		VEndLeft.x = _CAL * cos(dAngle) + VStart.x;
		VEndLeft.y = _CAL * sin(dAngle) + VStart.y;

		dAngle = dOAngle - _CAA * M_PI / 180.0;

		VEndRight.x = _CAL * cos(dAngle) + VStart.x;
		VEndRight.y = _CAL * sin(dAngle) + VStart.y;

		// trace lines
		UTIL_TraceLine(VStart, VEndLeft, dont_ignore_monsters,pEdict-
>v.pContainingEntity, &tr);
		fLeftDist = tr.flFraction;
#ifdef __COLLLINE
		if(fLeftDist != 1||1){
			if(pEdictPlayer){
				WaypointDrawBeam(pEdictPlayer, VStart, VEndLeft, 30, 0, 0, 0,
255, 250, 5);
			}
		}
#endif

		UTIL_TraceLine(VStart, VEndMiddle, dont_ignore_monsters,pEdict-
>v.pContainingEntity, &tr);
		fMiddleDist = tr.flFraction;

		// is the middle trace blocked by a breakable ent ???
		if(fMiddleDist!=1.0){
			if(strcmp(STRING(tr.pHit->v.classname),"func_breakable") == 0){
				if(tr.pHit->v.takedamage == DAMAGE_YES){ 		// boxes in
dust ?
					if(IsCWeaponGrenade()){ 	//dont try to destroy
func_breakable with grenades - can be unhealthy ( wooden boxes )
						if(HasSecondary() 		// try first with
secondary

	&&m_rgAmmo[weapon_defs[HasSecondary()].iAmmo1]>0){
							Change2Weapon(HasSecondary());
						}
						else if(HasPrimary()  // if no sec or no sec ammo,
try primary

	&&m_rgAmmo[weapon_defs[HasPrimary()].iAmmo1]>0){
							Change2Weapon(HasPrimary());
```

```
                                        }
                                        else{                                           //
try to destroy it with a knife
                                                Change2Weapon(CS_WEAPON_KNIFE);
                                        }
                                }
                                else{
                                        if(tr.flFraction*_CAL < 20.0){
                                                //f_move_speed = -f_max_speed;
                                        }
                                        //pEdict->v.angles.y = pEdict->v.v_angle.y;
                                        if(!PauseShoot()){ //BotFireWeapon(tr.pHit-
>v.origin,pBot);
                                                Vector v_destr = VEndMiddle -
GetGunPosition(pEdict);

                                                VATurnTo(v_destr);

                                                // select the best weapon to use at this
distance and fire...
                                                //if(f_Delay < gpGlobals->time)
                                                FireWeapon( v_destr );        // dod is
also for cs sufficient
                                                //lButton |= IN_ATTACK;
                                        }
                                }
                        }
                }
        }

#ifdef __COLLLINE
        if(fMiddleDist != 1||1){
                if(pEdictPlayer){
                        WaypointDrawBeam(pEdictPlayer, VStart, VEndMiddle, 30, 0, 0,
127, 255, 250, 5);
                }
        }
#endif

        UTIL_TraceLine(VStart, VEndRight, dont_ignore_monsters,pEdict-
>v.pContainingEntity, &tr);
        fRightDist = tr.flFraction;
#ifdef __COLLLINE
        if(fRightDist != 1||1){
                if(pEdictPlayer){
                        WaypointDrawBeam(pEdictPlayer, VStart, VEndRight, 30, 0, 0, 255,
255, 250, 5);
                }
        }
#endif

        if(fLeftDist == 1                               // if there's no obstacle
                && fMiddleDist == 1
                && fRightDist == 1){
                f_avoid_strafe = 0;
                return true;
        }
        else{
                // convert [-1;1]
                dCollNNIn[ICI_Left]           = fLeftDist * 2.0 - 1.0;
                dCollNNIn[ICI_Middle] = fMiddleDist * 2.0 - 1.0;
                dCollNNIn[ICI_Right]  = fRightDist * 2.0 - 1.0;

                nVCollIn.Normalize();         // normalize it;

                NNColl.SetInput(dCollNNIn);           // setting input for NN
                NNColl.Propagate();                   // propagating
                NNColl.GetOutput(dCollNNOut);         // and getting back the output

                if(dCollNNOut[0] > 0.5){
                        f_avoid_strafe = _MAXSTRAFE * 1.5f;

                        //f_move_speed *= ( 2.0f / 3.0f );          // just slow down a
little bit
                }
                else if(dCollNNOut[0] < -0.5){
                        f_avoid_strafe = - _MAXSTRAFE * 1.5f;
```

```
                                //f_move_speed *= ( 2.0f / 3.0f );          // just slow down a
little bit
                        }
                }
        }
        // check if the way to the next wp is blocked by a breakable ent ...
        if(i_CurrWP != -1){
                Vector VEndWP = waypoints[i_CurrWP].origin;
                VStart = pEdict->v.origin + pEdict->v.view_ofs;     // position of bot's eyes
                UTIL_TraceLine(VStart, VEndWP, ignore_monsters,pEdict->v.pContainingEntity,
&tr);

                if(strcmp(STRING(tr.pHit->v.classname),"func_breakable") == 0){
                        if(tr.pHit->v.takedamage == DAMAGE_YES){            // boxes in dust ?
                                if(IsCWeaponGrenade()){      //dont try to destroy
func_breakable with grenades - can be unhealthy ( wooden boxes )
                                        if(HasSecondary()            // try first with secondary
                                                &&m_rgAmmo[weapon_defs[HasSecondary()].iAmmo1]>0){
                                                Change2Weapon(HasSecondary());
                                        }
                                        else if(HasPrimary()  // if no sec or no sec ammo, try
primary
                                                &&m_rgAmmo[weapon_defs[HasPrimary()].iAmmo1]>0){
                                                Change2Weapon(HasPrimary());
                                        }
                                        else{                                          // try to
destroy it with a knife
                                                Change2Weapon(CS_WEAPON_KNIFE);
                                        }
                                }
                                else{
                                        if(tr.flFraction*_CAL < 20.0){
                                                f_move_speed = -f_max_speed;
                                        }
                                        if(current_weapon.iAmmo1 == 0){
                                                Change2Weapon(CS_WEAPON_KNIFE);
                                        }
                                        //pEdict->v.angles.y = pEdict->v.v_angle.y;
                                        if(!PauseShoot()){ //BotFireWeapon(tr.pHit-
>v.origin,pBot);
                                                Vector v_destr = VEndWP - GetGunPosition(pEdict);

                                                VATurnTo(v_destr);

                                                // select the best weapon to use at this distance
and fire...
                                                //if(f_Delay < gpGlobals->time)
                                                //BotFireWeapon( v_destr, pBot );
                                                lButton|=IN_ATTACK;
                                        }
                                }
                        }
                }
        }

        f_strafe += f_avoid_strafe;
        return false;
}
```

CBotCS_combat.cpp

```
#include "CBotCS.h"

void CBotCS :: Fight( void ){
	try{
		bool bDefensive = false;
		edict_t *pFlashbang=0;
		Vector v_enemy = pBotEnemy->v.origin - pEdict->v.origin;   // difference vector
from en and bot
		float fEDist = v_enemy.Length();

		bool bMeIVC = FInViewCone(&(pEdict->v.origin),pBotEnemy);

		if(!(Action.lAction & BA_FIGHT)             // check if it's a new fight
session
			&&f_LastFight + 5 < gpGlobals->time){       // don't allow delay in
fight, when en just ran out of view
			f_Delay = gpGlobals->time + f_ReactionTime;
		}

		if(pBotEnemy != pBotOldEnemy){              // enemy changed
			int index = ENTINDEX(pBotEnemy);
			if(index!=-1){
				if(f_ES[index] + _ESTIME < gpGlobals->time
					&&g_bUseRadio){               // i.e. not seen for a while
					FakeClientCommand(pEdict,"radio3",0,0);
	// and say enemy spotted
					FakeClientCommand(pEdict,"menuselect","2",0);
					f_UsedRadio = gpGlobals->time;
					int iETeam = UTIL_GetTeam(pBotEnemy);
					for(int i = 0;i < 32;i++){
						if(iETeam == UTIL_GetTeam(INDEXENT(i)))
			// set time back for every player of other team
							f_ES[i] = gpGlobals->time - _ESTIME /3.0;
		// don't say es too often, when groups hit each other
					}
				}
				f_ES[index] = gpGlobals->time;
			}
		}

		// del old ways
		//iGoal = -1;

		// till 'not going back to sniper point bug' not fixed ...
		f_Camp = 0;
		f_noCamp = 0;
		if(!GOrder.lTypeoG)
			f_LookTo = 0;

		// stop hiding, when seeing en
		if(f_Hide > gpGlobals->time)
			f_Hide = gpGlobals->time;

		// don't continue snipering when ya've no real sniper weapon
		if(HasSniper()){
			if(f_Camp > gpGlobals->time){        // dont move at sniper spot
				f_move_speed = 0;
			}
			else{                                        // slow down with sniper
weapon, to get a better accuracy
				f_move_speed /= 1.5f;
			}
		}

		// don't crouch long after hiding - no other functions use ducktill for such
long time ...
		if(f_ducktill > gpGlobals->time + 1.0f
			&& f_ducktill < gpGlobals->time + _HIDECAMPTIME -1.0f){
			f_ducktill = 0;
		}

		// don't check stuck for some time
		f_dont_check_stuck = gpGlobals->time + .5;

		if(f_NNNUpdate < gpGlobals->time){
```

```
                        //Vector bot_angles = UTIL_VecToAngles( v_enemy );  // convert ya
absolute dimensions to angles
                        //bot_angles.x>180.0?bot_angles.x -=360.0:1==1;            // just make
it to[-180;180]

                        lFightingAgainst =UTIL_FightingAgainst(pEdict,bot_teamnm,&pNEn) -

        UTIL_FightingAgainst(pBotEnemy,UTIL_GetTeam(pBotEnemy),&pNTeam)
                                                                + d_Manner
                                                                + f_Aggressivity;

                        lEnWeapon = WeaponModel2ID(STRING(pBotEnemy->v.weaponmodel));

                        /* Get Data for NN*/
                        dCombatNNIn[IWeapon]         =
WeaponDefs.dpWeaV[CSTRIKE_DLL][current_weapon.iId];
                        dCombatNNIn[IHealth]         = double(bot_health)/50.0-1.0;
                        dCombatNNIn[IDistance]       = ConvertDistance(fEDist);
                        dCombatNNIn[IEWeapon]        =
WeaponDefs.dpWeaV[CSTRIKE_DLL][lEnWeapon]/*bot_angles.x/180.0*/;
                        dCombatNNIn[IAmmo]                   = (double(current_weapon.iClip) /
double(WeaponDefs.ipClipSize[CSTRIKE_DLL][current_weapon.iId]))*2.0-1.0;
                        dCombatNNIn[ISituation]              = tanh(lFightingAgainst/2.0);

                        if (lEnWeapon==-1){
                                dCombatNNIn[IEWeapon] = 0;
                                sprintf(szTemp,"%s",STRING(pBotEnemy->v.weaponmodel));
                                FakeClientCommand(pEdict,"say",szTemp,0);
                        }

                        NNCombat.SetInput(dCombatNNIn);      // set input and propagate
                        NNCombat.Propagate();
                        NNCombat.GetOutput(dCombatNNOut);    // copy the outputs of the output
layers to this field

                        f_NNNUpdate = gpGlobals->time + (1.0f / gnn_update);
                        // collect data for SOM
                        /*if(dCombatNNIn[IAmmo]<=1&&dCombatNNIn[IAmmo]>=-1){
                                SP.AddPattern(dCombatNNIn);
                        }*/

                        // log nn vars for reinforcement learning

        /*memcpy(sbNNLog.dppNet[_LOGNETINPUT][sbNNLog.lAct],dCombatNNIn,sizeof(double)*IEND);

        memcpy(sbNNLog.dppNet[_LOGNETOUTPUT][sbNNLog.lAct],dCombatNNOut,sizeof(double)*OEND);
                        sbNNLog.lNum ++;
                        sbNNLog.lAct ++;
                        if(sbNNLog.lAct >= _MAXNNLOG){
                                sbNNLog.lAct = 0;
                        }*/
                        if(current_weapon.iId == CS_WEAPON_GLOCK18){
                                if(bGlockBurst){
                                        if(fEDist > 400){
                                                lButton |= IN_ATTACK2;
                                        }
                                }
                                else{
                                        if(fEDist < 250){
                                                lButton |= IN_ATTACK2;
                                        }
                                }
                        }
                }
                // ODuck
                if(dCombatNNOut[ODuck] > 0.5){
                        //lButton |= IN_DUCK;
                        f_ducktill = gpGlobals->time +.5;
                }
                // OJump
                if(dCombatNNOut[OJump] > 0.0){
                        //FakeClientCommand(pEdict,"say","NNJUMP",0);
                        if(!(pEdict->v.button & IN_JUMP)
                                && RANDOM_LONG(0,100) > 50)
                                lButton |= IN_JUMP;
                }

                // OHide
```

```
		if(dCombatNNOut[OHide] > .85){
			bDefensive=true;
		}

		// OMoveType
		if(current_weapon.iId != CS_WEAPON_KNIFE){
			if(dCombatNNOut[OMoveType] > 0.5){
				// do nothing cause default is max speed
			}
			if(dCombatNNOut[OMoveType] < 0.5
				&&dCombatNNOut[OMoveType] > -0.5){
				// walk ??
			}
			if(dCombatNNOut[OMoveType] <-0.5){
				f_move_speed = 0;
			}
		}

		// OStrafe
		if(dCombatNNOut[OStrafe] > 0.5){
			f_strafe += _MAXSTRAFE;
		}
		if(dCombatNNOut[OStrafe] <-0.5){
			f_strafe -= _MAXSTRAFE;
		}

		// ct bots shouldn't hide when the VIP is near to them
		if(bDefensive && g_iTypeoM == MT_AS){
			if(g_pVIP){
				if((pEdict->v.origin-g_pVIP->v.origin).Length() < 400.0)
					bDefensive = false;
			}
		}

		// handle stuff with runnign with knife
		if(f_RWKnife > gpGlobals->time){
			if(!IsCWeaponPrimary() && !IsCWeaponSecondary()){
				if(HasPrimary()){
					Change2Weapon(HasPrimary());
					bDefensive = false;
					if(f_RWKnife > gpGlobals->time){
						f_RWKnife = gpGlobals->time-1.1;
					}
				}
				else if(HasSecondary()){
					Change2Weapon(HasSecondary());
					bDefensive = false;
					if(f_RWKnife > gpGlobals->time){
						f_RWKnife = gpGlobals->time-1.1;
					}
				}
			}
		}

		pFlashbang = 0;
		if(RANDOM_LONG(0,100) < 20){
			while(pFlashbang = UTIL_FindEntityByClassname(pFlashbang,"grenade")){
				if(!strcmp(STRING(pFlashbang-
>v.model),"models/w_flashbang.mdl")){
					if(FVisible(pFlashbang->v.owner->v.origin,pEdict)){
						break;
					}
				}
			}
		}

		// check if bot's trying to get to a safer or better location
		//if(f_GotoHidingPlace > gpGlobals->time){
		if(Task.current&&Task.current->lType & BT_HIDE){
			// copy point to edict hiding from to bot_t.pHidefrom;
			//if(pEdictPlayer)WaypointDrawBeam(pEdictPlayer,pEdict-
>v.origin,VHidingPlace,100,10,255,255,255,100,50);
			//FakeClientCommand(pEdict,"say","hiding ...",0);
			// sometimes jump
			if(RANDOM_LONG(0,100) < 8
				&&moved_distance > 8.0f
				&&!(Action.lAction & BA_TKAVOID) ){         // only jump when
avoiding tks when nn wants it
```

```
                                lButton |= IN_JUMP;
                        }
                        GotoHidingPlace();
                }
                else{
                        // bot's not trying to reach a safe place - but just test now if it
would be better if he does ...
                        //bDefensive = BotCheckDefensive(pBot);
                        if(bDefensive
                        // is it important to get into a more defensive mode ?
                                || pFlashbang
                                && f_Hide < gpGlobals->time - 5.0f   // don't play hide and seek
for hours
                                /*&&f_GotoHidingPlace > gpGlobals->time+.5*/){      // just wait
one sec between two hidings to shoot shortly at the enemy
                                edict_t *pHidefrom;
                                if(!pFlashbang)
                                        pHidefrom = pBotEnemy;
                                else
                                        pHidefrom = pFlashbang;
                                if(SearchHidingPlace(pHidefrom)){    // then search for a good
place
                                        // found a nice place to camp for a while ...
                                        // set camp time based on current nn output
                                        f_HidingTime = (dCombatNNOut[OHide]-.25) * 1.6 *
_HIDECAMPTIME;
//                                      f_GotoHidingPlace = gpGlobals->time + 2.0;
                                        Task.AddTask(BT_HIDE, gpGlobals->time + 2.0, 0,
pBotEnemy);
//                                      pHidefrom = pBotEnemy;
                                        //FakeClientCommand(pEdict,"say","hiding cause of shitty
circumstances ....",0);
                                }
                        }
                }
                if(fEDist>150.0&&fEDist<1400.0
                        && bot_weapons & (1<<CS_WEAPON_HEGRENADE)
                        && (pNTeam != pEdict || !bMeIVC )){
                        //Change2Weapon(CS_WEAPON_HEGRENADE);
                        if(!GOrder.lTypeoG)
                                GOrder.Attack(pBotEnemy->v.origin,CS_WEAPON_HEGRENADE);
                }

                // check ammo
                if(!IsCWeaponGrenade()){             // don't check ammo with grenades - lol
                        if((m_rgAmmo[current_weapon.iAmmo1]+current_weapon.iClip) == 0){// if ya
clip  and ya reserve is empty
                                ChangeToLWeapon();
                                if(f_UsedRadio < gpGlobals->time - _RADIO_FREQ
                                        &&g_bUseRadio){
                                        FakeClientCommand(pEdict,"radio1",0,0);
        // and call for help
                                        FakeClientCommand(pEdict,"menuselect","6",0);
                                        f_UsedRadio = gpGlobals->time;
                                }
                        }
                        else if(current_weapon.iClip == 0){         // only ya clip is empty, so
you can propably reload, if the en is far away
                                ChangeToLWeapon();
        // use another weapon, worst case the knife - no special decisions concerning distance
of en
                                if(f_UsedRadio < gpGlobals->time - _RADIO_FREQ
                                        &&g_bUseRadio){
                                        FakeClientCommand(pEdict,"radio1",0,0);
        // and call for help
                                        FakeClientCommand(pEdict,"menuselect","6",0);
                                        f_UsedRadio = gpGlobals->time;
                                }
                        }
                }
                else{           // got grenade
                        if(fEDist < 150){
                                Change2Weapon(HasSecondary());
                        }
                }

                /*if(pEdict->v.punchangle.Length() > 3.5){          //  test the amount of
recoil
```

```
                        f_PauseShoot = gpGlobals->time +
WeaponDefs.dpWeaponPauseTime[current_weapon.iId];
                }*/

                if(!bMeIVC// enemy doesn't see current bot
                        //&& !IsCWeaponSniper(pBot)                                     //
don't try to hide with sniper weapon
                        //&& bDefensive
                // bot's defensive
                        && dCombatNNIn[IDistance]>0.0                       // when near, attack
in evry case
                        && f_AimHead < .9                                             //
begin to shoot when accurate
                        && !IsCWeaponGrenade()){                                 // don't wait
throwing grenades
                        f_Delay = gpGlobals->time +.01;             // don't shoot this time ...

                        float ffps = 1000.0 / msecval;
                        f_AimHead = pow(f_AimHead,sqrt(ffps/75.0));
                        f_move_speed /= 2.0f;                                        // slow down
to be less obvious ...
                }
                else
                if(bMeIVC){
                        f_AimHead -= msecval*.004 * f_AimHead;
                }

                // Set variable for knowing that last call was fight
                Action.lAction |= BA_FIGHT;

                // shoot at aim
                if(bBotsShoot){              // the friendly mode
                        if(current_weapon.iId == CS_WEAPON_KNIFE){
                                if(dCombatNNIn[IDistance] < -.5){
                                        ShootAtEnemy();                      // attack enemy with
knife only when near
                                        // Set time of last fight
                                        f_LastFight = gpGlobals->time;
                                }
                        }
                        else{
                                if(!PauseShoot()){                  // look at ya type of weapon
and decide to pause sometimes
                                        ShootAtEnemy();                      // attack enemy

                                        // Set time of last fight
                                        f_LastFight = gpGlobals->time;
                                }
                                else{
                                        //FakeClientCommand(pEdict,"say","Paused weapon",0);
                // sometimes the bots don't shoot after a map change - Why ??
                                        }
                        }
                }
                else{
                }

        }
        catch(...){
                FakeClientCommand(pEdict,"say","asdlkjglkjjljkglkjgljk",0);
                FILE *fhd = fopen("scheisse.txt","a");fprintf(fhd,"scheisse in
combat\n");fclose(fhd);
        }
}

bool CBotCS :: FireWeapon( Vector &v_enemy )
{
        // Use the Vector argument of the emeny to aim at the enemy and fire
        // here.  Return TRUE if bot fired weapon (enough ammo, etc.) otherwise
        // return FALSE to indicate failure (maybe bot would runaway then).
        if(f_Delay > gpGlobals->time)
                return false;

        // tk avoid
        Vector v_shoot;
        Vector v_gun;

        v_gun = GetGunPosition(pEdict);
```

```
        v_shoot = pEdict->v.v_angle + pEdict->v.punchangle;
        v_shoot.z = 0;  // reset roll to 0 (straight up and down)

        UTIL_MakeVectors( v_shoot );

        v_shoot = gpGlobals->v_forward.Normalize() * 1000.0;       // 1000 should be sufficient

        TraceResult tr;

        UTIL_TraceLine(v_gun,v_gun+v_shoot,dont_ignore_monsters,ignore_glass,pEdict-
>v.pContainingEntity,&tr);
        if(// tr.flFraction < .9
                UTIL_ClientIndex(tr.pHit) != -1
                && (tr.pHit != pBotEnemy)){// line is probably fract by enemy
                f_strafe += i_avoidstrafe * f_max_speed;
                Action.lAction |= BA_TKAVOID;               // set flag for only short hiding
or stop hiding when no teamm8 is in sight
                return FALSE;
        }
        else{
                if(Action.lAction & BA_TKAVOID){
                        // bot's currently avoid a team attack, but now he can shoot without
hurting team8s
                        Task.RemoveT(BT_HIDE);
                        Action.lAction &=~ BA_TKAVOID;
                }
                lButton |= IN_ATTACK;
                return TRUE;
        }
}
```

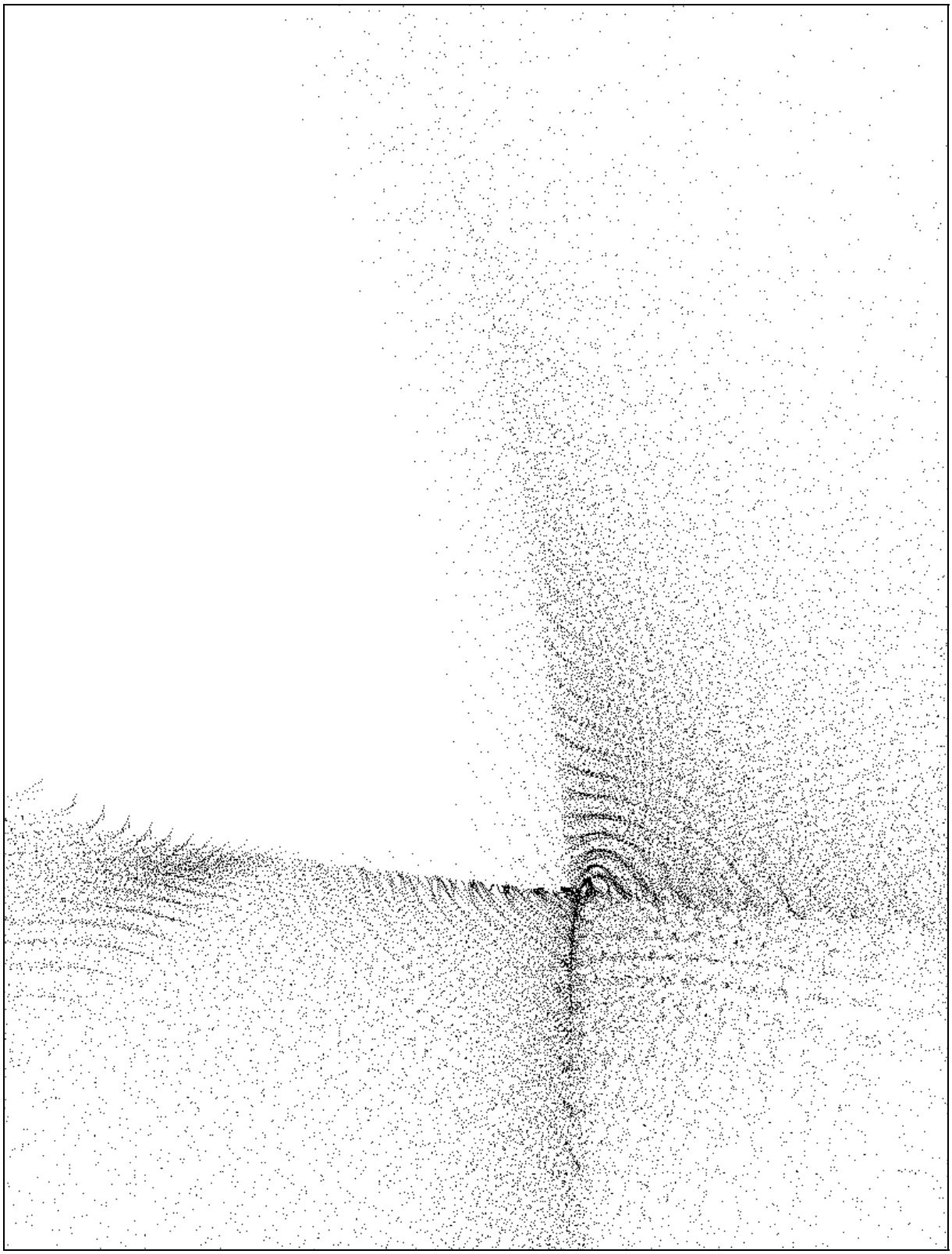
Anhang C : Erklärung

Ich habe diese Lernleistung selbstständig auf Basis der im Literaturverzeichnis angegebenen Quellen erbracht. Die Idee, einen Bot auf einem solchen NN basieren zu lassen, war meine, auch wenn der [NNBot] früher erschienen ist als der JoeBOT. (Dies lag daran, dass ich zu der Zeit keinen regelmäßig zu benutzenden Internetzugang hatte und somit erst ca. 8 Wochen nachdem [Botman] seinen Template Code veröffentlichte davon erfuhr.)
Die Bot-DLL verwendet keine fremden Quelltextfragmente außer dem Template Quellcode. Es werden nur Funktionen der Standard C/C++ Bibliotheken, der MS Win32 API und der HL API aufgerufen. Gleiches gilt für die Simulatoren und Hilfsprogramme mit dem Zusatz der OpenGL API bzw. bei älteren Versionen der MS DirectX Direct3DX API.
Ich bitte die beiliegenden Quelltexte nicht zu veröffentlichen.
Die Dokumentation und auch der darin enthaltende Quelltext darf veröffentlicht werden.

_____

Johannes Lampel

Anhang D: Verbreitung des Bots

Der Bot ist mittlerweile recht bekannt. Im Frühjahr und Sommer 2001 galt er sogar bei vielen als der beste Bot, was sich dann allerdings mit dem Erscheinen des PodBOTs 2.0 änderte. An der Wartung der JoeBOT Webseite arbeiten nun mehrere Personen und es hat sich ein kleines sogenanntes ‚Waypoint Team' gebildet, dass Waypoints für die einzelnen Maps von Counterstrike erstellen. Wenn ich eine neue Version herausbringe, kann ich mit einer Downloadzahl von 6000 innerhalb der ersten 24 Stunden nach der Veröffentlichung und 18000 innerhalb der ersten Woche rechnen. Der richtige Wert müsste jedoch höher liegen, da viele der zentralen Bot Seiten die Dateien auch auf ihren Servern bereitstellen und ich also nicht deren Downloadanzahl wissen kann.

Ausschnitt aus CBotBase (CBotBase::AvoidCollision)