Universität Leipzig
Fakultät für Mathematik und Informatik
(Institut für Informatik)

Neuronale Steuerungsparadigmen für autonome Roboter realisiert durch ein flexibles Software-Tool

Diplomarbeit

vorgelegt von : Jon Hennig

betreut von : Prof. Ralf Der, Universität Leipzig

Leipzig, Januar 2003

Zusammenfassung

Diese Arbeit gliedert sich im Wesentlichen in zwei große Teile. Der erste Teil beschäftigt sich mit der Implementation einer Bibliothek zur Simulation künstlicher neuronaler Netze. Der zweite Teil untersucht und vergleicht verschiede Lernverfahren zur Steuerung autonomer Roboter. Bei der Implementation der Lernverfahren wird dabei die Netzwerk-Bibliothek als Grundlage benutzt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Ausgangspunkt für diese Diplomarbeit war die Untersuchung neuronaler Steuerungsparadigmen zur Robotersteuerung. Auch wenn die Rechenkapazität moderner Computer die Kapazität einfacher Gehirne, wie z.B. bei Insekten vorhanden, bei dem aktuellen Entwicklungsstand bei weitem übersteigt, ist es bis heute nicht gelungen die von der Natur vorgegeben Steuerungsmechanismen, insbesondere in Bezug auf die Lernfähigkeit, einzuholen oder gar zu überbieten. Das deutet darauf hin, das es ein Defizit im Verständnis der natürlichen Lernverfahren gibt. Eine Untersuchung unterschiedlicher Lernverfahren für autonome Roboter kann vielleicht auch ein besseres Verständnis für die Vorgänge in natürlichen Systemen erzeugen.

Bei der Untersuchung der netzbasierten Lernverfahren zeigte es sich, dass viele neuronale Tools insbesondere zum Online-Lernen nicht oder nur in unzureichender Form zur Verfügung standen. Nach reiflicher Überlegung wurde sich dafür entschieden diese Werkzeuge in Form einer Klassenbibliothek als Teil dieser Arbeit mit zu entwickeln.

1.2 Überblick

Diese Arbeit gliedert sich im Wesentlichen in zwei große Teile. Der erste Teil (Kapitel 2) beschäftigt sich mit Neuronica, einer Bibliothek zur Simulation künstlicher neuronaler Netze, die als Teil dieser Diplomarbeit entwickelt wurde. Der zweite Teil (Kapitel 3) beschäftigt sich mit dem Vergleich verschiedener Lernverfahren zur Steuerung autonomer Roboter. Bei der Implementation der Lernverfahren wurde dabei die Neuronica-Netzwerk-Bibliothek als Grundlage benutzt.

In Kapitel 2 wird dargelegt, wieso die Entwicklung einer neuen Klassenbibliothek zur Simulation neuronaler Netze angebracht war. Das Prinzip und die Grundstruktur der Bibliothek wird vorgestellt. Es werden spezielle Lernverfahren und Netzstrukturen, wie Backpropagation und selbstorganisierende Karten näher betrachtet und erläutert. Dabei wird detailliert auf die algorithmische Umsetzung der Lernverfahren eingegangen. Die Funktionalität des Netzes wird mit dem Stuttgarter Neuronale Netzwerk Simulator (SNNS) als Referenz getestet. Weiter wird gezeigt, wie der SNNS-Simulator als Teil der Netzwerkklassen eingebunden werden kann.

Im Anhang A findet sich die Klassenreferenz von Neuronica. Diese wurde so gekürzt, dass lediglich die wesentliche Funktionalität widergespiegelt wird. Eine vollständige Referenz im HTML-Format befindet sich, ebenso wie der vollständige Quellcode der Klassenbibliothek, unter den Dateien, die Teil dieser Diplomarbeit sind.

Kapitel 3 widmet sich Steuerungsmechanismen für autonome Roboter und benutzt die entwickelte Netzwerkbibliothek um die beschriebenen Verfahren zu testen. Es werden im Wesentlichen drei Verfahren untersucht: Modell-/Controller-basiertes Reinforcement-Lernen, Homeokinese und ein netzbasiertes Verfahren, welches als Teil dieser Diplomarbeit entwickelt wurde, und welches das gleiche Lernziel wie das Homeokinese-Verfahren benutzt. Die Beziehungen der einzelnen Verfahren untereinander werden aufgezeigt, und die einzelnen Verfahren werden im Simulator und am realen Roboter getestet.

Kapitel 2

Klassenbibliothek zur Simulation neuronaler Netze

Dieses Kapitel widmet sich der Entwicklung und Implementation einer C++-Klassenbibliothek zur Simulation neuronaler Netze.

2.1 Vorbetrachtungen

2.1.1 Neuronale Netze - Überblick

Neuronale Netze sind informationsverarbeitende parallele Systeme. Unter neuronalen Netzen im engeren Sinne versteht man den Zusammenschluss von Nervenzellen, den Neuronen, zu einem reizverarbeitenden Netzwerk. Das sicherlich bekannteste und zur Zeit zugleich komplexeste neuronale Netz ist das menschliche Gehirn. In der Natur vorkommende neuronale Netze, wie zum Beispiel die Gehirne von Säugetieren, zeichnen sich durch besondere Eigenschaften aus. Zu ihren wesentlichen Merkmalen zählen unter anderem die Lernfähigkeit, die hochgradig parallele Informationsverarbeitung sowie die Robustheit gegenüber Fehlern.

Um sich diese Fähigkeiten von natürlichen neuronalen Netzen zu Nutze zu machen, hat man versucht die Prinzipien dieser Netze nachzubilden, d.h. die Vorgänge natürlicher neuronaler Netze zu simulieren. Dabei werden die Neuronen durch entsprechende (idealisierte) mathematische Modelle repräsentiert. In einer Simulation kann man nun verschiedene dieser Modelle nach dem Vorbild eines natürlichen Netzes zusammenwirken lassen. Man spricht dann von einem künstlichen neuronalen Netz (*artifical neural networks*).

Da der Schwerpunkt dieser Arbeit in der Auseinandersetzung mit künstlichen neu-

ronalen Netzen liegt, soll im Folgenden der Begriff neuronales Netz als Synonym für künstliches neuronales Netz und der Begriff Neuron für das Modell eines natürlichen Neurons verwendet werden.

2.1.2 Motivation zur Erstellung der Klassenbibliothek

Es gibt heute eine Vielzahl von Implementationen künstlicher neuronaler Netze (siehe z.B.[Lab01]). Es gibt Fachliteratur, die sich mit der Implementation verschiedener Netztypen beschäftigt (beispielsweise [Blu92]). Selbst einige frei verfügbare Bibliotheken existieren, die Implementationen von Backprop- und anderen neuronalen Netzen erlauben. Diese sind auch für viele Anwendungsgebiete vollkommen ausreichend und häufig bezüglich der Geschwindigkeit auf die Anwendung hin optimiert.

Innerhalb dieser Diplomarbeit war es aber von entscheidender Bedeutung, ein möglichst flexibles Netzwerk zur Verfügung zu haben, um neue Ansätze schnell testen zu können. Weiterhin sollte es möglich sein, von Standard-Architekturen abweichende Netzwerk-Layouts einfach einzusetzen. Deswegen wurden hier die vom Autor entworfenen Netzwerkklassen eingesetzt. Diese sind in C++geschrieben und wurden von Beginn an auf Flexibilität ausgelegt. Die Ausführungsgeschwindigkeit dieser Bibliothek liegt jedoch auf Grund des Overheads einer tiefen Klassenstruktur teilweise unterhalb anderer vergleichbarer Implementationen.[1]

2.2 Entwurf und grundlegendes Design

2.2.1 Ziele und Anforderungen

In Abschnitt 2.1.2 wurden bereits einige Ziele, die mit der Erstellung der Klassenbibliothek verbunden sind, angerissen. Im Folgenden sollen die Ziele und Anforderungen, die mit der Entwicklung der Klassenbibliothek verbunden waren, nochmals aufgeführt werden.

Ein Hauptkriterium für die Neuronica, so der Name der Bibliothek, ist es möglichst große Flexibilität beim Test und Entwurf neuer Algorithmen, die auf neuronalen Netzen aufbauen, zu bieten. Es soll weiterhin möglichst einfach sein bestehende Funktionalität abzuwandeln und zum Beispiel neue Neuronentypen zu verwenden. Dabei soll sich das Design möglichst nah am natürlichen Vorbild orientieren. Das Netz soll bei

[1] Wenn eine feste Netzstruktur für eine bestimmte Aufgabe gefunden wurde, ist es allerdings normalerweise relativ einfach diese Struktur so umzusetzen, dass eine optimale Geschwindikeit erreicht werden kann.

Bedarf dahingehend erweiterbar sein, dass natürliche neuronale Netze möglichst detailliert nachgebildet werden können. Jedes Neuron soll weitestgehend autonom auf Basis seiner Eingaben die Informationsverarbeitung vornehmen (hohe Lokalität der Informationsverarbeitung). Der Aufbau heterogener Netze soll ermöglicht werden, mehrere Arten von Neuronen sollen innerhalb eines Netzes auftreten können. Verschiedene Netze müssen zu einem größeren Netz zusammenschaltbar sein.

Neben diesen eher allgemeinen Anforderungen existieren auch weitere praktische Anforderungen, die sich teilweise daraus ergeben, dass das neuronale Netz zur Steuerung autonomer Roboter eingesetzt wird:

- einfache Einbindung in bestehende Programme
- Online-Training der Netze muss möglich sein, um die Biblitothek zur Steuerung eines autonomen Roboters einzusetzen.
- Die Verarbeitung muss schnell genug erfolgen, um zumindest für einfachere Netze eine Echtzeit-Steuerung eines Roboters zu ermöglichen.
- möglichst gute Zusammenarbeit mit anderer Software zur Simulation neuronaler Netze
- Implementation des Standard-Backpropagation-Algorithmus
- Erweiterungen des Backpropagation-Algorithmus sollen einfach möglich sein.
- Möglichkeit zur Erweiterung auf andere neuronale Lernverfahren
- Bei einem Netz, dass aus mehreren Teilnetzen besteht, soll jedes Teilnetz seperat speicherbar und sperat trainierbar sein.
- Interoperabilität mit dem SNNS-Simulator

2.2.2 Das grundlegende Konzept

Die Anforderungen an die Klassenbibliothek, insbesondere die leichte Erweiterbarkeit und Anpassbarkeit legen einen Entwurf in einem objektorientierten Design nahe. Um eine möglichst einfache Einbindung in andere Projekte zu ermöglichen, wurde C++ als Programmiersprache gewählt.

Objektorientierte Analyse zum neuronalen Netzwerk[2]

Die angestrebte Orientierung am natürlichen Vorbild legt die tatsächlichen Strukturen eines natürlichen neuronalen Netzes als Ausgangspunkt nahe. Natürliche neuronale

[2] Hinweise auf die Analyse-Techniken finden sich unter anderem in [Boo94]

Netze bestehen aus einer Vielzahl von Neuronen, die untereinander über Synapsen verbunden sind. Als Objektklassen bieten sich daher die Neuronen, die Synapsen als auch das Netz selbst an. Die Netzwerkbibliothek soll darüber hinaus fähig sein, bestehende Lernverfahren für künstliche neuronale Netze auszuführen. Da die dafür nötigen Algorithmen in der Literatur meist auf Netz-Ebene vorgegeben sind (siehe [Zel94]), könnte man auch an eine Abstraktion auf Basis der Netze denken. In der Tat wurde dieser Ansatz von einigen Autoren (z. B. [Blu92]) verfolgt. Ein Design, welches lediglich auf Netzen als Basisobjekte aufbaut, ist allerdings nicht gut geeignet, um die oben gestellten Anforderungen vollständig zu erfüllen.

Ein Netz sollte auf jeden Fall ein eigenständiges Objekt sein. Es ist die Basis für alle Zugriffe von Außen. Ein Programm, welches das neuronale Netz benutzt, sollte normalerweise nur auf die Funktionen des Netzes zugreifen müssen (Kapselung). Für fast alle neuronalen Netze besteht eine prinzipielle Gemeinsamkeit: Dem Netz werden bestimmte Werte präsentiert (die Input-Daten). Darauf (und auf seinen inneren Zustand) aufbauend erzeugt es ein Ergebnis[3], das im Allgemeinen an den Aufrufer zurück gegeben wird. Der innere Zustand des Netzes kann sich dabei auf Basis der Input-Daten ändern.

Für fast alle neuronalen Lernverfahren dient das Neuron als Basiseinheit. Es erhält dabei ein oder mehrere Eingangssignale und liefert daraufhin ein Ergebnis (Output) zurück. Die Eingangssignale und eventuelle Fehlersignale können den Zustand des Neurons verändern. Die Neuronen werden ebenfalls als eigenständige Objektklasse implementiert.

Einzelne Neuronen sind über Synapsen miteinander verbunden. In vielen neuronalen Lernverfahren wird von den Synapsen abstrahiert und lediglich von Gewichten gesprochen, die die Stärke einer Verbindung zwischen zwei Neuronen repräsentieren. Ein Gewicht ist dabei ein Zahlenwert, der für die Stärke der synaptischen Kopplung steht. Das Prinzip der Gewichte geht aber auf das Vorbild der Synapsen zurück. Synapsen natürlicher Nervenzellen sind nicht allein durch die Stärke der Reizweiterleitung gekennzeichnet, sondern ebenfalls durch weitere Parameter, wie zum Beispiel die Produktionsrate des Neurotransmitters, sowie die Inaktivierungsrate bereits gebundener (d.h. ausgeschütteter) Neurotransmitter. Des Weiteren gibt es Lernverfahren, bei denen die Verwendung spezialisierter Synapsen nahe liegt. Um diesen zusätzlichen Anforderungen Rechnung zu tragen und möglichst nah am Vorbild der Natur zu bleiben, wurden auch die Synapsen als eigenständige Objektklasse implementiert.

Als eine weitere Klasse wurde eine Schicht von Neuronen (im folgenden Neuro-Layer) gewählt. Die Notwendigkeit dieser Klasse ergibt sich nicht direkt aus dem natürlichen Vorbild, sondern ist vielmehr der Tatsache geschuldet, dass viele neuronale Lernverfah-

[3]Das gilt auch für Netze, bei denen dieser Zusammenhang nicht so offensichtlich ist. So ist zum Beispiel bei selbstorganisierenden Karten das Gewinnerneuron das Ergebnis der Verarbeitung. Ein Ergebnis können aber auch innere Parameter des Netzes sein, bei einer selbstorganisierenden Karte zum Beispiel die Gewichte der einzelnen Kohonen-Neuronen.

ren auf Schichten von Neuronen aufbauen. Eine Verwaltung einer Vielzahl von Neuronen wird durch die Aufteilung in verschiedene Schichten wesentlich erleichtert. Alle Neuronen einer Schicht sollten dabei parallel agieren können, d.h. kein Neuron einer bestimmten Schicht sollte zu einem festen Zeitpunkt t auf ein Ergebnis eines Neurons der selben Schicht angewiesen sein.

Ein Kandidat für eine Objektklasse ist weiterhin das zwischen den Neuronen übertragene Signal. Wie in [Derb] aufgeführt gibt es verschiedene Theorien zur Informationskodierung zwischen den Neuronen. Fast alle Algorithmen und Verfahren gehen aber davon aus, dass zur Beschreibung des Informationsaustausches zwischen den Neuronen die Feuerrate der Neuronen genügt. Diese lässt sich hinreichend gut als Zahlenwert modellieren, eine eigene Klasse ist daher nicht notwendig. Will man weitere Parameter wie die zeitliche Abfolge der einzelnen Spikes zwischen den Neuronen beachten, kann man eine neue Klasse einführen, die diese Übertragungsart kodiert.

Als Basis für die Bibliothek dienen also eine Klasse für das gesamte Netz, eine für eine Schicht von Neuronen, eine Klasse für die Neuronen sowie eine für die Synapsen. Die

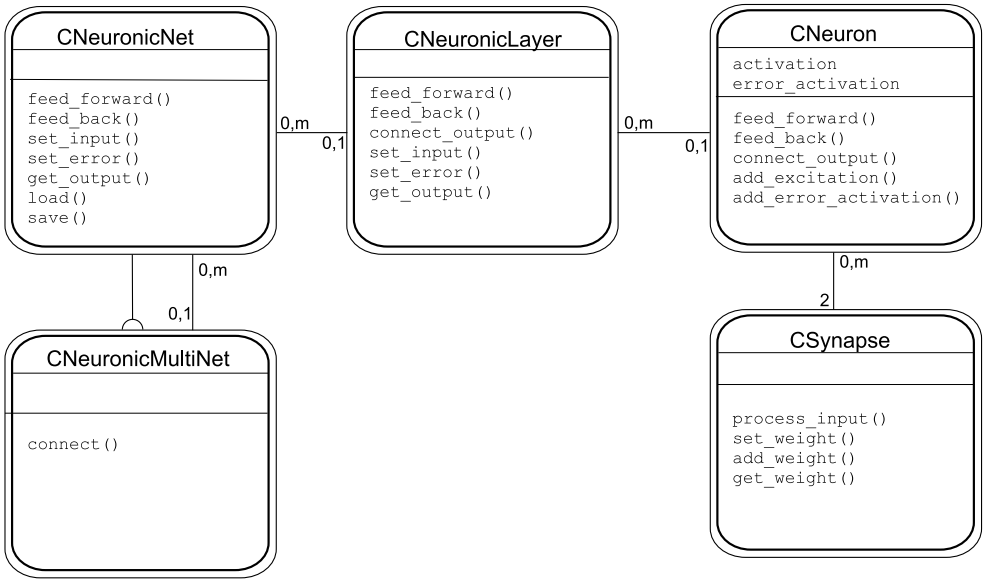


Abbildung 2.1: Basisklassen der Neuronica-Bibliothek mit den wichtigsten Parametern

eigentliche Funktionalität des Netzes wird soweit wie möglich auf der Ebene der Neuronen ausgeführt. Die in der Hierarchie höher liegenden Klassen, wie z.B. die Schichten- oder die Netzklassen, sind sofern möglich für die verschiedenen Netztypen gleich und verwalten lediglich die Funktionalität der einzelnen Neuronen. Für bestimmte Verfahren und Netztypen sind allerdings spezialisierte, von den Basisklassen abgeleitete Klassen notwendig (siehe beispielsweise Abschnitt 2.5.2).

Viele herkömmliche Implementationen repräsentieren einzelne Neuronen bzw. Synapsen nicht als eigenständige Klassen, sondern verwalten diese als Parameter-Vektoren

bzw. Matrizen. Die eigentliche Funktionalität muss dann von der entsprechenden Netz-Klasse verrichtet werden. Im Gegensatz zu den meisten herkömmlichen Implementationen werden bei diesem Ansatz sowohl Neuronen als auch Synapsen als eigene Klassen implementiert. Das bringt Vorteile in Bezug auf die Flexibilität, aber durch den erhöhten Verwaltungsaufwand eventuell Nachteile bei der Ausführungsgeschwindigkeit. Im Abschnitt 2.3 wird detailliert auf die Implementation der einzelnen Klassen eingegangen.

Funktionsprinzip

Ein neuronales Netz wird repräsentiert durch eine entsprechende Klasse. Es besteht aus mehreren Schichten, im Allgemeinen mindestens aus 2, der Eingabe-Schicht und der Ausgabe-Schicht. Eine Schicht besteht normalerweise aus mehreren Neuronen. Die einzelnen Neuronen einer oder verschiedener Schichten können untereinander über Synapsen verbunden sein. Der Aufbau des Netzes, und damit das Erzeugen der nötigen Objekte (und deren Verbindungen), erfolgt zur Laufzeit des Programmes durch den Benutzer der Klassenbibliothek. Dies geschieht entweder direkt oder indirekt (zum Beispiel durch Laden einer Netz-Datei).

Wenn die Struktur des Netzes komplett aufgebaut ist, kann das Netz benutzt werden. Dazu werden die dem Netz zu übergebenden Werte in die Input-Neuronen[4] geschrieben. Dann werden alle Neuronen gemäß ihrer zeitlich definierten Abfolge aufgerufen, um die anliegenden Eingangssignale zu verarbeiten und an die nachgeschalteten Neuronen weiterzuleiten. Der an den Neuronen der Output-Schicht erzeugte Output wird ausgelesen und bei Bedarf zurück geliefert. Nun kann den Output-Neuronen ein Fehlerwert übergeben werden. Dieser wird analog der Vorwärts-Propagierung nur in umgekehrter Richtung zurück geliefert. Der Aufruf erfolgt dabei genau umgekehrt zu der zeitlich definierten Abfolge. Mit diesem einfachen Prinzip lassen sich, bei geeigneter Implementation der Neuronen, Lernverfahren wie zum Beipsiel Backpropagation implementieren. Die Klassen für das Netz und die Schichten sind für diese Art der Verarbeitung nicht zwingend nötig. Sie erleichtern aber zum einen den externen Zugriff auf das Netz und zum anderen die (interne) Verwaltung der Neuronen. So können per Definition alle Neuronen einer Schicht zum selben Zeitpunkt aktiv sein - kein Neuron einer Schicht ist zu einem festgelegten Zeitpunkt t vom Output eines Neurons der selben Schicht zum selben Zeitpunkt abhängig. Daher ergibt sich, dass es zur Wahrung der zeitlichen Abfolge genügt die einzelnen Schichten gemäß ihrer zeitlichen Abfolge aufzurufen.

[4] siehe auch Abschnitt 2.3.1

2.3 Implementation der Basisarchitektur

2.3.1 Das verwendete Basis-Neuronen-Model

Das Neuron ist der Grundbestandteil eines jeden Neuronalen Netzes. Ein abstraktes Modell des hier verwendeten Neurons wird in Abbildung 2.2 dargestellt. Es ist in der Klasse `CNeuron` wie folgt implementiert: Jedes Neuron hat eine bestimmte Anzahl von

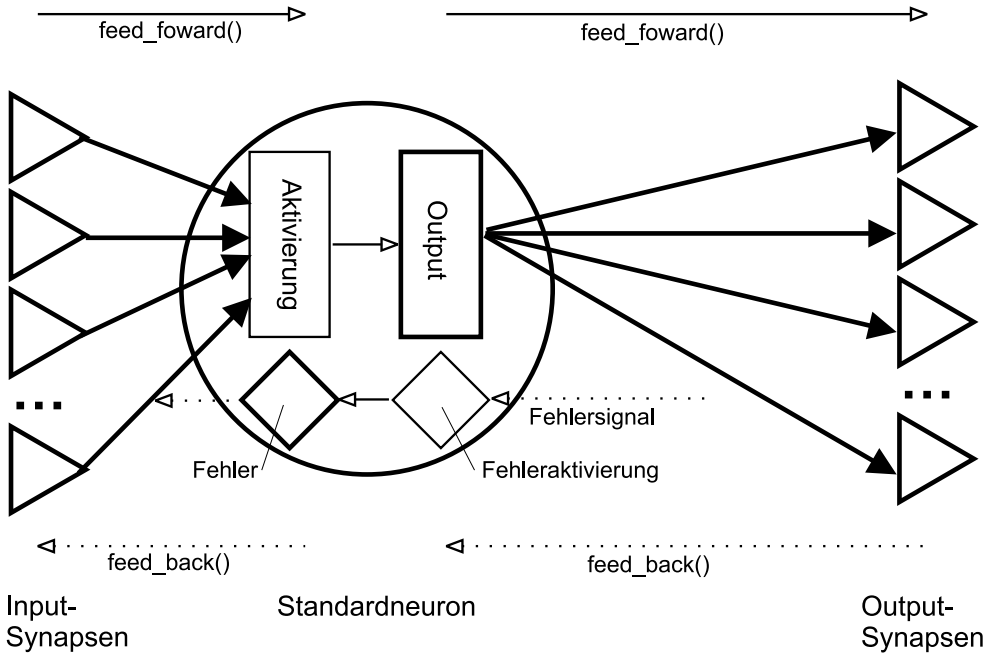


Abbildung 2.2: Basismodell eines Neurons

Eingängen (Inputs) und Ausgängen (Outputs). Die Inputs und Outputs werden realisiert durch Verweise auf jeweils eine Synapse (siehe auch Abschnitt 2.3.2). Im Folgenden werden Input-Synapsen, d.h. die Synapsen von denen das Neuron Signale erhält, und Output-Synapsen, d.h. die Synapsen an die das Neuron Signale sendet, unterschieden.

Die über die Inputs eingehenden Signale werden in der Aktivierung (`activation`) aufsummiert. Die Signale kommen dabei von den Input-Synapsen. Über die Aktivierungsfunktion wird der Ausgabewert (Output) aus der Aktivierung des entsprechenden Neurons berechnet. Der Output wird erst ermittelt, wenn alle Signale von den Input-Synapsen zu einem bestimmten Zeitpunkt eingegangen sind. Der Ausgabewert des Neurons kann an die Output-Synapsen (und damit an die nachfolgenden Neuronen) weiterpropagiert werden.

Für die Erregungsweiterleitung (d.h. die Weiterleitung der Signale von den Input- zu den Output-Neuronen) sind im Wesentlichen zwei Funktionen verantwortlich, und zwar `feed_forward()` und `add_excitation()`.

`feed_forward()` liest die aufsummierte Aktivierung des Neurons ein (wobei diese gleichzeitig auf 0 zurückgesetzt wird) und berechnet daraus mittels der Aktivierungsfunktion den Output, welcher daraufhin über die Synapsen an den Ausgängen an die Nachfolgerneuronen weiterpropagiert wird.

Die Funktion `add_excitation()` wird von den Input-Synapsen aufgerufen um einen bestimmten Wert zur Aktivierung des Neurons hinzuzufügen. Die Aktivierung kann bei Bedarf auch auf einen fest vorgegebenen Wert eingestellt werden.

Weiterhin exisitert die Funktion `connect_output()`. Diese Funktion legt eine neue Output-Synapse an und verbindet das Neuron über diese mit einem anderen Neuron.

Mit diesem Modell ist es bei entsprechender Implementation der Synapsen schon möglich einfache Feed-Forward-Netze zu erstellen. Es muss lediglich darauf geachtet werden, dass die Neuronen über die Synapsen entsprechend des gewünschten Netz-Layouts korrekt miteinander verbunden sind, sowie dass die `feed_forward()`-Funktionen der Neuronen in der entsprechenden Reihenfolge aufgerufen werden.

Bei vielen Lernverfahren wird für das Neuron ein Fehler berechnet und weiter geleitet. Um dieser Tatsache Rechnung zu tragen, wurden die Funktionen `feed_back()` und `add_error_activation()` aufgenommen. Diese dienen, ähnlich wie die oben beschriebenen Funktionen zur Erregungsweiterleitung, zur Weiterleitung eines Fehlersignales allerdings in die entgegengesetzte Richtung. In der Basisklasse sind diese Funktionen jedoch nicht implementiert.

Das Lernen in neuronalen Netzen erfolgt oftmals über das Adaptieren der Parameter der Synapsen. Dabei bestimmen die verbundenen Neuronen, wie diese Parameter geändert werden. Bei einer Implementation in der hier beschriebenen Art sind zumindest zwei Varianten denkbar: Zum einen, dass ein Neuron nur die Input-Synapsen modifiziert und zum anderen nur die Output-Synapsen. In diesem Framework wurde sich dafür entschieden, dass die Neuronen jeweils ihre Input-Synapsen modifizieren sollen.

Das Input-Neuron

Um neuronale Netze sinnvoll anwenden zu können, wird ein Eingangssignal (Input) benötigt, dass verarbeitet werden soll. Die Input-Neuronen übernehmen die Aufgabe diesen Input aufzunehmen und an die nachgeschalteten Neuronen weiter zu leiten. Theoretisch wäre es denkbar, die Daten direkt an die nachgeschalteten Neuronen zu senden, ohne dass ein Input-Neuron zwischen geschaltet wird. In diesem Fall müsste eine spezielle Behandlung des Inputs in den nachgeschalteten Neuronen erfolgen. Die Verwendung von Input-Neuronen erlaubt eine konsistente Kommunikation der Neuronen untereinander. Ein Neuron muss nicht unterscheiden, ob das Eingangssignal ein Netzinput oder das Signal von einem anderen Neuron ist, da die Kommunikation nur zwischen den Neuronen (über die zwischengeschalteten Synapsen) erfolgt.

Ein weiterer Vorteil der Verwendung von Input-Neuronen liegt darin, dass damit für das Netz irrelevant ist, ob das Input-Signal von außen kommt oder beispielsweise von einem anderen Netz stammt. Zu diesem Zweck kann ein Input-Neuron ein anderes Neuron kapseln (siehe Abbildung 2.3). Statt eines von außen vorgegebenen Outputs

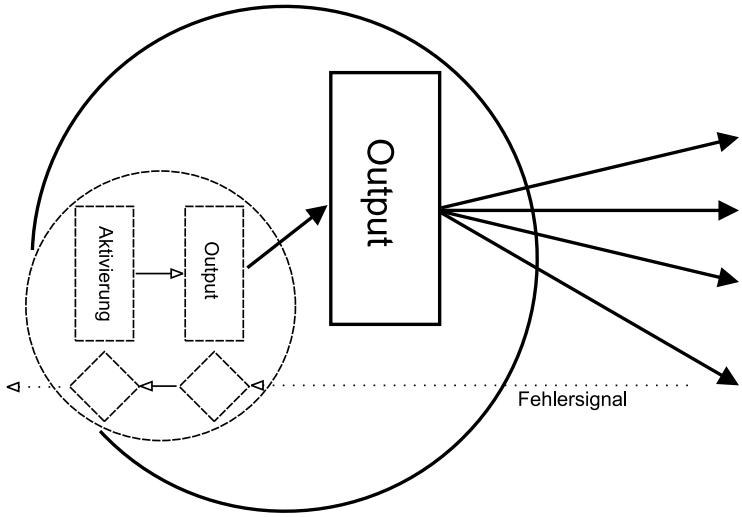


Abbildung 2.3: Modell eines Input-Neurons

wird dann der Output des gekapselten Neurons weitergeleitet. Diese Funktionalität wird verwendet, um mehrere Netze zusammenschalten zu können, ohne dass an den Teilnetzen eine Veränderung erfolgen muss (siehe auch Abschnitt 2.3.5).

Die Funktionalität des Input-Neurons wird in der Klasse `CInputNeuron` implementiert. Diese ist von `CNeuron` abgeleitet und implementiert zusätzlich Funktionen zum Kapseln von anderen Neuronen. Um ein Input-Neuron auf einen bestimmten Wert zu setzen wird die Funktion `set_output()` verwendet.

2.3.2 Modellierung einer Synapse

Eine Synapse (Klasse `CSynapse`) verbindet jeweils 2 Neuronen, den Vorgänger (präsynaptisches Neuron) und den Nachfolger (postsynaptisches Neuron) (siehe Abbildung 2.4). Jede Synapse besitzt eine Funktion `process_input()`. Diese empfängt den Output des präsynaptischen Neurons, verarbeitet diesen und leitet das Ergebnis an das postsynaptische Neuron weiter. In den meisten Lernverfahren besitzt eine Synapse nur einen beschreibenden Parameter, nämlich das Gewicht. Der Verarbeitungsschritt besteht dann darin, dass das anliegende Signal (der Output des Vorgänger-Neurons) mit dem Gewicht der Synapse multipliziert wird. Diese Funktionalität ist in der Basisversion der Synapse bereits modelliert.

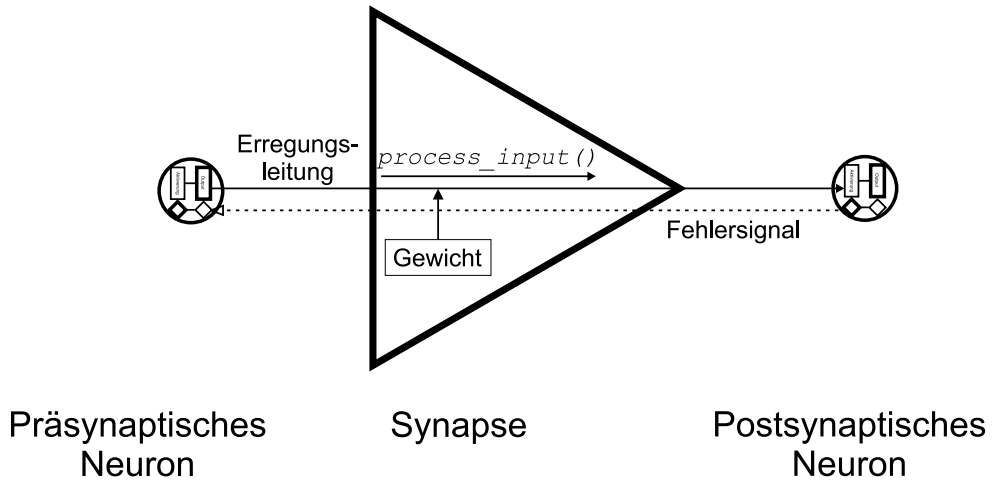


Abbildung 2.4: Basismodell einer Synapse

Des Weiteren hat die Standard-Synapse Funktionen zum Auslesen, Ändern und Setzen des Gewichtes (`set_weight()`, `add_weight()` und `get_weight()`).

2.3.3 Modellierung einer Schicht von Neuronen

Zur vereinfachten Verwaltung mehrerer Neuronen wird die Klasse `CNeuronicLayer` eingeführt, die eine Schicht von Neuronen darstellt. In einer solchen Schicht befindet sich eine bestimte Anzahl von Neuronen (siehe Abbildung 2.5). Da bestimmte Funktionen für alle Neuronen einer Schicht gleichzeitig aufgerufen werden können, werden solche Aufrufe vereinfacht (zum Beispiel das oben beschriebene Verknüpfen von Neuronen und das Aufrufen der `feed_forward()`-Funktion). Neuronen einer Schicht können über die Funktion `connect_output()` einfach mit Neuronen einer anderen Schicht verbunden werden. Die `feed_forward()`- und `feed_back()`-Funktionen können für alle Neuronen einer Schicht gleichzeitig aufgerufen werden. Über einen Vektor bietet die Klasse einen einfachen Zugriff auf die Outputs und die Inputs (d.h. die Aktivierungen) der Neuronen. Für jede Schicht ist eine Skalierung einstellbar. Damit kann bei einem Zugriff von außen die Skalierung der Werte in einem bestimmten für das Netz günstigen Bereich vorgenommen werden. Mit dieser Funktionalität kann man zum Beispiel eine Normierung der Input-Werte erreichen oder den Output, der beispielsweise im Bereich zwischen -1 und 1 liegt, in einen für den Nutzer relevanten Wert umwandeln.

Man kann leicht sehen, wie man durch Nacheinanderschaltung einzelner Schichten ein entsprechendes Feed-Forward-Netz aufbauen kann. Dazu muss jede Schicht mit ihrer Nachfolgerschicht verbunden werden. Um sich in einem solchen Netz aus einem Input ein Output berechnen zu lassen, muss man lediglich die `feed_forward()`-Funktion

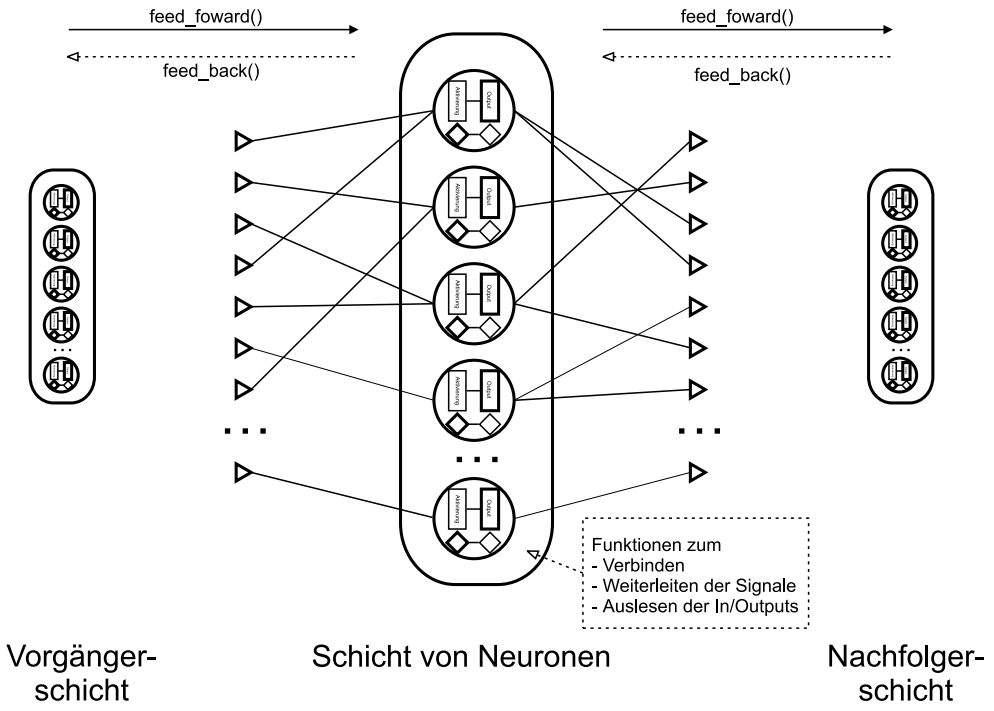


Abbildung 2.5: Modell einer Schicht von Neuronen

der Schichten der Reihe nach aufrufen.

2.3.4 Modellierung eines gesamten Netzes

Die Klasse `CNeuronicNet` dient zur Modellierung eines Netzes. Ein Netz besteht aus mehreren Schichten (im Allgemeinen mindestens aus der Eingabe- und der Ausgabeschicht). Die Klasse bietet Möglichkeiten zum Verwalten verschiedener Schichten und übernimmt auf Wunsch die Verknüpfung einzelner Schichten sowie die Weiterleitung des Feed-Forward-bzw. Feed-Back-Signals (siehe Abbildung 2.6). Für die meisten Anwendungsfälle genügt es auf diese Klasse zuzugreifen. Die erste Schicht des Netzes repäsentiert die Input-Schicht des Netzes, die letzte Schicht die Output-Schicht. (Es ist auch möglich mehrere Input-Schichten zu definieren). Die Funktionen `feed_forward()` und `feed_back()` dienen zum Weiterleiten der Erregung bzw. des Fehlersignals im Netz. Mit den Funktionen `set_input()`, `set_error()` und `get_output()` kann der Input des Netzes bestimmt werden, der Fehler für das Netz gesetzt werden, sowie das Netzergebnis ausgelesen werden.

Zum Laden und Speichern des Netzes existieren die Funktionen `load()` und `save()`. Als Format stehen das Neuronica-spezifische XML-basiserte NND-Format (Neuronica

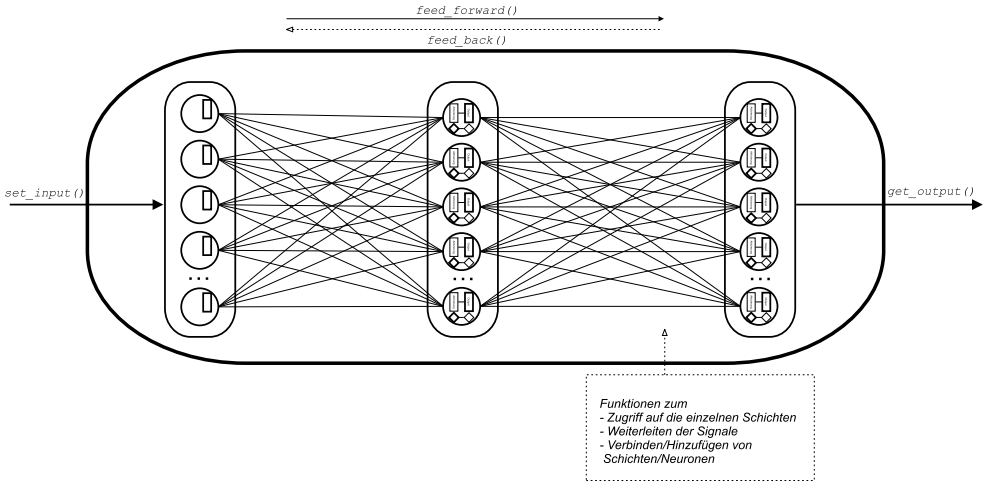


Abbildung 2.6: Modell eines kompletten neuronalen Netzes

Network Definition Format) und das SNNS-spezifische NET-Format[5] zur Verfügung. Die Nutzung eines eigenen Formates erwies sich als nötigt, da Neuronica teilweise mehr Parameter und Einstellungsmöglichkeiten anbietet als der SNNS.

Die Neuronica-Bibliothek stellt außerdem Funktionen zum Laden von Input-Daten für das Netz (Pattern-Files) zur Verfügung. Als Format für diese Pattern-Files wird das selbe Format verwendet wie beim SNNS[6]. Die Pattern-Files können direkt zum Lernen mit dem Netz benutzt werden.

2.3.5 Verknüpfen mehrerer Netze

Bei komplexen Netzwerkstrukturen ist es häufig nötig, das Netz in mehrere Teilnetze aufzuteilen. Dabei ist es oftmals sehr nützlich, wenn diese Teilnetze getrennt trainiert und gespeichert werden können. So kann man zum Beispiel die besten Teilnetze zu einem neuen Netz zusammensetzen. Die Klasse `CNeuronicMultiNet` implementiert diese Funktionalität und verwaltet mehrere Netze (`CNeuronicNet`). Eine schematische Darstellung für ein Mutli-Netz findet sich in Abbildung 2.7. Die Teilnetze können über die Funktion `connect()` miteinander verbunden werden. Da die Multi-Netz-Klasse selbst von CNeuronicNet abgeleitet ist, ist es möglich, dass ein Teilnetz selbst wiederum ein Multi-Netz ist. Dadurch kann man komplexe hierarchische Strukturen von miteinander verknüpften Netzen erzeugen.

[5]Eine detaillierte Beschreibung des NET-Formates findet sich im SNNS-User-Manual [AZ+]. Neuronica unterstützt das Laden und Speichern aus dem SNNS-Format noch nicht vollständig. Details hierzu finden sich in der Klassendokumentation zu Neuronica.

[6]Das Pattern-Format wird ebenfalls im SNNS-Manual ([AZ+]) beschrieben. Neuronica implementiert eine Teilmenge dieses Formates.

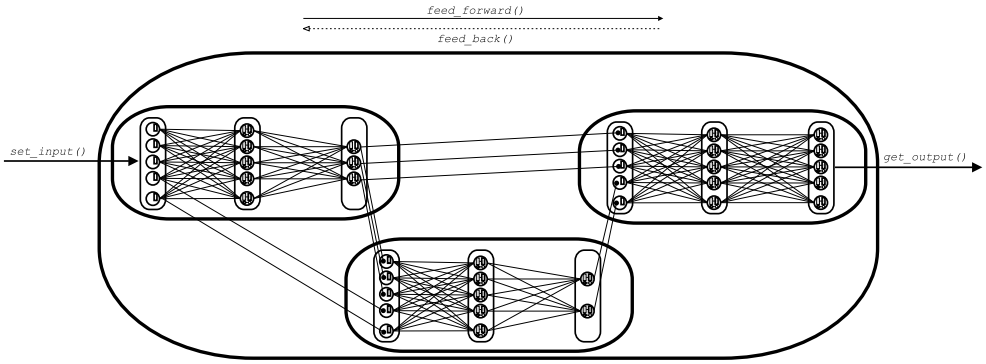


Abbildung 2.7: Beispiel eines Multi-Netzes (schematisch)

Eine Anforderung an die Zusammenschaltung mehrerer Netze ist es, die Teilnetze unverändert zu belassen. Es muss zum Beispiel möglich sein, diese getrennt (d.h. als einzelnes Netz) zu trainieren und abzuspeichern. Gleichzeitig soll das Prinzip der Lokalität erhalten bleiben, d.h. nach dem Verknüpfen der Netze muss die Erregungs- und Fehlerweiterleitung allein über das zeitlich korrekte Aufrufen der entsprechenden Funktionen in den einzelnen Neuronen zu erreichen sein. Zu diesem Zweck werden die bereits in Abschnitt 2.3.1 beschriebenen Input-Neuronen und ihre Fähigkeit andere Neuronen zu kapseln verwendet. Abbildung 2.8 zeigt eine schematische Darstellung dieses Prinzipes. Ein Netz kann also über die definierten Schnittstellen der Input-Neuronen

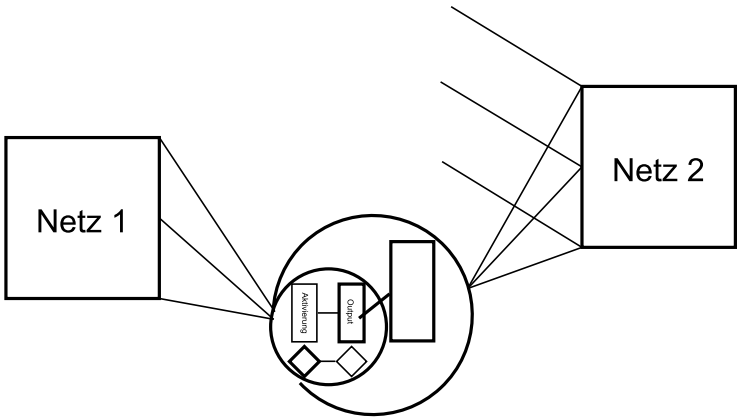


Abbildung 2.8: Schema der Kapselung eines Neurons durch ein Input-Neuron

mit einem anderen Netz verbunden werden. Dabei dient der Output der von den Input-Neuronen gekapselten Neuronen als Input für das Netz, in dem sich die Input-Neuronen befinden. Die Funktion `connect()` des Multi-Netzes sorgt für die nötige Kapselung.

2.3.6 Fazit

Die oben beschriebenen Einheiten bilden lediglich das Grundgerüst für die Modellierung bestimmter Netze. Von jeder einzelnen Klasse kann ein Nachkomme abgeleitet werden, der bestimmte weitere Funktionen implementiert, die in der Basisklasse nicht vorhanden sind oder das Verhalten der Basisklasse abändern. Für die Implementation eines neuen Netztypes ist dies sogar zwingend notwendig. Im Allgemeinen reicht es aus von den Neuronen bzw. Synapsen spezielle Klassen zu implementieren, um einen anderen Netztyp zu erzeugen. Ein Beispiel dafür wird im Abschnitt 2.4 detailliert beschrieben.

Da die verschiedenen Implementationen eines Neurons alle auf der gleichen Basisklasse aufbauen, können sie von außen her einheitlich angesprochen werden. Aus diesem Grund ist es oftmals möglich für verschiedene Neuronentypen (und Lernverfahren) die selben Klassen für Schicht und Netz zu verwenden.

Das hier beschriebene Modell hat den Vorteil, dass es sehr flexibel bezüglich Netzstruktur, eingesetzter Neuronen, Zusammenschaltung verschiedener Arten von Neuronen usw. ist. Von Nachteil ist die relativ geringe Ausführungsgeschwindigkeit und der Implementationsaufwand, der betrieben werden muss, um einen neuen Netztyp korrekt zu implementieren. Dieser ist vor allem deswegen höher, weil die Algorithmen für die meisten Netztypen auf der Ebene der Netze angegeben sind. Es wird oft eine Rechenvorschrift angegeben, wie die Ausgaben des Netzes aus den Inputs hervorgehen. Diese ist aber normalerweise nicht lokal für die einzelnen Neuronen, sondern global für das gesamte Netz vorgegeben. Damit das oben beschriebene Modell wie gewünscht funktioniert, ist es aber nötig, dass jedes Neuron seinen Rechenschritt unabhängig von den anderen Neuronen ausführt. (Ein Beispiel für eine solche Umwandlung einer globalen Regel in eine auf per-Neuron-Basis wird im Abschnitt 2.4.2 dargestellt.) Die dadurch erreichte Lokalität der Berechnung kann sogar ein Geschwindigkeitsvorteil sein, wenn das Programm auf einem entsprechenden Parallelrechner (oder spezieller Neuro-Hardware) eingesetzt wird. Dort ist es nämlich wichtig, dass eine Berechnung unabhängig von anderen erfolgen kann.

2.4 Das Backpropagation-Netz

Unter dem Begriff Backpropagation-Netz versteht man im Allgemeinen ein Feed-Forward-Netz, d.h. ein Netz in dem sich die Signale nur vorwärts von der Eingabe- zur Ausgabeschicht fortpflanzen, welches das Lernverfahren Backpropagation benutzt (siehe u.a. [Zel94],[Pat97],[Derb]).

2.4.1 Das Lernverfahren Backpropagation

Backpropagation ist ein Gradientenabstiegsverfahren. Mit diesem Verfahren wird versucht möglichst schnell ein globales Minimum der Fehlerfläche eines neuronalen Netzes zu finden, indem man sich orthogonal zum Gradienten in Richtung eines Minimums bewegt. Die Fehlerfläche ergibt sich aus der Fehlerfunktion

$$E(W) = E(w_1, \ ..., \ w_n) \tag{2.1}$$

, die den Fehler für das Neuronale Netz als Summe der Einzelfehler über alle Trainingsmuster bestimmt. Wobei $w_1, ..., w_n$ die Gewichte sind. Beim Backpropagation-Verfahren wird nun der Gradient der Fehlerfunktion berechnet, und alle Gewichte werden um einen Bruchteil des negativen Gradienten $-\nabla E(W)$ verändert. Für ein einzelnes Gewicht w_{ij} der Verbindung von Neuron i nach Neuron j gilt also:

$$\Delta w_{ij} = -\eta \frac{\partial}{\partial w_{ij}} E(W) \tag{2.2}$$

wobei η die Lernrate bezeichnet. Hieraus lässt sich unter Verwendung von $E(W) = \frac{1}{2}\sum_j (train_{pj} - out_{pj})^2$ als Fehlerfunktion eine konstruktive Regel, die Backpropagation-Regel, zur Veränderung der einzelnen Gewichte herleiten[7]. Diese erlaubt eine Berechnung der Gewichtsänderungen, beginnend mit der Ausgabeschicht. Für ein bestimmtes Muster p werden die Gewichte jeder Zelle des Netzes nach folgender Regel geändert:

$$\Delta_p w_{ij} = \eta out_{pi} \delta_{pj} \tag{2.3}$$

mit

$$out_{pi} = f_{act}(net_{pi}) \tag{2.4}$$

als Output des Neurons i und

$$\delta_{pj} = \begin{cases} f'_{act}(net_{pj})(train_{pj} - out_{pj}) & \text{falls } j \text{ Ausgabeneuron ist} \\ f'_{act}(net_{pj}) \sum_k \delta_{pk} w_{jk} & \text{falls } j \text{ verdecktes Neuron ist.} \end{cases} \tag{2.5}$$

Wobei $train_{pj}$ die Trainingsvorgabe, out_{pj} der Netz-Output eines Neurons j der Ausgabeschicht, f_{act} die Aktivierungsfunktion, sowie net_{pj} der Netz-Input (oder auch Aktivierung) eines Neurons j ist. Die Aktivierung berechnet sich dabei aus der Summe der Outputs aller Vorgängerneuronen eines Neurons multipliziert mit dem Gewicht der jeweiligen Verbindung:

$$net_{pj} = \sum_i out_{pi} w_{ij} \ . \tag{2.6}$$

Mit der Regel 2.3 lassen sich nun beginnend mit der Ausgabeschicht hin zu den Eingabezellen alle Gewichtsänderungen berechnen.

[7] Eine detailierte Herleitung findet man in [Zel94] oder in [Derb].

2.4.2 Die Implementation des Backprop-Neurons

Um das Backpropagation-Verfahren mit den oben beschriebenen Klassen zu modellieren, ist es nötig die Gleichungen 2.3 und 2.5 so umzuwandeln, dass sie mit dem lokalen Neuronenmodell aus Abschnitt 2.3.1 vereinbar sind. Weiterhin soll strukturell nicht zwischen verdeckten Neuronen und Ausgabeneuronen unterschieden werden.

Zunächst soll gezeigt werden, dass die Standard-Neuronen zusammen mit den Standard-Synapsen bereits die Feed-Forward-Funktionalität des Backpropagation-Verfahrens implementieren können: Angenommen die Neuronen wurden, wie in Abschnitt 2.3.1 bis 2.3.4 beschrieben, zu einem Feed-Forward-Netz verbunden. Das heisst das Netz besteht aus mehreren nacheinander liegenden Schichten, und jedes Neuron einer Schicht ist mit allen Neuronen der direkt nachfolgenden Schicht verbunden. Die Gewichte wurden dabei mit zufälligen Werten initialisiert.

Dem Netz wird nun das Muster p als Input präsentiert und bis zur Ausgabeschicht vorwärts propagiert. Wie in Abschnitt 2.3.1 beschrieben wird dazu die Funktion `feed_forward()` verwendet. Dabei wird die richtige Reihenfolge beachtet, zuerst wird die Funktion für alle Neuronen der ersten Schicht, dann für alle der zweiten Schicht usw. aufgerufen. Für alle Neuronen der Input-Schicht ist der Output vorgegeben und entspricht einem Element des Musters p. Angenommen, für alle Neuronen j einer Schicht m ist zum Zeitpunkt t die Aktivierung net_{pj} so gesetzt, dass die Gleichung 2.6 erfüllt ist. Wird nun für alle Neuronen dieser Schicht die Funktion `feed_forward()` aufgerufen, so wird für jedes Neuron der Schicht der Output über die eingestellte Aktivierungsfunktion gemäß Gleichung 2.4 berechnet. Dieser Output wird für jedes Neuron an alle Output-Synapsen und damit bei dem angenommenen Aufbau des Netzes an alle Neuronen der nachfolgenden Schicht $m + 1$ weitergeleitet. Dabei wird der Output bei Verwendung der Standard-Synapse mit dem Gewicht multipliziert und das Ergebnis dieser Operation zur Aktivierung des postsynaptischen Neurons hinzugefügt. Man kann also einfach sehen, dass wenn alle Aktivierungen der Neuronen der Schicht $m + 1$ zum Zeitpunkt t den Wert 0 hatten, jedes Neuron der Schicht $m + 1$ zum Zeitpunkt $t + 1$[8] eine Aktivierung gemäß Gleichung 2.6 besitzt. Dies gilt, da sämtliche Vorgängerneuronen eines jeden Neurons der Schicht $m + 1$ in der Schicht m liegen.

Nun soll erreicht werden, dass zusätzlich die Gewichte gemäß der Backpropagation-Regel abgeändert werden. Dazu wird die Klasse CBackpropNeuron von CNeuron abgeleitet. Dabei werden die Funktionen `feed_back()` und `add_error_activation()` der Basisklasse überschrieben. Sei nun $erract_j$ eine neue Variable, bezeichnet als die Fehleraktivierung des Neurons j. `Feed_back()` berechnet aus der Fehleraktivierung (wobei diese auf 0 zurück gesetzt wird) den zurückzupropagierenden Fehler δ_{pj} eines

[8] Dabei sei $t + 1$ der Zeitpunkt, nachdem für alle Neuronen der Schicht m die `feed_forward()`-Funktion aufgerufen wurde.

jeden Neurons j nach folgender Gleichung:

$$\delta_{pj} = f'_{act}(net_{pj})erract_j. \tag{2.7}$$

Danach wird δ_{pj} an alle Vorgängerneuronen des Neurons j weiter gereicht. Hierzu wird für alle Vorgängerneuronen i des Neurons j die Funktion `add_error_activation()`, die den ihr übergebenen Wert der Fehleraktivierung hinzuaddiert, mit $\delta_{pj}w_{ij}$ als Parameter aufgerufen. (Wobei w_{ij} das Gewicht der Verbindung von Neuron i nach Neuron j ist.) Als letzter Schritt werden die Gewichte der Verbindungen von den Vorgängerneuronen zum Neuron j gemäß Gleichung 2.3 angepasst.

Setzt man nun für jedes Neuron der Ausgabeschicht $erract_{pj} = (train_{pj} - out_{pj})$ und ruft dann beginnend bei der Ausgabeschicht für jedes Neuron die Funktion `feed_back()` auf, wobei die Neuronen in einer solchen Reihenfolge aufgerufen werden müssen, dass immer gilt,

$$\text{Feedback für Neuron } k \text{ wird vor Neuron } l \text{ aufgerufen, wenn } w_{lk} \text{ existiert,} \tag{2.8}$$

so erhält man, unter der Vorraussetzung, dass vor Beginn des Zurückpropagierens die Fehleraktivierungen aller Neuronen 0 waren, eine Anpassung der Gewichte nach der in Abschnitt 2.4 beschriebenen Backpropagation-Regel[9]. Dass dies tatsächlich der Fall ist, folgt daher, weil die Gewichte einer Zelle j nach Gleichung 2.3 angepasst werden. Es verbleibt also zu zeigen, dass die Gleichungen 2.5 und 2.7 äquivalent sind. Dazu genügt es zu zeigen, dass

$$erract_{pj} = (train_{pj} - out_{pj}), \tag{2.9}$$

falls j ein Neuron der Ausgabeschicht ist, was trivialerweise gegeben ist, und dass

$$erract_{pj} = \sum_k \delta_{pk} w_{jk} \tag{2.10}$$

für alle übrigen Neuronen gilt. Das ist aber der Fall, da, wenn die Funktion `feed_back()` für Neuron j aufgerufen wird, nach Regel 2.8 diese Funktion (und damit auch die Funktion `add_error_activation()`) schon für alle Neuronen k mit w_{jk} aufgerufen wurde. Unter der oben getroffenen Voraussetzung, dass $erract_{pj}$ zu Beginn gleich 0 ist, gilt damit auch Gleichung 2.10.

Das Backprop-Neuron (siehe auch Abbildung 2.9) entspricht in wesentlichen Punkten dem Basis-Neuron aus Abschnitt 2.3.1. Es wurden zusätzlich Funktionen zur Fehlerweiterleitung und zur Änderung der Gewichte implementiert.

Im folgenden Abschnitt sollen einige Erweiterungen des Backpropagation-Verfahrens und die Möglichkeiten der Implementation untersucht werden.

[9] Es gelte, w_{lk} existiert genau dann, wenn Neuron l Vorgängerneuron von Neuron k ist.
Dies impliziert mit Bedingung 2.8, dass in dem Netz kein Zyklus enthalten sein darf, da die Bedingung sonst nicht erfüllbar ist.

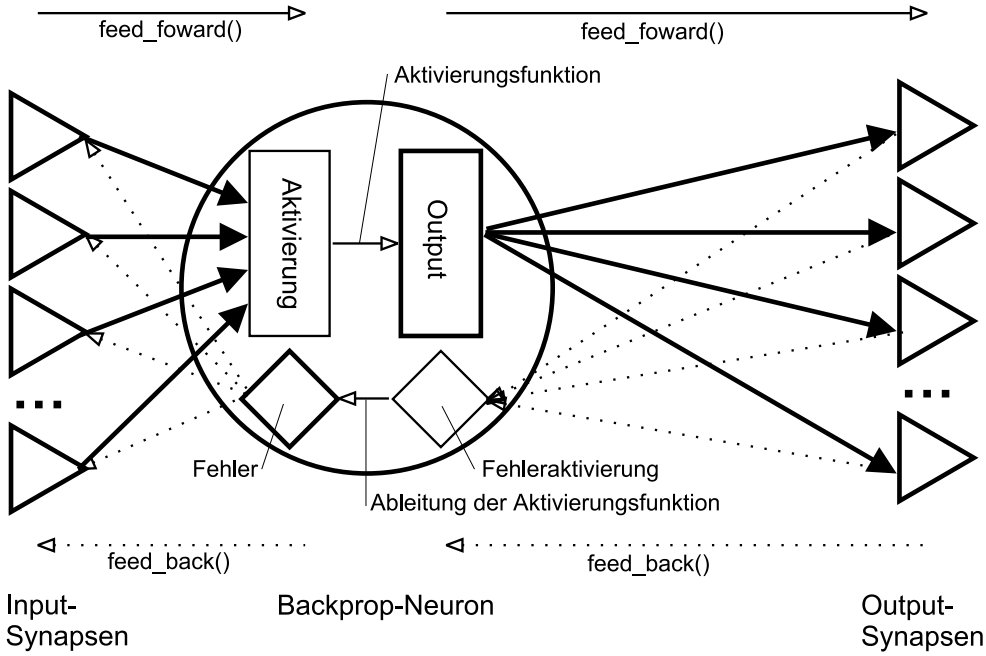


Abbildung 2.9: Schematische Darstellung des Backprop-Neurons

2.4.3 Erweiterungen von Backpropagation

Backpropagation mit Shortcut-Connections

Shortcut-Connections verbinden Neuronen nicht direkt benachbarter Schichten miteinander. Ein entsprechendes Netzwerk lässt sich einfach durch das entsprechende Verbinden der Neuronen realisieren. Voraussetzung für ein korrektes Funktionieren ist das Aufrufen der Funktionen `feed_forward()` und `feed_back()` in der richtigen Reihenfolge wie in Abschnitt 2.4.2 beschrieben. Die Betrachtung zur funktionellen Äquivalenz aus Abschnitt 2.4.2 lässt sich dabei einfach auch auf Netze mit Shortcut-Connections ausweiten.

An der Implementierung des Backprop-Neurons muss nichts geändert werden. Dieses Verfahren funktioniert mit den bereits beschriebenen Klassen.

Backpropagation mit Momentum-Term

Ein Momentum-Term vermeidet einige Probleme von Backpropagation bezüglich flacher Plateaus und steiler Schluchten der dem Gradientenabstiegsverfahren zu Grunde liegenenden Fehlerfunktion. Dabei erfolgt die Gewichtsänderung ähnlich dem Standard-Backpropagation-Verfahren allerdings unter zusätzlicher Beachtung der vorhergehenden Gewichtsänderung (zum Zeitpunkt $t-1$):

$$\Delta_p w_{ij}(t) = \eta out_{pi}\delta_{pj} + \alpha\Delta_p w_{ij}(t-1) \tag{2.11}$$

In dieser Gleichung ist α der Glättungskoeffizient ($\alpha \in [0..1]$), der den Einfluss des Momentum-Terms bestimmt. Im Neuronica-Netzwerk kann man dies implementieren[10], indem man die Neuronen nicht über die Standard-Synapsen verbindet, sondern eine neue Synapse erstellt, die bei einer Änderung des Gewichtes die letzte Gewichtsänderung mit beachtet und die Gewichte nur gemäß der Formel 2.11 abändert. Dazu muss die jeweils letzte Gewichtsänderung in den Synapsen zwischengespeichert werden.

CStochasticBackpropNeuron

In der Klasse `CStochasticBackpropNeuron` (abgeleitet von `CBackpropNeuron`) ist ein Backpropagation-Neuron, wie in Abschnitt 2.4.2 beschrieben, implementiert. Das Neuron besitzt aber einen zusätzlichen Parameter $temperature$. Wenn $temperature$ den Wert 0 hat, verhält sich das Neuron identisch zu dem Backpropagation-Neuron. Für Werte größer als 0 wird der Output des Neurons durch einen zufälligen Parameter mitbestimmt. Der Einfluss des zufälligen Parameters nimmt hierbei mit steigender Temperatur zu. In der aktuellen Implementierung wird die Gauß'sche Normalverteilung benutzt, um den zufälligen Einfluss zu erzeugen. Für $temperature = 0$ wird der vorläufige Output $vout$, der wie der Output out beim Backpropagation-Neuron berechnet wird, als Ergebnis geliefert. Für $temperature > 0$ wird der Output des Neurons durch die Gauß'sche Normalverteilung

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

mit $\mu = vout$ als Erwartungswert und $\sigma = temperature$ als Varianz bestimmt. Sollte ein Wert x gezogen werden, der außerhalb des Intervalles $[min, max]$[11] liegt, so wird erneut gezogen, diesmal aber eine gleichverteilte Zufallszahl aus dem Intervall $[min, max]$. Die Normalverteilung wird nach dem in [Knu68] beschriebenen Verfahren aus 2 gleichverteilten Zufallszahlen errechnet.

Diese Klasse von Neuronen wird z.B. von dem in Abschnitt 3.2.1 beschriebenen Lernverfahren verwendet.

[10] dieses Verfahren ist in der Klassenbibliothek zur Zeit noch nicht implementiert

[11] Das Intervall ist durch die Output-Werte bestimmt, die das Neuron annehmen darf. Für unterschiedliche Aktivierungsfunktionen sind hier z.B. verschiedene Werte möglich. Mit tanh als Aktivierungsfunktion würde man beispielsweise $[-1, 1]$ wählen.

Weitere Erweiterungen

Folgende Erweiterungen zum Standard-Backpropagation-Verfahren sind außerdem mit den in den Neuronica-Netzwerkklassen implementierten Funktionen möglich:

- automatisches Abkühlen der Lernrate
- individuelle Lernrate für jedes einzelne Neuron
- Steilheit des Anstiegs der Aktivierungsfunktion frei einstellbar, unterschiedliche Werte für verschiedene Neuronen wählbar

Es ist außerdem möglich verschiedene Neuronen bzw. Synapsenarten in einem Netz zu mischen. Zum Beispiel könnte man den Momentum-Term nur für bestimmte Neuronen im Netz verwenden oder andere Neuronen mit speziellen Sonderfunktionen in ein Netz mit aufnehmen.

2.5 Selbstorganisierende Karten

2.5.1 Prinzip und Lernverfahren selbstorganisierender Karten

Selbstorganisierende Karten (kurz SOMs vom engl. *self-organizing maps*) werden benutzt, um mit einem unüberwachten Lernverfahren, Cluster in den Eingabedaten zu erkennen und neue Eingabedaten entsprechend der Cluster zuzuordnen. Dabei erfolgt eine Topologie erhaltende Abbildung der mehrdimensionalen Eingabevektoren in das m-dimensionale Gitter der Neuronen. Das Verfahren geht auf Kohonen [Koh84, Koh95] zurück. Die Ausarbeitung innerhalb dieser Arbeit orientiert sich allerdings an der Darstellung in [Zel94]. Beim Lernverfahren der selbstorganisierenden Karten wird der n-dimensionale Eingabevektor X mit allen Gewichtsvektoren $W_i = (w_{1i}, ..., w_{ni})$

verglichen. Das Neuron c, dessen Gewichtsvektor W_c am ähnlichsten zum Eingabevektor X ist, ist der Gewinner. Als Maß für die Ähnlichkeit kann eine beliebige Norm über der Differenz der Vektoren verwendet werden, so dass man schreiben kann: Gewinner ist das Neuron c für welches gilt:

$$\|X - W_c\|_1 = min_j \|X - W_j\|_1 \tag{2.12}$$

Die Gewichte des Gewinnerneurons werden nun so adaptiert, dass der Gewichtsvektor W_i auf den Eingabe-Vektor zu bewegt wird. Diese Änderung der Gewichte wird nicht nur für das Gewinnerneuron sondern auch für alle zum Gewinnerneuron topologisch benachbarte Neuronen durchgeführt. Ob und wie stark jeweils ein anderes Neuron mit lernt, wird durch die Nachbarschaftsfunktion h bestimmt.

2.5.2 Implementation der Funktionalität von SOMs in Neuronica

Für das hier beschriebene Lernverfahren ist es nötig mehrere Neuronen zu vergleichen und den Gewinner zu ermitteln. Das ist auf der Ebene der Neuronen selbst nicht effizient möglich. Daher wird für dieses Verfahren die von `CNeuronicLayer` abgeleitete Klasse `CKohonenLayer` benutzt, welche diese Funktionalität implementiert. Zum Vergleich des Abstandes wird zusätzlich eine spezielle Synapse eingeführt.

Die Kohonen-Synapse[12]

Die Implementation benutzt eine spezielle Synapse (im Folgenden Kohonen-Synapse genannt - Klasse `CKohonenSynapse`). Diese liefert als postsynaptischen Wert den Betrag der Differenz aus präsynaptischen Wert und Gewicht der Synapse.

Das Kohonen-Neuron

In der Klasse `CKohonenNeuron` (abgeleitet von `CNeuron`) ist das Neuron implementiert, welches für das Lernverfahren der selbstorganisierenden Karten verwendet wird. Ein Kohonen-Neuron ist mit seinen Vorgänger-Neuronen über Kohonen-Synapsen verbunden. Der Abstand des Gewichtsvektors W_i eines Neurons i zum Eingabevektor X wird nicht über die euklidische Norm, sondern über folgende Norm bestimmt: $\|Y\|_1 = |\sum_{i=1}^{n} y_i|$ mit $Y = (y_1, ..., y_n)$. Durch die Wahl der Kohonen-Synapse als Input-Synapse ergibt sich als Aktivierung eines Neurons i genau der Abstand des Eingabevektors zum Gewichtsvektor

$$activation = \|X - W_i\|_1 \tag{2.13}$$

gemäß der verwendeten Norm. Ein Kohonen-Neuron besitzt ferner die Funktion `learn_step()`. Diese sorgt für die Adaption der Gewichtsvektors in Richtung des zu übergebenden Vektors X. Ein zweiter Parameter bestimmt die Stärke der Anpassung.

Der Output des Neurons ist definiert mit 1, falls das Neuron der Gewinner ist und mit 0 sonst. Für alle weiteren Funktionen ist das Kohonen-Neuron mit dem Standard-Neuron identisch.

Die Kohonen-Schicht

Über die von `CNeuronicLayer` abgeleitete Klasse `CKohonenLayer` wird die Topologie abhängige Funktionalität einer selbstorganisierenden Karte implementiert. Die

[12] Die Namensgebung bezieht sich auf `kohonen maps` eine ebenfalls übliche Bezeichnung für selbstorganisierende Karten und spiegelt die Namensgebung innerhalb der Neuronica-Bibliothek wider.

Funktionen `feed_forward()` und `feed_back()` von `CNeuronicLayer` werden überschrieben und durch eigene Varianten ersetzt.

Beim Feed-Forward-Schritt wird zunächst das Gewinnerneuron der Schicht ermittelt und als Gewinner gekennzeichnet. Das Gewinnerneuron ist gemäß Gleichung 2.12 das Neuron mit der niedrigsten Aktivierung. Die übrige Verarbeitung erfolgt wie in der Basisklasse.

Beim Feed-Back-Schritt wird zuerst der letzte Input-Vektor X des Gewinner-Neurons (entspricht dem der übrigen Neuronen der Schicht) ermittelt. Für jedes Kohonen-Neuron der Schicht wird die Funktion `learn_step()` mit X und h_{cj} als Parametern aufgerufen. h_{cj} ist der Wert der Nachbarschaftsfunktion des Gewinners für das Neuron j und basiert auf dem Abstand des jeweiligen Neurons zum Gewinner. Der Abstand wird durch die logische Topologie der Kohonenschicht und der Position der jeweiligen Neuronen bestimmt. In der implementierten Klasse sind 1-, 2- und 3-dimensionale Gitter erlaubt[13], eine Erweiterung auf ein m-dimensionales Gitter ist aber einfach möglich.

Wenn also für die Schicht zuerst `feed_forward()` und danach `feed_back()` aufgerufen wird, ergibt sich das in Abschnitt 2.5.1 beschriebene Verfahren.

2.6 Anbindung an den SNNS

2.6.1 Prinzip der Anbindung

Die Neuronica-Netzwerkklassen erlauben es auf die Funktionalität des Stuttgarter Neuronale-Netzwerk-Simulator (SNNS) zurückzugreifen. Damit ist es möglich innerhalb der Neuronica-Klassenstruktur über den SNNS-Kernel ein Netz zu trainieren und zu evaluieren. Dabei kann auf die Lernverfahren des SNNS zugegriffen werden. Eine ausführliche Beschreibung des SNNS findet sich im SNNS-Benutzerhandbuch [AZ$^+$]. Der SNNS besteht aus einem Simulatorkernel und aus einer auf dem Kernel aufbauenden Benutzeroberfläche. Der Kernel ist ausführlich dokumentiert und erlaubt den Zugriff auf die Funktionalität des SNNS. Der SNNS ist allerdings nicht zur Laufzeitverarbeitung von Daten geschaffen worden, sondern die Prämisse liegt auf dem Training von Netzen mit bereits vorhandenen Daten. Es existiert zwar ein zum SNNS gehörendes Werkzeug, das zum Einsatz als Laufzeitversion geschaffen wurde, der Netzwerk-Compiler "SNNS2C" [Zel94]. Dieser Compiler erhält als Eingabe ein trainiertes Netz im SNNS-Format und erzeugt daraus ein C-Programm, welches die Arbeitsphase des Netzwerkes ausführt. Das Trainieren und Lernen von Netzen ist mit dem erzeugten Code aber nicht möglich. Daher wird bei der Anbindung von Neuronica an den SNNS direkt auf den Simulator-Kernel zugegriffen.

[13]Die logische Topologie wird lediglich durch die Funktion get_abstand() festgelegt. Die Verwaltung der einzelnen Neuronen erfolgt wie in der Basisklasse über einen Vektor.

2.6.2 Die Klasse CNeuronicSnnsNet

Über die Klasse `CNeuronicSnnsNet` wird die Anbindung an den SNNS realisiert. Die Klasse wurde von `CNeuronicNet` abgeleitet und ist prinzipiell wie jedes andere Netz nutzbar. Das SNNS-Netz besitzt, wie andere Netze auch, eine Input-Schicht und eine Output-Schicht, bestehend aus einzelnen Neuronen. Diese Schichten werden beim Laden eines SNNS-Netzes automatisch angelegt und dienen lediglich dazu, den Input und Output des Netzes zwischenzuspeichern und einen konsistenten Zugriff von außen anzubieten. Soll zu dem anliegenden Input ein Output erzeugt werden, d.h. die Funktion `feed_forward()` des Netzes wurde aufgerufen, so werden die Werte der Input-Neuronen ausgelesen und über die SNNS-Kernel-Funktionen in den Kernel geschrieben. Danach wird die SNNS-Kernel-Prozedur aufgerufen, die ein Netz-Update mit den eingestellten Parametern durchführt. Das erzeugte Output wird aus dem SNNS ausgelesen und in die Output-Schicht übertragen. Abbildung 2.10 zeigt eine schematische Darstellung des Netzes.

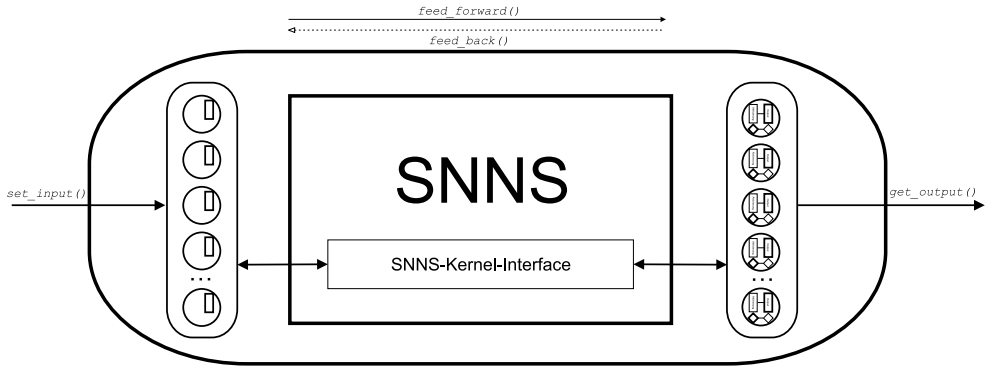


Abbildung 2.10: Modell eines SNNS-basierten-Netzes

Wenn ein Fehler zurückpropagiert werden soll (`feed_back()`), so wird ähnlich wie eben beschrieben verfahren. Allerdings unterstützt der SNNS kein explizites Zurückpropagieren, sondern lediglich das Durchführen eines Lernschrittes, wenn Input und zugehöriges Soll-Output gegeben sind. Um einen Fehler zurückzupropagieren wird aus dem Fehler und dem anliegenden Output berechnet, welcher Output hätte anliegen müssen, um den entsprechenden Fehler zu erzeugen. Mit diesem berechneten Output und dem anliegenden Input wird dann ein Lernschritt durchgeführt um eine entsprechende Aktualisierung des Netzes zu erreichen.

Der Zugriff auf den SNNS und der unterschiedliche Aufbau der beiden Simulatoren bedingen einige Einschränkungen bei der Benutzung der Klasse. So funktioniert das oben beschriebene Verfahren sicherlich nur mit Netzen, die nach dem Feed-Forward-Prinzip funktionieren. Ein Erzeugen des Netzes zur Laufzeit ist momentan (noch) nicht möglich, das Netz muss aus einer SNNS-NET-Datei geladen werden. Anders als bei einem normalen Neuronica-Netz existiert kein Feedback-Signal für die Input-Neuronen.

Von Bedeutung ist dies, wenn mehrere Netze zusammengeschaltet werden sollen. Der Zugriff auf einzelne Schichten und Neuronen des Netzes kann nicht über die Standardfunktionen der Netzwerk-Bibliothek ausgeführt werden.

Die Funktionalität der Klasse `CNeuronicSnnsNet` wurde bisher nur mit den SNNS-Lernverfahren `Std_Backpropagation`, `BackpropMomentum` und `Backpercolation` erfolgreich getestet.

2.7 Evaluation der Funktionalität der Netzwerkklassen

In diesem Kapitel wird die Funktionalität der Netzklassen untersucht und unter anderem mit dem SNNS als Referenzsystem verglichen.

2.7.1 Vergleich der Funktionalität in Bezug auf den SNNS

Die folgende Übersicht vergleicht die Funktionalität zwischen den Neuronica Netz-Klassen und dem SNNS-Paket.

Vergleich der Funktionalität zwischen Neuronica und SNNS		
Funktion	**Neuronica**	**SNNS**
Online-Training	problemlos möglich	wird nicht direkt unterstützt, über das Kernel-Interface nur eingeschränkt möglich
graphische Darstellung der Netze / des Trainingsergebnisses	nicht direkt möglich, aber Nutzung des SNNS als Netz-Betrachter	möglich
Laden und Speichern von Netzen	im SNNS-NET Format (nur für Backprop-Netze implementiert) sowie im (nativen) XML-basierten NND-Format	im (nativen) SNNS-NET-Format
Kombination unterschiedlicher Netztypen	Kombination verschiedener Netztypen sowie verschiedener Lernverfahren möglich	nur beschränkt möglich, da immer das gesamte Netz mit einem festgelegten Verfahren trainiert wird
Einbindung in C++ - Programme	übersichtliche Klassenstruktur, einfach möglich	relativ kompliziert

Vergleich zwischen Neuronica und SNNS		
Funktion	**Neuronica**	**SNNS**
Erweiterbarkeit der Funktionalität	auf Grund der verwendeten Klassenstruktur einfach möglich	relativ kompliziert
Geschwindigkeit	eher niedrig	eher hoch
Bedienbarkeit	keine graphische Benutzeroberfläche	graphische Benutzeroberfläche
Fokus	auf Online-Verarbeitung von Daten ausgerichtet, zur Einbindung in bestehende Software, Flexibilität steht im Vordergrund	auf Auswertung bereits gesammelter Daten ausgerichtet, primärer Fokus auf der Anwendung durch den Endnutzer Geschwindigkeit steht im Vordergrund
Anzahl der implementierten Netze / Lernverfahren	eher klein, aber Einbindung von SNNS-Lernverfahren möglich	groß

2.7.2 Training im Vergleich zum SNNS

Um die Funktionalität der über die Neuronica-Netzwerkklassen implementierten Lernverfahren zu testen, wurde ein bestimmtes Netz einmal über den SNNS und ein anderes Mal über ein Neuronica-Backprop-Netz trainiert. Dabei hatten die beiden Netze den gleichen Aufbau (2 Input-Neuronen, 2 versteckte Neuronen und 1 Ausgabeneuron). Die Gewichte beider Netze wurden auf die gleichen Start-Werte eingestellt. Beide Netze verwenden die gleiche Aktivierungsfunktion ($tanh$) und die gleiche Lernrate (0.5). Als Lernverfahren wurde das Standard-Backpropagation-Verfahren gewählt. Die Netze wurden darauf trainiert die XOR-Funktion zu erlernen. Für jedes der Netze wurden dabei 1000 Lern-Schritte ausgeführt. Nach vollzogenem Lernen wird für jedes der Netze, für die 4 Ausgangswerte, das Netzergebnis angezeigt. Das zugehörige Programm findet sich in Abschnitt B.1. Die Ausgabe dieses Programms ist hier im Folgenden dargestellt:

```
Einstellen der gleichen Lernrate (0.5) für beide Netze:
BP - Training   .........done
SNNS - Training .........done

Lernen liefert:
1. Fall BP   (      0,      0): -0.0158614
1. Fall SNNS (      0,      0): -0.0159014
2. Fall BP   (      0,      1): 0.986772
```

```
2. Fall SNNS (      0,      1): 0.986772
3. Fall BP   (      1,      0): 0.986815
3. Fall SNNS (      1,      0): 0.986816
4. Fall BP   (      1,      1): -0.00964981
4. Fall SNNS (      1,      1): -0.0096072
```

Das bedeutet Neuronica und der SNNS verhalten sich (zumindest in Bezug auf das Lernen mit Standard-Backpropagation) nahezu identisch. Die weitestgehende Übereinstimmung findet sich nicht nur nach Abschluss des Trainings sondern während des gesamten Trainingslaufes[14]. Die kleine Differenz zwischen den Werten kann über Rundungsfehler, die sich durch die unterschiedliche Art der Berechnung ergeben, erklärt werden.

2.7.3 Trainieren der SNNS-Beispiel-Netze

Das Trainieren einiger mit dem SNNS mitgelieferter Beispiel-Netze wurde erfolgreich getestet. Die Ergebnisse des Trainings lagen im Rahmen der durch die Dokumentation zu den SNNS-Pattern-Dateien vorgegebenen Werte.

Das Training des SNNS-Pattern-Files *`letters_untrained.pat`* zur Buchstabenerkennung ist Teil des mit Neuronica gelieferten Beispielprogrammes.

2.8 Resume

Die im Rahmen dieser Diplomarbeit entwickelte Klassenbibliothek Neuronica erlaubt es, wie beabsichtigt, künstliche neuronale Netze einfach in bestehende oder neu zu entwickelnde Programme einzubinden. Die Bibliothek im aktuellen Entwicklungsstand unterstützt dabei bereits verschiedene Backpropagation-Lernverfahren und selbstorganisierende Karten. Aufgrund des modularen Aufbaus ist eine Erweiterung für weitere Lernverfahren einfach realisierbar. Das Paradigma, die eigentliche Funktionalität soweit wie machbar auf einer möglichst niedrigen (lokalen) Ebene zu realisieren, wurde bei den implementierten Verfahren beachtet. Das heißt, die Lernverfahren funktionieren nicht nach dem klassischen Top-Down-Ansatz, bei dem der Algorithmus im Zentrum steht (prozeduraler Ansatz), sondern nach dem Bottom-Up-Prinzip mit dem Neuron als Basis-Baustein (objektorientierter Ansatz). Der eigentliche Algorithmus ergibt sich dabei aus der Summe der Funktionalität der einzelnen Komponenten. Für Backpropagation und selbstorganisierende Karten wurde gezeigt, dass das realisierte Prinzip

[14] In den Beispielprogrammen zur Neuronica-Klassenbibliothek sind auch ausführlichere Tests implementiert.

tatsächlich die selbe Funktionalität erzeugt, wie die in der Literatur beschriebenen Algorithmen.

Bei der praktischen Umsetzung wurde darauf geachtet soweit wie möglich mit bestehenden Werkzeugen kompatibel zu bleiben. Als Dateiformat zur Abspeicherung und zum Laden von Mustern (Pattern-Datei) wurde das vom SNNS bekannte Format benutzt[15]. Zum Laden und Speichern der Netze wird ein eigenes XML-basiertes Format benutzt, da keines der bestehenden untersuchten Formate in der Lage war alle relevanten Informationen aufzunehmen. Der Export in das SNNS-Netz-Format ist aber direkt durchführbar. Da für den SNNS viele nützliche Werkzeuge zur Verfügung stehen, sind diese damit auch mit den über Neuronica erzeugten Dateien verwendbar. Dies hat sich insbesondere bei der Visualisierung von mit Neuronica erzeugten und trainierten Netzen als nützlich erwiesen.

Die über Neuronica bereitgestellten künstlichen neuronalen Netze wurden ausgiebig auf ihre Funktionalität getestet. Als Referenz wurde dabei der Stuttgarter Neuronale Netzwerk-Simulator (SNNS) verwendet. Es wurde gezeigt, dass SNNS und Neuronica, wenn die gleichen Lernparameter eingestellt werden, bis auf Rundungsfehler zu exakt den gleichen Ergebnissen führen. Weiterhin wurden einige Probleme (bzw. die sie repräsentierenden Pattern-Dateien) aus den mit dem SNNS gelieferten Beispielen ausgewählt und erfolgreich mit Neuronica getestet. Damit wurde gezeigt, dass die Bibliothek für die klassischen Benchmark-Probleme (wie z.B. XOR-Netz) wie gewünscht funktioniert. Die Netz-Bibliothek wurde ferner als Grundlage der in Kapitel 3 beschriebenen Experimente benutzt und dabei ebenfalls ausgiebig und praxisnah getestet.

Das modulare Prinzip der Netzwerk-Bibliothek erlaubt eine einfach zu realisierende Einbindung der künstlichen neuronalen Netze. Durch die Möglichkeit einzelne Neuronen beliebig zusammen zu schalten und insbesondere dadurch, mehrere Netze jeweils zu einem großen Netz zu verbinden, lassen sich theoretisch beliebig komplexe Strukturen aufbauen. Die Funktionalität, Teilnetze getrennt zu trainieren und abzuspeichern, hat sich bei der praktischen Anwendung als besonders nützlich erwiesen. Durch das Paradigma, die Lernverfahren soweit wie möglich auf der (niederen) Ebene der Neuronen zu implementieren, ergeben sich weitere Vorteile. Zum einen erlaubt es diese Art der Implementation, auf bereits bestehende Grundfunktionalitäten aufzubauen und neue Verfahren und Strukturen mit relativ niedrigem Aufwand zu entwickeln und zu testen. Zum anderen ist es dadurch einfach heterogene Netze zu erstellen. Die Heterogenität kann sich dadurch ergeben, dass sich für jedes Neuron die einzelnen Parameter (Lernrate, Steilheit des Anstiegs der Aktivierungsfunktion usw.) individuell einstellen und verändern lassen, und dass es möglich ist, gezielt verschiedene Arten von Neuronen in einem Netz zu kombinieren. Ein Beispiel dafür ist die Aufnahme von `CStochastic-BackpropNeuron` (statt `CBackpropNeuron`) an einigen Stellen des Netzes[16]. Es ist

[15] Neuronica implementiert eine für die in der Bibliothek verwendeten Verfahren relevante Teilmenge des SNNS-Formates

[16] In Abschnitt 3.2.1 ist z.B. eine Anwendung beschrieben, die diese Ersetzung benutzt.

auch denkbar Neuronen, die mit verschiedenen Verfahren trainieren, in einem Netz zu kombinieren, z.B. Neuronen die nach dem Standard-Backpropagation- Verfahren trainieren und Neuronen die Quickprop (siehe [Zel94]) verwenden.

Insgesamt haben sich die Klassen der Neuronica-Netzwerk-Bibliothek als nützliche Werkzeuge bei der Anwendung neuronaler Netze erwiesen. Insbesondere beim Online-Training und bei der Implementation neuer Lernverfahren bestehen Vorteile gegenüber klassischen Tools (wie z.B. SNNS). Nachteile in der Ausführungsgeschwindigkeit können unter Umständen durch den zusätzlichen Verwaltungsaufwand, der sich durch die objektorientierte Implementierung ergibt, auftreten. Dieser Nachteil könnte aber durch eine Anpassung der Ausführung auf spezieller Hardware, die eine massive Parallelverarbeitung unterstützt, insbesondere bei großen Netzen durch die bessere Parallelisierbarkeit der implementierten Algorithmen mehr als ausgeglichen werden.

Kapitel 3

Steuerungsmechanismen für autonome Roboter

3.1 Einführung

3.1.1 Was ist ein autonomer Roboter

Als autonom bezeichnet man einen Roboter, der die Fähigkeit besitzt, in einer *a priori* unbekannten Umwelt selbständig zu agieren. Dabei wird im Allgemeinen ein bestimmtes Ziel verfolgt, dass durch die Aktionen des Roboters erreicht werden soll. Während der Interaktion mit der Umwelt baut der Roboter Wissen über die Umwelt und gegebenfalls über sich und seinen Einfluss auf die Umwelt auf.[1] Soll ein autonomer Roboter in einer natürlichen Umgebung agieren, so ergeben sich weitere Anforderungen. Eine natürliche Umwelt ist normalerweise ein dynamisches System, der Roboter muss befähigt sein, sich an eine ändernde Umgebung anzupassen. Dazu muss er lernfähig sein, d.h. sein Wissen erweitern können. Dabei genügt es in Ausnahmesituationen oftmals nicht, auf vorprogrammierte Verhaltensmuster zurückgreifen zu können, sondern der Roboter sollte möglichst selbständig neue Verhaltensmuster entwickeln, die diese Situationen zu bewältigen helfen. Ein weiteres Problem in einer natürlichen Umgebung besteht darin, dass der Roboter die Informationen über die Umwelt nur durch seine Sensoren erlangen kann. Diese können auch ungenau, defekt oder gestört sein. Ein autonomer Roboter sollte möglichst robust sein bezüglich dieser Fehlerquellen.

[1] Der Begriff Umwelt soll im Folgenden sowohl die eigentliche Umwelt selbst, als auch den Bereich des Roboters abdecken, für den nur Informationen über Sensoren zugänglich sind (also den "Körper").

3.1.2 Aufbau eines Roboters

Ein Roboter besteht im Allgemeinen aus folgenden 3 Komponenten: der Sensorkomponente, der Informationsverarbeitungskomponente und der Ausführungskomponente. Die Sensorkomponente liefert eine Beschreibung der Umwelt und stellt diese Daten der Informationsverarbeitungskomponente zur Verfügung. Diese hat 2 Aufgaben, zum einen den Aufbau von Wissen über die Umwelt (Aufbau eines Weltmodells) und zum anderen die Planung der weiteren Aktionen auf Basis der Sensordaten und des Weltmodells. Die Ausführungskomponente sorgt letztendlich für die Umsetzung der während der Planung festgelegten Aktionen.

Der Khepera-Roboter

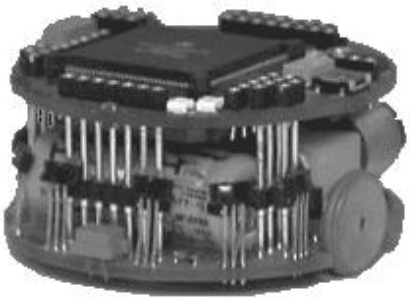
Bei allen für diese Arbeit durchgführten Versuchen wurde der Khepera-Roboter (bzw. ein Simulator für diesen Roboter) verwendet. Daher soll hier etwas detaillierter auf diesen Roboter eingegangen werden. Beim Khepera handelt es sich um einen Miniaturroboter mit zylindrischer Form (siehe 3.2a) mit einer Höhe von ca. 3cm (ohne Zusatzmodule) und einem Durchmesser von ca. 5,5cm. Der Roboter wurde speziell für wissenschaftliche Untersuchungen entwickelt und wird z.Zt. von der Fima K-Team vertrieben.

Zur Sensorkomponente des Roboters zählen die 8 analogen Entfernungssensoren[2]. Diese sind als Reflexlichtschranken implementiert und, wie in Abbildung 3.2b dargestellt, angeordnet. Die Sensoren haben eine Reichweite bis ca. 3 cm und liefern Informationen über Abstand zu einem Hindernis. Aufgrund der Art der Sensoren (Messung der Reflektion je eines radial abgegebenen Lichtsignales) hängt der tatsächliche Abstand des Roboters allerdings stark von der Beschaffenheit des Hindernises ab, insbesondere von dem Material aus dem es besteht (siehe auch [Khe]). Zur Sensorkomponente zählen weiterhin die Rad-Umdrehungszähler, die messen welcher Weg je Rad tatsächlich zurückgelegt wurde. Diese Zähler sind im Khepera-Simulator nicht implementiert und werden bei den in dieser Arbeit beschriebenen Versuchen auch nicht verwendet.[3]

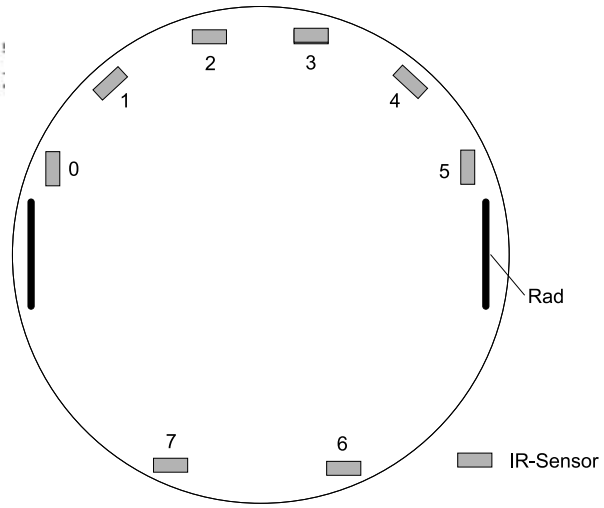
Die Ausführungskomponente des Khepera-Roboters besteht aus 2 Schrittmotoren, die unabhängig voneinander jeweils ein Rad antreiben. Je nachdem ob ein positiver oder ein negativer Wert für die Motoren vorgegeben wird, erfolgt eine Vorwärts- bzw. eine Rückwärtsbewegung der Räder. Die eingestellte Geschwindigkeit für die Motoren muss nicht tatsächlich in eine Drehbewegung der Räder umgesetzt werden. So kann es zum Beispiel sein, dass beide Räder durch ein Hindernis blockiert sind. Um die tatsächliche Radgeschwindigkeit abzulesen, muss der Rad-Umdrehungszähler ausgelesen werden.

[2]Ein Betrieb der Sensoren als Umgebungslicht-Sensoren ist ebenfalls möglich, wurde aber für diese Arbeit nicht verwendet.

[3]Es exisitert eine Weiterentwicklung des in Abschnitt 3.3 beschriebenen Homeokinese-Algorithmus, der die Radumdrehungszähler explizit verwendet: [Der01]



(a) Der Khepera-Roboter



(b) Schematischer Aufbau des Roboters, 6 Sensoren im Bug- und 2 im Heckbereich

Abbildung 3.1: Der Khepera-Roboter

Die Informationsverarbeitungskomponente des Roboters empfängt die Sensorwerte und erzeugt darauf aufbauend die entsprechenden Motorsignale. Diese Steuerung kann per Kabelfernsteuerung über einen angeschlossenen Computer erfolgen oder direkt durch ein in den Speicher des Roboters geladenes Assembler-Programm oder aber durch einen Hybrid-Ansatz, der die beiden vorigen Varianten kombiniert. Eine ausführlichere Erörterung der einzelnen Varianten findet sich in [Pan00]. Für die Versuche dieser Arbeit wurde immer die Kabelfernsteuerung verwendet, da dies der wesentlich flexiblere Ansatz ist.

Der Khepera-Simulator

Es gibt mehrere Simulatoren, die den Khepera-Roboter und dessen Umgebung nachbilden können. So zum Beispiel Webbots und der "Khepera Simulator" von Olivier Michel [Mic]. Für die Versuche in dieser Arbeit wurde der für wissenschaftliche Zwecke frei verfügbare "Khepera Simulator" verwendet. Abbildung 3.2 zeigt eine typische Arbeitsumgebung.

3.1.3 Arten von Steuerungsmechanismen

Bei der Kategorisierung von Steuerungsmechanismen für autonome Roboter unterscheidet man typischer Weise zwischen funktions- und verhaltensorientierten Ansät-

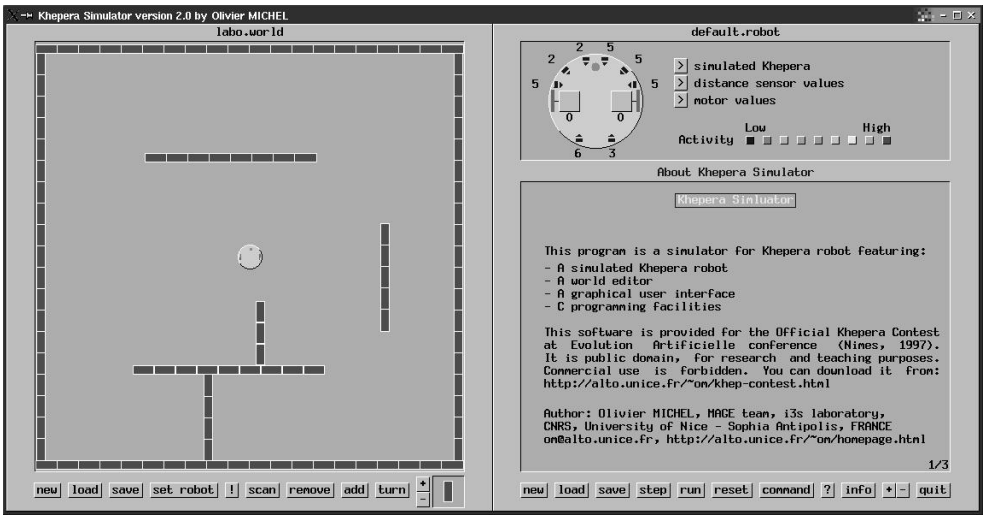


Abbildung 3.2: Der Khepera Simulator von Olivier Michel

zen. Basis für funktionsorientierte Ansätze sind die verschiedenen Funktionen der einzelnen Architekturkomponenten eines Roboters, die bei dieser Methode jeweils autonom eine Aufgabe im funktionellen Gesamtzusammenhang erfüllen. Basis einer verhaltensorientierten Architektur sind verschiedene einfache Verhaltensmuster, wie zum Beispiel Wandverfolgung und Kollisionsvermeidung.

In dieser Arbeit sollen Algorithmen zum Erlernen einfacher grundlegender Verhaltensweisen untersucht werden. Diese können dann beispielsweise als Komponenten einer verhaltensorientierten Architektur verwendet werden. Für das Erlernen grundlegener Verhaltensmuster steht häufig kein Lehrer zur Verfügung, d.h. die Muster müssen erlernt werden, ohne dass eine direkte Vorgabe aus der Umwelt zur Verfügung steht. Deshalb werden in dieser Arbeit nur Algorithmen untersucht, die ohne eigentlichen Lehrer auskommen[4].

3.2 Robotersteuerung mit Reinforcement-Lernen

Reinforcement Lernen (siehe u.a. [Dera, HKP91, KR95]) kann als eine Form des überwachten Lernens betrachtet werden. Dem System wird dabei ein Input (z.B. die Sensorwerte des Roboters) präsentiert, mit Hilfe dessen ein gewünschter Output (i.A. Steuersignal) erzeugt werden soll. Anders als beim Lernen mit Lehrer, bei dem genaue Informationen über den Ziel-Zustand vorliegen, gibt es hier aber nur ein Feedback-Signal,

[4]Reinforcement-Lernen kann als eine Form des überwachten Lernens (also Lernen mit Lehrer) angesehen werden. Für diese Art des Lernens steht aber kein eigentlicher Lehrer zur Verfügung, sondern das Lernsignal wird durch einfache Mechanismen erzeugt, beispielsweise durch einen Drucksensor als Kollisionsdetektor.

das Reinforcement-Signal , welches die Qualität des Outputs bewertet. Das Signal liefert nur Informationen über die Güte des Outputs (im einfachsten Fall nur "gut" oder "schlecht"), nicht aber darüber, wie das erwünschte Output aussehen soll. Für den Fall der Robotersteuerung bedeutet dies, dass ein bestimmtes Steuersignal (das aus dem Output erzeugt wurde) unter Umständen als fehlerhaft bewertet wird, aber keine Information verfügbar ist, wie das korrekte Steuersignal aussehen müsste.

Die Probleme, für die Reinforcement-Lernen angewendet werden soll, werden im Allgemeinen in verschiedene Klassen unterteilt (siehe [HKP91]):

- Klasse I ist der einfachste Fall, bei dem das Reinforcement-Signal für eine feste Kombination von Input- und Outputwerten immer gleich ist. Die Inputwerte und das Reinforcement-Signal hängen ferner nicht von vorangegangenen Output-Werten ab.
- Klasse II-Probleme sind dadurch gekennzeichnet, dass durch ein bestimmtes Input-Output-Paar lediglich die Wahrscheinlichkeit für ein positives (oder negatives) Reinforcement festgelegt ist. Reinforcement und Inputwerte dürfen wie bei Klasse I-Problemen nicht von vergangenen Outputs abhängen.
- Klasse III beinhaltet die übrigen Probleme, bei denen die Inputs und das Reinforcement-Signal von den Outputs der Vergangenheit abhängig sein dürfen. Ein negatives Reinforcement kann also beispielsweise durch eine Aktion (Output), die mehrere Schritte in der Vergangenheit liegt, ausgelöst worden sein.

Probleme der Klassen I und II werden als Probleme mit direkter Reinforcement-Vergabe bezeichnet. Bei Problemen der Klasse III sind im Allgemeinen wesentlich komplexer und werden auch Probleme mit verzögerter Reinforcement-Vergabe bezeichnet.

Viele Reinforcement-Lernverfahren, wie zum Beispiel das oft benutzte Q-Lernen, basieren auf einer Bewertung der möglichen Aktionen. Dazu wird von einer Menge diskreter Aktionen ausgegangen. Für viele Steuerungsaufgaben, insbesondere in der Robotik, ist aber eine Steuerung über kontinuierliche Parameter realistischer. [5] Es soll hier im Folgenden eine Methode beschrieben werden, die neuronale Netze benutzt und direktes Reinforcement-Lernen mit kontinuierlichen Steuerungswerten erlaubt.

3.2.1 Reinforcement-Lernen mittels Vorwärts-Modell

Das Verfahren, das hier beschrieben werden soll, wird in [HKP91, S.194ff] vorgestellt und basiert im Wesentlichen auf der Arbeit von Munro [Mun87][6]. Es werden neurona-

[5] Es gibt allerdings Ansätze die Verfahren, welche auf diskreten Aktionen basieren dahingehend anzupassen, dass diese auch über einen kontinuierlichen Aktionsraum funktionieren (siehe u.a. [KR95], [GWZ99]).

[6] Das hier vorgestellte Verfahren kann als ein Sonderfall des in [JR92] beschriebenen Prinzips betrachtet werden (siehe auch Abschnitt 3.4.1)

le Netze benutzt, um das Ziel-Output, d.h. das Steuerungsignal für den Roboter, auf Basis der Inputs zu bestimmen. Ein Problem beim Reinforcement-Lernen ist, dass keine direkte Information über die gewünschten Outputs vorhanden ist, sondern lediglich eine Information über deren Qualität. Um aber ein Netz beispielsweise mit Backpropagation trainieren zu können, ist für jeden Input-Wert ein zugehöriger Ziel-Wert für das Output notwendig. Die Idee dieses Verfahrens besteht darin, ein Vorwärts-Modell der Welt zu trainieren, welches vorhersagt, wie sich die Steuerungsparameter bei gegebenen Inputs auf das zu erwartende Reinforcement auswirken. Ist dieses Vorwärts-Modell hinreichend trainiert, stellt es Gradienteninformationen bezüglich der Wirkung der Steuerungsparameter auf das Reinforcement zur Verfügung. Diese Informationen können dann benutzt werden um ein weiteres Netz anzulernen, das auf Basis der Input-Daten die Steuerungsparamerter bestimmt.

Modell und Controller

Das Lernverfahren verwendet ein neuronales Netz, welches aus zwei Teilnetzen besteht, dem Controller-Netz (Steuerungs-Netz) und dem Modell-Netz (Vorhersage-Netz).

Das Controller-Netz erhält als Input die Sensordaten [7]. Als Output werden die Werte zur Steuerung des Roboters generiert.

Das Modell-Netz erhält die gleichen Sensordaten wie das Steuerungs-Netz und zusätzlich den Output des Steuerungs-Netzes als Input. Aufgabe des Vorhersage-Netzes ist es, auf Basis seiner Inputs das Reinforcement r, welches von der Umwelt geliefert wird, zu prognostizieren. Das Vorhersage-Netz hat dazu ein Neuron als Output. Das Ergebnis der Vorhersage, das prognostizierte Reinforcement, wird als $\widehat{r}$ bezeichnet.

Beide Netze sind direkt miteinander verbunden. Abbildung 3.3 zeigt eine schematische Darstellung des Aufbaus und Abbildung 3.4 die Implementation als neuronales Netz.

Vorrausgesetzt die Sensordaten sind exakt genug und angenommen die von den Sensoren erfassten Daten sind für die tatsächliche Umwelt bezüglich des Reinforcements ausschlaggebend, hat das Netz also die Parameter zur Verfügung, die für die Generierung des Reinforcements durch die Umwelt relevant sind: den augenblicklichen Zustand der Umwelt (basierend auf den Sensordaten) und die Aktionen des Roboters (basierend auf den Outputs des Steuerungs-Netzes). Das Vorhersage-Netz kann also unter den angenommenen Voraussetzungen ein Modell der Umwelt bezüglich r bilden. Das heißt, für jedes Input-Output-Paar des Steuerungs-Netzes gilt, falls das Modell gut genug ist, $\widehat{r} \approx r$. Als Trainingsmethode für das Vorhersage-Netz kann ein einfaches Backpropagation-Verfahren mit Lehrer verwendet werden. Denn für jedes Input-

[7] Hier sind Sensordaten im weiteren Sinne gemeint. Es ist z.B. auch möglich, als Input Roboter-interne Parameter zu verwenden.

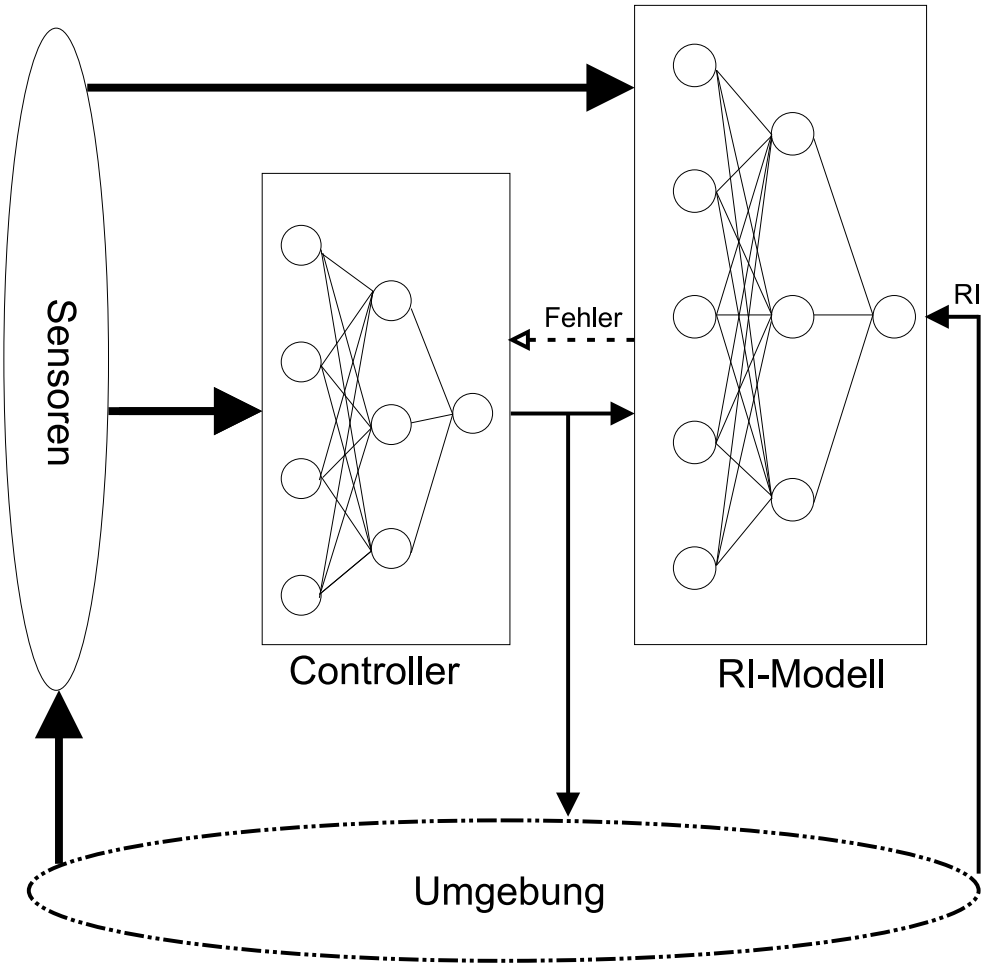


Abbildung 3.3: Modell/Controller-basiertes Netz zum Reinforcement-Lernen (schematisch)

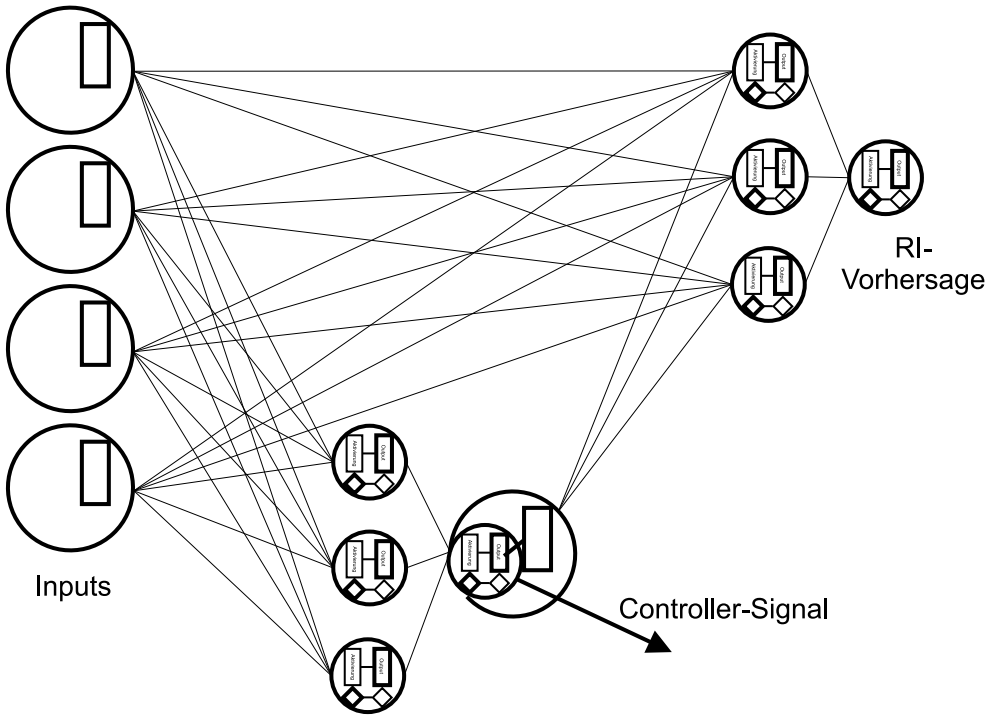


Abbildung 3.4: Beispiel für ein Netz zum Reinforcement-Lernen

Output-Paar liefert die Umwelt ein entsprechendes Reinforcement r. Das Netz muss trainiert werden den Vorhersagefehler (*prediction error*) $(r - \widehat{r})$ zu minimieren.

Wenn das Modell-Netzwerk hinreichend trainiert ist, kann es benutzt werden, um das Controller-Netzwerk zu trainieren. Ziel ist es dabei normalerweise, das Reinforcement zu maximieren[8]. Das gewünschte Reinforcement wird als r^* bezeichnet. Das Controller-Netzwerk muss nun trainiert werden, den Fehler im Verhalten (*performance error*) $(r^* - r)$ zu minimieren[9]. Dies kann durch das Backpropagation-Verfahren erreicht werden, indem der Fehler $(r^* - r)$ durch beide Netze hindurch zurückpropagiert wird. Dabei werden die Gewichte des Modell-Netzwerkes unverändert gelassen, aber die Fehlersignale über dieses Netz zu den Outputs des Controller-Netzwerkes und über diese durch das Controller-Netzwerk selbst hindurch zurückpropagiert. Die Gewichte im Controller-Netzwerk werden entsprechend der Fehlersignale gemäß dem Backpropagation-Verfahren abgeändert.

[8]Es kann aber auch durchaus sinnvoll sein, das Reinforcement zu minimieren, und zwar dann, wenn das Reinforcement-Signal benutzt wird, um schlechte Zustände zu signalisieren. Im biologischen Vorbild wird das zum Beispiel durch Schmerz realisiert.

[9]Es ist ebenso möglich den prognostizierten Fehler im Verhalten (*predicted performance error*) $(r^* - \widehat{r})$ zu minimieren. Bei einem perfekt gelernten Modell sind beide Verfahren äquivalent. In [JR92] wird aber gezeigt, dass bei einer Optimierung nach $(r^* - r)$ selbst dann die optimale Lösung gefunden werden kann, wenn das Vorwärts-Modell selbst nicht perfekt ist.

Das gesamte Netz bestehend aus Controller-und Modell-Teil kann als ein großes Backpropagation-Netz gesehen werden. Der Modell-Teil des Netzes beschreibt (bei einem fertig trainierten Prognose-Netz) ferner den Einfluss der Output-Neuronen des Controller-Teilnetzes auf das Reinforcement. Eine Änderung im Gesamtnetz zur Minimierung des Fehlers im Verhalten $(r^* - r)$ kann nur über eine Änderung im Controller-Teil des Netzes erfolgen. Daraus ergibt sich, dass über das oben beschriebene Verfahren das Steuerungs-Netz so trainiert wird, dass der Fehler $(r^* - r)$ minimiert wird. (Eine detaillierte Erklärung findet sich u.a. in [JR92].)

Das Training in der Praxis

Damit das Modell-Netzwerk optimal trainiert werden kann, ist es nötig möglichst alle Gebiete im Raum der Input-Output-Paare (bezüglich des Modell-Netzwerkes) durch Trainingsmuster abzudecken. Um diesbezüglich eine möglichst gute Abdeckung zu erreichen ist es erforderlich, bei gegebenen Sensorwerten mit vielen verschiedenen Steuerungsparametern zu trainieren. Das wird erreicht, indem man für das Controller-Netz stochastische Ausgabe-Einheiten verwendet, wie z.B. in 2.4.3 beschrieben. Dabei wird das "Abkühlen" des Temperatur-Parameters verwendet um im Laufe des Lernverfahrens von einem zufälligen zu einem durch den Controller determinierten Verhalten überzugehen.

Für den Erfolg der Modellierung ist es ebenfalls nötig dem Netzwerk Inputs (also Sensorwerte) aus dem gesamten Input-Spektrum zur Verfügung zu stellen. Das ist durch Anpassung des Steuerungs-Netzes nur sehr indirekt durchführbar. Um eine ausreichende Exploration des Inputraumes zu gewährleisten, verwendet man daher häufig eine bestimmte Anzahl von Versuchen. Ein Versuch beginnt mit einer beliebigen (zufälligen) Wahl der Inputs. (Bei dem hier verwendeten Beispiel eines Roboters entspricht dies dem Setzen an einen bestimmten Ort in der Umgebung.) Anschließend lässt man den Roboter eine bestimmte Zahl von Schritten entsprechend dem vorgegebenen Algorithmus agieren. Danach beginnt man von neuem mit der Wahl eines anderen Start-Ortes. Das Lernen muss dann auf diese Art über mehrere Versuche hinweg wiederholt werden, um eine ausreichend gute Exploration zur erreichen.[10]

[10]Für die vergleichenden Versuche in dieser Arbeit wurde teilweise auf die zufällige Wahl des Start-Ortes verzichtet, um eine besser Komparabilität zwischen den unterschiedlichen Lernverfahren zu gewährleisten.

3.3 Homeokinese als Steuerungsmechanismus

3.3.1 Der Homeokinese-Algorithmus

Einleitung

Ein entscheidendes Kriterium für den Erfolg von Lernverfahren wie Reinforcement-Lernen zur Steuerung autonomer Roboter besteht darin, die verwendeten Lernparameter günstig einzustellen, um bestimmte gewünschte (sinnvolle) Verhalten zu erzeugen. So ist beim Reinforcement-Lernen die geeignete Vergabe des Reinforcements von entscheidender Bedeutung . In bestimmten Lernphasen steht oftmals auch kein sinnvolles Lernsignal zur Verfügung, oder dieses ist nur schwierig zu finden.

Das Verfahren der Homeokinese, dass in [DSP99] vorgestellt wird, ist ein Lernverfahren, mit dem bestimmte Verhaltensweisen generiert werden können, ohne dass Wissen von außerhalb des Systems, wie es zum Beispiel die explizite Vergabe von Reinforcements für bestimmte Aktionen darstellt, vorgegeben wird

Homeokinese ist ein Lernverfahren, dass als Erweiterung des erstmalig von Cannon [Can39] publizierten Prinzips der Homeostase gesehen werden kann (bzw. als deren dynamisches Pendant). Cannon argumentiert, das Verhalten von Lebewesen ist darauf ausgerichtet bestimmte physiologische Parameter auf einem festgelegten konstanten Niveau zu halten. Abweichungen der tatsächlichen Werte von diesem Niveau lösen bestimmte Kontrollsignale aus, um den ursprünglichen Pegel wieder herzustellen. Die Kontrollsignale können auch komplexe Verhaltensweisen auslösen. Nach diesem Prinzip können bestimmte Verhaltensweisen, wie z.B. Nahrungssuche, als Ergebnis der Anforderung bestimmte physiologischen Parameter auf festgelegten Werten zu halten, gesehen werden.

Bei der Homeokinese ist das Ziel des Roboters nicht das Erreichen eines bestimmten durch die physiologischen Parameter definierten Zustandes, sondern die Aufrechterhaltung eines internen kinetischen Regimes. Dies wird realisiert durch ein adaptives Selbstmodell des Roboters. Die Diskrepanz zwischen Selbstmodell und tatsächlichem Verhalten wird dabei als ständig zur Verfügung stehendes Lernsignal benutzt.

Der Lern-Algorithmus

Die Beschreibung des Algorithmus orientiert sich an den Ausführungen in [DP99]. Allerdings wird hier eine leicht abgewandelte Bezeichnungsweise verwendet, die sich an der im Abschnitt 2.4 eingeführten orientiert.

Der Algorithmus geht davon aus, dass ein Controller und ein internes Modell des Roboters existieren. Ziel ist es, den Controller auf Basis der Sensordaten anzulernen, wobei

die Diskrepanz zwischen modellierten und tatsächlichen Sensorwerten als Basis für das Lernsignal dient.

Der Controller ist als formales Neuron definiert (siehe auch Abbildung 3.5). Der Output

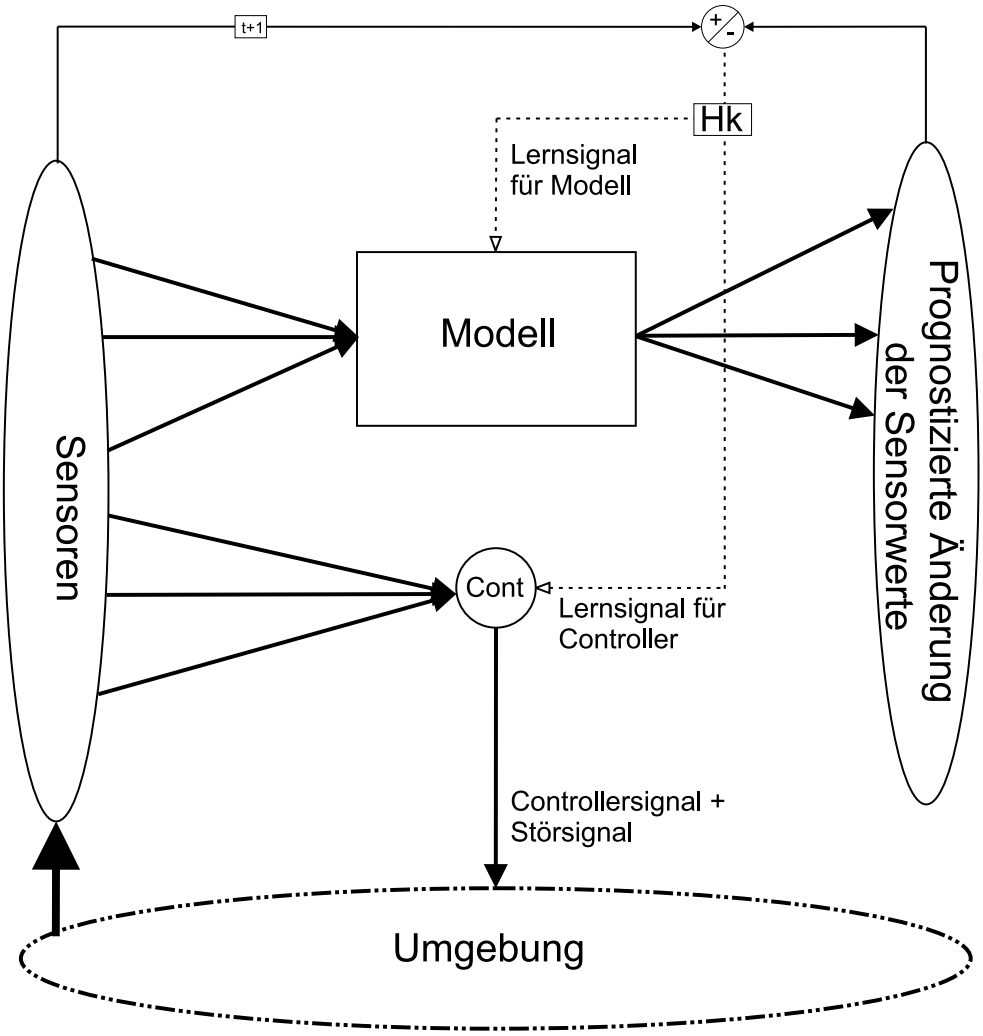


Abbildung 3.5: schematische Darstellung von Controller und Prädiktor beim Homeokinese-Verfahren

des Neurons dient als Steuerungssignal für den Agenten:

$$cont = f_{act}(\sum_i c_i sens_i) \tag{3.1}$$

Wobei $cont$ der Output des formalen Neurons ist (das als Steuerungssignal verwendet wird), $sens$ ist der Vektor der Sensorwerte und c der Vektor der Steuerungsparameter (Gewichte) des Controllers. Als Funktion $f_{act}()$ wird im Weiteren der Tangens hyperbolicus verwendet.

Angenommen die Sensorwerte werden in regulären Intervallen ausgelesen, dann kann man $\Delta sens$, die Veränderung der Sensorwerte im aktuellen Zeitschritt t, schreiben als Funktion von c und $sens$:

$$\Delta sens = F(sens, c) \tag{3.2}$$

Es gilt $sens(t+1) = sens + F(sens, c)$.

Als Selbstmodell des Agenten für dieses Lernverfahren soll nun die Prädiktion der Änderung der Sensordaten

$$\widehat{\Delta sens} = \Phi(sens; w) \tag{3.3}$$

dienen. Dabei ist w eine Menge von Parameterwerten, die zusammen mit der Funktion Φ das Selbstmodell (Prädiktor) definieren.

In [DP99] wird u.a. $\Phi = 0$ und

$$\Phi : \Phi_i(sens; w) = cont \sum_j w_{ij} sens_j \tag{3.4}$$

als Beispiel vorgeschlagen (siehe auch Abbildung 3.5).

Das Lernsignal für Controller und Modell ergibt sich nun, indem man eine Fehlerfunktion E einführt, die den Abstand zwischen vorhergesagten und tatsächlichen Sensorwerten ermittelt:

$$E = \frac{1}{2} \sum_i D_i^2 \tag{3.5}$$

wobei

$$D_i = \Delta sens_i - \widehat{\Delta sens}_i \tag{3.6}$$

ist. Die Änderung der Sensorwerte $\Delta sens$ ist nach Gleichung 3.2 abhängig von den aktuellen Sensorwerten $sens$ und den Controller-Parametern c. $\Delta pred$ ist neben den aktuellen Sensorwerten abhängig von den Parametern des Prädiktors $\widehat{\Delta sens}$ (siehe Gl. 3.3). Damit ergibt sich für die Fehlerfunktion $E = E(sens, c, w)$. Unter Zuhilfenahme des Gradientenabstiegsverfahrens kann man nun in jedem Zeitschritt ein Update für die Parameter c und w finden, mit dem Ziel, den Fehler E zu minimieren.

Für die Parameter des Selbstmodells ergibt sich mit Gl. 3.5

$$\Delta w_i = -\eta \frac{\partial}{\partial w_i} E = -\eta \frac{\partial}{\partial w_i} \frac{1}{2} \sum_j D_j^2 \tag{3.7}$$

und mit der Kettenregel

$$\Delta w_i = \eta \sum_j D_j \frac{\partial}{\partial w_i} \Phi_j(sens; w) \tag{3.8}$$

Die Ableitungen ergeben sich dann je nach verwendeter Funktion Φ.

Um die Änderung für die Parameter des Controllers zu berechnen wird analog zu oben die Ableitung von E bzgl. c_i benötigt, also

$$\frac{\partial}{\partial c_i} E = \frac{\partial E}{\partial cont} \frac{\partial cont}{\partial c_i} \tag{3.9}$$

Die Ableitung $\frac{\partial cont}{\partial c_i}$lässt sich über die Gleichung 3.1 explizit ermitteln. Die Gradienteninformation bezüglich $cont$ lässt sich nur indirekt bestimmen. Dazu wird eine Störung $\phi(t)$ zum Output $cont$ des Controllers addiert, also

$$cont = f_{act}(\sum_i c_i sens_i) + \gamma\phi(t) \tag{3.10}$$

wobei die Funktion ϕ(t) ein Rauschen mit dem Mittelwert 0 oder eine periodische Oszillation ist. Die Änderung von ϕ zwischen 2 Zeitintervallen sei dabei klein und ϕ sei so gewählt, dass bei Mittelung über einen längeren Zeitraum $\overline{\phi} \ll \overline{\phi^2}$gilt.

Unter den gegebenen Bedingungen ergibt sich als Update-Regel für die Parameter des Controllers

$$\Delta c_i = -\eta f'_{act} sens_i \phi(t) E \tag{3.11}$$

Die vollständige Herleitung dazu findet sich in [DP99].

3.4 Modell-/Controller-basiertes Lernen und die Homeokinese-Idee

3.4.1 Lernen mit Modell und Controller

Das in Abschnitt 3.2.1 beschriebene Verfahren zum Reinforcement-Lernen mit Modell- und Controller-Netzwerk kann relativ einfach auf allgemeinere Fälle übertragen werden. Und zwar soll das Verfahren dahingehend erweitert werden, dass das Netz trainiert wird, beliebige Ziel-Zustände zu erreichen. Der Ziel-Zustand sei dabei ausschließlich durch den Vektor der Sensorwerte $sens$ bestimmt. Das beobachtete Reinforcement r kann als ein Element dieses Vektors betrachtet werden. Auch ist der Sensor-Vektor nicht auf externe Sensoren beschränkt. Es ist zum Beispiel auch ein Sensor denkbar, der einen Zustand des Lernsystems beschreibt.

Um die verallgemeinerte Form des Lernverfahrens zu erhalten, ist es lediglich nötig, die in Abschnitt 3.2.1 eingeführte Beschränkung auf ein Output-Neuron, welches das Reinforcement vorhersagt, aufzuheben. Es soll weiter nicht nach dem gewünschten Reinforcement r^*optimiert werden, sondern nach dem Ziel-Zustand, der durch den Vek-

tor der erwünschten Sensorwerte $sens^*$ repräsentiert wird[11]. Die Output-Schicht des Vorhersage-Netzes enthalte nun soviele Neuronen, wie $sens^*$ Elemente hat. Das Ergebnis der Prognose des Modell-Netzes ist der Vektor $\widehat{sens}$. Die Strukur der Netze bleibt ansonsten wie im Abschnitt 3.2.1 beschrieben. Es ergibt sich damit ein Aufbau, wie in Abbildung 3.6 und 3.7 dargestellt.

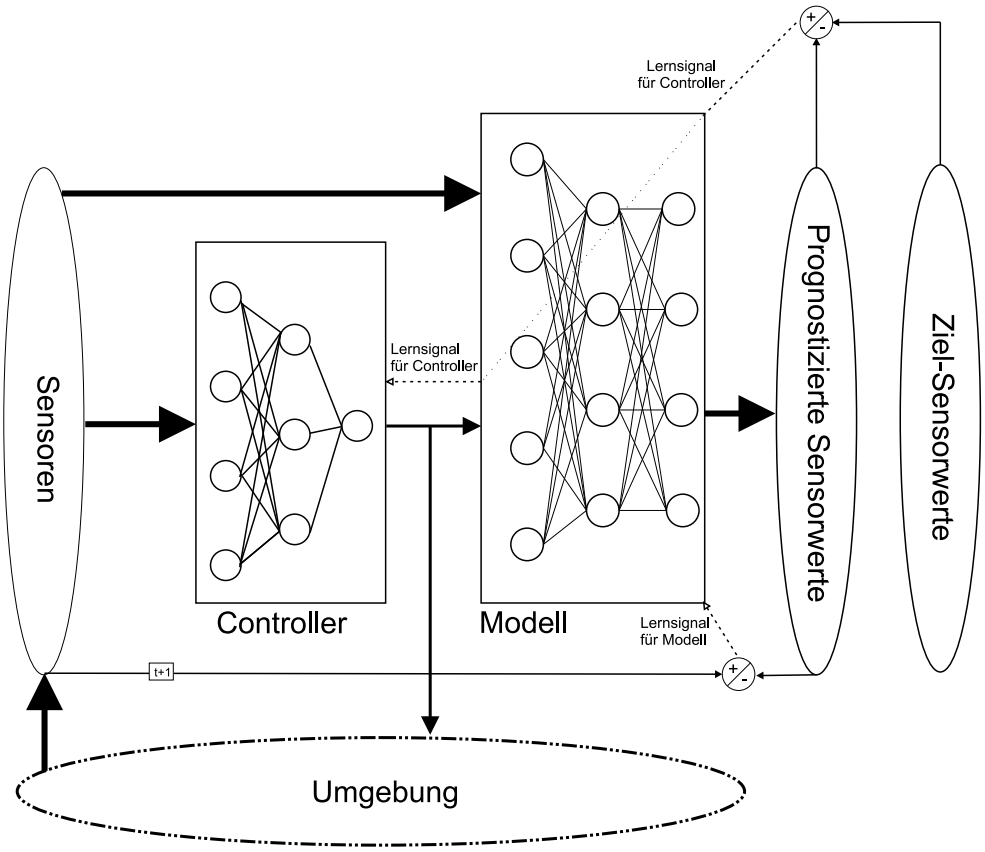


Abbildung 3.6: Modell/Controller-basiertes Netz zur Erzeugung gewünschter Sensorwerte (schematisch)

Das Modell-Netzwerk wird trainiert, auf Basis der Sensorwerte und der Outputs des Controller-Netzes die Sensorwerte im nächsten Zeitschritt vorherzusagen. Dazu wird es trainiert, den Vorhersagefehler

$$(sens - \widehat{sens}) \tag{3.12}$$

zu minimieren.

[11] Der Vektor $sens$, wie hier beschrieben, kann auch nur eine Teilmenge der dem Netz tatsächlich zur Verfügung stehenden Sensorwerte sein (oder auf berechneten Daten beruhen). Der Einfachheit halber soll aber im Folgenden angenommen werden, das $sens$ alle zur Verfügung stehenden Sensorwerte repräsentiert und $sens^*$ die entsprechenden gewünschten Zielwerte für die einzelnen Sensoren repräsentiert.

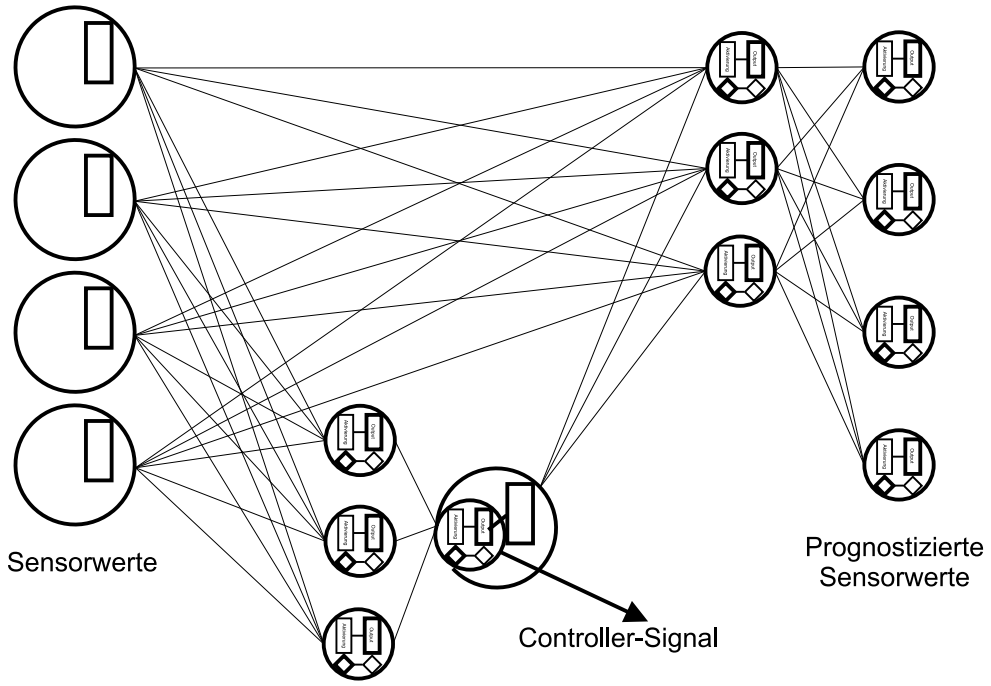


Abbildung 3.7: Beispiel eines Netzes zur Erzeugung vorgegebener Sensorwerte

Der Fehler im Verhalten wird nun nicht mehr durch die Differenz zwischen tatsächlichem und gewünschtem Reinforcement $(r^* - r)$, sondern allgemeiner durch die Differenz zwischen tatsächlichen und gewünschten Sensorwerten

$$(sens^* - sens) \tag{3.13}$$

bestimmt. Das Prinzip des Trainings bleibt gegenüber dem in Abschnitt 3.2.1 beschriebenen Verfahren unverändert: Der Vorhersagefehler wird zum Training des Modell-Teilnetzes benutzt und die Differenz zwischen realen und erwünschten Sensorwerten für das Training des Controller-Teilnetzes, wobei dieser Fehler wie beschrieben durch das Prognosenetz, aber ohne dieses abzuändern, hindurch propagiert wird.

Angenommen das Modell-Netz ist bereits perfekt angelernt. Wenn nun das Gesamtsystem, bestehend aus Modell- und Controller-Netz, wie oben beschrieben trainiert wird, eine Abbildung von den Input-Werten auf die erwünschten Sensorwerte $sens^*$ zu erzeugen ohne das Modell-Netz zu verändern, so kann diese Abbildung nur dadurch realisiert werden, dass die Abbildung von den Input-Werten auf den Output des Controller-Netzes modifiziert wird. Unter der Voraussetzung, dass das Modell perfekt trainiert ist, und angenommen das Backpropagation-Verfahren findet die global optimale Lösung des Lernproblems, so ist auch die erzeugte Abbildung von den Inputs auf die Outputs des Controller-Netzes dahingehend optimal, dass durch den Output als Steuerungsparemter die gewünschten Sensorwerte $sens*$ optimal erzeugt werden. Wenn das Modell-Netz nicht perfekt angelernt wurde, kann immer noch die global op-

timale Lösung gefunden werden. Voraussetzung dafür ist aber, dass das Modell-Netz zumindestens so gut trainiert wurde, dass die über das Backpropagation-Verfahren aus dem Netz gewonnenen Gradienteninformationen zumindest in die richtige Richtung weisen. Dieses Lernverfahren wird in [JR92] als Distal Supervised Learning bezeichnet und ausführlich beschrieben und diskutiert.[12]

3.4.2 Ein netzbasierter Ansatz unter der Verwendung der Homeokinese-Idee

Die Idee zu dem in diesem Abschnitt vorgestellten Verfahren entstand während der Erstellung dieser Diplomarbeit. Es verbindet Aspekte des Homeokineseverfahrens (Abschnitt 3.3) mit Modell-Controller-basiertem Lernen und kann als ein Spezialfall des in Abschnitt 3.4.1 vorgestellten Verfahrens gesehen werden.

Eine grundlegende Idee des Homeokinese-Algorithmus ist es, den Fehler zwischen Welt-Modell des Roboters und der Realität (genauer gesagt den Fehler zwischen prognostizierten und tatsächlichen Sensorwerten) als Lernsignal für die Steuerung und damit das Verhalten zu verwenden. Der Roboter wird also trainiert, ein Verhalten zu entwickeln, welches er gut prognostizieren kann (also eine "vernüftige" Verhaltensweise). Der im folgenden vorgestellte netzbasierte Algorithmus folgt dem gleichen Prinzip[13].

Für dieses Verfahren wird ein normales Feed-Forward-Netz benutzt. Das Netz hat sowohl in der Input-Schicht als auch in der Output-Schicht genau so viele Neuronen, wie der zu steuernde Roboter Sensorwerte liefert. Außerdem hat es mindestens eine versteckte Schicht. Eine dieser versteckten Schichten sei die Control-Schicht. Das Signal, welches als Output der Control-Schicht entsteht, wird benutzt um den Roboter zu steuern. Die Control-Schicht besitzt daher soviele Neuronen, wie Effektor-Werte geliefert werden sollen. Die Schichten sind so verbunden, dass die Control-Schicht direkt oder indirekt (über andere Schichten) mit der Eingabeschicht und der Ausgabeschicht verbunden ist. Die Ausgabeschicht ist weiterhin direkt oder indirekt, aber nicht über die Control-Schicht, mit der Eingabeschicht verbunden. Abbildung 3.8 zeigt ein Beispiel für ein solches Netz.

Dem Netz werden die Sensorwerte $sens$ zum Zeitpunkt t als Input präsentiert und es

[12]Mit dem hier vorgestellten Prinzip ist es auch möglich ein inverses Modell bezüglich der Sensorwerte zu trainieren, d.h. die Eingabe der gewünschten Sensorwerte als Inputs erzeugt direkt das zum Erreichen dieser Werte nötige Controller-Signal. Dazu muss man statt der tatsächlichen die gewünschten Sensorwerte als Input für das Controller-Lernen verwenden. Normalerweise will man aber die realen Sensorwerte nicht ignorieren, so dass man zusätzliche Inputs für die gewünschten Sensorwerte mit aufnimmt. (siehe [JR92])

[13]Die Idee, den Fehler zwischen Modell und Realität als Lernsignal sowohl für das Modell selbst, als auch für das Verhalten des Agenten zu benutzen, ist ein grundlegendes Element des Homeokinese-Algorithmus. Sie wird daher hier als "Homeokinese-Idee" bezeichnet. Der eigentliche Algorithmus allerdings erstreckt sich wie beschrieben weit über diese Basis-Idee hinaus.

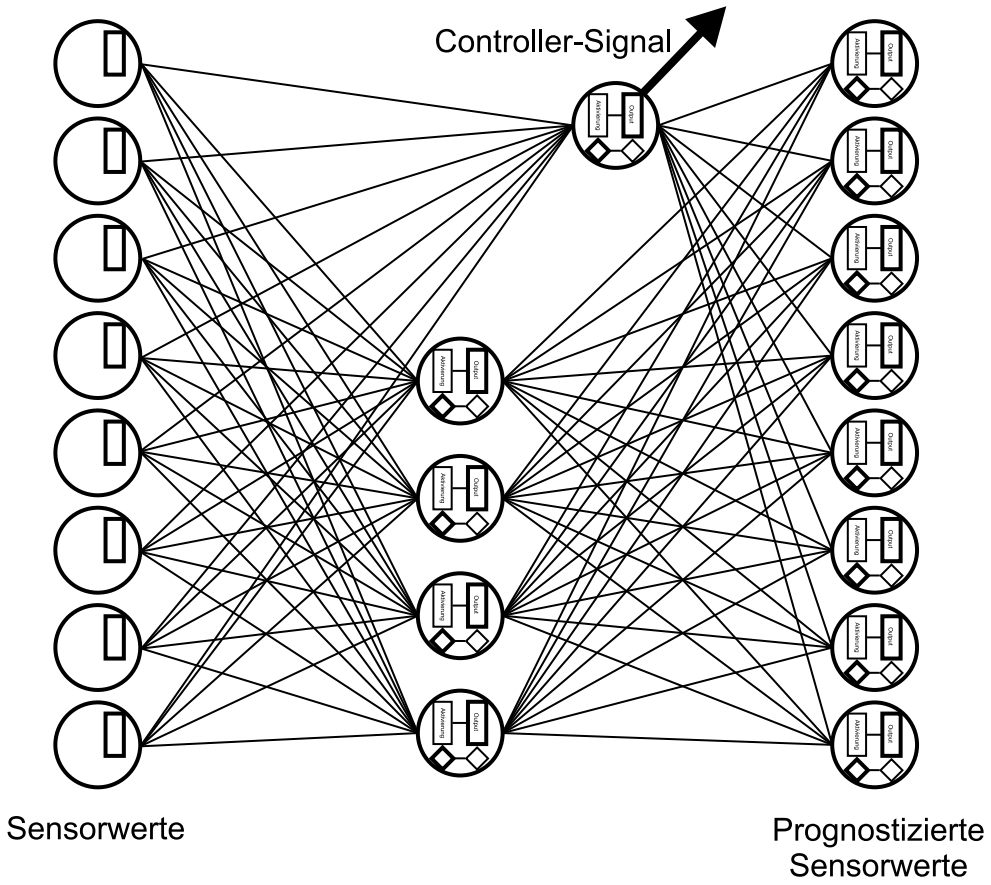


Abbildung 3.8: Der Fehler in der Vorhersage wird benutzt um das Steuersignal anzulernen

wird trainiert, die Sensorwerte des Zeitpunkts $t + 1$ vorherzusagen. Der Fehler in der Vorhersage $(sens - \widehat{sens})$ wird also als Lernsignal benutzt.

Wenn man das Netz so trainiert, wird das Controller-Signal so verändert, dass der Prognosefehler minimiert wird. D.h. es wird wie gewünscht ein Verhalten erzeugt, welches gut vorhersagbar ist. Dass dies tatsächlich der Fall ist kann man sehen, wenn man das Netz in zwei Teile aufgeteilt betrachtet. Die Control-Schicht und die Schichten, über die sie mit der Eingabeschicht verbunden ist, seien das Controller-Netz. Die restlichen Schichten inklusive der Inputschicht seien das Modell-Netz. Die Netze werden nach dem in Abschnitt 3.4.1 beschriebenen Verfahren trainiert. Der schematische Aufbau nach diesem Ansatz ist in Abbildung 3.9 dargestellt.

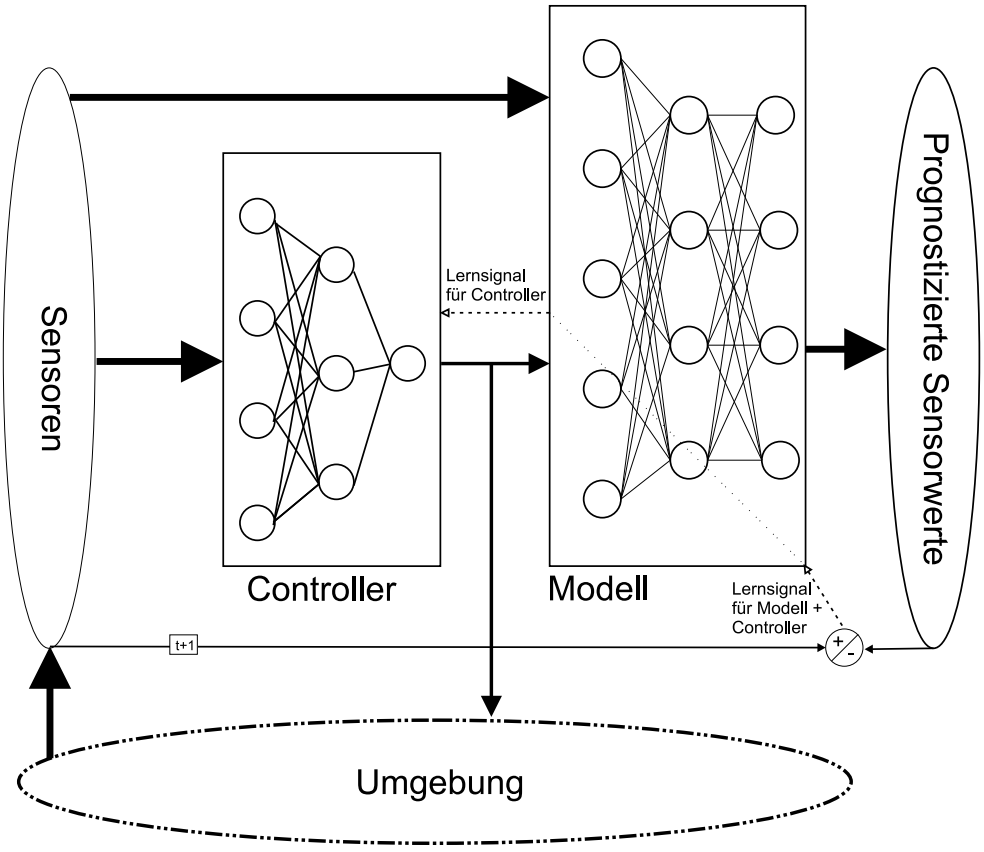


Abbildung 3.9: Modell-Controller-Netz zum Lernen nach der Homeokinese-Idee

Das Modell-Netz wird trainiert, die Sensorwerte möglichst gut zu prognostizieren, d.h. den Vorhersagefehler $(sens - \widehat{sens})$ zu minimieren. Das gesamte Netz wird nun im Gegensatz zu dem in Abschnitt 3.4.1 beschriebenen Verfahren mit dem gleichen Fehler wie das Modell-Netz dem Vorhersagefehler $(sens - \widehat{sens})$ trainiert.

Da Modell-Netz und Gesamt-Netz mit dem gleichen Fehler trainiert werden, können die beiden in Abschnitt 3.4.1 beschriebenen Schritte beim Training zu einem Schritt

zusammengefasst werden. Das Modell-Netz und das Controller-Netz müssen nun also nicht mehr seperat trainiert werden. Es ergibt sich damit das Lernverfahren wie zu Beginn dieses Abschnittes beschrieben. Der Controller wird angelernt den Fehler $(sens - \widehat{sens})$ zu minimieren.

Dieses Verfahren verfolgt also das gleiche Prinzip wie die Homeokinese, nämlich die Diskrepanz zwischen modellierten und tatsächlichen Verhalten als Lernsignal sowohl für das Modell als auch für den Controller zu benutzen.

3.4.3 Modell-Controller-basiertes Lernen im Vergleich mit Homeokinese

Hier soll ein einfaches Netz, bestehend aus Modell und Controller erzeugt werden, dass nach dem in Abschnitt 3.4.1 beschriebenen Verfahren trainiert wird. Aufbau und Training sollen mit dem in Abschnitt 3.3.1 beschriebenen Prinzip der Homeokinese verglichen werden. Für die Berechnungen in diesem Abschnitt wird w_{ij} das Gewicht der Verbindung von Neuron j zu Neuron i definiert und nicht umgekehrt wie im Rest der Arbeit[14].

Das Controller-Netzwerk hat ein Output-Neuron, welches das Steuersignal $cont$ für den Roboter erzeugt. Als Input stehen die Sensorwerte $sens$ zur Verfügung. Das Controller-Netz hat ferner keine verdeckten Neuronen. Das Modell-Netzwerk hat soviele Neuronen wie $sens$ Elemente hat. Als Inputs dieses Netzes dienen die Sensoren und der Output $cont$ des Controller-Netzes. Im Modell-Netz gibt es wie im Controller-Netz keine verdeckten Neuronen. Das Modell-Netz wird trainiert, die Änderung der Sensorwerte $\Delta sens$ vorherzusagen. Dazu wird das Netz trainiert, den Fehler Prognosefehler $(\Delta sens - \widehat{\Delta sens})$ zu minimieren.[15]. Abbildung 3.10 zeigt den Aufbau des Netzes. Für das Training des Controller-Teilnetzes wird, ebenso wie in Abschnitt 3.4.2, der gleiche Fehler wie zum Lernen des Modell-Teilnetzes verwendet. Da für beide Teilnetze der gleiche Fehler verwendet wird, kann das Lernen wie oben beschrieben in einem Schritt erfolgen. Für jedes Neuron i der Ausgabeschicht wird also

$$D_i = (\Delta sens_i - \widehat{\Delta sens_i}) \tag{3.14}$$

als Fehler gesetzt.

[14]Diese Umbennenung erfolgt hier um mit der Notation zum Homeokinese-Algorithmus in Abschnitt 3.3.1 kompatibel zu bleiben.

[15]Das entspricht nicht vollständig dem in Abschnitt 3.4.1 beschriebenen Verfahren, allerdings lassen sich die Prognose-Werte $\widehat{\Delta sens}$ einfach aus den aktuellen ($sens$) und den prognostizierten Sensorwerten ($\widehat{sens}$) berechnen. Das Verfahren funktioniert ferner auch dann, wenn Inputs und Outputs nicht übereinstimmen (siehe [JR92])

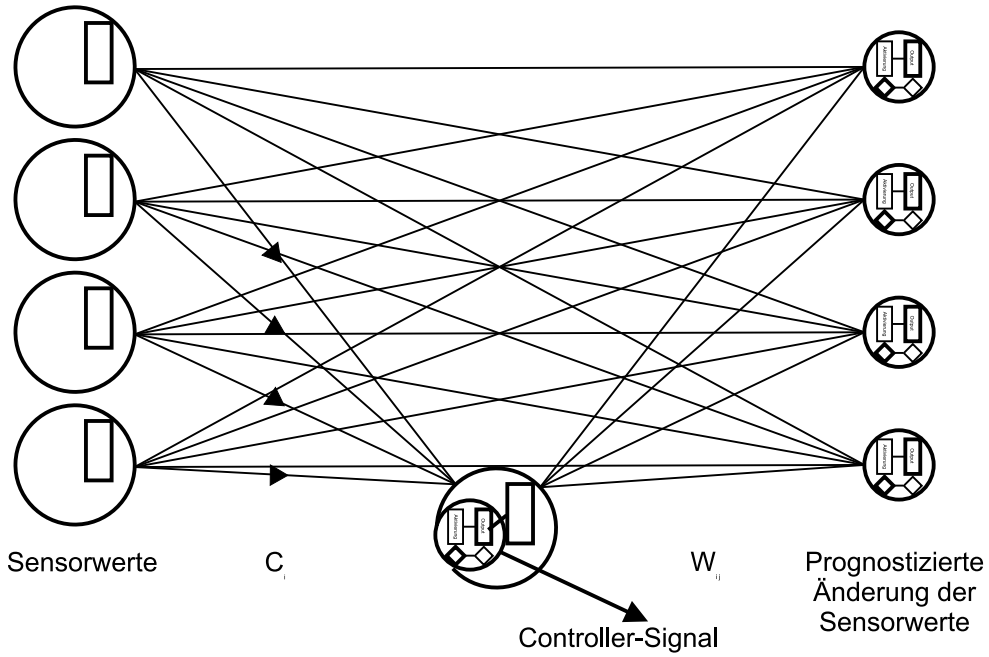


Abbildung 3.10: einfaches Modell/Controller-basiertes Netz

Der Output der Neuronen der Ausgabeschicht $\widehat{\Delta sens}$ ergibt sich nach den Gleichungen 2.4 und 2.6 als

$$out_i = \widehat{\Delta sens}_i = f_{act}(net_i) = f_{act}(\sum_{j=0}^{n} w_{ij} out_j) = f_{act}(\sum_{j=1}^{n} w_{ij} sens_j + w_{i0} cont) \tag{3.15}$$

Der Output $cont$ des Controller-Teils des Netzes ergibt sich zu

$$cont = f_{act}(\sum_{i} c_i sens_i) \tag{3.16}$$

Das hier beschriebene, und mit dem Backprop-Verfahren trainierte Netz soll nun mit dem in Abschnitt 3.3.1 beschriebenen Homeokinese-Algorithmus mit Φ definiert durch

$$\Phi_i = \widehat{\Delta sens}_i = w_{i0} cont + \sum_{j=1}^{n} w_{ij} sens_j \tag{3.17}$$

als Funktion für das Selbstmodell (Prädiktor) verglichen werden.

Das Training des Modell-Teilnetzes, also der Parameter W, erfolgt exakt wie durch den Homeokinese-Algorithmus (siehe Gleichung 3.7) vorgegeben nach dem Gradientenabstiegsverfahren:

$$\Delta w_{ij} = -\eta \frac{\partial}{\partial w_{ij}} E \tag{3.18}$$

mit der Kettenregel folgt

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial out_i} \frac{\partial out_i}{\partial net_i} \frac{\partial net_i}{\partial w_{ij}} \tag{3.19}$$

mit den Gleichungen 3.5 und 3.6 ergibt sich

$$\frac{\partial E}{\partial out_i} = -D_i \tag{3.20}$$

und mit Gleichung 3.15 folgt

$$\frac{\partial out_i}{\partial net_i} \frac{\partial net_i}{\partial w_{ij}} = f'_{act}(net_i) \frac{\partial}{\partial w_{ij}} \sum_j out_j w_{ij} = f'_{act}(net_i) out_j \tag{3.21}$$

Eingesetzt in Gleichung 3.18 ergibt sich die Standard-Backpropagation-Regel für die Neuronen der Output-Schicht

$$\Delta w_{ij} = \eta \cdot out_j \cdot f'_{act}(net_i)(\Delta sens_i - \widehat{\Delta sens_i}) \tag{3.22}$$

Das Training ist für das Modell also identisch zum Homeokinese-Algorithmus.

Das Update der Parameter des Controllers erfolgt nach dem Backpropagation-Algorithmus wie folgt:

$$\Delta c_i = \eta out_i \delta_{cont} = \eta sens_i \delta_{cont} \tag{3.23}$$

mit

$$\delta_{cont} = f'_{act}(net_{cont}) \sum_k \delta_k w_{k0}$$

und

$$\delta_k = f'_{act}(net_k) D_k$$

Es ergibt sich eingesetzt in Gleichung 3.23

$$\Delta c_i = \eta sens_i f'_{act}(net_{cont}) E^* \tag{3.24}$$

$$\Delta c_i = \eta sens_i f'_{act}(net_{cont}) \sum_k f'_{act}(net_k) D_k w_{k0} \tag{3.25}$$

als Update für die Gewichte des Controllers. Der Updateschritt unterscheidet sich dadurch, dass über den letzten Faktor die Gradienteninformation beim Homeokinese-Algorithmus über den Term $\phi(t)E$ und bei dem hier beschriebenen Netz über $E^* =$

$\sum_k f'_{act_k}(net_k) D_k w_{k0}$ bereitgestellt wird[16]. Im Falle des Homeokinese-Algorithmus wird die Gradienteninformation daher über die Störfunktion $\phi(t)$ erzeugt, bei der Modell-Controller-basierten Variante hier, über die Informationen, die in dem Modell-Netz implizit enthalten sind. Die Gradienteninformationen, die sich über das Modell-Netz ergeben, können allerdings nur dann sinnvoll zur Minimierung des Fehlers benutzt werden, wenn das Modell-Netz zumindestens teilweise trainiert ist.

3.4.4 Kombination der Homeokinese-Idee mit Reinforcement-Lernen

Das in Abschnitt 3.4.2 beschriebene Lernverfahren verwendet das gleiche Ziel wie der Homeokinese-Algorithmus. Es baut ferner ebenso wie das in dieser Arbeit vorgestellte Reinforcement-Prinzip auf dem in Abschnitt 3.4.1 geschilderten Modell-/Controller-basierten Lernen auf. Eine Kombination der beiden Verfahren im Rahmen des Modell-/Controller-basierten Lernen liegt also nahe.

Um diese Kombination zu erreichen betrachtet man das Reinforcement-Signal als einen weiteren Sensorwert. Damit wird nun wie in Abschnitt 3.4.1 beschrieben ein Modell-Controller-Netz trainiert, um die erwünschten Sensorwerte zu erzeugen. Der Modell Teil wird normal trainiert die Sensorwerte zu prognostizieren. Für den Controller-Teil des Netzes werden folgende Vorgaben für die gewünschten Sensorwerte verwendet: Für jedes Reinforcementsignal r wird das gewünschte Reinforcement r^* als Vorgabe verwendet, für alle übrigen Sensorwerte wird der jeweilige prognostizierte Sensorwert verwendet.

3.5 Experimentelle Untersuchung der Steuerungsmechanismen

Die hier vorgestellten Steuerungsalgorithmen sollen im Folgenden an einigen beispielhaften Problemen untersucht werden. Ziel der Lernverfahren für autonome Roboter ist deren Anwendung in der Praxis, d.h. die Implementation auf einem realen Roboter. Für die folgenden Untersuchungen wurde aber dennoch ein Simulator verwendet. Grund dafür ist die einfachere Vergleichbarkeit der einzelnen Ergebnisse untereinander. In einem Simulator kann man für jedes der Verfahren zur Robotersteuerung die gleichen

[16]Durch die Wahl einer individuellen linearen Aktivierungsfunktion $f_{act_i}(x) = \frac{1}{w_{i0}} x$ für jedes Neuron der Ausgabeschicht (des Modell-Teilnetzes) lässt sich erreichen, dass sich die Fehlerinformation E^* zu $E^* = \sum_k f'_{act_k}(net_k) D_k w_{k0} = \sum_k \frac{1}{w_{k0}} D_k w_{k0} = \sum_k D_k$ also $E^* = \sum_k D_k$im Gegensatz zu $E = \phi(t) \frac{1}{2} \sum_i D_i^2$ beim Homeokinese-Algorithmus ergibt. Wenn man $cont$ unter Addition einer Störfunktion $\phi(t)$ berechnet: $cont = f_{act}(\sum_i c_i sens_i) + \phi(t)$, so geht diese dann über D_k mit $-\sum_k w_{k0}\phi(t)$ als Summand in die Berechnung von E^* ein.

Ausgangsbedingungen schaffen. In der Realität ist dies nur schwer realisierbar. Außerdem ist es möglich die Experimente im Simulator schneller laufen zu lassen, als der Roboter sich in der realen Welt bewegen würde.

Die Verwendung einer Simualtionsumgebung bringt allerdings auch einige Nachteile mit sich. Keine Simulation spiegelt die Realität vollkommen authentisch wider. Die in einer Simulation gewonnenen Ergebnisse lassen sich daher nur in begrenztem Ausmaß auf die wirkliche Situation übertragen. Im nächsten Abschnitt soll daher kurz die verwendete Simulationsumgebung vorgestellt und mit den realen Verhalten verglichen werden.

3.5.1 Die Simulationsumgebung

Als Simulationsumgebung wurde der bereits in Abschnitt 3.1.2 erwähnte Khepera-Simulator von Olivier Michel in der Version 2.0 verwendet. Der Quelltext zu diesem Programm steht für nicht-kommerzielle Anwendungen zur freien Verfügung. Das ist neben anderen ein wesentlicher Grund, weshalb dieser Simulator ausgewählt wurde. Es war so möglich, einige benötigte Änderungen am Code des Simulators vorzunehmen und den Simulator als Teil eines allgemeinen Frameworks zur Roboteransteuerung zu verwenden. Der Khepera-Simulator bietet zusätzlich zum Betrieb als Simulator die Option einen realen Roboter anzusteuern. So kann man den selben Quellcode sowohl für den simulierten als auch für den realen Roboter zu verwenden.

Die Simulation beschränkt sich auf die Reflex-Lichtschranken zur Entfernungs- bzw. Umgebungslichtmessung und die Auswertung der Kommandos zur Radansteuerung. Die Auswertung der gemessenen tatsächlich erfolgten Radumdrehungen wird nicht unterstützt. Zusatzmodule für den Khepera wie Greifarme oder eine Zeilenkamera finden ebenfalls keine Unterstützung. Im Simulator wird nicht zwischen verschiedenen Oberflächen der Hindernisse unterschieden. Des weiteren sind (außer dem Roboter selbst) keine beweglichen Objekte verfügbar. Das Führen eines Balles durch den Roboter (wie z.B. in [DP99] als Verhalten beschrieben) lässt sich also nicht nachvollziehen.

Die verrauschten Sensorwerte des realen Roboters werden zwar durch den Simulator nachgebildet, allerdings entspricht das Verhalten der Sensoren teilweise nicht der Realität. Die simulierten Sensorwerte zeigen (zumindestens in der verwendeten Version) zum Teil ein sprunghaftes Verhalten. Während der reale Roboter beim Zusteuern auf die Wand die Sensorwerte relativ gleichmäßig abändert, erfolgt die Änderung im Simulator in plateauhaften Etappen[17]. Die beim realen Roboter auftretenden Signallauf-

[17]Durch dieses Verhalten entstehen teilweise auch Artefakte beim Test der Lernverfahren, die beim realen Roboter so nicht auftreten. Der simulierte Roboter findet z.B. teilweise einen Abstand zur Wand, bei dem ein Wechsel zwischen 2 verschiedenen Plateaus der Sensorwerte stattfindet. Durch Pendeln kann er nun ein ständiges Hin-und Herschalten zwischen den Werten der verschiedenen Plateaus erzeugen und somit die Sensorwerte scheinbar zuverlässiger prognostizieren.

zeiten werden vom Simulator ebenfalls ignoriert. Die Sensoren liefern wie beim realen Roboter Werte im Bereich von 0 bis 1023. Für alle hier dargestellten Versuche wurden die Werte normiert oder auf einen Bereich von -1 bis $+1$ skaliert.

Als Einstellungen für die Radmotoren sind Werte -10 bis 10 für die praktische Anwendung sinnvoll. Jedes Rad wird einzeln angesteuert. Positive Werte bedeuten ein vorwärts Drehen des Rades, negative Werte ein rückwärts Drehen. Ist die Einstellung 0, wird das Rad nicht gedreht. Durch unterschiedliches Einstellen der Motoren lassen sich ähnlich wie bei einem Rollstuhl verschiedene Bewegungsmuster erzeugen. Als Motorwerte stehen nur ganzzahlige Werte zur Verfügung. Die Umsetzung der Ansteuerung entspricht weitestgehend dem wirklichen Verhalten. Lediglich auf glatten, rutschenden Oberflächen sollten sich signifikante Unterschiede ergeben.

Durch die beschriebenen Gründe ist das Verhalten des Roboters im Simulator teilweise anders als in der Realität. Alle untersuchten Verfahren wurden daher auch kurz am realen Roboter untersucht.

3.5.2 Der Versuchsaufbau

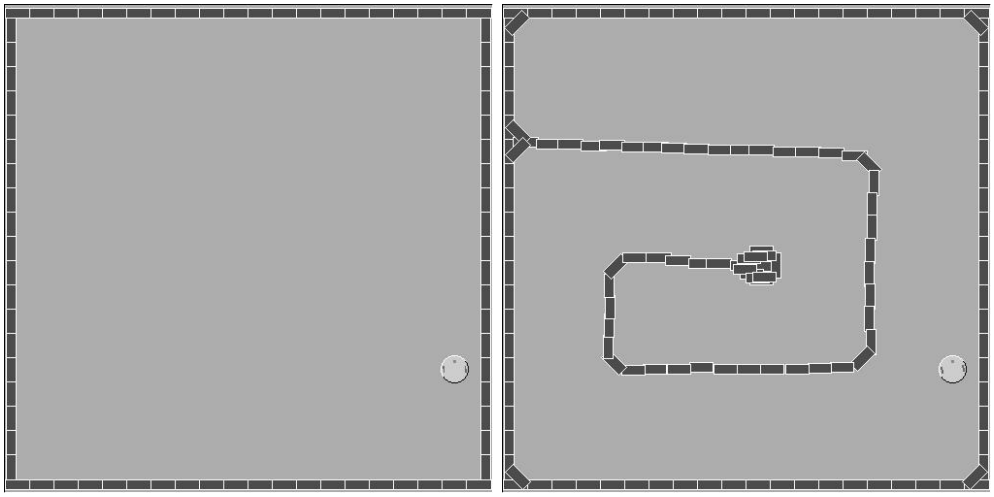
Das Verhalten des Roboters wurde mit den verschiedenen Lernverfahren jeweils in der selben Umgebung untersucht. Dabei wurden alle Parameter, die nicht direkt das Lernverfahren betreffen, weitestgehend unverändert gelassen.

Der Test der Algorithmen erfolgte in 4 verschiedenen Simulator-Umgebungen, die jeweils einen unterschiedlichen Schwierigkeitsgrad aufweisen. Umgebung 1 (im Folgenden auch "Welt 1") ist bis auf die Randbegrenzung vollständig leer (Abbildung 3.11a). In Welt 2 (Abbildung 3.11b) gibt es keine "scharfen" Kanten, was für verschiedene Algorithmen das Lernen vereinfacht. Die Hindernisse in Welt 3 kennzeichnen sich durch Linien aus, die nur horizontal oder vertikal verlaufen (siehe Abbildung 3.11c). Welt 4 (Abbildung 3.11d) ist eine Erweiterung von Welt 3. Es wurden zusätzliche Barrieren aufgenommen. Die Hindernisse weisen unregelmäßige Strukturen auf und sind nun nicht mehr horizontal oder vertikal ausgerichtet sondern beliebig angeordnet. Diese Welt erlaubt es aber immer noch, mit einem Wandverfolgungsverfahren durch sie hindurch zu navigieren.

Der Startpunkt für den Roboter ist für alle Verfahren im unteren rechten Bereich der Welt wie in den Abbildungen 3.11 dargestellt. Sobald eine Kollision mit einer Wand auftritt, wird der Roboter wieder an den Startpunkt zurückgesetzt. Der Lernalgorithmus läuft dabei unverändert weiter. Das jeweilige Lernverfahren wird trainiert den Steuerungsparameter $cont$ abzuändern. Die Einstellung der Radmotoren wird aus diesem Wert nach folgenden Formeln berechnet:

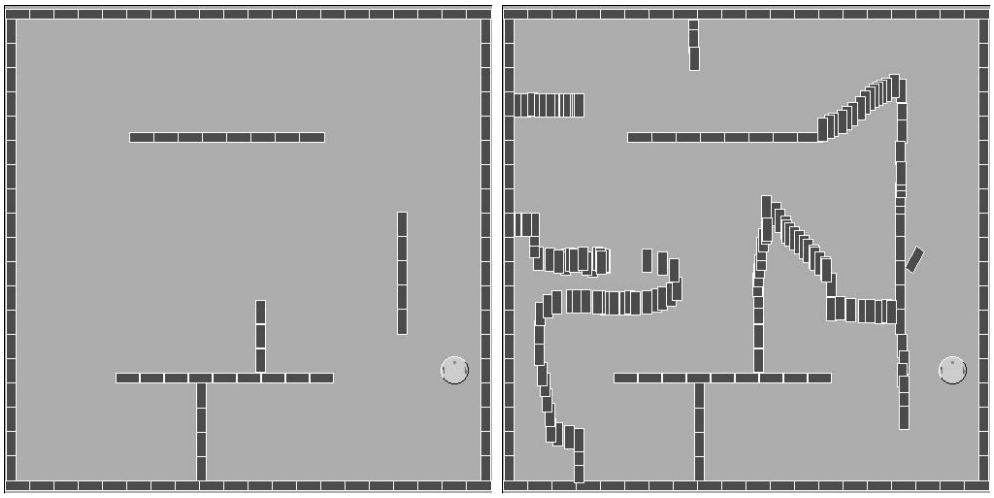
$$rad_{links} = forward - cont * turn \tag{3.26}$$

$$rad_{rechts} = forward + cont * turn \tag{3.27}$$



(a) Umgebung 1 (Empty)

(b) Umgebung 2 (Spiral) - mittlere Komplexität



(c) Umgebung 3 (Labo) - mittlere Komplexität

(d) Umgebung 4 (Maze) - hohe Komplexität

Abbildung 3.11: Die für die Versuche verwendeten Umgebungen

mit $forward = 0.9$ und $turn = 5$. Da zur Ansteuerung des Roboters lediglich ganzzahlige Werte erlaubt sind, wird das Ergebnis auf einen ganzen Wert gerundet. Mit dieser Art der Steuerung wird dem Roboter immer eine Vorwärtsbewegung aufgezwungen. Der Roboter kann über $cont$ nur die Drehung bestimmen. Der Parameter $cont$ wird so gewählt, dass er immer im Bereich zwischen -1 und 1 liegt. Wenn $cont = 0$ ist ergibt sich für beide Radmotoren der Wert 1 $(1; 1)$, d.h. der Roboter fährt geradeaus. Bei $cont = -1$ ergibt sich $(6; -4)$ also eine Drehung nach rechts, bei $count = 1$ erhält man $(-4; 6)$ also eine Linksdrehung.

Alle Verfahren wurden jeweils mehrmals mit unterschiedlichen Ausgangs-Initialisierungen (d.h. unterschiedlichen Initialisierungen des Zufallsgenerators, der bestimmend für die Vergabe der Gewichte ist) durchgeführt. Für jeden Versuch wurden 150.000 Schritte zurückgelegt.

3.5.3 Versuche zum Reinforcement-Lernen

Implementation des RI-Lernens zur Steuerung

Hier soll das Modell-/Controller-basierte Reinforcement-Lernen angewendet werden, um einen Roboter durch verschiedene Umgebungen zu steuern. Das Verfahren wurde wie in Abschnitt 3.2.1 beschrieben implementiert:

Das Controller-Netz besteht aus 3 Schichten: der Eingabeschicht mit den 8 Sensorwerten des Roboters als Input, einer versteckten Schicht mit 4 Neuronen und einer Ausgabeschicht mit einem Neuron. Bei dem Neuron der Ausgabeschicht handelt es sich um ein Neuron des Types `CStochasticBackpropNeuron` wie in Abschnitt 2.4.3 beschrieben. Die Ausgabe $cont$ des Controller-Netzes steuert den Roboter gemäß den Gleichungen 3.26 und 3.27.

Das Modell-Netz besteht ebenfalls aus 3 Schichten: einer Eingabeschicht mit den 8 Sensorwerten und dem Ausgabeneuron des Controller-Netzes[18], einer verdeckten Schicht mit 5 Neuronen und einer Ausgabeschicht mit einem Neuron zur Vorhersage des Reinforcement $\widehat{r}$.

Für diesen und die folgenden Versuche wurde ein Reinforcement 1 immer dann vergeben, wenn mindestens ein Sensorwert des Roboters den Wert von 1000 überschritten hatte - dies wird als Kollision betrachtet. Für alle übrigen Fälle wurde -1 als Reinforcement eingestellt. Das gewünschte Reinforcement r^* wird mit -1 vorgegeben. Die Art der Verteilung des Reinforcements kann man sich erklären, wenn man das Signal

[18]Die tatsächliche Implementation auf Basis von Neuronica verwendet zwei Eingabeschichten, eine Schicht mit 8 Neuronen für die Sensoren und eine Schicht mit einem Neuron für den Output des Controllers. Diese Struktur ist aber zu einer Eingabeschicht mit 9 Neuronen äquivalent.

als Schmerzsignal versteht. 1 bedeutet das Schmerzsignal ist an, -1 es ist abgeschaltet. Die Vergabe kann selbstverständlich auch umgekehrt erfolgen, dann muss lediglich r^* entsprechend angepasst werden. Die Reinforcementvergabe kann auch auf Basis einer Kollisionsdetektion erfolgen. Zuverlässig funktioniert diese aber nur im Simulator. Entsprechende Tests des Lernverfahrens auf dieser Basis verliefen erfolgreich.

Zum Training des Prognose-Netzes wurde wie in Abschnitt 3.2.1 beschrieben der Prognosefehler $(r - \widehat{r})$ und zum Lernen des Controllers der Fehler im Verhalten $(r^* - r)$ verwendet.

Als Aktivierungsfunktion für das Backpropagation-Verfahren wurde $\tanh$ verwendet. Als anfängliche Lernrate wurden Werte von $0,4$ bis $0,025$ verwendet. Im Laufe des Lernens wurde die Lernrate auf bis zu $0,004$ abgekühlt. Für die meisten Versuche wurde die Lernrate des Controllers niedriger als die des Modells eingestellt. Die Wahl einer eher kleinen Lernrate hat sich als sinnvoll erwiesen um korrekt zu lernen. Netz-Fehler die kleiner als $0,02$ waren, wurden auf 0 gesetzt und somit nicht zum Lernen verwendet.

Durchführung und Auswertung

Zunächst wurde das Verfahren wie in Abschnitt 3.2.1 beschrieben durchgeführt. Dabei wurde die Temperatur (als Parameter für die Stochastizität) des Ausgabeneurons im Verlauf des Lernens so abgekühlt, dass zu Beginn des Lernens eine rein zufällige Bewegung des Roboters erzeugt wurde. Im Verlauf des Trainings wurde dann zu einem durch den Output des eigentlichen Controller-Netzes determinierten Verhalten übergegangen ($temp \rightarrow 0$). Der Versuch wurde in den verschiedenen Welten jeweils mehrfach wiederholt. Trotz der Wahl verschiedenster Einstellungen für die Lernparameter und die Temperatur konnte kein Netz erzeugt werden, das nach 150.000 Schritten das Reinforcement-Modell genügend gut angelernt hatte, um damit den Controller befriedigend anzulernen. Abbildung 3.12 zeigt eine Trajektorie, in der ca. 700.000 Schritte des Roboters aufgezeichnet sind. Die Temperatur wurde dabei von 100 so abgekühlt, dass für die ersten 100.000-200.000 Schritte ein im Prinzip rein zufälliges Verhalten erzeugt wurde. Dadurch stand genügend Zeit zum Erlernen des Modells zur Verfügung. Der Roboter lernt es wie gewünscht, die Wand zu vermeiden. Zunächst bildet sich ein Wandverfolgungsverhalten (wobei die Sensoren auf niedrige Werte eingestellt sind) heraus und danach beginnt der Roboter im Freiraum zu rotieren und erfüllt so die Bedingung nicht mit einer Wand zu kollidieren.

Aufgrund der Probleme ein ausreichend schnelles Lernverfahren zu finden, wurden Modell und Controller zunächst 150.000 Schritte lang angelernt. Dabei wurde ein Startpunkt und eine Umgebung gewählt, die zum Erlernen der Abhängigkeit des Reinforcements von den Controller-Vorgaben geeignet schien. Die Steuerung erfolgte durch zufällige Wahl des Parameter $cont$. Nach einer Kollision wurde die Ausrichtung des

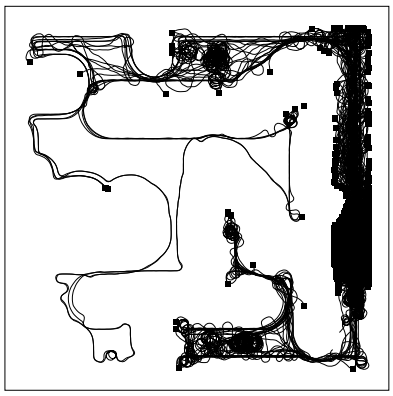


Abbildung 3.12: Trajektorie nach 700.000 Schritten mit dem RI-Prinzip

Roboters jeweils beibehalten. Mit den so trainierten Netzen wurde der Roboter dann 150.000 Schritte lang in den jeweiligen Welten gesteuert. Dabei wurde die Temperatur für das Steuerungsneuron auf 0 gesetzt, so dass eine rein deterministische Erzeugung von $cont$ erfolgte.

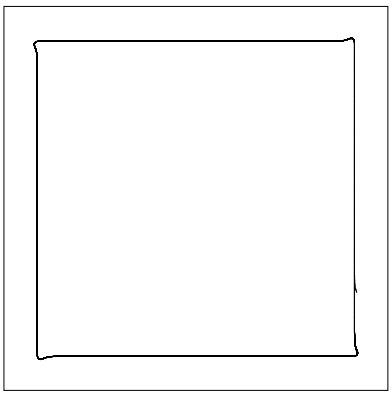
Abbildung 3.13 zeigt Trajektorien für die einzelnen Umgebungen. Das in den Abbildungen dargestellte Wandverfolgungsverhalten ergab sich nicht bei allen untersuchten Netzen. Oftmals steuerte der Roboter lediglich von der Wand weg und verfiel dann im freien Raum in eine Drehbewegung. Damit genügte er den beiden Anforderungen sich zu bewegen (ergibt sich aus der Art der Steuerung) und nicht mit der Wand zu kollidieren (ergibt sich nach dem Reinforcement-Prinzip). Dass sich mit der Vergabe eines Reinforcement-Signals für die Kollision mit einer Wand (bzw. falls zu nah an der Wand) überhaupt ein Wandverfolgungsverhalten ergibt war nicht unbedingt zu erwarten. Es kann aber durch die Tatsache erklärt werden, dass das Training in einem engem Raum erfolgte (Maze), in dem das Entlangfahren an einer Wand eine recht gute Strategie zur Fehlervermeidung ist.

Versuche mit dem realen Roboter

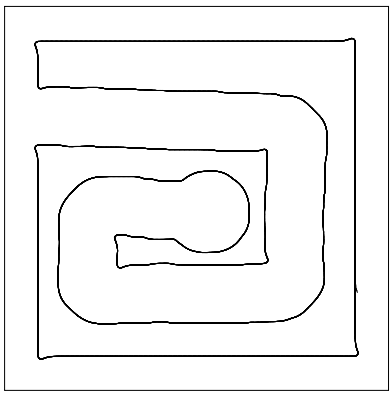
Bei den Versuchen mit dem realen Roboter ergaben sich keine wesentlichen Unterschiede zum Simulator. Lediglich das Erlernen des Modells schien etwas schneller (bezogen auf die Zahl der Schritte) zu verlaufen als im Simulator. Die auf dem Modell basierende Steuerung scheint allerdings im Simulator etwas besser zu sein.

Verdeutlichung der Lern-Problematik an einem einfachen Reinforcement-Problem

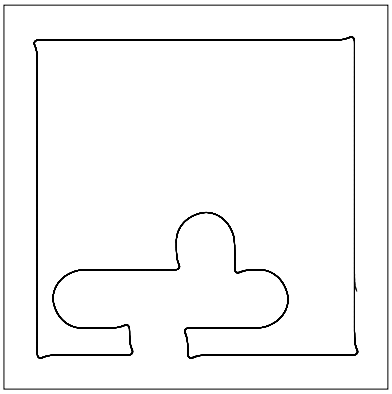
Hier wird zur Verdeutlichung der Problemstellung und des Lernverfahrens die Modellbildung beim Reinforcement-Lernen an einem einfachen Beispiel untersucht: Be-



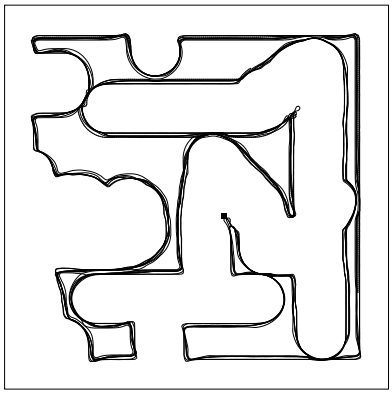
(a) Welt 1: exakte Wandverfolgung



(b) Welt 2: ebenfalls korrekte Wandverfolgung, keine Kollisionen



(c) gute Wandverfolgung in Welt 3



(d) Welt 4: Wandverfolgung weniger exakt, eine Kollision

Abbildung 3.13: Trajektorien des Roboters nach dem Reinforcement-Lernen

trachtet wird ein Roboter, der nur über einen Parameter, die Geschwindigkeit v, gesteuert wird. v kann in einem festgelegten Bereich ($v_{min}...v_{max}$) sowohl negative, als auch positive Werte annehmen. Es ist nur eine geradlinige Vorwärts- oder Rückwärtsbewegung möglich. Der Roboter wird nun zwischen zwei Wände positioniert (siehe Abbildung 3.14) und soll auf Basis seiner Sensorwerte und der Geschwindigkeit v das

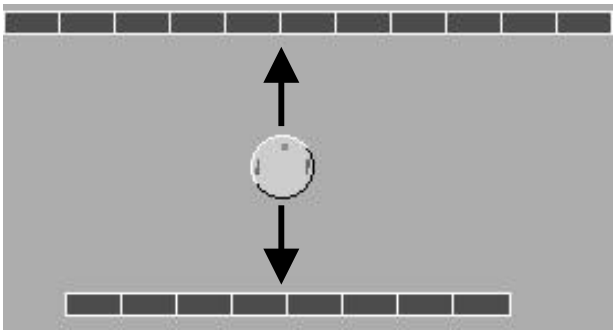
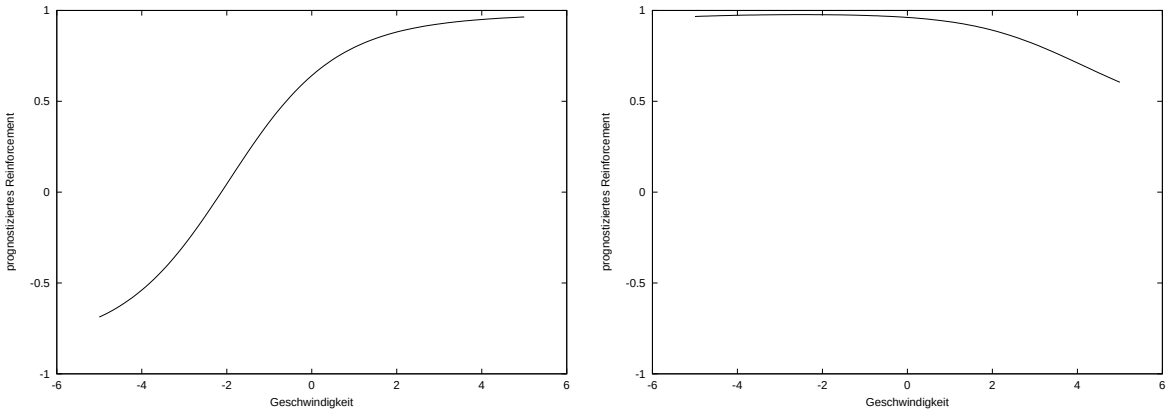


Abbildung 3.14: Einfaches Experiment zum Erlernen der Reinforcement-Prädiktion

Reinforcement nach dem nächsten Schritt vorhersagen. Interessant für das beabsichtigte Reinforcement-Lernen ist dabei die Abhängigkeit der Vorhersage des Reinforcement von der Geschwindigkeit v. Abbildung 3.15 zeigt diese Abhängigkeit für verschiedene Stellungen des Roboters nach erfolgreichem Lernen des Modells. Nur wenn diese Infor-



(a) Erwartetes Reinforcement im nächsten Schritt unmittelbar nach einer Kollision im Frontbereich

(b) Erwartetes Reinforcement unmittelbar nach einer Kollision im Heckbereich - unexaktere Prognose durch niedrigere Sensorauflösung

Abbildung 3.15: Erwartetes Reinforcement in Abhängigkeit von der Geschwindigkeit bei unterschiedlichen Positionen

mationen zumindest in die richtige Richtung verweisen (d.h. Rückwärtsfahren bringt

günstigeres Reinforcement als Vorwärtsfahren, wenn der Roboter vor der Wand steht und umgekehrt), können die Daten korrekt zum Lernen des Controllers benutzt werden.

In dem Versuch wird die Geschwindigkeit v von "außen" vorgegeben. Negative Werte bedeuten dabei rückwärts fahren. Die Anfangsgeschwindigkeit v_0 wird auf 0 gesetzt. Nach jedem Schritt t wird nun die Geschwindigkeit v_{t+1} neu bestimmt. Dabei gilt $v_{t+1} = v_t + x$ wobei $v_{t+1} = v_{max}$ gesetzt wird, falls $v_{t+1} > v_{max}$, $v_{t+1} = v_{min}$ gesetzt wird, falls $v_{t+1} < v_{min}$ und x ein Zufallswert zwischen $-x_{max}$ und x_{max} (mit $x_{max} \ll (v_{max} - v_{min})$) ist. Abbildung 3.16 zeigt den Verlauf der Abhängigkeit des erwarteten Reinforcements $\widehat{r}$ von der Geschwindigkeit jeweils für die Position unmittelbar vor einer Wand und mit der Wand im Heck. Dazu wird in jedem Schritt das erwartete Reinforcement $\widehat{r}$ jeweils für typische Sensorwerte unmittelbar vor der Wand ($sens_{front}$) und für typische Werte unmittelbar vor einer Kollision im Heckbereich ($sens_{back}$) jeweils für die minimale und die maximale Geschwindigkeit berechnet und die Differenzen dep_{front} und dep_{back} sowohl für den Front-als auch für den Heckbereich ausgegeben, mit

$$dep_{front} = \widehat{r}(v_{max}, sens_{front}) - \widehat{r}(v_{min}, sens_{front})$$

und

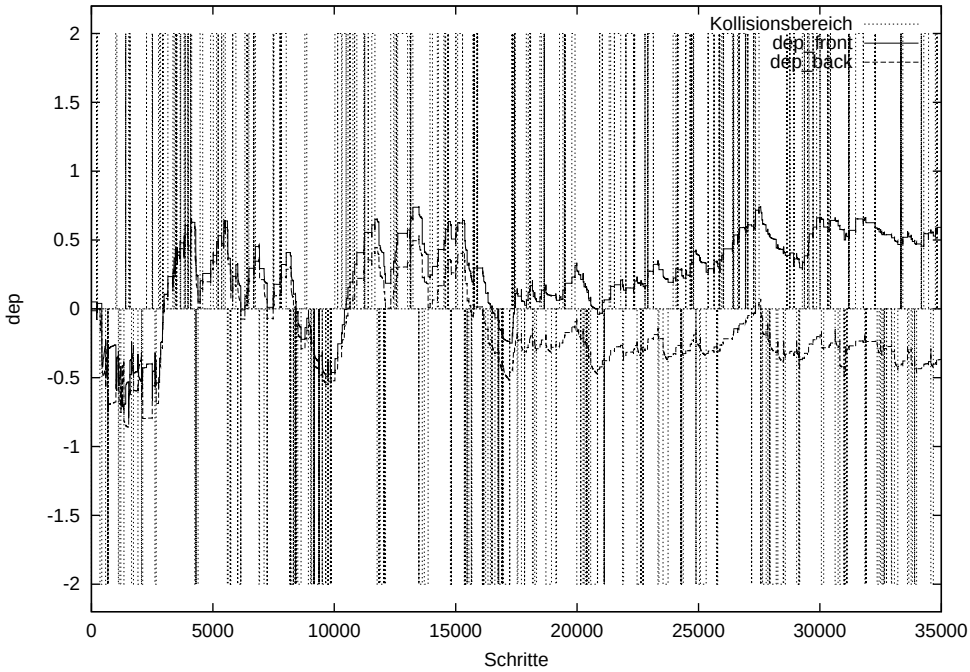
$$dep_{back} = \widehat{r}(v_{max}, sens_{back}) - \widehat{r}(v_{min}, sens_{back})$$

. Der Wert für den Kollisionsbereich wurde festgelegt mit 2, falls sich der Roboter in einem Bereich befindet, in dem Kollisionen im Frontbereich auftreten können (Frontkollisionsbereich), -2 für eine Position in einem Bereich, bei dem Kollisionen im Heckbereich auftreten können (Heckkollisionsbereich), und 0 für alle sonstigen Positionen.

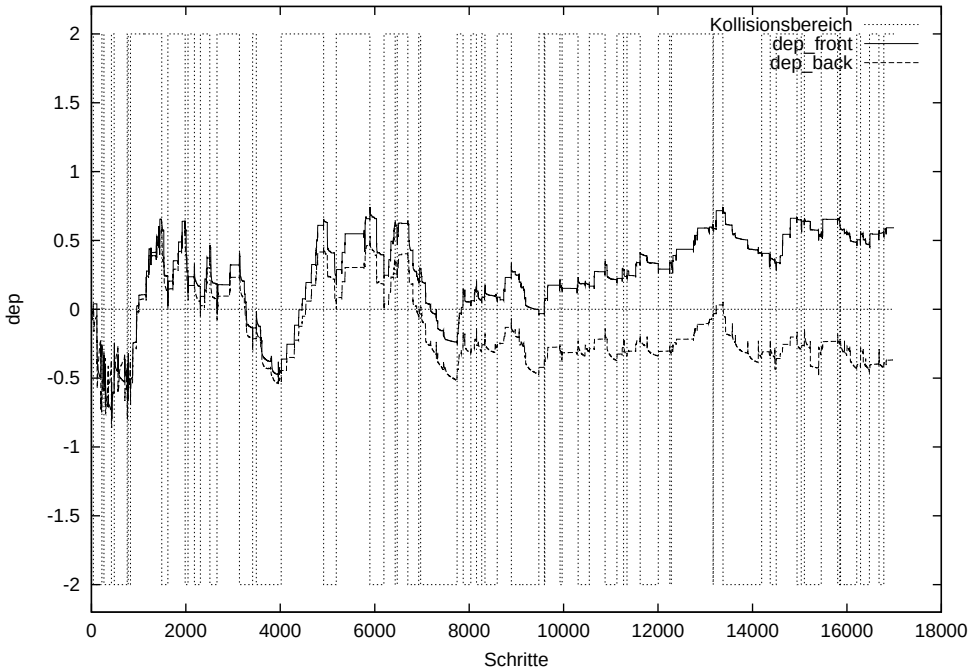
Der Roboter muss nun im Laufe des Versuches lernen, dass es günstiger ist, im Frontkollisionsbereich rückwärts zu fahren (also $dep_{front} > 0$) und im Heckkollisionsbereich vorwärts zu fahren (also $dep_{back} < 0$). Wie Abbildung 3.16 zeigt, sind dies zwei gegensätzliche Ziele. Am Anfang entwickeln sich dep_{front} und dep_{back} synchron, erst nach relativ vielen Schritten beginnt das Netz den Einfluss der Geschwindigkeit in Abhängigkeit von der Position korrekt zu modellieren. Erschwerend kommt hinzu, dass nur relativ wenige Kollisionen (im Vergleich zur Gesamtschrittzahl) ausgelöst werden. Eine Kollision im Heckbereich folgt nie unmittelbar nach einer Kollision im Frontbereich und umgekehrt. Die Vielzahl der Schritte zwischen den beiden Ereignissen begünstigt das Entlernen und verlängert damit die Dauer des korrekten Lernens der modellierten Abhängigkeiten[19].

Die Verwendung eines steileren Anstiegs der Aktivierungsfunktion kann das Prognoseverhalten verbessern (siehe Abbildung 3.17) , verschlechtert aber gleichzeitig die Generalisierungsfähigkeit des Netzes.

[19]Wurden die Sensorwerte, die Controlerwerte und die zugehörigen Reinforcements zunächst gesammelt und dann in zufälliger Reihenfolge einem Netz zum Lernen präsentiert, verbesserte dies die Lerngeschwindigkeit z.T. erheblich.



(a) alle Schritte



(b) nur Schritte in den Kollisionsbereichen wurden aufgezeichnet

Abbildung 3.16: Entwicklung der Abhängigkeit des erwarteten Reinforcements von der Geschwindigkeit im Verlauf des Lernens

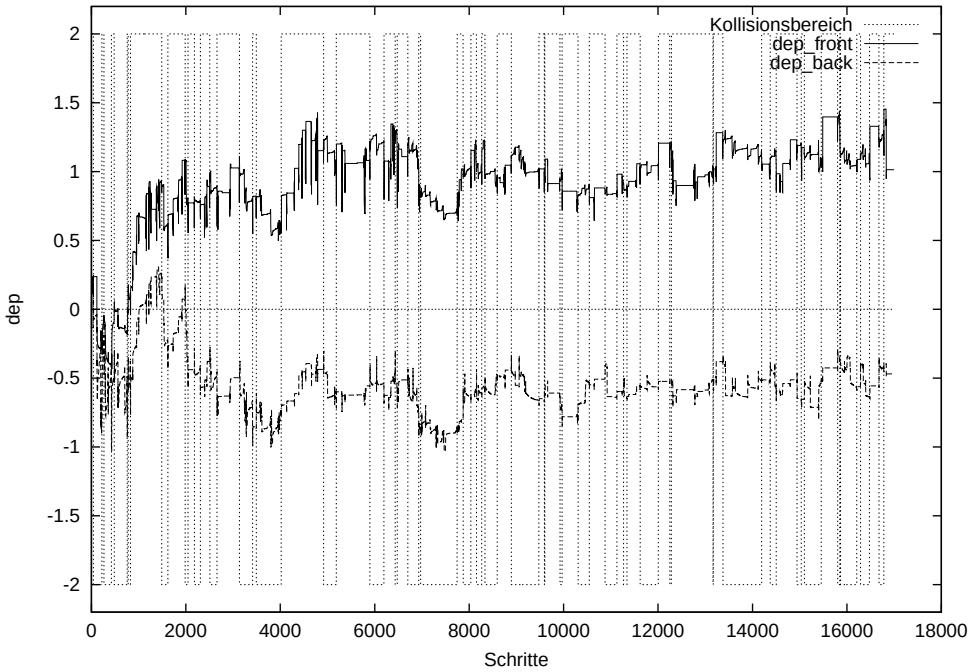


Abbildung 3.17: Versuch wie in Abbildung 3.16b, allerdings mit steilerer Aktivierungsfunktion (Faktor 3)

3.5.4 Versuche zum Homeokinese-Verfahren

Implementation

Die Implementation orientiert sich an Quellcodes von Professor Der[20] und hält sich im wesentlichen an die Vorgaben aus dem in Abschnitt 3.3 vorgestellten Algorithmus. Das Steuersignal $cont$ für den Roboter wurde ähnlich Gleichung 3.10 ermittelt mit

$$cont = tanh(\sum_i c_i sens_i + \gamma\phi(t))$$

mit der Störfunktion

$$\phi(t) = \gamma * cos(2\pi t * 0.1)$$

und $\gamma = 0.28$. Als Funktion für den Prädiktor wurde

$$\Phi = \widehat{\Delta sens} = 0$$

gewählt. Für die Ermittlung des Fehlers wurde gemäß Gleichungen 3.5 und 3.6

$$E = \sum_i \Delta sens^2$$

[20] siehe auch[DSP99]

verwendet. Die Änderung der Gewichte des Prädiktors erfolgte in Anlehnung an Gleichung 3.11[21] mit

$$\Delta c_i = -\eta sens_i \gamma \phi(t) E$$

und den oben angegebenen Parametern und $\eta = 0,7$ als Lernrate.

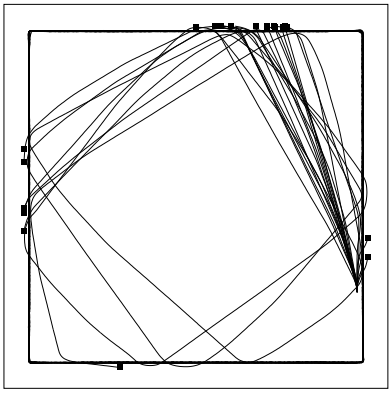
Durchführung und Auswertung

Der Versuch wurde in den verschiedenen Welten jeweils mehrmals wiederholt. Dazu wurden bei gleichen Ausgangsbedingungen die Anfangsgewichte des Controllers auf zufällige Werte im Bereich $[-0,05...0,05]$ gesetzt. In allen untersuchten Umgebungen ergaben sich mit diesem Verfahren nach wenigen Kollisionen meist ein kollisionsvermeidendes oder ein Wandverfolgungs-Verhalten. Ein weiteres seltener auftretendes Verhaltensmuster, war das Zusteuern und Kollidieren mit einer Wand. In der Spiral-Welt zeigte sich häufiger als in den anderen Welten ein Wandverfolgungsverfahren.

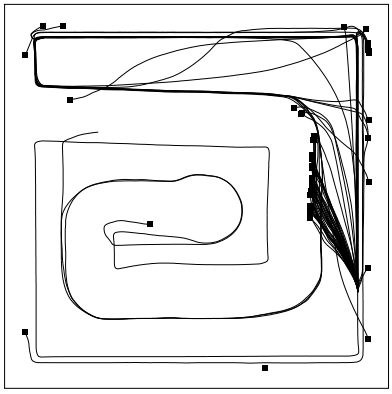
Die zwei wesentlichen Verhaltensmuster (Wandverfolgung und Kollisionsvermeidung) lassen sich mit der Art des Lernens erklären. Der Roboter erzeugt ein Verhalten, welches dem Selbstmodell genügt. Das Modell geht in der hier untersuchten Form davon aus, dass keine Änderung der Sensorwerte stattfindet. Beim Wandverfolgen an einer geraden Wand ergeben sich im Optimal-Fall keine Änderungen der Sensorwerte. Wenn der Roboter ein Kollisionsvermeidungs-Verhalten aufzeigt, bewegt er sich die meiste Zeit im Hindernis-freien Raum. D.h. es erfolgen ebenfalls keine Änderungen der Sensorwerte. Beim Ausweichen vor einem Hindernis kann der Roboter den Fehler minimieren, wenn er rechtzeitig ausweicht.

Der hier beschriebene Versuchsaufbau erschließt nicht das volle Potenzial des Algorithmus. Das untersuchte Prinzip kann auch dazu benutzt werden, dem Roboter bestimmte erwünschte Verhaltensweisen anzulernen. Erreichen kann man dies, indem man dem Roboter wiederholt bestimmte Sensormuster präsentiert, die für diese Verhaltensmuster typisch sind. Wenn man beispielsweise Wandverfolgung als Verhalten erreichen will, muss man den Roboter immer wieder im gewünschten Abstand parallel zur Wand positionieren. Der Roboter lernt den Abstand einzuhalten, da dies eine optimale Strategie im Sinne des Algorithmus ist, weil damit die Sensorwerte möglichst wenig geändert werden und so die Modellfunktion $\Phi = 0$ gut abgebildet werden kann.

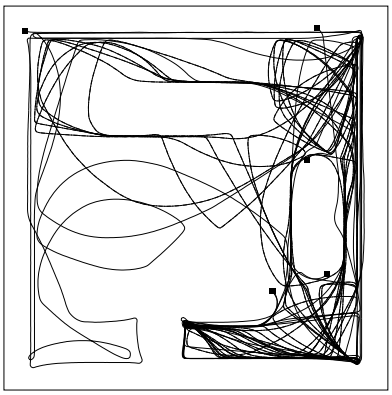
[21] Die Verwendung der Originalgleichungen 3.10 und 3.11 brachte bei den Experimenten für diesen Algorithmus keine Verbesserung. Es wurde deshalb die Version verwendet, die auch von Prof. Der in praktischen Tests verwendet wurde.



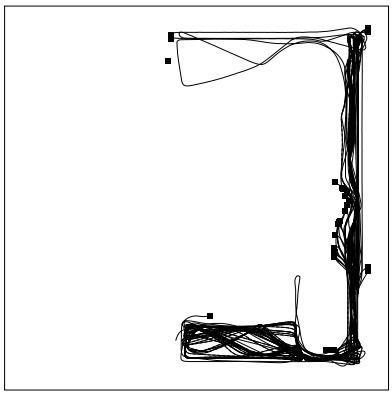
(a) Welt 1: zuerst Kollisionsvermeidung danach Wandverfolgung



(b) Welt 2: Nach anfänglichen Kollisionen bildet sich Wandverfolgung aus. Zum Schluss dringt der Roboter auch in den bisher unerkundeten Teil der Karte vor.



(c) Welt 3: Wandverfolgung und Kollisionsvermeidung



(d) Welt 4: Kollisionsvermeidung und Wandverfolgung

Abbildung 3.18: Typische Trajektorien des Roboters beim Lernen nach dem Homeokinese-Verfahren

Versuche mit dem realen Roboter

Die hier beschriebene Implementation wurde mit Ansteuerung über den Khepera-Simulator an einem realen Roboter untersucht. Dabei wurde eine leicht abgewandelte Form des Algorithmus benutzt, der die Signallaufzeiten berücksichtigt. Die Lernergebnisse waren dabei im Durchschnitt besser als im Simulator. In der realen Welt kann man darüber hinaus weitere Versuche durchführen. So konnte der Roboter so trainiert werden, dass dieser einen Ball (bzw. ein Stück Styropor) vor sich her schiebt. Dies wurde erreicht, indem der Ball immer vor dem Roboter platziert wurde, so dass sich der Roboter an die dadurch entstehenden Sensorwerte "gewöhnen" und sein kinetisches Regime darauf einstellen konnte.

Mit dem realen Roboter ist es oftmals wesentlich leichter bestimmte gewünschte Verhalten zu erzeugen, da man Roboter und Hindernisse einfacher positionieren kann. So kann man, um Wandverfolgung zu erreichen, ein Hindernis als Wand neben dem Roboter her führen.

3.5.5 Versuche zum Lernen nach der netzbasierten Homeokinese-Idee

Implementation

Das Verfahren wurde wie in Abschnitt 3.4.2 beschrieben implementiert.

Für den Versuch wurde ein einfacher Netzaufbau mit insgesamt 4 Schichten verwendet. Die erste Schicht besteht aus 8 Input-Neuronen, die die Sensorwerte des Robobters aufnehmen. Die zweite verdeckte Schicht besteht aus 4 Backprop-Neuronen und ist mit der ersten und der Ausgabeschicht verbunden und zählt logisch gesehen zum Modell-Teil des Netzes. In der 3. Schicht befindet sich nur ein verdecktes Neuron, dessen Ausgabewert als Signal $cont$ zur Ansteuerung des Roboters benutzt wird. Davon abgesehen verhält es sich wie jedes andere Backprop-Neuron auch. Es ist mit der ersten und der Ausgabeschicht verbunden. Die Ausgabeschicht besteht aus 8 Neuronen und gibt die prognostizierten Sensorwerte $\widehat{sens}$ aus. Abbildung 3.19 stellt das beschriebene Netz dar.

Das Netz wird trainiert, den Fehler in der Vorhersage $(sens - \widehat{sens})$ zu minimieren. Der beim Vorwärtspropagieren der Netz-Eingabe am Neuron der 3. Schicht anfallende Output $cont$ wird zur Ansteuerung des Roboters nach den Gleichungen 3.26 und 3.27 verwendet.

Für das Netz wurde $\tanh$ als Aktivierungsfunktion verwendet. Als Lernrate wurde $0,3$ gewählt.

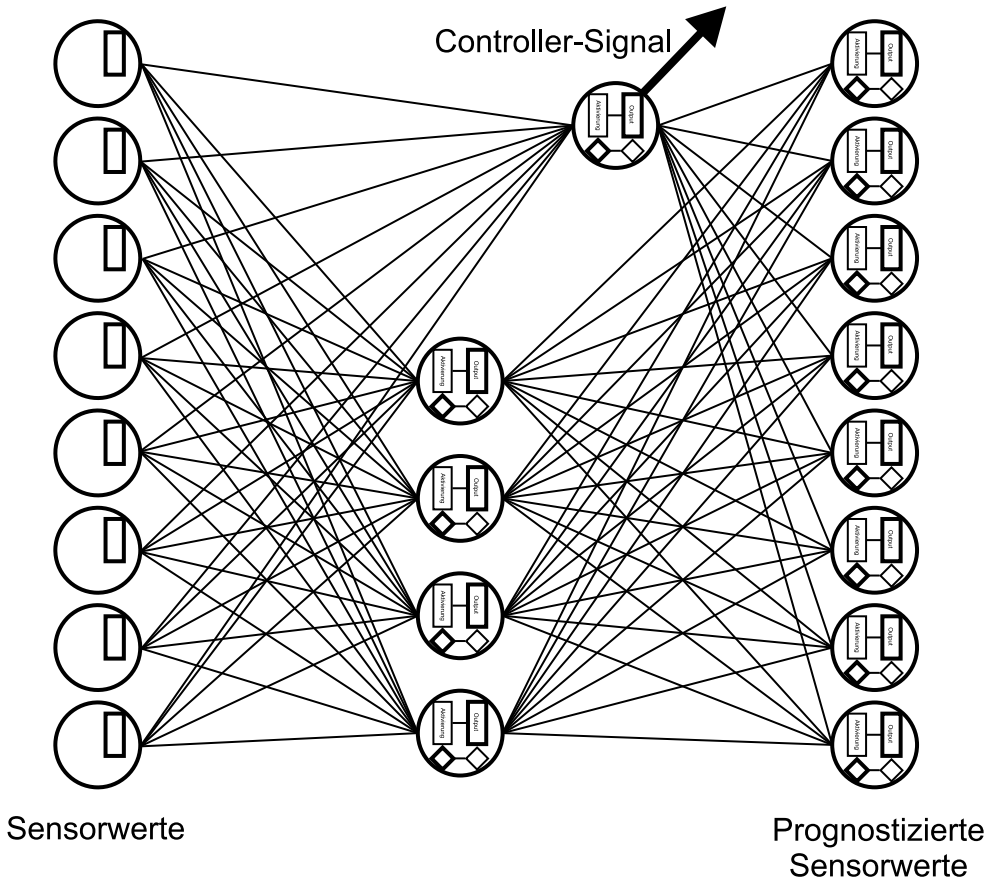
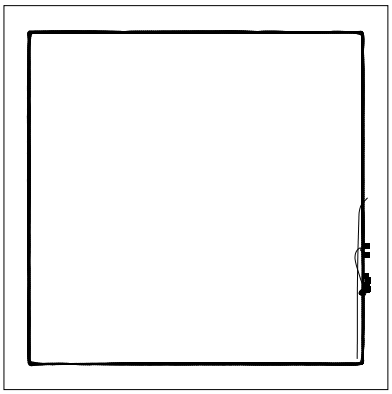


Abbildung 3.19: Aufbau des Netzes für das Experiment in Abschnitt 3.5.5

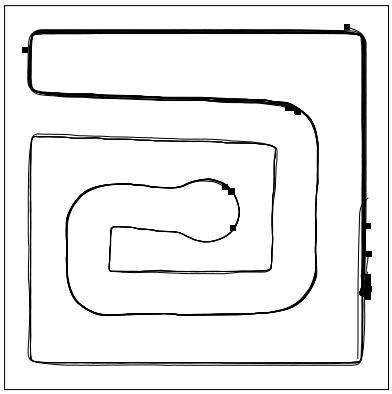
Durchführung und Auswertung

Wie bei den anderen Verfahren auch wurde jeder Versuch mehrmals mit verschiedenen Initialisierungen wiederholt.

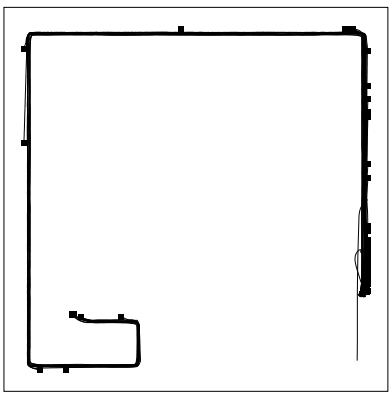
In der leeren Welt 1 kollidiert der Roboter in allen Versuchen zunächst mit der Wand. Nach einigen Kollisionen und Drehungen um die eigene Achse bildete sich bei fast allen Versuchen ein stabiles Wandverfolgungsverhalten aus. In einigen Versuchen bewegt sich der Roboter weg von der Wand und rotiert fortan nur noch um die eigene Achse. Dieses Verhalten ergibt sich unabhängig von der gewählten Welt. Welt 2 hat nur runde



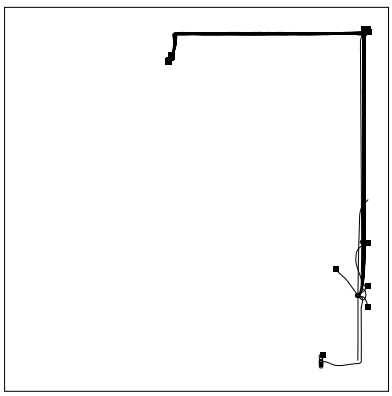
(a) Stabile Wandverfolgung in Welt 1



(b) Welt 2: anfänglich Kollisionen, danach relativ stabile Wandverfolgung



(c) Welt 3: Wandverfolgung nur bis zu bestimmten kritischen Punkten (herausragende Kanten)



(d) In Welt 4 erfolgt die Wandverfolgung nur bis zur ersten herausragenden Kante

Abbildung 3.20: Typische Trajektorien des Roboters für die verschiedenen Welten

Ecken. Auch hier bildet sich nach anfänglichen Kollisionen bei den meisten Versuchen schnell ein Wandverfolgungsverhalten aus. Bei dem durch Abbildung 3.20b gezeigten Versuch ereignen sich Kollisionen vor allem an der Ecke, an der der Roboter eine Rechtsdrehung vollziehen müsste.

Auch in Welt 3 und Welt 4 bildet sich bei vielen Versuchen ein Wandverfolgungsverhalten aus. Allerdings ergibt sich meist keine kollisionsfreie Wandverfolgung. In den, durch die Trajektorie in Abbildung 3.20c und 3.20d erfolgt über die Zeit des Versuches jeweils an der ersten herausragenden Ecke eine Kollision.

Versuche mit dem realen Roboter

Die Ergebnisse bei den Versuchen mit dem realen Roboter glichen im Wesentlichen den Ergebnissen im Simulator. Bei den Versuchen mit dem Khepera-Simulator hat sich häufig ein Wandverfolgungsverhalten ausgebildet, bei dem der Roboter zwischen zwei stark unterschiedlichen Sensorwerten, die nach jeweils einen Schritt wechselten, hin und her pendelte. Die Sensoren des realen Roboters besitzen nicht ein solches "sprunghaftes" Verhalten, so dass dieses Verhalten dort nicht beobachtet werden konnte. Ein Pendeln an der Wand entlang konnte mitunter auch festgestellt werden.

3.6 Auswertung und Vergleich der Steuerungsmechanismen

3.6.1 Überblick

Die hier untersuchten Lernverfahren verbindet die Eigenschaft, dass für die Steuerung des Roboters direkt oder indirekt ein Modell der Welt und/oder des Roboters verwendet wird. Die Unterscheidung nach Selbstmodell und Weltmodell muss man dabei eigentlich gar nicht treffen. Betrachtet man alle Signale, die der Roboter als Basis seiner Informationsverarbeitung nutzen kann (also auch interne Werte und beispielsweise Reinforcement-Signale), als Sensorwerte und versteht man ferner unter dem Selbstmodell des Roboters die Modellierung der zukünftig zu erwartenden Sensorwerte unter Beachtung der Auswirkungen der Effektoren, so ist ein Selbstmodell gleichzeitig auch ein Weltmodell. Genauer gesagt ist das Selbstmodell dann auch ein Modell für den, für den Roboter relevanten Bereich der Welt. Der Aufbau eines Weltmodells durch den Roboter kann nur auf Basis der Sensordaten erfolgen. Um zukünftige Sensorwerte zu prognostizieren muss der Einfluss der Umwelt auf diese Daten beachtet werden. Einflüsse, die sich nicht in den Sensordaten widerspiegeln, sind für den Roboter nicht relevant. Ein Weltmodell ist nach diesem Ansatz immer dann auch ein Selbstmodell, wenn der

Einfluss der Effektoren mit beachtet wird. Davon kann man aber bei einem in der Umwelt agierenden Roboter ausgehen. Die in dieser Arbeit untersuchten Verfahren haben einander also gemein, dass ein Selbstmodell zum Adaptieren des Verhaltens benutzt wird.

Den hier untersuchten Algorithmen ist weiterhin gemein, dass das Modell Werte prognostiziert, die nur einen Zeitschritt in der Zukunft liegen. Das ist insbesondere beim Reinforcement-Lernen als negativ zu bewerten, da dort für den Controller relevante Lernsignale nur an wenigen Stellen im Zustandsraum produziert werden. Für dieses Verfahren ist es nötig, Zustände zu bewerten, die nicht in direkter Nachbarschaft zu Zuständen liegen, in denen ein Reinforcement erzeugt wird. Bei den untersuchten Steuerungsaufgaben handelt es sich um Probleme mit verzögerter Reinforcement-Vergabe. Das beschriebene Reinforcement-Lernverfahren kann aber nur Probleme mit direkter Reinforcement-Vergabe korrekt lösen. Dass der Algorithmus trotzdem funktioniert liegt in der Generalisierungsfähigkeit der verwendeten neuronalen Netze begründet. Um eine Bewertung der einzelnen Zustände zu erreichen, bietet es sich an das Temporal Difference-Lernen von Sutton [Sut88] in den beschriebenen Algorithmus zu integrieren. Damit kann das Verfahren trainiert werden, auch mit verzögerten Reinforcements zurecht zu kommen.

3.6.2 Vergleichende Auswertung der Experimente

Reinforcement-Lernen

Reinforcement-Lernen nach dem Modell-Controller-basierten Ansatz hat bei den durchgeführten Experimenten dann stets gut funktioniert, wenn das Modell-Netz bereits gut angelernt war. Das gleichzeitige Lernen von Modell und Controller unter Benutzung eines Neurons mit stochastischem Output zur Erzeugung des Steuersignals, wie u.a. in [HKP91, S.194ff] beschrieben, hat sich in der Anwendung als teilweise nicht praktikabel erwiesen. Das Erlernen des (Vorwärts-)Modells kann, selbst bei einfach erscheinenden Zusammenhängen, eine komplexe Aufgabe sein. Für das Training des Modell-Netzes zur Verwendung durch den Simulator nach dem hier beschriebenen Verfahren sind nur die Schritte interessant, bei denen eine Änderung der Controller-Parameter eine Änderung des Reinforecements erzeugen könnte. Wenn nur eine geringe Zahl von Punkten diese Bedingung erfüllt, so erfolgt das für die Erzeugung des Steuersignals wichtige Lernen der Abhängigkeit des Reinforcements von den Controller-Signalen, bezogen auf die Gesamtschrittzahl nur sehr langsam. Die Dauer bis das Modell-Netz zum Anlernen des Controllers tatsächlich geeignet ist, variiert je nach verwendeter Lernparameter und Komplexität des zu erlernenden Zusammenhangs zum Teil erheblich.

Wird zu früh von einem zufälligen zu einem durch den Controller determinierten Verhalten übergegangen, so erfolgt unter Umständen ein Training des Controllers, welches

unerwünschte nicht mit den Vorgaben zu vereinbarende Verhaltensweisen erzeugt. Es kann passieren, dass der Controller ein solches Verhalten generiert, dass bestimmte für das Lernen des Modells nötige Sensormuster nicht mehr auftreten. Damit steckt das Lernen für das Modell und damit auch für den Controller in einer Sackgasse. Es erfolgt keine weitere Verbesserung.

Bei den im Rahmen dieser Diplomarbeit unternommenen Versuchen hat es sich als am günstigsten heraus gestellt, das Modell (oder Modell und Controller) zunächst mit einem vollkommen zufälligen Controller-Output anzulernen. Nach dieser Anlernphase wird dann direkt auf ein nur durch den Controller bestimmtes Verhalten umgeschaltet.

Der Zeitpunkt bzw. die Zeitspanne, in denen von einem stochastischen zu einem deterministischen Verhalten übergegangen wird, ist entscheidend für den Erfolg des Lernverfahrens. Der Fehler des Modell-Netzes eignet sich nur sehr bedingt, diesen Zeitpunkt zu ermitteln, da der Fehler entscheidend von der Umgebung und den Steuerungsparametern abhängt. So kann ein, in Bezug auf den Einfluss des Controller-Signales schlecht trainiertes Netz, durchaus einen niedrigen Fehler liefern, z.B. dann wenn der Einfluss des Controllers auf das zu erwartende Reinforcement nur gering ist. Ein Parameter der sich besser zur Bestimmung der Temperatur des Controller-Outputs (und damit der Stochastizität) eignet, ist das Reinforcement-Signal selbst. Wurde zu einem eher deterministisch bestimmten Verhalten übergegangen und treten wiederholt oder gehäuft negative Reinforcement-Signale auf, kann dies ein Zeichen sein, dass das Modell noch nicht hinreichend gut angelernt wurde. Die Temperatur muss dann wieder erhöht werden (eventuell genügt auch nur ein temporäres Erhöhen der Temperatur). In den unternommenen Versuchen konnte, wenn sich das Lernverfahren in einer oben beschriebenen Sackgasse befand, durch manuelle Erhöhung der Temperatur des Controller-Neurons wieder zu einem sinnvollen Lernverhalten zurück gefunden werden.

Kritisch ist bei diesem Ansatz anzumerken, dass es in der praktischen Anwendung häufig nicht machbar ist, die Steuerung mit rein zufälligen Parametern vorzunehmen. Eine solche Steuerung kann z.B. eine Gefahr für den Roboter selbst oder auch seine Umgebung darstellen. Ein anderes Paradigma zum Erlernen des Modells wäre daher wünschenswert.

Homeokinese

Mit dem Homeokinese-Verfahren konnten im praktischen Versuch verschiedene “vernünftige” Verhaltensmuster erzeugt werden, ohne dass Einfluss von außen genommen wurde. Diese Muster ergaben sich dabei meist nach wenigen Schritten. Dabei funktionierte das Verfahren am realen Roboter noch besser als im Simulator. Die durch den Algorithmus erzeugten Verhaltensmuster können beispielsweise als Basisverhalten in einem verhaltensbasierten Steuerungsansatz benutzt werden.

Bei der praktischen Anwendung ergibt sich manchmal das Problem, dass an ein Ver-

halten zusätzliche Restriktionen gebunden sind, oder ein sich entwickelndes Verhalten in eine bestimmte Richtung beeinflusst werden soll. Dafür scheint es sinnvoll, das Homeokinese-Verfahren um Reinforcement-Eigenschaften zu erweitern. Denkbar wäre zum Beispiel ein Ansatz, bei dem die Gewichte nicht nur nach dem Homeokinese-Algorithmus abgeändert werden, sondern zusätzlich eine Änderung erfolgt, die durch ein Reinforcement-Netz, wie oben beschrieben, bestimmt ist. Ein Gewichtungsparameter müsste dann bestimmen, wie groß der Einfluss der beiden beteiligten Verfahren auf das Gesamtverhalten ist.

Die schnelle Erzeugung von "vernünftigen" und gleichzeitig explorativen Verhaltensweisen macht das Homeokinese-Verfahren auch als Start-Verfahren zum Erlernen des Modells im Modell-/Controller-basierten-Reinforcement-Lernen interessant.

Modell-Controller-basiertes Lernen

Die Umsetzung der Homeokinese-Idee durch Modell-/Controller-basiertes Lernen, wie in Abschnitt 3.4.2 beschrieben, erzeugte im praktischen Versuch ähnliche Verhaltensmuster wie der Homeokinese-Algorithmus. Der Lernvorgang dauerte aber etwas länger. In der Mehrzahl der Fälle bildete sich ein Wandverfolgungsverhalten aus. Kollisionsvermeidung oder gezielte Kollision mit der Wand wurde weniger oft beobachtet. Der reale Roboter konnte, wie beim Homeokinese-Algorithmus auch, darauf trainiert werden ein Stück Styropor vor sich her zu führen.

Das dem Verfahren zugrunde liegende Selbstmodell ist komplizierter, als das triviale Modell, das bei der Umsetzung des Homeokinese-Algorithmus verwendet wurde. Damit ergibt sich, wie auch in [DSP99] argumentiert, dass die Bildung von "vernünftigen" Verhalten kein Beiprodukt der Verwendung einer einfachen Funktion als Selbstmodell ist.

Da sowohl das Modell-/Controller-basierte Reinforcement-Lernen als auch das hier beschriebene Verfahren als Spezialfall des in Abschnitt 3.4.1 vorgestellten Prinzipes gesehen werden können, bietet es sich an eine Kombination der beiden Verfahren wie in Abschnitt 3.4.4 skizziert zu untersuchen.

3.7 Biologische Motivation

Bei der Untersuchung grundlegender Verfahren zur Steuerung von Agenten, insbesondere dann, wenn neuronale Netzer als Teil der Steurerung verwendet werden, liegt immer auch der Vergleich mit dem biologischem Vorbild nahe.

Rick Grush argumentiert in [Gru95], dass bestimmte Bereiche im Zentralnervensystem (ZNS) [von Säugetieren], insbesondere im Klein- und Mittelhirn, als Modelle für die

muskuläre Dynamik fungieren. Er bezeichnet diese Modelle als Emulatoren. Diese Modelle erhalten, nach Grush, neben Informationen über das Zielsystem (=innerer Zustand + zu beeinflussende Sensoren), eine Kopie der vom ZNS an die Effektoren gesendeten efferenten Signale und modellieren die Auswirkungen dieser Signale auf das Zielsystem. Das System soll auch in der Lage sein, dass ein solcher Emulator nur die "Kopie" der efferenten Signale erhält, ohne dass diese tatsächlich an die Effektoren gesendet werden.

Der Emulator prognostiziert eine bestimmte Bewegung. Das Ergebnis der Prognose kann dann zum Beispiel als Korrektursignal benutzt werden. Auf diese Weise ist es möglich Vorgänge die so schnell ablaufen, dass normalerweise kein Feedback zur Steuerung benutzt werden kann (Open-Loop-Steuerung) mit einem Feedback zu versehen, der aber nicht aus der Beobachtung der realen Bewegung resultiert, sondern aus dem durch das Modell prognostizierten Wert (Pseudo-Closed-Loop-Steuerung). Als Beleg dafür wird u. a. ein Versuch angeführt, bei dem Personen aufgefordert werden plötzlich und so schnell wie möglich die Hand zu einem bestimmten Ziel zu führen. Dabei wurde beobachtet, dass bereits nach 70 ms Korrekturen an der Bewegungsbahn vorgenommen wurden. Diese Zeit ist aber so klein, dass diese Korrektur nicht durch ein proprioceptives (d.h. tiefensensibles) oder visuelles Feedback ausgelöst sein kann. Das Korrektursignal muss also intern erzeugt werden, unter Zuhilfenahme eines entsprechenden Modells.

Eine weitere Beobachtung, die die Hypothese von Grush unterstützt, stammt aus dem Forschungsbereich der Sport-Physiologie. Das wiederholte imaginäre Durchlaufen bestimmter physikalischer Aktivitäten verbessert die reale Performance der gedachten Aktivität. Dieses Phänomen ist erklärbar, wenn man davon ausgeht, dass normalerweise die Leistungsfähigkeit mit von außen stammen Feedbacks trainiert wird. Wenn man weiter annimmt, dass die durchdachte Aktivität genau wie die tatsächliche Aktivität initiiert wird, wobei die efferenten Signale unterdrückt werden und lediglich die Kopien an den Emulator gesandt werden, so kann man sich die verbesserte Performance dadurch erklären, dass die vom Emulator erzeugten Feedbacks als Trainingssignal benutzt werden.

Folgt man dem von Grush vorgestellten Prinzip existiert also im ZNS ein Mechanismus, der ein Steuerungssignal für die Effektoren produziert. Es existiert weiterhin ein Modell welches, neben allgemeinen Zustandsinformationen[22], Kopien der efferenten Steuerungsignale als Input erhält und auf dieser Basis eine Prognose des zukünftigen Zustands erstellt. Die Ausgabe des Modelsl wird des Weiteren dazu benutzt um den Steuerungsmechanismus zu trainieren.

Die Ähnlichkeit mit dem Aufbau der in dieser Diplomarbeit untersuchten Lernverfahren ist unverkennbar. Ob aber im ZNS auch ein ähnliches Prinzip zur Adaption des Steuerungsnetzes verwendet wird, ist eher fraglich.

[22] also auch (verarbeiteten) Sensorinformationen

Kapitel 4

Ergebnisse und Ausblick

4.1 Ergebnisse

Im Rahmen dieser Diplomarbeit, entstand die Klassenbibliothek Neuronica, ein voll einsatzfähiger Framework zur Simulation neuronaler Netze. Die Bibliothek unterstützt im aktuellen Entwicklungsstand Backpropagation-Lernverfahren und selbstorganisierende Karten. Durch die modulare Struktur ist sie eine ideale Basis für die Implementation weiterer Netztypen und Lernverfahren. Da darauf geachtet wurde, soweit wie möglich zum SNNS als Standard-Referenz kompatibel zu bleiben, lassen sich viele für den SNNS vorhandene Tools auch für Neuronica verwenden.

Die Struktur und Funktionalität der Netzwerkklassen wurden innnerhalb dieser Arbeit beschrieben und die einzelnen implementierten Verfahren wurden genauer untersucht. Schließlich wurde die Funktionalität der Bibliothek an realen Beispielen getestet.

Innerhalb dieser Diplomarbeit wurde weiterhin ein Lernverfahren entwickelt, das auf neuronalen Netzen aufbaut und nach der Homeokinese-Idee funktioniert. Dieses Verfahren hat sich in den durchgeführten Versuchen ebenso wie die beiden anderen untersuchten Verfahren als fähig erwiesen, einfache Verhaltensmuster (wie z.B. Wandverfolgung) für einen Roboter zu erzeugen. Es wurde ferner gezeigt, dass das Verfahren als ein Spezialfall des “Distal Supervised Learning”-Verfahrens von Jordan und Rumelhart betrachtet werden kann. Damit wurde gleichzeitig die Begründung für die Funktionalität erbracht.

Das untersuchte Reinforcement-Verfahren erwies sich in der implementierten Form als mäßig praktikabel. Das Potenzial des Verfahrens wurde aber dargestellt.

Das Homeokinese-Verfahren konnte wie erwartet benutzt werden, um einfache Verhaltensmuster zu erzeugen, dabei wurde festgestellt, dass sich ohne Eingriff von außen nicht in allen Fällen ein “vernünftiges” Verhalten ergab.

4.2 Ausblick

Die Netzwerk-Bibliothek Neuronica bildet eine gute Basis um weitere neuronale Funktionalität aufzubauen. Interessant und relativ einfach zu realisieren wäre beispielsweise die Implementation von Backpropagation Through Time oder Cascade Correlation Learning. Die Implementation von Backpropagtion mit Momentum-Term wurde bereits kurz in dieser Arbeit beschrieben und braucht nur noch umgesetzt werden.

Bei den Steuerungsmechanismen wäre eine Kombination des Reinforcement-Lernens mit dem Temporal Difference-Lernen von Sutton wünschenswert. Damit könnte das Verfahren auch mit verzögerten Reinforcements gute Lernergebnisse erzielen.

Ebenfalls sehr interessant scheint die bereits angerissene Kombination von Homeokinese und Reinforcement-Lernen.

Die Evaluierung der von Grush beschriebenen Theorie, der Verwendung von Emulatoren durch das ZNS ist ebenfalls eine interessante, wenn auch nicht unbedingt auf dem Feld der Informatik liegende Aufgabe. Auf diese Weise können aber vielleicht auch Rückschlüsse für die Lernverfahren für autonome Roboter gewonnen werden.

Anhang A

Neuronica-Klassenreferenz

Die Neuronica-Klassenreferenz in diesem Kapitel wurde automatisch aus den Kommentaren im Quellcode der Bibliothek erzeugt. Diese automatisch erzeugte Datei wurde dann nachbearbeitet und gekürzt. Aus Platzgründen werden hier nur die Basisklassen aufgelistet. Von diesen wurden wiederum nur wichtige Funktionen ausgewählt. Die vollständige Referenz im HTML-Format befindet sich, ebenso wie der Quellcode der Bibliothek, auf der zu dieser Diplomarbeit gehörigen CD.

A.1 *CNeuron* class reference (cbasicneuron.h)

Description: Basisklasse für die Implementierung eines Neurons

See Also: CBackpropNeuron , CInputNeuron , CKohonenNeuron

Basisklasse für die Implementierung eines Neurons. Diese Klasse selber ist nicht als anzuwendende Klasse gedacht, vielmehr bietet sie eine Basis für die Implementation einer spezialisierten Klasse.

Das Grundprinzip, das dieser Implementation zugrunde liegt, ist der lokale Charakter. Das heißt die ganze Funktionalität eines Algorithmus spielt sich auf der Ebene des Neurons ab. Idealerweise ist in den Schichten, bzw. im Netz keine zusätzliche globale Information nötig, um einen bestimmten Algorithmus zu implementieren (Bsp. CBackpropNeuron). Die Berechnung der Fehler, das Anpassen der Gewichte usw. erfolgt also nicht von "oben" sondern auf der Ebene des Neurons. Die höher liegenden Klassen (Schicht, Netz) steuern lediglich die Kommunikation der Neuronen untereinander.

Es gibt allerdings Fälle, in denen sich die Lokalität nicht vollständig wahren lässt. Zum Beispiel beim Ermitteln eines Gewinners einer Schicht von Neuronen (CKohonenLayer). Bei der Implementation neuer Algorithmen sollte trotzdem immer versucht

werden, den lokalen Charakter so weit wie möglich zu erhalten.

A.1.1 public members

CNeuron::CNeuron

```
CNeuron();
```

CNeuron::~CNeuron

```
virtual ~CNeuron();
```

Destruktor, entfernt alle Verbindungen zu dem Neuron

CNeuron::input_neuron

```
CNeuron* input_neuron(long number);
```

liefert das Input-Neuron mit der Nummer number. Als Input-Neuron wird ein Neuron bezeichnet, das eine Ausgangsverbindung (Synapse) zu dem aktuellen Neuron besitzt. Dieses muss in die Liste der verbundenen Neuronen aufgenommen sein. (wird mit connect automatisch gemacht)

CNeuron::output_neuron

```
CNeuron* output_neuron(long number);
```

liefert das Outputneuron mit der Nummer number

CNeuron::get_anzahl_inputs

```
virtual long get_anzahl_inputs();
```

liefert die Zahl der Input-Synapsen

CNeuron::get_anzahl_outputs

```
virtual long get_anzahl_outputs();
```

liefert die Zahl der Output-Synapsen

CNeuron::calc_output

```
virtual TNPreSynapticValue calc_output();
```

setzt den Output des Neurons, d.h. wendet die Aktivierungsfunktion auf die aktuelle Aktivierung an und speichert das Ergebnis in output. Die Aktivierung wird wieder auf 0 gesetzt, die alte Aktivierung als old_activation gespeichert

CNeuron::get_last_output()

```
virtual TNPreSynapticValue get_last_output();
```

liefert den (letzten) Output des Neurons

CNeuron::calc_error_output

```
virtual TNErrorValue calc_error_output();
```

berechnet den Fehler, den das Neuron ausweisen soll aus der Fehleraktivierung. Im Standard-Neuron wird Fehler=Fehleraktivierung gesetzt. Diese Funktion muss für die meisten abgeleiteten Klassen überschrieben werden.

CNeuron::get_last_error_output

```
get_last_error_output();
```

liefert den letzten festgestellten error_output (ohne ihn neu zu ermitteln)

CNeuron::connect_output

```
virtual void connect_output(CNeuron* other,TNSynapticWeight
weight);
```

verbindet das Neuron (Output) mit einem anderen Neuron (Input). Dabei wird eine Synapse mit dem entsprechenden Gewicht angelegt. weight gibt das Gewicht der Verbindung an.

CNeuron::feed_forward

```
virtual void feed_forward(TNTimeUnit time=-1);
```

gibt den output des Neurons über die Synapsen an alle nachgeschalteten Neuronen weiter. Dabei wird das Gewicht der Synapsen entsprechend jeweiligen Synapse beachtet.

CNeuron::feed_back

```
virtual void feed_back(TNTimeUnit time=-1);
```

führt einen Feedbackschritt aus. Der Fehler wird unter Beachtung der Synapsen an alle vorgeschalteten Neuronen weitergeleitet

CNeuron::add_excitation

```
virtual void add_excitation(TNPostSynapticValue
excitation,TNTimeUnit time=0);
```

das Neuron wird zum Zeitpunkt time mit dem Wert excitation erregt

CNeuron::add_error_activation

```
virtual void add_error_activation(TNErrorValue error);
```

fügt der Fehleraktivierung den entsprechenden Wert hinzu. Die Fehleraktivierung entspricht der Aufsummation des zurückgeschickten Fehlers der Nachfolgerneuronen.

CNeuron::set_error

```
virtual void set_error(TNErrorValue);
```

setzt den Fehlerwert für das Neuron

CNeuron::get_weight_vector

```
void get_weight_vector(CDoubleVector& vec);
```

liefert den Vektor der Inputgewichte des Neurons

CNeuron::get_activation

```
TNPostSynapticValue get_activation();
```

liefert die aktuelle Aktivierung des Neurons

CNeuron::create_input_synapse

```
virtual CSynapse* create_input_synapse(CNeuron*
other,TNSynapticWeight weight);
```

erzeugt eine Input-Synapse, die mit dem anderen Neuron verbunden ist. Kann in abgeleiteten Klassen überschrieben werden, wenn z.B. das Neuron eine andere als die Standardsynapse verwendet

CNeuron::process_cmd

```
virtual bool process_cmd(const char* command, const void*
data,TPtrVector* add_data=0,long id=0);
```

Ausführen eines Kommandos. Folgende stehen zur Verfügung:

"learning_rate","lr" ... bestimmt die Lernrate des Neurons (Startwert)

"learning_rate_goal","lr_goal" ... bestimmt den Wert, auf den die Lernrate konvergieren soll

"learning_rate_factor","lr_factor" ... Der Faktor mit dem sich die Lernrate dem Konvergenzziel nähert. (Üblicherweise > 0.99 und < 1)

"slope","sl" ... Der Anstieg der Aktivierungsfunktion des Neurons

"min_error","me" ... wenn der Fehler kleiner ist als min_error, werden auf 0 gesetzt. D.h. für diese Werte erfolgt kein Lernen mehr.(Vermeidung von Overtraining) "output_weights","ow" ... Setzt die Gewichte der ausgehenden Synapsen auf einen zufälligen Wert zw. data[0] (min) und data[1] (max).(Standard -1,1)

CNeuron::set_output

```
void set_output(double a_value);
```

setzt den Output des Neurons auf den durch a_value bestimmten Wert

CNeuron::save_to_xml

```
virtual xmlNodePtr save_to_xml(xmlNodePtr tree);
```

speichert in das XML-File, zu dem tree gehört, muss in den abgeleiteten Klassen überschrieben werden, wenn zusätzliche Werte abgespeichert werden sollen

CNeuron::load_from_xml

```
virtual void load_from_xml(xmlNodePtr tree);
```

lädt aus dem XML-File, zu dem tree gehört

CNeuron::learning_rate

```
CChangingProperty learning_rate;
```

die verwendete Lernrate (kann abgekühlt werden)

A.1.2 protected members

CNeuron::activation

```
TNActivationValue activation;
```

die aktuelle Aktivierung

CNeuron::old_activation

```
TNActivationValue old_activation;
```

der letzte Wert der Aktivierung

A.2 *CSynapse* class reference (csynapse.h)

Description:

Implementation einer Synapse. Die Synapse besitzt zwei Zeiger. Einen auf das Vorgänger- und einen auf das Nachfolger-Neuron. Um eine Synapse zwischen 2 Neuronen zu erzeugen muss diese zusätzlich bei beiden Neuronen eingetragen werden. Die Synapse besitzt ein Gewicht, welches die Stärke der Verbindung zwischen den beiden Neuronen symbolisiert. Dieses kann durch add_weight verändert werden. Die Funktion process_input ist die zentrale Funktion für die Synapse. Die auf die Synapse eintreffenden Signale werden dieser Funktion übergeben und diese erzeugt daraus das Signal, das an das postsynaptische Neuron weiter gegeben wird.

A.2.1 public members

CSynapse::~CSynapse

```
virtual ~CSynapse();
```

CSynapse::CSynapse

```
CSynapse(CNeuron* pre_neuron, CNeuron* post_neuron,
TNSynapticWeight a_weight);
```

erstellt eine Synapse zwischen 2 Neuronen, und trägt diese in den Neuronen entsprechend ein.

CSynapse::process_input

```
void CSynapse::process_input(TNPreSynapticValue a_input);
```

liefert eine Erregung an die entsprechende Synapse

CSynapse::set_weight

```
void set_weight(TNSynapticWeight new_weight);
```

ändert das Gewicht der Synapse auf weight

CSynapse::get_weight

```
TNSynapticWeight get_weight();
```

liefert das Gewicht der Synapse

CSynapse::add_weight

```
void add_weight(TNSynapticWeight to_add);
```

erhöht das Gewicht um to_add, falls to_add kleiner 0 wird das Gewicht entsprechend vermindert.

CSynapse::set_random_weight

```
void set_random_weight(TNSynapticWeight
min_weight=-1,TNSynapticWeight max_weight=1);
```

ändert das Gewicht der Synapse auf einen zufälligen Wert zwischen min und max

CSynapse::get_post_synaptic_value

```
virtual TNPostSynapticValue
get_post_synaptic_value(TNPreSynapticValue a_input);
```

liefert zu dem Signal (a_input) den entsprechenden postsynaptischen Wert. Diese Funktion ist virtuell und kann für verschiedene Synapsenarten unterschiedlich sein. Standardmäßig wird das Gewicht mit dem Input multipliziert.(Wird von process_input aufgerufen)

A.2.2 protected members

CSynapse::save_to_xml

```
xmlNodePtr save_to_xml(xmlNodePtr tree);
```

speichert in das XML-File, zu dem tree gehört

CSynapse::load_from_xml

```
void load_from_xml(xmlNodePtr tree);
```

lädt aus dem XML-File, zu dem tree gehört

CSynapse::weight

```
TNSynapticWeight weight;
```

Das Gewicht der Synapse

CSynapse::pre_synaptic_neuron

```
CNeuron* pre_synaptic_neuron;
```

CSynapse::post_synaptic_neuron

```
CNeuron* post_synaptic_neuron;
```

A.3 *CNeuronicLayer* class reference (cneuroniclayer.h)

Description:

Klasse zur Verwaltung mehrerer (=einer Schicht) Neuronen. Auf Grund des lokalen Charakters (siehe CNeuron) der Implementation genügt es im Normalfall die einzelnen Neuronen über diese Klasse zu steuern. Wenn der lokale Charakter verletzt wird, kann es nötig sein, spezialisierte Ableitungen dieser Klasse zu bilden (siehe CKohonenLayer).

Diese Klasse erlaubt es mehrere Neuronen gemeinsam anzusprechen (feed_forward, feed_back) und Werte zuzuweisen und auszulesen (set_output, get_output), sowie eine Skalierung/Normierung einzustellen (set_scaling). Außerdem ist es möglich mehrere Schichten miteinander zu verbinden (connect_output).

Beispiel siehe CNeuronicNet.

A.3.1 public members

CNeuronicLayer::CNeuronicLayer

```
CNeuronicLayer();
```

CNeuronicLayer::˜CNeuronicLayer

```
virtual ˜CNeuronicLayer();
```

CNeuronicLayer::connect_output

```
void connect_output(CNeuron* neuron, TNSynapticWeight
min_weight, TNSynapticWeight max_weight);
```

verbindet alle Neuronen der Schicht mit dem Neuron. Dabei wird die Synapse auf einen zufälligen Wert zwischen min_weight und max_weight gesetzt

CNeuronicLayer::get_anzahl_neurons

```
long get_anzahl_neurons();
```

liefert die Anzahl der Neuronen in der Schicht

CNeuronicLayer::connect_output

```
void connect_output(CNeuronicLayer* other, TNSynapticWeight
min_weight=-0.2, TNSynapticWeight max_weight=0.2,
EConnectionType connection_type=CT_ALL_BEETWEEN_LAYERS);
```

verbindet Neuronen dieser Schicht je nach connection_type mit Neuronen aus der Schicht other. Dabei wird die Synapse auf einen zufälligen Wert zwischen min_weight und max_weight gesetzt.

CNeuronicLayer::neuron

```
CNeuron* neuron(long number);
```

liefert das Neuron mit der Nummer number aus der Schicht. Die Zählung beginnt mit 0.

CNeuronicLayer::add_neurons

```
template <class T> void add_neurons(T* mother,long anzahl);
```

fügt anzahl Neuronen der Art T zur Schicht hinzu.

CNeuronicLayer::add_neuron

```
long add_neuron(CNeuron* a_neuron);
```

fügt das Neuron a_neuron zu der Menge der verwalteten Neuronen hinzu und liefert dessen Nummer zurück.

CNeuronicLayer::remove_neuron

```
void remove_neuron(long nrn_nr,bool delete_neuron=false);
```

entfernt das Neuron mit der Nummer nrn_nr.

CNeuronicLayer::feed_back

```
virtual void feed_back(TNTimeUnit time=-1);
```

Führt einen Feed-Back-Schritt durch. Für jedes Neuron der Schicht wird die Funktion feed_back aufgerufen.

CNeuronicLayer::feed_forward

```
virtual void feed_forward(TNTimeUnit time=-1);
```

Führt einen Feed-Forward-Schritt durch. Für jedes Neuron der Schicht wird die Funktion feed_forward aufgerufen.

CNeuronicLayer::get_output

```
void get_output(CDoubleVector& a_outputs, bool
use_scaling=false);
```

liefert die Outputs der Neuronen der Schicht im Vekor a_outputs zurueck. Dabei entspricht a_outputs[i] dem Output von Neuron i. Je nach eingestellter Skalierung können die Werte automatisch angepasst werden.

CNeuronicLayer::set_error

```
void set_error(const CDoubleVector& a_errors, bool
use_scaling=false);
```

setzt den Fehler für alle Neuronen der Schicht. Dabei entspricht a_errors[i] dem Fehler von Neuron i. Die Dimension des Vektors muss mit der Anzahl der Neuronen übereinstimmen. Je nach eingestellter Skalierung können die Werte automatisch angepasst werden.

CNeuronicLayer::set_scaling

```
void set_scaling(CDoubleVector& a_unscaled_min, CDoubleVector&
a_unscaled_max, double a_scaled_min, double a_scaled_max);
```

setzt die Skalierung so, dass der Input so eingerichtet wird, dass er zwischen scaled_min und scaled_max liegt. Wenn der Output skaliert wird, muss der ursprüngliche Wert zwischen scaled_min und scaled_max liegen, damit er erfolgreich zurückskaliert werden kann.

Beispiel: Für ein Netz mit 2 Inputs (Temperatur 0..100, Volumen 200...2000) soll die Skalierung gesetzt werden: Die skalierten Werte sollen zwischen -1 und 1 liegen:

```
CDoubleVector smin(2),smax(2);
smin[0]=0; smax[0]=100;
smin[1]=200; smax[1]=2000;
input_layer->set_scaling(smin,smax,-1,1);
```

Ein Aufruf von

```
CDoubleVector bsp(2);
bsp[0]=0; bsp[1]=2000;
input_layer->set_output(bsp,true)
```

stellt die output-Werte in den Neuronen wie gewollt auf -1 bzw. 1. Analog liefert get_output(x,true) für die in den Neuronen eingestellten Werte wieder die ursprünglichen Werte.

CNeuronicLayer::process_cmd

```
virtual bool process_cmd(const char* command, const void*
data,TPtrVector* add_data=0,long id=0);
```

Ausführen eines Kommandos. Folgende stehen zur Verfügung:

"neuron(i)" ... Gibt den Befehl an das Neuron i weiter ("neuron(*)" - an alle Neuronen der Schicht)

CNeuronicLayer::set_output

```
void set_output(const CDoubleVector & values, bool
use_scaling);
```

Setzt die Outputs der Neuronen in der Schicht auf die vorgegebenenWerte. Die Outputs der Neuronen werden in einem folgenden Feed-Forward-Schritt nicht (neu) berechnet. Dabei entspricht values[i] dem Output von Neuron i.

CNeuronicLayer::get_error

```
void get_error(CDoubleVector & a_errors, bool use_scaling);
```

liefert die letzten Fehler der Neuronen der Schicht in a_errors zurück. Dabei entspricht a_errors[i] dem Fehler von Neuron i.

A.3.2 protected members

CNeuronicLayer::load_from_xml

```
virtual void load_from_xml(xmlNodePtr tree);
```

lädt aus dem XML-File, zu dem tree gehört

CNeuronicLayer::save_to_xml

`xmlNodePtr save_to_xml(xmlNodePtr tree);`

speichert ins XML-File, an der Stelle tree

A.4 *CNeuronicNet* class reference (cneuronicnet.h)

Description:

implementiert ein (künstliches) neuronales Netz und ist gleichzeitig Basisklasse für spezialisierte Netzklassen. Das Netz besteht aus mehreren Schichten (CNeuronicLayer), welche es verwaltet. Die Funktionalität wird normalerweise in den Neuronen/Schichten festgelegt, so dass im Standardfall diese Basisklasse für die Implementation eines Netzes verwendet werden kann.

Beispiel für die Anwendung:

```
//-------------------------------------
//Netz erzeugen
CNeuronicNet net;
//die Schichten anlegen und zum Netz hinzufügen
net.add_layer(CInputNeuron::TYP,2);
//1. Schicht besteht aus 2 Inputneuronen
net.add_layer(CBackpropNeuron::TYP,2);
// 2 Backpropneuronen in der verborgenen Schicht
net.add_layer(CBackpropNeuron::TYP,1);
// 1 Backpropneuron in der Output-Schicht
//Neuronen des Netzes verbinden, zufäl-
liges Gewicht zwischen -1 und 1
net.connect(CT_ALL_BEETWEEN_LAYERS,-1,1);
//-------------------------------------
//Training vorbereiten (Pattern laden)
std::vector<CDoubleVector> inputs, outputs
CNeuronica::load_patterns("xor.pat",inputs,outputs);
//-------------------------------------
//Das eigentliche Training
CDoubleVector ergebnis;
for (long j=0; j < 5000; j++) {
  for (ulong i=0; i < inputs.size(); i++ ) {
    //dem Netz die Inputs repäsentieren
```

```
    net-
>set_input(inputs[i],false); //false=keine Skalierung ver-
wenden
    //die Werte vorwärts propagieren
    net->feed_forward();
    //das Netz-Ergebnis auslesen
    net->get_output(ergebnis,false);
    //den Fehler ermitteln und setzen
    net->set_error(outputs[i]-ergebnis,false);
    //den Fehler zurück propagieren net->feed_back();
  }
} //-------------------------------------
//Anwendung des Gelernten
CDoubleVector in,out;
in.push_back(0);
in.push_back(1);
net->calc_output(in,out,true);
cout << "\nDas Ergeb-
nis für (" << in << ") lautet " << out;
```

A.4.1 public members

CNeuronicNet::CNeuronicNet

```
CNeuronicNet();
```

erzeugt ein leeres Netz

CNeuronicNet::˜CNeuronicNet

```
virtual ˜CNeuronicNet();
```

CNeuronicNet::layer

```
CNeuronicLayer* layer(long nr);
```

liefert einen Zeiger auf die Schicht nr zurück

CNeuronicNet::set_error

```
void set_error(const CDoubleVector& a_errors, bool
use_scaling=false);
```

setzt den Fehler f. Die letzte Schicht des Netzes. Wird verwendet um z.B. einen Fehler zurückzupropagieren.

CNeuronicNet::get_anzahl_layers

```
long get_anzahl_layers();
```

liefert die Zahl der Schichten des Netzes

CNeuronicNet::get_output

```
void get_output(CDoubleVector& a_outputs, bool
use_scaling=false,ELayerCode nr=OL0);
```

liefert den Output der letzten Schicht des Netzes (das Netzergebnis). a_outputs ist der Vektor mit den zurückgelieferten Outputs. Falls use_scaling true ist wird die eingestellte Skalierung verwendet. nr bezeichnet die Schicht für die der Befehl ausgeführt wird. (OL0 bezeichnet die letzte Schicht, OL1 die vorletzte usw.) - Standardwert: OL0

CNeuronicNet::set_input

```
void set_input(const CDoubleVector& input, bool
use_scaling=false, long nr_input_layer=0);
```

setzt den Input für die angegebene Schicht des Netzes. Bei dieser sollte es sich um eine Input-Schicht handeln. Die Outputs der Input-Schicht werden auf die Werte in dem Vektor gesetzt.

CNeuronicNet::feed_back

```
virtual void feed_back(TNTimeUnit time=0);
```

führt einen Feed-Back-Schritt über das gesamte Netz aus. Dazu wird feed_back für jede Schicht (gemäß ihrer Reihenfolge - beginnend mit der ersten) aufgerufen.

CNeuronicNet::feed_forward

```
virtual void feed_forward(TNTimeUnit time=0);
```

führt einen Feed-Forward-Schritt über das gesamte Netz aus. Dazu wird feed_forward für jede Schicht (gemäß ihrer Reihenfolge - beginnend mit der letzten) aufgerufen.

CNeuronicNet::add_layer

```
virtual long add_layer(CNeuronicLayer* a_layer);
```

fügt dem Netz eine Schicht Neuronen hinzu

Returns: die Nummer der Schicht im Netz

CNeuronicNet::add_layer

```
template <class T> long add_layer(T* mother,long anzahl);
```

erstellt eine Schicht aus anzahl Neuronen der Art T und fügt diese zum Netz hinzu. Das übergebene Neuron bestimmt nur den Typ des Neurons und stellt keine Kopiervorlage dar.

Returns: die Nummer der Schicht im Netz

CNeuronicNet::connect

```
virtual void connect(EConnectionType
connection_type=CT_DEFAULT, TNSynapticWeight min_weight=-0.2,
TNSynapticWeight max_weight=0.2);
```

verbindet die Schichten eines Netzes je nach Vorgabe von connection_type. Die Synapse wird auf einen zufälligen Wert zwischen min_weight und max_weight gesetzt.

CNeuronicNet::process_cmd

```
virtual bool process_cmd(const char* command, const void*
data,TPtrVector* add_data=0,long id=0);
```

Ausführen eines Kommandos. Folgende stehen zur Verfügung:

"layer(i)" ... Gibt den Befehl an die Schicht i weiter ("layer(*)" - an alle Schichten)
"print_weights" ... Gibt die Gewichte des Netzes aus (nach cout)
"save" ... speichert das Netz in eine Datei
"save_snns" ... speichert das Netz in eine SNNS-NET-Datei
"save_nnd" ... speichert das Netz in eine Datei im NND-Format
"load" ... lädt das Netz aus der angegebenen Datei

CNeuronicNet::calc_output

```
void calc_output(CDoubleVector& in, CDoubleVector& out, bool
use_scaling=false);
```

liefert zu dem Netzinput in in den entsprechenden Output nach out. D.h. der Vektor in wird in die Input-Schicht des Netzes eingetragen. feed_forward wird für das Netz aufgerufen. Der Output der letzen Schicht wird ausgelesen und in dem Vektor out zurückgegeben.

CNeuronicNet::save

```
virtual bool save(const char * filename, const char*
format="DEFAULT");
```

speichert in eine Datei im NND-Format bzw. im SNNS-NET-Format. Beim Speichern im SNNS-Format gibt die x-Position des Neurons die Schicht im Netz wider.

Returns: true, falls erfolgreich

CNeuronicNet::load

```
virtual bool load(const char* filename , const char*
format="DEFAULT");
```

lädt das Netz aus einer Datei im XML-basierten NND-Format (Neuronica Network Definition) oder aus dem SNNS-NET-Format.

CNeuronicNet::learn_all

```
void learn_all(CPatterns& patterns, bool use_scaling=false);
```

führt einen Lernschritt mit jedem der Patterns in CPatterns durch

CNeuronicNet::get_error

```
void get_error(CDoubleVector & a_errors, bool
use_scaling=false);
```

liefert den Error der letzten Schicht des Netzes

CNeuronicNet::input_layer

```
virtual CNeuronicLayer * input_layer(long nr=0);
```

liefert einen Zeiger auf die Input-Schicht nr. Die Schicht muss existieren. Wenn mehr als eine Schicht eine Input-Schicht ist, müssen diese als Input-Schicht deklariert sein (mit set_noof_input_layers)

CNeuronicNet::output_layer

```
virtual CNeuronicLayer * output_layer(ELayerCode nr=OL0);
```

liefert einen Zeiger auf die Output-Schicht nr. Verwendet ELayerCode um die Outputschichten zu kennzeichnen. Parameter OL0 liefert die letzte-Layer des Netzes, Parameter OL1 liefert die vorletzte usw. nr=IL0 liefert erste Input-Schicht des Netzes, IL1 die zweite usw.

CNeuronicNet::remove_layer

```
void remove_layer(long lay_nr, bool delete_layer=false);
```

entfernt die Schicht mit der Nummer aus der Verwaltung des Netzes. Die Schicht wird gelöscht, wenn delete_layer true ist

CNeuronicNet::set_learning_rate

```
virtual void set_learning_rate(double lr);
```

stellt die Lernrate aller Neuronen des Netzes auf den übergebenen Wert ein

CNeuronicNet::train

```
void train(CPatterns & patterns, long noof_steps, bool
use_scaling=false);
```

Trainiert das Netz mit allen Pattern. Jeder Pattern wird noof_steps mal trainiert.

A.4.2 protected members

CNeuronicNet::load_from_xml

```
virtual void load_from_xml(xmlNodePtr tree);
```

lädt aus dem XML-File, zu dem tree gehört. Sollte normalerweise nicht direkt verwendet werden,sondern besser load().

CNeuronicNet::save_to_xml

```
virtual xmlNodePtr save_to_xml(xmlNodePtr tree);
```

speichert ins XML-File, an der Stelle tree. Serollte normalerweise nicht direkt verwendet werden,sondern besser save()

A.5 *CNeuronicMultiNet* class reference (cneuronicmultinet.h)

Inherits: CNeuronicNet

Description:

implementiert ein (künstliches) neuronales Netz wobei auch Teilnetze möglich sind. Das Netz kann entweder aus mehreren Schichten (layer) (siehe CNeuronicNet) oder wiederum aus mehreren Netzen bestehen (subnet). Beides zusammen ist (z.Zt.) nicht möglich. Die Sub-Netze können selbst wieder Multi-Netze enthalten.

Beispiel:

```
CNeuronicNet* net;
CNeuronicMultiNet* mnet=new CNeuronicMultiNet();
// 3 Xor-Netze laden
```

```
mnet-
>add_subnet(CNeuronica::load_new_net("xor.nnd")); //Netz0 mnet-
>add_subnet(CNeuronica::load_new_net("xor.nnd")); //Netz1 mnet-
>add_subnet(CNeuronica::load_new_net("xor.nnd")); //Netz2
// Zusammenschal-
tung der Netze (zu XOR(XOR(X1,X2),XOR(X3,X4))
mnet->connect(0,2,OL0,IL0,0); // verknüpfe Output-
layer von Netz0 mit Inputlayer von Netz2 (ab Neu-
ron 0) mnet->connect(1,2,OL0,IL0,1); // verknüpfe Out-
putlayer von Netz1 mit Inputlayer von Netz2 (ab Neu-
ron 1) //Testen:
CDoubleVector erg,tst;
tst.resize(2);
tst[0]=0;
tst[1]=0;
mnet->set_input(tst,true,0);
//Netz0 füttern
mnet->set_input(tst,true,1);
//Netz1 füttern
mnet->feed_forward();
//Ergebnis berechnen ...
mnet->get_output(erg,true);
//...und auslesen cout << " ERGEB-
NIS: " << erg[0] << endl;
```

A.5.1 public members

CNeuronicMultiNet::CNeuronicMultiNet

```
CNeuronicMultiNet();
```

erzeugt ein leeres Netz

CNeuronicMultiNet::˜CNeuronicMultiNet

```
virtual ˜CNeuronicMultiNet();
```

CNeuronicMultiNet::subnet

```
CNeuronicNet* subnet(long nr);
```

liefert einen Zeiger auf das Subnet nr zurück

CNeuronicMultiNet::input_layer

```
virtual CNeuronicLayer * input_layer(long nr=0);
```

liefert einen Zeiger auf die Input-Schicht nr. Meist gibt es nur eine Input-Schicht und der Aufruf kann ohne Parameter erfolgen.

CNeuronicMultiNet::output_layer

```
virtual CNeuronicLayer * output_layer(ELayerCode nr=OL0);
```

liefert einen Zeiger auf die Output-Schicht nr. Verwendet ELayerCode um auf die Outputschichten zuzugreifen. Parameter OL0 liefert die Output-Layer des letzten Subnetzes. Parameter OL1 liefert die Output-Layer des vorletzten Subnetzes usw. nr=0 liefert die output_layer des subnetztes 0, nr=1 die output_layer des subnetzes 1 usw.

CNeuronicMultiNet::get_anzahl_subnets

```
long get_anzahl_subnets();
```

liefert die Zahl der Subnetze des Netzes

CNeuronicMultiNet::feed_back

```
void feed_back(TNTimeUnit time=0);
```

führt einen Feed-Back-Schritt über das gesamte Netz aus. Dazu wird feed_back für jedes sub-Netz (gemäß ihrer Reihenfolge) aufgerufen.

CNeuronicMultiNet::feed_forward

```
void feed_forward(TNTimeUnit time=0);
```

führt einen Feed-Forward-Schritt über das gesamte Netz aus. Dazu wird feed_forward für jedes sub-Netz (gemäß ihrer Reihenfolge) aufgerufen.

CNeuronicMultiNet::add_subnet

```
long add_subnet(CNeuronicNet* a_net);
```

fügt dem Netz eine Teilnetz hinzu.

Returns: liefert als Ergebnis die Nummer des Teilnetzes im Netz.

CNeuronicMultiNet::process_cmd

```
virtual bool process_cmd(const char* command, const void*
data,TPtrVector* add_data=0,long id=0);
```

Ausführen eines Kommandos. Folgende stehen zur Verfügung:

"subnet(i)" ... Gibt den Befehl an das Sub-Netz i weiter ("subnet(*)" - an alle Sub-Netze)

CNeuronicMultiNet::connect

```
void connect(long net_out,long net_in, ELayerCode
layer_net_out=OL0,ELayerCode layer_net_in=IL0,long
startneuron=-1,bool store_connection_info=true);
```

Verbindet die Schicht layer_net_out des Subnetzes net_out mit der Schicht layer_net_in des Subnetzes net_in. Dabei steht OL für die Outputlayer und IL0...ILn für die Nr. der Inputlayer. Die Verbindung wird hergestellt und die Verbindungsinformationen werden gespeichert. Im Standard-Fall wird immer die letzte Output-Schicht von Net1 mit der ersten Inputsicht von net2 verbunden. Wenn Startneuron nicht angegeben ist (-1), müssen beide Schichten die selbe Zahl von Neuronen haben. Ist ein Startneuron angegeben, so wird das erste Neuron der layer_net_out mit dem neuron(start_neuron) der layer_net_in verbunden, das zweite mit neuron(startneuron+1) usw. In diesem Fall müssen genug Neuronen in layer_net_in vorhanden sein, um alle Neuronen aus layer_net_out zu verbinden.

CNeuronicMultiNet::load_subnet

```
bool load_subnet(long nr, const char* filename, const char*
format="DEFAULT");
```

Lädt das Netz mit der Nummer nr aus der Datei filename. Ein eventuell schon vorhandenes Netz wird gelöscht, die Verbindungen aber wie bisher wieder hergestellt. D.h. das Netz wird ersetzt.

CNeuronicMultiNet::replace_subnet

```
void replace_subnet(long nr,CNeuronicNet* net,bool
preserve_scaling=false);
```

löscht das vorhandene Subnet mit der Nummer nr und ersetzt es durch das Netz net. Die Verbindungen zu anderen Netzen (die mit connect() erzeugt wurden) bleiben dabei erhalten. (D.h. sie werden zunächst gelöscht und für das übergebene Netz neu angelegt.)

CNeuronicMultiNet::load

```
bool load(const char* filename , const char* format="DEFAULT"
);
```

lädt das Netz aus einer Datei

A.5.2 protected members

CNeuronicMultiNet::save_to_xml

```
xmlNodePtr save_to_xml(xmlNodePtr tree);
```

speichert ins XML-File, an der Stelle tree

CNeuronicMultiNet::load_from_xml

```
void load_from_xml(xmlNodePtr tree);
```

lädt aus dem XML-File, zu dem tree gehört

Anhang B

Beispielprogramme

B.1 Programm zum Vergleich von Neuronica mit dem SNNS

In diesem Abschnitt befindet sich das Listing zu dem zum Vergleich zwischen Neuronica und SNNS verwendeten Programm sowie die zugehörige Pattern-Datei (siehe auch Abschnitt 2.7.2):

Das Programm `vergleich_neuronica_snns.cpp`:

```
#include <iostream.h>
#include "neuronica/neuronica.h"

//Test ob SNNS und Neuronica identisch
int main(int argc, char *argv[])
{
    //Das Neuronica-BP-Netz erstellen
    //Schicht 1 - 2 Inputneuronen   - Inputschicht
    //Schicht 2 - 2 Backpropneuronen
    //Schicht 3 - 1 Backpropneuron  - Outputschicht
    CNeuronicNet bp_net;
    bp_net.add_layer(CInputNeuron::TYP,2);
    bp_net.add_layer(CBackpropNeuron::TYP,2);
    bp_net.add_layer(CBackpropNeuron::TYP,1);
    bp_net.connect(CT_ALL_BEETWEEN_LAYERS,-1,1);

    //Das BP-Netz als SNNS-NET-Datei speichern und über die
    //Anbindung an den SNNS laden
```

```
    bp_net.save("xor_test1.net","SNNSNET");
    CNeuronicSnnsNet s_net;
    s_net.load("xor_test1.net");

    //Die Muster für ein XOR-Problem laden
    CPatterns pat;
    pat.load_patterns("xor.pat");

    //Einstellungen
    const double LEARNING_RATE =0.5;
    cout<<"Einstellen der gleichen Lernrate ("
        <<LEARNING_RATE<<") für beide Netze:"<<endl;
    bp_net.set_learning_rate(LEARNING_RATE);
    s_net.set_learning_rate(LEARNING_RATE);

    //Die Netze nacheinander trainieren
    cout<<"BP - Training   "<<flush;
    for (long i=1;i<1000;i++){
        bp_net.learn_all(pat,false);
        if ((i%100)==0) cout<<"."<<flush;
    }
    cout<<"done\n";
    cout<<"SNNS - Training "<<flush;
    for (long i=1;i<1000;i++){
        s_net.learn_all(pat,false);
        if ((i%100)==0) cout<<"."<<flush;
    }
    cout<<"done"<<endl;

    //Das Lernergebnis ausgeben
CDoubleVector ergebnis;
    cout<<"\nLernen liefert:"<<endl;
    for (ulong i=0; i<pat.inputs.size();i++){
        bp_net.set_input(pat.inputs[i],false);
        bp_net.feed_forward();
        bp_net.get_output(ergebnis,false);
        cout<< i+1<<". Fall BP   ("<<pat.inputs[i]<<"): "
            <<ergebnis<<endl;
        s_net.set_input(pat.inputs[i],false);
        s_net.feed_forward();
        s_net.get_output(ergebnis,false);
        cout<< i+1<<". Fall SNNS ("<<pat.inputs[i]<<"): "
            <<ergebnis<<endl;
```

```
    }
    return 0;
}
```

Die verwendete Pattern-Datei `xor.pat`:

```
SNNS pattern definition file V3.2
generated at Fri Jun 17 14:55:46 1994

No. of patterns : 4
No. of input units : 2
No. of output units : 1

# Input pattern 1:
0 0
# Output pattern 1:
0
# Input pattern 2:
0 1
# Output pattern 2:
1
# Input pattern 3:
1 0
# Output pattern 3:
1
# Input pattern 4:
1 1
# Output pattern 4:
0
```

Abbildungsverzeichnis

Literaturverzeichnis

[AZ+] M.Vogt A. Zell, G. Mamier et al. *SNNS - Stuttgart Neural Network Simulator*. University of Stuttgart - Institute for Parallel and High Performance Computing, University of Tübingen - Wilhelm-Schickard-Institute for Computer Science. User Manual, Version 4.2.

[Blu92] Adam Blum. *Neural networks in C++: an object-oriented framework for building connectionist systems*. Wiley professional computing. John Wiley and Sons, Inc., New York, NY, USA; London, UK; Sydney, Australia, 1992.

[Boo94] G. Booch. *Objektorientierte Analyse und Design*. Addison-Wesley, Bonn, 1994.

[Can39] W. B. Cannon. *The wisdom of the body*. Norton, New York, 1939.

[Dera] R. Der. Vorlesung Maschinelles Lernen. Universität Leipzig Institut für Informatik. Kap.7 Reinforcement-Lernen. http://www.informatik.uni-leipzig.de/ der/Vorlesungen/.

[Derb] R. Der. Vorlesung Neuroinformatik. UniversitätLeipzig, Institut für Informatik. http://www.informatik.uni-leipzig.de/ der/Vorlesungen/skrpt-ni.html.

[Der01] Ralf Der. Self-organized aquisition of situated behaviors. Technical report, Universität Leipzig, Institut für Informatik, April 2001.

[DP99] R. Der and T. Pantzer. Emergent robot behavior from the principle of homeokinesis. Technical report, Universität Leipzig, Institut für Informatik, 1999.

[DSP99] R. Der, U. Steinmetz, and F. Pasemann. Homeokinesis - a new principle to back up evolution with learning. In M. Mohammadian, editor, *Computational Intelligence for Modelling, Control, and Automation*, volume 55 of *Concurrent Systems Engineering Series*, pages 43 – 47. IOS Press, 1999. URL=http://www.informatik.uni-leipzig.de/ der/Veroeff/wienfin3.ps.

[Gru95] Rick Grush. *Emulation and Cognition*. PhD thesis, University of California, San Diego, 1995.

[GWZ99] Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In *Australian Joint Conference on Artificial Intelligence*, pages 417–428, 1999.

[HKP91] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation*, chapter 7, pages 188–196. Addison Wesley, New York, 1991.

[JR92] M. Jordan and D. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354, 1992.

[Khe] *Khepera User Manual*. Version 4.06.

[Knu68] Donald E. Knuth. *The Art of Computer Programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley, 1968.

[Koh84] T. Kohonen. *Self–Organization and associative memory*, volume 8 of *Springer Series in Information Science*. Springer, 1984.

[Koh95] T. Kohonen. *The self-organizing map*. Springer, 1995.

[KR95] S. Keerthi and B. Ravindran. A tutorial survey of reinforcement learning. Sadhana (published by the Indian Academy of Sciences), January 1995.

[Lab01] Pacific Northwest National Laboratory. Artificial neural networks - available software and hardware, Jan 2001.

[Mic] Olivier Michel. Khepera simulator homepage. http://diwww.epfl.ch/lami/team/michel/khep-sim/.

[Mun87] P. Munro. A dual backpropagation scheme for scalar-reward learning. In *Ninth Annual Conference of the Cognitive Science Society*, pages 165–176. Cognitive Science Society, Lawrence Erlbaum, 1987.

[Pan00] Thomas Pantzer. Entwurf und Implementation einer echtzeitfähigen Entwicklungsumgebung fuer Lern- und Evolutionsexperimente mit autonomen Robotern. Master's thesis, Universität Leipzig, September 2000.

[Pat97] Dan Patterson. *Künstliche Neuronale Netze*. Prentice Hall Verlag GmbH, 1997.

[Sut88] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[Zel94] A. Zell. *Simulation Neuronaler Netze*. Addison–Wesley, 3rd edition, 1994.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, den 3. Januar 2003 Jon Hennig