Diplomarbeit zum Thema

Rekursiver Autoassoziativer Speicher und Holographisch Reduzierte Repräsentation

von

Torsten Seiler

geboren am 14. Januar 1969 in Berlin

Eingereicht am 31. Juli 1996

an der

Technischen Universität Dresden
Fakultät Informatik
FG Wissensverarbeitung

Betreuender Hochschullehrer: Prof. S. Hölldobler

Inhaltsverzeichnis

Vorwort

Die Anregung, Konnektionismus und symbolische Verarbeitung zu verbinden, verdanke ich Prof. Hölldobler. Er und die Mitarbeiter des Fachbereichs Wissensverarbeitung haben mich stets mit Rat und Tat unterstützt. Für viele fruchtbare Diskussionen möchte ich auch Yvonne Kalinke und Eckhard Koch danken.

Ohne meine Eltern hätte ich mein Studium wohl nicht in dieser Form vollbringen können, daher auch an sie ein großes Dankeschön, für die Jahre liebevoller und großzügiger Unterstützung. Aber auch meiner Susi danke ich aus vollstem Herzen, denn durch sie wurden selbst die verregneten Tage zu Sonnentagen.

1. Einleitung

Konnektionismus und symbolische Verarbeitung, sind zwei lange Zeit nebeneinander existierende Verarbeitungsmodelle. Jedes Modell hat seine ganz speziellen Vorteile, aber auch Nachteile. Der Versuch, beide Modelle zu verbinden, ist bisher nur in einigen Ausnahmefällen gelungen. So stellten Shastri und Ajjanagadde [SA93] das System Shruti vor, mit dem sie in der Lage sind, Schlußfolgerungen aus einer Formelmenge mit Hilfe eines konnektionistischen Systems zu ziehen. Bei diesem Ansatz ist es gelungen, die Vorteile der Parallelität im konnektionistischen System zu nutzen. Leider kann Shruti nur auf eine eingeschränkte Klasse von Formeln angewendet werden.

Für die Berechnung von Modellen für aussagenlogische Programme wurde von Kalinke in [Kal94] ein rekurrentes Netz angegeben, das entsprechend der gegebenen Formelmenge konstruiert wird.

Beiden Systemen ist eine lokale Repräsentation von Wissen eigen. Wird eine Verarbeitungseinheit oder Verbindung gestört, so ist das Ergebnis im ganzen davon beeinträchtigt. Verteilte Repräsentationen in konnektionistischen Systemen bieten den Vorteil, gegenüber Störungen unempfindlicher zu sein. Die Informationen sind über viele Elemente des Systems verteilt, so daß der Ausfall eines einzelnen Elementes nicht den Ausfall des Gesamtsystems nach sich zieht.

Wäre dies der einzige Anspruch an konnektionistische Repräsentationen, so könnte diesem, aus der Sicht der Kodierungstheorie, durch eine geeignete Menge von Redundanz und herkömmlichen Fehlererkennungs- und Fehlerkorrekturalgorithmen Genüge getan werden.

Hinton bemerkt in [Hin90], daß die Speicherung von rekursiven Strukturen, wie sie in der symbolischen Verarbeitung üblich ist, ein großes Problem für konnektionistische Systeme darstellt. Zur Lösung dieses Problems schlägt er frame-artige Strukturen vor und nennt sie Teil–Ganzes–Hierarchien. Darunter ist zu verstehen, daß das Ganze immer auch seine Teile enthält und jedes Teil wieder ein neues Ganzes ist. Für konnektionistische Systeme bedeutet dies, daß die zu repräsentierenden Strukturen in Muster der gleichen Dimensionalität gespeichert werden und auch Teilmuster darin enthalten sind.

Der Wunsch nach verteilten, reduzierten Repräsentationen leitet sich auch aus der Hoffnung her, direkt mit diesen Repräsentationen zu rechnen, ohne ihre innere (symbolische) Struktur entfalten zu müssen. Dieser Weg ist es, der es erlaubt, ein konnektionistisches System zu bauen, das parallel ehemals symbolische Aufgaben auf subsymbolischen Niveau löst und dabei die zuvor beschriebenen Eigenschaften konnektionistischer Systeme aufweist. Chalmers [Cha90] hat gezeigt, daß es möglich ist, mit reduzierten Repräsentationen syntaktische Transformationen auf subsymbolischem Niveau durchzuführen. Stolcke und Wu [SW92] nutzten reduzierte Repräsentationen für das Matchen von Termen.

In dieser Arbeit werden zwei Vertreter dieser Art von konnektionistischen Systemen vorgestellt und untersucht. Zum einen handelt es sich dabei um die Holographisch reduzierte Repräsentation (HRR[1]), einem Modell, das auf einer assoziativen Operation beruht und in der Dissertation von Plate [Pla94] beschrieben wurde. Zum anderen wird ein Neuronales Netz untersucht, das von Sperduti [Spe92] vorgeschlagen wurde und Rekursiver autoassoziativer Speicher mit Marken, kurz LRAAM[2], genannt wird. Die LRAAM ist eine Weiterentwicklung der RAAM, die ursprünglich von Pollack [Pol90] veröffentlicht wurde.

Die HRR basiert auf einer assoziativen Operation, der Zirkularkonvolution. Assoziativspeicher mit dieser Operation wurden zuvor schon in einer Arbeit von Bottini [Bot80] betrachtet (aus Brause [Bra91]). Plate untersucht unter anderem die Möglichkeiten, mit der HRR frame–artige Strukturen zu speichern. Die Zirkularkonvolution besitzt die Eigenschaft, daß die Dimensionalität des Ergebnisvektors gleich der Dimensionalität der Eingabevektoren ist. Somit lassen sich frame–artige Strukturen in Vektoren fester Länge speichern. Bei der Betrachtung der Kapazität eines solchen Speichers geht er allerdings nur von Mengen von Paaren aus, was nicht auf tiefe Strukturen verallgemeinerbar ist.

Bei der LRAAM handelt es sich um ein mehrschichtiges Neuronales Netz. Das Netz wird darauf trainiert, die Identitätsfunktion zu berechnen. Da die verdeckte Schicht kleiner als die Eingabe- und Ausgabeschicht ist, entsteht bei erfolgreichem Training eine reduzierte Repräsentation der Eingabe. Diese kann wieder rekursiv als Teil der Eingabe verwendet werden. Auf diese Weise lassen sich Baumstrukturen in Vektoren fester Länge speichern, was ebenfalls der Forderung von Hinton entspricht.

Natürlich existieren noch viele weitere Modelle zur Repräsentation von symbolischen Strukturen, so zum Beispiel das Modell von Smolensky [Smo90], welches auf dem Tensor-Produkt basiert oder verschiedene rekurrente Netze, wie beispielsweise das Elman-Netz [Elm90, Elm91].

Die Auswahl fiel auf die LRAAM und die HRR, weil beide Modelle jeweils Vertreter einer speziellen Anschauung sind. Die HRR liefert einen vorgegebenen Mechanismus, und ist in gewissen Grenzen analytisch zu untersuchen (wie es Plate auch tat). Die HRR berücksichtigt aber durch diese Allgemeinheit die Domäne so gut wie überhaupt nicht. Im Gegensatz dazu liefert die LRAAM eine auf die Anwendung zugeschnittene Repräsentation, von der angenommen wird, daß sie für die gewählte Domäne adequat ist.

Ziel dieser Arbeit war es, eine Klassenbibliothek zur Simulation der HRR und der LRAAM zu schaffen. Mit Hilfe der daraus entstandenen Programme wurden Tests zur Speicherkapazität, Generalisierungsfähigkeit und Zerbrechlichkeit (brittleness) der beiden Modelle durchgeführt. Die gewonnenen Ergebnisse bildeten die Basis für eine vergleichende Bewertung beider Modelle unter Berücksichtigung des Einsatzes als Termrepräsentation für prädikatenlogische Programme.

[1] Holographic Reduced Representation
[2] Labeling Recursive Autoassociative Memory

Die Arbeit umfaßt drei Hauptteile. In den Kapiteln 2 und 3 werden die theoretischen Grundlagen für die im Kapitel 4 beschriebene Implementation behandelt. Aufbauend darauf erfolgt im Kapitel 5 die Präsentation und Bewertung der Simulationsergebnisse. Im einzelnen gliedert sich die Arbeit wie nachfolgend beschrieben.

Das Kapitel 2 führt in die Begriffe Term und Termrepräsentation ein.

Auf die Modelle konnektionistischer System, insbesondere auf die Anforderungen als Speicher geht das Kapitel 3 ein. Im Abschnitt 3.3 werden die Neuronalen Netze formal eingeführt und der Backpropagation–Algorithmus und seine Erweiterung beschrieben. Die in dieser Arbeit betrachteten Modelle, die LRAAM und die HRR, werden in den Abschnitten 3.4 und 3.5 betrachtet. Abschließend wird die Möglichkeit der Äquivalenz von HRR und LRAAM untersucht.

Eine der Hauptaufgaben dieser Arbeit lag in der Implementation der beiden Modelle. Ausgewählte Details befinden sich im Kapitel 4. Die vollständige Schnittstellenbeschreibung befindet sich im Anhang A.

Die Simulationsergebnisse und deren Bewertung sind im Kapitel 5 zusammengefaßt. In diesem Kapitel befindet sich auch der abschließende Vergleich beider Modelle.

2. Termrepräsentationen

Bevor in diesem Kapitel auf verschiedenen Methoden zur Repräsentation von Termen eingegangen wird, sollen einführend einige Begriffe erläutert werden.

2.1 Terme

Definition 2.1 (Term)

- Ein Konstantenzeichen c ist ein Term.
- Sind t_1 bis t_n Terme und ist f ein n-stelliges Funktionszeichen, so ist auch $f(t_1, ..., t_n)$ ein Term. ■

Wird die Arität eines Funktionszeichens explizit mit angegeben, so wird diese durch einen Schrägstrich vom Funktionszeichen getrennt. Konstanten werden als Funktionszeichen mit der Stelligkeit Null betrachtet.

Beispiel 1 Gegeben sei eine Menge von Funktionszeichen: $a/0, b/0, c/0, f/2, g/2$. Aus diesen Funktionszeichen bildbare Terme sind u.a.:

$$f(a, g(f(a, b)))$$
$$g(c)$$
$$a$$

□

2.2 Klassische Repräsentationen

2.2.1 Hierarchische Strukturen

In herkömmlichen Systemen werden komplexe Strukturen meist als verkettete Objekte repräsentiert. Es existieren Knoten (z.B. deklariert als Record), die Felder mit Zeigern auf andere Knoten enthalten. Somit sind beliebig strukturierte Gebilde formbar, z.B. Bäume, Listen oder Graphen. Auf die Information innerhalb dieses Knotens kann relativ leicht zugegriffen werden. Die Informationen aus der gesamten Struktur auszulesen, ist ungleich schwieriger, da hierfür den einzelnen Verweisen nachgegangen werden muß. Die Zeiger beinhalten keine Information über die referenzierten Objekte[1]. Will man dies ändern, muß neben den Zeigern noch inhaltliche Information abgelegt werden. Dieses Verfahren ist jedoch nur praktikabel, solange die Verweise kleiner sind, als die referenzierten Objekte. Es entsteht eine reduzierte Repräsentation des referenzierten Objektes im referenzierenden Objekt.

2.2.2 Gerichtet Azyklische Graphen

Eine abstraktere Form der Termrepräsentation sind Graphen. Hierbei sind die *markierten gerichteten Graphen* eine bedeutende Teilklasse der allgemeinen Graphen und eignen sich sehr gut zur Repräsentation von Termen.

Definition 2.2 (Graph) Ein Graph G besteht aus einer nicht-leeren Menge V von Knoten und einer Menge E von Kanten; man schreibt

$$G = (V, E).$$

Eine Kante ist durch ihre Begrenzungspunkte, die Knoten $x, y \in V$, bestimmt. *Gerichtete* Kanten werden durch das (geordnete) Knoten*paar* (x, y), *ungerichtete* durch die zweielementige Knoten*menge* beschrieben. *Gerichtete Graphen* besitzen ausschließlich gerichtete Kanten, *ungerichtete Graphen* haben ungerichtete Kanten. ■

Definition 2.3 (LDG) Ein *markierter gerichteter Graph G* ist ein gerichteter Graph mit einer Markierungsfunktion $\lambda : V \to \Sigma$, die den Wertebereich eines endlichen Alphabets Σ besitzt; man schreibt

$$G = (V, E, \lambda).$$

■

Definition 2.4 (LDAG) Ein LDG wird zyklenfrei oder azyklisch genannt, wenn eine *topologische Ordnung* der Knoten existiert. Eine topologische Ordnung auf einem gerichteten azyklischen Graphen $G = (V, E)$ ist eine lineare Ordnung auf den Knoten von G, so daß, wenn G eine Kante (u, v), mit $u, v \in V$, enthält, u in dieser Ordnung vor v steht. ■

Definition 2.5 (RLDAG) Ein *rooted LDAG (RLDAG)* ist ein LDAG in dem genau ein Knoten mit dem Eingrad Null existiert – der Wurzelknoten. Der *Ausgrad* eines Knotens v gibt die Anzahl der aus einem Knoten herausführenden Kanten $|\{x \mid (v, x) \in E\}|$, der Eingrad die Anzahl der in einen Knoten v hineinführenden Kanten an,$|\{x \mid (x, v) \in E\}|$. Der Ausgrad eines Graphen G ist der maximale Ausgrad aller seiner Knoten $v \in V$. ■

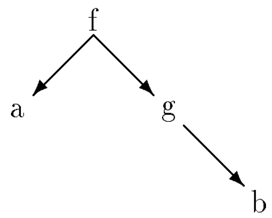
[1]Durch geeignete Auswahl der Adressen ist es denkbar, z.B. auf gerade Adressen nur Terminalsymbole und auf ungerade Adressen nur Nichtterminalsymbole abzuspeichern. Durch diese Anordnung enthält der Zeiger eine Typinformation über das referenzierte Objekt.

Jeder LDAG $G = (V, E, \lambda)$ (mit $V_0 \subseteq V$ der Menge von Knoten mit Eingrad Null) kann eineindeutig auf einen RLDAG $G' = (V', E', \lambda')$ abgebildet werden, indem ein neuer Knoten r, sowie Kanten von diesem Knoten zu allen anderen Knoten mit Eingrad Null eingefügt und die Abbildung λ um die Abbildung von r auf die leere Markierung ϵ erweitert wird; also $V' = V \cup \{r\}$, $E' = E \cup \{(r, v) \mid v \in V_0\}$ und $\lambda'(r) = \epsilon$.

Repräsentation von Termen als LDAGs

Terme können durch LDAGs repräsentiert werden, indem die Funktionszeichen auf Knoten ($Ausgrad > 0$) und die Variablen- und Konstantenzeichen auf Terminalknoten ($Ausgrad = 0$) abgebildet werden.

Beispiel 2 Repräsentation des Termes $f(a, g(b))$ als LDAG

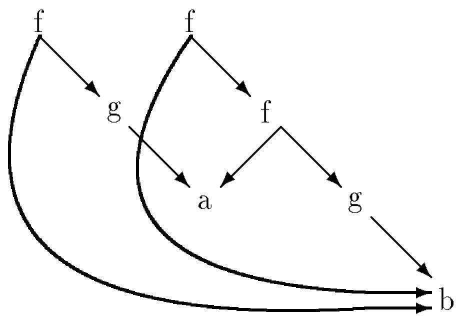


□

Repräsentation von Mengen von Termen

Eine Menge von Termen kann durch einen vergrößerten RLDAG $G = (R, V, E, \lambda)$ repräsentiert werden, indem dieselbe Prozedur wie bei der Repräsentation von Termen auch auf Teilterme, die in verschiedenen Termen vorkommen, angewendet wird. Dabei erhält man eine Menge R ($R \subseteq V$) von Wurzelknoten für die jeweiligen individuellen Terme.

Beispiel 3 Repräsentation der Menge {f(a,g(b)), f(b,f(a,g(b))), f(b,g(a)) } als RLDAG



□

2.3 Konnektionistische Repräsentationen

Durch die Beschränktheit der Ressourcen tritt bei den klassischen Termrepräsentationen ein Effekt auf, der als *brittleness* bezeichnet wird. Hierunter versteht man das totale Versagen der Repräsentation bei Überschreitung der Ressourcengrenzen. Wenn eine bestimmte Kodierungsvorschrift für Terme vorausgesetzt wird, kann schon zuvor entschieden werden, ob sich ein Term repräsentieren läßt oder nicht. Will man diese Grenze verschieben, so müßen entweder Ressourcen hinzugefügt, oder die Kodierungsvorschrift verändert werden.

Der Anspruch an konnektionistische Systeme ist, daß sie bei knapper werdenden Ressourcen die Repräsentation weiterhin erhalten können, und sich dies nur auf die Qualität der Repräsentation niederschlägt.

Eine andere Eigenschaft konnektionistischer Systeme, die Generalisierungsfähigkeit, ist ebenfalls interessant. Es werden nicht mehr alle möglichen Muster präsentiert, sondern nur noch einige Beispiele. Anhand dieser Beispiele verallgemeinert das System auf die Gesamtheit der Muster. Das System ist damit in der Lage, auch auf bisher noch nicht präsentierte Muster adäquat zu reagieren.

Eine zur Generalisierungsfähigkeit gegenläufige Eigenschaft ist die Fehlertoleranz. Von einem fehlertoleranten System wird erwartet, daß es trotz gestörter Eingabe, die richtige Antwort, d.h. die Antwort entsprechend dem ungestörten Original, liefert.

2.3.1 Lokale vs. verteilte Repräsentation

Den Aktivierungsmustern einer Menge von Einheiten, werden Bedeutungen, Objekte oder Konzepte zugeordnet. Bei der Repräsentation von Konzepten[2] werden:

a) lokale Repräsentationen (1 Einheit = 1 Konzept) und

b) verteilte Repräsentationen (mehrere Einheiten = 1 Konzept)

unterschieden. Die lokale Repräsentation ist die einfachste Form der Kodierung. Jedem Konzept wird eine Einheit im Neuronalen Netz zugeordnet. Die Verbindungen repräsentieren die Beziehungen zwischen diesen Konzepten. Die Ähnlichkeit zu semantischen Netzen ist auffällig.

Welche Bedeutung ein Aktivierungsmuster besitzt, kann bei einer lokalen Repräsentation praktisch „abgelesen“ werden, wodurch das resultierende Verhalten leicht nachvollziehund erklärbar ist.

[2]Konzepte werden hier als elementare Sinneinheiten betrachtet.

Die Verteilung der Ressourcen auf die einzelnen Konzepte ist mehr oder weniger gleichmäßig. Daher belegen sehr selten benötigte Konzepte die gleiche Menge an Ressourcen, wie häufig genutzte.

Die Ähnlichkeit von Konzepten wird in der Repräsentation nicht berücksichtigt.

Trotz dieser Nachteile enthalten die meisten konnektionistischen Systeme, zumindest in der Eingabe und/oder Ausgabe, lokale Repräsentationen.

Im Gegensatz zur lokalen, wird bei der verteilten Repräsentation versucht, ein Konzept auf mehrere Einheiten zu verteilen. Dadurch ist es möglich, daß sich mehrere Konzepte überlagern können und die Rechenleistung der Einheiten effizienter genutzt wird. Überlagerungen können als Ähnlichkeit der Konzepte interpretiert werden. Vorteilhaft an einer verteilten Speicherung eines Konzeptes ist, daß bei Wegfall oder Verfälschung einer Teilkomponente[3] die Repräsentation des Konzeptes nur gestört, jedoch nicht zerstört wird. Hieraus ergeben sich Eigenschaften wie *Fehlertoleranz* und *Generalisierungsfähigkeit*.

2.3.2 Konstruktion vs. Lernen

Die Struktur, d.h. die Verbindungen zwischen den einzelnen Verarbeitungseinheiten, wird meist durch eine sogenannte Verbindungs- bzw. Gewichtsmatrix[4] beschrieben. Die Frage ist, wie findet man eine geeignet Struktur für ein gegebenes Problem?

Die erste und einfachste Herangehensweise ist rein konstruktiv. Man überträgt das Wissen über ein Problem direkt in ein Netz und konstruiert auf diese Weise die Gewichtsmatrix von Hand. Für einfache und kleine Probleme mag dies durchaus praktikabel sein. Werden die Probleme größer, oder ist das eigene Wissen über die Vielfältigkeit der Domäne zu gering, so bieten sich automatisierte Verfahren an. Außerdem wird durch den konstruktiven Prozeß meist eine lokale Repräsentation erzeugt.

Überträgt man die Bestimmung der Gewichtsmatrix einem unabhängigen (externen) Prozeß, der die Gewichtsmatrix als Gesamtheit analytisch bestimmt, so spricht man von einem globalen Verfahren. Solche globalen Verfahren sind meist sehr unhandlich und widersprechen dem Grundgedanken der Parallelisierung von konnektionistischen Systemen [Dor91].

Wird der Prozeß der Adaption der Gewichtsmatrix dem Netz selbst übertragen, so spricht man von Lernen oder Selbstorganisation. Aus heutiger Sicht ist dies die weitaus wichtigste und am häufigsten angewendete Form der Bestimmung der Gewichtsmatrix. Das Netz paßt die Gewichtsmatrix, geleitet von einfachen Regeln, selbst an. Wird in den Regeln ein externer Feedback, z.B. eine Sollreaktion, verwendet, so spricht man von *überwachtem Lernen*, fehlt dieser, nennt man es *unüberwachtes Lernen*. Da beim Lernen anhand von Beispielen einer in ihrer Gesamtheit meist unzureichend bekannten Welt nur ein Bruchteil

[3] Diese Teilkomponente wird in der Literatur auch als *Microfeature* bezeichnet und hat inhaltlich keine direkte Bedeutung sondern stellt nur eine Facette des Konzeptes dar.

[4] Gehen die Verbindungen nur mit der Eigenschaft „existent/ nicht existent“ ein, so spricht man von Verbindungsmatrix. Ist es jedoch auch noch von Interesse, wie „stark“ eine Verbindung ist, so nennt man die Matrix Gewichtsmatrix.

der Informationen benötigt wird wie bei der Konstruktion, ist das Lernen dem konstruktiven Herangehen meist überlegen. Das System erzeugt ein internes Weltmodell, welches das durch die Trainigsmenge induzierte Weltmodell approximiert.

Außerdem entstehen durch den Lernprozeß, praktisch nebenbei, die verteilten Repräsentationen, die bei der Konstruktion ein großes Problem darstellten.

Wie gut die erzeugte Approximation gegenüber der wirklichen Welt ist, können nur überprüfende Experimente belegen.

Die Praxis hat gezeigt, daß viele Probleme, die konstruktiv nicht mehr zu bewältigen waren, durch geschickten Einsatz von adaptiven Systemen (z.B. Neuronalen Netzen) durchaus befriedigend gelöst werden konnten ([Zel94, RSM91]).

2.3.3 Strukturen Neuronaler Netze

Im letzten Abschnitt wurde beschrieben, wie die Strukturen konnektionistischer Systeme entstehen und verändert werden können. Entsprechend den Einschränkungen, die bezüglich der Struktur gemacht werden, kann man bestimmte Grundtypen Neuronaler Netze unterscheiden.

Vorwärtsgerichtete Netze: Keine Einheit enthält eine Eingabe, die von ihrer eigenen Ausgabe direkt oder indirekt abhängt.

Lassen sich einzelne Schichten lokalisieren, so spricht man von *geschichteten* Netzwerken. Existieren nur Verbindungen zwischen direkt benachbarten Schichten, so spricht man von *streng geschichteten* Netzen. Vorwärtsgerichtete Netze besitzen kein „Gedächtnis“ und können somit bestenfalls stetige Funktionen approximieren.

Rekurrente Netze: Es existieren Einheiten im Netz, die in die Bestimmung ihrer Aktivierung vergangene Aktivierungen von sich selbst direkt oder indirekt einbeziehen. Rekurrente Netze erhalten durch die Fähigkeit vergangene Aktivierungen zu berücksichtigen, eine Form von „Gedächtnis“. Man kann sich das Verhalten als Automat erklären oder im Fall von reellwertigen Ausgaben als System von iterierten Funktionen [Leh96].

Strukturveränderliche Netze: Die Anzahl der Einheiten oder die Zahl der Gewichte ist während der Existenz des Netzes nicht konstant. Die Verringerung der Gewichtsanzahl läßt sich auch durch Nullsetzen der entsprechenden Gewichte erreichen. Algorithmen, die dies bewerkstelligen, werden auch als *Pruning*-Algorithmen [Zel94] bezeichnet. Ein Netz, das während des Lernens Einheiten hinzufügt bzw. entfernt, wurde beispielsweise von Fritzke [Fri93] vorgestellt.

Die Struktur hat einen entscheidenden Einfluß auf die Funktionalität und damit natürlich auch auf die Repräsentationen. So können beispielsweise die symbolischen Strukturen in rekurrenten Netzen auch zeitlich verteilt sein und durch einen iterativen Prozeß schrittweise vervollständigt werden.

3. Modelle konnektionistischer Systeme

3.1 Ausgewählte Systeme

Einführend sollen einige konnektionistische Systeme betrachtet, und anhand dieser Beispiele, die verschiedenen Repräsentationsformen und strukturellen Möglichkeiten aufgezeigt werden.

Shastri: Semantische Netzwerke Shastri beschreibt in [Sha88] ein konnektionistisches Netzwerk, das in der Lage ist, mit einer eingeschränkten Klasse von Klauseln formale Schlußfolgerungen durchzuführen. Die Repräsentation von Objekten erfolgt dabei lokal. Knoten entsprechen Konzepten, Eigenschaften und Individuen. Die Verbindungen zwischen den Knoten repräsentieren die Beziehungen zwischen ihnen. Das Netz ist in der Lage, die richtige Antwort innerhalb eines begrenzten Zeitintervalls (Anzahl von Berechnungszyklen) zu liefern.

In [SA93] wurde das Netz durch Ajjanagadde und Shastri dahingehend erweitert, daß eine Bindung von Variablen an Regeln durch zeitliche Synchronisation erreicht wird. Jeder Netzwerkzyklus besteht damit aus mehreren Phasen, während denen die Netzwerkeinheiten aktiv sein können. Die Netze werden vollständig konstruiert, es findet also kein Lernen statt.

Sejnowski und Rosenberg: NETtalk Das von Sejnowski und Rosenberg konstruierte Netz wurde darauf trainiert, ihm vorgelegte englische Texte in Phoneme zu übersetzen. NETtalk ist im wesentlichen ein dreischichtiges feed-forward-Netz. Der Eingabevektor ist in sieben identische Eingabeblöcke unterteilt. Jeder Block repräsentiert einen Buchstaben bzw. ein Interpunktionszeichen. Die ersten und die letzten drei Blöcke liefern den Kontext für die richtige Bewertung des mittleren, gerade auszuwertenden Zeichens. Das Problem bei NETtalk liegt in der Beschränkung auf einen festen Kontext von jeweils drei benachbarten Zeichen. Für komplexere Aufgaben kann es notwendig sein, mehr Kontextinformationen einzubeziehen.

Dieses Mehr an Kontextinformationen versucht man mit rekurrenten Netzen zu erreichen.

In NETtalk wird eine lokale Repräsentation für die Ein- und Ausgabe gewählt. Die Repräsentation der Information in der verdeckten Schicht und den Wichtungen ist jedoch verteilt. Das Netz wird bezüglich seiner Struktur (Schichten, Anzahl

Kontexteinheiten usw.) konstruiert, die Werte der Wichtungen werden mittels eines Lernverfahrens bestimmt.

Elman-Netz Den Mangel an beschränktem Kontextwissen versucht Elman [Elm90, Elm91] mit einer rekursiven Netzwerkstruktur zu beheben. Hier ist es möglich, alle bisher durchlaufenen Aktivierungszustände in die nächste Aktivierung der verdeckten Schicht einzubeziehen. Verständlich wird dies, wenn man sich ein rekurrentes Netz in der Zeit „aufgerollt“ vorstellt. Von dieser Vorstellung ausgehend, wurde auch eine Variante des Backpropagation–Algorithmus entwickelt, Backpropagation Through Time (BPTT).

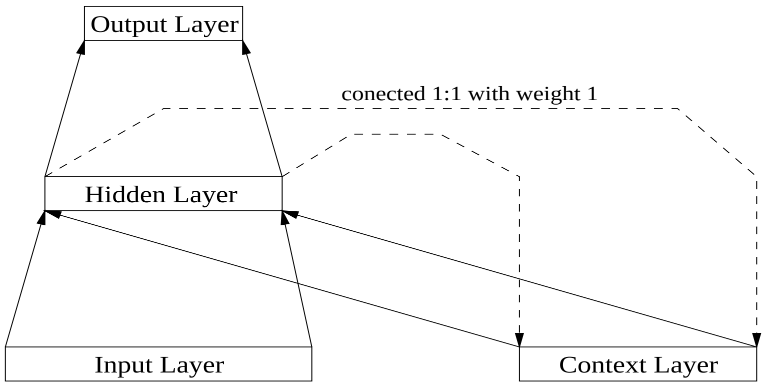


Abb. 3.1: Das Elman–Netz. Die Kontextinformation wird über den zurückgeführten Layer wieder mit in die neue Berechnung eingebracht.

Das Elman-Netz stellt nur einen möglichen Vertreter der rekurrenten Netze dar, siehe Abb. 3.1. In die verdeckte Schicht laufen die aktuellen Eingaben und die Aktivierungen der Kontexteinheiten ein, die ihre Eingaben von der letzen Ausgabe der verdeckten Schicht erhalten. Beim Elman-Netz trifft man auf eine verteilte Repräsenation und auf die Anwendung eines Lernverfahrens zur Bestimmung der Gewichte.

Rekursiver Autoassoziativer Speicher (RAAM) Der Anforderung entsprechend, baumartige Strukturen in Vektoren fester Länge zu speichern, wurde von Pollack [Pol90] die RAAM (Recursive Autoassociative Memory) vorgeschlagen. Im Prinzip handelt es sich hierbei um ein dreischichtiges feedforward-Netz. Es wird mit dem Backpropagation– Algorithmus trainiert und bildet Baumstrukturen mit einem wählbaren maximalen Verzweigungsgrad in Vektoren (Aktivierungsmuster der verdeckten Schicht) ab. Die Probleme der RAAM werden im Abschnitt 3.4.1 eingehender behandelt.

Holographisch Rekurrente Netze (HRN) Aufbauend auf einer assoziativen Verknüpfungsoperation (der Zirkularkonvolution) schlägt Plate in [Pla93] eine Neuronales Netz für die Speicherung von Sequenzen vor. Das Netz besitzt einen Kontextlayer. Die dort vorgefundene Aktivierung wird vor der Rückführung in die verdeckte Schicht mit einem gegebenen Schlüsselvektor gefaltet.

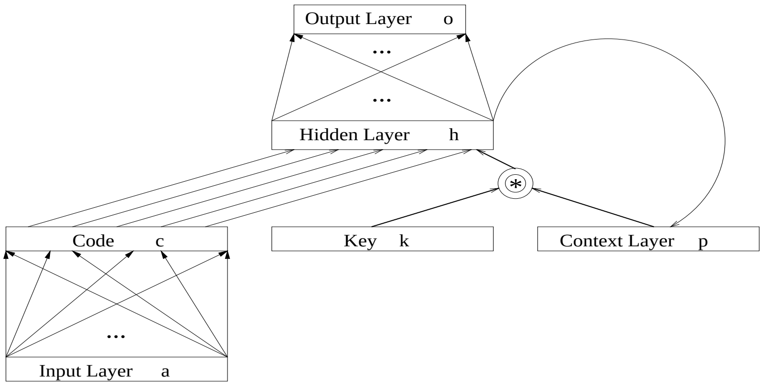


Abb. 3.2: Ein rekurrentes Neuronales Netz, welches das zirkulare Konvolutionsprodukt in der Kontextschicht nutzt (HRN nach Plate).

HRNs können mit Hilfe von Gradiantenverfahren sehr leicht angelernt werden. Die in HRNs eingesetzte Zirkularkonvolution wird auch in der HRR eingesetzt.

Holographisch reduzierte Repräsentation (HRR) Die HRR selbst ist eigentlich kein konnektionistisches System, da sie nur die Zirkularkonvolution als assoziative Verknüpfungsoperation vorsieht, die Einbindung in ein Gesamtsystem aber offenläßt. Die durch die Zirkularkonvolution erzeugte Repräsentation ist aber zumindest eine verteilte Repräsentation. Die Zirkularkonvolution läßt sich auf ein Higher-Order-Netz unter Nutzung von Sigma-Pi-Units abbilden. Durch diese Umsetzung entsteht eine Struktur, die der LRAAM ähnlich ist. Die HRR wird eingehend in Abschnitt 3.5 behandelt.

3.2 Anforderungen an verteilte Speicher

Eine der Schwierigkeiten, die symbolische und die konnektionistische Verarbeitung in Einklang zu bringen, ist die verteilte Repräsentation von rekursiven Strukturen.

Hinton schlägt in [Hin90] vor, frame-artige Strukturen zu verwenden. Allerdings erfordert eine solche Repräsentation eine Darstellung der Strukturen, die dimensionserhaltend ist. Es muß also in jeder Rekursionsstufe eine Kompression durchgeführt werden. Außerdem muß gewährleistet sein, daß die Kompression reversibel ist.

Wie leicht ein konnektionistisches System eine gegebene Problemstellung lösen kann, hängt in großem Maße auch von der Art der Repräsentation ab. Nachfolgend werden einige Eigenschaften verteilter Repräsentationen aufgeführt und untersucht.

Graceful Degradation: Hierbei handelt es sich um eine Eigenschaft, die in der Natur sehr häufig beobachtet werden kann und technischen Systemen möglichst auch eigen sein soll. Man spricht von *graceful degradation*, wenn die Funktionalität eines Systems bei einer Störung nur entsprechend der Stärke der Störung vermindert wird, das System also nicht vollständig ausfällt, wenn es sich nur um eine kleine Störung handelt. Dieser Begriff ist natürlich äußerst dehnbar, da nichts über den funktionalen Zusammenhang zwischen Störung und Verminderung der Funktionalität ausgesagt wird. Auf einen verteilten Speicher bezogen heißt das, daß trotz einer Störung der Speicherinhalt wieder rekonstruiert werden kann und nur graduell ein Abstrich an der Qualität des Inhalts gemacht werden muß, was sich beispielsweise durch überlagertes Rauschen äußern kann.

Explizitheit (sofortiger Zugriff): Die Informationen können durch einfache Operationen aus dem Speicher ausgelesen werden. Es werden darüberhinaus keine weiteren Speicherzugriffe benötigt.

Analoge Repräsentation: Ähnliche Objekte besitzen ähnliche Repräsentationen. Wie ähnlich zwei Objekte sindm wird durch ein Ähnlichkeitsmaß definiert. Handelt es sich bei den Objekten um Vektoren, so wird häufig das Skalarprodukt verwendet.

Effizienz: Die Ressourcen werden effizient genutzt.

Generalisierungsfähigkeit: Die Repräsentationen erlauben eine Generalisierung des gespeicherten Wissens. Üblicherweise versteht man darunter die „richtige“ Ausgabe des Systems auf eine bisher noch nie „gesehene“ Eingabe.

Fehlerkorrektur: Durch Redundanzen in der Repräsentation werden Fehler ausgeglichen, bzw. korrigiert.

Mustervervollständigung: Ein gegebener Teil eines gespeicherten Musters wird zum vollständigen Muster ergänzt.

Von den aufgeführten Eigenschaften sind die beiden ersten wohl die wichtigsten. Allerdings ist diese Wichtung sehr stark von der gewählten Anwendung abhängig.

Durch den expliziten Zugriff auf die Information ist es möglich, Operationen auf komplexen Strukturen durchzuführen, ohne sie dekodieren zu müssen.

Offensichtlich steht die Fähigkeit zu generalisieren im Gegensatz zur Fehlertoleranz. Hier muß ein Kompromiß zwischen beiden Eigenschaften eingegangen werden, um ein Optimum für die gewählte Anwendung zu erreichen.

3.3 Neuronale Netze

Der in dieser Arbeit gewählte Formalismus zur Beschreibung Neuronaler Netze ist dem Buch von Nauck, Klawonn und Kruse entnommen [NKK94].

Neuronale Netze bestehen aus elementaren Verarbeitungseinheiten[1], die über gewichtete Verbindungen Informationen austauschen. Die übertragenen Informationen besitzen elementaren Charakter, das heißt, es werden keine strukturierten Daten übertragen. Die

[1] Diese werden in der Literatur auch als Einheiten, Units oder (künstliche) Neuronen bezeichnet.

Verarbeitung der Informationen geschieht i.a. verteilt, jede Einheit berechnet lokal aus den ihr zugänglichen Informationen eine eigene Aktivierung und reicht diese an die mit ihr verbunden Einheiten weiter.

3.3.1 Definition

Ein Neuronales Netz läßt sich leicht als eine Struktur formalisieren, die durch eine Menge und einige Abbildungen beschreibbar ist.

Definition 3.1 Ein Neuronales Netz ist ein Tupel (U, W, A, O, NET, ex).

1. **U** ist die Menge der Verarbeitungseinheiten,
2. **W**, die Netzwerkstruktur, ist eine Abbildung $(U \times U \rightarrow \mathbb{R})$,
3. **A** ist eine Abbildung, die jedem $u \in U$ eine Aktivierungsfunktion $A_u : \mathbb{R}^3 \rightarrow \mathbb{R}$ zuordnet,
4. **O** ist eine Abbildung, die jedem $u \in U$ eine Ausgabefunktion $O_u : \mathbb{R} \rightarrow \mathbb{R}$ zuordnet,
5. **NET** ist eine Abbildung, die jedem $u \in U$ eine Netzeingabefunktion $NET_u : (\mathbb{R} \times \mathbb{R})^{|U|} \rightarrow \mathbb{R}$ zuordnet und
6. **ex** ist eine externe Eingabefunktion $ex : U \rightarrow \mathbb{R}$, die jedem $u \in U$ eine externe Eingabe in Form einer rellen Zahl $ex_u = ex(u) \in \mathbb{R}$ zuordnet.

■

Die einzelnen Bestandteile der Definition werden nun näher betrachtet.

Verarbeitungseinheiten

Die Verarbeitungseinheiten in einem Neuronalen Netz sind sehr einfache elementare Berechnungselemente, die ihre Eingabe entsprechend einer einfachen Berechnungsvorschrift, häufig ist dies das Skalarprodukt, zu einer Ausgabe transformieren.

Durch die Möglichkeit, in die Berechnung der neuen Ausgabe auch die aktuelle Aktivierung mit einfließen zu lassen, werden Speichereffekte realisiert. Die Verarbeitungseinheiten führen ihre Berechnungen vollständig parallel und unabhängig durch. Eine feinere Unterscheidung der Verarbeitungseinheiten bzgl. ihrer Funktion und Lage im Netz ist in einigen Netzwerkmodellen von Vorteil. So spricht man z.B. von Ein-/Ausgabe-Einheiten, wenn diese eine Kommunikation mit der Umwelt durchführen. In geschichteten Netzwerkmodellen, wie z.B. dem Multi-Layer-Perceptron (MLP), unterscheidet man zusätzlich zwischen Eingabe-, Ausgabe- und inneren (verdeckten) Einheiten.

Netzwerkstruktur

Die Abbildung W wird im allgemeinen als Netzwerkstruktur bezeichnet, siehe auch Abschnitt 2.3.3. Sie läßt sich als bewerteter gerichteter Graph darstellen, dessen Knoten die

Verarbeitungseinheiten sind. Die bewerteten Kanten repräsentieren dann die gerichteten Kommunikationsverbindungen zwischen den Verarbeitungseinheiten. W wird in Matrixform auch als *Konnektions-* oder *Gewichtsmatrix* bezeichnet.

w_{uv} wird (synaptisches) Gewicht, (synaptische) Verbindung oder Kopplung genannt. Die Netzwerkstruktur bildet die Grundlage der Kommunikation innerhalb eines Neuronalen Netzes. Die Ausgabe einer Verarbeitungseinheit ist Teil der Eingabe einer anderen Verarbeitungseinheit, wenn zwischen ihnen eine Kopplung existiert.

Die Gewichte werden bezüglich ihrer Größe in drei Kategorien eingeteilt:

$$\begin{aligned} w_{uv} = 0 &\ : \ \text{es existiert keine Verbindung von } u \text{ nach } v \\ w_{uv} < 0 &\ : \ \text{inhibitorische (hemmende) Verbindung von } u \text{ nach } v \\ w_{uv} > 0 &\ : \ \text{excitatorische (anregende) Verbindung von } u \text{ nach } v \end{aligned}$$

Die Abbildung W ist während eines Lernprozesses veränderbar. Das Wissen, was ein Neuronales Netz während des Lernvorganges erworben hat, bzw. was ihm während der Konstruktion mitgegeben wurde, wird innerhalb der Netzstruktur durch die Gewichte repräsentiert. Für eine Manipulation von W wird eine Lernregel[2] benötigt, die W so anpaßt, daß eine bestimmte Aufgabe bezüglich eines gewählten Fehlermaßes besser oder vollständig gelöst wird.

Aktivierungsfunktion

Die Abbildung A ordnet jeder Verarbeitungseinheit eine Aktivierungsfunktion zu. Diese Funktion berechnet die aktuelle Aktivierung a_u der Verarbeitungseinheit anhand der vorangegangenen Aktivierung $a_u^{(alt)}$, der aktuellen Netzeingabe net_u und der aktuellen externen Eingabe ex_u.

$$a_u = A_u({a_u}^{(alt)}, net_u, ex_u)$$

In vielen Fällen werden alle Verarbeitungseinheiten ihre Berechnungen nach der gleichen Vorschrift durchführen. Sollte es jedoch notwendig sein, daß einzelne Einheiten eine andere Berechnung durchführen müssen, so ist das durch die Abbildung A ebenfalls möglich.

Geht in die Aktivierungsfunktion nur die Netzeingabe ein, wird auf eine vereinfachte Form zurückgegriffen, $A_u : \mathbb{R} \rightarrow \mathbb{R}$.

Die meisten konnektionistischen Modelle verwenden lineare, Treppen-, Stufen- oder sigmoide Funktionen.

Die Wahl der Aktivierungsfunktion ist sehr stark modellabhängig. Für einige Lernverfahren (z.B. Error–Backpropagation) sind differenzierbare Aktivierungsfunktionen notwendig.

Ausgabefunktion

Die Ausgabefunktion dient der Anpassung unterschiedlicher Wertebereiche der Eingaben und der Ausgaben verschiedener Verarbeitungseinheiten. Häufig ist die Ausgabefunktion

[2] Im Falle eines MLP ist dies für gewöhnlich eine Variante des Error–Backpropagation–Algorithmus.

die Identitätsfunktion.

Netzeingabe

Die Abbildung NET ordnet jeder Einheit eine Netzeingabefunktion NET_u zu. Üblicherweise wird auch hier für alle Einheiten eine Funktion verwendet. In vielen konnektionistischen Systemen kommt die gewichtete Summe über die Ausgaben aller Einheiten zum Einsatz:

$$net_u = \sum_{v \in U} w_{uv} \cdot o_v - \theta_u.$$

Der Schwellenwert θ_u kann über einen zusätzlichen Eingang mit dem Gewicht $-\theta_u$ und einer konstanten Eingabe von eins ersetzt werden. Somit läßt sich die obige Formel immer in eine Form ohne θ bringen.

Externe Eingabefunktion

Durch die externe Eingabefunktion ex steht das Neuronale Netz mit seiner Umwelt (Außenwelt) in Verbindung und erhält über diesen Weg seine Eingaben. Das Netz reagiert auf diese externen Eingaben mit einer Veränderung seiner Aktivierungen und erzeugt dadurch eine Ausgabe. In vielen Neuronalen Netzen ist nur eine Teilmenge der Verarbeitungseinheiten über eine externe Eingabe zu erreichen, d.h. ex ist nur über einer Teilmenge von Einheiten $U_I \subseteq U$, den Eingabeeinheiten, definiert.

3.3.2 Arbeitsweise Neuronaler Netze

Bei der nun folgenden Betrachtung wird davon ausgegangen, daß die Gewichtsmatrix des Neuronalen Netzes konstant bleibt. Die Vorgänge während des Lernens bleiben somit vorerst unberücksichtigt.

Für die Betrachtung der zeitlichen Abläufe innerhalb eines Neuronalen Netzes werden drei Phasen definiert:

1. die Ruhephase,
2. die Eingabephase und
3. die Arbeitsphase.

In der *Ruhephase* bleiben die Aktivierungen aller Einheiten konstant. Dieser Zustand wird erreicht, wenn sich die Aktivierung keiner Verarbeitungseinheit, trotz Neuberechnung, ändert. Bei rückgekoppelten Netzen, ist es durchaus möglich, daß es eine solche Ruhephase nicht gibt. In einem solchen Falle wird davon ausgegangen, daß das Netz „eingefroren“ und so künstlich in einen Ruhezustand versetzt werden kann. Befindet sich ein Netz in einer Ruhephase, so bezeichnet man dies auch als *stabilen Zustand.*

Die *Eingabephase* schließt sich direkt an die Ruhephase an. In dieser Phase werden die Aktivierungen der Eingabeeinheiten berechnet. Als Eingabeeinheiten werden genau die Einheiten bezeichnet, für die die Abbildung ex definiert ist.

In der *Arbeitsphase* werden die Aktivierungen aller Einheiten berechnet und über die durch die Wichtungsmatrix definierten Verbindungen im Netz propagiert.

Die Berechnung der Aktivierung einer Einheit kann *asynchron* oder *synchron* bezüglich aller anderen Einheiten durchgeführt werden. Asynchron bedeutet, daß jede Einheit für sich zu einem beliebigen Zeitpunkt die Eingänge auswertet und ihre Aktivität anpaßt. Bei einer synchronen Berechnung werten alle Einheiten gleichzeitig ihre Eingänge aus und propagieren die Berechnungsergebnisse ebenfalls gleichzeitig.

3.3.3 Strukturveränderung – Lernen

Ein wesentlicher Aspekt Neuronaler Netze ist die Fähigkeit, mittels eines Lernverfahrens die Struktur (also die Abbildung W) des Netzes so zu adaptieren, daß eine gegebene Lernaufgabe erfüllt wird.

Definition 3.2 (Eingabe-,Ausgabemuster) Sei N ein beliebiges Neuronales Netz mit einer Menge von Verarbeitungseinheiten U, der externen Eingabefunktion ex und der für eine Ausgabeeinheit v gültigen Ausgabefunktion O_v.

$$U_I = \{u_1, ..., u_n\} \subseteq U$$

sei die Menge der Eingabeeinheiten und

$$U_O = \{v_1, ..., v_m\} \subseteq U$$

die Menge der Ausgabeeinheiten. Weiterhin bezeichne *dom* den Bildbereich (Domäne) einer Funktion.

1. Eine Eingabe i (Eingabemuster) von N ist ein Element der Menge

$$\mathcal{I} = \{(x_1, ..., x_n) | x_i \in dom(ex)\} \subseteq \mathbb{R}^{|U_I|}.$$

2. Eine Ausgabe t (Ausgabemuster) von N ist ein Element der Menge

$$\mathcal{T} = \{(y_{v_1}, ..., y_{v_n}) | y_{v_i} \in dom(O_{v_i})\} \subseteq \mathbb{R}^{|U_O|}.$$

■

Für die Durchführung des Lernvorganges ist neben einem geeigneten *Lernalgorithmus* die Vorgabe einer *Lernaufgabe* erforderlich. Man kann zwischen *freien* und *festen* Lernaufgaben unterscheiden.

Definition 3.3 (freie Lernaufgabe) Eine freie Lernaufgabe $\mathcal{L} \subseteq \mathcal{I}$ eines Neuronalen Netzes ist eine Menge von Eingabemustern. Das Neuronale Netz soll auf jede Eingabe $i \in \mathcal{L}$ der Lernaufgabe eine Ausgabe $t \in \mathcal{T}$ bestimmen. Ein Neuronales Netz erfüllt eine freie Lernaufgabe, wenn es zu jeder Eingabe eine Ausgabe bestimmt hat, so daß bezüglich eines geeigneten Abstandsmaßes[3] ähnliche Eingaben auch ähnliche Ausgaben hervorrufen. ■

Definition 3.4 (feste Lernaufgabe) Eine feste Lernaufgabe $\tilde{\mathcal{L}} \subseteq \mathcal{I}$ eines Neuronalen Netzes ist eine Menge von Ein-/Ausgabepaaren. Das Neuronale Netz soll auf jede Eingabe i eines Paares $(i,t) \in \tilde{\mathcal{L}}$ mit der dazugehörigen Ausgabe t reagieren. Ein Neuronales Netz erfüllt die feste Lernaufgabe, wenn es zu jeder Eingabe die laut Lernaufgabe dazugehörige Ausgabe erzeugt. ■

Definition 3.5 (Lernalgorithmus) Ein Lernalgorithmus ist ein Verfahren, das anhand einer Lernaufgabe die Netzwerkstruktur W eines Neuronalen Netzes verändert. Erfüllt das Netz diese Lernaufgabe nach Anwendung des Verfahrens, bzw. wird eine vorher festgelegte Fehlerschranke unterschritten, so war der Lernalgorithmus erfolgreich, andernfalls erfolglos. Ein Lernalgorithmus, der eine freie Lernaufgabe verwendet, heißt nichtüberwachter (unsupervised) Lernalgorithmus. Verwendet er eine feste Lernaufgabe, so heißt er überwachter (supervised) Lernalgorithmus. ■

Definition 3.6 (Fehlermaß) Seien $\tilde{\mathcal{L}}$ eine feste Lernaufgabe und $l \in \tilde{\mathcal{L}}$ ein zu lernendes Musterpaar.

1. Ein Fehlermaß für einen überwachten Lernalgorithmus ist eine Abbildung

   $$e : \mathbb{R}^{|U_O|} \times \mathbb{R}^{|U_O|} \to \mathbb{R}_0^+,$$

   so daß für alle $a, b \in \mathbb{R}^{|U_O|}$

   $$e(a,b) = 0 \iff a = b$$

   gilt. Der Fehler $e^{(l)}$, den das Neuronale Netz bei der Verarbeitung des Musterpaares $l \in \tilde{\mathcal{L}}$ macht, ist durch $e^{(l)} = e(t^{(l)}, o^{(l)})$ gegeben. Dabei ist $o^{(l)}$ die Ausgabe des Netzes auf das Eingabemuster $i^{(l)}$ und $t^{(l)}$ die erwünschte, durch die Lernaufgabe $\tilde{\mathcal{L}}$ vorgegebene Ausgabe (target).

2. Der Fehler $e_u^{(l)}$ einer Ausgabeeinheit $u \in U_O$ bei der Verarbeitung des Musterpaares $l \in \tilde{\mathcal{L}}$ ist die Differenz zwischen dem durch die Lernaufgabe vorgegebenen Ausgabewert $t_u^{(l)}$ und des tatsächlichen Ausgabewertes $o_u^{(l)}$:

   $$e_u^{(l)} = t_u^{(l)} - o_u^{(l)}.$$

■

[3] Ein Abstandsmaß definiert, wie ähnlich sich zwei Vektoren sind.

3.3.4 Der Backpropagation–Algorithmus

Die Berechnung des Fehlers in der Ausgabeschicht ist durch die Vorgabe eines Sollwertes kein Problem. Die von Widrow und Hoff erstmals vorgeschlagene Lernregel, die sogenannte *Delta-Regel*, läßt sich nur für Netze ohne verdeckte Schichten anwenden. Längere Zeit war es unklar, wie die Wichtungen der verdeckten Schicht zu ändern sind, damit der vom Netz gemachte Fehler minimiert werden kann. Der *Backpropagation–Algorithmus* stellt eine Verallgemeinerung der Delta–Regel dar. Hier werden die in einer Schicht gemachten Fehler rückwärts durch das Netz propagiert. Aufgrund dieses Fehlersignals können die Einheiten der verdeckten Schicht nun ihrerseits den Fehler bestimmen und diesen weiterpropagieren. Die Veröffentlichung von Rummelhart, Hinton und Williams [RHW86] verhalf dem Backpropagation–Algorithmus zum Durchbruch. Die Herleitung des Algorithmus ist in verschiedenen Büchern über Neuronale Netze zu finden, so daß hier nur noch die Definition aufgeführt wird, [Zel94, Roj93, NKK94].

Definition 3.7 Gegeben sei ein Multi-Layer-Perceptron (MLP) mit den Einheiten $U = U_1 \cup ... \cup U_n$ und der sigmoiden Aktivierungsfunktion

$$A_u(net_u) = f(net_u) = \frac{1}{1 + e^{-net_u}}$$

für alle $u \in U_i, i \geq 2$, sowie eine feste Lernaufgabe $\tilde{\mathcal{L}}$. Der überwachte Lernalgorithmus, der die Veränderung der Gewichtsstruktur W vom MLP nach der Propagation des Eingabemusters $i^{(p)}, p \in \tilde{\mathcal{L}}$ durch

$$\Delta_p w_{uv} = \eta \delta_v^{(p)} a_u^{(p)}$$

mit $u \in U_{i-1}, v \in U_i, 2 \leq i \leq n$ und $\eta > 0$, bestimmt, wobei

$$\delta_u = \begin{cases} f'(net_u^{(p)})(t_v^{(p)} - a_u^{(p)}) & \text{falls } u \in U_n \\ f'(net_u^{(p)}) \sum_{v \in U_{j+1}} \delta_v^{(p)} w_{uv} & \text{falls } u \in U_j, 2 \leq j \leq n-1 \end{cases}$$

gilt, heißt verallgemeinerte *Delta–Regel* oder *Backpropagation–Algorithmus*. Dabei ist $a_u^{(p)}$ die Aktivierung der Einheit u nach der Propagierung des Eingabemusters $i^{(p)}$ und $t_v^{(p)}, v \in U_n$ die durch das Ausgabemuster $t^{(p)}$ für eine Ausgabeeinheit v vorgegebene Ausgabe (Aktivierung). Der Backpropagation–Algorithmus minimiert den vom Netz gemachten Fehler

$$e = \sum_{p \in \tilde{\mathcal{L}}} \sum_{v \in U_n} (t_v^{(p)} - o_v^{(p)})^2.$$

■

3.3.5 Backpropagation Through Time (BPTT)

Der im vorigen Abschnitt beschriebene Backpropagation–Algorithmus ist in dieser Form nur für feedforward-Netze, also Netze ohne Zyklen geeignet. Nach Minsky und Papert

[MP69] läßt sich jedes zyklische Neuronale Netz mit diskreten Zeitschritten in ein äquivalentes feedforward-Netz transformieren. Bildhaft wird das Netz in der Zeit „abgerollt“. Ein wesentlicher Nachteil ist, daß dieses Vorgehen bei n betrachteten Zeitschritten das n-fache der Ressourcen des betrachteten rekurrenten Netzes benötigt. Im Bild 3.3 ist ein kleines rekurrentes Netz dargestellt. Die Eingaben werden an das Netz angelegt, das Netz rechnet einige Schritte, anschließend liest man die Ergebnisse wieder aus. Die Ausgaben werden mit dem Sollwert verglichen, und aus der Differenz wird das Fehlersignal bestimmt. Die Rückpropagierung geschieht durch alle Neuronen, für alle Zeitschritte. Um die Fehlersignale der früheren Zeitschritte bestimmen zu können, müssen die Ausgaben der Neuronen zu jedem Zeitschritt gespeichert werden. Die Gewichtskorrekturen werden über alle Iterationsstufen summiert und gespeichert. Die Änderung der Wichtungen erfolgt am Ende dieses Prozesses um eben diese Summe.

Die veränderte Propagierungsregel für das Netz in Bild 3.3 lautet:

$$o_u(t+1) = f(net_u(t)) = f(\sum_{v \in U} o_v(t) w_{vu} + ex_u(t))$$

Dabei ist $o_v(t)$ die Ausgabe des Neurons v zum Zeitpunkt t und $ex_u(t)$ die Eingabe in Neuron u zum Zeitpunkt t.

Durch ein als zeitliches Entfalten *(unfolding in time)* bezeichnetes Verfahren werden alle Neuronen und Gewichte für jeden zu berechnenden Zeitschritt kopiert. Das aus Bild 3.3 resultierende feedforward-Netz ist in 3.4 zu sehen. Da man jetzt ein gewöhnliches feedforward-Netz hat, kann man dieses auch ganz gewöhnlich mit dem Backpropagation-Algorithmus trainieren. Die Korrektur der Wichtungen darf aber erst nach vollständiger Rückpropagierung erfolgen, da alle Ebenen mit den gleichen Wichtungswerten rechnen müssen. Die durch den Lernalgorithmus ermittelten Wichtungsänderungen innerhalb der Zeitschritte werden gemittelt, und alle Wichtungen in den verschiedenen Zeitschritten werden gleichmäßig geändert, da im rekurrenten Netz jede Wichtung nur einmal existiert.

Ein Problem bei dieser Technik ist, daß der Speicherbedarf $O(t_{max}n^2)$ ist. Dies ist nicht unbedingt nötig, da in allen Zeitschritten die gleichen Gewichte benutzt werden. Eine aus dieser Überlegung abgeleitete Optimierung wird auch als *shared weights* bezeichnet. Jedes Neuron enthält nicht mehr „seine“ Wichtungen, sondern nur noch einen Verweis auf die Gewichtsmatrix. Somit teilen sich die Neuronen in den einzelnen Zeitschritten denselben Speicherplatz für die Gewichte. Dieser Schritt vereinfacht aber auch das Netz als solches, denn nun müssen die Neuronen nicht mehr für jeden Zeitschritt kopiert werden, es reicht, wenn die Ausgaben der Reihe nach auf einem Stack gespeichert werden. Bei der Rückpropagierung werden die Ausgabewerte rückwärts vom Stack geholt und die Gewichte im Anschluß nur einmal um die Summe der Gewichtskorrekturen angepaßt. Diese Anordnung ist in Bild 3.5 zu sehen.

Ist die Zeitdauer der zeitlich veränderlichen Muster bekannt, so kann man die Stapelspeicher durch Felder fester Länge ersetzen. In diesem Fall spricht man von *Truncated Backpropagation Through Time.*

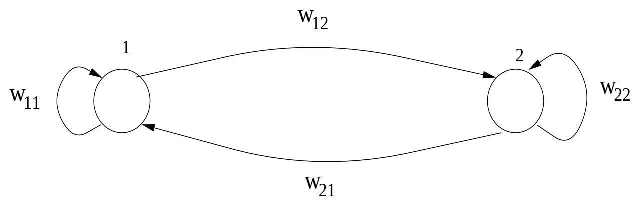


Abb. 3.3: Ein einfaches rekurrentes Netz.

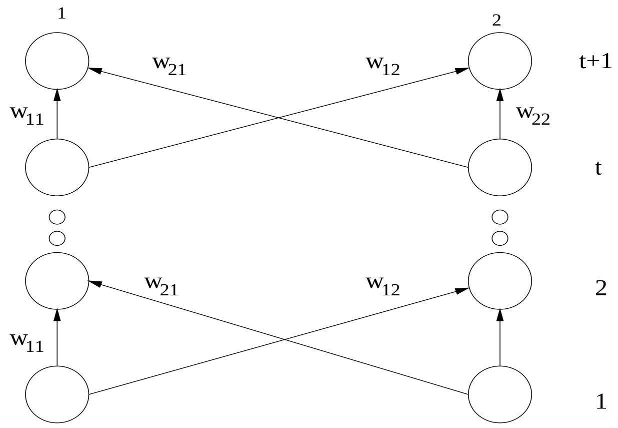


Abb. 3.4: Das zur letzten Abbildung äquivalente feedforward-Netz.

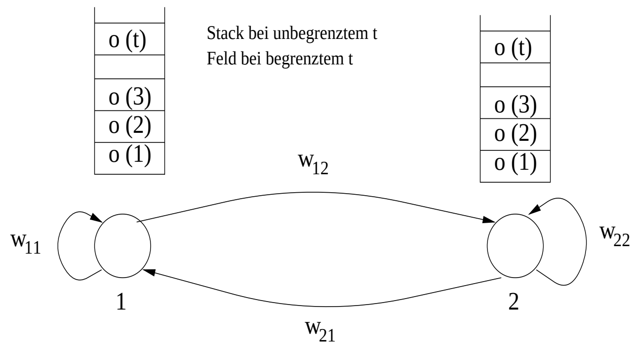


Abb. 3.5: Die während der einzelnen Zeitschritte erzeugten Ausgaben werden der Reihe nach auf dem Stack abgelegt. Nach dem letzten Zeitschritt, werden die Ausgaben schrittweise vom Stack gelöscht und zur Berechnung der δ–Werte benutzt. Ist die Anzahl der Zeitschritte nach oben begrenzt und diese Grenze bekannt, dann läßt sich der Stack auch durch ein Feld fester Länge ersetzen.

3.3.6 Backpropagation Through Structure (BPTS)

Küchler und Goller haben in [GK95] die Idee des BPTT auf gerichtete azyklische Graphen verallgemeinert. Werden Zeitreihen als Sequenzen betrachtet, so sind sie Spezialfälle von gerichteten azyklischen Graphen. Küchler und Goller benutzten diese Erweiterung zur Klassifikation von Termen. Die Lernaufgabe wird dabei so definiert, das die Eingabemuster aus einer symbolischen Domäne stammen und die Targetwerte aus dem reellen Vektorraum. Die symbolische Domäne ist die Gesamtheit aller RLDAGs bezogen auf eine gegebene Menge von Funktionszeichen, und wird im folgenden mit $\mathcal{S}$ bezeichnet.

Für das Problem der Termrepräsentation mit der LRAAM wird aus einer überwachten Lernaufgabe eine unüberwachte, da $i^{(p)} = t^{(p)}$ gilt. Als Ergebnis des Trainings entstehen zwei Teilnetze mit den Funktionen Kodieren und Dekodieren, die die Abbildungen der symbolische Domäne auf den reellen Vektorraum und zurück durchführen.

Die formale Definition befindet sich im Abschnitt 3.4.3, da zuvor noch die Struktur der LRAAM eingeführt werden muß.

3.4 Rekursiver Autoassoziativer Speicher mit Marken – LRAAM

3.4.1 Von der RAAM zur LRAAM

Die RAAM

Im Jahre 1990 stellte Pollack in [Pol90] ein Netzwerk vor, mit dem es möglich war, Bäume mit einem (wählbaren) maximalen Verzweigungsgrad kompakt in einem Vektor fester Länge zu speichern. Dieses Netz nennt er RAAM (Recursive Autoassociative Memory), siehe Bild 3.6.

Die Idee dieses Netzes entspricht einem Kodier-/Dekodiernetz. Die Eingabe- und Ausgabeschicht ist jeweils k-mal so groß, wie die verdeckte (mittlere) Schicht. Das Netz wird mit Backpropagation darauf trainiert, die Identitätsfunktion zu berechnen. Da die verdeckte Schicht kleiner als die Ein- bzw. Ausgabeschicht ist, bildet die Aktivierung dieser Schicht eine reduzierte Repräsentation des Eingabemusters. Wendet man die so erzeugte Repräsentation rekursiv als Teil eines neuen Eingabemusters an, so lassen sich Baumstrukturen in einer Repräsentation fester Länge abbilden.

Beispielhaft soll der Vorgang des Speicherns eines Binärbaumes in einer RAAM gezeigt werden. Ein Binärbaum hat den maximalen Verzweigungsgrad von zwei. Es wird also eine RAAM mit der Struktur 2m–m–2m benötigt. Der Beispielbaum ist in Abb. 3.7 abgebildet.

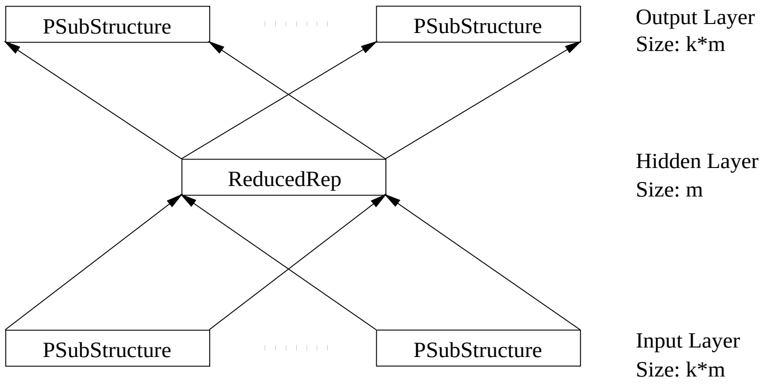


Abb. 3.6: Die allgemeine RAAM, für Bäume mit einem maximalen Verzweigungsgrad von k.

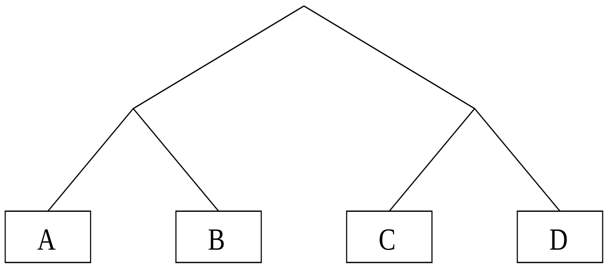


Abb. 3.7: Im Beispiel benutzter binärer Baum.

Trainiert wird die RAAM in folgender Weise:

$$\begin{array}{lccccl}
\text{Eingabe} & & \text{Verdeckte Schicht} & & \text{Ausgabe} \\
 & & & & \\
(A\ B) & \rightarrow & R_1(t) & \rightarrow & (A'(t)\ B'(t)) \\
(C\ D) & \rightarrow & R_2(t) & \rightarrow & (C'(t)\ D'(t)) \\
(R_1(t)\ R_2(t)) & \rightarrow & R_3(t) & \rightarrow & (R_1'(t)\ R_2'(t))
\end{array}$$

Zuerst werden die Teilbäume (AB) und (CD) angelernt. Ist die RAAM in der Lage, die Teilbäume richtig abzubilden, dann ist die Aktivierung der verdeckten Schicht genau die reduzierte Repräsentation des jeweils angelegten Teilbaums, also $R_1(t)$, bzw. $R_2(t)$. Die so erhaltenen Repräsentationen besitzen die gleiche Dimensionalität, wie die Blattknoten. Demzufolge können wir den Trainingsprozeß mit den neuen Repräsentationen wiederholen und erhalten eine Repräsentation für den gesamtem Binärbaum.

Der Parameter t gibt die Epoche während des Trainings an. Wurde mit dem Backpropagation-Algorithmus diese Trainigsmenge in der aufgeführten Weise (perfekt) angelernt,

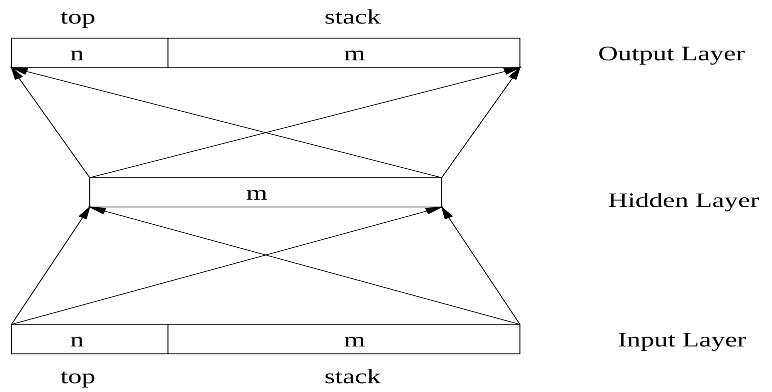


Abb. 3.8: Die Sequential RAAM

dann können folgende Aussagen über die Muster gemacht werden:

$$A' = A$$
$$B' = B$$
$$C' = C$$
$$D' = D$$
$$R_1' = R_1$$
$$R_2' = R_2$$

Diese Form des Lernens erzeugt außer einer reduzierten Repräsentation für den gesamten Baum auch solche für jeden Teilterm. Eine Variante von Backpropagation, Backpropagation Through Structure (BPTS), besitzt diese Eigenschaft leider nicht zwangsläufig, siehe Abschnitt 3.4.3.

Eine Variante der RAAM stellt die SRAAM (Sequential RAAM) dar. Mit der SRAAM können Sequenzen kodiert werden. Sequenzen lassen sich als linksverzweigende Binärbäume darstellen. Beispielsweise wird die Sequenz $(A\ B\ C)$ als $(((NIL\ A)\ B)\ C)$ kodiert. Die Ein- und Ausgabeschichten der SRAAM bestehen jeweils aus $n+m$ Neuronen, wobei die verdeckte Schicht m Neuronen enthält.

Das Modell der RAAM birgt einige technische Probleme. Das erste Problem betrifft den Lernalgorithmus selbst. Dadurch, daß die erzeugten Repräsentationen der verdeckten Schicht in die Trainingsmenge aufgenommen werden und sich diese Repräsentationen während des gesamten Trainingsprozesses anpassen, verändert sich auch die Trainingsmenge. Diese Problem wird als *Moving Target Problem* bezeichnet und charakterisiert eine während des Lernens dynamische Trainigsmenge. Die Folge ist, daß kleine Lernraten benutzt werden müssen, um durch die Anpassung eines Musters nicht die Repräsentation der anderen Symbole zu stark zu stören. Außerdem wirkt sich ein dynamische Trainingsmenge natürlich auch negativ auf die Konvergenz des Lernalgorithmus aus.

Ein weiteres Problem ist das Haltekriterium (*stopping criteria*) während des Lernens. In der RAAM werden die Terminalsymbole durch binäre Vektoren kodiert, die Nichtterminalsymbole jedoch durch reellwertige Vektoren, da in sie die Repräsentationen der verdeckten Schicht eingehen. In der RAAM werden daher zwei verschieden Toleranzmaße für Terminal- und Nichtterminalsymbole eingesetzt. Die Wahl dieser Toleranzmaße hat nach [Spe92] einen wesentlichen Einfluß auf die Fähigkeit der RAAM, die kodierten Bäume auch wieder zu dekodieren, insbesondere wenn es sich um tiefe Bäume handelt.

Das *Terminationsproblem* tritt während der Dekodierung auf. Die Frage ist, woran man erkennen kann, daß ein dekodiertes Muster ein Terminal- oder ein Nichtterminalzeichen ist? Diese Frage läßt sich auf die Entscheidung zurückführen, handelt es sich um einen binären oder einen reellwertigen Vektor. Dies ist jedoch nur schwer zu entscheiden, da die binären Vektoren ja nur Spezialfälle der reellwertigen Muster sind. Es wird also wieder ein Maß gebraucht, um zu enstcheiden, wie gut ein Muster „binär“ ist. Leider zeigt sich, daß die RAAM auch binäre Muster als reduzierte Repräsentation erzeugt. Eine andere Methode diese Unterscheidung zu treffen, wurde von Stolcke und Wu in [SW92] gezeigt. Sie benutzen eine Unit der verdeckten Schicht um zwischen Terminal- und Nichtterminalzeichen zu unterscheiden. Bei Terminalzeichen wird die Ausgabe dieser Unit auf null gesetzt, bei allen Nichtterminalzeichen auf eins. Diese Methode, auch als *Injection Method* bezeichnet, bewirkt den Aufbau eines „build-in“-Klassifikators.

Die Labeling RAAM

Die LRAAM wurde 1993 von Sperduti [Spe92] vorgestellt und ist im Prinzip eine Verallgemeinerung der RAAM für markierte Strukturen. Außerdem löst die LRAAM einige der Probleme der RAAM auf elegantere Weise, als es zuvor der Fall war.

Zu jedem Knoten im Baum wird zusätzlich eine Markierung mit abgespeichert. Weiterhin existiert für jedes Teilmuster ein zusätzliches Eingabefeld, welches bestimmt, ob es sich hierbei um eine gültige Repräsentation handelt oder um einen NIL-Zeiger. Durch diesen Kunstgriff ist es möglich, das Terminationsproblem sehr elegant zu lösen, da es nicht mehr notwendig ist, die Repräsentationen auf „Binärheit“ zu testen, sondern nur noch die zugehörigen NIL-Felder bezüglich einem festen Schwellwert auszuwerten. Dies ist ungleich einfacher, zumal es sich hierbei um ein statisches Muster handelt.

Die LRAAM, so wie sie Sperduti in seiner Arbeit [Spe92] vorstellt, ist im Bild 3.9 dargestellt.

Sperduti und Starita [SS95] benutzen die von einer LRAAM erzeugte Repräsentation als Ausgangspunkt für die Klassifikation von Termen. Es zeigte sich, daß die Trainingsphase sehr aufwendig war und die LRAAM selbst schon einen Engpaß darstellte. In einer späteren Arbeit von Küchler und Goller [KG96] wurde auf die vollständige LRAAM verzichtet und nur noch der Kodierer und ein nachgeschalteter Klassifikator benutzt. Diese Anordnung erbrachte bezüglich der Klassifikationsaufgabe bessere Resultate, da nicht mehr eindeutige und umkehrbare Repräsentationen notwenig sind. Trotz der Verbesserung gab es immer noch Probleme, die erst durch die Einführung weiterer Schichten im Kodierer bewältigt wurden.

Aus diesem Grunde wird in dieser Arbeit eine verallgemeinerte Form der LRAAM betrachtet. Diese besitzt im Gegensatz zur Standard-LRAAM eine variable Anzahl von Schichten,

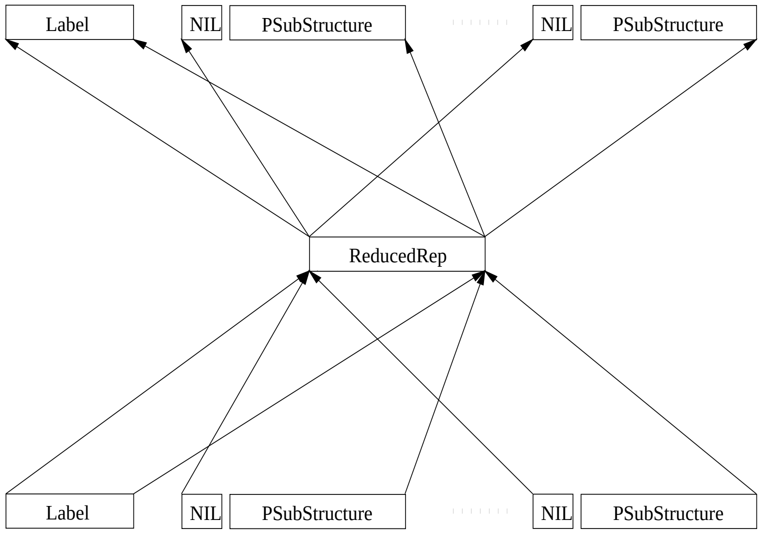


Abb. 3.9: Dieses Bild zeigt die Struktur einer LRAAM mit drei Schichten. Die NIL-Felder wurden aus dem Labelfeld herausgezogen und separat dargestellt.

wobei die mittelste Schicht die komprimierte Repräsentation liefert. Durch diese Erweiterung können Kodierer und Dekodierer nicht nur linear separable Probleme lösen, sondern sind Approximatoren für beliebige stetige Funktionen.

3.4.2 Formale Definition der LRAAM

Ausgehend von der Definition für ein allgemeines Neuronales Netz (siehe Abschnitt 3.3), soll die Struktur der LRAAM definiert werden.

Definition 3.8 (Struktur) Eine LRAAM:= (U, W, A, O, NET, ex) (Labeling RAAM) ist ein n-schichtiges Neuronales Netz (n ungerade, $n \geq 3$).

$U = \bigcup_{i=0}^{n-1} U_i$ ist die Menge der Einheiten, wobei die Teilmengen U_i paarweise disjunkt sind.

Drei dieser Mengen werden besonders bezeichnet.

$U_I ::= U_0$ ist die Menge der Eingabeeinheiten.

$U_O ::= U_{n-1}$ ist die Menge der Ausgabeeinheiten.

$U_R ::= U_{(n-1)/2}$ ist die Menge der Einheiten in der Repräsentationsschicht.

Die Eingabe- und Ausgabeschicht sind gleich strukturiert, und es gilt

$$U_I = U_{I,l} \cup U_{I,nil} \cup U_{I,p}$$

und

$$U_O = U_{O,l} \cup U_{O,nil} \cup U_{O,p}.$$

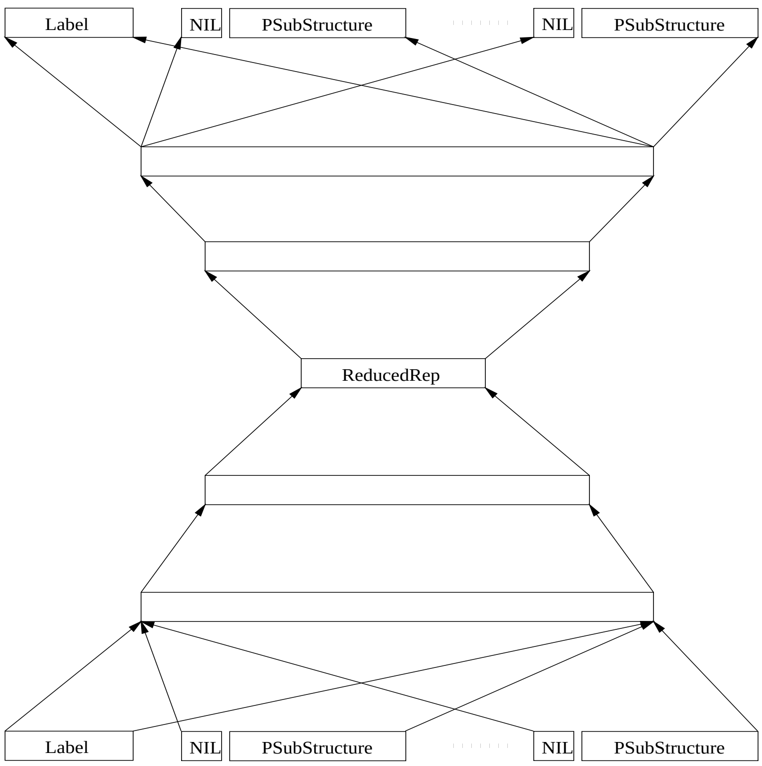


Abb. 3.10: Die verallgemeinerte LRAAM besitzt mehrere verdeckte Schichten, wobei die mittlere Schicht die reduzierte Repräsentation liefert.

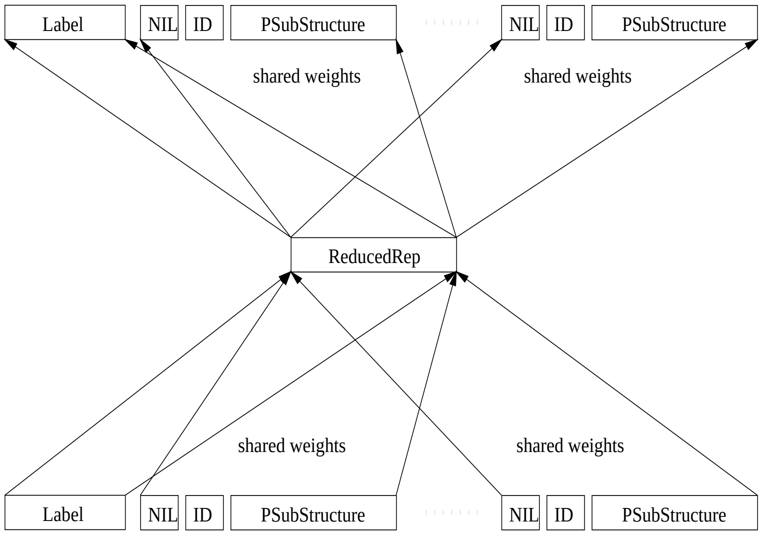


Abb. 3.11: Diese Abbildung zeigt eine Variante der LRAAM mit shared weights *innerhalb* der LRAAM. Für jede Teilrepräsention muß die gleiche Erkennungsaufgabe durchgeführt werden. Daher arbeiten die entsprechendenden „Teilnetze" mit derselben Wichtungsmatrix. Beim Anlernen werden diese Gewichte somit für alle Parameter gleich geändert. Damit das Netz in der Lage ist, die einzelnen Argumentpositionen zu unterscheiden, wird für jedes Argument ein zusätzlicher Identifikator eingeführt.

Die Einheiten in $U_{I,l}$ bilden das Labelfeld. Jene in $U_{I,nil}$ dienen als Markierung, ob die korrespondierenden Zeigereinheiten aus $U_{I,p}$ eine gültige Repräsentation enthalten. Es gilt weiterhin, daß

$$\begin{aligned} label\_size &:= |U_{I,l}| = |U_{O,l}|, \\ max\_branch &:= |U_{I,nil}| = |U_{O,nil}|, \\ ptr\_size \cdot max\_branch &= |U_{I,p}| = |U_{O,p}| \text{ und} \\ ptr\_size &= |U_R|. \end{aligned}$$

$$w_{uv} = \begin{cases} beliebig & : \quad u \in U_i \wedge v \in U_{i+1}, \text{ mit } 0 \leq i < n-1 \\ 0 & : \quad sonst \end{cases}$$

Die Aktivierungsfunktion sei, für alle $u \in U_i$, mit $1 < i \leq n-1$

$$a_u = \begin{cases} A_u(ex(u)) = ex(u) & : \quad u \in U_I \\ A_u(ex(u)) = ex(u) & : \quad \text{Dekoder aktiv und } u \in U_R \\ A_u(net_u) = f_u(net_u) & : \quad \text{sonst} \end{cases}$$

f ist die logistische Funktion, $f(x) = \frac{1}{1+e^{-\beta x}}$, wobei β die „Steilheit" der Funktion steuert.

Als Ausgabefunktion wird die Identitätsfunktion, $o_u = O_u(a_u) = a_u$ für alle $u \in U$, verwendet.

Die Netzeingabe wird bestimmt durch

$$net_v = \sum_{u \in U \setminus U_I} w_{uv} \cdot o_u - \theta_v.$$

$\theta_v \in \mathbb{R}$ ist der Schwellwert (Bias) der Einheit v.

■

3.4.3 Training

Ziel des Trainings der LRAAM ist es, Elemente einer symbolischen Domäne $\mathcal{S}$ auf sich selbst abzubilden. Durch diesen Prozeß entsteht in der LRAAM eine reduzierte Repräsentation der Elemente der symbolischen Domäne.

Iteratives Training

Es sei ein Term aus der symbolischen Domäne gegeben. Als Eingabe und Zielvorgaben werden zuerst die Konstantensymbole verwendet. Werden diese korrekt abgebildet, werden alle Teilterme, die die Konstantenzeichen als Argumente enthalten hinzugenommen. Im nächsten Schritt werden alle Teilterme, die die zuvor angelernten Teilterme enthalten angelernt. Dieser Prozeß wird solange fortgeführt, bis der gesamte Term angelernt ist. Zur Bestimmung der notwendigen Eingaben und Targetwerte, werden die erzeugten reduzierten Repräsentationen der zuvor durchgeführten Trainingsrunde benötigt. Innerhalb des Trainings müssen diese kontinuierlich aktualisiert werden, da sie sich verändern (siehe *moving target problem*).

Backpropagation Through Structure

Eine Möglichkeit, das Problem des veränderlichen Lernzieles zu umgehen, stellt der BPTS–Algorithmus dar. Hierbei wird der zu lernende Term als ganzes in einem einzigen feedforward–Netz angelernt. Da die zu nutzenden Abbildungen bei allen Teiltermen gleich sein sollen, ist es notwendig und möglich, die Gewichtsmatrix entsprechend der Struktur der LRAAM nur einmal zu halten, die Netzwerkstruktur aber baumartig, entsprechend des zu lernenden Terms, auszubilden. Auf das so entstandene Netz kann der normale Backpropagation–Algorithmus angewendet werden. Der Eingabevektor ist zusammengesetzt aus allen im Term vorkommenden Funktionszeichen[4] und der Repräsentation für die NIL-Zeiger. Diese sind im einfachsten Fall alles Null-Vektoren. Die Target-Vorgaben sind identisch mit der Eingabe.

Aus Gründen der Einfachheit soll angenommen werden, daß alle Terme in einem RLDAG $G = (V, E, \lambda)$ gespeichert sind. Die symbolische Domäne $\mathcal{S}$ enthalte nur Bäume. $t \in V$ sei ein Knoten von $s \in \mathcal{S}$. $C : \Sigma \to \mathbb{R}^n$ sei eine Kodierfunktion, die symbolische Marken auf numerische Kodes abbildet. $nil \in \mathbb{R}^m$ sei eine spezielle Repräsentation für den leeren DAG. Weiterhin gilt:

[4] Genauer müßte es heißen, „aus den Kodierungen der Funktionszeichen auf Aktivierungen der Neuronen“.

$o_i^{(l)}(t)$	Ausgabe von Neuron i in Layer l auf Rekursionsstufe t
$\theta_i^{(l)}$	Bias von Neuron i in Layer l
$w_{ij}^{(l+1)}$	Wichtung von Neuron i in Layer l zu Neuron j in Layer $l+1$
c	Anzahl der Units für Knotenlabel ($c = \|U_{I,l}\|$)
n	Anzahl der Schichten der LRAAM ($r = (n-1)/2$)
m	Anzahl Units im Repräsentationlayer ($m = \|U_R\|$)
k	maximaler Verzweigungsgrad
V_t	Menge der direkten Nachfolgerknoten des Knotens t

$$V_t = \{t_1, ..., t_d\} = \{v|(t,v) \in E\} \quad \text{mit } 0 \leq d \leq k$$

t_i	i-ter Nachfolgeknoten von t
$[\cdot]_i$	i-te Komponente des Vektors

Die LRAAM wird in die Teile Kodierer ($0 \leq l \leq r$) und Dekodierer ($r \leq l < n$) zerlegt.

Vorwärtsphase Kodierer

$$\begin{aligned}
o_j^{(l+1)}(t) &= f(net_j^{(l+1)}(t)) = f(\sum_i o_i^{(l)}(t) w_{ij}^{(l+1)} - \theta_j^{(l+1)}) \quad \text{für} \quad 0 \leq l < r \\
o_i^{(0)}(t) &= \begin{cases} [C(\lambda(t))]_i & : \quad 0 \leq i < c \\ o^{(r)}_{((i-c) \bmod m)}(t_{((i-c) \operatorname{div} m)}) & : \quad c \leq i < c + dm \\ [nil]_{((i-c) \bmod m)} & : \quad c + dm \leq i < c + km \end{cases}
\end{aligned}$$

Vorwärtsphase Dekodierer

$$\begin{aligned}
o_j^{(l+1)}(t) &= f(net_j^{(l+1)}(t)) = f(\sum_i o_i^{(l)}(t) w_{ij}^{(l+1)} - \theta_j^{(l+1)}) \quad \text{für} \quad r \leq l < n \\
o_i^{(r)}(t_x) &= o^{(n-1)}_{(c+(i+xm) \bmod m)}(t) \quad \text{für} \quad 1 \leq x < d
\end{aligned}$$

Rückwärtsphase Kodierer

$$\begin{aligned}
\delta_i^{(l)}(t) &= f'(net_i^{(l)}(t))(\sum_k \delta_k^{(l+1)}(t) w_{ik}^{(l+1)}) \quad \text{für} \quad 1 \leq l < r \\
\delta_i^{(r)}(t_x) &= \delta^{(0)}_{(c+(i+xm) \bmod m)}(t) \quad \text{und } t \text{ existiert} \\
\delta_i^{(r)}(t) &= \delta_i^{(r)}(t) \quad \text{und } t \text{ ist Wurzelknoten}
\end{aligned}$$

Rückwärtsphase Dekodierer

$$\delta_i^{(l)}(t) = f'(net_i^{(l)}(t))(\sum_k \delta_k^{(l+1)}(t) w_{ik}^{(l+1)}) \quad \text{für} \quad r \leq l < n$$

$$\delta_i^{(n-1)}(t) = \begin{cases} [C(\lambda(t))]_i - o_i^{(n-1)}(t) & : \quad 0 \leq i < n \\ \delta^{(r)}_{((i-c) \bmod m)}(t_{((i-c) \operatorname{div} m)}) & : \quad c \leq i < c + dm \\ [nil]_{((i-c) \bmod m)} - o_i^{(n-1)}(t) & : \quad c + dm \leq i < c + km \end{cases}$$

Gewichtsupdate Die Gewichtsänderungen werden über der gesamte Menge V aller Knoten $t \in V$ summiert.

$$\Delta w_{ij}^{(l)} = \eta \sum_{t \in V} o_i^{(l-1)}(t) \cdot \delta_j^{(l)}(t), \quad \Delta\theta_j^{(l)} = \eta \sum_{t \in V} \delta_j^{(l)}(t).$$

3.4.4 Eigenschaften der LRAAM

- Die Erweiterung des zugrundegelegten Alphabetes, insbesondere eine Vergrößerung der maximalen Verzweigung, ist nur sehr schwer möglich. Eine solche Veränderung zieht in jedem Fall eine Strukturveränderung und ein Neulernen nach sich.
- Die LRAAM erzeugt bezüglich der zu repräsentierenden Strukturen angepaßte Repräsentationen und Kompressions-/Dekompressionsfunktionen, die insbesondere auch die Auftrittswahrscheinlichkeiten der einzelnen Muster in der Trainingsmenge berücksichtigen.
- Die LRAAM muß angelernt werden. Folglich muß bei einem Datenaustausch auch die benutze LRAAM, gekennzeichnet durch ihre Struktur und Gewichtsmatrix, mit übertragen werden. Es sei denn, daß der Datenaustausch auf der symbolischen Ebene vonstatten geht.
- Das Training vorgegebener reduzierter Repräsentationen ist durch den Einsatz der Injektionsmethode möglich, erschwert den Trainingsprozeß aber zusätzlich.
- Da die Kompressions- und Dekompressionsfunktion erlernt wird, ist es möglich, davon zu sprechen, daß die LRAAM generalisieren kann, da nicht alle möglichen (symbolischen) Muster in der Trainingsmenge enthalten sind. Die Güte der Generalisierung ist abhängig vom Trainingsprozeß und der Trainingsmenge (siehe Auswertung der Testergebnisse).

3.5 Holografisch Reduzierte Repräsentation – HRR

Eine verteilte Repräsentationsform, beruhend auf einer assoziativen Operation, der Zirkularkonvolution, wird in der Dissertation von Toni Plate [Pla94] beschrieben. In dieser Arbeit zeigt er, das die Eigenschaften der Zirkularkonvolution geeignet sind, frameartige Strukturen in Vektoren fester Länge zu speichern.

3.5.1 Zirkularkonvolution

Die Zirkularkonvolution $\otimes$ kann als eine Kompression des äußeren Produktes zweier Vektoren betrachtet werden (siehe Abbildung 3.12). Der Ergebnisvektor einer Zirkularkonvolution zweier Vektoren der Länge n besitzt wiederum die Länge n. Diese Eigenschaft erlaubt es, die Zirkularkonvolution rekursiv auf Teilstrukturen anzuwenden, ohne die Größe der gewählten Repräsentation zu verändern.

Die Zirkularkonvolution kann als Multiplikationsoperator für Vektoren betrachtet werden. Sie besitzt ähnliche Eigenschaften, wie die Skalar- oder Matrixmultiplikation. Die Zirkularkonvolution ist

- kommutativ $\mathbf{x} \otimes \mathbf{y} = \mathbf{y} \otimes \mathbf{x}$,
- assoziativ $\mathbf{x} \otimes (\mathbf{y} \otimes \mathbf{z}) = (\mathbf{x} \otimes \mathbf{y}) \otimes \mathbf{z}$,
- bilinear $\mathbf{x} \otimes (\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha\mathbf{x} \otimes \mathbf{y} + \beta\mathbf{x} \otimes \mathbf{z}$.

Außerdem existiert

- ein Einselement $\mathbf{I} \otimes \mathbf{x} = \mathbf{x},\ \mathbf{I} = [1, 0, 0, \ldots]$,
- ein Nullelement $\mathbf{0} \otimes \mathbf{x} = \mathbf{0},\ \mathbf{x} + \mathbf{0} = \mathbf{x}$,
- für fast alle Vektoren $\mathbf{x}$ eine exakte Inverse $\mathbf{x}^{-1} \otimes \mathbf{x} = \mathbf{I}$,
- und eine approximative Inverse $\mathbf{x}^*$, $\mathbf{x}_i = \mathbf{x}^*_{-i}$ (Indizes modulo n).

Die Existenz der approximativen Inversen erlaubt eine einfache rechnerische Umsetzung der inversen Operation. Allerdings wird das mit einer Verschlechterung der Ergebnisqualität erkauft, und der Einsatz eines Clean-Up-Speichers wird umso wichtiger.

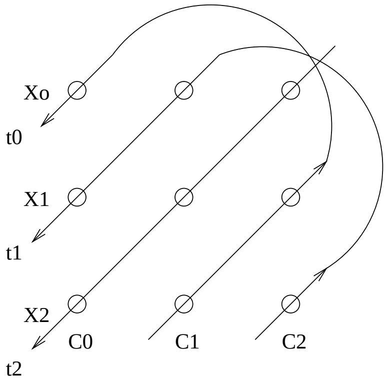
Die Zirkularkonvolution zweier n–dimensionaler Vektoren $\mathbf{x}$ und $\mathbf{y}$ ($\mathbf{z} = \mathbf{x} \otimes \mathbf{y}$) ist n–dimensional und besteht aus den Komponenten

$$z_i = \sum_{k=0}^{n-1} x_k y_{i-k} \quad i, k \in \mathbb{N}.$$

Sie kann mit der Zeitkomplexität $O(n \log n)$ durch Einsatz der Fast–Fourier–Transformation (FFT) berechnet werden.

Beispiel 4 Eine einfache Frame–Repräsentation für '*Spot bit Jane*' sieht wie folgt aus

$$\mathbf{P}_{bite} = \langle \mathbf{bite} + \mathbf{bite}_{agt} \otimes \mathbf{spot} + \mathbf{bite}_{obj} \otimes \mathbf{jane} \rangle$$



$$
\begin{aligned}
\mathbf{t} &= \mathbf{c} \otimes \mathbf{x} \\
t_0 &= c_0x_0 + c_2x_1 + c_1x_2 \\
t_1 &= c_1x_0 + c_0x_1 + c_2x_2 \\
t_2 &= c_2x_0 + c_1x_1 + c_0x_2 \\
t_j &= \sum_{k=0}^{n-1} c_k x_{j-k} \\
&\text{für } j = 0 \text{ bis } n-1 \\
&(\text{Indizes sind modulo } n)
\end{aligned}
$$

Abb. 3.12: Zirkularkonvolution ($\otimes$) repräsentiert durch das komprimierte äußere Produkt für $n = 3$. Die Kreise repräsentieren die Elemente des äußeren Produktes von $\mathbf{x}$ und $\mathbf{c}$. Die Elemente der Konvolution sind die Summen der Elemente des äußeren Produktes entlang der gezeichneten Diagonalen.

dabei ist **bite** ein Frame–Bezeichner, $\mathbf{bite}_{agt}$ und $\mathbf{bite}_{obj}$ sind Rollen und **jane** und **spot** sind Füller.

Die eckigen Klammern stehen für die Normalisierung des Vektors auf die Euklidische Länge 1.

Bindungen werden durch Addition nicht zerstört, es tritt jedoch eine Verfälschung durch zusätzliches Rauschen ein. Die Vektoren

$$\mathbf{bite}_{agt} \otimes \mathbf{spot} + \mathbf{bite}_{obj} \otimes \mathbf{jane}$$

und

$$\mathbf{bite}_{agt} \otimes \mathbf{jane} + \mathbf{bite}_{obj} \otimes \mathbf{spot}$$

sind verschieden voneinander.

Alle Frame–Komponenten im Beispiel sind n–dimensionale Vektoren, auch $\mathbf{P}_{bite}$, die Repräsentation für den vollständigen Frame. Dadurch wird es möglich, Frames als Füller für Rollen in anderen Frames zu benutzen.

Beispiel 5

$$
\begin{aligned}
\mathbf{P}_{bite} &= \langle \mathbf{bite} + \mathbf{bite}_{agt} \otimes \mathbf{spot} + \mathbf{bite}_{obj} \otimes \mathbf{jane} \rangle \\
\mathbf{P}_{flee} &= \langle \mathbf{flee} + \mathbf{flee}_{agt} \otimes \mathbf{jane} + \mathbf{flee}_{from} \otimes \mathbf{spot} \rangle \\
\mathbf{P}_{objects} &= \langle \mathbf{jane} + \mathbf{spot} \rangle \\
\mathbf{P}_{cause} &= \langle \mathbf{cause} + \mathbf{P}_{objects} + \mathbf{P}_{bite} + \mathbf{P}_{flee} + \mathbf{cause}_{antc} \otimes \mathbf{P}_{bite} + \mathbf{cause}_{cnsq} \otimes \mathbf{P}_{flee} \rangle
\end{aligned}
$$

Dekodierung der Bindungen

Die Existenz einer Inversen erlaubt die Dekodierung der Rolle–Füller–Bindungen. Hierfür wird die approximative Inverse verwendet, da sie einerseits einfach zu berechnen ist, denn es handelt sich ja nur um eine Permutation der Elemente, und andererseits numerisch stabil ist. Die exakte Inverse existiert zwar gewöhnlich, ist aber leider nicht immer numerisch stabil [Pla94].

Beispiel 6 Der Füller aus dem Vektor $\mathbf{bite}_{agt} \otimes \mathbf{spot}$ kann durch Konvolution mit $\mathbf{bite}^*_{agt}$ ermittelt werden. Es ergibt sich

$$\mathbf{bite}^*_{agt} \otimes \mathbf{bite}_{agt} \otimes \mathbf{spot} = \mathbf{spot}' \approx \mathbf{spot},$$

d.h. $\mathbf{spot}'$ ist ein zu **spot** ähnlicher Vektor, überlagert mit Rauschen. □

3.5.2 Eigenschaften der HRR

- HRRs besitzen die Eigenschaft des Graceful degradation.
- Die Speicherung der Objekte ist ähnlichkeitserhaltend, d.h. ähnliche Objekte werden durch ähnliche Vektoren repräsentiert.
- HRRs sind eine verteilte Repräsentation für frame-artige Strukturen und repräsentieren diese Objekte im quasikontinuierlichen Raum.
- Objekte, Frame-Labels und Rollen werden durch n-dimensionale, reellwertige Vektoren dargestellt.
- Instanziierte Frames werden als Summe (Überlagerung) von Rolle-Füller-Bindungen kodiert.
- Als Bindungsoperation wird eine zirkulare Faltungsoperation eingesetzt.
- Die Dimension der Vektoren bleibt erhalten, auch wenn geschachtelte Strukturen, z.B. Terme, gespeichert werden.
- Durch den Einsatz eines Clean-Up-Speichers sind HRRs von sich aus auch fehlertolerant.
- Genralisieren können HRRs nicht, da kein Training durchgeführt wird und somit auch keine Trainings- und Testmengen als Mustermengen existieren. Durch den Einsatz des Clean-Up-Speichers werden alle Veränderungen des Eingabemusters zum nächsten Muster im Clean-Up-Speicher korrigiert. Die HRR ist demzufolge nur fehlerkorrigierend.

3.6 Sind LRAAM und HRR gleichwertig?

Beide Repräsentationsformen haben zur Aufgabe, hierarchisch strukturierte Objekte in Vektoren fester Länge zu kodieren. Für diese Aufgabe werden eine Kompressionsfunktion,

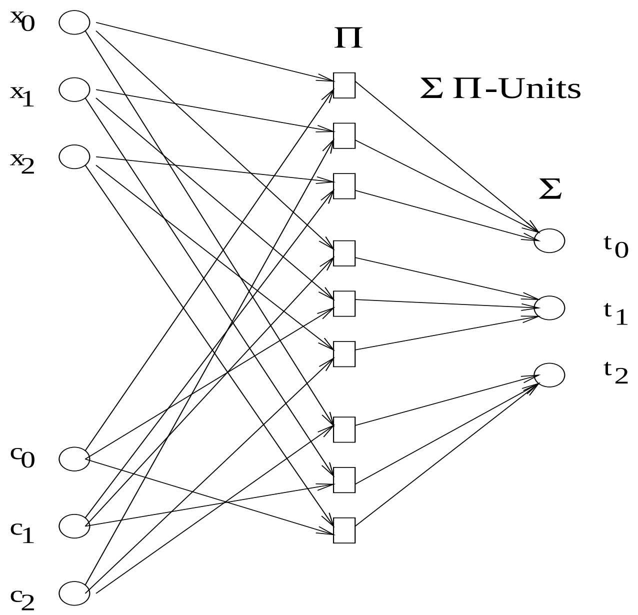


Abb. 3.13: Abbildung der Zirkularkonvolution $\otimes$ auf eine Neuronale Architektur mit Sigma-Pi-Units. Die Rechtecke führen die Multiplikation, die Kreise die anschließende Addition durch. Alle Wichtungen sind einheitlich eins.

die eine Menge von Vektoren auf einen einzelnen abbildet, und eine Dekompressionsfunktion benötigt. Bei der HRR ist genau diese Funktion durch die Zirkularkonvolution und die Vektoraddition gegeben. Die Funktion, die von der LRAAM benutzt wird, ist durch den Algorithmus nicht vorgegeben, sondern ergibt sich erst durch das Training und die benutzten Trainingsdaten.

Eine Annahme ist, daß die LRAAM eine zur Zirkularkonvolution äquivalente Funktion approximiert. Wenn dies der Fall ist, so ist anzunehmen, daß die LRAAM nicht besser als die HRR bei gleicher Aufgabenstellung ist. Andererseits kann die LRAAM die Kompressionsfunktion genau auf die präsentierten Trainingsdaten abstimmen und so möglicherweise bessere Ergebnisse bei der Kodierung und Dekodierung erzielen.

Die HRR läßt sich auf ein Neuronales Netz abbilden. Allerdings werden dafür Higher-Order-Netzwerke benötigt. Diese sind dadurch gekennzeichnet, daß Sigma-Pi-Units verwendet werden.

Eine Sigma-Pi-Unit (vom Grade n) berechnet ihren Netzeingabe nach der Vorschrift

$$net_u = \Sigma_k w_{ku} \Pi_n o_{v_n}.$$

Das Neuronales Netz zu einer Zirkularkonvolution ist in Bild 3.13 dargestellt. Von Vorteil ist immer noch, daß das Netz einfach konstruiert werden kann. Siegelmann zeigt in [Sie93], daß vom Standpunkt der Berechenbarkeit Higher-Order-Netzwerke gleichwertig zu mehrschichtigen Perceptronnetzwerken sind, denn beide Netze können beliebige stetige Funktionen approximieren. Somit also auch die durch das jeweils andere Netz berechnete

Funktion.

Gehen wir davon aus, daß die LRAAM die HRR approximiert, so tut sie das zumindest in einer effizienteren Weise, da sie die Auftrittswahrscheinlichkeiten der einzelnen Muster berücksichtigt und die Approximation natürlich auch von der präsentierten Trainingsmenge abhängt. Die Abbildung der HRR ist ein universaler Mechanismus, der keine Rücksicht auf die Struktur der vorliegenden Terme nimmt. Im Gegensatz dazu liefert die LRAAM einen auf die Domäne zugeschnittenen Kompressionsoperator. Dieser Vorteil wird jedoch mit dem Training erkauft.

4. Implementation

Ziel dieser Arbeit war es, die durch die verschiedenen Modelle vorgegebenen Algorithmen in Einklang zu bringen und einen einheitlichen Rahmen zum Vergleich der Modelle zu schaffen. Weiterhin sollte es möglich sein, neue Modelle zu implementieren und den gleichen Tests zu unterziehen.

Für die Realisierung dieser Anforderungen wurde eine Klassenbibliothek entworfen.

Den Kern der symbolischen Verarbeitung stellt die *Symboltabelle* mit ihren verschiedenen Methoden dar. Mit ihrer Hilfe ist es möglich, Funktionszeichen (im folgenden einfach als Symbole bezeichnet) zu verwalten. Es kann nach Symbolen gesucht werden, sie können eingefügt, gelöscht und einer reduzierten Repräsentation zugeordnet werden. Dies erlaubt, die Symboltabelle gleichzeitig als *Clean-Up-Memory* und autoassoziativen Speicher zu benutzen. Die Symboltabelle liefert demzufolge auch das Symbol zu einer gegebenen reduzierten Repräsentation, inklusive einem Ähnlichkeitswert. Als Ähnlichkeitsmaß wurde das Skalarprodukt über den normalisierten Vektoren gewählt.

Die Basisklasse für verteilte Speicher enthält die für Speicher elementaren Funktionen als abstrakte Methoden. Diese werden durch die erbenden Klassen entsprechend des gewählten Modells überschrieben.

Jeder Speicher, egal ob symbolisch oder subsymbolisch, besitzt mindestens die folgende Funktionalität:

- Ablegen von Informationen im Speicher (additiv oder überschreibend),
- Auslesen von Informationen aus dem Speicher (kopierend oder zerstörend).

Der eigentlich Speicher ist in unserem Fall ein reellwertiger Vektor der Länge n. Um in diesen Vektor die Terme speichern zu können, benötigen wir zusätzliche Operatoren. Dies sind ein *Kompositionsoperator* $C : \mathcal{S} \rightarrow \mathbb{R}^n$ und ein *Dekompositionsoperator* $D : \mathbb{R}^n \rightarrow \mathcal{S}$. In beiden untersuchten Modellen müssen diese Operatoren eine Kompression, bzw. Dekompression durchführen. Im Fall der HRR sind die Operatoren durch das Modell fest definiert, wogegen die LRAAM diese Operationen erst lernen muß.

Die Basisklasse für verteilte Speicher deklariert also nicht Zugriffsmethoden auf den Speicher, sondern nur eine Kompressions- und eine Dekompressionsmethode. Diese Methoden heißen `SS2RR()` und `RR2SS()`.

Unter Zuhilfenahme dieser beiden Methoden sind anschließend die Zugriffsfunktionen, wie Speicherinhalt schreiben oder lesen, zu implementieren. Da für die durchgeführten Untersuchungen die Ergebnisse der Kompression bzw. Dekompression ausreichend sind, wurde in dieser Arbeit auf weitere Zugriffsmethoden verzichtet. Diese können durch die Vererbungsmechanismen in abgeleiteten Klassen leicht hinzugefügt werden.

Für die Repräsentation eines Terms als symbolische Struktur, bzw. als reellwertigen Vektor, wurden eigene Klassen geschaffen. Die Klassen für die symbolische Repräsentation besitzen zusätzliche Funktionen zur Bewertung der repräsentierten Terme, wie mittlere und maximale Tiefe, mittlerer und maximaler Verzweigungsgrad, die Anzahl der Terminal- und Nichtterminalzeichen. Des weiteren ist ein Vergleich von Termen möglich, der als Ergebnis die Tiefe der Übereinstimmung liefert. Sind beide Termrepräsentationen gleich, ist das Ergebnis gleich der maximale Tiefe der Terme. Die so gewonnenen Daten werden für die Bewertung der Speicherkapazität benutzt. Eine erbende Klasse fügt die Funktionalität der zufälligen Erzeugung von Termen einer wählbaren maximalen Tiefe hinzu. Auf diese Weise wurde gleichzeitig der *Testmengengenerator* mit implementiert.

Von der Klasse `ReducedRepresentation` wurde eine weitere Klasse abgeleitet, die modellspezifisch für die HRR ist. In ihr wurde der Operator für die Zirkularkonvolution, die Methoden für die Initialisierung und Normierung der Vektoren implementiert.

Die Klassenbibliothek kann für die folgenden Aufgaben benutzt werden:

1. Repräsentation von symbolischer und subsymbolischer Information,
2. Operationen über den Repräsentationen entsprechend des gewählten Speichermodells,
3. Generatoren für symbolische Strukturen mit bestimmten Eigenschaften (Tiefe, Breite),
4. Generatoren für subsymbolische Repräsentationen als Basisvektoren für die HRR,
5. Transformation von symbolischen in subsymbolische Repräsentationen und umgekehrt und
6. Vergleich und Bewertung der symbolischen und subsymbolischen Repräsentationen.

Die Beschreibung der einzelnen Klassen und deren Deklaration befindet sich im Anhang A. Die folgenden Abschnitte beschreiben wichtige Auszüge und Implementationsdetails, die für die Arbeit von besonderem Interesse sind.

4.1 Die Symboltabelle

Wie im letzten Abschnitt beschrieben, ist die Symboltabelle der Kern der symbolischen Verarbeitung. Alle während des Programmlaufes definierten Symbole werden in der Symboltabelle verwaltet. In dieser Tabelle befinden sich neben den Symbole auch die assoziierten verteilten Repräsentationen, sofern diese bekannt sind. Dadurch ist es möglich,

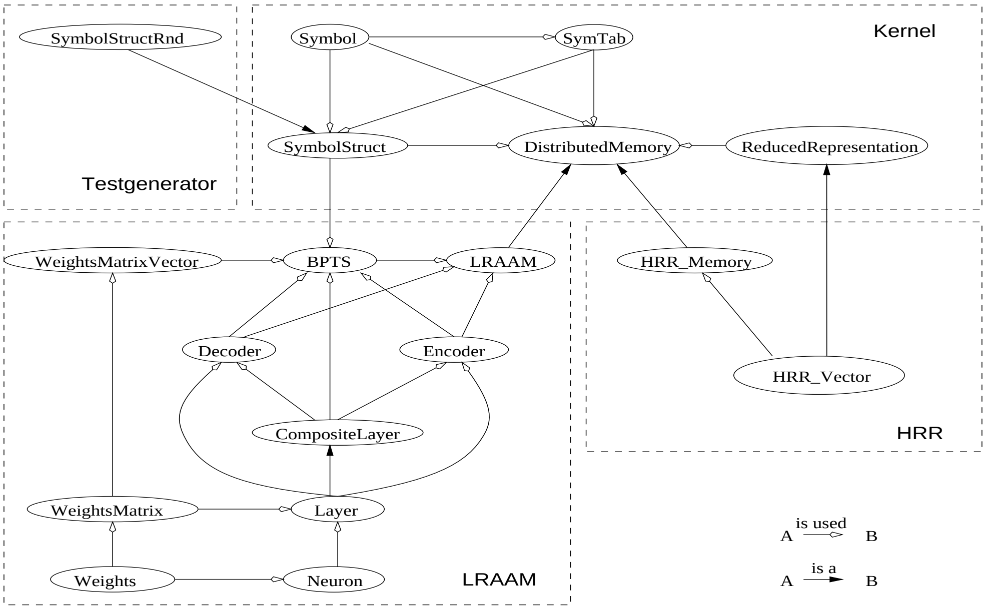


Abb. 4.1: Diese Abbildung stellt die Nutzungs- und Vererbungsbeziehungen zwischen den einzelnen Klassen dar. Die einzelnen Module wurden zusätzlich durch einen Rahmen gekennzeichnet. Je nach Aufgabenstellung können, bis auf den Kernel, einzelne Module weggelassen werden.

den Auffrischungsspeicher (cleanup-memory) mit in die Symboltabelle zu integrieren. Die Symboltabelle fungiert dabei als autoassoziativer Speicher und liefert zu einer gegebenen gestörten Repräsentation die am besten passende Repräsentation und deren Ähnlichkeit (Abstand). Als Ähnlichkeitsmaß wird dabei das Skalar-Produkt benutzt. Die Symboltabelle besitzt weiterhin entsprechende Zugriffs-, Manipulations- und Suchfunktionen.

4.2 Symbolische Repräsentation

Die Repräsentation von Termen wird durch die Klasse `SymbolStruct` realisiert. Diese Klasse verwaltet Terme als Baumstrukturen und verwendet dafür spezielle Knotenstrukturen.

```
struct SymStructNode
{
  int             sym_index;     /* symbol index from symtab */
  int             arity;         /* copy of symbol arity */
  PSymStructNode* child_symbols; /* pointer to an array with pointer */
  PSymStructNode  parent_symbol;
  double          error;         /* contains decode distance */
  PEncoder        encoder;       /* encoder module */
  PDecoder        decoder;       /* decoder module */
};
```

Je Knoten existieren Verweise auf einen Kodierer- und einen Decodierer. Diese Felder können für Lernverfahren benutzt werden, die die Baumstruktur nutzen, wie der BPTS–Algorithmus.

Eine Instanz kann mit einer Termbeschreibung erzeugt, kann die Repräsentation ändern und mit anderen Instanzen verglichen werden. Die vielfältigen Erzeugungsmethoden werden durch eine Anzahl verschiedener Konstruktoren realisiert.

```
SymbolStruct();
SymbolStruct(String composed_symbol_string);
SymbolStruct(const SymbolStruct &c);
```

Der erste Konstruktor liefert eine leere Symbolstruktur, quasi einen leeren Term. Der zweite initialisiert die Baumstruktur aus einem String. Dieser String ist eine linksassoziative Darstellung des entsprechenden Terms. „`f g a b`“ entspricht also `f(g(a),b)`, wenn die Funktionszeichen f/2, g/1, a/0 und b/0 angenommen werden. Dieses Format wird auch für die Ein- und Ausgabe von Termen in Dateien genutzt. Der letzte Kontruktor kopiert den Term von einer anderen Symbolstruktur.

Von besonderer Bedeutung sind die sogenannten Iteratoren. Mit ihnen ist es möglich, die Baumstruktur eines Terms in verschiedener Art und Weise zu durchmustern. Die bisher implementierte Iteratorklasse realisiert einen Tiefen- oder Breitendurchlauf. Es ist möglich auf eine Termstruktur mehrere Iteratoren anzuwenden.

4.3 Reduzierte Repräsentation für die HRR

Für die Nutzung der Klasse `ReducedRepresentation` in der HRR ist eine Erweiterung der Funktionalität erforderlich. Es müssen folgende zusätzliche Operatoren über dem reellwertigen Vektor definiert werden:

- Initialisierung mit zufälligen und unabhängig normalverteilten reellen Zahlen mit dem Mittelwert null und der Varianz $1/n$, $N(0, 1/n)$, wobei n die Vektorlänge ist.
- Normalisierung des Vektors auf die Euklidische Länge 1.
- Zirkularkonvolution $\otimes$ als binärer Operator über zwei Vektoren.

Zur Generierung der Zufallszahlen wurde auf die Klassen `Random` und `ACG` aus der GNU-libg++ zurückgegriffen. Die Implementierung der Zirkularkonvolution erfolgte unter Anwendung der FFT, aus einer Bibliothek von Peter Valkenburg (valke@cs.vu.nl). Die genutzte FFT ist nicht auf Vektorlängen der Potenz von zwei beschrängt.

4.4 Die Klasse HRR_Memory

Die durch die Klasse `HRR_Memory` implementierten Methoden sind `RR2SS()` und `SS2RR()`. Der genutzte Algorithmus ist in Alg. 4.1 zu sehen. Die reduzierte Repräsentation wird rekursiv aus der symbolischen erzeugt. Hierbei werden die schon in der Symboltabelle existierenden Repräsentationen für die Rollen und Funktionszeichen (Füller) benutzt. Diese Symbole und Repräsentationen werden einmalig bei der Erzeugung der ersten Instanz von `HRR_Memory` generiert und in die Symboltabelle eingetragen.

```
TERM2HRR(Term)
HRR = F ⊗ Term.Functor
if Arity(Term.Functor) > 0
   then for i ← 1 to Arity(Term.Functor)
           do HRR = HRR + P_i ⊗ TERM2HRR(Term.Arg[i])
HRR = ⟨HRR⟩
return HRR
```

Algorithmus 4.1: HRR-Komposition aus einem Term, bzw. seinen Teiltermen

Die Methode `RR2SS()` realisiert die notwendige Umkehrfunktion. Als Eingabegröße liegt eine reduzierte Repräsentation vor. Zuerst wird das Funktionszeichen dekodiert, anhand dessen aus der Symboltabelle die Stelligkeit ermittelt werden kann. Anschließend werden die einzelnen Argumentvektoren dekodiert und diese rekursiv ausgewertet. Die Zwischenergebnisse werden *nicht* mit in die Symboltabelle mit aufgenommen, obwohl dies die Speicherkapazität indirekt natürlich erhöhen würde, da damit bei der Dekodierung die Fehler sich nicht aufsummieren, sondern immer wieder auf die ungestörtere Repräsentation aus der Tabelle zurückgegriffen werden könnte.

Der Algorithmus ist in Alg. 4.2 zu sehen.

```
HRR2TERM(HRR)
Term.Functor = GetSymbol(CleanUp(F^-1 ⊗ HRR))
if Arity(Term.Functor) > 0
   then for i ← 1 to Arity(Term.Functor)
           do Term.Arg[i] = HRR2TERM(P_i^-1 ⊗ HRR)
return Term
```

Algorithmus 4.2: HRR-Dekomposition zu einem Term

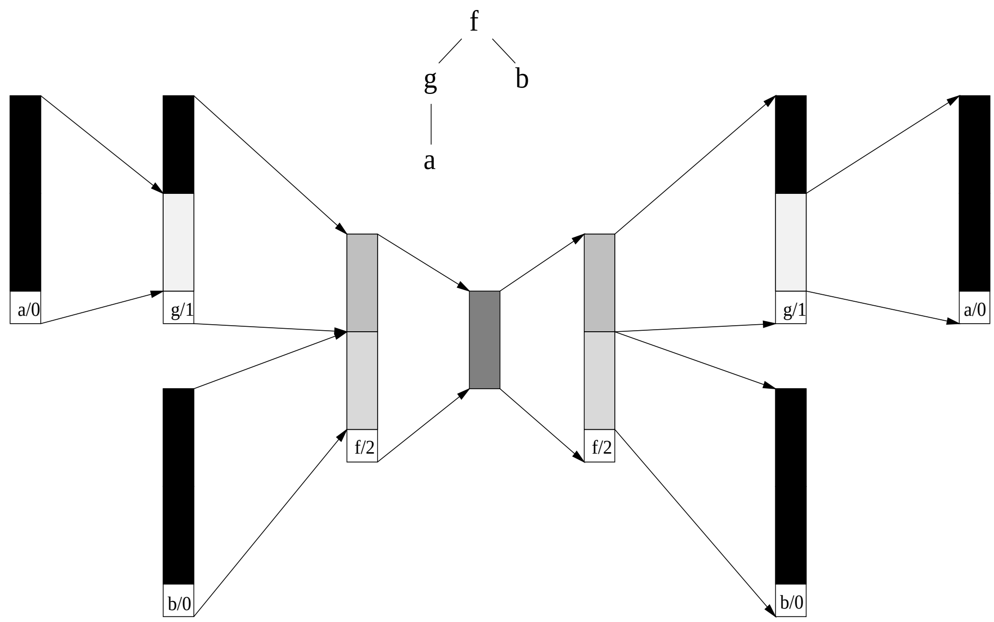


Abb. 4.2: Entfaltete LRAAM für den BPTS–Algorithmus. Die Repräsentationslayer der Kodierer sind gleichzeitig Teil der Eingabelayer. Bei den Dekodierern sind Teile der Ausgabelayer gleichzeitig die Repräsentationslayer der nachfolgenden Stufe. Das gesamte Netz kann in dieser Form mit dem normalen Backpropagation–Algorithmus trainiert werden, wenn shared Weights zum Einsatz kommen.

4.5 Die Klasse `LRAAM`

Bei der LRAAM ist sind die Operationen für die Kompression und Dekompression nicht durch das Modell vorgegeben. Die Operationen müssen erst angelernt werden. Im Standardmodell von Sperduti wird dafür ein leicht modifizierter Backpropagationalgorithmus verwendet. Die Änderungen beziehen sich auf die Anpassung der Steilheit der Transferfunktion während des Lernens. Dadurch soll das Netz eine einmal gefundene richtige Labelkodierung nicht mehr weiter verändern, die Abbildung für die Pointer aber weiter verbessern.

Küchler und Goller führten in ihrer Arbeit zur Termklassifikation eine weitere Variante des Backprop-Algorithmus, *Backpropagation Through Structure* (BPTS), ein. Ähnlich wie bei BPTT wird hier ebenfalls eine Entfaltung durchgeführt, jedoch diesmal nicht in der Zeit, sondern in der zu trainierenden Struktur, dem Term. Wie leicht einzusehen ist, reichen hier einfache Stacks nicht aus, da die entstehende Struktur baumartig und nicht linear ist. In Abb. 4.5 ist die Entfaltung der LRAAM beispielhaft dargestellt.

Die LRAAM teilt sich strukturell in zwei Komponenten, den Kodierer und den Dekodierer. Beide bestehen wiederum aus Schichten von Neuronen, wobei die Eingabeschicht des Kodierers teilweise die Repräsentationsschichten der vorgeschalteten Kodierer enthalten.

Wie beim BPTT, muß auch beim BPTS garantiert werden, daß die einzelnen Teilnetze mit den gleichen Wichtungen arbeiten, da sie eigentlich nur einmal vorhanden sind. Die

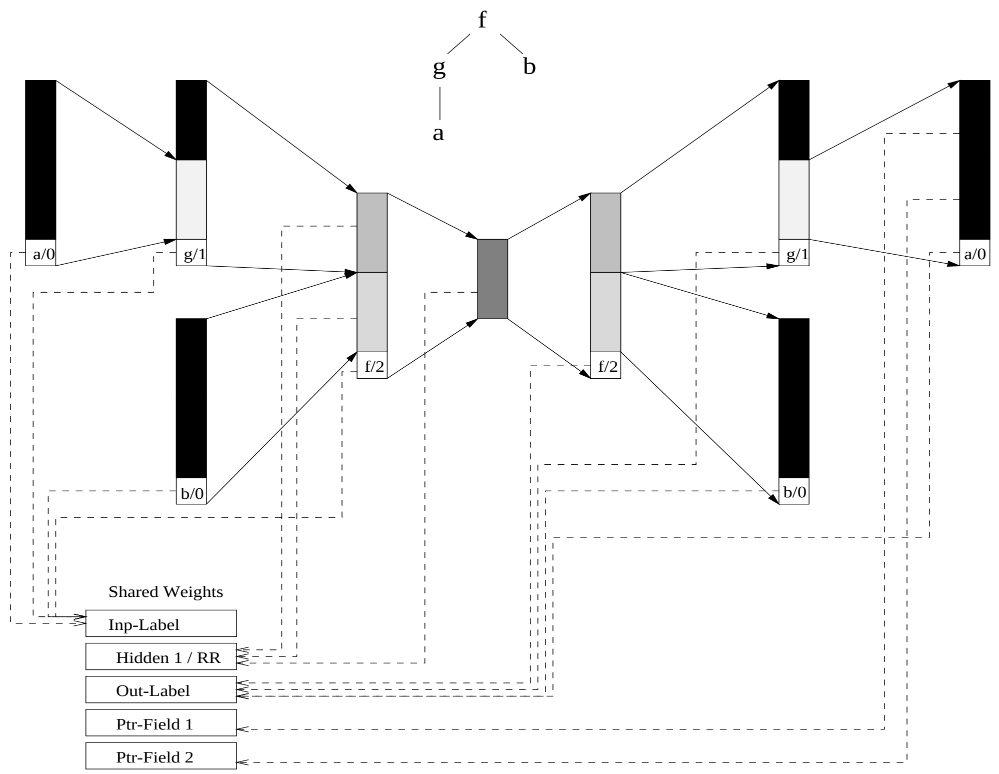


Abb. 4.3: Entfaltete LRAAM mit eingezeichneten Verweisen auf die verwendeten Wichtungsmattrizen.

in der Klasse `LRAAM` verwendete Struktur ist in Bild 4.3 zu sehen. Jedes Neuron besitzt einen Zeiger auf die Wichtungsmatrix. Neuronen, die strukturell gleich angeordnet sind, greifen auch auf die gleichen Wichtungen zu.

Betrachtet man den Unterschied zwischen BPTT und Truncated BPTT, so müßte der verwendete Algorithmus als Truncated BPTS bezeichnet werden. Die Unterscheidung ist aber unerheblich, da es sich bei Termen in jedem Fall um endliche Gebilde handelt, und die Größe der entfalteten LRAAM sich nur nach dem zu lernenden Term richtet.

Einführend soll der Trainingsprozeß an einem Term erläutert werden. Für einen gegebenen Term wird eine Instanz der Klasse `BPTS` erzeugt. Diese baut die Baumstruktur entsprechend der gewählten LRAAM-Struktur und des zu lernenden Terms auf. Die Eingabe- und Targetinformationen können direkt aus der Termbeschreibung entnommen werden. Die Initialisierung der NIL-Pointer ist in der Methode `Encoder::SetLabel(value,arity)` implementiert. In der vorliegenden Programmversion werden alle NIL-Pointer einheitlich auf Null gesetzt. Über die entsprechende Methode `Decoder::SetTarget(value, arity)` im Dekoder werden die Targetwerte der Ausgabeschicht festgelegt.

Nach der Initialisierung durchläuft die LRAAM den Trainingszyklus. Dieser propagiert die Eingabeinformation entsprechend der Struktur durch das Netz. Anschließend kann in jedem Ausgabeneuron, welches einen Targetwert besitzt, der Fehler durch Vergleich des Targetwertes mit der Ausgabe ermittelt werden. Dieser Fehler wird nun entsprechend des normalen Backpropagationalgorithmus und der vorliegenden Netzstruktur rückwärts durch das Netz propagiert. Die Wichtungsänderungen werden in einer der Wichtungsmatrix strukturell identischen Matrix aufsummiert. Erst nach erfolgter, vollständiger Rückpropagierung werden die aufsummierten Deltawerte zur Wichtungsmatrix hinzuaddiert. Damit ist ein Trainingszyklus abgeschlossen. Diese Zyklen werden nun so lange wiederholt, bis eine maximale Anzahl erreicht oder der an den Ausgabeneuronen gemachte Fehler unter eine gegebene Schranke gesunken ist.

Der ursprüngliche Lernalgorithmus der LRAAM ging davon aus, daß erst die Subterme und dann schrittweise komplexere Terme angelernt werden, bis der gesamte (komplexe) Term angelernt wurde. Der Vorteil dieses Verfahrens ist, daß mit jedem Term auch alle in ihm vorkommenden Subterme mit trainiert werden.

Der BPTS-Algorithmus liefert i.a. kein solches Ergebnis. Hierbei wird ausschließlich der gesamte Term angelernt. Das Problem liegt darin begründet, daß die Repräsentationen der inneren Kodier- und Dekodiernetzwerke nicht zwangsläufig gleich sein müssen. Es würde eine einfache Permutation der Elemente der Repräsenation ausreichen, um diesen Effekt zu erreichen. Um doch noch alle Teilterme mit anzulernen, wurde eine Erweiterung des BPTS-Algorithmus vorgenommen. In Anlehnung an die *Injektionsmethode* von Stolcke und Wu wird die erzeugte Repäsentation eines Teilterms im Kodierer als Target in die zugehörige Repräsentationsschicht des Dekodierers übertragen. Dadurch wird abgesichert, daß der Dekodierer wirklich mit der richtigen (d.h. zum Kodierer gleichen) Repräsentation trainiert wird. Andererseits kommt hierdurch wieder das *moving–target*–Problem ins Spiel.

Eine andere Möglichkeit zum Anlernen aller Teilterme ist, diese einfach mit in die Trainingsmenge aufzunehmen, was dazu führt, eine Menge von Termen anzulernen. Realisiert wird es dadurch, daß für jeden Term eine Instanz der Klasse BPTS erzeugt wird. Da alle Instanzen über der gleichen Wichtungsmatrix arbeiten, kann, je nachdem wann das Update der Wichtungsmatrix erfolgt, ein Online- oder Batch-Training durchgeführt werden. Die Speicherkomplexität ist, bezogen auf die Anzahl der Knoten in den Termstrukturen, linear, da die Wichtungsmatrizen nur genau einmal existieren.

Ist die LRAAM erst einmal angelernt, können die Kompressions- und Dekompressionsmethoden ähnlich wie bei der HRR benutzt werden. Der Kompressionsalgorithmus ist in Alg.4.3 zu sehen, der Dekompressionsalgorithmus in Alg.4.4.

Bei der Dekompression kann es passieren, daß der Dekoder eine falsche Stelligkeit ausgibt. Da anhand der Daten der Symboltabelle aber bekannt ist, welches Funktionszeichen welche Stelligkeit besitzt, kann dieser Fehler korrigiert werden. In der Implementation der LRAAM wird von den Daten in der Symboltabelle Gebrauch gemacht. Die dekodierte Stelligkeit wird aber trotzdem ermittelt und mit dem Wert in der Symboltabelle verglichen. Es ist möglich, sich beide Werte ausgeben zu lassen.

```
TERM2RR(Term)
Encoder.Create()
Encoder.SetLabel(Term.Functor)
Encoder.SetNilPtr(Arity(Term.Functor))
if Arity(Term.Functor) > 0
  then for i ← 1 to Arity(Term.Functor)
         do Encoder.SetPtr(TER2RR(Term.Arg[i]))
return Encoder.Process()
```

Algorithmus 4.3: Kompression eines Terms mit der LRAAM

```
RR2TERM(RR)
Decoder.Create()
Decoder.SetRR(RR)
Decoder.Process()
Term.Functor = Decoder.GetLabel()
if Arity(Term.Functor) > 0
  then for i ← 1 to Arity(Term.Functor)
         do Term.Arg[i] = RR2TERM(Decoder.GetPtr(i))
return Term
```

Algorithmus 4.4: Dekodierung mit der LRAAM

5. Simulation und Vergleich der Ergebnisse

5.1 Testmengen

Für die nachfolgend beschriebenen Experimente wurde der Testmengengenerator (siehe Seite 41) verwendet, um Terme mit den maximalen Tiefen von 2, 3, 4, 6, 8, 10 und 15 zu erzeugen. Die Testdatensätze werden mit dem Kürzel d<Tiefe> unterschieden.

Aus den Testmengen wurden alle Duplikate entfernt, so daß jeder Term in der jeweiligen Testmenge auch nur einmal enhalten ist. Ein Term kann nicht in zwei unterschiedlichen Testmengen enthalten sein, da die Testmengen Klassen bilden.

Die Menge der verwendeten Funktionszeichen war $\{a/0, b/0, c/0, f/2, g/1, h/3\}$. Es wurden keine Funktionszeichen mit einer Stelligkeit größer drei gewählt, da die Rechenzeiten besonders bei tieferen Strukturen den verfügbaren Zeitrahmen gesprengt hätten. Diese Experimente müssen einem späteren Zeitpunkt vorbehalten bleiben. Außerdem ist aus [Pla94] bekannt, daß insbesondere die HRR bei flachen Strukturen brauchbarere Ergebnisse liefert, als bei tiefen.

Jede Testmusterdatei enthält bis zu 10.000 Terme.

Ziel der Untersuchungen sind Aussagen über

1. die *brittleness* der verwendeten Repräsentation,
2. die Generalisierungsfähigkeit und
3. die Eignung als Repräsentation von Termen in prädikatenlogischen Programmen.

5.2 Untersuchungen an der HRR

In den Experimenten mit der HRR wurde vordergründig die Speicherkapazität untersucht. Plate gibt in [Pla94, Anhang E] eine Näherungsformel für n Symbole eines gegeben Alphabets an, die unabhängig und zufällig ausgewählt und paarweise $\otimes$-verknüpft und summiert werden.

- n sei die Dimension der verwendeten Vektoren. Die Vektorelemente sind zufällig und unabhängig normalverteilt; N(0,1/n).
- E sei eine Menge, bestehend aus m Vektoren mit obiger Eigenschaft.
- t sei ein Ergebnisvektor, der eine *nicht normalisierte* Überlagerung von k verschiedenen und ungeordneten Vektoren aus E ist. Außerdem gilt, daß zwei Vektoren in einem Paar und ebenso Vektoren in unterschiedlichen Paaren verschieden sein müssen.

$$t = \Sigma_{i=1}^{k} x_i \otimes y_j, \quad x_i, y_i \in E, \ \forall i, j x_i \neq y_i, \ \forall i \neq j x_i \neq x_j, \ \forall i \neq j y_i \neq y_j$$

- $P(AllCorrect)$ ist die Wahrscheinlichkeit, richtig zu entscheiden, welche Terme im Ergebnisvektor t enthalten sind und welche nicht; $q = 1 - P(AllCorrect)$.

Plate gibt die folgende numerisch ermittelte Näherung an:

$$n = 4.5(k + 0.7) \ln \frac{m}{30q^4}$$

wobei k im Bereich (2...14), m im Bereich (10^2...10^{10}) und q im Bereich von (10^{-2}...10^{-10}) angenommen wurde.

Es ist leicht einzusehen, daß die einzigen Störungen, die auftreten, durch die additive Überlagerung und die einmalige Faltung entstehen. Die benutzten Basisvektoren sind alle im Clean-Up-Memory enthalten und können demzufolge auch rekonstruiert werden.

Die von Plate aufgeführte Kapazitätsangabe ist für die Anwendung der HRR als Termrepräsentationssystem unbrauchbar, da die gemachten Einschränkungen der Repräsentation von Termen widersprechen. Erstens sind Terme tiefe Strukturen, d.h. es müssen Repräsentationen verschachtelt werden. Zweitens geht er davon aus, daß keine Funktionszeichen in einem Term mehrfach vorkommen. Diese Annahme ist unrealistisch.

Die in den Tabellen 5.1 bis 5.7 aufgeführten Ergebnisse wurden mit zufällig generierten Termen ermittelt. Die Alphabetgröße ist mit sechs Funktionszeichen relativ klein. Die Folge ist, daß gerade bei größeren Termen viele Funktionszeichen mehrfach vorkommen. Die Auswahlswahrscheinlichkeiten der Funktionszeichen sind, bis auf die Blattknoten in der maximalen Tiefe, gleichverteilt.

Auswertung der Ergebnisse

Die Abbildung 5.1 zeigt prozentual die vollständig reproduzierten Terme, bezogen auf die Größe der HRR. Es wurde für jeden Testdatensatz eine eigene Kurve gezeichnet. Weiterhin ist in dieser Graphik zu sehen, daß die Kurven flacher und sich der 50%-Punkt zu immer größeren Vektordimensionen bei wachsender Termtiefe verschiebt. Plates Aussage, daß die HRR sehr gut skalierbar ist, kann für den Fall von tiefen Strukturen, so wie sie hier vorliegen, nicht bestätigt werden. Die Vektordimension steigt exponentiell mit der Tiefe der zu kodierenden Strukturen, bei gleicher Reproduktionswahrscheinlichkeit.

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
4	0	0	0.0512821	NaN	2
8	1	2.5641	0.684211	2	2
16	1	2.5641	0.578947	2	2
32	10	25.641	0.517241	2	2
64	27	69.2308	1	2	2
128	37	94.8718	1	2	2
256	37	94.8718	1	2	2
512	39	100	NaN	2	NaN
1024	39	100	NaN	2	NaN
2048	39	100	NaN	2	NaN
4096	39	100	NaN	2	NaN
8192	39	100	NaN	2	NaN

Tabelle 5.1: HRR (d2,39)

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
4	0	0	0.0891473	NaN	2.71086
8	0	0	0.501259	NaN	2.71337
16	0	0	0.735516	NaN	2.71337
32	6	0.377834	0.726296	2.72222	2.71333
64	66	4.15617	1.00723	2.64015	2.71654
128	232	14.6096	1.57006	2.64648	2.72481
256	349	21.9773	1.36885	2.6733	2.72465
512	996	62.7204	1.62669	2.66232	2.79925
1024	1339	84.3199	1.97189	2.68404	2.87106
2048	1379	86.8388	1.98565	2.68449	2.90388
4096	1382	87.0277	2	2.68479	2.90507
8192	1383	87.0907	2	2.6849	2.90542

Tabelle 5.2: HRR (d3,1588)

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
128	157	2.52533	1.65363	3.23121	3.25576
256	213	3.42609	1.70736	3.10642	3.26042
512	954	15.345	2.07619	3.11293	3.28092
1024	2891	46.5015	2.27661	3.20651	3.29741
2048	3912	62.9242	2.16486	3.19066	3.36457
4096	4711	75.7761	2.21248	3.2116	3.39135
8192	4933	79.347	2.2243	3.21764	3.39923

Tabelle 5.3: HRR (d4,6217)

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
512	0	0	2.01358	NaN	4.38143
1024	285	2.88812	2.60816	4.48526	4.37834
2048	211	2.13822	2.41493	4.25344	4.38422
4096	598	6.05999	2.94391	4.1128	4.39876
8192	1347	13.6502	3.0372	4.11739	4.42317

Tabelle 5.4: HRR (d6,9868)

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
512	0	0	1.9487	NaN	5.44677
1024	6	0.06	2.54453	6.09722	5.44638
2048	0	0	2.4237	NaN	5.44677
4096	5	0.05	2.807	4.30714	5.44734
8192	2	0.02	2.81746	4.71429	5.44692

Tabelle 5.5: HRR (d8,10000)

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
512	0	0	1.9179	NaN	6.51433
1024	0	0	2.3617	NaN	6.51433
2048	0	0	2.3362	NaN	6.51433
4096	0	0	2.6454	NaN	6.51433
8192	0	0	2.7072	NaN	6.51433

Tabelle 5.6: HRR (d10,10000)

Größe	# OK	% OK	Fehler in Tiefe	Tiefe Term (OK)	Tiefe Term (NOK)
512	0	0	1.7594	NaN	9.12636
1024	0	0	2.2469	NaN	9.12636
2048	0	0	2.2205	NaN	9.12636
4096	0	0	2.61574	NaN	9.23839
8192	0	0	2.53091	NaN	9.17618

Tabelle 5.7: HRR (d15,10000)

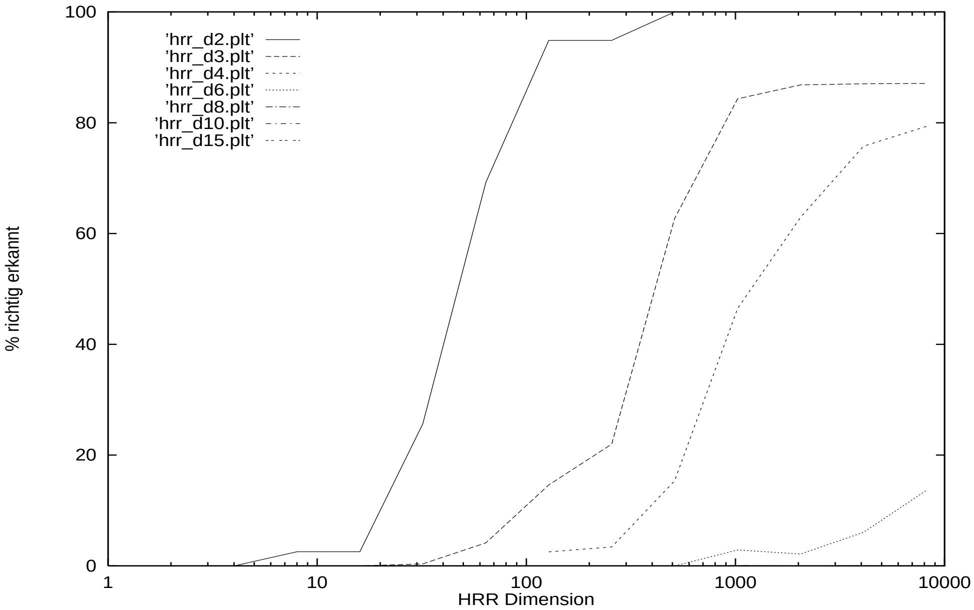


Abb. 5.1: Die Kurven zeigen die vollständige Erkennung der Terme für die verschiedenen Testdatensätze, mit Termen der Tiefe zwei bis 15. Es ist gut zu erkennen, daß die Wiedergabefähigkeit für tiefere Terme sehr stark nachläßt.

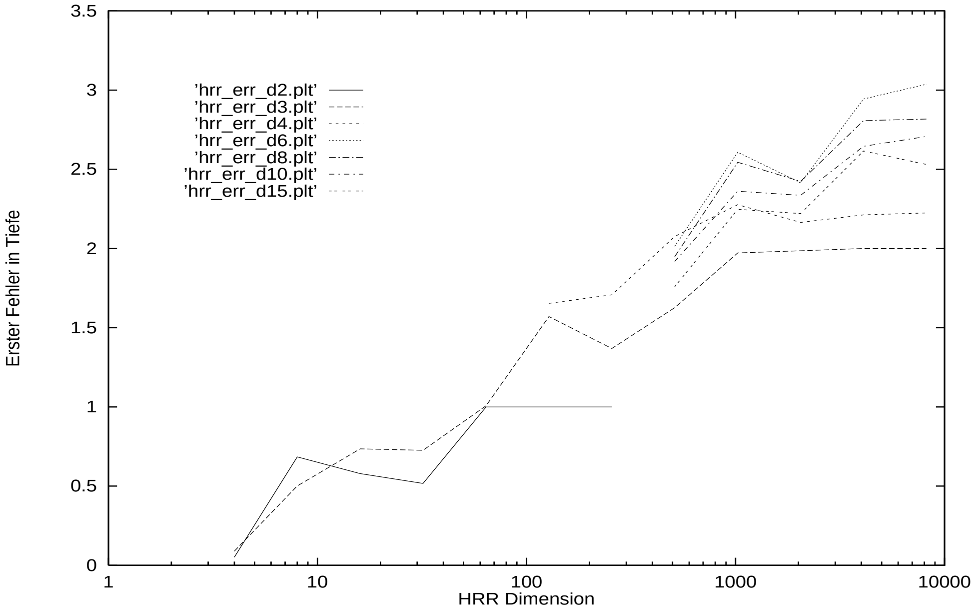


Abb. 5.2: Diese Graphik zeigt die mittlere Tiefe des ersten aufgetretenen Dekodierfehlers für die einzelnen Testdatensäte (max. Tiefen) und für die einzelnen HRR Größen. Mit zunehmnender Größe der HRR wächst auch die Tiefe der korrekten Reproduktion, obwohl die Kapazität der HRR eine vollständige Reproduktion nicht gewährleistet.

Die Abbildung 5.2 stellt die Tiefe des ersten aufgetreten Fehlers bezogen auf die Vektordimension dar. Hierbei handelt es sich um die Mittelwerte des jeweiligen Testdatensatzes. Die Kurve für die Terme der Tiefe zwei endet bei der Vektordimension von 512, da danach alle Terme korrekt reproduziert wurden. Die anderen Kurven beginnen erst bei der Dimension 512, da die Tests erst ab dieser Dimension begannen. Zu erkennen ist ebenfalls wieder der exponentielle Charakter der Reproduktionstiefe bezogen auf die Vektordimension. Es sei nochmals darauf hingewiesen, daß die hier eingehenden Terme alle nicht vollständig reproduziert wurden.

Die Schlußfolgerung ist, daß die HRR eine stabile Repräsentation liefert und bei Kapazitätsüberschreitung nicht die gesamte Repräsentation zerstört wird. Durch die Art der Kodierung der Baumstruktur bleiben die äußeren Baumteile erkennbar, das heißt eine Störung breitet sich von den Blättern zur Wurzel aus. Wird das bei Repräsentation von Termen berücksichtigt, so lassen sich auch tiefere Terme speichern, indem jeweils in der Stufe k ein Clean-Up-Speicher benutzt wird.

Generalisierungsfähigkeit und deren Test ist an eine Testmenge gebunden, die dem System zuvor nicht präsentiert wurde. Generalisierungsfähigkeit bedeutet Verallgemeinerung aus den bisherigen Erkenntnissen. Da die Abbildungsvorschrift der HRR vorgegeben ist, kann von „bisher nicht präsentierten Testmustern“ nicht gesprochen werden. Die Ähnlichkeit zwischen zwei Repräsentationen ist außerdem davon abhängig, welche Basisvektoren zur Kodierung der elementaren Objekte benutzt werden. Werden hierbei Ähnlichkeiten induziert, z.B. durch das Überlagern eines Vektors mit verschiedenen anderen Objektvektoren, um diese einander ähnlich zu machen, so treten diese Ähnlichkeiten auch in allen anderen Repräsentationen auf, wo diese Basisvektoren einbezogen wurden.

Der Einsatz eines Clean-Up-Speichers auf jeder Ebene zur Ermittlung der Funktionszeichen wirkt fehlerkorrigierend, wodurch die Generalisierungsfähigkeit durch die HRR selbst zunichte gemacht wird.

5.3 Untersuchungen an der LRAAM

Da es sich bei der LRAAM um ein Neuronales Netz handelt, das die zu benutzende Kompressions- und Dekompressionsfunktion erst erlernen muß, wurden die Tests etwas modifiziert.

Zuerst wurden Trainings- und Testdatensätze zusammengestellt. Dazu wurden aus einem Testdatensätz der HRR zufällig Terme ausgewählt und entweder in den Trainings- oder den Testdatensatz der LRAAM geschrieben. Für diese Aufgabe wurde durch das Hilfsprogramm `split_rnd` eingesetzt. Die Auswahlwahrscheinlichkeiten sind über das Programm frei wählbar. Die in den folgenden Tests benutzten Trainingsmengen enthalten 22 (d2) bzw. 21 (d3), die Testmengen 41 (d2) bzw. 462 (d3) Terme.

Ein Problem bei der Untersuchung der LRAAM stellt die Parametervielfalt dar. Abgesehen von dem prinizpiell schon schwierigen Lernproblem, hat es sich gezeigt, daß die LRAAM mit der Variante des Injektionsverfahrens sehr kritisch bezüglich Lernrate und Momentumterm reagiert. Die optimale Einstellung ist stark von den verwendeten Trainingsdaten abhängig. Eine mögliche Lösung ist die Aufnahme von Teiltermen schon in der Trainingsmenge enthaltener Terme.

Die LRAAM wurde mit Teilen der Testdatensätze d2 und d3, sowohl mit als auch ohne Injektionsmethode angelernt. Weiterhin wurden die Größe der Repräsentationsschicht und die Anzahl der Schichten variiert.

Die Struktur und die Parameter der LRAAM sowie die verwendeten Trainings- und Testdatensätze können der folgenden Tabelle entnommen werden.

Bezeichnung	Struktur	Injekt.	Lernrate η	Moment μ	Datensatz
x-20-x-d2	70-20-70	nein	0.3	0.9	d2
x-20-x-mt-d2	70-20-70	ja	0.3	0.9	d2
x-30-x-d3	100-30-100	nein	0.3	0.9	d3
x-30-x-mt-d3	100-30-100	ja	0.3	0.9	d3
x-30-20-30-x-d3	70-30-20-30-70	nein	0.05	0.3	d3
x-30-20-30-x-mt-d3	70-30-20-30-70	ja	0.05	0.3	d3
x-35-30-35-x-d3	100-35-30-35-100	nein	0.3	0.6	d3
x-35-30-35-x-mt-d3	100-35-30-35-100	ja	0.05	0.3	d3

Das Training wurde nach spätestens 10.000 Epochen beendet oder, wenn der absolute Fehler den Schranke von 0.5 unterschritt. Während des Trainings wurde alle 50 Epochen die Reproduktionsfähigkeit, sowahl an der Trainings- als auch an der gewählten Testmenge untersucht und protokolliert.

Auswertung der Ergebnisse

Die Abbildungen 5.3 bis 5.10 sind jeweils dreigeteilt. Im oberen Diagramm ist die Reproduktionsfähigkeit, bezogen auf die Trainings- und die Testmenge, angegeben. In diesem Diagramm werden jeweils nur die Terme berücksichtigt, die *vollständig* reproduziert, also ohne Fehler auf der symbolischen Ebene dekodiert wurden.

Die beiden, darunter liegenden Diagramme stellen die fehlerhaft dekodierten Terme, einmal für die Trainingsmenge (Mitte) und einmal für die Testmenge (unten) dar. Die Kurven repräsentieren die einzelnen Fehlergrade. Für die Terme heißt dies, ab welcher Tiefe trat zuerst ein Fehler auf. Für jede mögliche Tiefe wurde eine Kurve eingezeichnet, wobei *diff 0* einen Fehler auf der Stufe Null bedeutet, das Wurzelelement des Terms also schon falsch dekodiert wurde.

Die 70–20–70–LRAAM erwies sich als geeignet für Terme der maximalen Tiefe zwei. Es war unerheblich, ob mit oder ohne Injektionsmethode trainiert wurde. Bei Einsatz der Injektionsmethode ist das Lernverhalten jedoch „unruhiger“. Die Generalisierungsfähigkeit ist etwas besser.

Von diesen Ergebnissen ausgehend, wurde eine 100–30–100–LRAAM (Abbildung 5.5) auf Terme der Tiefe drei trainiert. Die Ergebnisse zeigen einen deutlichen Einbruch bei der Reproduktion der Trainingsmenge und noch einen viel stärkeren Einbruch bei der Testmenge. Die LRAAM ist aber noch in der Lage, zumindest das Wurzelelement des Terms bei 80% aller Testterme korrekt zu reproduzieren.

Die schlechte Generalisierungsfähigkeit ist auf die, im Verhältnis zur Testmenge, kleine Trainingsmenge verständlich. Ob sich jedoch die Trainingsleistung durch mehr Trainingsbeispiele verbessert, bleibt zu untersuchen, ist aber unwahrcheinlich, da dadurch die Lernaufgabe weiter erschwert wird.

Eine echte Verbesserung konnte mit einer erweiterten LRAAM (x-35-30-35-x-mt-d3, Abbildung 5.10) erreicht werden. Sie erreichte eine Reproduktion von 58% auf der Trainingsmenge und konnte alle anderen Terme der Trainingsmenge bis zur Tiefe zwei korrekt reproduzieren. Die Terme der Testmenge wurden zu 16% bis zur Tiefe zwei und zu 80% bis zur Tiefe eins reproduziert.

Die Verkleinerung der Schichten führte zu einer Verbesserung der Generalisierungsfähigkeit von 16% auf 28% für die Tiefe zwei, Abbildung 5.8. Die Trainingsmenge konnte jedoch nicht mehr angelernt werden.

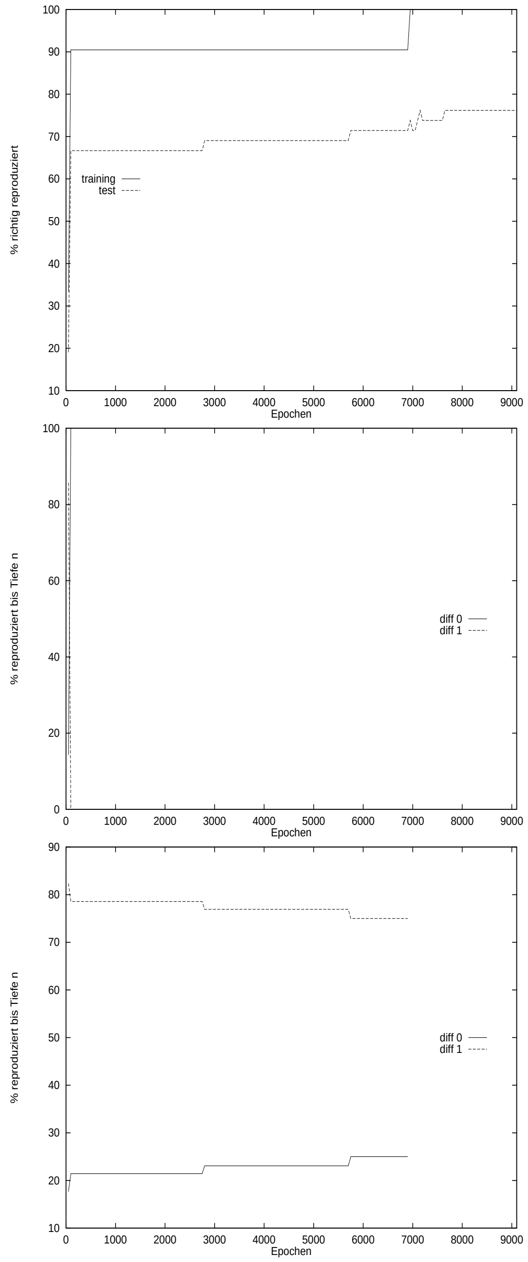


Abb. 5.3: lraam–x–20–x–d2

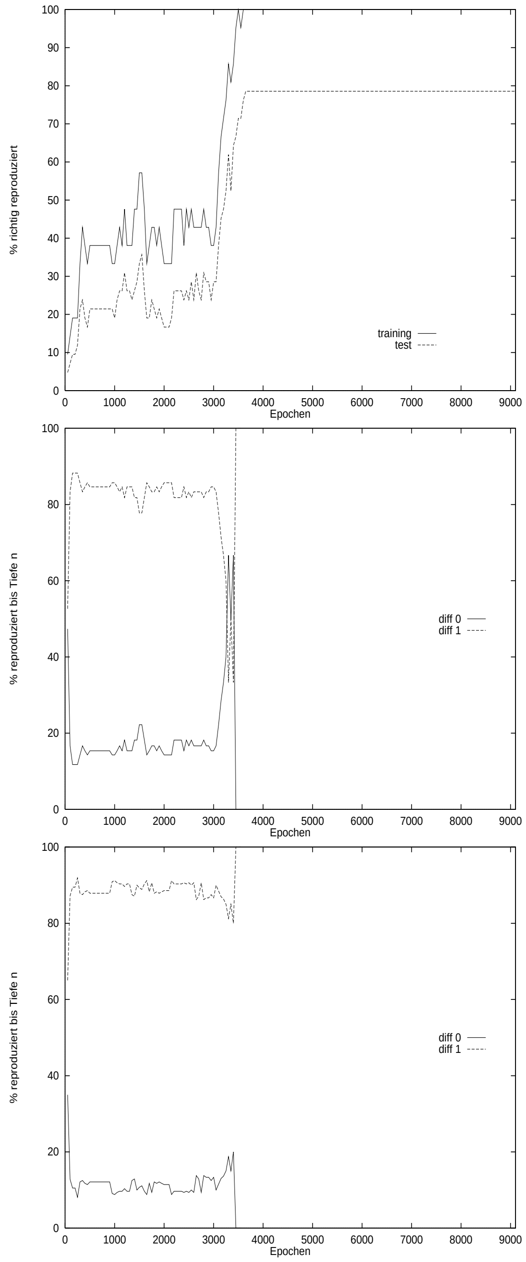


Abb. 5.4: lraam–x–20–x–mt–d2

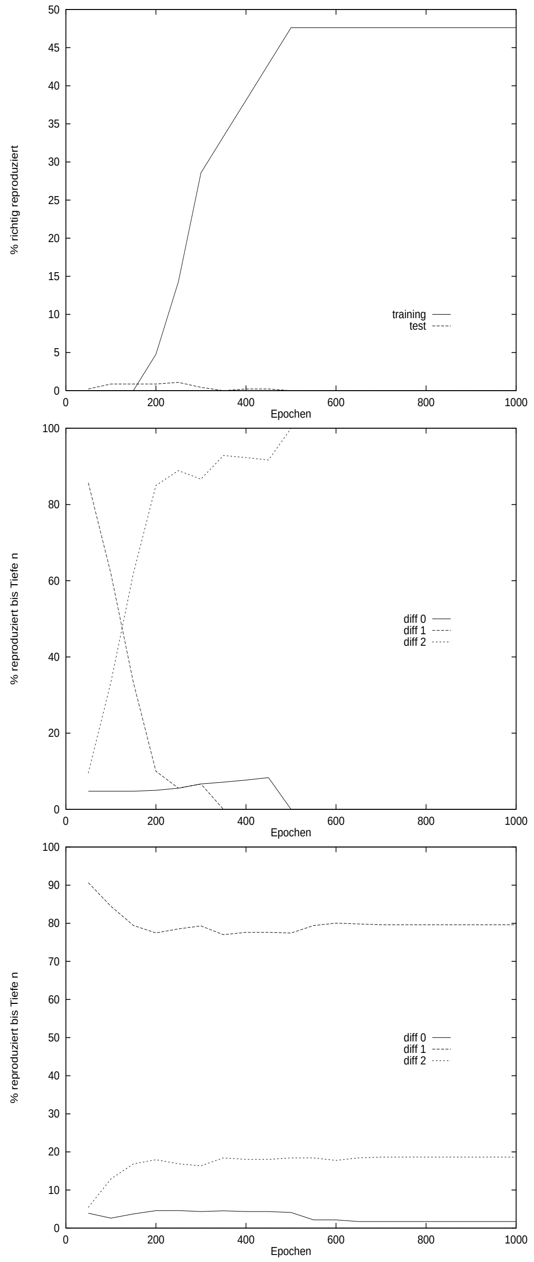


Abb. 5.5: lraam–x–30–x–d3

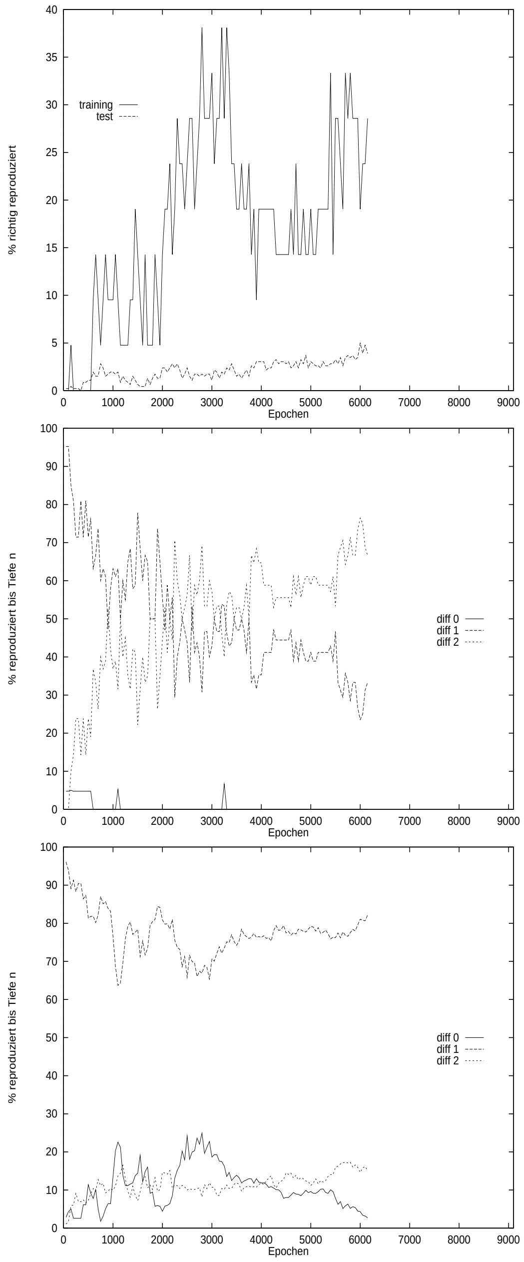


Abb. 5.6: lraam–x–30–x–mt–d3

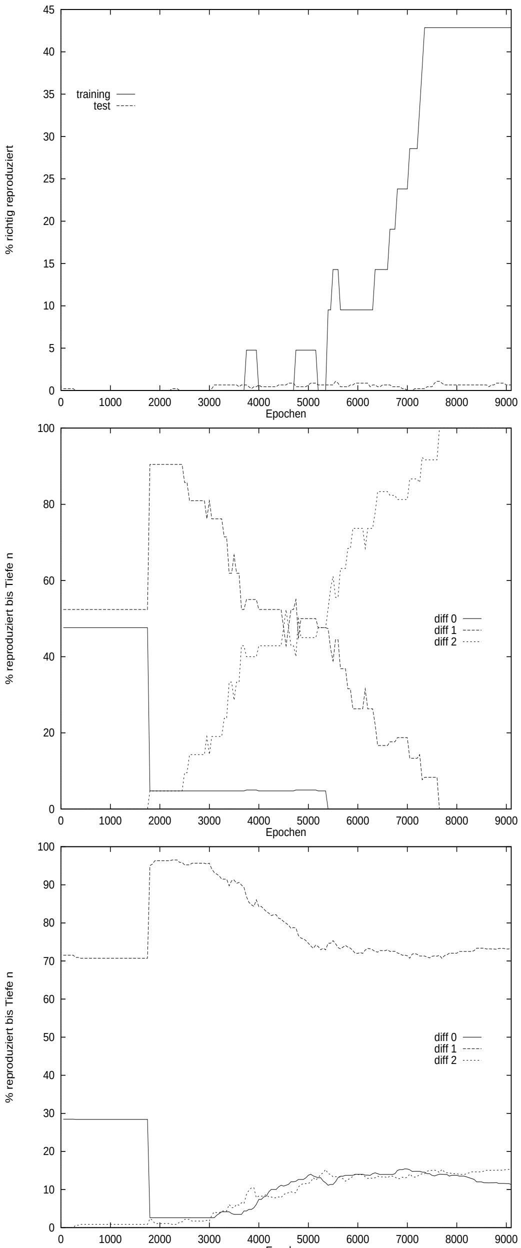


Abb. 5.7: lraam–x–30–20–30–x–d3

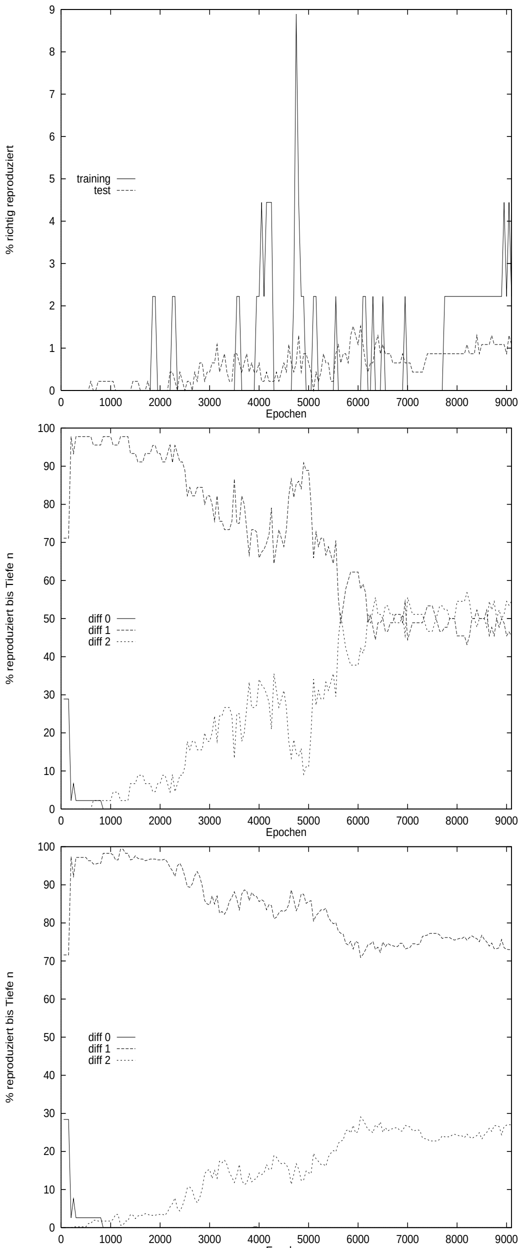


Abb. 5.8: lraam–x–30–20–30–x–mt–d3

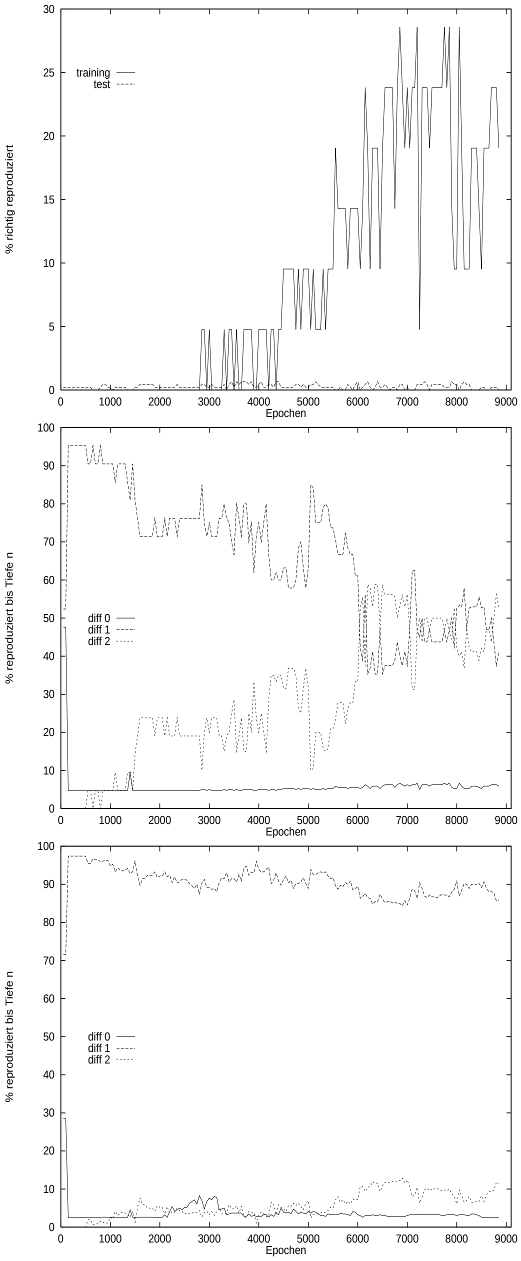


Abb. 5.9: lraam–x–35–30–35–x–d3

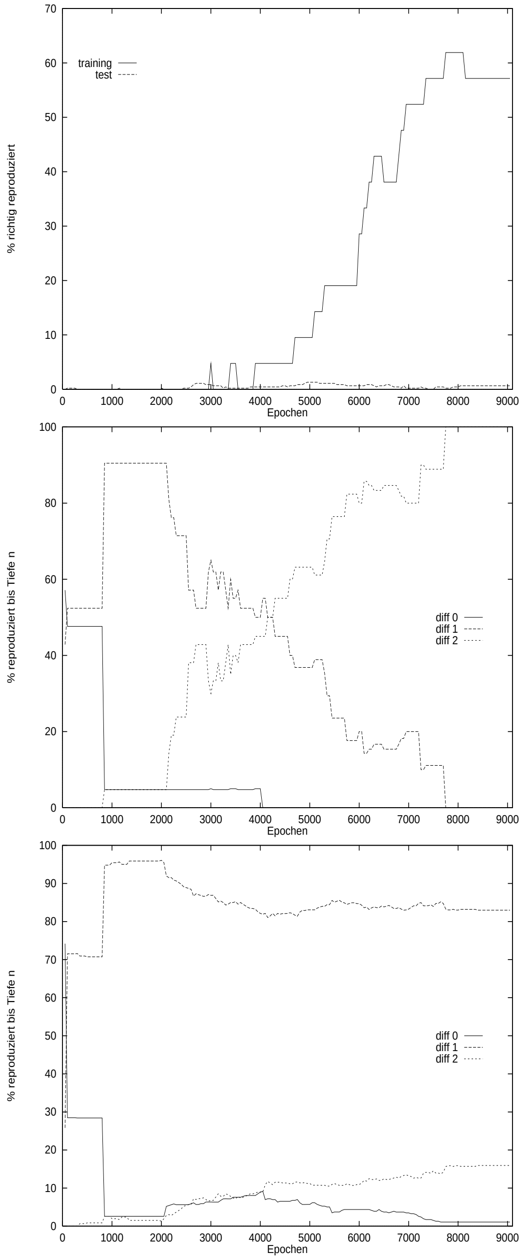


Abb. 5.10: lraam–x–35–30–35–x–mt–d3

5.4 Vergleich HRR und LRAAM

Gemeinsamkeiten von HRR und LRAAM

- Beide Modelle benutzen dimensionserhaltende Repräsentation für frame-artige Strukturen.
- Beide haben Probleme, tiefe Strukturen zu repräsentieren und korrekt wiederzugeben.
- Sie sind gut zu parallelisieren.
- Die gesamte Repräsentation muß sequentiell durchmustert werden, um eine Konvertierung von der symbolischen in die subsymbolische Domäne (bzw. zurück) durchzuführen.
- Beide Modelle sind fehlertolerant und weniger zerbrechlich als symbolische Repräsentationen.

Vorteile der HRR gegenüber der LRAAM

- Es ist kein Training nötig.
- Die Eigenschaften der HRR sind einfach und vorhersagbar und somit einer Analyse zugänglich (zumindest bis zu einem gewissen Grad, da Plate bei seinen Analysen von geeignet guten Zufallszahlen für die Basisrepräsentationen ausgeht und recht starke Annahmen bezüglich der zu repräsentierenden Strukturen macht).
- Es können beliebige (azyklische) Strukturen in HRRs gespeichert werden.
- HRRs verteilen die Informationen zwischen
  - dem Konvolution-Memory-Trace und
  - dem Clean-Up-Memory.
- HRRs haben eine sehr hohe Variabilität bzgl. der Rollen, wohingegen die RAAMs auf eine kleine feste Anzahl von positionalen Rollen festgelegt sind.
- Nach Plate sind HRRs nicht sehr von der Genauigkeit der benutzten reellen Vektoren abhängig, da sich eine geringere Genauigkeit nur in größerem Rauschen niederschlägt.

Nachteile der HRR gegenüber der LRAAM

- HRRs ziehen keinen Nutzen aus der Wahrscheinlichkeitsverteilung des Auftretens von Strukturen.
- HRRs benötigen sehr hochdimensionale Vektoren.

- Damit die statistischen Eigenschaften der HRR zutreffen, müssen die Basisvektoren zufällig und unabhängig verteilt sein.
- Die HRR ist nicht in der Lage zu generalisieren.

6. Zusammenfassung und Ausblick

Inhalt dieser Arbeit ist die Schaffung eines Werkzeuges für die Untersuchung von verteilten Termrepräsentationen. Die so entstandenen Klassenbibliothek wurde an zwei Vertretern verteilter Termrepräsentationen erprobt. Die Ergebnisse der Experimente erlauben eine Einschätzung der Brauchbarkeit der hier untersuchten Modelle für den Einsatz als Termrepräsentation für prädikatenlogische Programme.

Es konnte durch die Experimente gezeigt werden, daß die HRR einen exponentiellen Aufwand an Ressourcen bezogen auf die Tiefe der repräsentierten Terme benötigt. Es wurden aber auch Lösungswege aufgezeigt, dieses Problem zu umgehen, so zum Beispiel daß von Plate vorgeschlagene „Chunking“.

Der Einsatz der HRR ist in Systemen angebracht, wo hochdimensionale Vektoren problemlos verarbeitet werden können und überwiegend flache Strukturen auftreten. Außerdem besitzen Systeme, die auf der HRR beruhen die Eigenschaft, daß sie nicht erst die Repräsentation erlernen müssen. Dies kann aber auch ein Nachteil sein, wenn in der betrachteten Anwendung nur ein verschwindend kleiner Anteil aller möglichen Terme auftritt, denn die Abbildung ist nicht problemspezifisch.

Ein weiteres Problem stellt der Clean-Up-Speicher dar. Sollen tiefe Strukturen mittels Chunking in flache überführt werden, werden Assoziativspeicher größerer Kapazität und der Fähigkeit des nachträglichen Lernens benötigt.

Sollen domänenspezifische Ähnlichkeiten in die Vektoren aufgenommen werden, so bewirkt dies eine zusätzliche Störung der Repräsentation durch Übersprechen. Die Folge ist wiederum eine Kapazitätsverringerung.

Der Einsatz der LRAAM ist sinnvoll, wenn effiziente, domänenspezifische Repräsentationen benötigt werden. Die für den jeweiligen Zweck optimale Repräsentation wird durch den Lernprozeß ermöglicht. Leider haben die Experimente aber auch gezeigt, daß es äußerst schwierig ist, eine LRAAM für Terme größerer Tiefe anzulernen. Die in dieser Arbeit untersuchten Lernverfahren, BPTS und BPTS mit Injektion, können als nicht zufriedenstellend bezeichnet werden. Die von Küchler und Goller erzielten Termtiefen bei der *Klassifikation* von Termen konnten für die Repräsentation nicht erreicht werden. An dieser Stelle müssen unbedingt weitere Tests durchgeführt werden. Es hat sich auch gezeigt, daß die Lernalgorithmen äußerst empfindlich auf die gewählten Parameter reagieren und häufig in lokalen Minima hängen bleiben oder zum Schwingen neigen.

Für Terme der maximalen Tiefe zwei hat die 70–20–70–LRAAM eine effizientere Kodierung gefunden, als die HRR, denn die Repräsentationsschicht ist nur 20 Elemente groß und erreicht eine 77 prozentige Reproduktion auf der Testmenge. Die HRR erreicht erst bei der Vektorlänge 64 eine 70 prozentige Erkennung bei gleichen Termtiefen.

Dieses Ergebnis ist auch ein Indiz dafür, daß die LRAAM keine Approximation der HRR darstellt, sondern eine effizientere, dafür aber domänenabhängige Repräsentation liefert.

A. Klassenbeschreibungen

A.1 Einleitung und Übersicht

Die nachfolgende Übersicht enthält alle Klassen mit einer Kurzbeschreibung. Die Funktionalität sowie die Schnittstellen sind dem nachfolgenden Abschnitt zu entnehmen.

Klasse	Beschreibung
`BPTS`	Lernalgorithmus BPTS
`Encoder`	Encoder-Struktur für LRAAM
`Decoder`	Decoder-Struktur für LRAAM
`Layer`	Neuronen-Layer
`CompositeLayer`	zusammengesetzter Neuronen-Layer
`Weights`	Gewichtsvektor (für ein Neuron)
`WeightsMatrix`	Gewichtsvektoren für einen Layer
`WeightsMatrixVector`	Alle Gewichtsmatrizen für komplette LRAAM
`LRAAM`	Verteilter Speicher, basierend auf der LRAAM
`Neuron`	Neuron
`ReducedRepresentation`	Basisklasse für reduzierte (verteilte) Repräsentation
`Symbol`	Symbol mit Namen und Stelligkeit
`SymbolStruct`	Baumstruktur, bestehend aus Symbolen
`SymbolStructIterator`	Iterator für elementweises Durchlaufen einer Symbolstruktur
`SymbolStructRnd`	Generator für zufällige Symbolstrukturen
`DistributedMemory`	Basisklasse für verteilte Speicher
`HRR_Vector`	Reellwertiger Vektor mit Faltungsoperation
`HRR_MEM`	Verteilter Speicher, basierend auf der HRR
`SymTab`	Symboltabelle
`Printable`	Vereinheitlichte Ausgabe

Die Auszüge aus den Header-Dateien enthalten wenigstens alle *Public*-Memberfunktionen der beschriebenen Klassen. Sollten Teile der Klassendeklaration entfernt worden sein, so ist dies durch `[...]` kenntlich gemacht. Für eine vollständige Beschreibung möchte ich auf die Quelldateien verweisen, da auch hier zusätzliche Bemerkungen zur Implementation enthalten sind (`.cc`-Files).

A.2 Deklarationen

Symbol

Klasse: `Symbol` in `symbol.hh`

Klassendefinition:

```
class Symbol
{
private:
protected:
    int   arity;
    char* value;
public:                      // constructors
    /* Constructors */
    Symbol(const char* sym_value, int sym_arity = 0);
    Symbol(const Symbol &c);

    /* Destructor */
    virtual ~Symbol();

    inline char * GetValue() { return value; };
    inline int    GetArity() { return arity; };

    inline void   SetArity(int a) { arity = a; };
    void   SetValue(char* new_value);

    int operator == (const Symbol &s)
    {
      return ( (arity == s.arity) && (strcmp(value, s.value) == 0) );
    };
};
```

Beschreibung: Die Klasse `Symbol` speichert den Namen und die Stelligkeit eines Symbols. Symbole könnnen Funktionszeichen oder aber während der Konstruktion von verteilten Speichern nachträglich eingeführte Bezeichner sein. Solche zusätzlichen Symbole sind beispielsweise die Schlüssel für die Kodierung eines Terms in einem HRR-Speicher.

Symboltabelle

Klasse: `SymTab` in `symtab.hh`

Klassendefinition:

```
#define SYMTAB_NO_SYMBOL -1

class SymTab
{
protected:
  SymTabEntryVec sym_array;
  int            num_of_symbols;
public:
  SymTab();
  virtual ~SymTab();

  void Resize(int newlen) { sym_array.resize(newlen); };
  int GetSymTabSize() { return sym_array.capacity(); };

  /* return idx of new symbol */
  int AddSymbol(char* value, int arity);
  int AddSymbol(String& value, int arity);

  /* return idx of symbol or -1 if symbol idx doesn't exist */
  int SetReferenceSymbolRR(const int sym_idx,
                           ReducedRepresentation* rr);

  /* return index of symbol or -1 if no symbol found */
  int GetSymbolIndexByName(char* value, int arity);
  int GetSymbolIndexOnlyByName(char* value);
  int GetSymbolIndexByRR(ReducedRepresentation& rr);

  /* return ptr to RedRepresentation of symbol with index 'index'
     or NULL if no symbol found */
  ReducedRepresentation* GetRRByIndex(int index);

  /* return ptr to Symbol with index or NULL if no symbol found */
  Symbol* GetSymbol(int index);
  void Print(ostream& o);
};
```

Beschreibung: Alle während des Programmlaufes definierten Symbole werden in einer Symboltabelle verwaltet. In dieser Tabelle befinden sich nicht nur die Symbole, sondern auch ihre verteilten Repräsentationen, sofern diese bekannt sind. Dadurch ist es möglich, den Auffrischungsspeicher (cleanup-memory) mit in der Symboltabelle zu integrieren. Die Symboltabelle fungiert dabei als Autoassoziativer Speicher und liefert zu einer gegebenen gestörten Repräsentation die am besten passende Repräsentation und deren Ähnlichkeit. Als Ähnlichkeitsmaß wird dabei das Skalar-Produkt benutzt. Die Symboltabelle besitzt weiterhin entsprechende Zugriffs-, Manipulations- und Suchfunktionen.

Symbolstrukturen / Terme

Klasse: `SymbolStruct` in `symstruct.hh`

Klassendefinition:

```
class SymbolStruct : public Printable
{
protected:
  [...]
public:
  SymbolStruct();
  SymbolStruct(String composed_symbol_string);
  SymbolStruct(const SymbolStruct &c);

  virtual ~SymbolStruct();

  SymStructNode* CopySymStructTree(SymStructNode* src_tree);
  void DeleteSymStructTree(SymStructNode* node);

  void IteratorLock() { iterator_lock++; };
  void IteratorUnlock() { iterator_lock--; };

  SymStructNode* GetSymStructRootNode() const;
  void SetSymStructRootNode(SymStructNode* p_ssn);
  /* returns 0 iff s and this vector are equal otherwise 1 */
  int Difference(SymbolStruct& s, SymStructDiff& diff);
  int MaxDepth();
  double MBranch(); // mean branch
  double MDepth();  // mean depth
  int NumOfNodes();
  int NumOfLeaves();

  void SetPrintStyle(int style) { prt_style = style; };
  virtual void PrintClassName(ostream& o) { o << "SymbolStruct"; };
  virtual void PrintContent(ostream& o);
  virtual void PrintContentPlain(ostream& o);

protected:
  [...]
};
```

Beschreibung: Die Klasse `SymbolStruct` dient der Konstruktion und Speicherung von Termen, bestehend aus Funktionszeichen. Wähend der Dekodierung einer reduzierten Repräsentation ist es aber durchaus möglich, daß auch andere Symbole in die Termstruktur geraten. Diese werden dann entsprechend Ihrer definierten Stelligkeit als Funktionszeichen betrachtet. Für die durchzuführenden Tests wird eine abgeleitete Klasse `SymbolStructRnd` genutzt, die zufällige Terme einer maximalen Tiefe und Breite über einem gegeben Alphabet erzeugt.

Symbolstrukturiterator

Klasse: `SymbolStructIterator` in `symstruct.hh`

Klassendefinition:

```
enum IterationMode {depth_first,
                    right_assoc,
                    breadth_first};

class SymbolStructIterator
{
private:
protected:
  IterationMode          iteration_mode;
  SymStructNode          *curr_node;
  SymbolStruct           *p_sym_struct;
  IterationStructSLStack stack;

public:
  SymbolStructIterator(SymbolStruct& sym_struct,
                       IterationMode im=depth_first);
  SymbolStructIterator(const SymbolStructIterator &c);

  virtual ~SymbolStructIterator();

  void SetIterationMode(IterationMode im);

  void Reset();
  /* returns symbol id otherwise -1 */
  int GetSymbolIndex();
  int GetArity();

  /* returns pointer to SymStructNode or NULL */
  PSymStructNode GetNode();

  /* get next symbol index with search strategy iter_mode
     returns -1 if there is no next symbol
     */
  int NextSymbolIndex();
};
```

Beschreibung: Für die Algorithmen, die Operationen über Symbolstrukturen durchführen, ist es günstig, iterativ durch die Baumstruktur zu laufen. Die Art des Durchlaufens kann über einen Parameter der Klasse eingestellt werden. Es ist auch möglich, mehrere Iteratoren auf die gleiche Baumstruktur anzuwenden. Ein aktiver Iterator setzt in der Symbolstruktur einen Semaphor, so daß dieser nicht versehentlich gelöscht oder verändert werden kann, solange noch ein Iterator über der Struktur arbeitet.

Symbolstruktur, zusätzliche Definitionen

Klasse: `SymStructNode`, `SymStructDiff`, `PrtStyle` in `symstruct.hh`

Klassendefinition:

```
typedef struct SymStructNode* PSymStructNode;

typedef enum {PRT_WITH_ERROR=1,
              PRT_PRETTY=2,
              PRT_LEFT=4} PrtStyle;

/* PDecoder, PEncoder declared in bpts.hh */
typedef class Encoder*  PEncoder;
typedef class Decoder*  PDecoder;

struct SymStructNode
{
  int             sym_index;     /* symbol index from symtab */
  int             arity;         /* copy of symbol arity */
  PSymStructNode* child_symbols; /* pointer to an array with pointer */
  PSymStructNode  parent_symbol;
  double          error;         /* contains decode distance */
  PEncoder        encoder;       /* encoder module */
  PDecoder        decoder;       /* decoder module */
};

struct SymStructDiff
{
  int depth; // last identical level
  int ok;    // structures are equal
};
```

Beschreibung: Die Klasse `SymbolStruct` nutzt zur Konstruktion von Termen (Bäumen) die Struktur `SymStructNode`. Entsprechend der Stelligkeit des im Knoten repräsentierten Symbols befinden sich Zeiger auf weitere Knoten in der Struktur. Die Struktur `SymStructDiff` wird als Ergebnistyp für den Vergleich zweier Symbolstrukturen benötigt. Sie enthält ein Flag, das die vollständige Gleichheit bestätigt und eine Variable, die im Falle der Ungleichheit, die maximale Tiefe der Übereinstimmung angibt. Sollen Symbolstrukturen über Streams ausgegeben werden, so kann die Ausgabe durch Angabe eines Druckstiles beeinflußt werden.

Lernalgorithmus BPTS

Klasse: `BPTS` in `bpts.hh`

Klassendefinition:

```
extern int move_target_enabled; // default is 0

class BPTS
{
protected:
  SymbolStruct           symbol_struct;
  PWeightsMatrixVector   p_weights;
  PWeightsMatrixVector   p_delta_weights;
  PWeightsMatrixVector   p_momentum;
  int                    num_of_layers;
  int                    label_size;
  int                    num_of_pointers;
  int                    pointer_size;
  int                    rr_layer;
  int*                   p_layer_sizes;
  PLayer                 p_rr_layer;
public:
  BPTS(SymbolStruct&        sym_struct,
       int                  num_of_l,    /* # layers */
       int                  lbl_size,    /* # units for labels */
       int                  num_of_ptrs, /* # branches */
       int                  ptr_size,    /* # units for each pointer */
       PLayerSizes          p_l_sizes,   /* [0..num_of_layers] of int */
       PWeightsMatrixVector p_wg,
       PWeightsMatrixVector p_delta_wg,
       PWeightsMatrixVector p_mom);
  virtual ~BPTS();
  PLayer CreateEncoderTree(PSymStructNode src_tree);
  void   CreateDecoderTree(PSymStructNode src_tree, PLayer prev_layer);
  double Train(double min_error, int max_iter=10000);
  double TrainOneStep();                  /* without weight update */
  void   Process() { ProcessTree(); };
  ReducedRepresentation GetRR();
  void   PrintTree();
  void   UpdateWeights();
protected:
  void   MoveTarget(PSymStructNode sym_node);
  void   ProcessTree();
  double BackpropTree();
};
```

Beschreibung: Die Klasse `BPTS` baut mit den Klassen `Encoder` und `Decoder` eine entfaltete Form der LRAAM, bezogen auf *eine* Symbolstruktur, auf. Weiterhin sind hier die Einstiegsmethoden für die Musterpropagierung und das Lernen enthalten.

LRAAM Kodierer

Klasse: `Encoder` in `bpts.hh`

Klassendefinition:

```
class Encoder
{
protected:
  PWeightsMatrixVector p_weights_matrix;
  PWeightsMatrixVector p_delta_weights_matrix;
  PWeightsMatrixVector p_momentum_matrix;
  int                  num_of_layers;
  int                  label_size;
  int                  num_of_pointers;
  int                  pointer_size;
  int                  rr_layer;
  PLayer*              layer_array; /* Array of Pointer to Layer */
public:
  Encoder(int num_of_l,            /* # layers */
          int lbl_size,            /* # units for labels */
          int num_of_ptrs,         /* # branches */
          int ptr_size,            /* # units for each pointer */
          int rr_l,                /* idx of reduced represent. */
          PLayerSizes p_l_sizes,   /* array[0..num_of_layers] of int */
          PWeightsMatrixVector p_wg,
          PWeightsMatrixVector p_delta_wg,
          PWeightsMatrixVector p_mom);
  virtual ~Encoder();

  inline PLayer  GetLayer(int idx);
  inline PLayer  GetLastLayer();

  /* branch is in range [0 <= branch < num_of_pointers] */
  void    SetPrevLayer(int branch, PLayer p_prev_layer);
  void    SetLabel(int value, int arity);

  void    SetPointerRR(int idx, ReducedRepresentation& rr);

  /* returns rr only if encoder contains rr_layer otherwise NULL*/
  ReducedRepresentation GetRR();

  inline void Process();
  double Backprop();
  virtual void Print();
};
```

Beschreibung: Die Klasse realisiert den Kodierteil der LRAAM. Sie enthält die Methoden zur Musterpropagierung, Verknüpfung mit weiteren Dekodern und der Ergebnisgewinnung.

LRAAM Dekoder

Klasse: `Decoder` in `bpts.hh`

Klassendefinition:

```
class Decoder
{
protected:
  PWeightsMatrixVector p_weights_matrix;
  PWeightsMatrixVector p_delta_weights_matrix;
  PWeightsMatrixVector p_momentum_matrix;
  int                  num_of_layers;
  int                  label_size;
  int                  num_of_pointers;
  int                  pointer_size;
  int                  rr_layer;
  PLayer*              layer_array; /* [num_of_layers] */
public:
  Decoder(int num_of_l,            /* # layers */
          int lbl_size,            /* # units for labels */
          int num_of_ptrs,         /* # branches */
          int ptr_size,            /* # units for each pointer */
          int rr_l,                /* idx of reduced represent. */
          PLayerSizes p_l_sizes,   /* array[0..num_of_layers-1] of int */
          PWeightsMatrixVector p_wg,
          PWeightsMatrixVector p_delta_wg,
          PWeightsMatrixVector p_mom,
          PLayer               p_prev_layer);
  virtual ~Decoder();

  PLayer GetLastLayer();
  PLayer GetPointer(int idx);

  /* set rr direct, only if decoder contains rr_layer */
  void SetRR(ReducedRepresentation& rr);
  void GetLabel(int& value, int& arity);
  ReducedRepresentation GetPointerRR(int idx);

  void SetTarget(int value, int arity);

  void Process();
  double  Backprop();
  virtual void Print();
};
```

Beschreibung: Die Klasse realisiert den Dekodierteil der LRAAM. Sie enthält die Methoden zur Propagierung der reduzierten Repräsentation, Zielmusterfestlegung, Verknüpfung mit anderen Dekodern bzw. Kodierern und zur Ergebnisgewinnung.

Neuronenschicht

Klasse: `Layer` in `bpts.hh`

Klassendefinition:

```
class Layer
{
protected:
  int            size;
  PNeuron        neuron_array; /* [size] */
  PLayer         p_prev_layer;
  PWeightsMatrix p_weights;
  PWeightsMatrix p_delta_weights;
  PWeightsMatrix p_momentum;
public:
  Layer(int            layer_size,
        PLayer         p_prev,
        PWeightsMatrix p_wg,
        PWeightsMatrix p_delta_wg,
        PWeightsMatrix p_mom);
  virtual ~Layer();

  void SetPrevLayer(PLayer p_prev);

  inline PLayer GetPrevLayer() { return p_prev_layer; };
  inline virtual int GetSize() { return size; };
  inline virtual Neuron& Get(int idx);

  inline virtual void SetInput(int idx, double value);
  inline virtual double GetOutput(int idx);
  inline virtual void SetTarget(int idx, double value);

  /* Process Input from previous layer an calculate output */
  virtual void Process();
  /* for each neuron in layer
       get current error and calculate delta weights */
  virtual void UpdateDeltaWeights();

  /* for each neuron in layer set error to zero */
  virtual void ClearUnitError();
  /* set error of unit (idx) to err */
  virtual void SetUnitError(int idx, double err);
  /* add err to unit (idx) error */
  virtual void AddUnitError(int idx, double err);

  virtual double Backprop();
  virtual void Print();
};
```

Beschreibung: Diese Klasse fast eine Neuronenschicht zusammen. Sie bildet die Neuronenfunktionalität Vorwärts- und Rückwärtspropagierung auf Methoden einer Schicht ab.

Zusammengesetze Neuronenschicht

Klasse: `CompositeLayer` in `bpts.hh`

Klassendefinition:

```
class CompositeLayer : public Layer
{
protected:
  int     num_of_layers;
  PLayer* p_layers; /* layers[number of layers] */
public:
  CompositeLayer(int number);
  virtual ~CompositeLayer();

  void CalcSize(); // add size of all sublayers

  PLayer GetLayer(int idx) { return p_layers[idx]; };
  void AddLayer(int idx, PLayer p_l);

  inline virtual Neuron& Get(int idx);

  /* Process Input from previous layer an calculate output */
  virtual void Process();

  /* for each neuron in layer
       get current error and calculate delta weights */
  virtual void UpdateDeltaWeights();
  /* for each neuron in layer set error to zero */
  virtual void ClearUnitError();

  virtual double Backprop();
  virtual void Print();
};
```

Beschreibung: Diese Klasse fast mehrere Schichten zu einer Schicht zusammen. Diese Klasse verhält sich wie eine einzelne Schicht. Diese Funktionalität wird in der Eingabeschicht des Kodierers und der Ausgabeschicht des Dekodierers benötigt.

Gewichtsvektor für ein Neuron

Klasse: `Weights` in `weights.hh`

Klassendefinition:

```
class Weights : public doubleVec
{
public:
  Weights() : doubleVec() {InitRandom();};
  Weights(int l) : doubleVec (l) {InitRandom();};
  Weights(int l, double  fill_value) : doubleVec (l, fill_value) {};
  Weights(const doubleVec& c) : doubleVec (c) {};

  virtual ~Weights() {};
  virtual void InitRandom();

  /* init from file */
  void Write(ofstream& out);
  void Read(ifstream& in);
};
```

Beschreibung: Diese Klasse implementiert einen reellwertigen Vektor. Dieser dient für ein Neuron als Wichtungsvektor.

Gewichtsvektoren für Neuronenschicht

Klasse: `WeightsMatrix` in `weights.hh`

Klassendefinition:

```
class WeightsMatrix
{
protected:
  PWeights*  weights;
  int        size;
public:
  WeightsMatrix(int l, int d);
  virtual ~WeightsMatrix();

  inline PWeights operator[] (int n);
  inline int GetSize();
  void SetAll(double d);
  virtual void InitRandom();

  /* init from file */
  void Write(ofstream& out);
  void Read(ifstream& in);

  WeightsMatrix & operator = (const WeightsMatrix & a);
  WeightsMatrix & operator += (const WeightsMatrix & a);
};
```

Beschreibung: Diese Klasse implementiert eine Matrix mit reellen Elementen. Die Matrix dient als Wichtungsmatrix für eine Schicht von Neuronen.

Gewichtsvektoren für LRAAM

Klasse: `WeightsMatrixVector` in `weights.hh`

Klassendefinition:

```
class WeightsMatrixVector
{
protected:
  PWeightsMatrix* matrix;
  int             size;
public:
  WeightsMatrixVector(int label_size,
                      int max_branch,
                      int ptr_size,
                      int num_layers,
                      int* layer_sizes);

  virtual ~WeightsMatrixVector();

  inline PWeightsMatrix operator[] (int n);
  inline int GetSize();
  void SetAll(double d);
  inline void Clear();
  void InitRandom();

  /* init from file */
  void Write(ofstream& out);
  void Read(ifstream& in);

  WeightsMatrixVector & operator = (const WeightsMatrixVector & a);
  WeightsMatrixVector & operator += (const WeightsMatrixVector & a);
};
```

Beschreibung: Diese Klasse implementiert einen Vektor mit Gewichtsmatrizen und stellt für jede Schicht in der LRAAM eine Gewichtsmatrix zur Verfügung.

LRAAM

Klasse: `LRAAM` in `lraam.hh`

Klassendefinition:

```
class LRAAM : public DistributedMemory
{
protected:
  int  max_branch;   /* max arity of symbols */
  int  label_size;   /* size of label field */

  int* layer_sizes;  /* sizes of each layer */
  int  input_layer;  /* index of input layer, always 0 */
  int  hidden_layer; /* index of hidden layer */
  int  output_layer; /* index of output layer */

  /* all structures using the same weights */
  PWeightsMatrixVector p_weights;        /* array[wmatrix_size] */
  PWeightsMatrixVector p_delta_weights;  /* array[wmatrix_size] */
  PWeightsMatrixVector p_momentum;       /* array[wmatrix_size] */
public:
  LRAAM();
  LRAAM(int max_syms, int max_b, int rr_s, int* l_sizes, int layers=3);

  virtual ~LRAAM();

  virtual ReducedRepresentation* SS2RR(const SymbolStruct &sym_struct);
  virtual SymbolStruct*          RR2SS(const ReducedRepresentation &red_rep);
  virtual double Reliability();  /* [0,1] 1 == high reliability */

  virtual void Read(char* filename);
  virtual void Write(char* filename);

  /* return quadratic error for current term */
  double TrainOnePattern(SymbolStruct& sym_struct,
                         double min_error,
                         int max_iter=500);

  double TrainPatternSet(PSymbolStruct* p_sym_struct_array,
                         int num_of_pattern,
                         double min_error_per_pattern,
                         double min_error_all_pattern,
                         int    max_iter_per_pattern,
                         int    max_iter_all_pattern);

  /* default training algorithm from Sperduti */
  double TrainIterative(SymbolStruct& sym_struct,
                        double min_error,
                        int max_iter=500);
protected:
  [...]
};
```

Beschreibung: Diese Klasse implementiert den Kern der LRAAM. Sie enthält die vollständige Wichtungsmatrix und alle notwendigen Strukturinformationen, wie Größe des Labelfeldes, Anzahl der Pointer-Felder, sowie Zahl und Größe der einzelnen Schichten. Diese Klasse nutz für das Training die Klasse BPTS.

Neuron

Klasse: `Neuron` in `neuron.hh`

Klassendefinition:

```
class Neuron
{
protected:
  int       inp_size;       /* get from P_weights.GetSize() */
  int       has_external_input;
  double    external_input;
  int       has_target;     /* use target for backprop if exist */
  double    target_value;
  double    output_value;
  double    error;
  double    steepness;
  double    learn_rate;
  double    momentum_rate;
  PLayer    p_prev_layer;
  PWeights p_weights;
  PWeights p_momentum;
  PWeights p_delta_weights;
private:
  void Init();
public:
  Neuron();
  Neuron(const PLayer p_prev_l,
         PWeights p_wg, PWeights p_delta_wg, PWeights p_momentum);
  virtual ~Neuron();

  void ChangeLayer(const PLayer p_prev_l,
                   PWeights p_wg, PWeights p_delta_wg, PWeights p_momentum);
  void SetPrevLayer(PLayer p_layer);

  virtual void   Process();
  virtual double Calculate();

  /* limit x values to avoid domain errors
     dynamic range is [-1/2,1/2]   */
  inline double transfer(double x); // use logistic for default

  inline void   ChangeLearnRate(double lr);
  inline void   ChangeMomentumRate(double mr);
  inline void   ChangeSteepness(double st);

         void   ClearError();
  inline void   SetError(double err);
  inline void   AddError(double err);
  inline void   SetExtInp(double inp);
  inline double GetOutput();
  inline void   SetTarget(double val);
  inline double GetTarget();

  double Backprop();
protected:
  /* use current error to calculate delta_weights */
  virtual void UpdateWeights();
};
```

Beschreibung: In dieser Klasse wurde die Funktionalität der benutzen Neuronen implementiert. Jedes Neuron besitzt Zeiger auf die zu verwendenden Wichtungsmatrizen, die Vorgängerschicht und kennt auch deren Größe. Weiterhin enthält das Neuron Parameter zur Steuerung des Lernverhaltens und einen Speicherplatz für einen möglichen Sollwert. Wurde kein Sollwert vorgegeben, so verhält sich das Neuron wie ein Hiddenneuron und nutzt den zurückpropagierten Fehler zur Bestimmung der Wichtungsänderung, ansonsten bestimmt es den Fehler anhand des definierten Sollwertes. Die Korrektur der Wichtungsmatrix wird nur vorgenommen, wenn es sich bei dem Neuron nicht um ein Eingabeneuron handelt, da in diesem Fall die externe Eingabe direkt als Ausgabe verwendet wird.

Reduzierte Repräsentation

Klasse: `ReducedRepresentation` in `redrep.hh`

Klassendefinition:

```
class ReducedRepresentation : public doubleVec, public Printable
{
protected:
    ReducedRepresentation(int l, double* d) : doubleVec(l,d){};
public:
    /* Constructors */
    ReducedRepresentation() : doubleVec(){};
    ReducedRepresentation(int l) : doubleVec(l){};
    ReducedRepresentation(int l, double  fill_value) :
        doubleVec(l, fill_value){};
    ReducedRepresentation(const ReducedRepresentation &c) :
        doubleVec (c){};

    virtual ~ReducedRepresentation();

    /* distance function use vector dot product */
    friend double distance(ReducedRepresentation& rr1,
                           ReducedRepresentation& rr2);

    virtual void PrintClassName(ostream& o);
    virtual void PrintContent(ostream& o);
};
```

Beschreibung: Im wesentlichen handelt es sich bei einer verteilten Repräsentation um einen reellwertigen Vektor. Die Funktionalität ist von einem dynamischen double-Vektor der GNU-Libg++-Klassenbibliothek geerbt. Von dieser Klasse wurde die reduzierte Repräsentation für die HRR abgeleitet. weitere Klasse `HRR_Vector` abgeleitet.

Basisklasse für verteilte Speicher

Klasse: `DistributedMemory` in `distmem.hh`

Klassendefinition:

```
class DistributedMemory
{
protected:
    int                   rr_size; /* Size of reduced representation */

public:
    DistributedMemory(int rr_s) ;
    DistributedMemory(const DistributedMemory &c);

    virtual ~DistributedMemory();

    virtual ReducedRepresentation* SS2RR(const SymbolStruct &sym_struct) = 0;
    virtual SymbolStruct*          RR2SS(const ReducedRepresentation &red_rep) = 0 ;

    virtual double Reliability() = 0;  /* [0,1] 1 == high reliability */
};
```

Beschreibung: Hierbei handelt es sich um eine abstrakte Basiklasse für verteilte Speicher. Es wurden die Methoden `RR2SS()` und `SS2RR()` abstrakt deklariert. Diese müssen durch die abgeleiteten Klassen überschrieben werden.

Reduzierte Repräsentation mit Faltungsoperator

Klasse: `HRR_Vector` in `hrr_vector.hh`

Klassendefinition:

```
class HRR_Vector : public ReducedRepresentation
{
protected:
  HRR_Vector(int l, double* d) : ReducedRepresentation(l,d){};
public:
  HRR_Vector() :                          ReducedRepresentation(){};
  HRR_Vector(int l) :                     ReducedRepresentation(l){};
  HRR_Vector(int l, double fill_value) : ReducedRepresentation(l, fill_value){};
  HRR_Vector(const HRR_Vector &c) :            ReducedRepresentation (c){};
  HRR_Vector(const ReducedRepresentation &c) : ReducedRepresentation (c){};

  virtual ~HRR_Vector(){};

public:
  /* circular convolution */
  HRR_Vector operator* (HRR_Vector& v);
  HRR_Vector operator+ (HRR_Vector& v);

  HRR_Vector Inverse();
  void       Normalize();
  void       Initialize();

  virtual void PrintClassName(ostream& o) { o << "HRR_Vector"; };

protected:
  inline int mod (int i, int k);
};
```

Beschreibung: In dieser Klasse wurde der Operator für die Zirkularkonvolution implementiert. Außerdem besitzt diese Klasse eine spezielle Initialisierungsmethode, die garantiert, daß die Elemente des Vektor normalverteilt initialisiert werden (N(0,1/Vektorlänge)). Weiterhin wurde noch eine Methode zur Normalisierung des Vektors implementiert.

Verteilter Speicher basierend auf der HRR

Klasse: `HRR_MEM` in `hrr_mem.hh`

Klassendefinition:

```
const int F_NAME_KEY      = 0;
const int NIL_KEY         = 1;
const int FIRST_PARAM_KEY = 2;

class HRR_MEM : public DistributedMemory
{
private:
    HRR_Vector memory;
protected:
    int max_branch; /* max arity of represented functions */
    int* key_id;    /* keys */
    int* inv_key_id;/* inverse keys */

    virtual void InitMemory();
    virtual void CopyMemory(const HRR_MEM &c);
public:
    HRR_MEM(int max_b, int rr_s);
    HRR_MEM(const HRR_MEM &c);

    virtual ~HRR_MEM();

    virtual ReducedRepresentation* SS2RR(const SymbolStruct &sym_struct);
    virtual SymbolStruct*          RR2SS(const ReducedRepresentation &red_rep);

    /* add the new pattern and returns the whole memory representation */
    virtual ReducedRepresentation* Add(const ReducedRepresentation &red_rep);

    /* overwrite memory with this representation */
    virtual ReducedRepresentation* Store(const ReducedRepresentation &red_rep);

    /* get nearest representation */
    virtual ReducedRepresentation* Get(const ReducedRepresentation &key);

    /* remove this representation from memory */
    virtual ReducedRepresentation* Remove(const ReducedRepresentation &red_rep);
    virtual double Reliability();  /* [0,1] 1 == high */

    virtual void PrintClassName(ostream& o);
    virtual void PrintContent(ostream& o);

protected:
    HRR_Vector GetFuncRR(const int sym_idx);
    virtual HRR_Vector*  ComposeRR(const PSymStructNode p_ssn);
    virtual SymStructNode* ComposeSS(HRR_Vector rr);

};
```

Beschreibung: In dieser abgeleiteten Klasse wurden die Initialisierung, Normierung und die Zirkularkonvolution $\otimes$ hinzugefügt. Die Zirkularkonvolution wurde unter Nutzung der FFT implementiert, um auch HRR's mit größeren Vektorlängen (> 1024) untersuchen zu können. Zur Berechnung der FFT wurde eine Bibliothek von Peter Valkenburg (valke@cs.vu.nl) benutzt.

Formatierte Ausgabe

Klasse: `Printable` in `printable.hh`

Klassendefinition:

```
class Printable
{
public:

  Printable();
  Printable(const Printable &c);

  virtual ~Printable();

  virtual void Print(ostream& o);
  virtual void PrintClassName(ostream& o);
  virtual void PrintContent(ostream& o);
  virtual void PrintDebugInfos(ostream& o);

  friend ostream& operator <<(ostream& o, Printable& p);
};
```

Beschreibung: Dies ist die Basisklasse für alle abgeleiteten Klassen, die selbst als Objekte über die C++-Streams ausgegeben werden sollen. Die Ausgabe gehorcht dabei einem einheitlichen Schema, wo erst der Klassenname und dann einheitlich Formatiert der Inhalt der Klasse ausgegeben wird.

Literaturverzeichnis

[Bö90] Gert Böhme. *Algebra: Anwendungsorientierte Mathematik*. Springer-Verlag, 1990.

[Bot80] S. Bottini. An algebraic model of an assosciative noise-like coding memory. *Biol. Cybernetics*, 36:221–228, 1980.

[Bra91] Rüdiger Brause. *Neuronale Netze*. B.G. Teubner Stuttgart, 1991.

[Cha90] D. J. Chalmers. Syntactic transformations on distributed representations. *Connection Science*, 2, 1990.

[Dor91] Georg Dorffner. *Konnektionismus : von neuronalen Netzwerken zu einer „natürlichen“ KI*. B.G.Teubner Stuttgart, 1991.

[Elm90] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[Elm91] J. L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–224, 1991.

[Fri93] B. Fritzke. Growing cell structures – a self–organized network for unsupervised and supervised learning. Technical Report TR–93–026, ICSI, 1993.

[GK95] Christoph Goller and Andreas Küchler. Learning task-dependent distributed representation by backpropagation through structure. NIPS, 1995.

[Hin90] G. F. Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46:47–75, 1990. ICSI.

[Kal94] Y. Kalinke. Ein massiv paralleles Berechnungsmodell für normale logische Programme. Diplomarbeit, TU Dresden, 1994.

[KG96] Andreas Küchler and Christoph Goller. How structure-driven recurrent neural networks could be utilized for inductiv learning in symbolic domains, 1996.

[Leh96] Helko Lehmann. Untersuchungen zur Dynamik rekurrenter neuronaler Netze. Großer Beleg, TU Dresden, 1996.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

[NKK94] Detlef Nauck, Frank Klawonn, and Rudolf Kruse. *Neuronale Netze und Fuzzy-Systeme*. Vieweg-Verlag, 1994.

[Pla93] T. A. Plate. Holographic recurrent networks. In C. L. Giles, S. J. Hannson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5*, San Mateo, CA, 1993. NIPS*92, Morgan Kaufmann.

[Pla94] T. A. Plate. *Distributed Representations and Nested Compositional Structure.* PhD thesis, Department of Computer Science, University of Toronto, 1994.

[Pla95] T. A. Plate. Distributed representations for nested compositional structure. In R. Sun and F. Alexandre, editors, *Working Notes of the IJCAI-95 Workshop on Connectionist–Symbolic Integration: From Unified to hybrid Approaches*, pages 69–74, 1995.

[Pol90] J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46:77–105, 1990.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagation errors. *Nature*, 323:533–536, 1986.

[Roj93] Raúl Rojas. *Theorie der neuronalen Netze.* Springer, 1993.

[RSM91] Helge Ritter, Klaus Schulten, and Thomas Martinez. *Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke.* Addison Wesley, 1991.

[SA93] L. Shastri and V. Ajjanagadde. From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioural and Brain Sciences*, 16(3):417–494, September 1993.

[Sha88] L. Shastri. *Semantic Networks: An Evidential Formalization and its Connectionist Realization.* Research notes in Artificial Intelligence. Pitman, London, 1988.

[Sie93] H. T. Siegelmann. *Foundations of recurrent neural networks.* PhD thesis, Graduate School – New Brunswick, 1993.

[Smo90] P. Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46:159–216, 1990.

[Spe92] A. Sperduti. Labeling RAAM. Technical Report TR-93-029, International Computer Science Institute, Berkeley, CA, 1992.

[SS95] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. Technical Report TR-16/95, Dipartimento di Informatica, Università di Pisa, 1995.

[SSG95] A. Sperduti, A. Starita, and C. Goller. Learning distributed representations for the classification of terms. In C. S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 509–515. Morgan Kaufmann CA, 1995.

[SW92] A. Stolcke and D. Wu. Tree matching with recursive distributed representations. Technical Report TR-92-025, International Computer Science Institute, Berkeley, CA, 1992.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze*. Addison-Wesley, 1994.

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und Zitate kenntlich gemacht habe.

Torsten Seiler
Dresden, den 31.7.1996