Vorstrukturierte Verallgemeinerte Rekurrente Neuronale Netze

Till Brychcy*

20. Oktober 1997

Zusammenfassung

Eine verallgemeinerte Klasse getaktet laufender rekurrenter neuronaler Netze wird vorgestellt. Durch Verwendung beliebiger multivariater, differenzierbarer Aktivierungsfunktionen können rekurrente Netze so vorstruktiert werden, daß dynamische Systeme damit effizient modelliert und identifiziert werden können. Mittels Darstellung in Matrixnotation wird eine sehr übersichtliche Form für die Lernregeln gewonnen. Es werden Hinweise zur effizienten Implementierung unter Verwendung dünn besetzter Matrizen gegeben.

1 Einführung

In der Regelungstechnik werden Blockdiagramme zur Darstellung von dynamischen Systemen verwendet. Jeder Block eines solchen Diagrammes entspricht dabei einem Teilsystem, welches mathematisch gesehen einen Operator darstellt: Es bildet also Eingabefunktionen (der Zeit) auf Ausgabefunktionen (der Zeit) ab, wobei natürlich gewisse Stetigkeits- und Kausalitätbedingungen gelten. Sind diese Operatoren linear und zeitinvariant (also nicht explizit von der Zeit abhängig: eine zeitlich verschobenen Eingangsfunktion führt zur entsprechenden zeitlich verschobenen Ausgangsfunktion), so lassen sich diese gut mit der Methode der Laplacetransformierten behandeln-Allerdings sind die Operatoren oft nur bis auf Parameter bekannt. Schwieriger wird es bei nichtlinearen Operatoren: Man versucht, solche Operatoren in einen linearen, dynamischen und einen nichtlinearen, statischen Teil zu zerlegen und die somit zustandsfreien Nichtlinearitäten mit Messungen zu identifizieren.

Hier bietet sich für verschiedene Aufgabenstellungen der Einsatz (künstlicher) neuronaler Netze an. Zunächst eignen sich die Netze zur Darstellung der sta-

*Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, Wissenschaft, Forschung und Technologie unter dem Förderkennzeichen 01IN510C8 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

tischen Nichtlinearitäten. Sofern direkt vor und hinter der Nichtlinearität gemessen werden kann, kann ein Feedforward-Netz als generalisierender Funktionsapproximator eingesetzt werden, was sich in schwierigeren Fällen als vorteilhaft gegenüber anderen Approximationsverfahren wie Kennlinienfelder erweisen kann. Oft hat man aber nicht so viel Glück: Das Verhalten der Nichtlinerität ist oft nur durch lineare Operatoren hinweg meßbar oder die Nichtlinearität von der Dynamik abgetrennt werden.

Rekurrente Neuronale Netze bieten die Möglichkeit, auch dynamische Zusammenhänge zu modellieren. Man kann zunächst versuchen, das komplette System als zu modellieren. Aus Sicht der Regelungstechnik ist dieser Ansatz jedoch relativ uninteressant, da weder das Wissen über das System eingebracht werden kann, noch das resultierende Netz interpretiert werden kann, wodurch Zuverlässigkeitsaussagen schwierig bis unmöglich werden. Darüber hinaus ist die Trainingskomplexität für rekurrente Netze in jedem Lernschritt linear in dem Produkt aus der Anzahl der Neurone und der Anzahl der trainierbaren Gewichte-Netze mit vielen trainierbaren Gewichten wie z.B. vollverbundene Netze scheiden daher aus Performanzgründen sofort aus.

Eine Möglichkeit, diese Probleme zu meistern, besteht darin, stark vorstrukturierte Netze zu verwenden. Der bekannte Teil des Blockdiagramms wird in ein rekurrentes neuronales Netz mit festen Gewichten „kodiert“, für die unbekannten, zu identifizierenden Systemteile und Parameter werden Neurone und Gewichte mit veränderlichen Gewichten an den passenden Stellen eingebaut.

Hierzu reichen allerdings die typischerweise in neuronalen Netzen verwendeten einfachen Aktivierungsfunktionen wie die Sigmoide oder die Identität nicht aus. Beispielsweise können auch Produkte von Größen oder sogar Funktionen von drei und mehr Eingaben auftreten. Wir stellen daher im folgenden eine entsprechend verallgemeinerte Klasse von Netzen vor und zeigen, wie beliebige getaktet laufende Netze in dieser Klasse enthalten sind. Durch geschickte Darstellung in Matrixschreibweise kann die Lernregel für die Gewichte in eine sehr einfache Form als Produkt von Matrizen gebracht werden. Beim Lernen in rekurrenten Netzen ergibt sich allerdings das Problem, das Netzes in einen konsistenten Anfangszustand zu bringen. Dies läßt sich durch entsprechende Optimierungsschritte erreichen. Im weiteren stellen wir Details zur effizienten Implementierung der Verfahren unter Verwendung von dünn besetzten Matrizen dar. Zuletzt werden einige Beispiele gezeigt.

2 Eine Klasse verallgemeinerter rekurrenter neuronaler Netze

Um eine sehr übersichtliche Darstellung der Lernregeln zu gewinnen, müssen wir zunächst einige Bezeichnungen geschickt wählen.

Wir betrachten ein Neuronales Netz mit einer Gesamtanzahl von N Neuronen. Diese Neurone wollen wir durch ganze Zahlen zwischen 1 und N identifizieren.

Jedes dieser Neurone hat zu jedem Zeitpunkt t eine Aktivierung, welche wir als $x_i(t)$ bezeichnen wollen. Die Gesamtheit dieser Aktivierungen bildet den Aktivierungsvektor $x(t)$. Die Netze werden nicht kontinuierlich laufen, sondern in einem diskreten Takt, weshalb die Zeitpunkte t durch ganze Zahlen repräsentiert werden. Ein gewisser Teil von Ne entsprechend markierten Neuronen stellt Eingabeneuronen dar. Die Aktivierung dieser Eingabeneurone wird von aussen vorgegeben. Innerhalb der Netze darf es deshalb keine Verbindungen zu diesen Neuronen hin geben. Die Aktivierung dieser Neurone ist also eine Funktion der Zeit $x_i(t) = f_i(t)$.

Die Ausgabe des Netzes wird durch N_a entsprechend markierte Ausgabeneurone bestimmt. Das Netz bildet dann also eine N_e-dimensionale Eingabetrajektorie auf eine Na-dimensionale Ausgabetrajektorie ab.

Die gewichteten Verbindungen der Neurone werden wir einfach als Gewichte bezeichnen. Wenn es eine Verbindung von Neuron j zum Neuron i gibt, dann soll $w_{i,j}$ deren Gewicht bezeichnen. Der erste Index gibt also das Ziel der Verbindungen an, was die Notation in Matrixschreibweise erleichtert. Nicht jedes Gewicht muß trainierbar sein: Es wird zu jedem Gewicht markiert, ob es trainiert werden darf. Nicht jedes Neuron muß mit jedem anderen verbunden sein; um jedoch den Formalismus einfach zu gestalten, werden wir so tun als wäre jedes mit jedem verbunden (sogar mit Eingabeneuronen!) und die Gewichte von eigentlich nicht existenten Verbindungen haben den Wert Null und sind als nicht trainierbar gekennzeichnet. Die Gesamtheit der Verbindungen läßt sich dann durch die Verbindungsmatrix W beschreiben, deren Elemente genau die $w_{i,j}$ sind (wobei der erste Index die Zeile und der zweite die Spalte angibt). Die Gesamtheit der Aktivierungen, die ein Neuron i zu einem Zeitschritt t anregen, bezeichen wir mit $s_i(t)$ und sie berechnet sich dann als:

$$s_i(t) = \sum_{j=1}^{N} w_{i,j}\, x_j(t-1) \tag{1}$$

Den Gesamtvektor dieser Anregungen bezeichnen wir als $s(t)$ und wir können Schreiben: $s(t) = Wx(t-1)$.

Zur Darstellung von Blockdiagrammen als Neuronale Netze werden Aktivierungsfunktionen benötigt, die mehr als nur eine Eingabe haben. Um nun formal nicht für jedes Neuron unterschiedliche Eingänge berücksichtigen zu müssen, greifen wir zu folgender Darstellung: Die Berechnung der neuen Aktivierung $x_i(t)$ jedes Neurons i darf nicht nur von der eigenen Anregung $s_i(t)$ abhängen, sondern auf den gesamten Aktivierungsvektor $s(t)$ zugreifen.

Statt also für jedes Neuron wie gewöhnlich eine Aktivierungsfunktion zu definieren, welche die neue Aktivierung des Neurons als $x_i(t) = f_i(s_i(t))$ berechnet und $\mathbb{R}$ auf $\mathbb{R}$ abbildet, gilt für die neue Aktivierung $x_i(t) = f_i(s(t))$ und diese Funktion bildet $\mathbb{R}^N$ auf $\mathbb{R}$ ab (wobei sie gewöhnlich die meisten Eingabekomponenten bis auf $s_i(t)$ ignorieren wird). Wenn wir den Vektor aller Aktivierungsfunktionen

$f_i(x)$ und $f_i(t)$ (für die Eingabeneurone) zu einer Gesamtaktivierungsfunktion $f(x,t)$ zusammenfassen, welche $\mathbb{R}^N + 1$ auf $\mathbb{R}^N$ abbildet, können wir schreiben:

$$x(t) = f(s(t), t) = f(Wx(t-1), t) \tag{2}$$

Hierbei gilt die Nebenbedinung, daß jede x-Komponente nur entweder nur von dem expliziten t-Parameter abhängt gar nicht von ihm abhängt. Will man also ein Neuron realisieren, dessen Aktivierungsfunktion k Eingaben hat (und beispielsweise das Produkt dieser Eingaben berechnet), so fügt man $(k-1)$ zusätzliche Platzhalter-neurone hinzu, zu denen nur Verbindungen hinführen, aber keine wegführen, und greift in der Aktivierungsfunktion des Neurons auf die $s_k(t)$-Werte dieser Platzhalterneurone zu. Diese Darstellung macht es auch unnötig, den Neuronen zugeordnete interne trainierbare Parameter der Aktivierungsfunktion zuzulassen, da diese als Gewichte von konstant aktivierten Biasneuronen zu einer zusätzlichen Eingabe des Neurons realisiert werden können. Beispielsweise läßt sich ein RBF-Neuron in einem n-dimensionalen Datenraum als Neuron mit $2n$ Eingaben repräsentieren: Die ersten n Eingaben sind mit einem Biasneuron mit konstanter Aktivierung 1 verbunden und die Gewichte zu diesen Verbindungen enthalten die Position des RBF-Neurons. Die anderen n Eingaben liefern die aktuelle Eingabe des RBF-Neurons und haben Gewichte mit dem Wert 1.

Da in jedem Zeitschritt jede Aktivierung nur auf die direkt verbundenen Neurone Einfluß hat, sind gewöhnlich mehrere Verarbeitungschritte nötig, bis eine Eingabe von den Eingabeneuronen einen Einfluß auf die Ausgabeneurone hat. Deshalb macht man in rekurrenten Netzen gewöhnlich mehrere „interne“ Zeitschritte für jeden „externen“ Zeitschritt, hält währenddessen die Eingabeneurone konstant und berücksichtigt nur die Ausgabe des Netzes im jeweils letzten internen Zeitschritt. Damit sind auch normale Feedforwardnetze in dieser Klasse darstellbar, wobei als Anzahl der internen Schritte der längste mögliche Weg im Netze zu wählen ist. Durch geschicketes Umsortieren der Neurone kann natürlich im Feedforwardfall dass Ergebnis dieser internen Iteration in einem einzigen Durchgang berechnet werden (ebenso für Lernregeln).

Auch Netze mit unterschiedlichen Verarbeitungsgeschwindigkeiten in verschiedenen Netzteilen lassen sich realisieren. Die Vorgehensweise hierzu ist, eine Verbindung, die langsamer laufen soll, in ein Zwischenneuron mit identischer Funktion als Aktivierungsfunktion münden zu lassen und die Ausgabe dieses Neurons mit dem nicht trainierbaren Gewicht 1 mit dem ursprünglichen Ziel der Verbindung zu verbinden. Will man einen Netzteil schneller laufen lassen, so macht man alle anderen Netzteile auf diese Weise langsamer und macht mehr interne Netzschritte für jede externe Eingabe.

3 Lernregeln für Gewichte und Anfangszustand

Das Netz soll beim Training seine trainierbaren Gewichte so verändern, daß es die Lernbeispiele möglichst gut reproduzieren kann. Im Fall von rekurrenten

Netzen ist jedoch zu beachten, daß die Ausgabe des Netzes nicht nur von der Eingabe, sondern auch vom Zustand der Neurone am Anfang der Eingabesequenz abhängt. Wir betrachten zunächst die Lernregel für die Gewichte, dann die Optimierung des Anfangszustandes.

3.1 Gewichte

Wir behandeln hier nur die Berechnung der partiellen Ableitung der Ausgabeneurone nach den Gewichten. Welche Fehlerfunktion dann genau verwendet wird, und ob dann Gradientenabstieg gemacht wird oder spezielle Varianten wie Rprop eingesetzt werden, ist dann beliebig wählbar.

Zur notationellen Vereinfachung fassen wir alle *trainierbaren* Gewichte $w_{i,j}$ der Gewichtsmatrix zu einem Vektor w mit Komponenten w_k zusammen. Die Anzahl dieser trainierbaren Gewichte wollen wir mit N_t bezeichnen. Um auszudrücken, daß k der Index desjenigen trainierbaren Gewichts ist, mit dem die Ausgabe des Neurons j mit der Eingabe des Neurons i verbunden ist, schreiben wir $w.z(k) = i$ und $w.s(k) = j$.

Zur Berechnung der partiellen Ableitung der Ausgabeneurone nach den trainierbaren Gewichten gehen wir im Prinzip wie bei dem als *RTRL* bekanntgemachten Ansatz (siehe [1])vor, wobei wir durch Verwendung von Matrixnotationen eine sehr übersichtliche Darstellung erreichen, welche auch mit Aktivierungsfunktionen mit mehreren Eingaben noch gültig ist.

Es sei $p_{i,j}(t) := \frac{\partial x_i(t)}{\partial w_j}$ und $P(t)$ die Matrix aus $\mathbb{R}^{N \times N_t}$, welche aus allen $p_{i,j}(t)$ gebildet wird. Die $p_{i,j}(t)$ berechnen sich als

$$p_{i,j}(t) = \frac{\partial x_i(t)}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial f_i(s(t),t)}{\partial s_k(t)} \frac{\partial s_k(t)}{\partial w_j} \tag{3}$$

mit

$$\frac{\partial s_k(t)}{\partial w_j} = \sum_{l=1}^{N} w_{k,l}\, p_{l,j}(t-1) + \delta_{w.z(j),k}\, x_{w.s(j)}(t-1) \tag{4}$$

unter Verwendung des Kroneckerdeltas $\delta_{i,j}$, welches 1 ist, wenn die beiden Indizes gleich sind und sonst 0. Wir definieren uns zwei Hilfsmatrizen $F(t)$ mit

$$f_{i,k}(t) := \frac{\partial f_i(s(t),t)}{\partial s_k(t)} \tag{5}$$

und $D(t)$ mit

$$d_{k,j}(t) := \delta_{w.z(j),k}\, x_{w.s(j)}(t-1) \tag{6}$$

Dann lassen sich die Gleichungen für die $p_{i,j}(t)$ zusammenfassen als:

$$P(t) = F(t)(WP(t-1) + D(t)) \tag{7}$$

Da die Netze gewöhnlich intern öfter laufen als externe Lernbeispiele angelegt werden, wird $P(t)$ gar nicht für jedes t benötigt. Typischerweise gibt es wesentlich mehr trainierbaren Gewichte als Neurone. In diesem Fall ist es effizierter, $P(t)$ direkt aus dem letzten bekannten $P(k)$ zu berechnen, indem man P(t-1) und dann P(t-2) usw. in Gleichung 7 rekursiv einsetzt:

$$\begin{aligned} P(t) &= F(t)\left[D(t-1) + WP(t-1)\right] \\ &= F(t)D(t-1) + F(t)WP(t-1) \\ &= F(t)D(t-1) + F(t)W\left[F(t-1)D(t-2) + F(t-1)WP(t-2)\right] \\ &= F(t)D(t-1) + F(t)WF(t-1)D(t-2) + F(t)WF(t-1)WP(t-2) \\ &\dots \end{aligned} \tag{8}$$

Sei $P(k)$ bekannt und definieren wir $Z(t-1) := F(t)$ und $Z(t-i-1) := Z(t-i)WF(t-i)$ für $i = 1..k-1$. Dann ist:

$$\begin{aligned} P(t) &= Z(t-1)D(t-1) + Z(t-2)D(t-2) + ... + Z(k)D(k) + Z(k)WP(k) \\ &= Z(k)WP(k) + \sum_{i=k}^{t-1} Z(i)D(i) \end{aligned} \tag{9}$$

Die Matrix $P(0)$ (am Anfang der Trajektorie) ist die Nullmatrix, da zu diesem Zeitpunkt die Gewichte noch keinen Einfluß auf die Aktivierungen hatten.

Diese Expansion ist im Prinzip eine Matrixdarstellung der als Kombination von RTRL und BPTT bekannt gewordenen Verfahren angewandt auf die verallgemeinerten Netze.

3.2 Anfangszustand

Da rekurrente Netze einen internen Zustand besitzen, hängt ihre Ausgabe nicht nur von der Eingabetrajektorie ab, sondern auch vom Anfangszustand. Wenn das Netze bereits vorher trainiert wurde und die Gewichte somit bereits sinnvolle Werte haben, ist es sinnvoll, zunächst den Anfangszustand des Netzes zu optimieren und dann die Gewichte zu trainieren, um ein Verlernen zu vermeiden. Die zu optimierende Größe ist also der Fehler der Netzausgabe während einer k Schritte langen Trajektorie (wobei nicht in jedem Zeitschritt ein Fehlerterm beigetragen werden muß), und die hierzu zu bestimmende Größe ist der Anfangszustand $x(0)$ welchen wir in diesem Abschnitt einfach als x bezeichnen werden: x_i ist also der Anfangszustand des Neurons mit der Nummer i.

Es ist also für alle Zeitpunkte t, an denen ein Sollwert vorgegeben ist, die Ableitung der Ausgabeneurone nach dem Anfangszustand zu berechnen. Bezeichnen wir mit $Q(t)$ die Jakobimatrix für die Ableitung der aktuellen Aktivierung $x(t)$ nach der Anfangsaktivierung x. Für die Elemente $q_{i,j}(t)$ von $Q(t)$ erhalten wir durch ableiten von 2 für $t > 0$:

$$q_{i,j}(t) = \frac{\partial x_i(t)}{\partial x_j} = \sum_{k=1}^{N} \frac{\partial f_i(s(t),t)}{\partial s_k(t)} \frac{\partial s_k(t)}{\partial x_j} \tag{10}$$

Für $t = 0$ gilt $q_{i,j}(0) = \delta_{i,j}$ bzw. $Q(0) = E_N$ (die N-dimensionale Einheitsmatrix). Durch berechnen der Ableitung von s als

$$\frac{\partial s_k(t)}{\partial x_j} = \sum_{l=1}^{N} w_{k,l}\, q_{l,j}(t-1)$$

erhalten wir in Matrixschreibweise unter Verwendung der in Gleichung 5 definierten $F(t)$-Matrizen:

$$Q(t) = F(t)WQ(t-1) \tag{11}$$

Diese Werte können in einer einfachen Vorwärtsiteration berechnet werden.

Der Fehler E der bisher nicht näher spezifizierten Fehlerfunktion ist eine von Funktion von x(0) bis x(t). Für die Ableitung des Fehler nach dem Anfangszustand x_i des Neurons i ergibt sich dann:

$$\frac{\partial E}{\partial x_i} = \sum_{\tau=0}^{t} \sum_{k=1}^{N} \frac{\partial E}{\partial x_k(\tau)} Q_{k,i}(\tau)$$

3.3 Komplexitätbetrachtungen und Details zur effizienten Implementierung

3.3.1 Dünn besetzte Matrizen

Die Grundlage für eine effiziente Implementierung besteht in einer geeigneten Datenstruktur für dünn besetzte Matrizen. Die Matrizen W, $F(t)$ und insbesondere $D(t)$ enthalten gewöhnlich überwiegend Elemente, die unveränderlich gleich Null sind. Einerseits um Platz zu sparen, aber vor allem um effizierte Algorithmen zu bekommen, ist es sinnvoll, nur die Elemente abzuspeichern, die ungleich Null sind. Eine geeignete Datenstruktur besteht aus folgenden Teilen:

1. Eine Tabelle mit Tripeln (Matrixelement, Zeilennr, Spaltennr). Gegebenenfalls kommen noch weitere Datenelemente hinzu in denen z.B. vermerkt wird, ob ein Gewicht beim Training verändert werden darf.

2. Für jede Spalte s eine Liste mit den Nummern der Tabellenelemente, deren Spaltennr gleich s ist als *Spaltenindex*.

3. Für jede Zeile z eine Liste mit den Nummern der Tabellenelemente, deren Zeilennr gleich z ist als *Zeilenindex*.

Die Zeilen- und Spaltenindexe werden dabei für effizientes Einfügen und Löschen von Matrixelementen und für manche Berechnungen, wie etwa der Berechnung des Produktes von zwei dünn besetzten Matrizen verwendet. Für eine Matrix mit m Zeilen und n Spalten kann ein beliebiges Element dann in $O(\min(m, n))$ geändert oder ggf. ein neues eingefügt werden (wenn sicher ist, daß es das Element noch nicht gibt und bei geschickter Speicherverwaltung kann ein neues Element in $O(1)$ hinzugefügt werden). Für Matrix-Vektorprodukte und Matrix-Matrixprodukte ist über die Tabellenelemente der dünn besetzten Matrix zu iterieren und alle Terme, auf die dieses Matrixelement Einfluß hat, abzuarbeiten. Beispielsweise für eine Gewichtsmatrix W mit Elementen $w_{z,s}$ ergibt sich für das Produkt $s = Wx$ der Algorithmus:

```
s:=0;
Für jedes Tabellenelemente t von W mit der Form (t.w, t.z, t.s):
  s[t.z]:=s[t.z]+t.w*x[t.s];
```

Die Komplexität dieses Algorithmus ist $O(N_W)$, also linear in der Anzahl der Matrixelemente von W. Durch Anwenden auf die Zeilen- oder Spaltenvektoren einer voll besetzten Matrix ergibt sich entsprechend ein Matrixmultiplikationsalgorithmus der Komplexität $O(N_W N_z)$, wobei N_z die Anzahl der Zeilen bzw. Spalten der anderen Matrix sei. Da mittels der Zeilen- bzw. Spaltenindexe auf alle Elemente einer Zeile ohne zusätzlichen Aufwand zugegriffen werden kann, ist auch die Multiplikation von zwei dünn besetzten Matrizen in dieser Komplexitätsordnung möglich. Insbesondere im Fall der $F(t)$-Matrizen ist für gewöhnlich die Anzahl N_F der Matrixelemente ungleich Null linear in der Anzahl N der Neurone. Daher ergibt sich somit eine Komplexitätsverbesserung auf $O(N^2)$ gegenüber $O(N^3)$ für die Matrixmultiplikation mit voll besetzen Matrizen. Die Verbindungsmatrix W, sowie die Matrizen $F(t)$ sollten als dünn besetzte Matrizen wie hier beschrieben werden realisiert werden. Bei den $F(t)$ ist zu berücksichtigen, daß alle $F(t)$ die gleiche Struktur haben und die $F(t)$ nie aktualisiert, sondern stets komplett neu erzeugt werden, was eine effiziente Speicherverwaltung für diese Datenstrukturen möglich macht. Ein Sonderfall sind die $D(t)$-Matrizen, welche sich nach einem festen Schema aus $x(t-1)$ ergeben. Da alle Informationen über diese Matrizen in $x(t-1)$ und der Struktur des Netzes stecken, lohnt es sich, $D(t)$ nicht explizit aufzubauen sondern die Multiplikation von $Z(t)$ mit $D(t)$ in einem Spezialalgorithmus direkt durch Zugriffe auf $x(t-1)$ und W zu realisieren.

4 Symbolverzeichnis

Es werden die folgenden Bezeichnungen verwendet:

Symbol	Bereich	Beschreibung
$\mathbb{N}$	Menge	Die natürlichen Zahlen
$\mathbb{R}$	Menge	Die reellen Zahlen
N	$\mathbb{N}$	Die Anzahl der Neurone.
N_e	$1..N$	Anzahl der Eingabeneurone
N_a	$1..N$	Anzahl der Ausgabeneurone
N_W	$1..N^2$	Anzahl der Verbindungen.
t	$\mathbb{N}$	Zeitindex
$x(t)$	$\mathbb{R}^N$	Vektor der Aktivierungen zum Zeitpunkt t.
$x_i(t)$	$\mathbb{R}$	i-te Komponente von $x(t)$. Entsprechend auch für andere Vektoren
$s(t)$	$\mathbb{R}^N$	Vektor der Erregungen der Neurone. $x(t+1) = f(s(t+1), t), s(t+1) = Wx(t)$
N_t	$1..N_W$	Anzahl der trainierbaren Gewichte
w	$\mathbb{R}^{N_t}$	Zu trainierbaren Verbindungen gehörige Gewichte als Vektor.
$w.z(i)$	$1..N$	Index des zur trainierbaren Verbindung i gehörigen postsynaptischen Neurons
$w.s(i)$	$1..N$	Index des zur trainierbaren Verbindung i gehörigen präsynaptischen Neurons
W	$\mathbb{R}^{N\times N}$	Die Gewichte dargestellt als Gewichtsmatrix
$w_{i,j}$	$\mathbb{R}$	Element der Gewichtsmatrix: Gewicht vom Neuron j zum Neurons i. Zwischen Gewichtsvektor und Gewichtsmatrix besteht also folgender Zusammenhang: Für $i = 1..N_t$ gilt: $w_{w.z(i),w.z(i)} \equiv w_i$ Alle anderen Elemente der Gewichtsmatrix sind 0 oder bezeichnen feste Verbindungen.
$f(x,t)$	$\mathbb{R}^{N+1} \to \mathbb{R}^N$	Gesamt-Aktivierungsfunktion. Jede Komponente $f_i(x,t)$ darf entweder nur von t abhängen (Eingabeneuron) oder nur von x (andere Neurone), aber nicht beides.
$P(t)$	$\mathbb{R}^{N\times N_t}$	Jakobimatrix der partiellen Ableitungen nach den trainierbaren Gewichten zum Zeitpunkt t

Danksagung

Mein Dank für Unterstützung und anregende Diskussionen geht an Prof. W. Brauer, Michael Sturm, Clemens Kirchmair und unsere Projektpartner im BMBF-Projekt ACON, insbesondere die Gruppe bei Prof. D. Schröder.

Literatur

[1] R.J. Williams and D. Zipser: *Gradient-Based Learning Algorithms For Recurrent Connectionist Networks.* Tech. Rep. NU-CSS-90-9, College of Computer Science, Northeastern University, Boston, MA, 1990