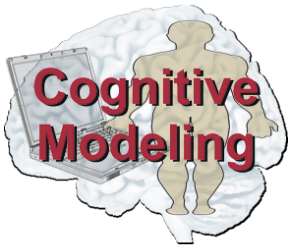
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Technical Report
No. COBOSLAB_Y2013_N001
21. November 2013

RNNPBlib 1.1 - Recurrent Neural Networks with Parametric Biases Library

Deutsche Dokumentation

Fabian Schrodt



Wilhelm-Schickard-Institut für Informatik
University of Tübingen
Sand 14
72076 Tübingen, Germany
http://www.cm.inf.uni-tuebingen.de

RNNPBlib 1.1 - Recurrent Neural Networks with Parametric Biases Library

Fabian Schrodt*

Kurzfassung

Diese Dokumentation dient sowohl dem Verständnis für neuronale Lernverfahren als auch der technischen Beschreibung des RNNPBlib-Frameworks. RNNPBlib (Recurrent Neural Networks with Parametric Biases Library) ist ein quelloffenes Framework für das Erstellen und Untersuchen modulierbarer Varianten bedeutender neuronaler Architekturen. Das Framework ist im Rahmen der Masterthese von Fabian Schrodt („Imitation von Verhalten mittels neuronaler Netze“) entstanden und verfolgt die Zielsetzung, öffentliche Algorithmen und Implementierungen bereitzustellen, welche es ermöglichen, modulierbare neuronale Netze auf einfache Weise zu erstellen und anzuwenden. Es ist für Windows/Linux 32/64 Bit unter GPL 3.0 verfügbar. Eine Kopie der Lizenz befindet sich im Anhang dieses Dokuments.

Diese Dokumentation ist wie folgt gegliedert: Zunächst wird in Kapitel 1 eine Erläuterung überwachter neuronaler Netze und speziell ihrer modulierbaren Verwandten sowie entsprechender Lernverfahren gegeben. Im Zuge dessen werden allgemeine Lernregeln hergeleitet, die es erlauben, diese Netze und alle Kombinationen daraus mit einem einheitlichen Lernverfahren zu trainieren. Es werden bekannte und neue Methoden zur Modulierung neuronaler Netze und Beschleunigung der Konvergenz erklärt.

Kapitel 2 skizziert die technische Umsetzung und Implementierung der vorangegangenen Aspekte in RNNPBlib und beschreibt die wesentlichen Methoden für deren Anwendung. Ebenso werden die Optimierungen bezüglich der Laufzeit dargelegt und Hinweise für die Erweiterung des Frameworks gegeben.

*FSchrodt@gmx.de

1 Modulierte neuronale Netze

1.1 Überblick

Dieses Kapitel erklärt die allgemeinen Grundlagen mehrschichtiger überwachter neuronaler Netze mit und ohne Rekurrenz, beschreibt Netzwerke mit radialen Basisfunktionen und führt insbesondere in Netzwerke ein, deren Aktorik anhand exogener Stimuli moduliert werden kann. Es dient als theoretische Grundlage für das Verständnis des in Kapitel 2 vorgestellten RNNPBlib-Frameworks.

Überwachte künstliche neuronale Netze (im Folgenden NN) sind ein biologisch inspirierter Ansatz für die Approximation beliebiger Funktionen. Allgemein werden sie aus abstrahierten Neuronen aufgebaut und anhand gerichteter Verbindungen verknüpft, welche als Gewichte bezeichnet werden. Die Topologie eines NN entspricht prinzipiell der eines beliebigen gerichteten Graphen, wobei Neurone die Knoten und Gewichte die Kanten des Graphen dargestellen. Neuronen eines NN werden zur Vereinfachung der topologischen Darstellung in Schichten organisiert, wobei man zwischen Eingabeschicht (Funktionsvariablen), verdeckten Schichten (Struktur der Funktion) und Ausgabeschicht (Funktionsbild) unterscheidet. Sowohl Neuronen als auch Gewichte weisen in der Regel eine bestimmte parametrisierte Funktionalität auf, die eigehende Informationen transformiert und weiterreicht. Gemäß der Topologie eines NN können daher Informationen von der Eingangsschicht zur Ausgabeschicht propagiert werden, so dass ein Funktionsbild entsteht. Die Anpassung dieses Funktionsbildes ist anhand der Adaption der Netz-Parametrisierung und/oder -Topologie möglich und erfolgt in der Regel durch Rückpropagierung der Abweichung von der zu approximierenden Zielfunktion.

Rein vorwärtsgerichtete, nicht-rekurrente NN – in diesem Sinne Graphen ohne Zyklen – können unter der Voraussetzung der Existenz beliebig vieler verdeckter Neurone jede wohldefinierte, endliche und surjektive Funktion approximieren. Sie können jedoch keine dynamischen Funktionen approximieren, da diese eine implizite Abhängigkeit von der Reihenfolge der Funktionswertberechnung (Historie) aufweisen. Rekurrente neuronale Netze (im Folgenden: RNN) weisen dagegen topologische Zyklen auf, welche nach dem Prinzip der dynamischen Programmierung aufgelöst werden können. Dies bedeutet, dass die durch ein RNN propagierten Informationen zur Berechenbarkeit mindestens an einem Element jedes Zyklus zeitlich verzögert werden müssen, so dass sich ein berechenbares Netzwerk mit einem internen, von der Historie der Eingaben abhängigen Zustand ergibt. Man spricht in diesem Zusammenhang auch vom Zustandsraum eines RNN. Durch diese Eigenschaft ist mittels RNN die Approximation (super) Turing-berechenbarer Funktionen möglich.

Jedoch gilt für beide Formen (nicht-probabilisitischer) neuronaler Netze

die Einschränkung, dass lediglich *eine* Funktion approximiert werden kann: Unter gleichen Voraussetzungen, d.h. bei gleicher Eingabefolge ergibt sich immer die gleiche, deterministische Ausgabefolge. Im Gehirn können dagegen verschiedenartige Funktionen durch gegenseitige Modulation neuronaler Strukturen entstehen (siehe zum Beispiel kontext-abhängiges Greifen [1] oder kontextuelle Modulation der Amygdala [2, 3]), selbst wenn die vorausgegangene sensorische Situation nicht unterscheidbar ist. Da der Grad der Abstraktion neuronaler Modelle über deren biologische Plausibilität entscheidet, ist es angebracht, Modelle zu untersuchen, die diese Eigenschaft teilen. Im Verlauf dieses Kapitels wird daher nach Klarstellung aller benötigten Grundlagen beschrieben, wie NN mit und ohne Rekurrenz durch gezielte Modulation einer Teilfunktionalität artverwandte Funktionen approximieren können.

Das Kapitel ist wie folgt strukturiert: Zunächst werden die Grundelemente (Neuronen und Gewichte) neuronaler Netze und deren Funktionalitäten erläutert. Nachfolgend wird beschrieben, wie die Parameter der beschriebenen Komponenten zur Approximation einer Funktion gelernt werden können. Dabei wird eine Notation verwendet, die es erlaubt, allgemeine Lernregeln für Kombinationen dieser Komponenten zu formulieren. Anschließend werden Beispiele für klassische neuronale Topologien gegeben, die anhand der beschriebenen Komponenten umgesetzt werden können, bevor auf modulierbare Netzwerktopologien eingegangen und das Training entsprechender Modulatoren beschrieben wird.

1.2 Grundkomponenten eines Netzwerks

1.2.1 Neuronen

Ein abstraktes Neuron ist durch die Beobachtung motiviert, dass biologische Neuronen synaptische Aktionspotentiale aufintegrieren und nach Überschreitung eines Schwellwertes ein axonisches Aktionspotential ausstoßen („feuern“). Für die Art der daraus resultierenden Informationsübertragung existieren mehrere Theorien. Im Folgenden wird der Schwerpunkt auf neuronale Modelle gelegt, die auf der freuqenzcodierten Informationsübertragung basieren, d.h. im Gegensatz zu den spikenden neuronalen Netzen (sog. Netze dritter Generation [4]) von der These ausgehen, dass die Bedeutung neuronal propagierter Information lediglich in der mittleren Frequenz der Aktionspotentiale codiert ist, nicht dagegen in der genauen zeitlichen Abfolge derselben [5].

Ein solches künstliches Neuron mit Index j besteht im Wesentlichen aus einer *Eingabe-* und einer *Aktivierungsfunktion*: Die Eingabefunktion verrechnet alle über Gewichte zu diesem Neuron propagierten Informationen zur Netzeingabe net_j. Die Netzeingabe ist gewöhnlich durch das gewichtete

Mittel der Aktivierungen aller Vorgängerneuronen definiert:

$$\text{net}_j := \text{net}_j^{\text{MD}} = \sum_i w_{ij} \cdot o_{ij} \tag{1}$$

o_{ij} entspricht der Aktivierung des Vorgänger-Neurons i, welche zum aktuellen Zeitpunkt mit dem Gewicht w_{ij} (von Neuron i zu Neuron j) verrechnet wird. Die Wahl dieser Notation wurde getroffen, da o_{ij} vom verwendeten Gewicht abhängen kann, wie später gezeigt wird. Die Indizes i und j laufen über alle Werte, für die Gewichte w_{ij} existieren [1].

Die Aktivierungsfunktion beschreibt in der Regel eine monotone, stetig[2] differenzierbare Transformation der Netzeingabe, wobei ihr Bild in einem definierten Intervall liegen kann. Aktivierungsfunktionen können lineare, sigmoide oder Sprungfunktionen sein, was über die Schwellwerteigenschaften des Neurons entscheidet. Die Höhe eines Schwellwerts kann in künstlichen neuronalen Netzen durch ein sog. Bias-Neuron vorgegeben werden: Das Bias-Neuron ist ein Neuron mit konstanter Aktivierung (oft 1), welches je über ein veränderliches Gewicht den Schwellwert eines jeden Neurons bestimmt.

Die Linearkombination der Vorgängeraktivierungen in Gleichung 1 entspricht unter Verwendung einer schwellwertbehafteten Aktivierungsfunktion einer monotonen Diskriminierung der Eingangskomponenten, weswegen – angedeutet durch den hochgestellten Index MD – Neuronen mit dieser Eigenschaft im Folgenden auch als monotone Diskriminatoren bezeichnet werden. Die häufigsten Aktivierungsfunktionen $o_j(x)$ mit $x = \text{net}_j^{\text{MD}}$ sind in Übersichtstabelle 1 angegebenen.

Einen anderen Ansatz für die Informationsverarbeitung verfolgen Netzwerke mit radialen Basisfunktionen (RBFs). Hierbei wird jedem Neuron ein radialer Zuständigkeitsbereich zugewiesen, so dass es bei einer eindeutig bestimmten Eingangssituation die maximale Aktivierung aufweist. Diese Eingangssituation wird durch den Vektor aller auf das Neuron zeigenden Gewichte bestimmt. Man spricht in diesem Zusammenhang auch vom Zentrum oder der Basisfunkion eines radialen Neurons. Die Netzeingabe wird hier aus Gründen der Differenzierbarkeit, was die Herleitung eines einfachen überwachten Lernalgorithmus erlaubt, nicht wie gewöhnlich durch die Euklidische Distanz, sondern durch den halben quadratischen Fehler zwischen Ausgaben der Vorgängerneuronen und dem Gewichtsvektor beschrieben, was dem Abstand der Eingabe zum Zentrum entspricht:

$$\text{net}_j := \text{net}_j^{\text{RBF}} = \frac{1}{2} \sum_i (w_{ij} - o_{ij})^2 \quad . \tag{2}$$

Als Aktivierungsfunktion kann jede differenzierbare endliche Funktion gewählt werden, die die Bedingung der eindeutigen Maximalaktivität

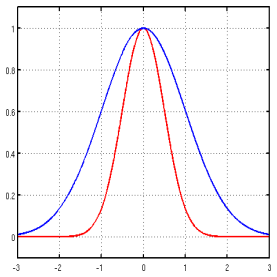
[1] Es werden keine vollständigen Gewichtsmatrizen bezeichnet, um die Konsistenz bei der Fomulierung von radialen Neuronen und Gewichten höherer Ordnung zu wahren.

[2] Bei diskreter Berechnung können Unstetigkeitsstellen vernachlässtigt werden.

erfüllen. Oftmals wird dazu eine Gauß'sche Aktivierungsfunktion gewählt, die auf das Maximum 1 normiert ist, Mittelwert 0 und Standardabweichung $1/s$ aufweist. Radiale Aktivierungsfunktionen müssen dabei nicht wie zuvor monoton sein. Eine Alternative zur Gauß-Funktion ist zum Beispiel die radial-lineare Funktion, die sich im Radius $1/s$ um das Zentrum linear verhält, wie in Tabelle 2 dargestellt. Die radial-lineare Aktivierungsfunktion ist zu bevorzugen, wenn die Eingabe eines radialen Neurons nicht verzerrt werden soll. In diesem Fall trifft ein radiales Neuron eine (beliebigdimensionale) Symmetrieannahme über seine Eingaben.

Normalisierte RBF nach Gauß:

$$o_j(x) := \exp\left(-\tfrac{1}{2}\,(x \cdot s)^2\right) \qquad \in (0,1)$$



Radial-lineare Funktion:

$$o_j(x) := \max(1 - |x| \cdot s, 0) \qquad \in [0,1]$$

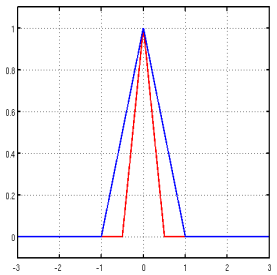
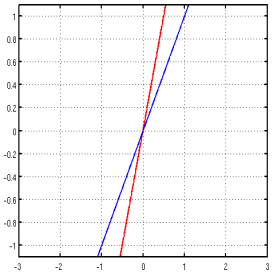


Tabelle 2: Beispiele radialer Aktivierungsfunktionen. Im Fall der **Gauß'schen RBF** entspricht die Skalierung s dem Kehrwert der Standardabweichung: $s = \frac{1}{\sigma}$. Bei der **radial-linearen Aktivierungsfunktion** entspricht der Kehrwert der Skalierung dem radialen Einzugsbereich der Funktion: $r = \frac{1}{s}$.

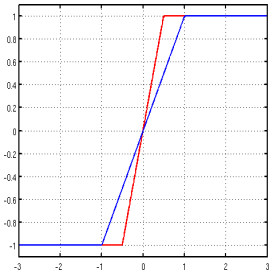
Lineare Aktiverungsfunktion:

$$o_j(x) := x \cdot s \qquad \in (-\infty, \infty)$$



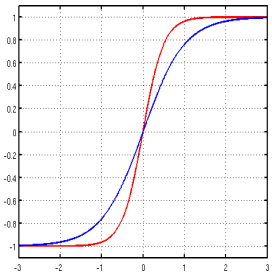
Quasi-lineare Aktivierungsfunktion:

$$o_j(x) := \begin{cases} -1 & : x \cdot s \leq -1 \\ x \cdot s & : -1 < x \cdot s < 1 \\ 1 & : x \cdot s \geq 1 \end{cases} \qquad \in [-1, 1]$$



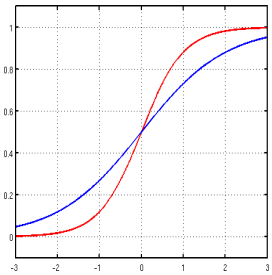
Tangens-Hyperbolicus:

$$o_j(x) := \tanh(x \cdot s) \qquad \in (-1, 1)$$



Logistische Funktion:

$$o_j(x) := \frac{1}{1+e^{-x \cdot s}} \qquad \in (0, 1)$$



Heaviside-Aktiverungsfunktion:

$$o_j(x) := \begin{cases} 0: & x \cdot s < 0 \\ 1: & x \cdot s \geq 0 \end{cases} \qquad \in [0, 1]$$

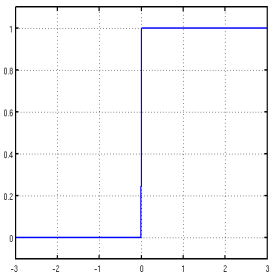


Tabelle 1: Übersichtstabelle verschiedener Aktivierungsfunktionen. Die Netzeingabe kann generell per Multiplikation mit s skaliert werden. Unter Verwendung sigmoider Aktivierungsfunktionen bestimmt s somit die Schwellwerteigenschaften der Funktion. Die **lineare Aktiverungsfunktion** weist keine Schwellwerteigenschaften auf. Sie eignet sich inbesondere für Eingabe- und Ausgabeneuronen, da sie auf kein Interval beschränkt ist und ihre Eingabe nicht verzerrt. Die **Heaviside-Aktivierungsfunktion** (nach dem britischen Mathematiker Oliver Heaviside) ist nicht monoton und nicht differenzierbar, daher kann sie ohne Absicherung bei Lernverfahren, die differenzierbare Aktivierungsfunktionen fordern zu Problemen führen. Die **quasi-lineare Aktivierungsfunktion** weist Schwellwerteigenschaften auf, ist jedoch ebenfalls nicht monoton und nicht stetig differenzierbar.

1.2.2 Gewichte

Gewichte sind in künstlichen neuronalen Netzen dem Vorbild der Synapsen des Zentralnervensystems nachempfunden. In klassischen NN sind Gewichte direktional, weisen also ein Neuron i als Vorgänger, und ein Neuron j als Nachfolger auf, wie in Abbildung 1(a) dargestellt. Ihre Funktionalität wird in diesen Architekturen allein durch einen Netzparameter w_{ij}^p beschrieben, welcher als Faktor für die über das Gewicht bei Neuron j eintreffende Ausgabe des Neurons i dient. Folglich errechnet sich die Netzeingabe nach Gleichung 1 oder Gleichung 2 mit

$$\begin{aligned} w_{ij} &:= w_{ij}^p \\ o_{ij} &:= o_i(t) \end{aligned} \quad .$$

Die Rückwärtspropagierung von Fehlerinformationen δ_{ji} (rückwärts von Neuron j zu Neuron i) wird in Kapitel 1.3.3 aus dem Gradientenabstiegsverfahren hergeleitet und ergibt für gewöhnliche Gewichte

$$\delta_{ji} := \begin{cases} \delta_j(t) \cdot w_{ij} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \delta_j(t) \cdot (o_{ij} - w_{ij}) & \text{wenn Neuron } j \text{ radial ist} \end{cases} \quad .$$

Diese allgemeine Formulierung für Gewichte wird im Verlaufe dieses Kapitels benötigt werden, um allgemeingültige Aussagen zu Netzwerken mit kombinierten Architekturen und entsprechender Lernverfahren zu erlauben.

Rekurrenzen: Rekurrenzen – dargestellt in Abbildung 1(b) – können als zeitlich verzögernde Gewichte dargestellt werden. Dies bietet gegenüber der Betrachtungsweise als zeitverzögernde Zustands-Neurone Vorteile bei der Formulierung von Lernalgorithmen. Sie werden teils auch „tapped delay lines“ oder „time-delayed connections“ genannt, und reichen im Gegensatz

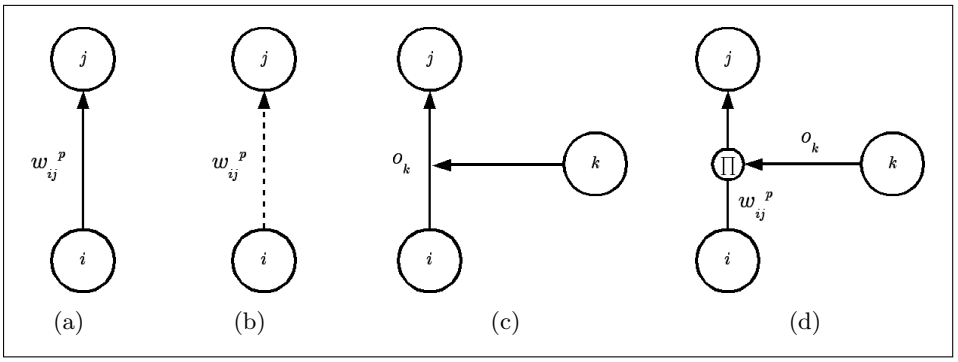


Abbildung 1: Verschiedene Gewichts-Typen: **Normales Gewicht** (a), **Rekurrenz** (b), **Gewicht zweiter Ordnung** (c), **moduliertes Gewicht** (d).

zu klassischen Gewichten die Aktivierung des Vorgängerneurons mit einer Zeitverzögerung weiter. Bei der Verzögerung kann es sich um einen diskreten Zeitschritt handeln, oder ein exponentiell abfallendes Mittel aller vorangegangenen Aktivierungen verwendet werden. In letzterem Fall entspricht dieses Gewicht einem Leckintegrator (leaky integrator), so dass in jedem Zeitschritt eine Abhängigkeit von der kompletten Historie entsteht. Rekurrente Architekturen können generell über diesen Gewichtstyp realisiert werden, da die Zeitverzögerung der Informationspropagierung topologische Zyklen auflöst. Für vorwärts- und rückwärtspropagierte Informationen folgen daraus die Funktionalitäten

$$
\begin{aligned}
w_{ij} &:= w_{ij}^{p} \\
o_{ij} &:= \overline{o_i}(t) \\
\delta_{ji} &:= \begin{cases} \overline{\delta_j}(t) \cdot w_{ij} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \overline{\delta_j}(t) \cdot (\overline{o_i}(t) - w_{ij}) & \text{wenn Neuron } j \text{ radial ist} \end{cases} .
\end{aligned}
$$

mit

$$
\overline{o_i}(t) := \overline{o_i}(t-1) \cdot (1 - \lambda^{-1}) + o_i(t-1) \cdot (\lambda^{-1})
$$

und

$$
\overline{\delta_j}(t) := \overline{\delta_j}(t-1) \cdot (1 - \lambda^{-1}) + \delta_j(t-1) \cdot (\lambda^{-1}) .
$$

wobei λ den Horizont in Zeitschritten angibt. Für $\lambda = 1$ ergibt sich eine Verzögerung um einen Zeitschritt ohne exponentielle Mittelung.

Gewichte zweiter Ordnung Gewichte zweiter Ordnung – dargestellt in Abbildung 1(c) – beziehen neben Vorgänger und Nachfolger noch ein weiteres Neuron k mit ein, dessen Aktivierung den Gewichtsparameter ersetzt. Das Gewicht selbst ist also nicht parametrisiert. Die Formulierungen der Netzeingaben 1 und 2 verhalten sich für Gewichte mit drei Indizes analog.

Die Bestimmung des Gewichtes anhand der Ausgabe eines Neurons kann als Modulation eines Netzparameters und damit der Funktionalität des nachfolgenden Netzes betrachtet werden. Gewichte zweiter Ordnung bilden daher einen wichtigen Bestandteil der in Kapitel 1.5 vorgestellten modulierbaren Netze. Die Funktionalität eines solchen Gewichts ist definiert durch:

$$
\begin{aligned}
w_{ikj} &:= o_k(t) \\
o_{ikj} &:= o_i(t)
\end{aligned}
$$

Bei der Rückwärtspropagierung ergeben sich durch die zusätzliche Abzweigung zum modulierenden Neuron k zwei Wege:

$$\delta_{ji} := \begin{cases} \delta_j(t) \cdot w_{ikj} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \delta_j(t) \cdot (o_{ikj} - w_{ikj}) & \text{wenn Neuron } j \text{ radial ist} \end{cases}$$
$$\delta_{jk} := \begin{cases} \eta \cdot \delta_j(t) \cdot o_{ij} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \eta \cdot \delta_j(t) \cdot (w_{ikj} - o_{ikj}) & \text{wenn Neuron } j \text{ radial ist} \end{cases} . \tag{3}$$

Die Rückpropagierung der Fehlerinformation δ_{jk} zum modulierenden Neuron k entspricht der nach dem Rückpropagierungs-Lernalgorithmus angestrebten Gewichtsänderung in Gleichung 9, wobei η die Lernrate darstellt.

Diese Form der Gewichte eignet sich wie beschrieben für die Stimulusabhängige Bestimmung von Gewichten, und im Umkehrschluss anhand der Rückpropagierung für die Bestimmung eines angemessenen Stimulus. Sie eignet sich hingegen nicht für die kontinuierliche Veränderung von Gewichten anhand einer Sensorik, da die Rückwärtspropagierung nicht-symmetrisch verläuft.

Modulierte Gewichte: Als modulierte Gewichte – dargestellt in Abbildung 1(d) – werden in RNNPBlib Gewichte vorgestellt, deren Funktionalität auch ohne Modulation erhalten bleiben kann. Dies wird erreicht, indem sie wie normale Gewichte einen Gewichtsparameter w^p_{ikj} aufweisen, der unabhängig von der Modulation gelernt werden kann. Im Gegensatz zu den Gewichten zweiter Ordnug moduliert die Aktivierung des Neurons k den Gewichtsparameter w^p_{ikj}, anstatt diesen zu ersetzen. Bei der Fehler-Rückpropagierung erhält das Neuron k ferner einen Anteil μ der nach dem Lernverfahren angestrebten Gewichtsänderung, so dass der Lerneinfluss zwischen Netzwerk und Modulator aufgeteilt werden kann. Für $\mu = 0$ entsprechen modulierte Gewichte den gewöhnlichen Gewichten, für $\mu = 1$ entsprechen sie den Gewichten zweiter Ordnung. Sie sind definiert durch:

$$\begin{aligned} w_{ikj} &:= w^p_{ikj} \cdot o_k(t) \\ o_{ikj} &:= o_i(t) \end{aligned} \tag{4}$$

Bei der Rückwärtspropagierung ergeben sich wie bei den Gewichten zweiter Ordnung zwei Wege. Jedoch wird die Propagierung zusätzlich durch den Modulator-Anteil μ beeinflusst:

$$\delta_{ji} := \begin{cases} \delta_j(t) \cdot w_{ikj} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \delta_j(t) \cdot (o_{ikj} - w_{ikj}) & \text{wenn Neuron } j \text{ radial ist} \end{cases} \tag{5}$$

$$\delta_{jk} := \begin{cases} \mu \cdot \eta \cdot \delta_j(t) \cdot o_{ikj} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \mu \cdot \eta \cdot \delta_j(t) \cdot (w_{ikj} - o_{ikj}) & \text{wenn Neuron } j \text{ radial ist} \end{cases} . \tag{6}$$

Biologisch gesehen ist dieser Gewichtstyp durch die präsynaptische Hemmung inspiriert: Bei dieser wird anhand einer axo-axonischen Synapse eine Modulation (meist Hemmung) der Transmitterfreisetzung in einem *Axonterminal* einer Nervenzelle bewirkt. Diese Art der Modulation findet zum Beispiel bei der differenzierten Hemmung sensorischer Signale im Rückenmark [6, 7, 8] statt. Präsynaptische Modulation durch Heterorezeption wurde ferner im Hippocampus beobachtet [9], das heißt durch den Einfluss von nicht-eigenen Neurotransmittern, -modulatoren und -hormonen an präsynaptischen Rezeptoren.

Sieht man ein abstraktes Gewicht als beliebig geartete Konnektivität zwischen Nervenzellen an, kann man anhand der zitierten Beobachtungen folgern, dass diese aus einer morphologischen und einer modulatorischen Komponente besteht, und damit mindestens durch zwei Faktoren repräsentiert werden muss. Diese Sachverhalte sind durch Gewichte zweiter Ordnung nicht abgedeckt, da diese von einer unveränderlichen morphologischen Konnektivität ausgehen.

Initialisierung der Parameter: Gewichte sollten so initialisiert werden, dass in ihren Nachfolgeneuronen anfänglich keine übermäßige Aktivierung provoziert wird, damit die Ausgabe eines neuronalen Netzes nicht von der Anzahl Neuronen abhängig ist. Je nach Aktivierungsfunktion des Nachfolge-Neurons kann also eine Initialisierung um 0 (Tangens-Hyperbolicus und (quasi-)lineare Funktionen) oder kleiner (logistische Funktion und Heaviside) sinnvoll sein. Gewichte, die auf monoton diskriminierende Neurone zeigen, sollten jedoch niemals genau 0 entsprechen, da sonst ein Lernen mangels Rückpropagierung eines Fehlerterms verhindert wird.

Generell sollten Gewichte randomisiert initialisiert werden. Werden als Extrembeispiel alle Gewichte eines Netzes zur Approximation einer Funktion $f(x) : \mathbb{R} \to \mathbb{R}$ gleich initialisiert, dann erfahren sie alle die gleiche Rückpropagierung und damit die gleiche Anpassung, was die Propagierungs-Redundanz und somit die Lernfähigkeit des Netzwerks einschränkt. Gewichte sollten also immer mit einem (normalverteilten) Rauschen initialisiert werden, um eine Auflockerung der Gewichtssymemtrie zu erreichen (sog. „symmetry breaking“ [10]).

Ist ein Gewicht ein Element des Zentrums einer radialen Basisfunktion, so kann es gleichverteilt im Intervall seines Eingaberaums initialisiert werden, so dass das Zentrum der RBF initial zufällig gewählt wird.

1.3 Netzwerk-Lernverfahren

In diesem Kapitel werden Verfahren für das überwachte Lernen aller aus Kombinationen der oben genannten Komponenten zusammengesetzten Netze beschrieben.

1.3.1 Fehlermaß

Ziel des Lernens ist die Minimierung der Abweichung der Ausgabe $\mathbf{o}$ eines NN von der zu approximierenden Zielfunktion $\mathbf{y}(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^n$. Als Maß für diese Abweichung wird in der Regel das quadratische Fehlermaß

$$E = \frac{1}{2} \sum_{i=1}^{n} (y_i - o_i)^2 \tag{7}$$

herangezogen, wobei der Index i alle Neuronen der Ausgabeschicht des NN durchläuft. Die Verwendung dieses Fehlermaßstabs erlaubt aufgrund seiner einfachen Differenzierbarkeit, aus dem nachfolgend beschriebenen Gradientenabstiegsverfahren den Rückpropagierungsalgorithmus herzuleiten, bei dem ein Fehlerterm rückwärts durch das Netzwerk propagiert werden kann, um fehlerminimierende Parameteränderungen im Netz zu bestimmen.

1.3.2 Gradientenabstieg

Das Gradientenabstiegsverfahren stellt ein allgemeines Verfahren zur Parameteroptimierung dar, welches einen Parameter umgekehrt proportional zur Steigung der Fehlers bezüglich des Parameters verändert, um eine Verringerung desselben zu erreichen. Im Fall überwachter NN sind die zu optimierenden Parameter die Gewichte w_{ij}, so dass der Gradientenabstieg aus der Ableitung des Fehlermaßes E nach den Gewichten folgt. Die Anpassung wird mit einer (fixen oder variablen) Lernrate η vorgenommen, so dass folgt:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \quad . \tag{8}$$

Unter der Annahme eines lokal linearen Verhaltens des Fehlers führt dies immer zu einer Verbesserung des Netzes. Allerdings ist die einzelne Ableitung des Fehlers nach allen Parametern rechnerisch kostspielig, da hierzu jeweils die Ausgabe des gesamten Netzes neu berechnet werden muss.

Das Gradientenabstiegsverfahren tendiert aufgrund der häufig nicht gegebenen Linearität des partiellen Fehlers dahin, zu lokalen Parameteroptima zu konvergieren.

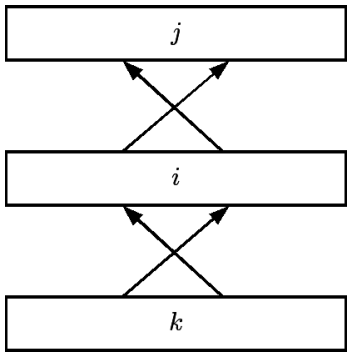


Abbildung 2: Eine Beispielhafte mehrschichtige Topologie. Jedes Rechteck stellt eine Schicht Neuronen dar, überkreuzte Pfeile bedeuten Vollverknüpfungen zwischen den Schichten.

1.3.3 Rückpropagierungsverfahren

Das Rückpropagierungsverfahren (auch generalisierte Delta-Lernregel oder Fehlerrückführung) folgt (nach Rummelhart et al. 1983 [11]) aus der Ausformulierung des Gradientenabstiegs in Gleichung 8 unter Annahme des Fehlermaßes 7 und Kenntnis der Netzeingabefunktion jedes Neurons (Gleichung 1 bzw. Gleichung 2). Der Einfachheit halber wird für die Herleitung an dieser Stelle ein vorwärtsgerichtetes Netzwerk mit drei Schichten angenommen, wobei die Neurone der Ausgabeschicht die Indizes j, die der verdeckten Schicht die Indizes i und die der Eingabeschicht die Indizes k tragen, wie Abbildung 2 aufzeigt.

Die partielle Ableitung des Fehlers E nach einem auf die Ausgabeschicht zeigenden Gewicht w_{ij} ergibt mit $o_j = o_j(\text{net}_j)$:

$$\frac{\partial E}{\partial w_{ij}} = -(y_j - o_j) \cdot \frac{\partial o_j(\text{net}_j)}{\partial w_{ij}} \quad .$$

Eingesetzt in den Gradientenabstieg nach Gleichung 8 folgt erstens für $\text{net}_j := \text{net}_j^{\text{MD}} = \sum_{i=1} w_{ij} \cdot o_{ij}$, also Gewichte, die auf monoton diskriminierende Ausgabeneurone zeigen:

$$\Delta w_{ij}^{\text{MD}} = \eta \cdot \delta_j \cdot o_{ij}$$

wobei δ_j als Fehlerterm für Ausgabeneurone anzusehen ist, welcher zu

$$\delta_j = (y_j - o_j) \cdot \frac{\partial o_j(\text{net}_j)}{\partial \text{net}_j}$$

folgt. Analog ergibt sich zweitens für $\text{net}_j := \text{net}_j^{\text{RBF}} = \frac{1}{2}\sum_{i=1}^{n}(w_{ij} - o_{ij})^2$ und Gewichte, die auf radiale Ausgabeneurone zeigen:

$$\Delta w_{ij}^{\text{RBF}} = \eta \cdot \delta_j \cdot (w_{ij} - o_{ij})$$

mit selbigem δ_j. Gewichte, die auf verdeckte Neurone zeigen müssen indirekt berechnet werden. Aus der Ableitung der Fehlerfunktion nach einem solchen Gewicht w_{ki} folgt

$$\begin{aligned}
\frac{\partial E}{\partial w_{ki}} &= \sum_j -(y_j - o_j) \cdot \frac{\partial o_j(\text{net}_j)}{\partial w_{ki}} \\
&= \sum_j -(y_j - o_j) \cdot \frac{\partial o_j(\text{net}_j)}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ki}} \\
&= \sum_j -\delta_j \cdot \frac{\partial \text{net}_j}{\partial w_{ki}}
\end{aligned}$$

so dass sich mit der linearkombinierenden Netzeingabe $\text{net}_j := \text{net}_j^{\text{MD}}$ ergibt:

$$\begin{aligned}
&= \sum_j -\delta_j \cdot w_{ij} \cdot \frac{\partial o_i(\text{net}_i)}{\partial w_{ki}} \\
&= -\frac{\partial o_i(\text{net}_i)}{\partial \text{net}_i} \cdot \frac{\partial \text{net}_i}{\partial w_{ki}} \cdot \sum_j \delta_j \cdot w_{ij} \\
&= -\frac{\partial o_i(\text{net}_i)}{\partial \text{net}_i} \cdot o_{ki} \cdot \sum_j \delta_j \cdot w_{ij} \qquad .
\end{aligned}$$

Für Gewichte, die auf *diskriminierende* verdeckte Neurone zeigen, folgt also durch Einsetzen in 8 analog zur obigen Lernregel:

$$\Delta w_{ki}^{\text{MD}} = \eta \cdot \delta_i^{\text{MD}} \cdot o_{ki}$$

jedoch mit dem durch Rückpropagierug gewonnenen Fehlerterm für verdeckte Neurone, deren Nachfolger monoton diskriminierend sind:

$$\delta_i^{\text{MD}} = \frac{\partial o_i(\text{net}_i)}{\partial \text{net}_i} \cdot \sum_j \delta_j \cdot w_{ij} \qquad .$$

Die Herleitung für Netzwerke mit mehreren verdeckten Schichten ist analog durchführbar. Anhand des Fehlerterms δ_i wird deutlich, dass der an der Ausgabe entstehende Fehler gemäß der Gewichtung bis zur Eingabeschicht zurückpropagiert werden kann, um die Parameter des Netzes zu optimieren.

Für Gewichte, die auf *radiale* verdeckte Neurone zeigen, ergibt sich die Lernregel anhand $\text{net}_j = \text{net}_j^{\text{RBF}}$ auf selbige Weise:

$$\begin{aligned}
\frac{\partial E}{\partial w_{ki}} &= \sum_j -\delta_j \cdot \frac{\partial \text{net}_j}{\partial w_{ki}} \\
&= \sum_j -\delta_j \cdot (w_{ij} - o_{ij}) \cdot \left(-\frac{\partial o_i(\text{net}_i)}{\partial w_{ki}} \right) \\
&= -\frac{\partial o_i(\text{net}_i)}{\partial \text{net}_i} \cdot \frac{\partial \text{net}_i}{\partial w_{ki}} \cdot \sum_j \delta_j \cdot (o_{ij} - w_{ij}) \\
&= -\frac{\partial o_i(\text{net}_i)}{\partial \text{net}_i} \cdot (w_{ki} - o_{ki}) \cdot \sum_j \delta_j \cdot (o_{ij} - w_{ij}) \\
\Rightarrow \Delta w_{ki}^{\text{RBF}} &= \eta \cdot \delta_i^{\text{RBF}} \cdot (w_{ki} - o_{ki})
\end{aligned}$$

mit dem Fehlerterm für verdeckte Neurone, deren Nachfolger radial sind:

$$\delta_i^{\text{RBF}} = \frac{\partial o_i(\text{net}_i)}{\partial \text{net}_i} \cdot \sum_j \delta_j \cdot (o_{ij} - w_{ij}) \quad .$$

Es wird deutlich, dass auch mehrschichtige Netzwerke mit radialen Basisfunktionen anhand einer Rückpropagierung trainiert werden können. Dabei ist zu beachten, dass die Ableitung radialer Aktivierungsfunktionen immer negativ ist, da die Netzeingabe als quadratische Summe positiv ist. Ist der Fehlerterm eines Neurons also negativ (d.h. die Ausgabe ist zu gering), wird ein partielles Zentrum linksverschoben (bzw. ein Gewicht verringert), wenn es rechtsseitig der partiellen Eingabe liegt, oder rechtsverschoben, wenn es linksseitig der partiellen Eingabe liegt, so dass sich in beiden Fällen die Ausgabe des Neurons erhöht. Umgekehrtes gilt für positive Fehlerterme, so dass sich die Ausgabe verringert.

Oftmals werden für das Lernen von RBFs allerdings evolutionäre Ansätze oder der einfache Gradientenabstieg vorgeschlagen. Ein Vorteil dieser Vorgehensweise ist, dass so auch (Ko-)Varianzen der radialen Funktionen gelernt werden können. Außerdem können sich evolutionäre Verfahren leichter über lokale Optima bei der Parameterfindung hinwegsetzen, welche bei RBFs besonders häufig anzutreffen sind. Das hier vorgestellte Lernverfahren erlaubt jedoch die einfachere Kombination anderer Architekturen mit RBFs. Ferner ist (unter Beachtung des Vorzeichens von δ) die Nähe zur Hebb'schen Feedback-Lernregel nach Carpenter und Grossberg (ARTMAP, 1991) bemerkenswert [12, Seite 17].

In Anlehnung an die Definitionen aus Kapitel 1.2.2 lässt sich unter Berücksichtigung der obigen Herleitungen für alle Kombinationen der vorgestellten Gewichte und Neuronen eine allgemeine Lernregel formulieren:

$$\Delta w_{ij} = \begin{cases} \eta \cdot \delta_j \cdot o_{ij} & \text{wenn Neuron } j \text{ monoton diskr. ist} \\ \eta \cdot \delta_j \cdot (w_{ij} - o_{ij}) & \text{wenn Neuron } j \text{ radial ist} \end{cases} \tag{9}$$

$$\delta_j = \begin{cases} \frac{\partial o_j(\text{net}_j)}{\partial \text{net}_j} \cdot (y_j - o_j) & \text{wenn } j \text{ ein Ausgabe-Neuron ist} \\ \frac{\partial o_j(\text{net}_j)}{\partial \text{net}_j} \cdot \sum_i \delta_{ij} & \text{wenn } j \text{ ein verdecktes Neuron ist} \end{cases} \tag{10}$$

Diese Lernregel berücksichtigt, dass ausgehende Gewichte und deren Nachfolgeneuronen sowie eingehende Gewichte und deren Vorgänger nicht alle vom selben Typ sein müssen. Dadurch sind auch Kombinationen von linearen Diskriminatoren und RBFs realisierbar, selbst wenn Gewichte höherer Ordnung oder Rekurrenzen enthalten sind.

1.3.4 Batch-Rückpropagierungsverfahren

Im obigen Lernverfahren wurde vorgestellt, wie Parameter eines Netzes unter Kenntnis der Abweichung einer Netzausgabe von einer zugehörigen Sollausgabe mit einer bestimmten Lernrate angepasst werden können. Wird diese Anpassung sequentiell wiederholt, das heißt, werden für jedes Paar $(\mathbf{y}(\mathbf{x}), \mathbf{x})$ der zu approximierenden Funktion die Netzparameter nacheinander angepasst, spricht man von Online-Lernen. Ein Nachteil dieses Verfahrens ist, dass die Netzparameter dabei in Richtung lokaler Optima streben können, bevor alle Trainingsdaten beurteilt wurden. Anders ausgedrückt sind nicht alle Fehlerterme gleichberechtigt, da frühere Parameteranpassungen nachfolgende beeinflussen, und somit die Reihenfolge des Trainings auf Datenpaare eine Rolle für die Parameterfindung spielt. Diesem Effekt kann zwar durch eine Randomisierung der Reihenfolge entgegengewirkt werden, gleichsam entfällt damit aber auch die Möglichkeit, rekurrente Netze zu trainieren, da deren Ausgabe von der Reihenfolge der Eingaben abhängig ist.

Ein gängiges und mathematisch korrekteres Verfahren ist es, Parameteränderungen erst vorzunehmen, wenn das Netzwerk eine ganze *Epoche* von Trainingsdaten gesehen hat. Dies wird in Kombination mit den zuletzt vorgestellten Parameteranpassungen als Batch-Rückpropagierung bezeichnet. Gewichtsänderungen werden dabei über eine Epoche von $T_e - T_{e-1}$

Zeitschritten aufsummiert, und anschließend angewendet:

$$\Delta w_{ij}^{BBP}(T_e) = \frac{1}{T_e - T_{e-1}} \sum_{t=T_{e-1}}^{T_e} \Delta w_{ij}(t) \quad . \tag{11}$$

Es sei darauf hingewiesen, dass Batch-Lernen in der Regel ohne den Mittelungsterm $\frac{1}{T_e - T_{e-1}}$ beschrieben wird. Ungemitteltes Batch-Lernen kann jedoch zu einer Destabilisierung des Netzwerks führen, wenn viele Gewichtsänderungen mit geichem Vorzeichen aufsummiert werden, ohne dass das Netzwerk sich zwischenzeitig anpassen kann. Da die Epochen von unterschiedlicher Länge sein können, wird durch den Mittelungsterm zudem sichergestellt, dass sich alle Epochen gleichberechtigt auf die Netzparameter auswirken. Eine oft vorgenommene Anpassung der Lernrate η in Abhängigkeit von der Epochenlänge entfällt damit ebenfalls.

1.3.5 Batch Backpropagation Through Time

Backpropagation Through Time (BPTT) beschreibt ein Lernverfahren, das den zeitlichen Horizont rekurrenter Netze berücksichtigt. Die Grundlage des Algorithmus ist, dass jedes zyklische Netz durch ein (größeres) azyklisches Netzwerk repräsentiert werden kann [13]: Zur Berechnung eines rekurrenten Netzes kann dieses sinnbildlich an jeder Rekurrenz bzw. jedem verzögenden Gewicht zeitlich rückwärts entfaltet werden. Anschließend wird die mittlere aus dem entfalteten Netz bestimmte Änderung der rekurrenten Gewichte zur Anpassung dieser herangezogen. Der Lernalgorithmus folgt damit und anhand obiger Formulierungen zu:

$$\Delta w_{ij}^{\text{BPTT MD}}(t) = \frac{\eta}{t} \sum_{\tau=0}^{t} \delta_j(\tau) \cdot o_{ij}(\tau) \tag{12}$$

für monoton diskriminierende Folgeneuronen, bzw. zu

$$\Delta w_{ij}^{\text{BPTT RBF}}(t) = \frac{\eta}{t} \sum_{\tau=0}^{t} \delta_j(\tau) \cdot (w_{ij}(t) - o_{ij}(\tau)) \tag{13}$$

für radiale Folgeneuronen, wobei w_{ij} einem rekurrenten Gewicht entspricht.

Hier ist zu beachten, dass anhand des getroffenen Formalismus (mit Gewichten als verzögerne Elemente in Rekurrenzen) kein Abspeichern aller Aktivierungen und Delta-Terme der Neurone und damit kein Neuberechnen einer Summe pro Zeitschritt nötig ist. Es kann (unter der Voraussetzung eines unendlichen Horizonts) bei der Implementierung inkrementell vorgegangen werden: In jedem Zeitschritt wird im Falle von Formel 12 das Produkt

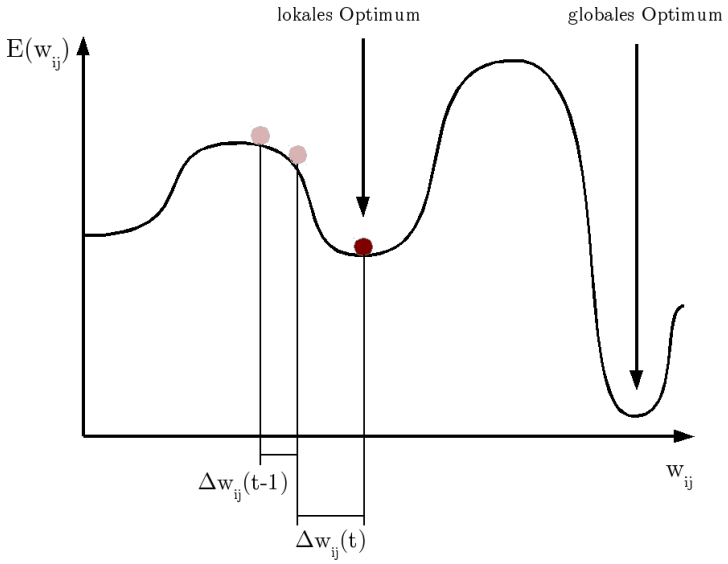


Abbildung 3: Gradientenabstieg über die Fehlerlandschaft. Das Lernen der Gewichte kann in lokalen Optima zum Erliegen kommen.

$\delta_j(t) \cdot o_{ij}(t)$ auf eine Epochen-globale Variable aufaddiert. Bei Verwendung von Formel 13 muss entsprechend ihrer Ausmultiplizierung *zudem* das Produkt $\delta_j(t) \cdot w_{ij}(t)$ aufsummiert werden.

Im Fall von *Batch*-BPTT wird das Entfalten des Netzes in jedem Zeitschritt durchgeführt und das Mittel aller per BPTT berechneten Gewichtsanpassungen nach einer Epoche angewendet.

1.3.6 Erweiterterungen

Aus dem Gradientenabstieg hergleitete Lernalgorithmen weisen einige aus der lokalen Suche nach Parameteroptima resultierende Probleme auf. Betrachtet man die Fehlerlandschaft eines Lerners in Bezug auf die Parameter, wie in Abbildung 3 gezeigt, wird offensichtlich, dass die Parameter bei zu geringer Lernrate gegen lokale Minima des Fehlers konvergieren können. Ist die Lernrate dagegen zu hoch eingestellt, können Parameter in Schluchten der Fehlerlandschaft oszillieren, was das Finden guter Parameter erschwert oder gar verhindert. Schlimmstenfalls können bei zu hoher Lernrate auch gute lokale Minima des Fehlers übersprungen werden, so dass das Netz gar nicht oder gegen schlechtere Parameter konvergiert. Für diese und ähnliche Probleme werden in den folgenden Abschnitten Lösungsansätze beschrieben.

Trägheitsterm: Eine Methode, welche die Probleme der Parameterkonvergenz gegen lokale Optima sowie das „Verdursten“ des Gradienten auf

kurzen Ebenen abschwächt, wird als Trägheitsterm oder Momentum bezeichnet. Dabei wird bei Parameteranpassung zum Zeitpunkt T_e immer ein Teil der zuletzt angewendeten Parameteranpassungen übernommen, so dass selbige eine gewisse Trägheit aufweisen:

$$\Delta w_{ij}^{EBBP}(T_e) = \Delta w_{ij}^{BBP}(T_e) + \Delta w_{ij}^{BBP}(T_{e-1}) \cdot \alpha \quad . \tag{14}$$

Das Momentum $\alpha \in [0, 1]$ ist heuristisch so zu bestimmen, dass kleinere lokale Minima sowie kurze Ebenen in der Fehlerlandschaft übersprungen werden, wenn der Gradient zuvor eine hinreichende Geschwindigkeit besaß, damit die Parameter eines Netzes gegen gute Optima konvergieren.

Unter Verwendung eines Momenti-Terms wird das Batch-Rückpropagierungsverfahren an dieser Stelle als erweitertes Batch-Rückpropagierungsverfahren (EBBP, „Enhanced Batch Backpropagation“) bezeichnet.

Gewichts-Dämpfung: Ein weiteres Problem des Gradientenabstiegs ist, dass keine quantitative Beschränkung für die Parameter existiert: Gewichte können beliebig groß werden, falls stets gilt, dass sich der Fehler verringert, wenn sich das Gewicht erhöht. Dies kann beispielsweise geschehen, wenn zwei Gewichte entgegengesetzten Einfluss auf den Fehler haben. Große Gewichte können jedoch die Varianz der Ausgabefunktion erhöhen [14] und damit die Generalisierungsfähigkeit eines Netzes verschlechtern.

Zur Verbesserung der Generalisierung kann eine Gewichts-Dämpfung eingesetzt werden [15], welche die Gewichtsanpassung anhand des Betrages der Gewichte reduziert:

$$\Delta w_{ij}^{WD} = \Delta w_{ij} - w_{ij} \cdot \beta \tag{15}$$

wobei $\beta \in [0, 1]$ die Gewichtsdämpfung bestimmt. Durch diese Form der Gewichts-Dämpfung konvergieren neuronale Netze gegen kleinere Parameter.

Flat-Spot Elimination: Das Lernen eines Gewichtsparameters kommt zum Erliegen, wenn die Ableitung der Aktivierungsfunktion des nachfolgenden Neurons gegen 0 geht, da dann auch dessen Delta-Term gegen 0 geht (siehe Gleichung 10). Bei allen endlichen Aktivierungsfunktionen trifft dies zu, wenn die Netzeingabe hinreichend weit vom Schwellwert oder Zentrum des Neurons entfernt liegt. Bei radialen Neuronen kommt es zudem dazu, wenn die Netzeingabe direkt im Zentrum des Neurons liegt (bei Maximalaktivierung). Eine Methode, dies zu verhindern, wurde 1988 von Scott Fahlman als „flat-spot elimination“ vorgestellt [16]: Dabei wird die Ableitung der Aktivierungsfunktion eines Neurons um eine Konstante $C > 0$ erhöht, so dass diese nie 0 erreichen kann. Zu beachten ist, dass dies bei radialen Neuronen

betragsgemäß (d.h. unter Beachtung des Vorzeichens) geschehen muss. Daraus folgt nach den obigen Lernregeln 9 und 10, dass bei Rückpropagierung eines Fehlers und einer Aktivierung größer 0 des Vorgängerneurons immer eine geringe Gewichtsänderung vorgenommen wird. Ferner kann dieses Verfahren auch zum Lernen neuronaler Netze mit Aktivierungsfunktionen eingesetzt werden, deren Ableitung stellenweise genau 0 ist, was beispielsweise auf die Heaviside-Funktion zutrifft.

Alternativ kann C auch als das betragsgemäßes Minimum der Ableitung der Aktivierung definiert werden, so dass die Präzision der Ableitung in nicht-kritischen Regionen erhalten bleibt. Das später vorgestellte RNNPBlib-Framework verwendet diese Variante.

Simulierte Abkühlung: Bei der Wahl der Lernrate ist in der Regel ein Kompromiss zu treffen: Während ein Netzwerk mit hoher Lernrate in der Regel schnell zu einem schlechten Optimum konvergiert und anschließend anfäng zu oszillieren, oder sich von vornherein destabilisiert, konvergiert ein Netzwerk mit zu geringer Lernrate eventuell sehr langsam. Es kann sogar vorkommen, dass die geringe Lernrate verhindert, dass über schlechte lokale Optima hinweggegangen wird, so dass das Netzwerk keinen nennenswerten Lernerfolg erzielt.

Ein Verfahren, das die Lernrate im Verlauf des Lernens anpasst, wird als simulierte Abkühlung („simulated annlealing“ [17]) bezeichnet. Dabei wird die Lernrate η anhand einer Abkling-Konstante exponentiell verrigert. Diese folgt (bei Batch-Lernen) in Epoche e zu:

$$\eta^{\mathrm{SA}}(T_e) = \eta^{\mathrm{SA}}(T_{e-1}) \cdot \gamma \quad . \tag{16}$$

Die Wahl der Abkling-Konstante $\gamma \in (0, 1)$ sollte so getroffen werden, dass das Netzwerk sich am Anfang des Lernens nicht destabilisiert, und gegen Ende des Lernens eine Lernrate aufweist, die die oszillationsfreie Konvergenz gegen ein hinreichend gutes Optimum erlaubt. Diese Wahl ist in der Regel heuristisch zu treffen und stark von der verwendeten Architektur und der zu erlernenden Zielfuntion abhängig. Zudem ist die Lernrate bei der simulierten Abkühlung ein globaler Parameter, so dass nur die gesamte Konvergenzgeschwindigkeit des Netzes eingestellt werden kann.

Es sei darauf hingewiesen, dass Netzwerke, die einer simulierten Abkühlung unterzogen werden, in Verbindung mit Gewichts-Dämpfung nach dem Lernen zum Divergieren neigen, sofern der Dämpfungs-Parameter β nicht ebenfalls abgekühlt wird. Um dem entgegenzuwirken sollte dieser Parameter unter Verwendung der simulierten Abkühlung ebenfalls exponentiell verringert werden:

$$\beta^{\mathrm{SA}}(T_e) = \beta^{\mathrm{SA}}(T_{e-1}) \cdot \gamma \quad . \tag{17}$$

Separate simulierte Abkühlung: Ein Verfahren, das die Konvergenzgeschwindigkeit eines Netzwerkes für jeden Parameter einzeln bestimmt, wird bei RNNPBlib als separate simulierte Abkühlung vorgestellt (eigenes Verfahren). Um dies zu realisieren wird ein Korridor $K = [\Delta_{\min}, \Delta_{\max}]$ festgelegt, der die maximale bzw. minimale Adaption eines Gewichts pro Epoche beschreibt. Überschritte die Anpassung eines Gewichts bei Anwendung der zuletzt verwendeten Lernrate die obere Schranke des Korridors, so wird die Lernrate dieses Gewichts separat so verringert, dass die Gewichtsänderung in dieser Epoche am oberen Ende des Korridors liegt. Analog wird die Lernrate für das Gewicht bei hypothetischer Unterschreitung der unteren Schranke so erhöht, dass die Gewichtsänderung am unteren Ende des Korridors liegt. Damit gilt also immer $\Delta w_{ij} \in K$. Sowohl die obere als auch die untere Schranke des Korridors können wie bei der simulierten Abkühlung im Verlauf des Lernens abgesenkt werden. Somit werden alle Gewichte in konstanter Bewegung gehalten, wobei der Bewegungsdruck stetig nachlässt. Das Verfahren lässt sich anhand des folgenden Formalismus' beschreiben:

$$\eta_{ij}^{\mathrm{SSA}}(T_e) = \begin{cases} \left| \frac{\Delta_{\max(T_e)} \cdot \eta_{ij}^{\mathrm{SSA}}(T_{e-1})}{\Delta w_{ij}\left(T_e, \eta_{ij}^{\mathrm{SSA}}(T_{e-1})\right)} \right| , \\ \qquad \text{wenn } |\Delta w_{ij}\left(T_e, \eta_{ij}^{\mathrm{SSA}}(T_{e-1})\right)| > \Delta_{\max}(T_e) \\ \left| \frac{\Delta_{\min(T_e)} \cdot \eta_{ij}^{\mathrm{SSA}}(T_{e-1})}{\Delta w_{ij}\left(T_e, \eta_{ij}^{\mathrm{SSA}}(T_{e-1})\right)} \right| , \\ \qquad \text{wenn } |\Delta w_{ij}\left(T_e, \eta_{ij}^{\mathrm{SSA}}(T_{e-1})\right)| < \Delta_{\min}(T_e) \end{cases} \tag{18}$$

$$\Delta w_{ij}^{\mathrm{SSA}}(T_e) = \Delta w_{ij}(T_e, \eta_{ij}^{\mathrm{SSA}}(T_e)) \tag{19}$$

$$\Delta_{\min}(T_e) = \Delta_{\min}(T_{e-1}) \cdot \gamma \tag{20}$$

$$\Delta_{\max}(T_e) = \Delta_{\max}(T_{e-1}) \cdot \gamma \quad . \tag{21}$$

Die separate Lernrate wird vor der Anpassung der Gewichte bestimmt. Sie ergibt sich aus der hypothetischen Gewichtsänderung ohne Anpassung der Lernrate $\Delta w_{ij}\left(T_e, \eta_{ij}^{\mathrm{SSA}}(T_{e-1})\right)$, wobei die Umrechnung nach Formel 18 einem einfachen Dreisatz entspricht. Das Verfahren bewirkt ein individuelles Einschwingen der Netzparameter in einem stetig verkleinernden Suchraum bei gleichzeitiger Verhinderung einer Destabilisierung durch Überhöhung von Gewichten. Dabei ist zu beachten, dass eine Dividion durch Null abgesichert werden muss, wenn die unadaptierte Gewichtsänderung 0 entspricht. Auch sollte die individuelle Lernrate auf ein Maxium von 1 beschränkt werden. Wie bei der simulierten Abkühlung ist die Einstellung der Parameter γ, $\Delta_{\min}$ und $\Delta_{\max}$ nicht trivial, es entfällt jedoch das Problem der initialen Wahl der Lernrate η. Versuchsauswertungen haben eine deutliche Verbesserung sowohl in der Lerngeschwindigkeit als auch in der Konvergenz des Fehlers gegenüber der simulierten Abkühlung und normalem Lernen gezeigt.

Spekulativ lässt sich ein Bezug zur Biologie vor dem Hintergrund der neuronalen Plastizität herstellen: Diese ist generell an morphologische und physiologische Bedingungen geknüpft, so dass Veränderungen nicht mit beliebiger Geschwindigkeit fortschreiten können (vergleichsweise: obere Schranke $\Delta_{\max}$). Eine minimale Plastizität ($\Delta_{\min}$) erscheint ferner in Bezug auf neuronales Rauschen plausibel, und ist aus technischer Sicht zudem durch die Begründungen der Flat Spot Elimination zu rechtfertigen.

1.4 Beispiele für Topologien

Im Folgenden werden zunächst einige bekannte Architekturen mit und ohne Rekurrenz angesprochen, welche anhand der vorgestellten Komponenten realisierbar sind, bevor genauer auf modulierte Netzwerke und deren Training Bezug genommen wird.

1.4.1 Mehrlagiges Perzeptron

Das mehrlagige Perzeptron (MLP) ist eine Erweiterung des Perzeptrons nach Rosenblatt (1958 [18]). Es ist einer der grundlegenden neuronalen Ansätze zur Approximation einer Funktion $\mathbf{y}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$. Die Neuronen eines MLP sind in einer beliebigen Anzahl Schichten organisiert: Eine Eingabeschicht mit n Neuronen, beliebig viele verdeckte Schichten mit beliebig vielen Neuronen, und eine Ausgabeschicht mit m Neuronen. Die Schichten sind hierarchisch vollverknüpft, das heißt jedes Neuron einer Schicht ist mit allen Neuronen der darüberliegenden Schicht mit normalen Gewichten verbunden. In Abbildung 4 ist dies durch sich überkreuzende Pfeile dargestellt. Einfache Pfeile bedeute eins-zu-eins Verknüpfungen. Es handelt sich also um eine rein vorwärtsgerichtete Architektur ohne Zyklen. Die Neuronen von Eingangs- und Ausgangsschicht werden oftmals linear, die Neuronen der verdeckten Schichten oftmals sigmoid modelliert. Jedes sigmoide Neuron einer verdeckten Schicht kann an genau einer Hyperebene seines

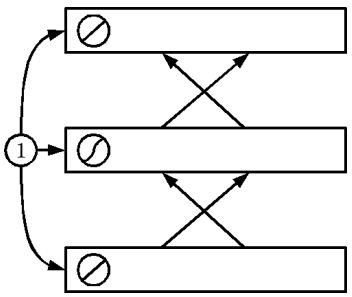


Abbildung 4: Ein Perzeptron mit drei Schichten. Zusätzlich ist der Schwellwert für alle Neuronen über ein Bias-Neuron dargestellt.

Eingangsraums diskriminieren. Durch die Mehrschichtigkeit sind also nichtlinear separierbare Funktionen (zum Beispiel die XOR-Funktion) mit einem MLP approximierbar, was den wesentlichen Fortschritt gegenüber dem Rosenblatt-Perceptron darstellt.

1.4.2 RBF-Netzwerke

Klassische RBFs bestehen aus einer linearen Eingabeschicht, einer verdeckten Schicht mit radialen (oft Gauß'schen) Neuronen und einer linearen Ausgabeschicht. Die Gewichte von der Eingabeschicht zur verdeckten Schicht bestimmen die Zentren der radialen Neuronen, während die Gewichte von der verdeckten zur Ausgabeschicht die Gewichtung der radialen Aktivierungen darstellen. Oft werden der gewöhnliche Gradientenabstieg oder evolutionäre Ansätze für das Training verwendet. Wie das MLP können RBFs jede Funktion ohne Zustandsraum approximieren. Dafür reicht jedoch im Gegensatz zum Perzeptron eine verdeckte Schicht aus.

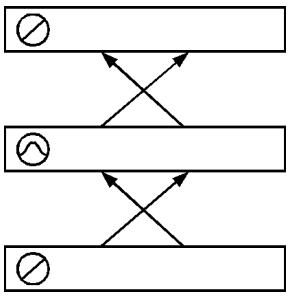


Abbildung 5: Ein RBF-Netzwerk. Die Gewichte von der Eingabe zur verdeckten Schicht legen die Zentren der radialen Funkionen fest. Die Gewichte zur Ausgabeschicht bestimmen die Gewichtung.

1.4.3 Rekurrente Netzwerke

Rekurrente neuronale Topologien weisen allgemein Zyklen auf. Man unterscheidet dabei zwischen direkten, indirekten und seitlichen Rückkopplungen. Direkte Rückkopplungen liegen vor, wenn die Ausgabe eines Neurons direkt mit dessen Eingang verbunden ist, so dass eine Selbstexitation oder -inhibition entstehen kann. Ein rekurrentes Netz weist indirekte Rückkopplungen auf, wenn die Ausgabe eines Neurons mit einem Neuron verbunden ist, das direkt oder indirekt mit ihm verbunden ist, was zum Beispiel bei einem Gewicht einer höheren Schicht zu einer darunterliegenden Schicht der Fall sein kann. Von seitlichen Rückkopplungen spricht man, wenn Neuronen der selben Schicht untereinander verbunden sind, und dadurch ein Zyklus entsteht (dies muss nicht der Fall sein). Durch die verzögernden Eigenschaften rekurrenter Gewichte (siehe Kapitel 1.2.2) werden diese Zyklen aufgelöst. Dadurch entsteht ein Zustandsraum in Abhängigkeit von der Historie der Eingaben des Netzes.

Der Vorteil eines rekurrenten Netzes liegt in dessen Fähigkeit, Korrelationen zeitlich versetzter Ereignisse zu erfassen. Dadurch ist eine bessere Prädiktion von Zeitreihen möglich [19]. Beim Training rekurrenter Netze ist folglich die Reihenfolge beim Lernen von Trainingsdaten zu beachten.

Jordan/Elman-Architektur: Jordan oder Elman-Netzwerke (Jordan, 1990 [20] bzw. Elman, 1990 [21]) zählen zu den ersten Formen rekurrenter Netze und werden auch „Simple Recurrent Networks“ (SRN) genannt. Ein SRN ist im Wesentlichen als MLP mit direkten Rückkopplungen in der Ausgabeschicht (Jordan) oder einer verdeckten Schicht (Elman) zu betrachten.

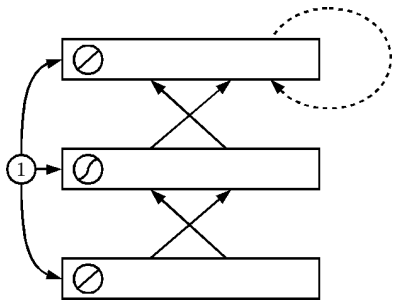


Abbildung 6: Jordan-Architektur mit direkten Rekurrenzen in der Ausgabeschicht.

1.4.4 Weitere Topologien

Anhand der Komponenten aus Kapitel 1.2 lassen sich neben den bekanntesten neuronalen Architekturen auch viele weniger bekannte und diskutierte Topologien aufbauen und trainieren, wie beispielsweise mehrschichtige, modulierte oder rekurrente RBFs und Kombinationen mit den oben genannten Netzwerken. Unter Verwendung von verzögernden Gewichten sind prinzipiell auch abgeschwächte Formen neuronaler Architekturen der dritten Generation (spikende neuronale Netze) realisierbar ähnlich dem Leckintegrator-Modell (siehe z.B. beim Graben et al., 2008 [22]), wobei jedoch die Gewichte anstelle der Neuronen die Integrationsaufgabe übernehmen.

1.5 Modulierte Topologien

Einen Vektor $\mathbf{CV}_v$, der die Funktion eines Netzes ohne Beeinflussung der Sensorik für eine bestimmte Zielfunktion $\mathbf{y}_v(\mathbf{x})$ moduliert, bezeichnet man als *Konzept-Vektor*. In Bezug auf kognitiv inspirierte Lerner werden diese Zielfunktionen im Folgenden auch als *Verhalten* bezeichnet.

Allen nachfolgend vorgestellten Modulationsarten ist gemein, dass ein Konzept-Vektor eine Teilmenge der Parameter eines neuronalen Netzwerks separat für ein Verhalten bestimmt, jedoch nicht als direkte Eingabe des Netzes dient. Diese Art der Modulation kann also – wie in den nachfolgenden Abschnitten erklärt wird – die Schwellwerte oder Gewichte eines Netzes betreffen. Abbildung 7 zeigt diesen Vorgang schematisch.

Durch Fehlerrückpropagierung ist es möglich, Konzept-Vektoren zu trainieren. Es wird gezeigt, dass dies bei geeigneter Wahl einer Topologie anhand der bekannten Lernverfahren geschehen kann.

Die partielle Modulierung eines Netzwerks lässt eine generelle Unterscheidung zwischen *verhaltensglobalen* und *verhaltenslokalen* Parametern zu: Während verhaltensglobale Parameter von der Modulierung des Netzes unbeeinflusst sind, werden verhaltenslokale Parameter allein durch den Konzept-Vektor bestimmt. Sind also alle Parameter eines Netzes verhaltenslokal, ist dies äquivalent zur Verwendung eines separaten Netzwerks zur Erzeugug jedes Verhaltens. Die Idee hinter der Modulation ist jedoch, möglichst effektiv und gezielt Teile eines Netzwerks zu modulieren, so dass Funktionen mit verwandten Eigenschaften approximiert werden können und sich eine Metrik im Raum der gelernten Konzept-Vektoren ergibt.

Im Folgenden wird erklärt, wie Konzept-Vektoren zur Modulation von Schwellwerten und Gewichten eingesetzt und erlernt werden können, und durch Verteilung der Modulation eine kompakte Repräsentation von Konzept-Vektoren erreicht werden kann.

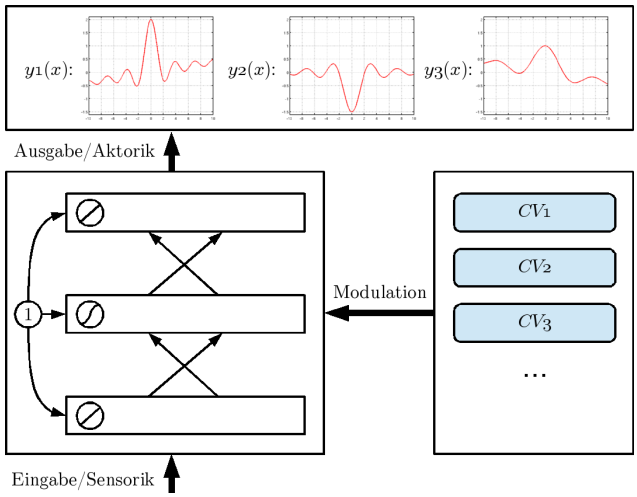


Abbildung 7: Ein Netzwerk, das anhand von Konzept-Vektoren moduliert wird. Die Modulation anhand $\mathbf{CV}_v$ erzeugt die Funktion $\mathbf{y}_v(\mathbf{x})$. Die Eingabe $\mathbf{x}$ kann für alle Funktionen identisch sein.

1.5.1 Konzept-Vektoren und -Schichten

Jeder Konzept-Vektor $\mathbf{CV}_v$ mit der Dimension n ist eindeutig einem Verhalten v zugeodnet. Solange ein Verhalten erzeugt werden soll, dient dieser als Eingabe einer Schicht mit n Neuronen, welche nicht an der Verarbeitung anderer Eingaben beteiligt ist. Eine solche Schicht wird im folgenden als *Konzept-Schicht* bezeichnet, wobei an dieser Stelle die Möglichkeit mehrerer Konzept-Schichten pro Netzwerk der Einfachheit halber außer Acht gelassen wird.

Eine Konzept-Schicht kann eine Teilmenge der Parameter eines Netzes entweder direkt modulieren, oder indirekt mittels der Entfaltung über ein Subnetzwerk. Zur Berechenbarkeit anhand der unten vorgestellen Lernverfahren weist jedes Neuron einer Konzept-Schicht genau ein ausgehendes Gewicht auf.

Konzept-Vektoren sind nach Möglichkeit so zu initialisieren, dass die Empfehlungen aus Kapitel 1.2.2 in Bezug auf die modulierten Parameter nicht verletzt werden. Beim Trainieren unter Verwendung von Trägheiten (siehe Kapitel 1.3.6) ist ferner ist zu beachten, dass jeder Konzept-Vektor ein eigenes Momentum verwendet.

1.5.2 Parametrische Schwellwerte

Parametrische Schwellwerte wurden von 2003 von Tani vorgestellt [23]. In dieser Dokumentation wird unter einer Schicht parametrischer Schwellwerte eine Konzept-Schicht verstanden, die eins-zu-eins mit einer Schicht

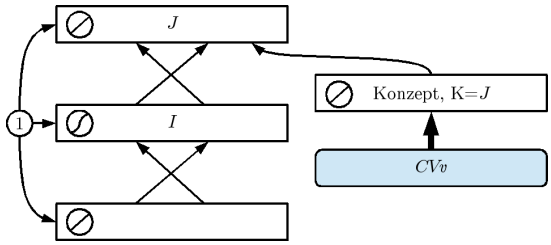


Abbildung 8: Ein MLP mit parametrischen Schwellwerten in der Ausgabeschicht. Es existiert ein parametrischer Schwellwert pro Ausgabeneuron.

eines Netzwerks verknüpft ist[3] wie in Abbildung 8 gezeigt. Jedes Neuron einer solchen Konzept-Schicht stellt in diesem Sinne ein Bias-Neuron für das nachfolgende Neuron dar. In diesem Sinne wird dadurch also der Schwellwert eines Neurons parametrisch bestimmt.

Als parametrischer Schwellwert eines Neurons j kann also die Ausgabe des zugehörigen Konzept-Neurons k mit Aktivierung $o_k(\text{net}_k)$ angesehen werden, wenn für das Gewicht $w_{kj} = 1$ gilt. Analog kann das Gewicht selber als parametrischer Schwellwert angesehen werden, wenn die Ausgabe des Konzept-Neurons 1 entspricht. Der resultierende Schwellwert beträgt in beiden Fällen $w_{kj} \cdot o_k(\text{net}_k)$. Unter Ausnutzung dieser Analogie lassen sich parametrische Schwellwerte wie gewöhnliche Gewichte lernen: Zu Beginn einer Epoche e für Verhalten v werden die Elemente des Konzept-Vektors $c_{k,v} \in \mathbf{CV}_v$ als Eingabe net_k der Konzept-Neuronen verwendet, während die ausgehenden Gewichte w_{ij} auf den Wert 1 gesetzt werden. Zu beachten ist, dass gegebenenfalls das Momentum des Konzept-Vektors für das Lernen der Gewichte verwendet wird. Am Ende der Epoche wird die Änderung der Elemente des Konzept-Vektors gemäß der berechneten Gewichtsänderung bestimmt:

$$c_{k,v}(T_e) := o_k^{-1}\left(w_{kj}(T_e) \cdot o_k(c_{k,v}(T_{e-1}))\right) \tag{22}$$

wobei das Gewicht $w_{kj}(T_e)$ anhand der gewöhnlichen Lernregel 9 gewonnen wird. Um den Erfolg der Invertierung zu garantieren müssen die Bias-Neurone eindeutig invertierbare und unendliche Aktivierungsfunktionen aufweisen. Die lineare Aktivierungsfunktion stellt hier also eine gute Wahl dar.

1.5.3 Parametrische Gewichte zweiter Ordnung

Gewichte zweiter Ordnung können in analoger Vorgehensweise zur direkten Bestimmung verhaltenslokaler Gewichte herangezogen werden (siehe Abbildung 9). Existiert ein Gewicht zweiter Ordnung w_{ikj} mit einem

[3] Die Vorgehensweise weicht in dieser Dokumentation von durch Tani beschriebenen ab, ergibt jedoch ein äquivalentes Lernverfahren.

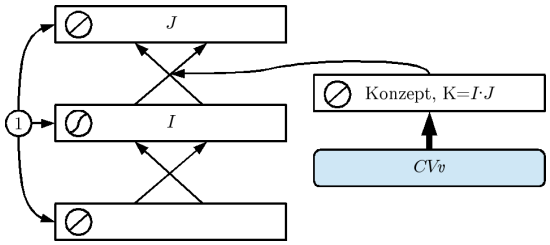


Abbildung 9: Ein MLP mit parametrischen Gewichten zweiter Ordnung zur Ausgabeschicht. Pro parametrischem Gewicht existiert ein Neuron in der Konzept-Schicht.

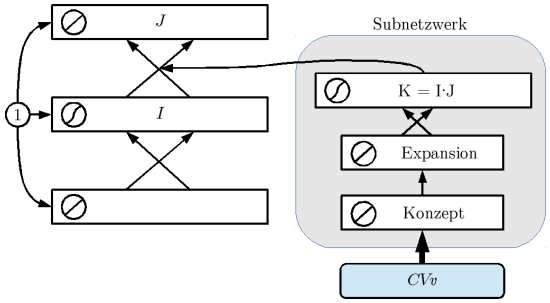


Abbildung 10: Ein MLP mit Gewichten zweiter Ordnung zur Ausgabeschicht und Modulation durch ein Subnetzwerk. Die Konzept-Schicht kann auf eine beliebige Dimension verkleinert werden.

Konzept-Neuron k, so lassen sich die Elemente des Konzept-Vektors für das Verhalten v bestimmen mittels:

$$c_{k,v}(T_e) := o_k^{-1}\left(w_{ikj}^h(T_e)\right) \tag{23}$$

wobei das Gewicht w_{ikj}^h zum Zeitpunkt T_e als *hypothetisches Gewicht* nach dem Training auf Epoche e zu bezeichnen ist, da im eigentlichen Sinne kein Gewichtsparameter existiert. Dieses kann anhand des Rückpropagierungs-Algorithmus (unter Verwendung des Konzept-Momenti) berechnet werden. Wieder muss die Aktivierungsfunktion an jeder Stelle eindeutig invertierbar sein.

Die so durchgeführte direkte Bestimmung von verhaltenslokalen Gewichten kann zu hochdimensionalen Konzeptvektoren führen, beispielsweise wenn eine komplette Vollverknüpfung zwischen zwei Schichten moduliert werden soll. Ein solcher Konzept-Vektor ist schwierig zu analysieren, verhindert eventuell die Bildung einer Metrik im Raum der Konzept-Vektoren, und ist daher – wie sich zeigen wird – nicht unbedingt von Vorteil beim Auffinden adäquater verhaltenslokaler Parameter. Daher kann eine Expansion von

Konzept-Vektoren auf eine größere Anzahl Parameter sinnvoll sein. Dies ist in Abbildung 10 dargestellt: Dort werden Gewichte zweiter Ordnung durch ein von der Sensorik unabhängiges Teilnetzwerk moduliert, welches einen Konzept-Vektor in Form parametrischer Schwellwerte expandiert.

Die Rückpropagierung des Fehlers in das Subnetzwerk folgt nach Formel 3. Die parametrischen Schwellwerte werden dann anhand obiger Formel 22 gelernt.

1.5.4 Parametrische modulierte Gewichte

Unter Verwendung modulierter Gewichte (Abbildung 11) können globale Gewichte verhaltensabhängig moduliert werden. Die Vorgehensweise beim direkten Bestimmen der lokalen Komponenten ist grundlegend ähnlich zur obigen. Allerdings wird ein Anteil der berechneten hypothetischen Gewichts-Änderung auf das globale Gewicht umgewälzt.

Allgemein ergibt sich ein moduliertes Gewicht w_{ikj} nach Gleichung 4 zum Produkt der Aktivierung des Neurons k (lokales Gewicht), und dem Gewichtsparameter w^p_{ikj} (globales Gewicht):

$$w_{ikj} = w^p_{ikj} \cdot o_k \quad .$$

Nun gelte für die globale Gewichtsänderung $\Delta w^p_{ikj}(T_e)$ nach dem Training auf Epoche e:

$$\Delta w^p_{ikj}(T_e) = (1 - \mu) \cdot \Delta w^h_{ikj}(T_e) \tag{24}$$

so dass diese den Anteil $(1 - \mu) \in [0, 1]$ an der hypothetischen gesamten Gewichtsänderung $\Delta w^h_{ikj}(T_e)$ erhält. Durch Umstellung von

$$w^h_{ikj}(T_e) = \Big(w^p_{ikj}(T_{e-1}) + \Delta w^p_{ikj}(T_e)\Big)\ \Big(o_k(T_{e-1}) + \Delta o_k(T_e)\Big)$$

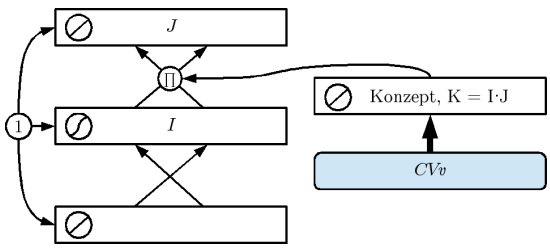


Abbildung 11: Ein MLP mit parametrischen modulierten Gewichten zur Ausgabeschicht.

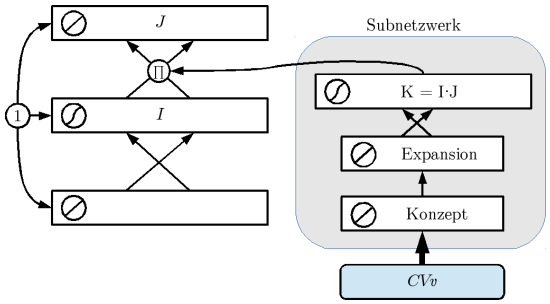


Abbildung 12: Ein MLP mit modulierten Gewichten zur Ausgabeschicht und Modulation durch ein Subnetzwerk.

folgt damit das neue lokale Gewicht $o_k(T_{e-1}) + \Delta o_k(T_e)$ zu:

$$o_k(T_{e-1}) + \Delta o_k(T_e) = \frac{w^h_{ikj}(T_e)}{w^p_{ikj}(T_{e-1}) + (1-\mu) \cdot \Delta w^h_{ikj}(T_e)} \tag{25}$$

so dass sich daraus das Konzept-Element $c^b_{k,v}$ zum Zeitpunkt T_e zu

$$c_{k,v}(T_e) := o_k^{-1}\Big(o_k(T_{e-1}) + \Delta o_k(T_e)\Big) \tag{26}$$

ergibt. Für $\mu = w^p_{ikj} = 1$ ist diese Vorgehensweise vollständig analog zur Bestimmung parametrischer Gewichte zweiter Ordnung, für $\mu = 0$ wird keine Modulation gelernt, wie bei gewöhnlichen Gewichten.

Ein verteiltes Training modulierter Gewichte (Abbildung 12) ist analog zur Vorgehensweise bei Gewichten zweiter Ordnung möglich. Lediglich wird die angestrebte Gewichtsänderung nicht komplett propagiert, sondern gemäß des Anteils μ, wie in Gleichung 6 angegeben.

2 RNNPBlib-Framework

2.1 Einführung

Bei der *Recurrent Neural Networks with Parametric Biases Library* (RNNPBlib) handelt es sich um eine Entwicklungs-Bibliothek für den modularen Aufbau aller künstlichen neuronalen Architekturen, die aus den im vorangegangenen Kapitel erwähnten Komponenten erstellt werden können, wobei es insbesondere auf modulierte Netze mit mehreren Zielfunktionen spezialisiert ist. Für die Komposition der meisten Netzwerke steht eine Anzahl Funktionen bereit, während die Erweiterung um neue Funktionen zugänglich gestaltet ist. Es implementiert zudem alle zuvor beschriebenen Lernalgorithmen und deren Erweiterungen. Zur Vereinfachung der Benutzung umfasst es eine Anwendungs- bzw. Problemschnittstelle, die die Verwendung neuronaler Lernverfahren für die Approximation bekannter oder unbekannter Zielfunktionen erleichtert. Das Erheben von Trainingdaten kann automatisch entweder bei Bedarf (*online*, d.h. während des Lernens) erfolgen, oder bevor ein Netz trainiert wird (*offline*). Dies erleichtert insbesondere die Anbindung an Simulationen oder Messdaten-erhebende Systeme und kann die Komplexität des Lernvorgangs verringern. Ferner stehen Funktionen für das Ausschreiben und Laden von Trainingsdaten sowie für das Auswerten von Netzwerken bereit. Ein Überblick über das Framework und dessen Komponenten ist in Abbildung 13 gegeben.

Die Implementierung des Frameworks wurde so vorgenommen, dass ein angemessener Kompromiss aus Anwendbarkeit und Erweiterbarkeit auf der einen Seite, und Performanz auf der anderen Seite entsteht. Daher ist es komplett objektorientiert in C++ programmiert, so dass das Framework gegenüber späteren Erweiterungen (beispielsweise in Richtung autoadaptiver Topologien oder alternativer Lernverfahren) offen bleibt. RNNPBlib ist sowohl auf 32 und 64-Bit Systemen als auch unter Linux (Makefile/gcc) und Windows (Microsoft Visual Studio) funktionsfähig.

2.2 Anwendungs-Schnittstelle

Anwendungen, welche das RNNPBlib-Framework verwenden, wird die Vererbung von der Interface-Klasse `RNNPB_ApplicationInterface` vorgeschrieben. Sie dient im Wesentlichen der Ausformulierung einer oder mehrerer zu approximierender Funktionen bzw. Verhalten $\mathbf{y}_v(\mathbf{x})$.

2.2.1 Bekannte Zielfunktionen

Die Schnittstelle lässt prinzipiell die Anwendung entscheiden, welche Daten als Eingabe und Ziel-Ausgabe eines neuronalen Netzes pro Zeitschritt gewünscht sind. Dazu müssen folgende Funktionen implementiert werden, die vom später beschriebenen Netzwerk-Trainer angefragt werden:

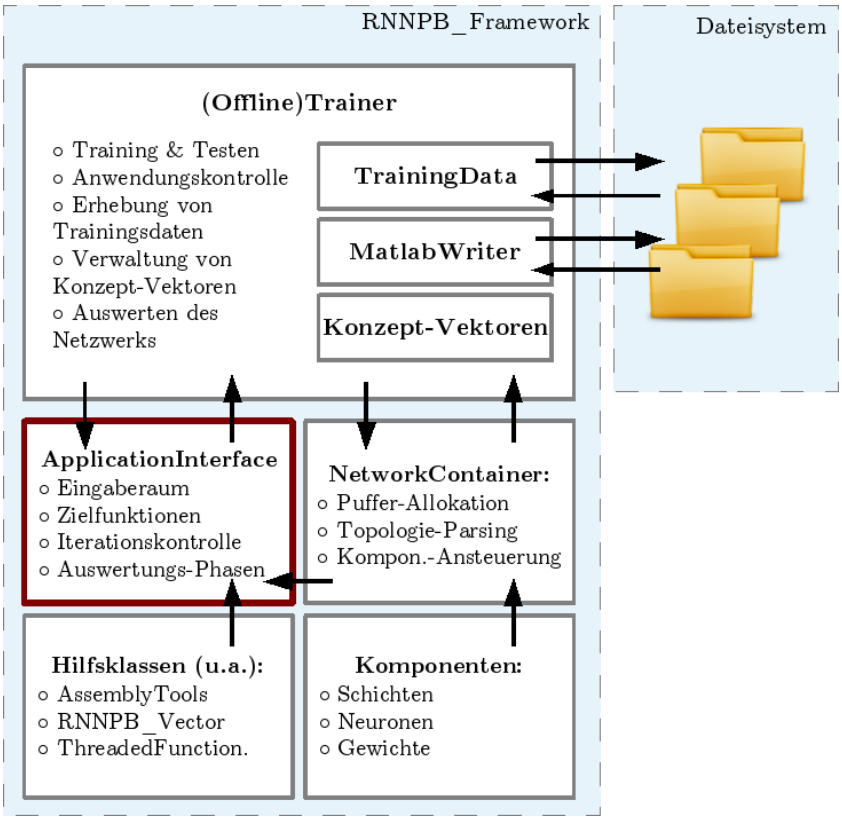


Abbildung 13: Der modulare Aufbau des RNNPBlib-Frameworks sowie ein Überblick über die Zuständigkeiten und Funktionalitäten der Module. In rot dargestellt ist die Anwendungs-Schnittstelle, welche Implementierungen bezüglich der Zielfunktion(en) fordert.

- `virtual void get_nn_input(RNNPB_Vector* returnValue)`:
  Stellt die Eingabe $\mathbf{x}(t)$ des neuronalen Netzes zum aktuellen Zeitpunkt zur Verfügung. Diese kann anhand der Hilfsfunktion `RNNPB_Vector get_app_input()` bestimmt werden, wie in Kapitel 2.2.3 beschrieben wird. Der Index der aktuell zu approximierenden Zielfunktion v ist über `unsigned int get_feedback_method()` abrufbar.

  Der Rückgabewert wird direkt als Funktionsparameter übergeben. Dies hat den Vorteil, dass der Trainer den Speicherbereich im Vorfeld allozieren kann. Ferner ist dadurch möglich, bei einer vor dem Training stattfindenden Generierung von Trainingsdaten (siehe Kapitel 2.3.2) den übergebenen Speicherbereich erst später zu belegen. So ist es möglich, bei Aufruf dieser Funktion Sensordaten von einer Simulation anzufordern, ohne auf diese zu warten.

- `virtual void get_nn_target(RNNPB_Vector* returnValue)`:
  Analog zur obigen Funktion liefert diese Funktion den Zielwert $\mathbf{y}_v(\mathbf{x}(t))$ des Netzwerks zum aktuellen Zeitpunkt, um den Fehler berechnen zu können. Hierzu kann erneut die Hilfsfunktion `get_app_input()` verwendet werden.

Die Implementierung dieser beiden Funktionen reicht für mathematisch formulierbare Probleme aus.

2.2.2 Unbekannte Zielfunktionen

Für Zielfunktionen, für die keine mathematische Formulierung bereitsteht, müssen während des Lernens oder während der Generierung von Trainingsdaten Simulationen oder Messsysteme herangezogen werden. Im Framework existieren daher Funktionen zur Initialisierung und Beendung von Epochen (in diesem Sinne: Messreihen) und die Ausführung von Aktionen zwischen einzelnen Zeitschritten, die bei Bedarf zusätzlich zu den obigen Funktionen überladen werden können:

- `virtual void action(RNNPB_Vector output)`: Diese Funktion kann verwendet werden, um eine Aktion durchzuführen, bevor der Zielwert der zu approximierenden Funktion bestimmt wird (d.h. zwischen `get_nn_input` und `get_nn_target`).

  Eine solche Aktion kann beispielsweise die Ansteuerung einer Aktorik anhand der aktuellen Netzausgabe `output` bzw. $\mathbf{o}(t)$ sein. Werden Trainingsdaten im Vorfeld generiert steht keine solche Ausgabe zur Verfügung, der obige Vektor ist dann leer. Unabhängig davon können in dieser Funktion Simulationsschritte durchgeführt oder ein Messsystem kalibriert werden.

- `virtual void initEpoch()`: Diese Funktion wird *vor* jeder Epoche aufgerufen und kann implementiert werden, um diese auf Anwendungsseite zu initialisieren, beispielsweise indem eine Simulation gestartet oder zurückgesetzt wird.

- `virtual void stopEpoch()`: Analog wird diese Funktion *nach* jeder Epoche aufgerufen und dementsprechend implementiert, beispielsweise indem eine Simulation beendet wird.

2.2.3 Iteration durch den Eingaberaum der Zielfunktionen

Als Hilfsfunktion für die Bestimmung des aktuellen Netzeingabe-Vektors steht der Anwendung eine Funktion `RNNPB_Vector get_app_input()` bereit. Sie liefert einen Vektor $\mathbf{x}(t)$ zurück, welcher direkt oder indirekt (z.B. nach Skalierung) als Variable einer formulierbaren zu approximierenden Funktion dienen kann, das heißt als Eingabe für ein neuronales Netzwerk. Das mehrdimensionale Intervall dieser Variable wird dem Anwendungs-Konstruktor per `RNNPB_Vector min_iteration` und `RNNPB_Vector max_iteration` mitgeteilt.

Dabei kann zwischen *zufälliger* und *sequentieller Iteration* durch den Eingaberaum gewählt werden (im Konstruktor per `bool random_sampling`): Wird zufällige Iteration gewählt, ist $\mathbf{x}$ ein zufälliges Element aus dem hyperkubischen Eingaberaum der Zielfunktion, welcher durch das angegebene Intervall aufgespannt wird. Bei sequentieller Iteration wird dagegen nacheinander in gleichen Abständen durch alle Dimensionen des Eingaberaums iteriert. Dies erfolgt im Sinne einer n-fach verschachtelten `for`-Schleife, wobei n die Dimension des Vektors $\mathbf{x}$ ist, so dass der Eingaberaum gleichverteilt abgedeckt wird.

In beiden Fällen ist eine maximale Anzahl Iterationen pro Epoche zu wählen. Dies erfolgt im Konstruktor unter `unsigned int set_nr_of_iterations`. Bei zufälligem Sampling trägt dieser Parameter die Bedeutung der maximalen Anzahl Iterationen *insgesamt*, bei sequentiellem Sampling dagegen *pro Dimension*. Eine Epoche kann gegebenenfalls frühzeitig beendet werden, indem die Abbruchbedingung (`bool abortEpochConditionMet()`) überladen wird. Der aktuelle Interationsschritt t kann mittels `unsigned int get_iteration()` abgefragt werden.

Zusätzlich kann eine ganze Epoche wiederholt werden, falls diese als nicht erfolgreich erachtet wird (z.B. aufgrund einer fehlgeschlagenen Simulation, zu hoher Messfehler oder korrupter Sensordaten). In diesem Fall wird die Zielfunktion nicht gewechselt, eventuell erhobene Daten und Lernfortschritte werden verworfen. Die Umsetzung erfolgt durch Überladung der Funktion `bool epochGoalReached()`.

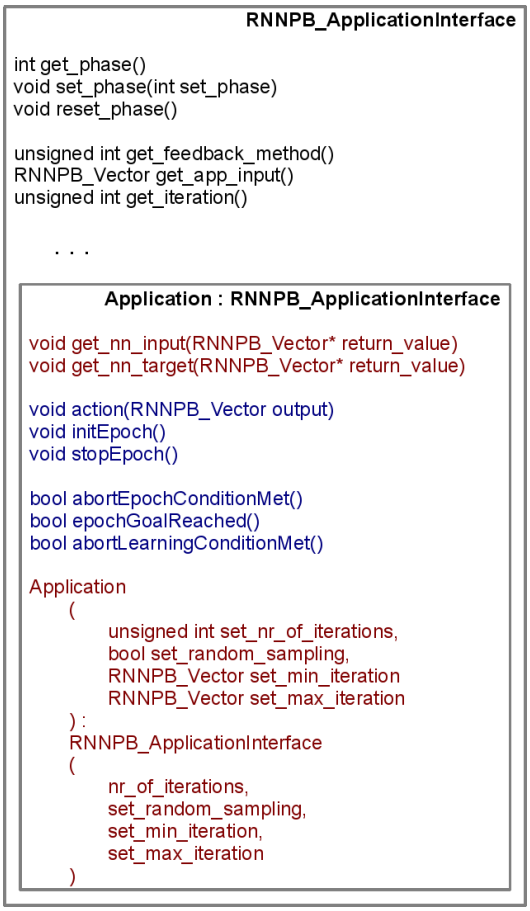


Abbildung 14: Die Anwendungs-Schnittstelle stellt unter anderem Funktionen für die Unterscheidung von Phase und Zielfunktion und für die Abdeckung des jeweiligen Eingaberaums bereit. Eine Anwendung muss die in rot gezeigten Funktionen implementieren und kann optional die blauen Funktionen überladen.

Per Überladung der generellen Abbruchbedingung für das Lernen `bool abortLearningConditionMet()` kann das Trainieren nach eigenen Kriterien vorzeitig beendet werden.

2.2.4 Phasen-Unterscheidung

Für komplexere Anwendungen, sowie zur Durchführung verschiedenartiger Experimente kann es notwendig sein, die Implementierungen obiger Funktionen situationabhängig zu wählen. Das Anwendungs-Interface stellt dazu eine Phasen-Unterscheidung bereit. Die aktuelle Phase lässt sich per `int get_phase()` abfragen. Generell sind die Phasen 0, 1 und 2 vorkonfiguriert. Sie werden vom Trainer bestimmt und tragen folgende Bedeutungen:

- **Phase 0**: Trainingsdaten werden *generiert*, es wird nicht auf die Daten trainiert. Diese Phase existiert nur bei Verwendung des Offline-Trainers, welcher im nachfolgenden Kapitel vorgestellt wird.

- **Phase 1**: Ein neuronales Netz wird *trainiert*. Dazu werden entweder in Phase 0 erstellte Trainingsdaten herangezogen (beim Offline-Trainer), oder diese in Echtzeit von der Anwendung erhoben (Online-Trainer).

- **Phase 2**: Ein neuronales Netzwerk wird *getestet*. Dabei findet kein Lernen statt und es wird nur die Abweichung von Soll-Werten gemessen. Diese werden genauso wie die Eingabe eines Netzes generell von der Anwendung geholt (auch beim Offline-Trainer). Die Anwendung kann folglich neue Daten generieren oder aus einem Satz noch nicht gelernter Trainingsdaten wählen, um die Generalisierung zu testen.

Alle weiteren Phasen > 2 können manuell belegt und von der Anwendung beliebig interpretiert werden. Die Phase kann über `void set_phase(int set_phase)` eingestellt und anhand `void reset_phase()` wieder der automatischen Kontrolle durch den Trainer überlassen werden.

2.3 Netzwerk-Trainer

Die Klasse `RNNPB_CVTrainer` regelt das überwachte Trainieren eines Netzwerks und kann insbesondere zwischen verschiedenen Zielfunktionen bzw. Verhalten unterscheiden, welche mit dem selben Netz erlernt werden sollen. Die Initialisierung des Trainers erfolgt unter Angabe eines Netzwerk-Containers (siehe dazu Kapitel 2.4.3), der Anwendung, sowie einer initialen Anzahl von Verhalten/Zielfunktionen. Prinzipiell können Trainingsdaten entweder on-demand von der Anwendung erhoben werden (Online-Trainer), oder bevor trainiert wird (Offline-Trainer).

2.3.1 Online-Trainer

Trainieren und Testen: Das Trainieren der Gewichte eines Netzes kann mittels der Funktion `trainNetwork(unsigned int nr_of_runs, int behavior = -1, unsigend int print_runs = 10)` unter Verwendung der Implementierung der zuvor genannten Anwendungs-Funktionen erfolgen. Der Parameter `nr_of_runs` gibt an, wie oft das Lernen aller Verhalten maximal wiederholt wird. Ein optionaler Parameter `behavior` > -1 gibt dabei an, ob nur auf ein einzelnes Verhalten trainiert werden soll, standardmäßig wird auf alle Verhalten abwechselnd trainiert. `print_runs` gibt an, nach wievielen gelernten Epochen pro Verhalten Informationen über den aktuellen Lernfortschritt ausgegeben werden sollen (standardmäßig 10). Der Trainer wechselt beim Aufruf der Funktion in Phase 1, sofern keine manuelle Phase eingestellt ist.

Die Funktion `testNetwork(unsigned int nr_of_runs, int behavior = -1, unsigend int print_runs = 10)` ist analog zur obigen gestaltet, jedoch mit dem Unterschied, dass das Netzwerk keine Parameteränderung erfährt. Dazu wechselt die Anwendung in Phase 2. Bei entsprechender Phasenunterscheidung obiger Anwendungs-Funktionen kann so die Funktion und Generalisierung eines Netzwerks getestet werden.

Durchführung eines Iterationsschritts: In jedem Zeitschritt des Lernens t holt der Trainer zuerst die Eingabe $\mathbf{x}$ von der Anwendung und speist diese in die Eingabeschicht des Netzes. Anschließend wird die Ausgabe des Netzwerks berechnet und gegebenenfalls die Anwendung per `setAction` angewiesen, diese in eine Aktion umzusetzen. Nachfolgend wird eine Soll-Ausgabe von der Anwendung angefordert, anhand welcher der Netzwerk-Fehler berechnet wird, der nach dem Prinzip des Batch-Gradientenabstiegs über eine Epoche kumuliert wird.

Organisation und Training von Konzept-Vektoren: Der Trainer verwaltet für jede per Konstruktor angegebene Konzept-Schicht pro Verhalten einen Konzept-Vektor sowie ein Konzept-Momentum. Der Konzept-Vektor dient so lange als Eingabe der zugehörigen Konzept-Schicht, bis das Verhalten gewechselt wird (in der Regel eine Epoche). Analog wird das Konzept-Momentum beim Wechseln des Verhaltens an die der Konzept-Schicht nachfolgenden Gewichte übergeben, um das verhaltenslokale Momentum wiederherzustellen. Das Erlernen der lokalen Konzept-Vektoren und Momentum erfolgt gemäß der in Kapitel 1.5 angegebenen Lernverfahren am Ende eine Epoche. Die Repräsentation der Konzept-Vektoren ist dynamisch erweiterbar, es stehen also Funktionen bereit, die neue Verhalten auch nach der Konstruktion von Trainer und Netzwerk hinzufügen können.

2.3.2 Offline-Trainer

Der Offline-Trainer `RNNPB_CVOfflineTrainer` erhebt als Erweiterung des on-demand Trainers die Daten nicht erst bei Bedarf, sondern in einer gesonderten, dem Lernen vorgezogenen Phase 0. Dazu stellt er Funktionen für das Generieren (`generateAllTrainingData(unsigned int nr_of_epochs )`, das Abspeichern (`saveTrainingData(const char* filename)`) und das Laden (`loadTrainingData(const char* filename)`) von Trainingsdaten bereit.

Der Offline-Trainer lernt die Verhalten ebenfalls abwechselnd. Da jedoch (anhand `nr_of_epochs`) mehrere Epochen pro Verhalten generiert worden sein können, heißt das, dass nach einer Epoche eines Verhaltens eine Epoche eines anderen Verhaltens gelernt wird, bis alle Epochen einmal herangezogen wurden. Der Offline-Trainer berücksichtigt dabei, dass nicht für alle Verhalten gleich viele Epochen bereitstehen müssen. Im Unterschied zum Online-Trainer tragen die Lern- bzw. Testfunktions-Parameter `nr_of_epochs` und `print_runs` daher die Bedeutung, wie oft auf alle *Epochen* trainiert und der Lernfortschritt ausgegeben wird.

Iteration und Erhebung von Trainingsdaten: Die Abfolge der Anwendungs- und Netzwerkansteuerung erfolgt analog zu Kapitel 2.3.1. Jedoch wird bei Generierung von Trainingsdaten (Phase 0) auf eine Verwendung des Netzwerks verzichtet, und die von der Anwendung zurückgegebenen Trainingsdaten stattdessen in einem Container abgelegt, während beim Lernen (Phase 1) diese Daten als Eingabe- und Zielwerte verwendet werden, anstatt diese von der Anwendung zu beziehen. Beim Testen eines Netzwerks wird jedoch – wie erwähnt – genau wie beim Online-Trainer die Anwendung konsultiert.

Trainingsdaten-Container: Der Offline-Trainer organisiert einen Trainingsdaten-Container `RNNPB_TrainingDataContainer`, welcher für eine beliebige Anzahl Verhalten jeweils eine beliebige Anzahl Epochen mit jeweils einer beliebigen Anzahl Trainingsdaten enthält. Die Datenstruktur wird bei Schreibzugriff implizit dynamisch erweitert, so dass das Hinzufügen neuer Daten möglich ist. Ebenfalls existieren Funktionen für das Überschreiben oder Löschen existierender Epochen und Trainingsdaten. Das Laden und Speichern des Containers ist über Boost [24] realisiert. Die C++-Bibliothek `boost` ist eine umfangreiche Sammlung zuverlässiger Bibliotheken, die die Möglichkeiten der Programmiersprache ausbaut und unter Anderem die Serialisierung von Datenstrukturen und Speicherung in Archiven unterstützt.

Um die Konsistenz der Daten in Hinblick auf die Lernbarkeit mittels neuronaler Netze nötigenfalls zu korrigieren, stellt der Container eine Funktion `filterTrainingData(double threshold, int behavior =`

`-1)` bereit, welche in den Trainingsdaten eines Verhaltens (für `behavior` > -1) oder separat für jedes Verhalten Paare sucht, die für hinreichende Ähnlichkeit der Netzwerk-Eingabe eine hinreichende Divergenz der Netzwerk-Ausgabe aufweisen. Solche Daten sind mit gewöhnlichen neuronalen Netzen nicht lernbar, da diese gleiche Eingangsdaten nicht auf unterschiedliche Ausgaben abbilden können. Mittels rekurrenter neuronaler Netze können diese Inkonsistenzen unter der Voraussetzung unterschiedlicher Daten-Historien mit intertemporalen Abhängigkeiten jedoch aufgelöst werden.

Die Konsistenz-Filterung löscht jedes bis auf das erste Vorkommnis eines Trainings-Datums $(\mathbf{x}_{e,v}(t), \mathbf{y}_v(\mathbf{x}_{e,v}(t)))$ für Verhalten v und Epoche e, für das gilt:

$$\frac{||\mathbf{y}_v(\mathbf{x}_{e,v}(t)) - \mathbf{y}_v(\mathbf{x}_{e',v}(t'))||}{||\mathbf{x}_{e,v}(t) - \mathbf{x}_{e',v}(t')||} > \theta, e \neq e' \vee t \neq t' \tag{27}$$

wobei der Schwellwert für das Löschen inkonsistenter Daten θ (bzw. Funktionsparameter `threshold`) von der Größenordnung der Eingaben und Soll-Ausgaben der Trainingsdaten abhängig ist. Der Schwellwert kann heuristisch bestimmt werden: Wenn ein nicht-rekurrentes neuronales Netz bei Filterung der Daten anhand eines relativ niedrigen Thresholds (das heißt unter Filterung weniger Daten) nach dem Lernen eine deutliche Verbesserung des Fehlers aufweist, trägt die Konsistenzfilterung wahrscheinlich zum Lernen einer angemessenen Approximation der Zielfunktion bei.

2.3.3 Auswertung neuronaler Lerner

Neben der Ausgabe des Lernfortschritts auf der Konsole wurde für die Auswertung neuronaler Lerner eine Klasse `RNNPB_MatlabWriter` geschrieben. Bei Bedarf schreibt sie während des Trainings Informationen über den mittleren quadratischen Fehler aller Epochen *jedes* Verhaltens (MMSEb_i) und aller Epochen *aller* Verhalten (MMSEbAll, später Durchlauf genannt) sowie über Konzept-Vektoren und Momentum in ein per Matlab lesbares Dateiformat. Ferner steht eine Instanz der Klasse im Trainer öffentlich zur Verfügung, so dass eine Anwendung weitere Daten ausschreiben kann. Der `RNNPB_MatlabWriter` bietet dazu insbesondere die Template-Funktion `template <typename T> write(string name, T value)`, welche die per `value` übergebenen Daten in eine Datei, die Informations-Bezeichner `name` enthält, schreibt. Das Template `T` kann von jedem Typ sein, der den Standard-Einfügeoperator `std::ostream::operator<<` (siehe [25]) implementiert. Auch der Framework-eigene Vektor-Datentyp `RNNPB_Vector` (siehe 2.6.1) wird unterstützt. Die erzeugten Dateien befinden sich generell im Verzeichnis `bin/matlab` oder einem wählbaren Unterverzeichnis.

Intern organisiert der `RNNPB_MatlabWriter` ein Mapping von Informations-Bezeichnern zu Datei-Handles (als assoziativer Template-

Container `std::map` nach C++ Standard [26]). Als Informations-Bezeichner können alle beliebigen (dateisystemkonformen) Strings gewählt werden. Beim ersten Zugriff auf einen Informations-Bezeichner wird implizit eine Datei erstellt oder geöffnet, welche für die Laufzeit des `RNNPB_MatlabWriter`-Objekts Schreibzugriffe ermöglicht.

Die generierten Dateien tragen den Namen der Informations-Bezeichner, sowie ein Präfix und ein inkrementelles Suffix, letztere aus der Datei `bin/matlab/exp.config`. Anhand des Präfixes können ausgeschriebene Informationen erstens in einem Unterverzeichnis von `bin/matlab` platziert werden, zweitens kann eine Kennzeichnung des aktuellen Versuchs angegeben werden. Das Suffix wird angehängt und bei jedem Start eines `RNNPB_MatlabWriter`s (d.h. bei Konstruktion eines Trainers) inkrementiert, um Daten verschiedener sequentieller oder paralleler Instanzen des Netzwerk-Trainers bei der Versuchsauswertung unterscheiden zu können.

2.4 Architektonische Funktionen

Das RNNPBlib-Framework umfasst neben der Anwendungsschnittstelle und den Trainern eine Reihe topologischer Funktionen, die es ermöglichen, alle aus Kapitel 1 bekannten neuronalen Architekturen zu generieren. Ferner wurden ein vereinfachender Vektor-Datentyp, diverse Funktionen für die Implementierung nebenläufiger Funktionen, sowie einige häufig benötigte Hilfsfunktionen geschrieben.

2.4.1 Aufbau eines Netzwerkes

In der Regel werden neuronale Netze in Schichten organisiert. Daher bietet auch das RNNPBlib-Framework eine Klasse `RNNPB_NeuronLayer`, die eine Schicht Neuronen erstellt. Sie fordert per Konstruktor die Anzahl Neuronen, und optional den Typ der Aktivierungsfunktionen und ihre Skalierung.

Prinzipiell müssen nur Eingangs-, Konzept- und Ausgabeneurone in Schichten organisiert werden, da diese vom Trainer angesteuert werden und somit als Zugriffspunkte dienen. Die Topologie eines Netzwerks ist auf programmtechnischer Ebene allein durch die beliebige Verkettung von Objekten realisiert, so dass gerichtete Graphen jeder Form realisiert werden können. Die Neuron-Klasse des Frameworks `RNNPB_Neuron` verwaltet dazu eine Liste aller eingehenden und ausgehenden Gewichte (Unterklassen von `RNNPB_Weight`). Gewichte und Neurone haben generell einen lokalen Zuständigkeitsbereich, so dass sie beim Lernen benötigte Informationen (Aktivierungen, Fehlerterme, Gewichts-Parameter) selbstständig durch rekursive Aufrufe anfordern. Alle rekursiven Aufrufe sind aus Effizienzgründen puffernd umgesetzt.

Die Anordnung verdeckter Neurone in Schichten ist dennoch vorteilhaft,

da es die Verknüpfung von Netzwerken durch die Funktionen der Klasse `RNNPB_AssemblyTools` erleichtert. Sie ermöglicht die Generierung aller im Rahmen dieser Arbeit erwähnten neuronalen Topologien und Kombinationen daraus mittels der folgenden Funktionen:

- `connectLayersSecondorderCV`: Verknüpft zwei Schichten mit Gewichten zweiter Ordnung und erstellt dazu eine Konzept-Schicht mit entsprechender Größe. Es muss eine Liste von Konzept-Schichten (`std::vector <RNNPB_NeuronLayer*>* cl_container`) übergeben werden, welche um die erstellte Schicht erweitert wird.
- `connectLayersModulatedCV`: Analog, jedoch mittels modulierter Gewichte.
- `addParametricBiasToLayer`: Analog, jedoch mittels parametrischer Schwellwerte.
- `connectLayersSecondorder`: Verknüpft zwei Schichten unter Angabe einer Schicht, welche als Modulation dient. Diese muss die korrekte Größe aufweisen.
- `connectLayersModulated`: Analog, jedoch mittels modulierter Gewichte.
- `connectLayers`: Vollverknüpfung zwischen zwei Schichten.
- `connectLayersConstant`: Konstante Vollverknüpfung zwischen zwei Schichten. Diese Gewichte können nicht gelernt werden.
- `connectLayersDelayed`: Rekurrente Vollverknüpfung zwischen zwei Schichten oder von einer Schicht zu sich selbst. Der Parameter λ kann angegeben werden.
- `addBiasToLayer`: Erstellt ein Bias-Neuron und verknüpft dies mit den Neuronen der angegebenen Schicht.
- `addRecurrency`: Erstellt eine einzelne Rekurrenz zwischen zwei (nicht zwingend unterschiedlichen) Neuronen.
- `connectReservoir`: Erstellt rekurrente Gewichte zwischen allen Paaren aus nicht-gleichen Neuronen einer Schicht. Der Parameter λ kann angegeben werden.
- `addSelfInhibitionToLayer`: Verknüpft jedes Neuron einer Schicht rekurrent mit sich selbst.

Die Klasse `RNNPB_AssemblyTools` ist leicht um weitere Funktionen erweiterbar. Grundsätzlich können erstellte Topologien auch während der Laufzeit um einzelne Neuronen und Gewichte, Schichten oder Reservoirs erweitert werden, wodurch sich das Framework für autoadaptive Topologien eignet.

2.4.2 Neuronen- und Gewichtstypen

Der Neuron-Typ des Frameworks ist als Klasse `RNNPB_Neuron` implementiert. Die bei Konstruktion gewählte Aktivierungsfunktion entscheidet über den Typ des Neurons (monotoner Diskriminator oder radiale Basisfunktion). Bei der Berechnung der Netzeingabe und Aktivierung eines Neurons werden Sprungtabellen für die gewählten Typen verwendet (`switch`es). Zur Wahl stehen die in der Datei `RNNPBlib/RNNPB_NeuronActivationTypes.h` enumerierten Aktivierungsfunktionen (Vergleich siehe Kapitel 1.2.1). Die Funktionalität des abstrakten Neurons ist überwiegend Framwork-intern, da die Ansteuerung eines neuronalen Netzes anhand des Trainers, des Netzwerk-Containers und der Eingabe- und Ausgabeschichten zugänglich gemacht ist. Die verschiedenen Gewichts-Typen wie in Kapitel 1.2.2 beschrieben sind vom abstrakten Typ `RNNPB_Weight` abgeleitet. Ausgehend von diesem wurde eine Klassenhierarchie aufgestellt.

2.4.3 Netzwerk-Container

Der Netzwerk-Container `RNNPB_NetworkContainer` ist eine Klasse für die Zusammenfassung und effiziente Ansteuerung eines künstlichen neuronalen Netzes. Sie stellt Funktionen für den Trainer bereit, die ihn bei der Durchführung einzelner Lernschritte und die daraus resultierenden Netzparameter-Änderungen unterstützen. Ein Netzwerk-Container kann unter Angabe der Eingabe-, Ausgabe- und Konzept-Schichten des Netzes erstellt werden. Ausgehend von diesen führt die Klasse jeweils eine rekursive Topologie-Analyse durch, was einerseits eine zusammenhängende Pufferallokation und andererseits die Auffindung einer Subnetzwerk-Hierarchie ermöglicht. Der Netzwerk-Container bietet ferner Funktionen für das Beschränken des Trainings auf Teilmengen der Netzparameter.

Topologie-Analyse und Puffer-Allokation: Alle Neurone und Gewichte des RNNPBlib-Frameworks bestimmen ihre Vorwärts- und Rückpropagierung in eigener Zuständigkeit anhand ihrer Vorgänger bzw. Nachfolger. Bei Abruf der Ausgabe eines Netzes wird also anhand der Topologie rückwärts bis zur Eingabeschicht abgefragt, welche Netzeingaben und Aktivierungen sich für alle verbundenen Neurone ergeben. Diese Werte werden nur einmal berechnet, gepuffert, und bei Obsoleszenz verworfen. Daraus folgt auch eine bestimmte Reihenfolge beim Auslesen der Pufferspeicher der Neurone, angefangen bei der Eingabeschicht. Auch die Delta-Terme werden gepuffert, also einmalig pro Zeitschritt berechnet. Analog beginnt das Auslesen der Delta-Terme der Neurone bei der Ausgabeschicht.

Um die Lokalitätseigenschaften der Pufferspeicher zu optimieren, führt der Netzwerk-Container bei Konstruktion eine Topologie-Analyse durch: Je-

des Neuron registriert sich einmal in gleicher Reihenfolge wie bei der Berechnung von Aktivierungen – also beginnend bei der Eingabe – und einmal in gleicher Reihenfolge wie bei der Berechnung von Deltas – beginnend bei der Ausgabe – beim Container, was jeweils über rekursive Aufrufe erreicht wird. Gewichte registrieren sich nur einmalig beim Container. Da für das Trainieren von Gewichten sowohl die Ausgabe des Vorgängers, als auch das Delta des Nachfolgers benötigt wird, werden die Pufferspeicher für Aktivierungen und Netzeingaben in der Reihenfolge der ersten, vorwärtsgerichteten Registrierung der Neurone, und die Pufferspeicher der Deltas in der Reihenfolge der zweiten, rückwärtsgerichteten Registrierung in zusammenhängenden Speicherbereichen alloziert. Bei Zugriff auf ein Element eines Puffers wird der gesamte diesbezügliche Speicherblock vom Prozessor gecacht, so dass bei Zugriff auf subsequente Elemente mit hoher Wahrscheinlichkeit kein weiterer Lesezugriff auf den vergleichsweise langsamen Hauptspeicher durchgeführt wird.

Subnetzwerk-Hierarchie: Aus der von der Ausgabeschicht ausgehenden Topologie-Analyse ergibt sich ferner die Möglichkeit, ein Netzwerk in eine Hierarchie von (Sub-)Netzwerken zu zerlegen: Alle Netzkomponenten, die lediglich an der Modulation eines Obernetzwerks beteiligt sind, also keine Eingabeinformationen desselben verarbeiten, können als unabhängiges Netzwerk betrachtet werden. Das RNNPBlib-Framework kann dies an jedem Paar Schichten, das mittels Gewichten zweiter Ordnung oder modulierten Gewichten miteinander verknüpft ist, erkennen, vorausgesetzt, diese sind in Schichten organisiert. Die Konzept-Schicht dient dabei als Ausgabe-Schicht des Subnetzwerks. Auf diese Weise kann ein Netzwerk hierarchisch gegliedert werden, so dass alle Subnetzwerke eines Obernetzes unabhängig und parallel berechnet werden können, bevor das Obernetz berechnet wird. die hierarchische Zerlegung eines Netzwerks lohnt sich theoretisch ab zwei Subnetzwerken von nicht-trivialer Größe.

2.5 Framework-Parametrisierung

Die initiale Parametrisierung von Netzwerkkomponenten, die verwendeten Lernverfahren und ihre Einstellungen sowie einige laufzeitbezogene Konfigurationen befinden sich in der Datei `src/RNNPBlib/RNNPB_Definitions.h`. Alle dort angegebenen Einstellungen sind als Präprozessoranweisungen implementiert, so dass auf dem Framework basierende Programme Laufzeitoptimal kompiliert werden können, da nicht benötigte Codefragmente nicht mitkompiliert werden.

Die laufzeitbezogenen Präprozessor-Direktiven umfassen:

- `#define ENABLE_DEBUG`: Wenn definiert, aktiviert dies die Ausgabe von Debugging-Informationen, beispielsweise bei fehlerhaften Berechnungen oder eventuell unvollständigen Architekturen.

- `#define ENABLE_RUNTIME`: Aktiviert die Ausgabe von Laufzeit-Informationen. Es werden Informationen zur aktuellen Geschwindigkeit des Trainers in Datensamples und gelernten Gewichten pro Sekunde (*Connections Per Second*, CPS) angegeben.

- `#define ENABLE_LOOKUP_TABLES`: Aktiviert die Verwendung von Lookup-Tables für Funktionen, bei denen das Nachschlagen eines Ergebnisses anstelle der Neuberechnung einen Laufzeitvorteil erzielt. Dies sind insbesondere die sigmoiden Aktivierungsfunktionen und deren Ableitungen. Die Lookup-Tables werden standardmäßig für alle skalierten Netzeingaben ($s \cdot \text{net}_j$) zwischen -10 und 10 verwendet. Es werden in diesem Intervall 1.000.000 Werte berechnet, so dass der Präzisionsverlust zu vernachlässigen ist. Sowohl Intervall und Präzision sind jedoch einstellbar. Die Verwendung von Lookup-Tables beschleunigt das Lernen eines Netzes um bis zu 50%. Die Lookup-Tables werden vom Netzwerk-Container generiert.

- `#define ENABLE_EXP_APPROX`: Aktiviert die schnelle Berechnung der Exponential-Funktion nach Schraudolph [27]. Dieses Verfahren kann alternativ oder zusätzlich zu Lookup-Tables verwendet werden. Der Performanzvorteil dieser Direktive ist mit den Lookup-Tables vergleichbar. Es ist jedoch zu bemerken, dass die Exponential-Approximation relativ ungenau ist, daher wird die Verwendung nur im flachen Bereich der Aktivierungsfunktion (außerhalb von $[-10, 10]$ empfohlen), d.h. zusätzlich zu Lookup-Tables.

- `#define ENABLE_SUBNET_TREE`: Aktiviert die Erstellung von Subnetzwerk-Hierarchien (siehe oben) und das parallele Lernen aller Subnetzwerke. Der Performanz-Vorteil dieser Direktive ist stark abhängig von der Topologie.

- `#define ENABLE_EXPERIMENT`: Aktiviert das standardmäßige Ausschreiben von Informationen bezüglich des Lernfortschritts (siehe 2.3.3) mittels des `RNNPB_MatlabWriter`s.

Der Batch-Gradientenabstieg kann unter anderem mit den folgenden Anweisungen für die Erweiterungen aus Kapitel 1.3.5 und 1.3.6 konfiguriert werden:

- `#define ENABLE_BPTT`: Aktiviert Batch Backpropagation Through Time für rekurrente Gewichte.

- `#define ENABLE_ANNEALING`: Aktiviert simulierte Abkühlung. Diese ist anhand des Parameters `ANNEALING_FACTOR` zu parametrisieren, welcher der Abklingkonstante γ entspricht.

- `#define ENABLE_SEPARATE_ANNEALING`: Aktiviert die separate simulierte Abkühlung. Zusätzlich zu `ANNEALING_FACTOR` ist der Gewichts-Änderungs-Korridor per `MIN_WEIGHT_UPDATE` und `MIN_WEIGHT_UPDATE` zu parametrisieren.
- `#define ENABLE_WEIGHT_DECAY`: Aktiviert Gewichts-Dämpfung. Zusätzlich ist der Faktor `WEIGHT_DECAY` anzugeben, welcher dem Faktor β entspricht.
- `#define ENABLE_FLAT_SPOT_ELIM`: Aktiviert Flat Spot Elimination. Es ist der Parameter `MINIMUM_DIFFERENTIAL` anzugeben, welcher der unteren Schranke für die Ableitung einer Aktivierungsfunktion entspricht.

Die Parameter der Neuronen und Gewichte werden generell pro Komponente verwaltet. Dennoch werden sie inital festgelegt anhand:

- `#define DEFAULT_X_STEEPNESS`: Beschreibt die initiale Skalierung s der Aktivierungsfunktion mit Namen `X`. Diese kann für alle Aktivierungsfunktionen mit Schwellwerteigenschaften oder Radien separat festgelegt werden. Bei Bedarf kann auch die Steigung linearer und quasi-linearer Aktivierungsfunktionen eingestellt werden.
- `#define DEFAULT_INIT_WEIGHT`: Gewichte werden generell anhand dieser Definition initialisiert. Standardmäßig werden Gewichte mit $0 \pm \xi$ initialisiert, wobei ξ einem normalverteilten Rauschterm entspricht.
- `#define DEFAULT_INIT_RBFCENTER`: Gewichte, welche als partielle Zentren radialer Neuronen dienen, sollten gleichverteilt in einem adäquaten Eingaberaum initialisiert werden.
- `#define DEFAULT_INIT_X_CV`: Konzept-Vektoren können je nach nachfolgendem Gewichts-Typ X initialisiert werden. Standardmäßig werden alle Konzept-Vektoren gleich und ohne Rauschen initialisiert, um eine konzeptuelle Vergleichbarkeit verschiedener Verhalten zu gewährleisten.

2.6 Hilfsfunktionen

2.6.1 Vektor-Datentyp

Der `RNNPB_Vector` ist ein vollständiger und effizienter Vektor-Datentyp. Er liegt garantiert zusammenhängend im Speicher und implementiert diverse Operatoren wie Vektorprodukt, Vektorlänge, die euklidische Distanz oder den Winkel zu einem anderen Vektor, und viele weitere. Frameworkinterne Berechnungen werden, wo möglich, anhand dieses Typs durchgeführt.

Die Klasse steht folglich auch Anwendern des RNNPBlib-Frameworks zur Verfügung.

2.6.2 Parallele Funktionen

Die Template-Klasse `<typename retT, typename contT> RNNPB_ThreadedFunction` bietet die Möglichkeit, Funktionen beliebiger Klassen nicht-blockierend auszuführen. Sie verwaltet die funktionsinterne Thread-Sicherheit, das virtuelle Aufrufen und Zusammenführen des Funktions-Threads, und garantiert, dass nur eine Instanz der Funktion pro Objekt läuft. Die Parallelisierung der Subnetzwerke ist anhand dieses Typs realisiert. Die Klasse benötigt die Angabe des Rückgabe-Templates `retT`, Kenntnis über das Kontext-Objekt, das die Funktion beinhaltet per `setBase(contT* set_base)`, und die Implementierung der virtuellen Funktion `virtual retT func()`, welche die zu parallelisierende Funktionalität enthält. Der Funktionsaufruf kann mit `void call()` gestartet werden, mit `void join()` wird auf dessen Beendung gewartet. Nach der Beendung steht gegebenenfalls per `retT get()` eine Rückgabe bereit. Wenn die Funktion beim Aufruf von `get()` noch nicht beendet ist, wird gewartet. Intern ist die Klasse so realisiert, dass bei Konstruktion der Klasse ein Thread belegt wird, und nicht beim Aufruf von `call()`, um die Thread-Ressourcen zu schonen.

2.6.3 Rauschen und anderes

Die Klasse `RNNPB_Helper` bietet einige Hilfsfunktionen für gleich- und normalverteiltes Rauschen und weitere mathematische Operatoren. Auch die Approximation der Exponentialfunktion ist dort implementiert. Die probabilistischen Funktionen können zum Beispiel für die Initialisierung der Netzparameter verwendet werden (siehe `/src/RNNPBlib/RNNPB_Definitions.h`).

2.7 Kompilierung und Performanz

Neben den bereits beschriebenen Beschleunigungsverfahren (Netzwerk-Parsing für Speicherallokation, Subnetzwerk-Parallelisierung, Präprozessor-Direktiven, Pufferung, kohärenter Vektor-Datentyp, Lookup-Tables, schnelles Exponenzieren, Sprungtabellen) empfiehlt es sich bei der Kompilierung des Frameworks, alle benötigten Bibliotheken statisch zu verlinken, alle Prozessoroptimierungen (-Ofast und prozessorspezifische Einstellungen) zu aktivieren und falls möglich einen Profiler zur Optimierung zu verwenden (Profile Guided Optimization, PGO). Im Framework ist ein entsprechendes Beispiel-Programm (`Example.cpp`) mit Makefile (`make -B RNNPBExample`) enthalten, das diese Optimierungen für Intel i7 Prozessoren durchführt. PGO kann in Verbindung mit statischem

Linken einen Geschwindigkeitsvorteil von 17 % bewirken. Je nach Hardware und Topologie können mit dem RNNPBlib-Framework auf diese Weise bis zu 40 Millionen Gewichte pro Sekunde sequentiell trainiert werden.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Markus A Baumann, Marie-Christine Fluet, and Hansjörg Scherberger. Context-specific grasp movement representation in the macaque anterior intraparietal area. *The Journal of Neuroscience*, 29(20):6436–6448, 2009.

[2] Hackjin Kim, Leah H Somerville, Tom Johnstone, Sara Polis, Andrew L Alexander, Lisa M Shin, and Paul J Whalen. Contextual modulation of amygdala responsivity to surprised faces. *Journal of Cognitive Neuroscience*, 16(10):1730–1745, 2004.

[3] Karl Zipser, Victor AF Lamme, and Peter H Schiller. Contextual modulation in primary visual cortex. *The Journal of Neuroscience*, 16(22):7376–7389, 1996.

[4] Thomas Natschläger. Netzwerke von Spiking Neuronen: Die dritte Generation von Modellen für neuronale Netzwerke. *Jenseits von Kunst. Passagen Verlag*, 1996. `www.igi.tugraz.at/tnatschl`.

[5] Hugh R Wilson and Jack D Cowan. A mathematical theory of the functional dynamics of cortical and thalamic nervous tissue. *Kybernetik*, 13(2):55–80, 1973.

[6] JC Eccles, RF Schmidt, and WD Willis. Pharmacological studies on presynaptic inhibition. *The Journal of physiology*, 168(3):500–530, 1963.

[7] Robert F Schmidt. Presynaptic inhibition in the vertebrate central nervous system. In *Ergebnisse der Physiologie Reviews of Physiology, Volume 63*, pages 20–101. Springer, 1971.

[8] DANIEL Cattaert, Abdeljabbar El Manira, and François Clarac. Direct evidence for presynaptic inhibitory mechanisms in crayfish sensory afferents. *Journal of neurophysiology*, 67(3):610–624, 1992.

[9] E Sylvester Vizi and Janos P Kiss. Neurochemistry and pharmacology of the major hippocampal transmitter systems: synaptic and nonsynaptic interactions. *Hippocampus*, 8(6):566–607, 1998.

[10] E Pessa. Symmetry breaking in neural nets. *Biological cybernetics*, 59(4-5):277–281, 1988.

[11] David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[12] Gail A Carpenter, Stephen Grossberg, and John H Reynolds. Artmap: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. *Neural networks*, 4(5):565–588, 1991.

[13] M.L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[14] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

[15] JE Moody, SJ Hanson, Anders Krogh, and John A Hertz. A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4:950–957, 1995.

[16] Scott E Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report Paper 1800, Computer Science Department, 1988.

[17] Scott Kirkpatrick, D. Gelatt Jr., and Mario P Vecchi. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[18] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[19] Georg Dorffner. Neural networks for time series processing. *Neural Network World*, 6:447–468, 1996.

[20] Michael I. Jordan. Artificial neural networks. chapter Attractor dynamics and parallelism in a connectionist sequential machine, pages 112–127. IEEE Press, Piscataway, NJ, USA, 1990.

[21] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[22] Peter beim Graben, Thomas Liebscher, and Jürgen Kurths. Neural and cognitive modeling with networks of leaky integrator units. In *Lectures in Supercomputational Neurosciences*, pages 195–223. Springer, 2008.

[23] Jun Tani. Learning to generate articulated behavior through the bottom-up and the top-down interaction processes. *Neural Networks*, 16(1):11–23, 2003.

[24] Boost Serialization. `http://www.boost.org/doc/libs/1_54_0/libs/serialization/doc/index.html`, 21. November 2013.

[25] std::ostream::operator¡¡ - C++ Reference. `http://www.cplusplus.com/reference/ostream/ostream/operator%3C%3C/`, 21. November 2013.

[26] class template std::map - C++ Reference. `http://www.cplusplus.com/reference/map/map/`, 21. November 2013.

[27] Nicol N Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless

of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

(d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install

and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part

of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express

agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing

courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.