NeuroLibrary

Stefan Huber

23. Februrar 2006

Zusammenfassung

NeuroLibrary ist eine dynamische C++ Bibliothek unter Linux, welche künstliche neuronale Netze implementiert.

Der aktuelle Stand beinhaltet Multilayer-Perzeptrone und Error-Backpropagation Lernalgorithmus mit diversen Aktivierungsfunktionen. Durch den großen Einsatz von Linearer Algebra wurde die VecMaLibrary für Matrizen- und Vektorenoperationen verwendet.

Es wird zunächst auf die zu Grunde liegenden Modelle und Mathematik eingegangen um im Weiteren die konkrete Implementierung zu besprechen.

Inhaltsverzeichnis

1 Grundlagen

1.1 Multilayer-Perzeptron

Das Multilayer-Perzeptron ist eines der bekanntesten und am häufigsten zur Anwendung kommende Neuronale Netz. Es gehört zu den klassischen Vertretern der Feedforward-Netzen und besitzt eine in Abb. 1 illustrierte Struktur.

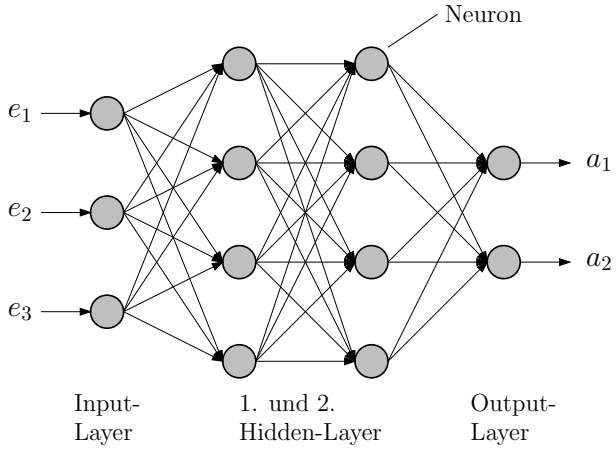

Abbildung 1: Beispiel einer Struktur eines MLP

Jedes Verarbeitungs-Element/Neuron (in Abb. 1 als Kreis dargestellt) verknüpft seine Eingänge durch eine bestimmte mathematische Funktion [1]. Die Neurone werden in Layer organisiert, wobei Neuronen in einem Layer nicht untereinander verbunden sind. Ein Multilayer-Perzeptron [2] besitzt einen Input-Layer, eventuell mehrere Hidden-Layer und einen Output-Layer. Alle Neuronen sind mit allen Neuronen benachbarter Layer verbunden. Von Layer zu Layer wird demnach ein vollständiger bipartiter Graph gebildet.

Die Verarbeitung eines Input-Vektors $e = (e_0, \ldots, e_n)$ geschieht durch die Präsentation des Vektors am Input-Layer. Jedes Neuron verarbeitet ihre Eingänge und die Werte werden bis zum Output-Layer weiterverarbeitet.

1.1.1 Das Perzeptron

Betrachten wir zunächst ein einzelnes Neuron, im Speziellen das Perzeptron. Die grundlegende Struktur wird durch Abb. 2 illustriert. Die Kombination der Eingänge wird durch eine Summe realisiert und anschließend durch eine Aktivierungsfunktion φ weiterverarbeitet.

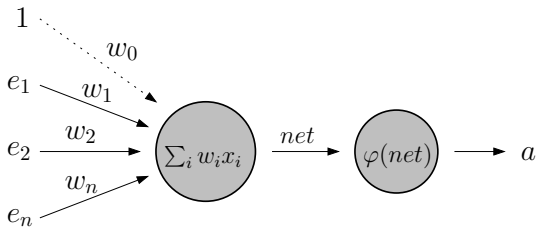

Abbildung 2: Struktur eines Perzeptrons

[1] Meist wird hierfür eine Summenfunktion und eine anschließende Aktivierungsfunktion verwendet

[2] Fortan als MLP bezeichnet

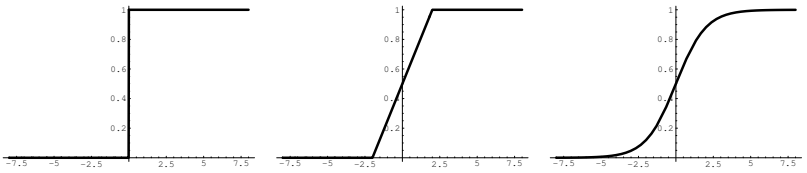

Abbildung 3: Hardlimiter, Stückweise linear, Sigmoid

Funktion	**Ausdruck**
Hardlimiter	$\varphi_{hl}(x) = \begin{cases} 0 & \textit{für } x < a \\ 1 & \textit{für } x \geq a \end{cases}$
St.weise Linear	$\varphi_{pwl}(x) = \begin{cases} 0 & \textit{für } x < a \\ \frac{x-a}{b-a} & \textit{für } x \in [a,b) \\ 1 & \textit{für } x \geq b \end{cases}$
Sigmoid	$\varphi_{sig}(x) = \frac{1}{1+e^{-x}}$

Tabelle 1: Gängige Aktivierungsfunktionen

Oft wird optional ein weiterer *Bias*-Eingang angeboten. Dieser liefert eine konstante Eins, welches durch das Gewicht w_0 gewichtet wird. Das Neuron erzeugt also die gewichtete Summe der Eingänge um eine Konstante w_0 erhöht und lässt darauf die Aktivierungsfuntkion φ wirken.

Die Aktivierungsfunktion Das Ziel der Aktivierungsfunktion ist es, eine Nichtlinearität in das System zu einzuführen. Betrachten wir rückwirkend nocheinmal Abb. 1 mit der Zusatzinformation, dass jedes Neuron eine lineare Verarbeitung vornimmt. Es wäre unzweckmäßig mehrere Schichten zu verwenden, da das gesamte System ein lineares Verhalten aufweisen würde. Man könnte alle Gewichte zwischen zwei benachbarten Layer durch eine Matrix beschreiben - somit ergäbe sich für die Übertragung des gesamten Netzes das Produkt der Matrizen. Allerdings sind mit einschichtigen Netzen lediglich linear separierbare Verhalten behandelbar (siehe Lineare Separierbarkeit Par.-1.1.1). Es ist demnach eine Nichtlineare Übertragungsfunktion von Nöten um komplexere Aufgaben zu bewältigen.

In Abb. 3 werden die drei häufigsten Übertragungsfunktion dargestellt. Während der Hardlimiter für binäre Netze geeignet ist, stellt die Sigmoid-Funktion eine auf ganz $\mathbb{R}$ differenzierbare Funktion dar.

Die angegebenen Beispiele sind für Netze geeignet, welche entweder binär mit $0, 1$-Werten arbeiten oder kontinuierlich im Intervall $[0, 1]$ Werte annehmen. Eine ebenfalls gängige Wahl stellen Netze dar, welche mit Werten $-1, 1$ arbeiten. In diesem Fall werden die Aktivierungsfunktionen dementsprechend gestreckt.

Sigmoid-Aktivierung Die Sigmoid-Aktivierung hat die Besonderheit der Differenzierbarkeit. Da diese im wesentlichen aus einer e-Funktion besteht, gestaltet

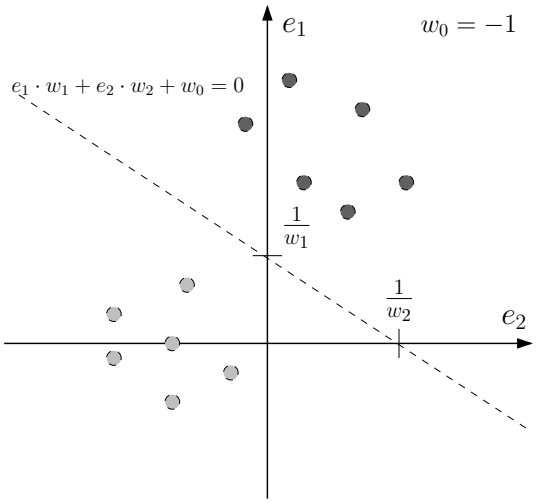

Abbildung 4: Lineare Seperierung des $\mathbb{R}^2$

sich die Ableitung besonders schön:

$$\begin{aligned}
\varphi'(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\frac{1}{1+e^{-x}} \\
&= -\left(\frac{1}{1+e^{-x}}\right)^2 \cdot (-e^{-x}) \\
&= \varphi(x)^2(1+e^{-x}-1) \\
&= \varphi(x)^2\left(\frac{1}{\varphi(x)}-1\right) \\
&= \varphi(x)\,(1-\varphi(x))
\end{aligned}$$

Lineare Separierbarkeit Ein einziges Perzeptron kann einfache Klassifizierungsaufgaben übernehmen. Betrachten wir das Beispiel in Abb. 4. Es werden auf dem $\mathbb{R}^2$ zwei Arten von Punkte $(e_1, e_2) \in \mathbb{R}^2$ platziert, die einen dunkel, die anderen hell. Es sollen die Gewichte eines Perzeptrons so bestimmt werden, dass es Auskunft erteilen kann, zu welcher der zwei Klassen ein Punkt gehört.

Das Ergebnis hat einen binären Charakter: Ein Punkt gehört in die eine oder in die andere Menge. Wir verwenden als Aktivierungsfunktion einen Hardlimiter und identifizieren mit 0 die eine Menge, mit 1 die Andere.

Das Neuron liefert für $w_0 + e_1w_1 + e_2w_2 \geq 0$ eine 1 und sonst eben eine 0. Der $\mathbb{R}^2$ wird in zwei Klassen von Punkten partitioniert und es wird eine Gerade definiert, welche die beiden Klassen trennt [3]. Die Gerade wird durch eine Gleichung beschrieben und durch die Gewichte w_0, w_1, w_2 determiniert:

$$w_0 + e_1w_1 + e_2w_2 = 0$$

Ziel ist, die Gewichte so zu bestimmen, dass die zugehörige Seperierungs-Gerade die Punkte so trennt, dass die gesamten Klassen auf gegenüberliegenden

[3] In der Mathematik entspricht dies dem *Rand* der beiden Mengen.

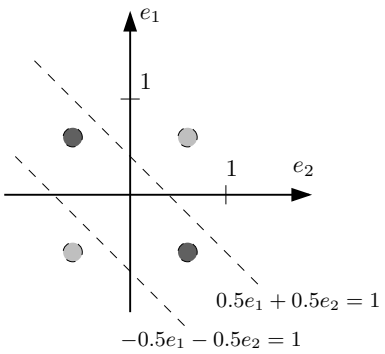

Abbildung 5: Das XOR-Problem

Halbebenen liegen. Die Gerade wird durch die drei Gewichte w_0, w_1, w_2 mehrdeutig bestimmt. Eine naheliegende Möglichkeit ist bereits angegeben. Setzen wir $w_0 := -1$ so ist die Gerade durch die beiden Gewichte w_1, w_2 in eindeutiger Weise festgelegt:

$$w_1 e_1 + w_2 e_2 = 1$$

Verwendet wir die Gerade in Abb. 4. Auf diese Weise führen alle Punktepaare (e_1, e_2) unterhalb der Geraden im Neuron zum Wert 0, alle Paare oberhalb der Geraden zu 1.

Es ist einfach einsehbar, dass die Verallgemeinerung zu höheren Dimension die Anzahl der Eingänge entsprechend erhöht. Betrachten wir Punktemengen im $\mathbb{R}^n$. Die Separierungsgerade wird zu einer Hyperebene, welche durch ihren Normalvektor $w := (w_1, w_2, \ldots, w_n)$ definiert ist:

$$w \cdot e = -w_0$$

Wobei hier $e := (e_1, e_2, \ldots e_n)$ den Eingangsvektor darstellt und $\cdot$ das Standard-Skalarprodukt von Vektoren. Hieraus ergibt sich, dass ein Perzeptron lediglich linear separierbare Mengen klassifizieren kann. Erhöht sich die Komplexität der Mengen derart, dass keine trennende Hyperebene gefunden werden kann, so ist es dem Perzeptron nicht möglich eine Klassifizierung vorzunehmen. Das einfachste Beispiel hierfür: Das XOR-Problem.

1.1.2 XOR-Problem

Das XOR-Probem stellt das einfachste nicht-lösbare Problem für ein Perzeptron dar, da es nicht linear separierbar ist. Es können keine Gewichte w_0, w_1, w_2 so bestimmt werden, dass die zugehörige Gerade $w_1 e_1 + w_2 e_2 = -w_0$ eine Trennung der Punktmengen vornimmt.

Um dieses Problem zu bewältigen erzeugen wir zwei Neurone, welche die Trennungsgeraden in Abb. 5 implementieren. Feuert [4] keines der beiden Neurone oder Beide, so entspricht dies einem hellen Punkt. Feuert lediglich das Neuron der unteren Gerade, das der Oberen nicht, so entspricht dies den dunklen Punkten. Diese zweite Kodierung wird durch ein nachgeschaltenes Neuron realisiert, welches in Abb. 6 illustriert wird.

[4] Aus der Biologie stammend: Das Neuron wird aktiv, es liefert 1

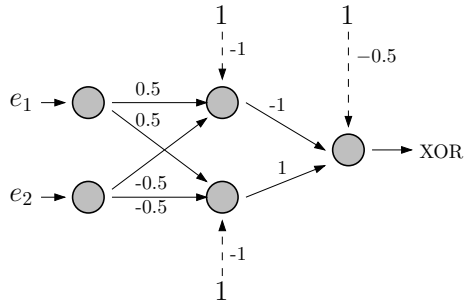

Abbildung 6: Lösung zum XOR-Problem

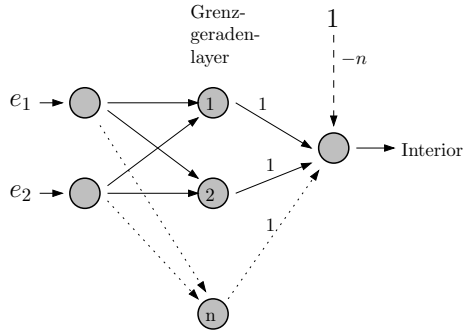

Abbildung 7: Lösung zum Polygon-Problem

Die Lösung des XOR-Problem wird also durch die Einführung einer weiteren Schicht erreicht. Das XOR-Problem stellt also eine wesentliche Motivation für die Verwendung von Multilayer-Perzeptronen dar, da dadurch wesentlich komplexere Klassifizierungen vorgenommen werden können.

1.1.3 Polygon Klassifizierung

Im 2-dimensionalen Fall können durch dieses Schema im wesentlichen konvexe Polygone als Klassifizierungsmenge abgedeckt werden. Eine spezielle Möglichkeit, um ein konvexes Polygon zu klassifzizieren besteht darin, dass jede seitlich begrenzende Gerade des Polygons durch ein Perzeptron dargestellt wird. Die Ausrichtung muss allerdings so gewählt werden, dass eine 1 auf der Seite des Polygons geliefert wird. Feuert jedes Perzeptron, so lässt man auch das nachgeschaltene Feuern. Feuert zumindest eines nicht, so feuert auch das nachgeschaltene nicht. In Abb. 7 wird die Lösung illustriert, wobei hier die Bias-Eingänge des Hidden-Layer nicht eingetragen sind.

Polytope im $\mathbb{R}^n$ Die Verallgmeinerung ins Mehrdimensionale klassifiziert dann Polytope im $\mathbb{R}^n$ nach dem vorgegebenen Schema. Die Randgeraden verallgemeinern sich dann eben zu Hyperebenen, welche das Polytop eingrenzen.

Das Multilayer-Perzeptron verallgemeinert dieses Netz zu einer beliebigen Anzahl von Layern zu beliebigen Dimensionen. Bereits mit drei Layern können

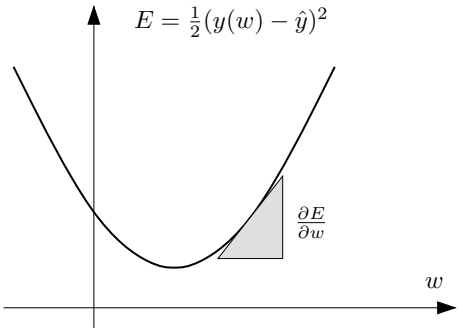

Abbildung 8: Graph des Fehlerquadrat

beliebig komplexe Mengen klassifiziert werden.

1.2 Error-Backpropagation

Nach den Überlegungen von Sec.-1.1.3 haben wir die Leistungsfähigkeit von MLP abgeschätzt. Nun überlegen wir uns, wie wir die Gewichte zu bekannten Trainingspaaren von Eingangs-/Ausgangsvektoren anpassen. Diesen Vorgang kommt einem Lernen gleich.

1.2.1 Quadratischer Fehler

Zur Findung der gesuchten Gewichte verwenden wir ein bekanntes Konzept - die *Fehlerquadrat-Optimierung*. Wir drücken den Fehler als Funktion der Gewichte aus und Optimieren diese zur Minimierung des Fehlers.

Sei also zunächst $y(w)$ eine zu optimierende Größe und $\hat{y}$ der anzustrebende Größe. Die Größe $y(w)$ hängt von einem Parameter w ab, welcher optimiert werden soll.

Das Ziel ist es, denn Fehler so weit wie möglich zu minimieren. Der quadratische Fehler wird wie folgt ausgedrückt:

$$E = \frac{1}{2}(y(w) - \hat{y})^2$$

Um ein (lokales) Minimum von E zu finden, wird iterativ vom Fehler die Steigung $\frac{\partial E}{\partial w}$ abgezogen. Hat man das Minimum erreicht, ergibt sich dieser zu 0 - auf dem Weg dorthin, wird diese immmer kleiner. Man kann eine gewisse Ähnlichkeit zum Newton-Verfahren zum Finden von Nullstellen feststellen.

Man kann dadurch eine Folge von Gewichten (w_k) definieren, wobei rekursiv definiert wird:

$$w_{k+1} = w_k - \frac{\partial E_k}{\partial w}$$

1.2.2 Ausgangsschicht

In Verbindung mit dem Backpropagation Lernalgorithmus wird als Basis ein MLP mit der sigmoiden Aktivierung verwendet. Um die Gewichte der Ausgangsschicht zu optimieren, formulieren wir einen Fehler.

Sei der $a = (a_1, \ldots, a_m)$ der Ausgangsvektor. Der Vektor $x = (x_1, \ldots, x_n)$ sei der Ausgangsvektor des letzten Hidden-Layer. Die Gewichte w_{ji} seien die Gewichte zwischen dem i-ten Neuron des letzten Hidden-Layer und des j-ten Neuron der Ausgangsschicht.

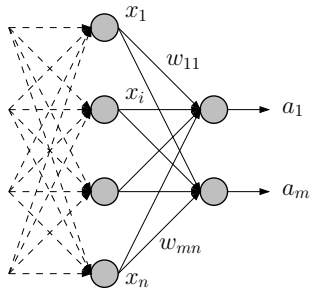

Abbildung 9: Optimierung Ausgangsschicht

Der Vektor $\hat{a} = (\hat{a}_1, \ldots, \hat{a}_m)$ sei der anzustrebende Ausgangsvektor. Der Fehler ergibt sich zu:

$$E = \sum_{j=1}^{m} \frac{1}{2}(a_j - \hat{a}_j)^2 = \sum_{j=1}^{m} \frac{1}{2} \left(\varphi(\sum_{i=1}^{n} w_{ji} x_i) - \hat{a}_j \right)^2$$

Die erforderliche Änderung des Fehlers berechnet man wie folgt:

$$\begin{aligned}
\frac{\partial E}{\partial w_{lk}} &= \frac{\partial}{\partial w_{lk}} \sum_{j=1}^{m} \frac{1}{2} \left(\varphi(\sum_{i=1}^{n} w_{ji} x_i) - \hat{a}_j \right)^2 \\
&= \frac{\partial}{\partial w_{lk}} \frac{1}{2} \left(\varphi(\sum_{i=1}^{n} w_{li} x_i) - \hat{a}_l \right)^2 \\
&= \left(\varphi(\sum_{i=1}^{n} w_{li} x_i) - \hat{a}_l \right) \frac{\partial}{\partial w_{lk}} \varphi(\sum_{i=1}^{n} w_{li} x_i) \\
&= (a_l - \hat{a}_l) \cdot a_l(1 - a_l) \cdot x_k
\end{aligned}$$

Wobei wir aus Sec.-1.1.1 die Identität $\varphi' = \varphi(1 - \varphi)$ verwenden. Wir definieren ein Fehlersignal δ^1 und fassen wir das Ergebnis zusammen:

$$\delta_j^1 := (a_j - \hat{a}_j) \cdot a_j(1 - a_j) \tag{1}$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j^1 x_i \tag{2}$$

Auf die gleiche Weise sieht man schnell:

$$\frac{\partial E}{\partial x_i} = \sum_{j=1}^{m} \delta_j^1 w_{ji} \tag{3}$$

Das Fehlersignal δ_j^1 beschreibt die Error-Backpropagation, also die Rückkopplung des Fehlers.

1.2.3 Innere Schicht

Für die Betrachtung der inneren Schicht, muss der Ausgang als Funktion der Ausgänge der inneren Schichten betrachtet werden. Nehmen wir als Beispiel den letzten Hidden-Layer.

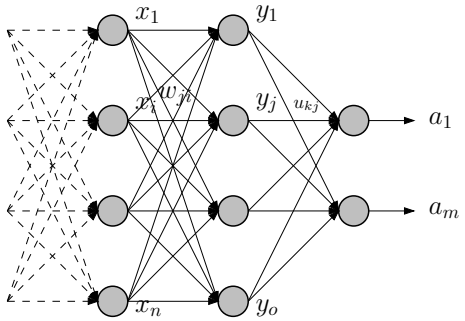

Abbildung 10: Optimierung Hidden Layer

Die zu optimierenden Gewichte lauten wieder w_{ji}, welche die inneren Werte x_i gewichten und zu den Werten y_j weiterverarbeitet werden. Diese werden über die Gewichte u_{kj} zu den Ausgängen des Netzes. Wir bestimmen den Fehler:

$$\begin{aligned} E &= \sum_{k=1}^{m} \frac{1}{2}(a_k - \hat{a}_k)^2 \\ &= \sum_{k=1}^{m} \frac{1}{2} \left(\varphi(\sum_{j=1}^{o} u_{kj} \underbrace{\varphi(\sum_{i=1}^{n} w_{ji} x_i)}_{y_j}) - \hat{a}_k \right)^2 \end{aligned}$$

Im Sinne des quadratischen Fehlers bilden wir die Ableitung:

$$\begin{aligned} \frac{\partial E}{\partial w_{ab}} &= \sum_{j=1}^{o} \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{ab}} \\ &= \sum_{j=1}^{o} \sum_{k=1}^{m} \delta_k^1 u_{kj} \frac{\partial y_j}{\partial w_{ab}} \\ &= \sum_{k=1}^{m} \delta_k^1 u_{ka} y_a (1 - y_a) x_b \\ &= x_b \cdot y_a (1 - y_a) \sum_{k=1}^{m} \delta_k^1 u_{ka} \end{aligned}$$

Hierbei verwenden wir in der ersten Zeile das Ergebnis aus Gl. 3. Definiere wir auch hier wieder ein Fehlersignal δ_j^2:

$$\delta_j^2 := y_j (1 - y_j) \sum_{k=1}^{m} \delta_k^1 u_{kj} \tag{4}$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j^2 x_i \tag{5}$$

Definieren wir wie oben $\delta_j^3, \delta_j^4, \ldots$ entsprechend der Gleichung Gl. 4 so können wir die Berechnung des Fehlers analog vornehmen. Somit gestalten sich sämtliche δ_j in den verdeckten Schichten gleich und wir erhalten eine Rekursionsform wie in Gl. 4 für alle weiteren inneren Schichten.

1.2.4 Algorithmus

Wir betrachten ein Multilayer-Perzeptron mit S Schichten, wobei jede Schicht $s = 1, \ldots S$ die Dimension von $d(s)$ Neuronen besitze. Hierbei sei die Schicht 1 die Ausgangsschicht und die Schicht S der Eingangslayer. Weiters sei w_{ji}^s das Gewichte vom i-ten Neuron der $(s+1)$-ten Schicht zum j-ten Neuron der s-ten Schicht.

Es seien weiters a_j^s die Ausgangswerte des j-ten Neurons in der s-ten Schicht. Für $s = S$ erhält man den Inputvektor.

Wir berechnen rekursiv die Fehlersignale δ_j^s des j-ten Neurons in der s-ten Schicht, mit $s \in [1, S-1]$:

$$\delta_j^1 := (a_j^1 - \hat{a}_j^1) \cdot a_j^1 (1 - a_j^1) \tag{6}$$

$$\delta_j^s := a_j^s \cdot (1 - a_j^s) \sum_{k=1}^{d(s-1)} \delta_k^{s-1} w_{kj}^{s-1} \tag{7}$$

Für jedes Gewicht w_{ji}^s gilt nun:

$$\frac{\partial E}{\partial w_{ji}^s} = \delta_j^s \cdot a_i^{s+1} \tag{8}$$

Jedes Gewicht w_{ji}^s wird nach der folgenden Gleichung verändert:

$$w_{ji,neu}^s = w_{ji,alt}^s + \eta \cdot \delta_j^s a_i^{s+1} \tag{9}$$

Die Zahl η stellt die Lernrate dar: Also wie schnell das Netz lernen soll, seine Gewichte verändert. Eine zu niedrige Lernrate führt zu langsamen Lernverhalten. Eine zu hohe Lernrate führt dazu, dass über das Minimum des Lernfehlers hinausgeschosssen wird und ein Einpendeln stattfindet.

2 Implementierung

Da aus Sec.-1 ersichtlich ist, dass lineare Algebra zur Anwendung kommt, wird in der Implementierung die VecMaLibrary verwendet. Da sich die NeuroLibrary in einem noch sehr jungen Stadium befindet, ist die Klassenhierarchie (siehe Fig. 11) noch sehr bescheiden.

Sie teilt sich in drei Bereiche:

- Aktivierungsfunktionen
- Neuronale Netze
- Lernalgorithmen

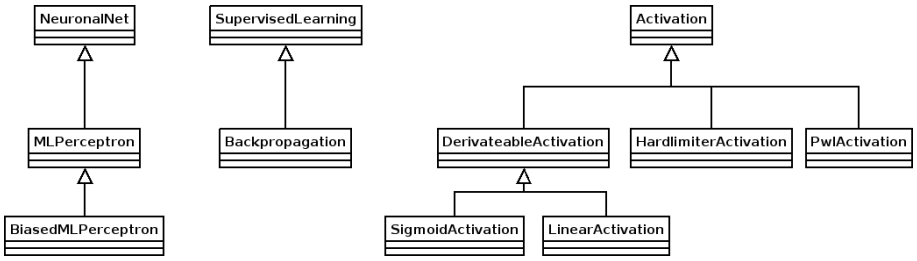

Abbildung 11: Klassenhierarchie

2.1 Aktivierungsfunktionen

Es werden die gängingen Aktivierungsfunktionen angeboten – manche davon sind differenzierbar. Jede Aktivierungsfunktion bietet folgende Funktionalitäten

- **virtual void** evaluate(**double** input) = 0
- **virtual** Activation* clone() = 0
- Vector evaluate(Vector input)

HardlimiterActivation In binären Netzen wird der Hardlimiter eingesetzt. Ist der Input unterhalb einer Schwelle wird der Output meist auf 0 gesetzt. Erreicht der Input diese Schwelle wird 1 zurückgeliefert.

Listing 1: Konstruktor HardlimiterActivation

```
1 HardlimiterActivation(double limit=0.0,
2         double left=0.0, double right=1.0);
```

PwlActivation Der Name PwlActivation steht für *Part Wise Linear*, also stückweise linear. Zwischen zwei Grenzen verhält sich diese Funktion linear, außerhalb konstant.

Listing 2: Konstruktor PwlActivation

```
1 PwlActivation(double xright=1.0, double yright=1.0,
2         double xleft=0.0, double yleft=0.0);
```

2.1.1 DerivateableActivation

Manchmal kann es erforderlich sein, dass die Aktivierungsfunktion differenzierbar sein muss – Backpropagation ist ein Beispiel dafür. Die entsprechende Klasse bietet zusätzlich noch folgende Funktionen an:

- **virtual double** revDerivation(**double** output) = 0
- Vector revDerivation(Vector output)

Die Ableitung wird als Funktion der Funktion selbst betrachtet. Betrachte man die Funktion $f(x) = x^2$, die Ableitung ergibt sich zu $f'(x) = 2x$. Als Funktion der Funktion selbst ergibt sich $f'(x) = 2\sqrt{f(x)}$.

SigmoidActivation Ein Beispiel hierfür ist die Sigmoid-Funktion. Sie ist noch parametrisierbar bezüglich des Grenzwerts $x \to -\infty$ und $x \to \infty$, sowie der Steigung im Nullpunkt und der Verschiebung dieses.

Listing 3: Konstruktor SigmoidActivation

```
1 SigmoidActivation(double min=0, double max=1,
2         double xoff=0, double grad=1);
```

LinearActivation Die lineare Aktivierung kommt selten zum Einsatz und lediglich in der Steigung parametrisierbar.

Listing 4: Konstruktor LinearActivation

```
1 LinearActivation(double grad=1.0);
```

2.2 Netze

Die *NeuroLibrary* stellt ein generisches Netz zur Verfügung, nachdem alle Paradigmen arbeiten. Es wird ein Eingangsvektor dem Netz präsentiert und daraus ein Ausgangsvektor berechnet. Die Klasse *NeuronalNet* stellt diese Schnittstelle zur Verfügung:

```
1 virtual Vector evaluate(Vector input) = 0
```

2.2.1 Multilayer-Perzeptron

Das Multilayer-Perceptron kommt in zwei Ausführungen:

- unbiased
- biased

Im zweiteren Fall erhält jeder Layer einen zusätzlichen Eingang, der konstant 1 liefert. Die Gewichte zwischen den Layern werden in Matrizen gespeichert. Auf diese Weise ist die Evaluierung eines Input-Vektors besonders einfach. In jeder Schicht werden folgende Schritte vorgenommen:

1. Multipliziere Gewichtungsmatrix mit Inputvektor
2. Wende Aktivierungsfunktion auf Ergebnisvektor an.

Sei also die Matrix $W = (w_{ji})_{1,m}^{1,n}$ die Gewichtungsmatrix der Gewichte vom i-ten Neuron einer Schicht zum j-ten Neuron der nächsten Schicht. Die erste Schicht besitze n Neuronen, die nächste m Neuronen. Weiters treffe ein Inputvektor $e = (e_1, \ldots, e_n)$ auf diese Schicht. Durch die Summenbildung der Neuronen wird für jede Komponente des Ergebnisvektors $a = (a_1, \ldots, a_m)$ folgendes berechnet:

$$a_j = \sum_{i=1}^{n} w_{ji} e_i$$

Da dies für jede Ergebniskomponente geschieht erhalten wir:

$$a = M \cdot i$$

Biased MLP Im Fall eines biased Multilayer Perzeptrons werden alle Funktionen, mit Evaluierung und Layer-Größe zu tun haben überschrieben:

- **virtual** Vector evaluate(Vector input)
- **virtual** Vector* createSubResults(Vector input)
- **virtual int** getLayerSize(**int** layer)

2.3 Lernalgorithmen

Die *NeuroLibrary* unterstützt bis jetzt lediglich den Backprogagation-Algorithmus er fällt in die Klasse des *Supervised Learning*. Das Supervised Learning stellt Paare von Eingangs- und Ausgangsvektoren zur Verfügung, die gelernt werden sollen. Das Ergebnis (Ausgangsvektor) ist demanch schon bekannt und man kann direkt die Entfernung zum Ergebnis (Soll-Ist) berechnen.

Die Klasse *SupervisedLearning* stellt eine Schnittstelle für diese Art von Lernen bereit:

```
1  virtual void train(Vector input, Vector destination) = 0;
2  virtual Vector evaluate(Vector input) = 0;
3
4  virtual void setLearnrate(double learnrate) = 0;
5  virtual double getLearnrate() = 0;
```

2.3.1 Error-Backpropagation

Der *Backpropagation-Algorithmus* stellt ein Beispiel für das Supervised Learning dar. Er arbeitet nach dem Prinzip der quadratischen Fehler-Optimierung auf einem Multilayer-Perceptron.

```
1  Backpropagation(int layerSizes[], int cntLayers,
2          DerivateableActivation* activation, double learnrate,
3          double dynamic=0.0, double init=2, bool biased=true);
```

Dem Backpropagation-Algorithmus werden zunächst die Geometrien des zugrundeliegenden MLPs übergeben, sowie dessen Aktivierung. Diese muss Algorithmusbedingt eine differenzierbare Funktion sein. Danach folgen Lernkonvergenz beeinflussende Parameter learnate und dynamic. Der Parameter init noch an, in welchem Intervall die Gewichte zufällig initialisiert werden. Mit biased kann man regeln, ob ein biased MLP oder ein unbiased verwendet wird.

Es werden noch zur einfachen Anwendung zwei statische Methoden angeboten, mit denen man eine Backpropagation-Instanz erzeugen kann.

```
1  static Backpropagation createSigmoidBipolar(int* layerSizes,
2          int cntLayers, double learnrate, double dynamic=0.0)
3  static Backpropagation createSigmoidUnipolar(int* layerSizes,
4          int cntLayers, double learnrate, double dynamic=0.0)
```