Deep Learning

22. Juni 2015

Inhaltsverzeichnis

1 Grundlagen

Als Grundlage für Deep-Belief Netze dienen eingeschränkte Boltzmann Maschinen. Diese sind eng mit Hopfield Netzen verwandt und der Weg von Hopfield Netzen zu eingeschränkten Boltzmann Maschinen wird in diesem Kapitel erläutert. Hopfield Netze und allgemeine Boltzmann Maschinen werden hierbei jedoch nur kurz erläutert und am Ende des Kapitels werden eingeschränkte Boltzmann Maschinen ausführlich erklärt.

1.1 Hopfield Netze

In diesen Netzen ist jedes Neuron mit allen anderen Neuronen außer sich selber verbunden, wie in Abbildung 1 dargestellt ist. Die Neuronen in einem Hopfield Netz sind binär, dass bedeutet die Ausgabe 1 ist wenn ein Schwellwert θ überschritten wird und ansonsten 0.

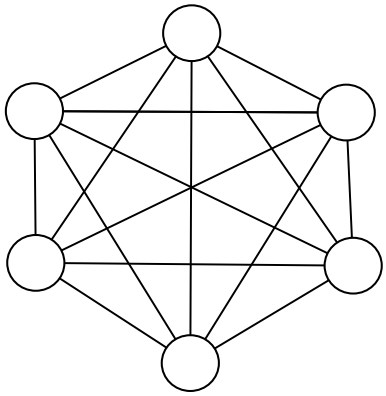

Abbildung 1: Modell eines Hopfield Netzes. Alle Neuronen besitzen eine Ein- und Ausgabe und sind über Gewichte jeweils mit allen anderen Neuronen verbunden

Die Gewichte in einem Hopfield Netz sind symmetrisch, also gilt $w_{ij} = w_{ji}$. Hopfield Netzen wird eine Energie zugewiesen, die beim lernen des Netzes minimiert werden soll [6]:

$$E = -\frac{1}{2}\sum_{i \neq j} w_{ij} s_i s_j + \sum_i \theta_i \; s_i \tag{1}$$

s_i bzw. s_j sind hierbei die Zustände der Neuronen. Sobald die Energie unverändert bleibt, wurde ein stabiler Zustand erreicht und man befindet sich in einem lokalen Minimum der Energielandschaft.

1.2 Boltzmann Maschinen

Boltzmann Maschinen sind ähnlich wie Hopfield Netze. Ein wesentlicher Unterschied besteht darin, dass die Neuronen stochastisch binär sind, also nicht mehr feuern wenn ein Schwellwert überschritten ist, sondern mit einer bestimmten Wahrscheinlichkeit eine 1 als Ausgabe haben. Dies wird in der Praxis so realisiert, dass ein Wurf zwischen 0 und 1 größer

als die Wahrscheinlichkeit sein muss um das Neuron zu aktivieren. Boltzmann Maschinen wird auch eine Energie zugewiesen, die ähnlich der von Hopfiel Netzen ist:

$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right) \tag{2}$$

Hierbei ist θ_i jedoch nicht der Schwellwert, sondern der Bias des Neurons i.
Desweiteren werden bei Boltzmann Maschinen die Neuronen in sichtbare und in versteckte Neuronen unterteilt. Die sichtbaren Neuronen haben die Möglichkeit eine Eingabe von außen zu bekommen, die versteckten Neuronen nicht. Der Aufbau ist in Abb. 2 dargestellt.

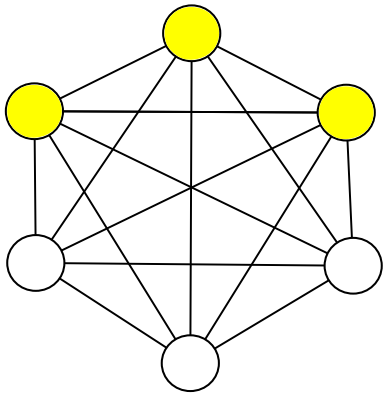

Abbildung 2: Modell einer Boltzmann Maschine. Gelbe Kreise stellen verstecke Neuronen dar, weiße Kreise sind sichtbare Neuronen.

Das Training von Boltzmann Maschinen läuft in zwei Phasen ab: Die erste Phase wird auch als positive Phase bezeichnet und besteht darin, dass man die binären Zustände der sichtbaren Neuronen vorgibt und die Maschine laufen lässt bis sich ein Equilibriumszustand einstellt, sich also die Aktivierungswahrscheinlichkeiten der einzelnen Neuronen nicht mehr ändern.
Danach beginnt man mit der Negativen Phase. In dieser Phase lässt man die Maschine ohne eine Eingabe von außen laufen bis diese wieder in einem Equilibriumszustand ist. Daraus ergibt sich eine einfache Lernregel:

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{R}[p_{ij}^{+} - p_{ij}^{-}] \tag{3}$$

Hierbei ist R die Lernrate, und p_{ij} die Wahrscheinlichkeit das Neuron i und j in der positiven, bzw. negativen Phase jeweils aktiviert sind. Theorethisch wären Boltzmann Maschinen sehr gut in der Lage Modelle von Daten zu trainieren und zu erkennen, allerdings sind diese in der Praxis recht untauglich, da sie bei großen Maschinen zu lange brauchen um trainiert zu werden. Dieses Problem wird mit eingeschränkten Boltzmann Maschinen gelöst.

1.3 Eingeschränkte Boltzmann Maschinen

Bei eingeschränkten Boltzmann Maschinen werden die sichtbaren und versteckten Neuronen als Ebenen betrachtet. Die Ebenen sind symmetrisch miteinander verbunden, jedoch existieren zwischen den Neuronen auf einer Ebene keine Verbindungen wie in Abbildung 3 dargestellt.

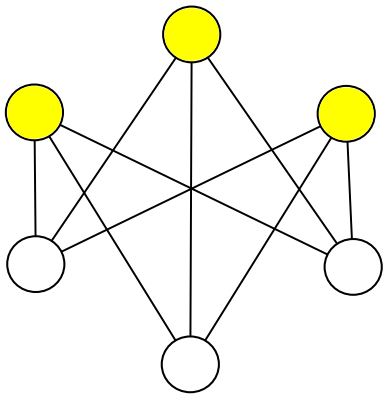

Abbildung 3: Modell einer eingeschränkten Boltzmann Maschine. Oben sind die versteckten Knoten und unten die nach außen sichtbaren Knoten.

Auch eingeschränkte Boltzmann Maschinen benutzen stochastisch binäre Neuronen wie normale Boltzmann Maschinen.
Als Eingabewerte in so ein Netzwerk sind reelle Werte möglich. Diese würden dann Aktivierungswahrscheinlichkeiten der sichtbaren Neuronen interpretiert und man müsste dann noch die binären Zustände der sichtbaren Neuronen bestimmen. Im Folgenden wird davon ausgegangen, dass binäre Eingabevektoren vorliegen. Eine Konfiguration $(\vec{v}, \vec{h})$ aus sichtbaren und unsichtbaren Neuronen wird auch bei eingeschränkten Boltzmann Maschinen eine Energie zugewiesen:

$$E(\vec{v}, \vec{h}) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{4}$$

Hierbei sind v_i und h_j die binären Zustände des sichtbaren Knotens i und des versteckten Knotens j. a_i und b_j sind die Bias der entsprechenden Knoten und w_{ij} ist das Gewicht zwischen den beiden Knoten. Mit Hilfe dieser Energiefunktion weist das Netz jedem möglichen Paar von sichtbaren und versteckten Knoten eine Wahrscheinlichkeit zu:

$$p(\vec{v}, \vec{h}) = \frac{1}{Z} e^{-E(\vec{v}, \vec{h})} \tag{5}$$

Die Zustandssumme (partition function) Z ergibt sich aus der Summe über alle möglichen Werte aus sichtbaren und versteckten Knoten und stellt einen Normierungsfaktor dar, damit die Gesamtwahrscheinlichkeit 1 ergibt:

$$Z = \sum_{\vec{v},\vec{h}} e^{-E(\vec{v},\vec{h})} \tag{6}$$

Die Wahrscheinlichkeit, dass sich das Netzwerk einem Eingabebild anpasst, ergibt sich aus der Summe über alle versteckten Vektoren [1].

$$p(\vec{v}) = \frac{1}{Z} \sum_{\vec{h}} e^{-E(\vec{v},\vec{h})} \tag{7}$$

Die Wahrscheinlichkeit, dass sich das Netz einem Trainingsbild anpasst, kann erhöht werden, indem man die Energie der anderen Bilder erhöht. Vor allem für Bilder mit einer niedrigen Energie ist dies wichtig, da diese einen hohen Beitrag zur Zustanddsumme haben. Die Ableitung der Gewichte der logarithmischen Wahrscheinlichkeiten von Trainingsdaten ergibt sich wie folgt:

$$\frac{\partial \log p(\vec{v})}{\partial w_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{8}$$

Hierbei ist $\langle ... \rangle$ der Erwartungswert der realen Verteilung, als *data* bezeichnet, und der vom Netz modellierten Verteilung *model*. Dies entspricht der positiven und der negativen Phase von normalen Boltzmann Maschinen. Der Erwartungswert ist das Mittel der Wahrscheinlichkeitsverteilung wenn und kann durch das ziehen von ausreichend vielen Knotenpaaren approximiert werden. Dies führt zu einer einfachen Lernregel für den stochastisch stärksten Aufstieg in der logarithmischen Wahrscheinlichkeit der Trainingsdaten:

$$\Delta w_{ij} = \epsilon \left(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \right) \tag{9}$$

ϵ ist die Lernrate und kann zur Steuerung wie stark die Gewichte verändert werden frei gewählt werden. In der Praxis hat eine zu hohe Lernrate zur Folge, dass die Gewichte zu groß werden und nichts gelernt wird. Eine zu kleine Lernrate führt dazu dass das Training sehr langsam ist, jedoch empfiehlt es sich für ein besseres Ergebnis gegen Ende des Trainings die Lernrate zu verkleinern [**?**]. Da die versteckten Knoten innerhalb einer RBM nicht miteinander verbunden sind, sind diese statistisch voneinander unabhängig. Dies führt dazu, dass man sehr einfach eine Stichprobe ohne Bias für $\langle v_i h_j \rangle_{data}$ erhalten kann. Wenn ein zufällig ausgewähltes Trainingsbild $\vec{v}$ gegeben ist, wird der binäre zustand h_j jedes versteckten Knotens j mit folgender Wahrscheinlichkeit auf 1 gesetzt:

$$p(h_j = 1|\vec{v}) = \sigma(b_j + \sum_i v_i w_{ij}) \tag{10}$$

$\sigma(x)$ ist die logistische sigmoide Funktion $1/(1 + e^{-x})$. $v_i h_j$ ist dann eine Stichprobe der Verteilung ohne Bias a_i, da die einzelnen versteckten Knoten nicht miteinander verbunden und nur abhängig von den Eingabedaten sind.
Da auch die sichtbaren Knoten keine Verbindungen untereinander haben und damit voneinander unabhängig sind, ist es genau so einfach eine Stichprobe für den Zustand eines sichtbaren Knotens zu erhalten, wenn ein versteckter Vektor $\vec{h}$ gegeben ist:

$$p(v_i = 1|\vec{h}) = \sigma(a_i + \sum_j h_j w_{ij}) \tag{11}$$

Eine Stichprobe ohne Bias b_j für $< v_i h_j >_{model}$ zu erhalten, ist aufwändiger und kann über abwechselndes " Gibbs Sampling" über einen langen Zeitraum berechnet werden. Beim Gibbs sampling beginnt man mit einem zufälligen Trainingsvektor und hört auf, wenn ein Equilibriumszustand erreicht wird. Eine Iteration des Gibbs sampling besteht daraus, dass man zuerst parallel die Zustände aller versteckten Knoten mit Gleichung 10 berechnet und im Anschluss mit Gleichung 11 die sichtbaren Knoten neu berechnet.

1.4 Kontrastive Divergenz

Ein schnelleres Verfahren als das Gibbs sampling besteht daraus, die Zustände der sichtbaren Knoten mit einem Trainingsvektor zu belegen. Anschließend werden die versteckten Knoten mit Gleichung 10 berechnet. Nachdem die binären Zustände der versteckten Knoten gefunden sind, wird eine "Rekonstruktion" angefertigt indem man jeden sichtbaren Knoten mit einer Wahrscheinlichkeit aus Gleichung 11 auf 1 setzt. Die Änderung in einem einzelnen Gewicht wird dadurch zu:

$$\Delta w_{ij} = \epsilon \left(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon} \right) \tag{12}$$

Diese Lernregel wird Kontrastive Divergenz (Contrastive Divergence(CD)) genannt und liefert bessere Ergebnisse als die vorherige Regel. Diese Lernregel approximiert grop den Gradienten der logarithmischen Wahrscheinlichkeit, sondern approximiert eher die Differnz zweier Kullback-Liebler Divergenzen. Was so eine Divergenz ist, wird in [7] erklärt und würde hier zu weit führen. Da hierbei ein wichtiger Term jedoch vernachlässigt wird, folgt es auch diesem Gradienten nicht genau und Untersuchungen ergaben, dass die Lernregel gar keinem Gradienten einer Funktion folgt und trotzdem gut funktioniert [3].
Die Idee hinter Kontrastiven Divergenz besteht darin, dass beim normalen Gibbs Sampling die Varianz innerhalb der entstehenden Markov-Kette immer weiter zunimmt und letztendlich den Schätzer der Ableitung verfälscht. Gleichzeitig wird die Varianz in den Proben immer abhängiger von den Parametern. Um dem entgegen zu wirken, wird für die Lernregel nicht eine Probe aus dem Gleichgewichtszustand der Boltzmann Maschine verwendet, der entsteht wenn man das Gibbs Sampling lange genug betreibt, sondern eine Probe aus der rekonstruierten Wahrscheinlichkeitsverteilung nach einem Schritt des Gibbs Sampling. Dies sorgt dafür dass die Markov-Kette näher an der tatsächlichen Wahrscheinlichkeitsverteilung der Eingabe bleibt und die Varianz geringer wird. Da die rekonstruierte Eingabe näher am Gleichgewichtzustand der Boltzmann Maschine ist als die Eingabe im Gibss Schritt davor, ist garantiert, dass die kontrastive Divergenz niemals negativ wird und nur Null erreicht, wenn das Modell perfekt ist. [2]
Ein weiterer Schritt, um das Lernen eines Modells auf RBMs zu verbessern, besteht darin, n Schritte des Gibbs samplings zu machen, bevor man die Statistik für $\langle v_i h_j \rangle_{recon}$ ermittelt. Dies wird mit CD_n notiert, wobei n die Anzahl der Schritte des Gibbs samplings angibt.

2 Deep-Belief Netze

Deep-Belief Netze sind eine Form von künstlichen neuronalen Netzen. Diese werden meist unüberwacht trainiert, da angenommen wird, dass Labels zu wenig Informationen enthalten über die vorhandenen Daten oder bei großen Datenmengen sehr oft keine Labels für alle Daten vorhanden sind. Zum einen ermöglichen Deep-Beielf Netze das rekonstruieren von einmal trainierten Modellen, z.B. kann ein Deep-Belief Netz wenn es das Modell von Bildern lernt teile von Bildern oder auch ganze Bilder rekonstruieren. Desweiteren können Deep-Belief Netze zur Klassifikation benutzt werden oder zum vortrainieren von Feedforward Netzen, da diese nicht so leicht in lokalen Minima stecken bleiben und somit das Netz näher am globalen Minimum konvergiert. Zum trainieren von Deep-Belief Netzen hat sich ein einfacher Algorithmus basierend auf eingeschränkten Boltzmann Maschinen etabliert.

2.1 Greedy Algorithmus zum trainieren von Deep-Belief Netzen

Ein effizienter Weg ein kompliziertes Modell zu lernen, besteht darin, eine Menge von einfacheren Modellen nacheinander zu lernen. Die Idee des hier vorgestellten greedy Algorithmus besteht darin, dass jedes Modell eine andere Repräsentation der Daten darstellt. Dazu berechnet jedes Modell eine nichtlineare Transformation auf seine Eingabe und die Ausgabe eines Modells wird als Eingabe des nächsten Modells verwendet.

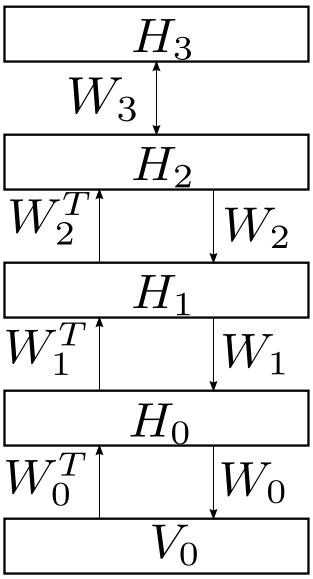

Abbildung 4: Hyprid Netzwerk. Die Ebenen H_3 und H_2 sind mit ungerichteten Kanten verbunden und bilden einen Assoziativspeicher. Die anderen Ebenen sind mit gerichteten Kanten verbunden.

In Abbildung 4 ist ein generatives Modell mit mehreren Ebenen abgebildet. Als generatives

Modell bezeichnet man Modelle, die mit Hilfe von versteckten Parametern zufällige und beobachtbare Daten generieren, wie zum Beispiel bei einer Markov-Kette. Die oberen beiden Ebenenen kommunizieren über ungerichtete Kanten und simulieren damit unendlich viele weitere Ebenen mit verbundenen Gewichten. Es gibt keine Verbindungen zwischen den Knoten einer Ebene, sodass diese unabhängig voneinander sind. Man kann gute Parameter für W_0 finden, indem man annimmt, dass die Gewichte zwischen den höheren Ebenen eine komplementäre Verteilung (complementary prior) abbilden, um den "Explained Away"-Effekt für W_0 auszulöschen [4].
Der "Explained Away"-Effekt beschreibt hierbei dass zwei Knoten stark antikorrelieren, also je höher die Wahrscheinlichkeit in einem Knoten ist, desto geringer ist sie in einem anderen. Bildhafter in Abbildung 5 beschrieben ist die Ursache für ein wackelndes Haus durch einen hineinrasenden LKW sehr gering wenn bereits ein Erdbeben das Haus wackeln lässt. Hierbei wäre das wackelnde Haus ein Ausgangsknoten und Erdbeben und der LKW jeweils ein Eingangsknoten. Wenn einer der beiden Eingangsknoten also aktiviert ist, reicht ein Ereignis aus zur Erklärung, da das Eintreten beider Ereignisse sehr unwahrscheinlich ist. Dieses Verhalten macht es schwierig Schlussfolgerungen des Netzes nachzuvollziehen.

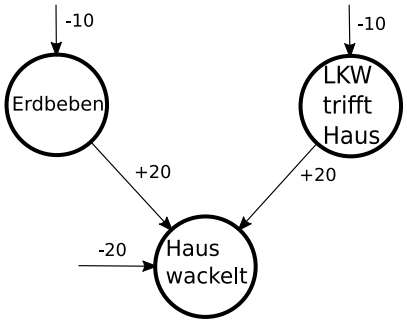

Abbildung 5: Beispiel für den Explained Away Effekt. Der Bias von -10 am Erdbebenknoten bedeutet dass dieser ohne beobachtet zu werden e^10 mal wahrscheinlicher aus als aktiviert ist. Beide Ereignisse zusammen haben eine Wahrscheinlichkeit von e^{-20}. Dies ist also sehr unwahrscheinlich. Deshalb erklärt die Aktivierung eines Knotens den anderen weg.

Die Annahme, dass die höheren Ebenen eine komplementäre Verteilung bilden, führt dazu, dass das Lernen von W_0 lediglich das Trainieren einer RBM darstellt und mithilfe der Contrastive Divergence eine gute Approximination gefunden wird. Sobald W_0 gelernt ist, kann man W_0^T, also die transponierte Gewichtsmatrix, als Eingabe für die erste versteckte Ebene benutzen, indem man die Eingabedaten mit dieser multipliziert.
Im allgemeinen findet die eingeschränkte Boltzmann Maschine kein perfektes Modell für die Eingabedaten. Das Modell kann durch einen einfachen greedy Algorithmus verbessert werden:

1. Lerne W_0.

2. Benutze W_0^T um die Verteilung der einzelnen Variablen in der ersten versteckten Ebene zu approximieren.

3. Lerne eine weitere Restricted Boltzmann Maschine mithilfe der "Daten", die durch W_0^T generiert wurden, für die nächste Ebene.

Jede durch den Algorithmus trainierte Ebene verbessert das Modell. Gleichzeitig bedeutet dies auch, dass jedes Modell durch hinzufügen von neuen Ebenen verbessert werden kann, wenn man jede Ebene sorgfältig genug trainiert. Wenn alle Ebenen die gleiche Anzahl von Knoten haben, besteht eine Verbesserung darin, dass jede neue Ebene mit den bereits gelernten Gewichten initialisiert wird. [4].

2.2 Deep-Belief Netze zur Klassifikation

Nachdem man das Netz mit dem im vorigen Kapitel beschriebenen Algorithmus trainiert hat, muss man immer noch die Labeldaten ins Netz speisen wenn man dieses für eine Klassifikation nutzen möchte. Dazu gibt es im wesentlichen zwei verschiedene Methoden:
Die erste Methode besteht darin, beim Training auf der vorletzten Ebene eine Gruppe von Softmax-Neuronen hinzuzufügen. Diese haben eine andere Aktivierungsfunktion als die normalen Neuronen:

$$p_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \tag{13}$$

Diese Aktivierungsfunktion sorgt dafür, dass die Aktivierung eines Neurons von der Aktivierung der anderen Neuronen abhängig ist und somit immer nur ein Neuron in der Gruppe aktiviert ist. Indem man abhängig vom Label ein Neuron fest auf 1 setzt beim trainieren, lernt das Netz welche Aktivierungszustände zu einer Klasse gehören. Wenn man nach dem Training einen Datensatz durchs Netz propagiert und anschließend ein paar Gibbs Samplings zwischen den letzten beiden Ebenen macht, kann man in der Softmax Gruppe das erkannte Label ablesen.
Die zweite Methode besteht darin, dass Deep-Belief Netz normal zu trainieren und nach dem Training eine Ebene mit Ausgabenneuronen an die letzte Ebene hinzuzufügen. Diese Neuronen können entweder die sigmoide Funktion als Aktivierung nutzen oder auch eine Softmax Gruppe bilden. Nun trainiert man die Gewichte zwischen der neuen Ebene und der letzten Ebene mit Backpropagation um die Labelinformationen ins Netz zu speisen. Um das Ergebnis im anschluß noch weiter zu verbessern, kann man die Gewichte des ganzen Netzes nochmals mit Backpropagation feiner einstellen, allerdings sollte das Netz auch ohne diesen Schritt bereits positive Ergebnisse liefern.
[5].

2.3 Implementation

Als Grundgerüst wurde ein bereits implementiertes Feedforward-Netz, welches bereits Backpropagation kann, genutzt. Abbildung 6 zeigt die Klassenherarchie des verwendeten Grund-

gerüsts. In rot sind die für den Aufbau und die Verwendung von Deep-Belief Netzen benötigten Klassen markiert.
Die Klasse ContrastiveDivergenceConfig liest die Parameter für die Kontrastive Divergenz aus einer Datei ein. Die Klasse kann einfach erweitert werden und setzt momentan die Anzahl der Gibbsschritte, die Anzahl der Epochen, die Lernrate und die Größe der zu trainierenden Batches. Der letzte Parameter hat jedoch keine Auswirkung, da die Implementation der Kontrastiven Divergenz aktuell nur Batchgrößen von 1 zulässt, also nach jedem Trainingsbild die Gewichte aktualisiert.
DBNLayer erweitert die Layerklasse soweit, dass statt normalen Neuronen Neuronen für das Deep-Belief-Netz benutzt werden und neben den normalen Gewichten, die daraus bestehen, dass diese bei zwei Layern nur von dem oberen Layer angesprochen werden können, auch vorwärtsgerichtete Gewichte initialisiert werden. Diese Gewichte stellen jedoch lediglich Zeiger auf die normalen Gewichte dar. Desweiteren wurde eine Möglichkeit geschaffen die Gewichte für das Deep-Belief Netz zu initialiseren. Dazu wird für Jedes Gewicht eine zufällige Zahl einer gausschen Normalverteilung mit einem Erwartungswert von 0 und einer Standardabweichung von $0,1$ gezogen.

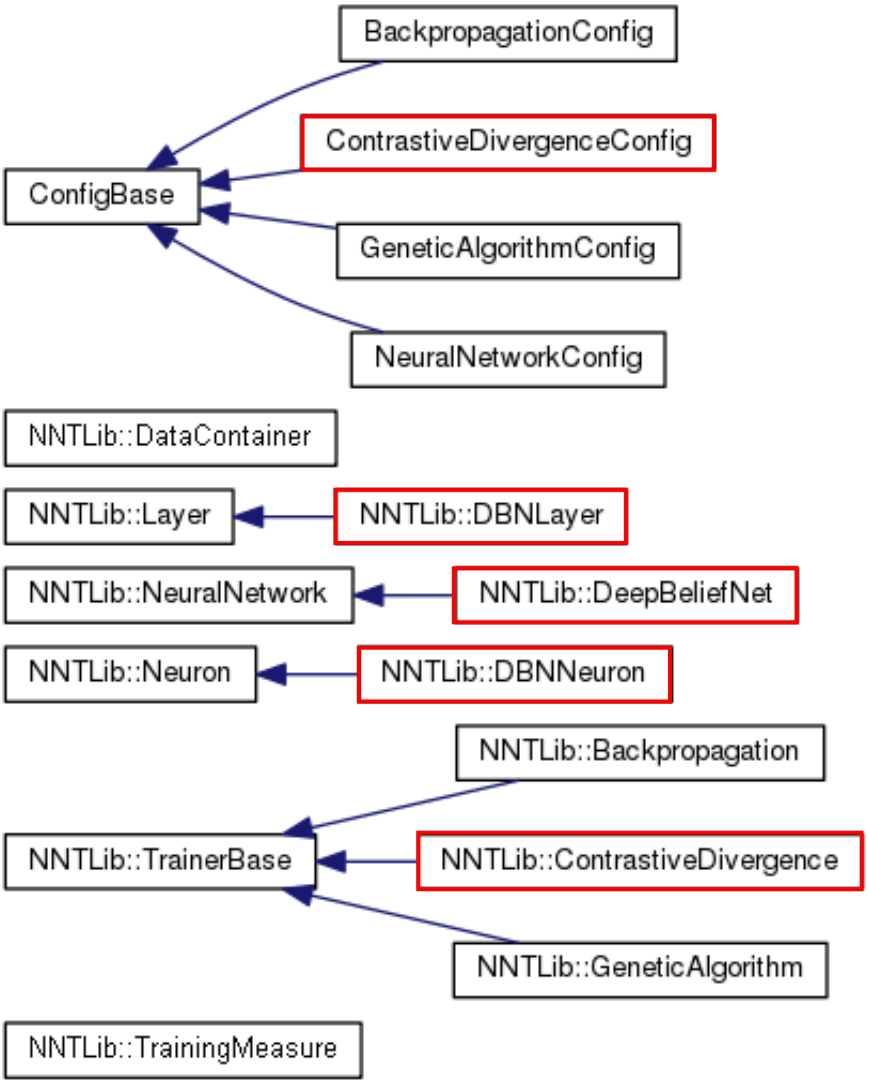

Abbildung 6: Klassenherarchie des verwendeten Frameworks, rot umrandete Klassen wurden in dieser Arbeit hinzugefügt.

DeepBeliefNet stellt die Netzstruktur für Deep-Belief Netze zur Verfügung. Im Gegensatz zu den normalen Netzwerken des Frameworks, besitzen Deep-Belief Netze auch auf der Eingabeebene eigene Neuronen. Dies ist wichtig, da man für die Kontrastive Divergenz die Aktivierungswahrscheinlichkeiten der einzelnen Neuronen speichern muss. Desweiteren besitzt das Netz die Funktion seine Gewichte so abzuspeichern, dass diese von einem normalen neuronalen Netz wieder eingelesen werden können. Dabei gehen die Biasgewichte in Rückwärtsrichtung verloren, da diese von dem normalen neuronalen Netz mit Backpropagation nicht benötigt werden.
DBNNeuron ist eine Erweiterung der Neuron Klasse. DBNNeuronen können ihre Aktivierungswahrscheinlichkeit speichern und besitzen die Möglichkeit auf die Gewichte zum nächsthöheren Layer zuzugreifen. Außerdem kam eine Funktion zum initialisieren von Bias-

gewichten hinzu. Diese bekommt entweder den Datencontainer für die Eingabedaten des Layers oder NULL übergeben. Wenn kein Container übergeben wird, wird der Bias auf 0 gesetzt. Dies ist der Fall, wenn der Biasknoten für die versteckte Ebene zuständig ist. Wenn ein Container übergeben wird, wird aus den Eingabedaten der Initialwert des Bias für jeden sichtbaren Knoten i berechnet:

$$\log[p_i/(1-p_i)] \tag{14}$$

p_i ist hierbei die Durchschnittliche Aktivierungswahrscheinlichkeit des Knoten i über alle Eingabedaten.
Als letzte wichtige Erweiterung wurde die Klasse ContrastiveDivergence implementiert, diese enthält den Lernalgorithmus für das Deep-Belief Netz.

Data: Eingabedaten, Epochenanzahl, Anzahl der Gibsschritte, Lernrate
Result: Traniertes Deep Belief Netz
Initialisiere leere Datencontainer für Eingabedaten für jede Ebene
for *jede Ebene* **do**
  Initialisiere Bias für versteckte Ebene mit 0
  Initialisiere Bias für sichtbare Ebene mit $\log[p_i/(1-p_i)]$
  for *Jeden Datensatz* **do**
    Setze Eingabedaten als Aktivierungswahrscheinlichkeit für sichtbare Ebene
    Bestimme Aktivierungen der sichtbaren Knoten
    Berechne Wahrscheinlichkeiten für versteckte Knoten:
    $p(h_j = 1|\vec{v}) = \sigma(b_j + \sum_i v_i w_{ij})$
    Sammle Statistik für $\langle v_i h_j \rangle_{data}$
    for *Anzahl Gibbsschritte* **do**
      Berechne Wahrscheinlichkeiten für sichtbare Knoten:
      $p(v_i = 1|\vec{h}) = \sigma(a_i + \sum_j h_j w_{ij})$
      Berechne Wahrscheinlichkeiten für versteckte Knoten
    end
    Sammle Statistik für $\langle v_i h_j \rangle_{model}$
    Passe Gewichte an: $w_{ij} = w_{ij} + \epsilon\left(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon}\right)$
    Passe Bias für sichtbare Knoten an: $a_i = a_i + \epsilon\left(\langle v_i \rangle_{data} - \langle v_i \rangle_{recon}\right)$
    Passe Bias für versteckte Knoten an: $b_i = b_i + \epsilon\left(\langle h_i \rangle_{data} - \langle h_i \rangle_{recon}\right)$
  end
  for *Jeden Datensatz* **do**
    Propagiere Eingabedaten durch trainierte Ebene und speichere Ergebnis in Datencontainer für die nächste Ebene
  end
end

Algorithm 1: Implementation der Kontrastiven Divergenz

Literatur

[1] Geoffrey Hinton, *A Practical Guide to Training Restricted Boltzmann Machines*, Department of Computer Science, University of Toronto, 2010

[2] Geoffrey Hinton, *Training Products of Experts by Minimizing Contrastive Divergence*, Neural Computation 14 Seite 1771-1800, 2002

[3] Ilya Sutske, Tijmen Tieleman, *On the convergence properties of contrastive divergence*, Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), 2010

[4] Geoffrey Hinton, Yee-Whye Teh, *A Fast Learning Algorithm for Deep Belief Nets*, Neural Computation 18 Seiten 1527-1554, 2006

[5] G. E. Hinton, R. R. Salakhutdinov, *Reducing the dimensionality of data with neural networks*, Science Vol. 313. no. 5786n Seite 504 - 507, 2006

[6] J. J. Hopfield, *Neural networks and physical systems with emergent collective computational properties*, Proceedings of the National Academy of Sciences (USA) 79, Seite 2554-2558, 1982

[7] S. Kullback, R.A Leibler, *On information and sufficiency*, Annals of Mathematical Statistics 22 No. 1, Seite 79–86, 1951