NN4PatternRecognition   
‚Neural Network for Pattern Recognition’

Teilleistung 4 im Kurs „Soft Computing“, Studiengang VAWi, Universität Duisburg-Essen, Sommersemester 2015

Wolfgang Bongartz

# Zusammenfassung

Das von mir erstellte Programm implementiert ein neuronales Netzwerk zur Erkennung von Bitmustern. Die zu erkennenden Bitmuster und alle für das Training des Netzes und den Programmablauf relevanten Parameter werden aus einem Textfile eingeladen. Danach wir das neuronale Netz zunächst auf die eingelesenen Bitmuster trainiert. Schließlich werden aus den eingelesenen Bitmustern zufällig veränderte Bitmuster erzeugt und dem neuronalen Netz dann in zufälliger Reihenfolge präsentiert.

# Der Aufruf des Programms

Das Programm wurde in Java erstellt und ist auf jedem PC lauffähig, auf dem eine Java-Laufzeitumgebung installiert ist (Version 1.8 oder höher). Es handelt sich um ein Terminalprogramm, das durch folgende Eingabe gestartet wird:

java –jar NN4PatternRecognition <filename>

Als <filename> wird eine Textdatei übergeben, die sowohl die Bitmuster als auch die Angaben zur Konfiguration enthält. Wird kein Dateiname angeben, so wird die Datei ‚good\_set.txt’ geladen.

# Die Topologie des neuronalen Netzes

Das implementierte neuronale Netz besteht aus einer Eingabe- und einer Ausgabeschicht. Eine Zwischenschicht existiert nicht. Jedem Neuron der Eingabeschicht ist genau ein Bit des Bitmusters zugeordnet. Jedem Neuron der Ausgabeschicht ist wiederum der Name eines der zu erkennenden Bitmuster zugeordnet. Jedes Eingabe-Neuron ist mit jedem Ausgabe-Neuron verbunden. Verbindungen zwischen den Neuronen einer Schicht existieren nicht.

Orientiert habe ich mich dabei am Buch "Neuronale Netze" von Günter Daniel Rey[[1]](#footnote-1).

# Der Aufbau der Input-Dateien

Alle Konfigurationsangaben werden nach dem Muster

<argument> = <wert>

angegeben.

## Programm-Konfiguration

| **Argument** | **Werte** |
| --- | --- |
| mode | Nach dem Training werden werden dem Netz nacheinander alle generierten Zufalls-Bitmuster präsentiert. Der Parameter ‚mode’ steuert, die einzelnen Bitmuster auch angezeigt werden.   * batch Die einzelnen Bitmuster werden nicht angezeigt. Erst nachdem das Netz alle Bitmuster bewertet hat wird eine Zusammenfassung angezeigt und das Programm wird beendet. * interactive Jedes Bitmuster wird zusammen mit dem Ergebnis der Bewertung durch das Netz angezeigt. Mit der Eingabetaste gelangt der Benutzer zum nächsten Bitmuster. Mit der Eingabe des Wortes ‚ende’ kann das Programm vorzeitig beendet werden. |
| numberOfRandomSamples | Definiert, wie viele Bitmuster nach dem Zufallsprinzip erzeugt werden sollen.  Muss größer oder gleich Null sein. Falls Null: Es werden dem Netz nur die Original-Bitmuster vorgelegt. |
| maxErrorsPerSample | Legt fest, wie viele Fehler jedes Zufalls-Bitmuster höchstens enthalten soll.  Mögliche Werte: Zwischen 0 und 10 (einschließlich). |
| presentOriginals | * yes Neben den Zufalls-Bitmustern werden dem Netz auch die Original-Bitmuster zur Erkennung vorgelegt, mit denen es trainiert wurde. * no Vorgelegt werden nur die Zufalls-Bitmuster. |

## Netz-Konfiguration

| **Argument** | **Werte** |
| --- | --- |
| epsilon | Legt die beim Training zu verwendende Lernrate fest. |
| laps | Legt fest, wie oft die Menge der Original-Bitmuster trainiert werden soll (Anzahl der Trainingsdurchläufe). |
| activation\_function | Legt die zu verwendende Aktivierungsfunktion fest. Möglich sind folgende Werte[[2]](#footnote-2):   * Linear a = nettoInput * Binary a = 1, falls nettoInput >= 0 a = 0, wenn nettoInput < 0 * TangensHyperbolicus a = tanh( nettoInput ) * Logistic Logistische Aktivierungsfunktion |
| learnmode | Legt fest, zu welchem Zeitpunkt die Verbindungsgewichte zwischen den Input- und den Output-Neuronen angepasst werden sollen:   * batch Zunächst werden alle Bitmuster präsentiert. Die dabei ermittelten Änderungen werden aufsummiert und erst im Anschluss zu den Verbindungsgewichten addiert. * online Die Verbindungsgewichte werden nach jedem einzelnen Bitmuster aktualisiert. |

## Bitmuster

| **Argument** | **Werte** |
| --- | --- |
| width | Höhe der Bitmuster in Bit. Alle Bitmuster in der Datei müssen die gleiche Höhe haben. |
| height | Breite der Bitmuster in Bit. Alle Bitmuster in der Datei müssen die gleiche Breite haben. |
| name | Name des der Zeile folgenden Bitmusters. Jedes Bitmuster muss einen eindeutigen Namen haben. |

Ein Bitmuster wird einfach durch zeilenweise Angabe der einzelnen Bitwerte definiert. Dabei werden das Zeichen ‚.’ („Punkt“) als 0 und das Zeichen ‚X’ als 1 interpretiert.

Beispiel:

name=a

.XXX..

X...X.

....X.

.XXXX.

X...X.

X..XX.

.XX.X.

# Die Input-Dateien

Ich habe drei Input-Dateien erstellt und mit diesen Dateien ein wenig experimentiert. Die Ergebnisse möchte ich hier kurz erläutern.

Zunächst wurden dem neuronalen Netz immer nur die Original-Bitmuster zur Erkennung vorgelegt. So war zu erkennen, ob das Netz diese Bitmuster tatsächlich „gelernt“ hat[[3]](#footnote-3). Dies habe ich für jede Aktivierungsfunktion getan und dabei die restlichen Konfigurations-Parameter für das neuronale Netz möglichst passend gewählt. Zu jeder Aktivierungsfunktion habe ich so die „beste“ erreichbare Erkennungsrate ermittelt.

Danach wurden dem neuronalen Netz jeweils 100.000 zufällig erzeugte Bitmuster zur Erkennung vorgelegt[[4]](#footnote-4). So war zu erkennen, ob das Netz die Original-Bitmuster nur „auswendig gelernt“ hat oder ob es auch fehlerhafte Bitmuster korrekt zuordnen kann. Dabei bin ich vorgegangen wie oben beschrieben.

## Zusammenfassung der Ergebnisse

Die Einzelergebnisse folgen in den Abschnitten 5.2 bis 5.4.

### Original ohne Fehler

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Datei** | **‚Linear’** | **‚Binary’** | **‚Logistic’** | **‚TangensHyperbolicus’** |
| **good\_set.txt** | 100% | 100% | 100% | 100% |
| **avg\_set.txt** | 100% | 100% | 100% | 100% |
| **bad\_set.txt** | 90% | 90% | 90% | 90% |

Interpretation: Das neuronale Netz ist offenbar in der Lage, bis zu 9 Bitmuster zu lernen, wenn diese sich genügend voneinander unterscheiden. Für das Lernen von 10 Bitmustern, die sich nur in 42 Bits voneinander unterscheiden ist ein zweischichtiges Netz anscheinend ungeeignet.

### Zufallsmuster mit max. 10 Fehlern

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Datei** | **‚Linear’** | **‚Binary’** | **‚Logistic’** | **‚TangensHyperbolicus’** |
| **good\_set.txt** | 88% | 71% | 91% | 90% |
| **avg\_set.txt** | 87% | 70% | 93% | 90% |
| **bad\_set.txt** | 80% | 62% | 82% | 80% |

Interpretation: Bis zu 10 Bitmuster werden auch bei hoher Fehlerrate noch zuverlässig erkannt. Dabei scheint die logistische Aktivierungsfunktion einen leichten Vorteil zu bringen.

## Die Datei ‚good\_set.txt’

Die Datei enthält sechs Bitmuster, die jeweils 42 Bits umfassen. Davon sind 35 Bits relevant für die Unterscheidung der Bitmuster[[5]](#footnote-5).

### Aktivierungsfunktion ‚Linear’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.05 |
| laps | 100 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 99% |
| Zufall, max. 5 Fehler | 96% |
| Zufall, max. 10 Fehler | 88% |

### Aktivierungsfunktion ‚Binary’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.1 |
| laps | 25 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 91% |
| Zufall, max. 5 Fehler | 82% |
| Zufall, max. 10 Fehler | 71% |

### Aktivierungsfunktion ‚Logistic’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.5 |
| laps | 50 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 99% |
| Zufall, max. 5 Fehler | 97% |
| Zufall, max. 10 Fehler | 91% |

### Aktivierungsfunktion ‚TangensHyperbolicus’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.1 |
| laps | 25 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 98% |
| Zufall, max. 5 Fehler | 96% |
| Zufall, max. 10 Fehler | 90% |

## Die Datei ‚avg\_set.txt’

Die Datei enthält neun Bitmuster, die jeweils 72 Bits umfassen. Davon sind 49 Bits relevant für die Unterscheidung der Bitmuster.

### Aktivierungsfunktion ‚Linear’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.01 |
| laps | 20 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 97% |
| Zufall, max. 5 Fehler | 93% |
| Zufall, max. 10 Fehler | 87% |

### Aktivierungsfunktion ‚Binary’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.2 |
| laps | 50 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 92% |
| Zufall, max. 5 Fehler | 82% |
| Zufall, max. 10 Fehler | 70% |

### Aktivierungsfunktion ‚Logistic’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.2 |
| laps | 30 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 99% |
| Zufall, max. 5 Fehler | 97% |
| Zufall, max. 10 Fehler | 93% |

### Aktivierungsfunktion ‚TangensHyperbolicus’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.05 |
| laps | 30 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 100% |
| Zufall, max. 2 Fehler | 98% |
| Zufall, max. 5 Fehler | 95% |
| Zufall, max. 10 Fehler | 90% |

## Die Datei ‚bad\_set.txt’

Die Datei enthält 10 Bitmuster, die jeweils 72 Bits umfassen. Davon sind 49 Bits relevant für die Unterscheidung der Bitmuster.

### Aktivierungsfunktion ‚Linear’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.02 |
| laps | 30 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 90% |
| Zufall, max. 2 Fehler | 88% |
| Zufall, max. 5 Fehler | 85% |
| Zufall, max. 10 Fehler | 80% |

### Aktivierungsfunktion ‚Binary’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.1 |
| laps | 50 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 90% |
| Zufall, max. 2 Fehler | 82% |
| Zufall, max. 5 Fehler | 73% |
| Zufall, max. 10 Fehler | 62% |

### Aktivierungsfunktion ‚Logistic’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.1 |
| laps | 50 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 90% |
| Zufall, max. 2 Fehler | 88% |
| Zufall, max. 5 Fehler | 86% |
| Zufall, max. 10 Fehler | 82% |

### Aktivierungsfunktion ‚TangensHyperbolicus’

Konfiguration:

|  |  |
| --- | --- |
| **Argument** | **Wert** |
| epsilon | 0.05 |
| laps | 30 |
| learnmode | batch |

Ergebnisse:

|  |  |
| --- | --- |
| **Bitmuster** | **Erkennungsrate** |
| Original | 90% |
| Zufall, max. 2 Fehler | 88% |
| Zufall, max. 5 Fehler | 85% |
| Zufall, max. 10 Fehler | 80% |

# Die Programmstruktur

## Das Hauptprogramm

Die Klasse NN4PatternRecognition beinhaltet das Hauptprogramm (die Methode main()). Beim Programmstart wird eine Instanz dieses Objektes erzeugt, die sich dann um das Einlesen der Konfigurationsdatei und um die Steuerung des weiteren Programmablaufs kümmert.

Die öffentlichen Methoden in der Reihenfolge, in der sie nach dem Programmstart durchlaufen werden:

| **Methode** | **Funktion** |
| --- | --- |
| main() | Dies ist eine Klassen-Methode (static).  Erzeugen eines Objekts der Klasse und Übergabe des weiteren Ablaufs an die Run-Methode des Objekts. |
| NN4PatternRecognition() | Dies ist der Konstruktor der Klasse.  Laden der Konfigurationsdatei und interpretieren des Inhalts dieser Datei (via loadConfigFile()). Initialisieren der Objekt-Attribute. |
| run() | Steuert den Programmablauf:   1. Erzeugen des neuronalen Netzes 2. Trainieren des neuronalen Netzes 3. Erzeugen zufälliger Bitmuster 4. Bitmuster zur Erkennung ans Netz übergeben 5. Abschließende Statistik anzeigen   Der Schritt 4 wird wiederholt, bis alle Bitmuster abgearbeitet sind oder der Benutzer die weitere Bearbeitung abbricht. |

## Das Package ‚nnet’

### Die Klasse PatternAssociator

Diese Klasse implementiert das oben beschriebene neuronale Netz. Anstatt die einzelnen Neuronen ihrerseits in Form von Objekten zu implementieren (was anschaulicher wäre) werden sie innerhalb der Klasse PatternAssociator in Form der beiden Vektoren \_inputVector und \_outputVector repräsentiert, welche die aktuellen Aktivierungswerte enthalten. Da jedes Neuron der Input-Schicht mit jedem Neuron der Output-Schicht verbunden ist wird keine explizite Repräsentation der Verbindungen benötigt. Statt dessen beinhaltet die Matrix \_weigths die aktuelle Gewichtung aller Verbindungen. Auf diese Weise lassen sich die Operationen auf die Neuronen und die Verbindungen recht leicht über Matrizen- bzw. Vektor-Operationen implementieren[[6]](#footnote-6).

Die öffentlichen Methoden:

| **Methode** | **Funktion** |
| --- | --- |
| PatternAssociator() | Dies ist der Konstruktor der Klasse.  Erzeugen der Netz-Topologie (via createNetworkTopology()). Das bedeutet im Wesentlichen: Vorbereiten der Gewichtungs-Matrix. |
| train() | Trainiert das Netz mit den als Parameter übergebenen Mustern. Die Anpassung der Gewichtungs-Matrix erfolgt dabei anhand der Delta-Regel, die in der Klasse DeltaRule implementiert ist. |
| match() | Präsentiert dem Netz ein einzelnes als Parameter übergebenes Bitmuster, wertet dann den Aktivierungszustand der Output-Neuronen aus und leitet daraus die Antwort des Netzes ab.  Dabei wird das übergebene Bitmuster in die Input-Neuronen (also den Input-Vektor) übertragen. Daraufhin wird dann der Aktivierungszustand der Output-Neuronen (des Output-Vektors) berechnet. Die Aktivierungsfunktion selbst ist in der dabei verwendeten Klasse ActivationFunction (bzw. deren Abkömmlingen) enthalten. |

Anmerkung: Für einige Matrizen-Berechnungen wird eine Methode der Matrix-Klasse verwendet (Array2DRowRealMatrix.walkInDefaultOrder()), die für jedes Element der Matrix die Methode visit() eines übergebenen Objekts ausführt.

### Die ActivationFunction-Klassen

Diese Klassen beinhalten verschiedene Aktivierungsfunktionen, die sich dann konfigurativ aktivieren lassen.

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| ActivationFunction | Dies ist die abstrakte Basisklasse aller weiteren hier beschriebenen Klassen. Sie beinhaltet alle Methoden, die nötig sind, um in der Matrix-Methode walkInDefaultOrder() verwendet zu werden sowie die grundsätzliche Berechnungsvorschrift für Aktivierungswerte. Alle abgeleiteten Klassen implementieren lediglich die Methode getFunctionResult(), die dann den konkreten Wert liefert. |
| ActivationFunction\_Binary | Implementiert eine binäre Aktivierungsfunktion ohne Schwellwert. |
| ActivationFunction\_Linear | Implementiert eine lineare Aktivierungsfunktion ohne Schwellwert und Anpassungsfaktor. |
| ActivationFunction\_Logistic | Implementiert eine logistische Aktivierungsfunktion. |
| ActivationFunction\_TangensHyperbolicus | Implementiert die Aktivierungsfunktion 'Tangens Hyperbolicus'. |

### Die Klasse DeltaRule

Implementiert die Delta-Lernregel als Visitor-Objekt für die Matrix-Methode walkInDefaultOrder(). Berechnet also für jedes Element der Delta-Gewichtsmatrix den Anpassungswert.

### Sonstige Klassen

|  |  |
| --- | --- |
| **Klasse** | **Funktion** |
| SetValueVisitor | Implementiert ein Visitor-Objekt für die Matrix-Methode walkInDefaultOrder(), dass alle Elemente der Matrix auf einen bestimmten, im Konstruktor übergebenen Wert setzt. |
| InappropriateDataException | Exception die geworfen wird, falls versucht wird, dem Netz Trainingsdaten zu präsentieren, die mehr Output-Neuronen erfordern als tatsächlich vorhanden sind. |
| NeuralNetworkIsUntrainedException | Exception die geworfen wird, falls ein neuronales Netz nicht trainiert wurde. |

## Das Package ‚patterns’

Dieses Package enthält die für die Verwaltung von Bitmustern nötigen Klassen.

### Die Klasse Pattern

Objekte dieser Klasse repräsentieren ein Bitmuster. Ein Bitmuster hat einen Namen, eine Breite und eine Höhe (jeweils in Bit). Neben Methoden, die das Setzen und das Auslesen einzelner Bits erlauben bietet die Klasse noch die folgenden Methoden an, welche verwendet werden, um abgeleitete Bitmuster zu erzeugen.

|  |  |
| --- | --- |
| **Methode** | **Funktion** |
| swapRandomly | Vertauscht den Inhalt zweier zufällig ausgewählter Zellen des Bitmusters. |
| changeRandomly | Kehrt den Inhalt einer zufällig ausgewählten Zelle um. |

### Die Klasse PatternSet

Objekte dieser Klasse verwalten eine Menge von Bitmustern.

|  |  |
| --- | --- |
| **Methode** | **Funktion** |
| get\_pattern | Liefert ein Bitmuster. |
| size | Liefert die Gesamtanzahl der derzeit gespeicherten Bitmuster. |
| chooseRandomly | Liefert ein zufällig ausgewähltes Bitmuster. |
| createRandomized | Erzeugt ein neues PatternSet-Objekt, dessen Inhalt aus zufällig ausgewählten und nach dem Zufallsprinzip veränderten Bitmustern besteht. |
| getListOfName | Liefert eine Liste aller vorkommenden Bitmuster-Namen. |
| getNumberOfRelevantBits | Liefert die für das aktuelle PatternSet gültige Anzahl relevanter Bits. Das sind die Bits, in denen sich wenigstens zwei Muster voneinander unterscheiden. |

### Die Klasse RandomPatternIterator

Diese Klasse implementiert einen Iterator, mit dessen Hilfe die Pattern-Objekte in einem PatternSet in zufälliger Reihenfolge durchlaufen werden können, ohne das ein Pattern-Objekt zweimal „besucht“ wird.

Der Standard-Iterator, den die Klasse PatternSet mit der Methode iterator() liefert durchläuft die Pattern-Objekte dagegen immer in derselben Reihenfolge.

# Die Bestandteile dieser Teilleistung

Diese Teilleistung besteht aus den folgenden Dateien und Verzeichnissen:

* NN4PatternRecognition.jar  
  Dies ist das ausführbare Programm.
* Dokumentation.doc  
  Die vorliegende Dokumentation.
* good\_set.txt, avg\_set.txt und bad\_set.txt  
  Die oben beschriebenen Beispieldateien.
* Dem Verzeichnis ‚src’  
  Enthält alle Quelltexte.
* Das Verzeichnis ‚doc’  
  Enthält die mit JavaDoc generierte HTML-Dokumentation aller Packages und Klassen. Startpunkt: ‚index.html’.
* Das Verzeichnis ‚lib’  
  Enthält die für Matrix- und Vektor-Operationen verwendete Library „Apache Commons Math“ (siehe <http://commons.apache.org/proper/commons-math/> ).

1. Neuronale Netze", Günter Daniel Rey & Karl F. Wender, 2. Auflage, 2001, ISBN 978-3-456-84881-5; im Folgenden mit [Rey] referenziert [↑](#footnote-ref-1)
2. Siehe auch [Rey], S. 18 ff [↑](#footnote-ref-2)
3. Konfiguration: numberOfRandomSamples=0 und presentOriginals=yes [↑](#footnote-ref-3)
4. Konfiguration: numberOfRandomSamples=100000 und presentOriginals=no [↑](#footnote-ref-4)
5. Ein Bit ist dann relevant für die Unterscheidung der Bitmuster, wenn es wenigstens zwei Bitmuster gibt, in denen dieses Bit unterschiedliche Werte hat. [↑](#footnote-ref-5)
6. Für Matrizen- und Verktoroperationen wird die Library ‚Apache Commons Math’ verwendet (http://commons.apache.org/proper/commons-math/). [↑](#footnote-ref-6)