Contents

Chapter 1

TODOs

1.1 TechDoc

Update of chapters Neuron, Connection, MultiLayerPerceptron

1.2 General

Description, Header (File-Header too), Comments

1.3 class MultiLayerPerceptron

Default Konstructoren (mit weniger Variablen), die Hauptkonstruktor, mit Default-Werten aufrufen

1.4 Tests

artificialNeuralNetwork durchgehen und Test entwerfen, die jede Zeile und Sonderfaelle abdecken
Vergleich von verschiedenen Topologien, welche erreicht am schnellsten die Loesung
Alle Tests rein kommentieren, Ergebnisse in extra txt-files

Chapter 2

class Neuron

Uses step function for calculating the output.

2.1 Attributes

2.1.1 int inputs

For all inputs one integer variable is used. So the neuron can handle up to 32 inputs, which can just be 0 or 1.

2.1.2 int nrInputs

Specifies how many inputs (respectively bits of the variable `inputs`) shall be used.

2.1.3 float netInput

The result of the net input function:
$netInput = input.1 * weights[0] + ... + input.32 * weights[31]$
So far also used as output.

2.1.4 float weights[]

The weights of the inputs. Array size depends on the number of inputs.

2.1.5 float threshold

The threshold when output is activated.

2.2 Constructors

2.2.1 Neuron(int nrInputs)

Sets `nrInputs`, size of array `weights` and initialzes `inputs`, `netInput`, and `weights` to 0, `activated` to false, and `threshold` to max ($0x7fffffff$).

2.3 Methods

2.3.1 Boolean setInput(int nrInput)

Sets given input to 1 by shifting a 1 to the corresponding bit in `inputs`.
Returns true if input was set, returns false if input number is to high.

2.3.2 Boolean unsetInput(int nrInput)

Sets given input to 0 by shifting a 1 to the corresponding bit in `inputs` and bitwise inverting it.
Returns true if input was unset, returns false if input number is to high.

2.3.3 Boolean setWeight(int nrInput, float weight)

Puts the given weight into the for the given input corresponding field in the weights array.
Returns true if weight was set, returns false if input number is to high.

2.3.4 void setThreshold(float threshold)

Sets threshold for the neuron.

2.3.5 float getOutput()

Calculates the `netInput`, the result of the net input function, and uses the activation function for returning the output.
So far, there is no extra output variable in the class, since the output function is just the identity. So the method returns the variable `netInput`.
So far, the activation function is just a step function, which is implemented as a simple if-else.

Chapter 3

class Connection

Connects the output of a neuron to an input of another neuron. The weight of the following input is the output value times the weight of the connection.

3.1 Attributes

3.1.1 Neuron neuronFrom

Neuron the output is taken from.

3.1.2 Neuron neuronTo

Neuron the input is set.

3.1.3 int input

Input of `neuronTo` which is set.

3.1.4 float connWeight

Weight of the connection.

3.1.5 float weightToSet

Weight set in `neuronTo`. Output of `neuronFrom` $*$ weight of connection.

3.2 Constructors

3.2.1 Connection(Neuron neuronFrom, Neuron neuronTo, int input, float connWeight)

Just sets `neuronFrom`, `neuronTo`, `input`, and `connWeight`.

3.3 Methods

3.3.1 void run()

Calculates the weight for `neuronTo`. The weight, which is set, is calculated by `getOutput()` from `neuronFrom` $*$ the weight of the connection.for `neuronTo`. The calculated weight is

set to the given input.
The method uses `getActivation()` from `neuronFrom` to set or unset the given input of `neuronTo`.

Chapter 4

class MultiLayerPerceptron

Automatically builds a multi-layer perceptron. The number of input neurons and output neurons can be set in any case, more possibilities depends on the constructors.
The maximum number of neurons for one layer is 32.

4.1 Attributes

4.1.1 Neuron inputNeuron[]

Input layer, size depends on definition.

4.1.2 Neuron hiddenNeuron[][]

Hidden layer(s), dimension 1 depends on how many hidden layers are defined (or calculated). Dimension 2 depends on how many hidden neurons are defined (or calculated).

4.1.3 Neuron outputNeuron[]

Output layer, size depends on definition.

4.1.4 int hiddenLayers

Size of 1st dimension of array `hiddenNeuron`. Depending on the constructor this value may be settable or be calculated in the constructor.

4.1.5 float inputConnWeights[]

Weights of the connections between input layer and 1st hidden layer.

4.1.6 float hiddenConnWeights[][]

Weights of the connections between the hidden layers (if more than 1 is defined or calculated). The size of the 1st dimension of this array is set by `hiddenLayers`.

4.1.7 float outputConnWeights[]

Weights of the connections between the last hidden layer and the output layer.

4.1.8 float outputVector[]

The result of the multi layer perceptron, is returned by the method `run()`. The size of the array depends on the number of output neurons.

4.1.9 Connection inputConnection[]

Connections between input layer and 1st hidden layer.

4.1.10 Connection hiddenConnection[][]

Connections between the hidden layers (if more than 1 is defined or calculated). The size of the 1st dimension of this array is set by `hiddenLayers`.

4.1.11 Connection outputConnection[]

Connections between the last hidden layer and the output layer.

4.2 Constructors

4.2.1 MultiLayerPerceptron(int inputNeurons, int hiddenNeuronsPerLayer, int hiddenLayers, int outputNeurons)

The constructor creates a multi-layer perceptron with the given number of input neurons, number of hidden neurons per layer, number of hidden layers and number of output neurons.
Each input neuron is implemented with one input. All neurons will be connected automatically to each neuron of the following layer, except of the output layer. So the number of inputs for the hidden neurons is calculated by the number of input neurons for the first layer and by the number of hidden neurons for all hidden layers and, in case of just one output neuron, for the output layer. If there are more than one output neuron the outputs of the last hidden layer are split to the output neurons, the lower hidden neurons to the lower output neurons etc., so the number of inputs will be calculated by dividing the number of hidden neurons by the number of output neurons. Therefore it is not possible to have more output neurons than hidden neurons and there must be at least one hidden layer. In case the division has a remainder one more input is added to each output neuron.
The weights and the thresholds of all neurons and the weights of all connections will be initialised with 1.
The constructor calculates the amount of weights for each layer, stored in the corresponding arrays and then sets the connections.
The output vector is set to 0.

4.3 Methods

4.3.1 float[] run(int inputVector)

The method just executes the built multi-layer perceptron. The inputs of the input neurons are set with the given input vector. Than the connections of the different layers are executed one after another, starting with the first connection of the input layer, by calling the `run()`-methods of the connections. Finally, the `getOutput()`-methods of the output-neurons are called and the output vector returned.

4.3.2 void backpropagation(float[] resultOut, float[] wantedOut)

```
1  /*
     ********************************************************************************
2    * TODO: Klaeren ob inkl. threshold
3    * Kommentare in TechDoc
4    *
5    *
       ------------------------------------------------------------------------------

6    * Beispiel: 2 Input, 4 Hidden, 2 Hiddenlayer, 2 Output
7    *   Connection each, Oupuzt connection 2 Groups
8    *                 ------------------------------------------------------
9    *   inputConnection[] | hiddenConnection[][] | outputConnection[]
10   * 0 : 0 >      0 >      0
11   *          1
12   *          2
13   *          3
14   *   1 >      0
15   *          1
16   *          2
17   *          3
18   *   2 >      0
19   *          1 >      0
20   *          2
21   *          3
22   *   3 >      0
23   *          1
24   *          2
25   *          3
26   * 1 : 0 >      0
27   *          1
28   *          2 >      1
29   *          3
30   *   1 >      0
31   *          1
32   *          2
33   *          3
34   *   2 >      0
35   *          1
36   *          2
37   *          3 >      1
38   *   3 >      0
39   *          1
40   *          2
41   *          3
42   *                -----------------------------------------------------
43   *
       ------------------------------------------------------------------------------

44   *
45   * Backpropagation allgemein:
46   * deltaGewichtsvektor_u = Lernfaktor (Summe Ã¼ber Folgeneuronen((Summe Ã¼ber
       Ausgabeneuronen(
47   * (Erwartete Ausgabe - Ausgabe) dAusgabe_Ausgabeneuron nach
       dNetzeingabe_Folgeneuron))*Gewicht_Neuron-FolgeNeuron)
48   * ) dAusgabe_u nach dNetzeingabe_u * Eingabevektor_u (S.71)
49   *
50   * FÃ¼r Identiaet als Ausgabefunktion und logistischer Aktivierungsfunktion:
51   * deltaGewichtsvektor_u = Lernfaktor (Summe Ã¼ber Folgeneuronen((Summe Ã¼ber
       Ausgabeneuronen(
52   * (Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron)dAusgabe_Ausgabeneuron
         nach dNetzeingabe_Folgeneuron))*Gewicht_FolgeNeuron-u)
53   * ) Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
54
55   * -> SCHICHTENWEISE
56   -> Summe Ã¼ber Ausgabeneuronen:
```

```
57    * -> Summe Ã¼ber Folgeneuronen: Pfad im Netz muss mathematisch nachgebildet
          werden
58    * -> Eingabevektor muss angepasst werden, enthaelt Bitmuster
59    * -> bei logistischer Funktion:
60    * dAusgabe_Ausgabeneuron nach dNetzeingabe_Folgeneuron = Ausgabe_Ausgabeneuron (1
          - Ausgabe_Ausgabeneuron)
61    *
62    * Einzelner Gradientenabstieg:
63    * delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 -
          Ausgabe) * Eingabevektor_u
64    * -> pro inputneuron:
65    * delta Gewicht_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 - Ausgabe
          ) * Eingabe_u
66    *
67    * Um deltaGewicht fÃ¼r jede verbindung zu berechnen erst mit for-loop Ã¼ber
          outputconnection.length
68    * dann for (int i = hiddenLayer-2; i>=0; i++) -> loop Ã¼ber hiddenLayer[i].length
69    * dann for-loop Ã¼ber inputConnection.length
70    *
71    * Um Folgeneuronen zu ermitteln Position in Array benÃ¶tigt, dann loops Ã¼ber die
          folgende
72    * Netzstruktur (nicht einfach Ã¼ber die ConnectionArrays)
73    ********************************************************************************
          */
74  private void backpropagation(float[] resultOut, float[] wantedOut){
75      // TODO: Kommentare wofÃ¼r Variablen, Richtige Reihenfolge
76      int connectionsOfNeuron;
77      int neuronsInLayer;
78      int neuronsInNextLayer;
79      int offset;
80      float trainingCoefficient = 0.2f;
81      float weightDelta = 0;
82      float connWeights[];
83      Connection connections[];
84      float outputOfNeuron;
85      float inputVectorOfNeuron;
86
87      // Um deltaGewicht fÃ¼r jede verbindung zu berechnen loop Ã¼ber alle (
            Verbindungs-)Layer...
88      for (int layer = hiddenLayers; layer >= 0; layer--){
89        // Bestimmen der Position des Neurons im Netz und der folgenden
90        // Struktur fÃ¼r die Summe Ã¼ber die Folgeneuronen,
91        // d.h. fÃ¼r die Grenze der for-Schleife. Daher nur int neuronsInLayer
              benÃ¶tigt
92        // -> mit layer ermitteln in welchem Layer: output, hidden o input
93        // -> mit neuron an welcher Stelle im Layer
94        // Grenze fÃ¼r Summe Ã¼ber Folgeneuronen
95
96        // DEBUG
97        System.out.println("int layer = " + hiddenLayers + "; layer >= 0; layer: " +
              layer + "{");
98
99        // Outputconnection layer
100       if (layer == hiddenLayers){
101         neuronsInLayer = hiddenNeuron[layer-1].length;
102         neuronsInNextLayer = 1;
103
104         // Es gibt nur Verbindung(en) vom letztem Hidden Layer zu OutputNeuron(en)
105
106         // TODO: Abbruchbedingung, ist noch falsch, benÃ¶tigt allgemein gÃ¼ltigen
                Ausdruck
107         // Go to next output neuron if all inputs are connected
108         // Funktioniert nur wenn neuron > 0
109         //while ((neuron % (hiddenNeuron[hiddenLayers-1].length/outputNeuron.length
                )) != 0){}
110
111         // Im Moment nur eine Verbindung von hidden Neuron zu outputLayer
```

```
112           connectionsOfNeuron = 1;
113
114         /*************************************************************
115         // Connections between last hidden and output layer
116         int inp = 0;
117         int outNeur = -1;
118
119         // From each hidden neuron...
120         for (int hidNeur = 0; hidNeur < hiddenNeuron[hiddenLayers-1].length; hidNeur
                ++, inp++){
121
122           // Go to next output neuron if all inputs are connected
123           if ((hidNeur % (hiddenNeuron[hiddenLayers-1].length/outputNeuron.length))
                  == 0){
124             outNeur++;
125             inp = 0;
126           }
127
128           // ...to the "nearest" output neuron (if several are available)
129           outputConnection[hidNeur] = new Connection(hiddenNeuron[hiddenLayers-1][
                  hidNeur],
130               outputNeuron[outNeur], inp, outputConnWeights[hidNeur]);
131         }
132         *************************************************************/
133
134           // TODO: Auch hier allgemeine Formulierungen
135           connWeights = new float [outputConnWeights.length];
136           connWeights = outputConnWeights;
137
138           connections = new Connection [outputConnection.length];
139           connections = outputConnection;
140         }
141
142         // InputConnection layer
143         else if (layer == 0){
144           neuronsInLayer = inputNeuron.length;
145           neuronsInNextLayer = hiddenNeuron[0].length;
146
147           // TODO: Allgemein gÃ¼ltiger Ausdruck
148           // Im Moment nur von jeden inputNeuron zu jedem hiddenNeuron
149           connectionsOfNeuron = hiddenNeuron[0].length;
150
151         /*************************************************************
152         // Connections between input layer and 1st hidden layer
153         int positionInLayer = 0;
154
155         // From each input neuron...
156         for (int inp = 0; inp < inputNeuron.length; inp++){
157
158           // ...to each neuron of the 1st hidden layer
159           for (int conn = 0; conn < hiddenNeuron[0].length; conn++){
160
161             inputConnection[positionInLayer] = new Connection(inputNeuron[inp],
162                 hiddenNeuron[0][conn], inp, inputConnWeights[positionInLayer]);
163
164             positionInLayer++;
165           }
166         }
167         *************************************************************/
168
169           connWeights = new float [inputConnWeights.length];
170           connWeights = inputConnWeights;
171
172           connections = new Connection [inputConnection.length];
173           connections = inputConnection;
174
175         /*************************************************************
176         // Connection weights
177         inputConnWeights = new float[inputNeurons * hiddenNeuronsPerLayer];
178
```

```
179         hiddenConnWeights = new float[hiddenLayers-1][hiddenNeuronsPerLayer *
                hiddenNeuronsPerLayer];
180
181         outputConnWeights = new float[hiddenNeuronsPerLayer];
182
183         Struktur siehe Connections weiter oben
184         ************************************************************/
185         }
186
187         // Hidden Layer
188         else {
189           neuronsInLayer = hiddenNeuron[layer-1].length;
190           neuronsInNextLayer = hiddenNeuron[layer].length;
191
192           // TODO: Allgemein gÃ¼ltiger Ausdruck
193           // Im Moment nur von jeden hiddenNeuron zu jedem hiddenNeuron im Folgelayer
194           connectionsOfNeuron = hiddenNeuron[layer].length;
195
196           connWeights = new float [hiddenConnWeights[layer-1].length];
197           connWeights = hiddenConnWeights[layer-1];
198
199           connections = new Connection [hiddenConnection[layer-1].length];
200           connections = hiddenConnection[layer-1];
201         }
202         //DEBUG
203         System.out.println("\tconnWeights.lenght: " + connWeights.length);
204
205         // ..., Ã¼ber alle Neuronen in dem Layer...
206         for (int neuron = 0; neuron < neuronsInLayer; neuron++){
207           //DEBUG
208           System.out.println("\tfor (int neuron = 0; neuron < "+neuronsInLayer+";
                neuron: "+neuron+"{");
209
210           // Position of 1st connection of this neuron (Is the same
211           // for all neurons in this layer)
212           offset = neuron * connectionsOfNeuron;
213
214           // ...und nu Ã¼ber alle Verbindungen des Neurons
215           for (int conn = 0; conn < connectionsOfNeuron; conn++){
216             //DEBUG
217             System.out.println("\t\tfor (int conn = 0; conn < "+connectionsOfNeuron+"
                    ; conn" +conn+ "{");
218
219             /*** Notwendig?
220             // pro Inputneuron:
221             // Notwendig? wie sonst inputVector[inp] verwenden?
222             for (int inp = 0; inp < inputNeuron.length; inp++){
223             }// for() pro Inputneuron
224             ***/
225
226             // Beginn der Berechnung
227             // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
228             // weightDelta = trainingCoefficient *
229
230             // DEBUG
231             System.out.println("\t\tneuronsInNextLayer: " + neuronsInNextLayer);
232
233             // Summe Ã¼ber Folgeneuronen
234             for (int succ = 0; succ < neuronsInNextLayer; succ++){
235
236               // Summe Ã¼ber Ausgabeneuronen
237               for (int out = 0; out < outputNeuron.length; out++){
238                 weightDelta += (
239                 // Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron
240                 (wantedOut[out] - resultOut[out]) *
241                 //dAusgabe nach dNetzeingabe
242                 resultOut[out] * (1- resultOut[out]));
243               }// for() Summe Ã¼ber Ausgabeneuronen
```

```
244
245             // Bestimmen der Position im Netz um Gewicht_Neuron-FolgeNeuron zu
                    laden
246             // layer und neuron: Position von Neuron im Netz
247             // connectionsOfNeuron: Anzahl der Folgeneuronen von Neuron
248             // succ: x-tes FolgeNeuron von Neuron
249
250             // * Gewicht_Neuron-FolgeNeuron
251
252             //DEBUG
253             System.out.println("\t\t\tconnWeights["+ (offset+succ) + "]");
254
255             weightDelta *= connWeights[offset + succ];
256           }// for() Summe Ã¼ber Folgeneuronen
257
258           // * Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
259           // Ausgabe_u  = getOutput von aktuellem Neuron -> connection[neuron +
                  conn].getNeuronTo.getOutput
260           // Eingabevektor_u  = inputs[] von aktuellem Neuron -> getInputVector()
                  in Neuron implementieren
261           // Eingabevektor_u = netInput, da Verbindungsgewichte schon in inputs
                  integriert
262           outputOfNeuron = connection[neuron + conn].getNeuronTo.getOutput;
263           inputVectorOfNeuron = connection[neuron + conn].getNeuronTo.getNetInput;
264
265           weightDelta *= (outputOfNeuron * (1 - outputOfNeuron) *
                  inputVectorOfNeuron;
266
267           // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
268           weightDelta *= trainingCoefficient;
269
270           // Add calculated weight difference to connection weight
271           connections[neuron + conn].addWeightDelta(weightDelta);
272         }// for() Ã¼ber alle Verbindungen
273       }// for() Ã¼ber alle Neuronen
274     }// for() Ã¼ber alle (Verbindungs-)Layer
275   }// backpropagation()
```

```
1   /*
        ********************************************************************************

2   * TODO: Aufraeumen
3   *
4   *
        ---------------------------------------------------------------------------------

5   * Beispiel: 2 Input, 4 Hidden, 2 Hiddenlayer, 2 Output
6   *   Connection each, Oupuzt connection 2 Groups
7   *                 ------------------------------------------------------
8   *   inputConnection[] | hiddenConnection[][] | outputConnection[]
9   * 0 : 0 >     0 >     0
10  *         1
11  *         2
12  *         3
13  *   1 >     0
14  *         1
15  *         2
16  *         3
17  *   2 >     0
18  *         1 >     0
19  *         2
20  *         3
21  *   3 >     0
22  *         1
23  *         2
24  *         3
25  * 1 : 0 >     0
26  *         1
27  *         2 >     1
```

```
28   *           3
29   *    1 >      0
30   *           1
31   *           2
32   *           3
33   *    2 >      0
34   *           1
35   *           2
36   *           3 >      1
37   *    3 >      0
38   *           1
39   *           2
40   *           3
41   *                ---------------------------------------------------------------
42   *
     ------------------------------------------------------------------------------------------

43   *
44   * Backpropagation allgemein:
45   * deltaGewichtsvektor_u = Lernfaktor (Summe Ã¼ber Folgeneuronen((Summe Ã¼ber
       Ausgabeneuronen(
46   * (Erwartete Ausgabe - Ausgabe) dAusgabe_Ausgabeneuron nach
       dNetzeingabe_Folgeneuron))*Gewicht_Neuron-FolgeNeuron)
47   * ) dAusgabe_u nach dNetzeingabe_u * Eingabevektor_u (S.71)
48   *
49   * FÃ¼r Identiaet als Ausgabefunktion und logistischer Aktivierungsfunktion:
50   * deltaGewichtsvektor_u = Lernfaktor (Summe Ã¼ber Folgeneuronen((Summe Ã¼ber
       Ausgabeneuronen(
51   * (Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron)dAusgabe_Ausgabeneuron
        nach dNetzeingabe_Folgeneuron))*Gewicht_FolgeNeuron-u)
52   * ) Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
53
54   * -> SCHICHTENWEISE
55   -> Summe Ã¼ber Ausgabeneuronen:
56   * -> Summe Ã¼ber Folgeneuronen: Pfad im Netz muss mathematisch nachgebildet
       werden
57   * -> Eingabevektor muss angepasst werden, enthaelt Bitmuster
58   * -> bei logistischer Funktion:
59   * dAusgabe_Ausgabeneuron nach dNetzeingabe_Folgeneuron = Ausgabe_Ausgabeneuron (1
       - Ausgabe_Ausgabeneuron)
60   *
61   * Einzelner Gradientenabstieg:
62   * delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 -
       Ausgabe) * Eingabevektor_u
63   * -> pro inputneuron:
64   * delta Gewicht_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 - Ausgabe
       ) * Eingabe_u
65   *
66   * Um deltaGewicht fÃ¼r jede verbindung zu berechnen erst mit for-loop Ã¼ber
       outputconnection.length
67   * dann for (int i = hiddenLayer-2; i>=0; i++) -> loop Ã¼ber hiddenLayer[i].length
68   * dann for-loop Ã¼ber inputConnection.length
69   *
70   * Um Folgeneuronen zu ermitteln Position in Array benÃ¶tigt, dann loops Ã¼ber die
       folgende
71   * Netzstruktur (nicht einfach Ã¼ber die ConnectionArrays)
72   *
73   * Umwandlung des Schwellenwertes in ein Gewicht: Der Schwellenwert wird auf 0
       festgelegt, als Ausgleich wird ein zusaetzlicher (imaginaerer)
74   * Eingang (x_0) eingefuehrt, der den festen Wert 1 hat und mit dem negierten
       Schwellenwert gewichtet wird.(S.32)
75   ********************************************************************************
       */
76 private void backpropagation(float[] resultOut, float[] wantedOut){
77     // TODO: wofÃ¼r Variablen, Richtige Reihenfolge
78     float trainingCoefficient = 2;
79     float weightDelta = 0;
```

```
80      float succWeights[] = new float[1];
81      Connection connections[] = new Connection[1];
82      int succNeurons = 0;
83      int numberOfThresholds;
84      float outputOfNeuron;
85      float inputOfNeuron;
86      int position;
87
88      // Um deltaGewicht fÃ¼r jede verbindung zu berechnen erst loop Ã¼ber alle (
            Verbindungs-)Layer...
89      int connectionLayer = hiddenLayers - 1;
90
91      for (int layer = connectionLayer; layer >= 0; layer--){
92
93        // Outputconnection layer
94        if (layer == connectionLayer){
95          connections = new Connection [outputConnections.length];
96          connections = outputConnections;
97
98          numberOfThresholds = outputNeurons.length;
99        }
100
101       // InputConnection layer
102       else if (layer == 0){
103         if (connectionLayer > 1){
104           succWeights = new float [hiddenConnWeights[0].length];
105           succWeights = hiddenConnWeights[0];
106
107           // Number of inputs of neuron in 1st hidden layer = number of
108           // successive neurons
109           succNeurons = hiddenConnWeights[0].length/hiddenNeurons[0].length;
110         }
111         else {
112           succWeights = new float [outputConnWeights.length];
113           succWeights = outputConnWeights;
114
115           succNeurons = outputConnWeights.length/outputNeurons.length;
116         }
117
118         connections = new Connection [inputConnections.length];
119         connections = inputConnections;
120
121         numberOfThresholds = inputNeurons.length;
122       }
123
124       // Hidden Layer
125       else {
126         if (layer == connectionLayer - 1){
127           succWeights = new float [outputConnWeights.length];
128           succWeights = outputConnWeights;
129
130           // Nur eine Verbingung pro Neuron zum outputLayer
131           succNeurons = 1;
132         }
133
134         else {
135           succWeights = new float [hiddenConnWeights[layer + 1].length];
136           succWeights = hiddenConnWeights[layer + 1];
137
138           // To each neuron of the following layer
139           succNeurons = hiddenNeurons[layer + 1].length;
140         }
141
142         connections = new Connection [hiddenConnections[layer].length];
143         connections = hiddenConnections[layer];
144
145         numberOfThresholds = hiddenNeurons[layer].length;
146       }
147
148       // ...und nu Ã¼ber alle Verbindungen
```

```
149       for (int conn = 0; conn < (connections.length); conn++){
150
151         // Gradientenabstieg fÃ¼r outputLayer
152         if (layer == connectionLayer){
153           // delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe)
                Ausgabe (1 - Ausgabe) * Eingabevektor_u
154           // -> delta Gewichtsvektor_u = Lernfaktor Ausgabe (1 - Ausgabe) *
                Eingabevektor_u auch in backpropagation (hinter for-loops)
155           // -> nur (Erwartete Ausgabe - Ausgabe) benÃ¶tigt
156           position = connections[conn].getPositionNeuronTo();
157
158           weightDelta = wantedOut[position] - resultOut[position];
159         }
160
161         // Da real backpropagation
162         else {
163           // Beginn der Berechnung
164           // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
165           // weightDelta = trainingCoefficient *
166
167           // Summe Ã¼ber Folgeneuronen
168           for (int succ = 0; succ < succNeurons; succ++){
169             int sumOutputNeurons = 0;
170
171             // Summe Ã¼ber Ausgabeneuronen
172             for (int out = 0; out < outputNeurons.length; out++){
173               sumOutputNeurons += (
174               // Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron
175               (wantedOut[out] - resultOut[out]) *
176               //dAusgabe nach dNetzeingabe
177               resultOut[out] * (1- resultOut[out]));
178             }// for() Summe Ã¼ber Ausgabeneuronen
179
180             weightDelta += sumOutputNeurons;
181
182             // * Gewicht_Neuron-FolgeNeuron
183             position = connections[conn].getPositionNeuronTo();
184
185             weightDelta *= succWeights[position];
186           }// for() Summe Ã¼ber Folgeneuronen
187         }
188
189         // * Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
190         // Ausgabe_u  = getOutput von aktuellem Neuron -> connection[neuron + conn
                ].getNeuronTo.getOutput
191         // Eingabevektor_u  = inputs[] von aktuellem Neuron -> getInputVector() in
                Neuron implementieren
192         outputOfNeuron = connections[conn].getNeuronTo().getOutput();
193         inputOfNeuron = connections[conn].getNeuronFrom().getOutput();
194
195         weightDelta *= (outputOfNeuron * (1 - outputOfNeuron) * inputOfNeuron);
196
197         // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
198         weightDelta *= trainingCoefficient;
199
200         // Add calculated weight difference to connection weight
201         connections[conn].addWeightDelta(weightDelta);
202       }// for() Ã¼ber alle Verbindungen
203
204       // for Ã¼ber die Thresholds
205       for (int thresh = 0; thresh < numberOfThresholds; thresh++){
206         // TODO: Der Schwellenwert wird auf 0 festgelegt??? noch notwendig?
207         // , als Ausgleich wird ein zusaetzlicher (imaginaerer)
208         // Eingang (x_0) eingefÃ¼hrt, der den festen Wert 1 hat und mit dem
                negierten Schwellenwert gewichtet wird.(S.32)
209         // Position in Layer = threshold * Anzahl der Verbindungen pro Neuron ->
                Anzahl der Verbinungen in Layer / Anzahl der Neuronen (Thresholds)
```

```
210         int positionInLayer = thresh * (connections.length / numberOfThresholds);
211         float oldThresh = -1 * connections[positionInLayer].getNeuronTo().
                getThreshold();
212
213         // Gradientenabstieg fÃ¼r outputLayer
214         if (layer == connectionLayer){
215           // delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe)
                  Ausgabe (1 - Ausgabe) * Eingabevektor_u
216           // -> delta Gewichtsvektor_u = Lernfaktor Ausgabe (1 - Ausgabe) *
                  Eingabevektor_u auch in backpropagation (hinter for-loops)
217           // -> nur (Erwartete Ausgabe - Ausgabe) benÃ¶tigt
218           // Erwartete Ausgabe - Ausgabe
219           // Note: Anzahl thresholds entspricht Anzahl, und somit Position, von
                  OutputNeuron
220           weightDelta = wantedOut[thresh] - resultOut[thresh];
221         }
222
223         // Da real backpropagation
224         else {
225           // Beginn der Berechnung
226           // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
227           // weightDelta = trainingCoefficient *
228
229           // Summe Ã¼ber Folgeneuronen
230           for (int succ = 0; succ < succNeurons; succ++){
231             int sumOutputNeurons = 0;
232
233             // Summe Ã¼ber Ausgabeneuronen
234             for (int out = 0; out < outputNeurons.length; out++){
235               sumOutputNeurons += (
236               // Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron
237               (wantedOut[out] - resultOut[out]) *
238               //dAusgabe nach dNetzeingabe
239               resultOut[out] * (1- resultOut[out]));
240             }// for() Summe Ã¼ber Ausgabeneuronen
241
242             weightDelta += sumOutputNeurons;
243
244             // * Gewicht_Neuron-FolgeNeuron
245             weightDelta *= succWeights[succ];
246           }// for() Summe Ã¼ber Folgeneuronen
247         }
248
249         // * Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
250         // Ausgabe_u  = getOutput von aktuellem Neuron -> connection[neuron + conn
                ].getNeuronTo.getOutput
251         // Eingabevektor_u  = inputs[] von aktuellem Neuron -> getInputVector() in
                Neuron implementieren
252         outputOfNeuron = connections[positionInLayer].getNeuronTo().getOutput();
253         // inputVectorOfNeuron nicht benÃ¶tigt, da nur = 1;
254
255         weightDelta *= (outputOfNeuron * (1 - outputOfNeuron));
256
257         // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
258         weightDelta *= trainingCoefficient;
259
260         // Add calculated weight difference to old threshold and set new threshold
261         connections[positionInLayer].getNeuronTo().setThreshold(oldThresh +
                weightDelta);
262       }// for() Ã¼ber thresholds
263     }// for() Ã¼ber alle (Verbindungs-)Layer
264   }// backpropagation()
```

Chapter 5

Prospective Enhancements

5.1 Execution of multiLayerPerceptron using integer

```
1  /*
       ********************************************************************************
2  * Executes the built multi-layer perceptron with given input vector
3  * Returns the output vector as single integer
4  ********************************************************************************
       */
5  int runInt(int inputVector){
6    int output = 0;
7    float networkInputVector[] = new float[inputNeuron.length];
8    float networkOutputVector[] = new float[outputVector.length];
9
10   //Set input if corresponding bit is high, unset if low
11   for (int i = 0; i < inputNeuron.length; i++){
12
13     if ((inputVector & (1 << i)) == (1 << i))
14       networkInputVector[i] = 1;
15     else
16       networkInputVector[i] = 0;
17   }
18
19   networkOutputVector = run(networkInputVector);
20
21   for (int i = 0; i < networkOutputVector.length || i < 32; i++){
22     if(networkOutputVector[i] >= 0.5f)
23       output |= (1 << i);
24   }
25
26   return output;
27 }// runInt()
```

5.2 Execution of multiLayerPerceptron using integer, defined threshold

```
1  /*
       ********************************************************************************
2  * Executes the built multi-layer perceptron with given input vector
3  * Returns the output vector as single integer
4  * Bit is set if depending output value is > threshold
5  ********************************************************************************
       */
6  int runInt(int inputVector, float threshold){
7    int output = 0;
8    float networkOutputVector[] = new float[outputVector.length];
9
```

```
10      networkOutputVector = run(inputVector);
11
12      for (int i = 0; i < networkOutputVector.length || i < 32; i++){
13        if(networkOutputVector[i] >= threshold)
14          output |= (1 << i);
15      }
16
17      return output;
18    }// runInt()
```

5.3 Training for execution using integer

```
1 Boolean training(int trainigInVector, int trainingOutVector, float errorTolerance)
```

5.4 Topology rising

```
1 case "rising":
2 // Number of hidden neurons is increasing with every layer.
3 // Start with hiddenNeuronsPerLayer, increasing with + 2 until hiddenLayers
4 hiddenConnWeights = new float[hiddenLayers-1][hiddenNeuronsPerLayer + (2 * (
      hiddenLayers-1))];
5
6 for (int i = 0; i < hiddenLayers-1; i++){
7
8   for (int j = 0; j < hiddenConnWeights[i].length; j++)
9     hiddenConnWeights[i][j] = 1;
10 }
11
12 neededNeuronInputs = 2;
13 break;
```

5.5 Topology diamond

```
1 //TODO: case "diamond"
2 // rising until hiddenLayers/2, then declining
```

5.6 Topology with different numbers of hidden neurons per layer

```
1 /*
      ********************************************************************************
2 * Multi-layer perceptron with different numbers of hidden neurons per layer
3 * int[] hiddenNeuronsPerLayer, - int hiddenLayers,
4 * hiddenLayers = hiddenNeuronsPerLayer.length;
5 ********************************************************************************
      */
```

5.7 Up to 64 inputs/outputs

For function int runInt()

5.8 Activation function and Output function settable

(Into extra classes so class Neuron remains as small as possible)
(siehe ComputaionalIntelligence S.52)
Step function: if (netInput ¿= threshold) return netInput; else return 0;
semi-lineare Funktion: if (netInput ¿ threshold + 1/2) return netInput;
else if ((netInput ¡ threshold + 1/2) && (netInput ¿ threshold - 1/2)) output = (netInput

- threshold) + 1/2;
else return 0;
Sinus bis Saettigung: if (netInput ¿ threshold + pi/2) return netInput;
else if ((netInput ¡ threshold + pi/2) && (netInput ¿ threshold - pi/2)) output = (sin(netInput
- threshold) + 1)/2;
else return 0;
logistische Funktion: output = 1 / (1 + $e^{(-(netInput-threshold))}$) // geht nur von 0 - 1
radiale Basis Funktionen
enum fÃ¼r Funktionen in Klasse Ã¼ber Neuron

Chapter 6

Test results

6.1 Unit test for NeuronFloat

Chapter 7

Discarded

```
/*
    ********************************************************************************************
 * class ThresholdItem:
 * up to 32 inputs possible
 * inputs are just 0 or 1
 * output = input.1*weights[0] + ... + input.32*weights[31]
 * So far, output = netInput
 * activated just returns true if output >= threshold
 * Note! So far, activated can just be called if getOutput was called before
 ********************************************************************************************
    */
class ThresholdItem
{
protected
  int inputs;
  int nrInputs;
  // So far also used as output
  float netInput;
  float weights[];
  float threshold;
  // Used as output
  Boolean activated;

public
  // Constructors
  ThresholdItem(){};

  ThresholdItem(int nrInputs){
    this.nrInputs = nrInputs;
    weights = new float[nrInputs];

    // Initialize inputs, output and weights to 0, threshold to max
    inputs = 0;
    netInput = 0;
    activated = false;
    threshold = 0x7fffffff;

    for (int i = 0; i < nrInputs; i++)
      weights[i] = 0;
  }

  Boolean setInput(int nrInput){
    if(nrInput < nrInputs){
      inputs |= 1 << nrInput;

      return true;
    }

    return false;
  }
```

```
49
50   Boolean unsetInput(int nrInput){
51     if(nrInput < nrInputs){
52       inputs &= ~(1 << nrInput);
53
54       return true;
55     }
56
57     return false;
58   }
59
60   Boolean setWeight(int nrInput, float weight){
61     if(nrInput < weights.length){
62       weights[nrInput] = weight;
63       return true;
64     }
65
66     return false;
67   }
68
69   void setThreshold(float threshold){
70     this.threshold = threshold;
71   }
72
73   float getOutput(){
74     // Reset net input
75     netInput = 0;
76
77     // Net input function f_net
78     for (int i = 0; i < nrInputs; i++){
79       if((inputs & (1 << i)) == (1 << i))
80         netInput += weights[i];
81     }
82
83     // Activation function f_act: step function
84     if (netInput >= threshold){
85       activated = true;
86
87       // Output function f_out: Identity
88       return netInput;
89     }
90     else {
91       activated = false;
92       return 0;
93     }
94   }
95
96   Boolean getActivation(){
97     return activated;
98   }
99 }// class ThresholdItem
```