
Contents

| | |
|--|----------|
| Contents | 1 |
| 1 TODOs | 1 |
| 1.1 TechDoc | 1 |
| 1.2 General | 1 |
| 1.3 class MultiLayerPerceptron | 1 |
| 1.4 Tests | 2 |
| 2 class Neuron | 3 |
| 2.1 Attributes | 3 |
| 2.1.1 int inputs | 3 |
| 2.1.2 int nrInputs | 3 |
| 2.1.3 float netInput | 3 |
| 2.1.4 float weights[] | 3 |
| 2.1.5 float threshold | 3 |
| 2.2 Constructors | 3 |
| 2.2.1 Neuron(int nrInputs) | 3 |
| 2.3 Methods | 4 |
| 2.3.1 Boolean setInput(int nrInput) | 4 |
| 2.3.2 Boolean unsetInput(int nrInput) | 4 |
| 2.3.3 Boolean setWeight(int nrInput, float weight) | 4 |
| 2.3.4 void setThreshold(float threshold) | 4 |
| 2.3.5 float getOutput() | 4 |
| 3 class Connection | 5 |
| 3.1 Attributes | 5 |
| 3.1.1 Neuron neuronFrom | 5 |
| 3.1.2 Neuron neuronTo | 5 |
| 3.1.3 int input | 5 |
| 3.1.4 float connWeight | 5 |
| 3.1.5 float weightToSet | 5 |
| 3.2 Constructors | 5 |
| 3.2.1 Connection(Neuron neuronFrom, Neuron neuronTo, int input, float connWeight) | 5 |
| 3.3 Methods | 5 |
| 3.3.1 void run() | 5 |

| | | |
|----------|---|-----------|
| 4 | class MultiLayerPerceptron | 7 |
| 4.1 | Attributes | 7 |
| 4.1.1 | Neuron inputNeuron[] | 7 |
| 4.1.2 | Neuron hiddenNeuron[][] | 7 |
| 4.1.3 | Neuron outputNeuron[] | 7 |
| 4.1.4 | int hiddenLayers | 7 |
| 4.1.5 | float inputConnWeights[] | 7 |
| 4.1.6 | float hiddenConnWeights[][] | 7 |
| 4.1.7 | float outputConnWeights[] | 7 |
| 4.1.8 | float outputVector[] | 8 |
| 4.1.9 | Connection inputConnection[] | 8 |
| 4.1.10 | Connection hiddenConnection[][] | 8 |
| 4.1.11 | Connection outputConnection[] | 8 |
| 4.2 | Constructors | 8 |
| 4.2.1 | MultiLayerPerceptron(int inputNeurons, int hiddenNeuronsPerLayer, int hiddenLayers, int outputNeurons) | 8 |
| 4.3 | Methods | 8 |
| 4.3.1 | float[] run(int inputVector) | 8 |
| 4.3.2 | void backpropagation(float[] resultOut, float[] wantedOut) | 9 |
| 5 | Prospective Enhancements | 18 |
| 5.1 | Up to 64 inputs/outputs | 18 |
| 5.2 | Activation function and Output function settable | 18 |
| 6 | Test results | 19 |
| 6.1 | Unit test for NeuronFloat | 19 |
| 7 | Discarded | 20 |

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 Konstruktoren (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:

$$netInput = input.1 * weights[0] + \dots + input.32 * weights[31]$$

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 initializes `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 too 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 too 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 too 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. Then 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, Ouputz 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  $\tilde{A}_{\frac{1}{4}}$ ber FolgeNeuronen ((Summe  $\tilde{A}_{\frac{1}{4}}$ ber
   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 *  $F\tilde{A}_{\frac{1}{4}}$  Identiaet als Ausgabefunktion und logistischer Aktivierungsfunktion:
51 * deltaGewichtsvektor_u = Lernfaktor (Summe  $\tilde{A}_{\frac{1}{4}}$ ber FolgeNeuronen ((Summe  $\tilde{A}_{\frac{1}{4}}$ ber
   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  $\tilde{A}_{\frac{1}{4}}$ ber Ausgabeneuronen:

```

```

57  * -> Summe  $\tilde{A}_{\frac{1}{4}}$ ber 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  $\tilde{f\tilde{A}}_{\frac{1}{4}}r$  jede verbindung zu berechnen erst mit for-loop  $\tilde{A}_{\frac{1}{4}}$ ber
    outputconnection.length
68  * dann for (int i = hiddenLayer-2; i>=0; i++) -> loop  $\tilde{A}_{\frac{1}{4}}$ ber hiddenLayer[i].length
69  * dann for-loop  $\tilde{A}_{\frac{1}{4}}$ ber inputConnection.length
70  *
71  * Um Folgeneuronen zu ermitteln Position in Array ben $\tilde{A}\P$ tigt, dann loops  $\tilde{A}_{\frac{1}{4}}$ ber die
    folgende
72  * Netzstruktur (nicht einfach  $\tilde{A}_{\frac{1}{4}}$ ber die ConnectionArrays)
73  *****/
74  private void backpropagation(float[] resultOut, float[] wantedOut){
75      // TODO: Kommentare wof $\tilde{A}_{\frac{1}{4}}r$  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  $\tilde{f\tilde{A}}_{\frac{1}{4}}r$  jede verbindung zu berechnen loop  $\tilde{A}_{\frac{1}{4}}$ ber alle (
        Verbindungs-)Layer...
88      for (int layer = hiddenLayers; layer >= 0; layer--){
89          // Bestimmen der Position des Neurons im Netz und der folgenden
90          // Struktur  $\tilde{f\tilde{A}}_{\frac{1}{4}}r$  die Summe  $\tilde{A}_{\frac{1}{4}}$ ber die Folgeneuronen,
91          // d.h.  $\tilde{f\tilde{A}}_{\frac{1}{4}}r$  die Grenze der for-Schleife. Daher nur int neuronsInLayer
            ben $\tilde{A}\P$ tigt
92          // -> mit layer ermitteln in welchem Layer: output, hidden o input
93          // -> mit neuron an welcher Stelle im Layer
94          // Grenze  $\tilde{f\tilde{A}}_{\frac{1}{4}}r$  Summe  $\tilde{A}_{\frac{1}{4}}$ ber 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, ben $\tilde{A}\P$ tigt allgemein  $\tilde{g\tilde{A}}_{\frac{1}{4}}$ ltigen
                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
121         ++, inp++){
122
123         // Go to next output neuron if all inputs are connected
124         if ((hidNeur % (hiddenNeuron[hiddenLayers-1].length/outputNeuron.length))
125             == 0){
126             outNeur++;
127             inp = 0;
128         }
129
130         // ...to the "nearest" output neuron (if several are available)
131         outputConnection[hidNeur] = new Connection(hiddenNeuron[hiddenLayers-1][
132             hidNeur],
133             outputNeuron[outNeur], inp, outputConnWeights[hidNeur]);
134     }
135     *****/
136
137     // TODO: Auch hier allgemeine Formulierungen
138     connWeights = new float [outputConnWeights.length];
139     connWeights = outputConnWeights;
140
141     connections = new Connection [outputConnection.length];
142     connections = outputConnection;
143 }
144
145 // InputConnection layer
146 else if (layer == 0){
147     neuronsInLayer = inputNeuron.length;
148     neuronsInNextLayer = hiddenNeuron[0].length;
149
150     // TODO: Allgemein  $g\tilde{A}^{\frac{1}{4}}$ ltiger Ausdruck
151     // Im Moment nur von jeden inputNeuron zu jedem hiddenNeuron
152     connectionsOfNeuron = hiddenNeuron[0].length;
153
154     /******
155     // Connections between input layer and 1st hidden layer
156     int positionInLayer = 0;
157
158     // From each input neuron...
159     for (int inp = 0; inp < inputNeuron.length; inp++){
160
161         // ...to each neuron of the 1st hidden layer
162         for (int conn = 0; conn < hiddenNeuron[0].length; conn++){
163
164             inputConnection[positionInLayer] = new Connection(inputNeuron[inp],
165                 hiddenNeuron[0][conn], inp, inputConnWeights[positionInLayer]);
166             positionInLayer++;
167         }
168     }
169     *****/
170
171     connWeights = new float [inputConnWeights.length];
172     connWeights = inputConnWeights;
173
174     connections = new Connection [inputConnection.length];
175     connections = inputConnection;
176
177     /******
178     // Connection weights
179     inputConnWeights = new float [inputNeurons * hiddenNeuronsPerLayer];

```

```

179     hiddenConnWeights = new float [hiddenLayers - 1][hiddenNeuronsPerLayer *
180         hiddenNeuronsPerLayer];
181
182     outputConnWeights = new float [hiddenNeuronsPerLayer];
183
184     Struktur siehe Connections weiter oben
185     *****
186 }
187
188 // Hidden Layer
189 else {
190     neuronsInLayer = hiddenNeuron[layer - 1].length;
191     neuronsInNextLayer = hiddenNeuron[layer].length;
192
193     // TODO: Allgemein  $g_{\tilde{A}_4^1}$  tiger Ausdruck
194     // Im Moment nur von jeden hiddenNeuron zu jedem hiddenNeuron im Folgelayer
195     connectionsOfNeuron = hiddenNeuron[layer].length;
196
197     connWeights = new float [hiddenConnWeights[layer - 1].length];
198     connWeights = hiddenConnWeights[layer - 1];
199
200     connections = new Connection [hiddenConnection[layer - 1].length];
201     connections = hiddenConnection[layer - 1];
202 }
203 //DEBUG
204 System.out.println("\tconnWeights.lenght: " + connWeights.length);
205
206 // ...,  $\tilde{A}_4^1$ ber alle Neuronen in dem Layer...
207 for (int neuron = 0; neuron < neuronsInLayer; neuron++){
208     //DEBUG
209     System.out.println("\tfor (int neuron = 0; neuron < "+neuronsInLayer+"
210         neuron: "+neuron+"}");
211
212     // Position of 1st connection of this neuron (Is the same
213     // for all neurons in this layer)
214     offset = neuron * connectionsOfNeuron;
215
216     // ...und nu  $\tilde{A}_4^1$ ber alle Verbindungen des Neurons
217     for (int conn = 0; conn < connectionsOfNeuron; conn++){
218         //DEBUG
219         System.out.println("\t\tfor (int conn = 0; conn < "+connectionsOfNeuron+"
220             ; conn" +conn+ "}");
221
222         *** Notwendig?
223         // pro Inputneuron:
224         // Notwendig? wie sonst inputVector[inp] verwenden?
225         for (int inp = 0; inp < inputNeuron.length; inp++){
226             }// for() pro Inputneuron
227         ***
228
229         // Beginn der Berechnung
230         // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
231         // weightDelta = trainingCoefficient *
232
233         // DEBUG
234         System.out.println("\t\tneuronsInNextLayer: " + neuronsInNextLayer);
235
236         // Summe  $\tilde{A}_4^1$ ber Folgeuronen
237         for (int succ = 0; succ < neuronsInNextLayer; succ++){
238
239             // Summe  $\tilde{A}_4^1$ ber Ausgabeneuronen
240             for (int out = 0; out < outputNeuron.length; out++){
241                 weightDelta += (
242                     // Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron
243                     (wantedOut[out] - resultOut[out]) *
244                     //dAusgabe nach dNetzeingabe
245                     resultOut[out] * (1- resultOut[out]));
246             }// for() Summe  $\tilde{A}_4^1$ ber 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  $\tilde{A}_{\frac{1}{4}}$ ber 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()  $\tilde{A}_{\frac{1}{4}}$ ber alle Verbindungen
273 } // for()  $\tilde{A}_{\frac{1}{4}}$ ber alle Neuronen
274 } // for()  $\tilde{A}_{\frac{1}{4}}$ ber alle (Verbindungs-)Layer
275 } // backpropagation()

1  /*
           *****

2  * TODO: Aufräumen
3  *
4  *

5  * Beispiel: 2 Input, 4 Hidden, 2 Hiddenlayer, 2 Output
6  * Connection each, Output connection 2 Groups
7  *
8  * inputConnection[] | hiddenConnection[][] | outputConnection[]
9  * 0 : 0 > 0 > 0
10 *
11 * 1
12 * 2
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  $\tilde{A}_{\frac{1}{4}}$ ber Folgeneuronen ((Summe  $\tilde{A}_{\frac{1}{4}}$ ber
46 * Ausgabeneuronen(
47 * (Erwartete Ausgabe - Ausgabe) dAusgabe_Ausgabeneuron nach
48 * dNetzeingabe_Folgeneuron))*Gewicht_Neuron-FolgeNeuron)
49 * ) dAusgabe_u nach dNetzeingabe_u * Eingabevektor_u (S.71)
50 *
51 *  $\tilde{f}'_{\frac{1}{4}}$  Identiaet als Ausgabefunktion und logistischer Aktivierungsfunktion:
52 * deltaGewichtsvektor_u = Lernfaktor (Summe  $\tilde{A}_{\frac{1}{4}}$ ber Folgeneuronen ((Summe  $\tilde{A}_{\frac{1}{4}}$ ber
53 * Ausgabeneuronen(
54 * (Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron)dAusgabe_Ausgabeneuron
55 * nach dNetzeingabe_Folgeneuron))*Gewicht_FolgeNeuron-u)
56 * ) Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
57 *
58 *  $\rightarrow$  SCHICHTENWEISE
59 *  $\rightarrow$  Summe  $\tilde{A}_{\frac{1}{4}}$ ber Ausgabeneuronen:
60 *  $\rightarrow$  Summe  $\tilde{A}_{\frac{1}{4}}$ ber Folgeneuronen: Pfad im Netz muss mathematisch nachgebildet
61 * werden
62 *  $\rightarrow$  Eingabevektor muss angepasst werden, enthaelt Bitmuster
63 *  $\rightarrow$  bei logistischer Funktion:
64 * dAusgabe_Ausgabeneuron nach dNetzeingabe_Folgeneuron = Ausgabe_Ausgabeneuron (1
65 * - Ausgabe_Ausgabeneuron)
66 *
67 * Einzelner Gradientenabstieg:
68 * delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 -
69 * Ausgabe) * Eingabevektor_u
70 *  $\rightarrow$  pro inputneuron:
71 * delta Gewicht_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 - Ausgabe
72 * ) * Eingabe_u
73 *
74 * Um deltaGewicht  $\tilde{f}'_{\frac{1}{4}}$  jede verbindung zu berechnen erst mit for-loop  $\tilde{A}_{\frac{1}{4}}$ ber
75 * outputconnection.length
76 * dann for (int i = hiddenLayer-2; i>=0; i++)  $\rightarrow$  loop  $\tilde{A}_{\frac{1}{4}}$ ber hiddenLayer[i].length
77 * dann for-loop  $\tilde{A}_{\frac{1}{4}}$ ber inputConnection.length
78 *
79 * Um Folgeneuronen zu ermitteln Position in Array benoetigt, dann loops  $\tilde{A}_{\frac{1}{4}}$ ber die
80 * folgende
81 * Netzstruktur (nicht einfach  $\tilde{A}_{\frac{1}{4}}$ ber die ConnectionArrays)
82 *
83 * Umwandlung des Schwellenwertes in ein Gewicht: Der Schwellenwert wird auf 0
84 * festgelegt, als Ausgleich wird ein zusaetzlicher (imaginaerer)
85 * Eingang ( $x_0$ ) eingefuehrt, der den festen Wert 1 hat und mit dem negierten
86 * Schwellenwert gewichtet wird.(S.32)
87 *
88 *
89 *
90 *
91 *
92 *
93 *
94 *
95 *
96 *
97 *
98 *
99 *
100 *
101 *
102 *
103 *
104 *
105 *
106 *
107 *
108 *
109 *
110 *
111 *
112 *
113 *
114 *
115 *
116 *
117 *
118 *
119 *
120 *
121 *
122 *
123 *
124 *
125 *
126 *
127 *
128 *
129 *
130 *
131 *
132 *
133 *
134 *
135 *
136 *
137 *
138 *
139 *
140 *
141 *
142 *
143 *
144 *
145 *
146 *
147 *
148 *
149 *
150 *
151 *
152 *
153 *
154 *
155 *
156 *
157 *
158 *
159 *
160 *
161 *
162 *
163 *
164 *
165 *
166 *
167 *
168 *
169 *
170 *
171 *
172 *
173 *
174 *
175 *
176 *
177 *
178 *
179 *
180 *
181 *
182 *
183 *
184 *
185 *
186 *
187 *
188 *
189 *
190 *
191 *
192 *
193 *
194 *
195 *
196 *
197 *
198 *
199 *
200 *

```

```

201 *
202 *
203 *
204 *
205 *
206 *
207 *
208 *
209 *
210 *
211 *
212 *
213 *
214 *
215 *
216 *
217 *
218 *
219 *
220 *
221 *
222 *
223 *
224 *
225 *
226 *
227 *
228 *
229 *
230 *
231 *
232 *
233 *
234 *
235 *
236 *
237 *
238 *
239 *
240 *
241 *
242 *
243 *
244 *
245 *
246 *
247 *
248 *
249 *
250 *
251 *
252 *
253 *
254 *
255 *
256 *
257 *
258 *
259 *
260 *
261 *
262 *
263 *
264 *
265 *
266 *
267 *
268 *
269 *
270 *
271 *
272 *
273 *
274 *
275 *
276 *
277 *
278 *
279 *
280 *
281 *
282 *
283 *
284 *
285 *
286 *
287 *
288 *
289 *
290 *
291 *
292 *
293 *
294 *
295 *
296 *
297 *
298 *
299 *
300 *
301 *
302 *
303 *
304 *
305 *
306 *
307 *
308 *
309 *
310 *
311 *
312 *
313 *
314 *
315 *
316 *
317 *
318 *
319 *
320 *
321 *
322 *
323 *
324 *
325 *
326 *
327 *
328 *
329 *
330 *
331 *
332 *
333 *
334 *
335 *
336 *
337 *
338 *
339 *
340 *
341 *
342 *
343 *
344 *
345 *
346 *
347 *
348 *
349 *
350 *
351 *
352 *
353 *
354 *
355 *
356 *
357 *
358 *
359 *
360 *
361 *
362 *
363 *
364 *
365 *
366 *
367 *
368 *
369 *
370 *
371 *
372 *
373 *
374 *
375 *
376 *
377 *
378 *
379 *
380 *
381 *
382 *
383 *
384 *
385 *
386 *
387 *
388 *
389 *
390 *
391 *
392 *
393 *
394 *
395 *
396 *
397 *
398 *
399 *
400 *
401 *
402 *
403 *
404 *
405 *
406 *
407 *
408 *
409 *
410 *
411 *
412 *
413 *
414 *
415 *
416 *
417 *
418 *
419 *
420 *
421 *
422 *
423 *
424 *
425 *
426 *
427 *
428 *
429 *
430 *
431 *
432 *
433 *
434 *
435 *
436 *
437 *
438 *
439 *
440 *
441 *
442 *
443 *
444 *
445 *
446 *
447 *
448 *
449 *
450 *
451 *
452 *
453 *
454 *
455 *
456 *
457 *
458 *
459 *
460 *
461 *
462 *
463 *
464 *
465 *
466 *
467 *
468 *
469 *
470 *
471 *
472 *
473 *
474 *
475 *
476 *
477 *
478 *
479 *
480 *
481 *
482 *
483 *
484 *
485 *
486 *
487 *
488 *
489 *
490 *
491 *
492 *
493 *
494 *
495 *
496 *
497 *
498 *
499 *
500 *
501 *
502 *
503 *
504 *
505 *
506 *
507 *
508 *
509 *
510 *
511 *
512 *
513 *
514 *
515 *
516 *
517 *
518 *
519 *
520 *
521 *
522 *
523 *
524 *
525 *
526 *
527 *
528 *
529 *
530 *
531 *
532 *
533 *
534 *
535 *
536 *
537 *
538 *
539 *
540 *
541 *
542 *
543 *
544 *
545 *
546 *
547 *
548 *
549 *
550 *
551 *
552 *
553 *
554 *
555 *
556 *
557 *
558 *
559 *
560 *
561 *
562 *
563 *
564 *
565 *
566 *
567 *
568 *
569 *
570 *
571 *
572 *
573 *
574 *
575 *
576 *
577 *
578 *
579 *
580 *
581 *
582 *
583 *
584 *
585 *
586 *
587 *
588 *
589 *
590 *
591 *
592 *
593 *
594 *
595 *
596 *
597 *
598 *
599 *
600 *
601 *
602 *
603 *
604 *
605 *
606 *
607 *
608 *
609 *
610 *
611 *
612 *
613 *
614 *
615 *
616 *
617 *
618 *
619 *
620 *
621 *
622 *
623 *
624 *
625 *
626 *
627 *
628 *
629 *
630 *
631 *
632 *
633 *
634 *
635 *
636 *
637 *
638 *
639 *
640 *
641 *
642 *
643 *
644 *
645 *
646 *
647 *
648 *
649 *
650 *
651 *
652 *
653 *
654 *
655 *
656 *
657 *
658 *
659 *
660 *
661 *
662 *
663 *
664 *
665 *
666 *
667 *
668 *
669 *
670 *
671 *
672 *
673 *
674 *
675 *
676 *
677 *
678 *
679 *
680 *
681 *
682 *
683 *
684 *
685 *
686 *
687 *
688 *
689 *
690 *
691 *
692 *
693 *
694 *
695 *
696 *
697 *
698 *
699 *
700 *
701 *
702 *
703 *
704 *
705 *
706 *
707 *
708 *
709 *
710 *
711 *
712 *
713 *
714 *
715 *
716 *
717 *
718 *
719 *
720 *
721 *
722 *
723 *
724 *
725 *
726 *
727 *
728 *
729 *
730 *
731 *
732 *
733 *
734 *
735 *
736 *
737 *
738 *
739 *
740 *
741 *
742 *
743 *
744 *
745 *
746 *
747 *
748 *
749 *
750 *
751 *
752 *
753 *
754 *
755 *
756 *
757 *
758 *
759 *
760 *
761 *
762 *
763 *
764 *
765 *
766 *
767 *
768 *
769 *
770 *
771 *
772 *
773 *
774 *
775 *
776 *
777 *
778 *
779 *
780 *
781 *
782 *
783 *
784 *
785 *
786 *
787 *
788 *
789 *
790 *
791 *
792 *
793 *
794 *
795 *
796 *
797 *
798 *
799 *
800 *
801 *
802 *
803 *
804 *
805 *
806 *
807 *
808 *
809 *
810 *
811 *
812 *
813 *
814 *
815 *
816 *
817 *
818 *
819 *
820 *
821 *
822 *
823 *
824 *
825 *
826 *
827 *
828 *
829 *
830 *
831 *
832 *
833 *
834 *
835 *
836 *
837 *
838 *
839 *
840 *
841 *
842 *
843 *
844 *
845 *
846 *
847 *
848 *
849 *
850 *
851 *
852 *
853 *
854 *
855 *
856 *
857 *
858 *
859 *
860 *
861 *
862 *
863 *
864 *
865 *
866 *
867 *
868 *
869 *
870 *
871 *
872 *
873 *
874 *
875 *
876 *
877 *
878 *
879 *
880 *
881 *
882 *
883 *
884 *
885 *
886 *
887 *
888 *
889 *
890 *
891 *
892 *
893 *
894 *
895 *
896 *
897 *
898 *
899 *
900 *
901 *
902 *
903 *
904 *
905 *
906 *
907 *
908 *
909 *
910 *
911 *
912 *
913 *
914 *
915 *
916 *
917 *
918 *
919 *
920 *
921 *
922 *
923 *
924 *
925 *
926 *
927 *
928 *
929 *
930 *
931 *
932 *
933 *
934 *
935 *
936 *
937 *
938 *
939 *
940 *
941 *
942 *
943 *
944 *
945 *
946 *
947 *
948 *
949 *
950 *
951 *
952 *
953 *
954 *
955 *
956 *
957 *
958 *
959 *
960 *
961 *
962 *
963 *
964 *
965 *
966 *
967 *
968 *
969 *
970 *
971 *
972 *
973 *
974 *
975 *
976 *
977 *
978 *
979 *
980 *
981 *
982 *
983 *
984 *
985 *
986 *
987 *
988 *
989 *
990 *
991 *
992 *
993 *
994 *
995 *
996 *
997 *
998 *
999 *
1000 *

```

```

1001 *
1002 *
1003 *
1004 *
1005 *
1006 *
1007 *
1008 *
1009 *
1010 *
1011 *
1012 *
1013 *
1014 *
1015 *
1016 *
1017 *
1018 *
1019 *
1020 *
1021 *
1022 *
1023 *
1024 *
1025 *
1026 *
1027 *
1028 *
1029 *
1030 *
1031 *
1032 *
1033 *
1034 *
1035 *
1036 *
1037 *
1038 *
1039 *
1040 *
1041 *
1042 *
1043 *
1044 *
1045 *
1046 *
1047 *
1048 *
1049 *
1050 *
1051 *
1052 *
1053 *
1054 *
1055 *
1056 *
1057 *
1058 *
1059 *
1060 *
1061 *
1062 *
1063 *
1064 *
1065 *
1066 *
1067 *
1068 *
1069 *
1070 *
1071 *
1072 *
1073 *
1074 *
1075 *
1076 *
1077 *
1078 *
1079 *
1080 *
1081 *
1082 *
1083 *
1084 *
1085 *
1086 *
1087 *
1088 *
1089 *
1090 *
1091 *
1092 *
1093 *
1094 *
1095 *
1096 *
1097 *
1098 *
1099 *
1100 *
1101 *
1102 *
1103 *
1104 *
1105 *
1106 *
1107 *
1108 *
1109 *
1110 *
1111 *
1112 *
1113 *
1114 *
1115 *
1116 *
1117 *
1118 *
1119 *
1120 *
1121 *
1122 *
1123 *
1124 *
1125 *
1126 *
1127 *
1128 *
1129 *
1130 *
1131 *
1132 *
1133 *
1134 *
1135 *
1136 *
1137 *
1138 *
1139 *
1140 *
1141 *
1142 *
1143 *
1144 *
1145 *
1146 *
1147 *
1148 *
1149 *
1150 *
1151 *
1152 *
1153 *
1154 *
1155 *
1156 *
1157 *
1158 *
1159 *
1160 *
1161 *
1162 *
1163 *
1164 *
1165 *
1166 *
1167 *
1168 *
1169 *
1170 *
1171 *
1172 *
1173 *
1174 *
1175 *
1176 *
1177 *
1178 *
1179 *
1180 *
1181 *
1182 *
1183 *
1184 *
1185 *
1186 *
1187 *
1188 *
1189 *
1190 *
1191 *
1192 *
1193 *
1194 *
1195 *
1196 *
1197 *
1198 *
1199 *
1200 *
1201 *
1202 *
1203 *
1204 *
1205 *
1206 *
1207 *
1208 *
1209 *
1210 *
1211 *
1212 *
1213 *
1214 *
1215 *
1216 *
1217 *
1218 *
1219 *
1220 *
1221 *
1222 *
1223 *
1224 *
1225 *
1226 *
1227 *
1228 *
1229 *
1230 *
1231 *
1232 *
1233 *
1234 *
1235 *
1236 *
1237 *
1238 *
1239 *
1240 *
1241 *
1242 *
1243 *
1244 *
1245 *
1246 *
1247 *
1248 *
1249 *
1250 *
1251 *
1252 *
1253 *
1254 *
1255 *
1256 *
1257 *
1258 *
1259 *
1260 *
1261 *
1262 *
1263 *
1264 *
1265 *
1266 *
1267 *
1268 *
1269 *
1270 *
1271 *
1272 *
1273 *
1274 *
1275 *
1276 *
1277 *
1278 *
1279 *
1280 *
1281 *
1282 *
1283 *
1284 *
1285 *
1286 *
1287 *
1288 *
1289 *
1290 *
1291 *
1292 *
1293 *
1294 *
1295 *
1296 *
1297 *
1298 *
1299 *
1300 *
1301 *
1302 *
1303 *
1304 *
1305 *
1306 *
1307 *
1308 *
1309 *
1310 *
1311 *
1312 *
1313 *
1314 *
1315 *
1316 *
1317 *
1318 *
1319 *
1320 *
1321 *
1322 *
1323 *
1324 *
1325 *
1326 *
1327 *
1328 *
1329 *
1330 *
1331 *
1332 *
1333 *
1334 *
1335 *
1336 *
1337 *
1338 *
1339 *
1340 *
1341 *
1342 *
1343 *
1344 *
1345 *
1346 *
1347 *
1348 *
1349 *
1350 *
1351 *
1352 *
1353 *
1354 *
1355 *
1356 *
1357 *
1358 *
1359 *
1360 *
1361 *
1362 *
1363 *
1364 *
1365 *
1366 *
1367 *
1368 *
1369 *
1370 *
1371 *
1372 *
1373 *
1374 *
1375 *
1376 *
1377 *
1378 *
1379 *
1380 *
1381 *
1382 *
1383 *
1384 *
1385 *
1386 *
1387 *
1388 *
1389 *
1390 *
1391 *
1392 *
1393 *
1394 *
1395 *
1396 *
1397 *
1398 *
1399 *
1400 *
1401 *
1402 *
1403 *
1404 *
1405 *
1406 *
1407 *
1408 *
1409 *
1410 *
1411 *
1412 *
1413 *
1414 *
1415 *
1416 *
1417 *
1418 *
1419 *
1420 *
1421 *
1422 *
1423 *
1424 *
1425 *
1426 *
1427 *
1428 *
1429 *
1430 *
1431 *
1432 *
1433 *
1434 *
1435 *
1436 *
1437 *
1438 *
1439 *
1440 *
1441 *
1442 *
1443 *
1444 *
1445 *
1446 *
1447 *
1448 *
1449 *
1450 *
1451 *
1452 *
1453 *
1454 *
1455 *
1456 *
1457 *
1458 *
1459 *
1460 *
1461 *
1462 *
1463 *
1464 *
1465 *
1466 *
1467 *
1468 *
1469 *
1470 *
1471 *
1472 *
1473 *
1474 *
1475 *
1476 *
1477 *
1478 *
1479 *
1480 *
1481 *
1482 *
1483 *
1484 *
1485 *
1486 *
1487 *
1488 *
1489 *
1490 *
1491 *
1492 *
1493 *
1494 *
1495 *
1496 *
1497 *
1498 *
1499 *
1500 *
1501 *
1502 *
1503 *
1504 *
1505 *
1506 *
1507 *
1508 *
1509 *
1510 *
1511 *
1512 *
1513 *
1514 *
1515 *
1516 *
1517 *
1518 *
1519 *
1520 *
1521 *
1522 *
1523 *
1524 *
1525 *
1526 *
1527 *
1528 *
1529 *
1530 *
1531 *
1532 *
1533 *
1534 *
1535 *
1536 *
1537 *
1538 *
1539 *
1540 *
1541 *
1542 *
1543 *
1544 *
1545 *
1546 *
1547 *
1548 *
1549 *
1550 *
1551 *
1552 *
1553 *
1554 *
1555 *
1556 *
1557 *
1558 *
1559 *
1560 *
1561 *
1562 *
1563 *
1564 *
1565 *
1566 *
1567 *
1568 *
1569 *
1570 *
1571 *
1572 *
1573 *
1574 *
1575 *
1576 *
1577 *
1578 *
1579 *
1580 *
1581 *
1582 *
1583 *
1584 *
1585 *
1586 *
1587 *
1588 *
1589 *
1590 *
1591 *
1592 *
1593 *
1594 *
1595 *
1596 *
1597 *
1598 *
1599 *
1600 *
1601 *
1602 *
1603 *
1604 *
1605 *
1606 *
1607 *
1608 *
1609 *
1610 *
1611 *
1612 *
1613 *
1614 *
1615 *
1616 *
1617 *
1618 *
1619 *
1620 *
1621 *
1622 *
1623 *
1624 *
1625 *
1626 *
1627 *
1628 *
1629 *
1630 *
1631 *
1632 *
1633 *
1634 *
1635 *
1636 *
1637 *
1638 *
1639 *
1640 *
1641 *
1642 *
1643 *
1644 *
1645 *
1646 *
1647 *
1648 *
1649 *
1650 *
1651 *
1652 *
1653 *
1654 *
1655 *
1656 *
1657 *
1658 *
1659 *
1660 *
1661 *
1662 *
1663 *
1664 *
1665 *
1666 *
1667 *
1668 *
1669 *
1670 *
1671 *
1672 *
1673 *
1674 *
1675 *
1676 *
1677 *
1678 *
1679 *
1680 *
1681 *
1682 *
1683 *
1684 *
1685 *
1686 *
1687 *
1688 *
1689 *
1690 *
1691 *
1692 *
1693 *
1694 *
1695 *
1696 *
1697 *
1698 *
1699 *
1700 *
1701 *
1702 *
1703 *
1704 *
1705 *
1706 *
1707 *
1708 *
1709 *
1710 *
1711 *
1712 *
1713 *
1714 *
1715 *
1716 *
1717 *
1718 *
1719 *
1720 *
1721 *
1722 *
1723 *
1724 *
1725 *
1726 *
1727 *
1728 *
1729 *
1730 *
1731 *
1732 *
1733 *
1734 *
1735 *
1736 *
1737 *
1738 *
1739 *
1740 *
1741 *
1742 *
1743 *
1744 *
1745 *
1746 *
1747 *
1748 *
1749 *
1750 *
1751 *
1752 *
1753 *
1754 *
1755 *
1756 *
1757 *
1758 *
1759 *
1760 *
1761 *
1762 *
1763 *
1764 *
1765 *
1766 *
1767 *
1768 *
1769 *
1770 *
1771 *
1772 *
1773 *
1774 *
1775 *
1776 *
1777 *
1778 *
1779 *
1780 *
1781 *
1782 *
1783 *
1784 *
1785 *
1786 *
1787 *
1788 *
1789 *
1790 *
1791 *
1792 *
1793 *
1794 *
1795 *
1796 *
1797 *
1798 *
1799 *
1800 *
1801 *
1802 *
1803 *
1804 *
1805 *
1806 *
1807 *
1808 *
1809 *
1810 *
1811 *
1812 *
1813 *
1814 *
1815 *
1816 *
1817 *
1818 *
1819 *
1820 *
1821 *
1822 *
1823 *
1824 *
1825 *
1826 *
1827 *
1828 *
1829 *
1830 *
1831 *
1832 *
1833 *
1834 *
1835 *
1836 *
1837 *
1838 *
1839 *
1840 *
1841 *
1842 *
1843 *
1844 *
1845 *
1846 *
1847 *
1848 *
1849 *
1850 *
1851 *
1852 *
1853 *
1854 *
1855 *
1856 *
1857 *
1858 *
1859 *
1860 *
1861 *
1862 *
1863 *
1864 *
1865 *
1866 *
1867 *
1868 *
1869 *
1870 *
1871 *
1872 *
1873 *
1874 *
1875 *
1876 *
1877 *
1878 *
1879 *
1880 *
1881 *
1882 *
1883 *
1884 *
1885 *
1886 *
1887 *
1888 *
1889 *
1890 *
1891 *
1892 *
1893 *
1894 *
1895 *
1896 *
1897 *
1898 *
1899 *
1900 *
1901 *
1902 *
1903 *
1904 *
1905 *
1906 *
1907 *
1908 *
1909 *
1910 *
1911 *
1912 *
1913 *
1914 *
1915 *
1916 *
1917 *
1918 *
1919 *
1920 *
1921 *
1922 *
1923 *
1924 *
1925 *
1926 *
1927 *
1928 *
1929 *
1930 *
1931 *
1932 *
1933 *
1934 *
1935 *
1936 *
1937 *
1938 *
1939 *
1940 *
1941 *
1942 *
1943 *
1944 *
1945 *
1946 *
1947 *
1948 *
1949 *
1950 *
1951 *
1952 *
1953 *
1954 *
1955 *
1956 *
1957 *
1958 *
1959 *
1960 *
1961 *
1962 *
1963 *
1964 *
1965 *
1966 *
1967 *
1968 *
1969 *
1970 *
1971 *
1972 *
1973 *
1974 *
1975 *
1976 *
1977 *
1978 *
1979 *
1980 *
1981 *
1982 *
1983 *
1984 *
1985 *
1986 *
1987 *
1988 *
1989 *
1990 *
1991 *
1992 *
1993 *
1994 *
1995 *
1996 *
1997 *
1998 *
1999 *
2000 *

```

```

2001 *
2002 *
2003 *
2004 *
2005 *
2006 *
2007 *
2008 *
2009 *
2010 *
2011 *
2012 *
2013 *
2014 *
2015 *
2016 *
2017 *
2018 *
2019 *
2020 *
2021 *
2022 *
2023 *
2024 *
2025 *
2026 *
2027 *
2028 *
2029 *
2030 *
2031 *
2032 *
2033 *
2034 *
2035 *
2036 *
2037 *
2038 *
2039 *
2040 *
2041 *
2042 *
2043 *
2044 *
2045 *
2046 *
2047 *
2048 *
2049 *
2050 *
2051 *
2052 *
2053 *
2054 *
2055 *
2056 *
2057 *
2058 *
2059 *
2060 *
2061 *
2062 *
2063 *
2064 *
2065 *
2066 *
206
```

```

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  $\tilde{A}_4^1$  jede verbindung zu berechnen erst loop  $\tilde{A}_4^1$ ber 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  $\tilde{A}_4^1$ ber alle Verbindungen

```



```

149     for (int conn = 0; conn < (connections.length); conn++){
150
151         // Gradientenabstieg  $\tilde{A}_{\frac{1}{4}}$  outputLayer
152         if (layer == connectionLayer){
153             // delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe)
154             // Ausgabe (1 - Ausgabe) * Eingabevektor_u
155             // -> delta Gewichtsvektor_u = Lernfaktor Ausgabe (1 - Ausgabe) *
156             // Eingabevektor_u auch in backpropagation (hinten for-loops)
157             // -> nur (Erwartete Ausgabe - Ausgabe) benötigt
158             position = connections[conn].getPositionNeuronTo();
159
160             weightDelta = wantedOut[position] - resultOut[position];
161         }
162
163         // Da real backpropagation
164         else {
165             // Beginn der Berechnung
166             // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
167             // weightDelta = trainingCoefficient *
168
169             // Summe  $\tilde{A}_{\frac{1}{4}}$ ber Folgeneuronen
170             for (int succ = 0; succ < succNeurons; succ++){
171                 int sumOutputNeurons = 0;
172
173                 // Summe  $\tilde{A}_{\frac{1}{4}}$ ber Ausgabeneuronen
174                 for (int out = 0; out < outputNeurons.length; out++){
175                     sumOutputNeurons += (
176                         // Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron
177                         (wantedOut[out] - resultOut[out]) *
178                         // dAusgabe nach dNetzeingabe
179                         resultOut[out] * (1 - resultOut[out]));
180                 } // for() Summe  $\tilde{A}_{\frac{1}{4}}$ ber Ausgabeneuronen
181
182                 weightDelta += sumOutputNeurons;
183
184                 // * Gewicht_Neuron-FolgeNeuron
185                 position = connections[conn].getPositionNeuronTo();
186
187                 weightDelta *= succWeights[position];
188             } // for() Summe  $\tilde{A}_{\frac{1}{4}}$ ber Folgeneuronen
189         }
190
191         // * Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
192         // Ausgabe_u = getOutput von aktuellem Neuron -> connection[neuron + conn
193         // ].getNeuronTo().getOutput
194         // Eingabevektor_u = inputs[] von aktuellem Neuron -> getInputVector() in
195         // Neuron implementieren
196         outputOfNeuron = connections[conn].getNeuronTo().getOutput();
197         inputOfNeuron = connections[conn].getNeuronFrom().getOutput();
198
199         weightDelta *= (outputOfNeuron * (1 - outputOfNeuron) * inputOfNeuron);
200
201         // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
202         weightDelta *= trainingCoefficient;
203
204         // Add calculated weight difference to connection weight
205         connections[conn].addWeightDelta(weightDelta);
206     } // for()  $\tilde{A}_{\frac{1}{4}}$ ber alle Verbindungen
207
208     // for  $\tilde{A}_{\frac{1}{4}}$ ber die Thresholds
209     for (int thresh = 0; thresh < numberOfThresholds; thresh++){
210         // TODO: Der Schwellenwert wird auf 0 festgelegt??? noch notwendig?
211         // , als Ausgleich wird ein zusätzlicher (imaginaerer)
212         // Eingang ( $x_{-0}$ ) eingefügt, der den festen Wert 1 hat und mit dem
213         // negierten Schwellenwert gewichtet wird. (S.32)
214         // Position in Layer = threshold * Anzahl der Verbindungen pro Neuron ->
215         // Anzahl der Verbindungen in Layer / Anzahl der Neuronen (Thresholds)

```

```

210     int positionInLayer = thresh * (connections.length / numberOfThresholds);
211     float oldThresh = -1 * connections[positionInLayer].getNeuronTo().
        getThreshold();
212
213     // Gradientenabstieg  $\tilde{A}_{\frac{1}{4}}r$  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) ben $\tilde{A}_{\frac{1}{4}}$ 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  $\tilde{A}_{\frac{1}{4}}$ ber Folgeuronen
230         for (int succ = 0; succ < succNeurons; succ++){
231             int sumOutputNeurons = 0;
232
233             // Summe  $\tilde{A}_{\frac{1}{4}}$ ber 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  $\tilde{A}_{\frac{1}{4}}$ ber Ausgabeneuronen
241
242             weightDelta += sumOutputNeurons;
243
244             // * Gewicht_Neuron-FolgeNeuron
245             weightDelta *= succWeights[succ];
246         } // for() Summe  $\tilde{A}_{\frac{1}{4}}$ ber Folgeuronen
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 ben $\tilde{A}_{\frac{1}{4}}$ 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()  $\tilde{A}_{\frac{1}{4}}$ ber thresholds
263     } // for()  $\tilde{A}_{\frac{1}{4}}$ ber alle (Verbindungs-)Layer
264     } // backpropagation()

```

Chapter 5

Prospective Enhancements

5.1 Execution of multiLayerPerceptron using integer

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

5.2 Execution of multiLayerPerceptron using integer, defined threshold

```
1  /*
   * *****
   *
   * Executes the built multi-layer perceptron with given input vector
   * Returns the output vector as single integer
   * Bit is set if depending output value is > threshold
   * *****
   */
2  int runInt(int inputVector, float threshold){
3      int output = 0;
4      float networkOutputVector[] = new float[outputVector.length];
5
6
7
8
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 ComputationalIntelligence S.52)

Step function: if (netInput \geq threshold) return netInput; else return 0;

semi-lineare Funktion: if (netInput \geq threshold + 1/2) return netInput;

else if ((netInput \geq threshold + 1/2) && (netInput \geq 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

```
1  /*
   *****
2  * class ThresholdItem:
3  * up to 32 inputs possible
4  * inputs are just 0 or 1
5  * output = input.1*weights[0] + ... + input.32*weights[31]
6  * So far, output = netInput
7  * activated just returns true if output >= threshold
8  * Note! So far, activated can just be called if getOutput was called before
9  *****
   */
10 class ThresholdItem
11 {
12 protected
13     int inputs;
14     int nrInputs;
15     // So far also used as output
16     float netInput;
17     float weights[];
18     float threshold;
19     // Used as output
20     Boolean activated;
21
22 public
23     // Constructors
24     ThresholdItem() {};
25
26     ThresholdItem(int nrInputs){
27         this.nrInputs = nrInputs;
28         weights = new float[nrInputs];
29
30         // Initialize inputs, output and weights to 0, threshold to max
31         inputs = 0;
32         netInput = 0;
33         activated = false;
34         threshold = 0x7fffffff;
35
36         for (int i = 0; i < nrInputs; i++)
37             weights[i] = 0;
38     }
39
40     Boolean setInput(int nrInput){
41         if(nrInput < nrInputs){
42             inputs |= 1 << nrInput;
43
44             return true;
45         }
46
47         return false;
48     }
49 }
```

```

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

```