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
- 2.2 Constructors . . . 3
- 2.3 Methods . . . 3

3 class Connection **5**
- 3.1 Attributes . . . 5
- 3.2 Constructors . . . 5
- 3.3 Methods . . . 5

4 class MultiLayerPerceptron **6**
- 4.1 Attributes . . . 6
- 4.2 Constructors . . . 7
- 4.3 Methods . . . 7

5 Prospective Enhancements **13**
- 5.1 Up to 64 inputs/outputs . . . 13
- 5.2 Activation function and Output function settable . . . 13

6 Test results **14**
- 6.1 Unit test for NeuronFloat . . . 14

7 Discarded **15**

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

```
int runInt(int inputVector)
```

```
int runInt(int inputVector, float threshold)
```

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

Constructor mit Connections fuer input layer: each: NrConnections = hidden, twoGroups: NrConnections = hidden/2, one: NrConnections = inputNeur

Constructor mit + Connections fuer hidden Layer: each, cross und zigzag

```
/*
    ***********************************************************************************
* Multi-layer perceptron where number of input neurons, number of hidden neurons
    per
* layer, number of layers and number of output neurons can be defined.
* Also it is possible to define the type of connection between input layer and 1st
    hidden
* layer and between the hidden layers.
/*********************************************************************************
    */
MultiLayerPerceptron(int inputNeurons, int hiddenNeuronsPerLayer, int hiddenLayers,
        int outputNeurons, String inputConnections, String hiddenConnections){
  // Connections for input layer: each: NrConnections = hidden, twoGroups:
        NrConnections = hidden/2, one: NrConnections = inputNeur
  // Connections for hidden Layer: each, cross and zigzag
}// MultiLayerPerceptron()
```

Constructor mit + Output topologies each und groups

```
/*
    ***************************************************************************************
* Multi-layer perceptron with output topologies each und groups
***************************************************************************************
    */
```

Constructor mit hidden neurons rising from inputs to variable / layer

```
/*
    ***************************************************************************************
* Multi-layer perceptron with hidden neurons rising from inputs to variable / layer
***************************************************************************************
    */
```

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

1.4 Tests

Rerun of unitTestNeuron
Redraft tests for MultiLayerPerceptron: What is need? What needs to be tested?

```
//run test again with int input vector
int outputVector = MultiLayerPerceptron[i].runInt(0b11111111);
```

Finish test for backpropagation

Chapter 2

class Neuron

Uses step function for calculating the output.

2.1 Attributes

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.

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

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

float weights[] The weights of the inputs. Array size depends on the number of inputs.

float threshold The threshold when output is activated.

2.2 Constructors

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

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.

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.

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.

void setThreshold(float threshold) Sets threshold for the neuron.

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

Neuron neuronFrom Neuron the output is taken from.

Neuron neuronTo Neuron the input is set.

int input Input of `neuronTo` which is set.

float connWeight Weight of the connection.

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

3.2 Constructors

Connection(Neuron neuronFrom, Neuron neuronTo, int input, float connWeight) Just sets `neuronFrom`, `neuronTo`, `input`, and `connWeight`.

3.3 Methods

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

Neuron inputNeuron[] Input layer, size depends on definition.

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).

Neuron outputNeuron[] Output layer, size depends on definition.

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

float inputConnWeights[] Weights of the connections between input layer and 1st hidden layer.

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`.

float outputConnWeights[] Weights of the connections between the last hidden layer and the output layer.

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.

Connection inputConnection[] Connections between input layer and 1st hidden layer.

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`.

Connection outputConnection[] Connections between the last hidden layer and the output layer.

4.2 Constructors

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

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.

```
/*
    ***************************************************************************************
 * TODO: Klaeren ob inkl. threshold
 * Kommentare in TechDoc
 *
 *
      ________________________________________________________________________________________
 * Beispiel: 2 Input, 4 Hidden, 2 Hiddenlayer, 2 Output
 *    Connection each, Oupuzt connection 2 Groups
 *               ______________________________________________________________
 *    inputConnection[] | hiddenConnection[][] | outputConnection[]
 * 0 : 0 >      0 >      0
 *          1
 *          2
 *          3
 *    1 >      0
 *          1
 *          2
 *          3
 *    2 >      0
```

```
 *              1 >        0
 *              2
 *              3
 *     3 >        0
 *              1
 *              2
 *              3
 * 1 : 0 >        0
 *              1
 *              2 >        1
 *              3
 *     1 >        0
 *              1
 *              2
 *              3
 *     2 >        0
 *              1
 *              2
 *              3 >        1
 *     3 >        0
 *              1
 *              2
 *              3
 *                     ____________________________________________________________
 *  ________________________________________________________________________________
 *
 * Backpropagation allgemein:
 * deltaGewichtsvektor_u = Lernfaktor (Summe Ã¼ber Folgeneuronen((Summe Ã¼ber Ausgabeneuronen(
 * (Erwartete Ausgabe - Ausgabe) dAusgabe_Ausgabeneuron nach dNetzeingabe_Folgeneuron))*Gewicht_Neuron-FolgeNeuron)
 * ) dAusgabe_u nach dNetzeingabe_u * Eingabevektor_u (S.71)
 *
 * FÃ¼r Identiaet als Ausgabefunktion und logistischer Aktivierungsfunktion:
 * deltaGewichtsvektor_u = Lernfaktor (Summe Ã¼ber Folgeneuronen((Summe Ã¼ber Ausgabeneuronen(
 * (Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron)dAusgabe_Ausgabeneuron nach dNetzeingabe_Folgeneuron))*Gewicht_FolgeNeuron-u)
 * ) Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u

 * -> SCHICHTENWEISE
-> Summe Ã¼ber Ausgabeneuronen:
 * -> Summe Ã¼ber Folgeneuronen: Pfad im Netz muss mathematisch nachgebildet werden
 * -> Eingabevektor muss angepasst werden, enthaelt Bitmuster
 * -> bei logistischer Funktion:
 * dAusgabe_Ausgabeneuron nach dNetzeingabe_Folgeneuron = Ausgabe_Ausgabeneuron (1 - Ausgabe_Ausgabeneuron)
 *
 * Einzelner Gradientenabstieg:
 * delta Gewichtsvektor_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 - Ausgabe) * Eingabevektor_u
 * -> pro inputneuron:
 * delta Gewicht_u = Lernfaktor (Erwartete Ausgabe - Ausgabe) Ausgabe (1 - Ausgabe) * Eingabe_u
 *
 * Um deltaGewicht fÃ¼r jede verbindung zu berechnen erst mit for-loop Ã¼ber outputconnection.length
 * dann for (int i = hiddenLayer-2; i>=0; i++) -> loop Ã¼ber hiddenLayer[i].length
 * dann for-loop Ã¼ber inputConnection.length
 *
 * Um Folgeneuronen zu ermitteln Position in Array benÃ¶tigt, dann loops Ã¼ber die folgende
 * Netzstruktur (nicht einfach Ã¼ber die ConnectionArrays)
```

```
 *****************************************************************************************/
private void backpropagation(float[] resultOut, float[] wantedOut){
    // TODO: Kommentare wofÃ¼r Variablen, Richtige Reihenfolge
    int connectionsOfNeuron;
    int neuronsInLayer;
    int neuronsInNextLayer;
    int offset;
    float trainingCoefficient = 0.2f;
    float weightDelta = 0;
    float connWeights[];
    Connection connections[];
    float outputOfNeuron;
    float inputVectorOfNeuron;

    // Um deltaGewicht fÃ¼r jede verbindung zu berechnen loop Ã¼ber alle (Verbindungs-)Layer...
    for (int layer = hiddenLayers; layer >= 0; layer--){
      // Bestimmen der Position des Neurons im Netz und der folgenden
      // Struktur fÃ¼r die Summe Ã¼ber die Folgeneuronen,
      // d.h. fÃ¼r die Grenze der for-Schleife. Daher nur int neuronsInLayer benÃ¶tigt
      // -> mit layer ermitteln in welchem Layer: output, hidden o input
      // -> mit neuron an welcher Stelle im Layer
      // Grenze fÃ¼r Summe Ã¼ber Folgeneuronen

      // DEBUG
      System.out.println("int layer = " + hiddenLayers + "; layer >= 0; layer: " + layer + "{");

      // Outputconnection layer
      if (layer == hiddenLayers){
        neuronsInLayer = hiddenNeuron[layer-1].length;
        neuronsInNextLayer = 1;

        // Es gibt nur Verbindung(en) vom letztem Hidden Layer zu OutputNeuron(en)

        // TODO: Abbruchbedingung, ist noch falsch, benÃ¶tigt allgemein gÃ¼ltigen Ausdruck
        // Go to next output neuron if all inputs are connected
        // Funktioniert nur wenn neuron > 0
        //while ((neuron % (hiddenNeuron[hiddenLayers-1].length/outputNeuron.length)) != 0){}

        // Im Moment nur eine Verbindung von hidden Neuron zu outputLayer
        connectionsOfNeuron = 1;

      /************************************************************
      // Connections between last hidden and output layer
      int inp = 0;
      int outNeur = -1;

      // From each hidden neuron...
      for (int hidNeur = 0; hidNeur < hiddenNeuron[hiddenLayers-1].length; hidNeur++, inp++){

        // Go to next output neuron if all inputs are connected
        if ((hidNeur % (hiddenNeuron[hiddenLayers-1].length/outputNeuron.length)) == 0){
          outNeur++;
          inp = 0;
        }

        // ...to the "nearest" output neuron (if several are available)
        outputConnection[hidNeur] = new Connection(hiddenNeuron[hiddenLayers-1][hidNeur],
          outputNeuron[outNeur], inp, outputConnWeights[hidNeur]);
      }
```

```
    ***********************************************************/

      // TODO: Auch hier allgemeine Formulierungen
      connWeights = new float [outputConnWeights.length];
      connWeights = outputConnWeights;

      connections = new Connection [outputConnection.length];
      connections = outputConnection;
    }

    // InputConnection layer
    else if (layer == 0){
      neuronsInLayer = inputNeuron.length;
      neuronsInNextLayer = hiddenNeuron[0].length;

      // TODO: Allgemein gÃ¼ltiger Ausdruck
      // Im Moment nur von jeden inputNeuron zu jedem hiddenNeuron
      connectionsOfNeuron = hiddenNeuron[0].length;

    /***********************************************************
    // Connections between input layer and 1st hidden layer
    int positionInLayer = 0;

    // From each input neuron...
    for (int inp = 0; inp < inputNeuron.length; inp++){

      // ...to each neuron of the 1st hidden layer
      for (int conn = 0; conn < hiddenNeuron[0].length; conn++){

        inputConnection[positionInLayer] = new Connection(inputNeuron[inp],
            hiddenNeuron[0][conn], inp, inputConnWeights[positionInLayer]);

        positionInLayer++;
      }
    }
    ***********************************************************/

      connWeights = new float [inputConnWeights.length];
      connWeights = inputConnWeights;

      connections = new Connection [inputConnection.length];
      connections = inputConnection;

    /***********************************************************
    // Connection weights
    inputConnWeights = new float[inputNeurons * hiddenNeuronsPerLayer];

    hiddenConnWeights = new float[hiddenLayers-1][hiddenNeuronsPerLayer *
        hiddenNeuronsPerLayer];

    outputConnWeights = new float[hiddenNeuronsPerLayer];

    Struktur siehe Connections weiter oben
    ***********************************************************/
    }

    // Hidden Layer
    else {
      neuronsInLayer = hiddenNeuron[layer-1].length;
      neuronsInNextLayer = hiddenNeuron[layer].length;

      // TODO: Allgemein gÃ¼ltiger Ausdruck
      // Im Moment nur von jeden hiddenNeuron zu jedem hiddenNeuron im Folgelayer
      connectionsOfNeuron = hiddenNeuron[layer].length;

      connWeights = new float [hiddenConnWeights[layer-1].length];
      connWeights = hiddenConnWeights[layer-1];

      connections = new Connection [hiddenConnection[layer-1].length];
```

```
      connections = hiddenConnection[layer -1];
    }
    //DEBUG
    System.out.println("\tconnWeights.lenght: " + connWeights.length);

    // ..., Ã¼ber alle Neuronen in dem Layer...
    for (int neuron = 0; neuron < neuronsInLayer; neuron++){
      //DEBUG
      System.out.println("\tfor (int neuron = 0; neuron < "+neuronsInLayer+"; neuron: "+neuron+"{");

      // Position of 1st connection of this neuron (Is the same
      // for all neurons in this layer)
      offset = neuron * connectionsOfNeuron;

      // ...und nu Ã¼ber alle Verbindungen des Neurons
      for (int conn = 0; conn < connectionsOfNeuron; conn++){
        //DEBUG
        System.out.println("\t\tfor (int conn = 0; conn < "+connectionsOfNeuron+"; conn" +conn+ "{");

        /*** Notwendig?
        // pro Inputneuron:
        // Notwendig? wie sonst inputVector[inp] verwenden?
        for (int inp = 0; inp < inputNeuron.length; inp++){
        }// for() pro Inputneuron
        ***/

        // Beginn der Berechnung
        // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
        // weightDelta = trainingCoefficient *

        // DEBUG
        System.out.println("\t\tneuronsInNextLayer: " + neuronsInNextLayer);

        // Summe Ã¼ber Folgeneuronen
        for (int succ = 0; succ < neuronsInNextLayer; succ++){

          // Summe Ã¼ber Ausgabeneuronen
          for (int out = 0; out < outputNeuron.length; out++){
            weightDelta += (
            // Erwartete Ausgabe_Ausgabeneuron - Ausgabe_Ausgabeneuron
            (wantedOut[out] - resultOut[out]) *
            //dAusgabe nach dNetzeingabe
            resultOut[out] * (1- resultOut[out]));
          }// for() Summe Ã¼ber Ausgabeneuronen

          // Bestimmen der Position im Netz um Gewicht_Neuron-FolgeNeuron zu laden
          // layer und neuron: Position von Neuron im Netz
          // connectionsOfNeuron: Anzahl der Folgeneuronen von Neuron
          // succ: x-tes FolgeNeuron von Neuron

          // * Gewicht_Neuron-FolgeNeuron

          //DEBUG
          System.out.println("\t\t\tconnWeights["+ (offset+succ) + "]");

          weightDelta *= connWeights[offset + succ];
        }// for() Summe Ã¼ber Folgeneuronen

        // * Ausgabe_u (1 - Ausgabe_u) * Eingabevektor_u
        // Ausgabe_u  = getOutput von aktuellem Neuron -> connection[neuron + conn].getNeuronTo.getOutput
        // Eingabevektor_u  = inputs[] von aktuellem Neuron -> getInputVector() in Neuron implementieren
        // Eingabevektor_u = netInput, da Verbindungsgewichte schon in inputs integriert
```

```
          outputOfNeuron = connection[neuron + conn].getNeuronTo.getOutput;
          inputVectorOfNeuron = connection[neuron + conn].getNeuronTo.getNetInput;

          weightDelta *= (outputOfNeuron * (1 - outputOfNeuron) *
              inputVectorOfNeuron;

          // deltaGewichtsvektor_u = Lernfaktor * -> hinter die Summen verschoben
          weightDelta *= trainingCoefficient;

          // Add calculated weight difference to connection weight
          connections[neuron + conn].addWeightDelta(weightDelta);
        }// for() Ã¼ber alle Verbindungen
      }// for() Ã¼ber alle Neuronen
    }// for() Ã¼ber alle (Verbindungs-)Layer
  }// backpropagation()
```

Chapter 5

Prospective Enhancements

5.1 Up to 64 inputs/outputs

For function int runInt()

5.2 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;
  }
```

```

  Boolean unsetInput(int nrInput){
    if(nrInput < nrInputs){
      inputs &= ~(1 << nrInput);

      return true;
    }

    return false;
  }

  Boolean setWeight(int nrInput, float weight){
    if(nrInput < weights.length){
      weights[nrInput] = weight;
      return true;
    }

    return false;
  }

  void setThreshold(float threshold){
    this.threshold = threshold;
  }

  float getOutput(){
    // Reset net input
    netInput = 0;

    // Net input function f_net
    for (int i = 0; i < nrInputs; i++){
      if((inputs & (1 << i)) == (1 << i))
        netInput += weights[i];
    }

    // Activation function f_act: step function
    if (netInput >= threshold){
      activated = true;

      // Output function f_out: Identity
      return netInput;
    }
    else {
      activated = false;
      return 0;
    }
  }

  Boolean getActivation(){
    return activated;
  }
}// class ThresholdItem
```