Neural Network - Steepest Gradient Method - Conjugate Gradient Method

Philipp Gloor

2. Februar 2014

Inhaltsverzeichnis

1 Einführung

1.1 Mathematische Neuronen

Die Anfänge der künstlichen Neuronen gehen auf Wareen McCulloch und Walter Pitts im Jahr 1943 zurück. Sie zeigten an einem vereinfachten Modell eines neuronalen Netzwerkes, dass diese logische und arithmetische Funktionen berechnen kann. Dieses Neuronenmodell ist heute als McCulloch-Pitts Neuron bekannt [1].

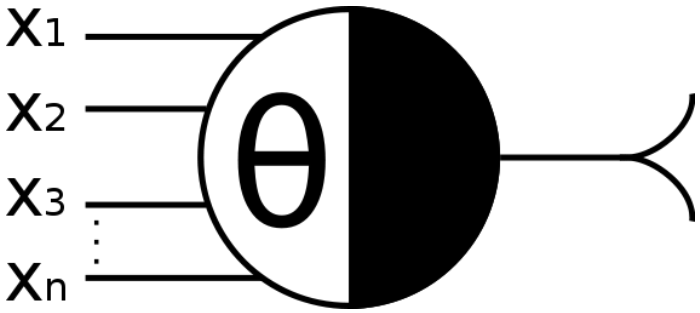


FIG. 1.1: Das McCulloch-Pitt Neuronenmodell

1.1.1 McCulloch-Pitt Neuron

Ein McCulloch-Pitt Neuron besteht aus mehreren Eingängen (stellvertretend für die Dendriten) und in der Regel einem einzigen Ausgang (das Axon). Die Werte aus jedem Eingang werden in der Regel summiert und falls die Summe dann höher als eine Schwelle Θ ist, als Ausgang ausgegeben. Der Ausgangswert war dann entweder 1 (*true*) oder 0 (*false*).

1949 beschrieb Donald Hebb die Hebbsche Lernregel

$$\Delta w_{ij} = \eta \cdot a_i \cdot o_i \tag{1.1}$$

- Δw_{ij} Veränderung des Gewichtes von Neuron j zu Neuron i
- η *Lernrate*
- a_i Aktivierung von Neuron i
- o_i Ausgabe von Neuron j, das mit Neuron i verbunden ist

Das bedeutet: Je häufiger ein Neuron A gleichzeitig mit Neuron B aktiv ist, umso bevorzugter werden die beiden Neuronen aufeinander reagieren ("what fires together, wires together").

2 Theorie

2.1 Perzeptron-Modell

1958 von Frank Rosenblatt entwickelt bildet das Perzeptron-Modell bis heute die Grundlage künstlicher neuronalen Netze. Das klassiche Neuron ist das sogennante Perzeptron. Es summiert über alle Eingänge, wobei es die Eingänge mit Gewichten, welche geeignet gelernt werden können, versieht. Es hat ausserdem eine Feuerschwelle (auch Bias genannt) und eine Ausgabefunktion. Auf die Feuerschwelle kann in vielen Fällen verzichtet werden. Das Lernen findet mit Hilfe eines Gradientenabstiegs statt.

Das Perzeptron-Modell ist aber limitiert. Es kann z.B. nicht das XOR Problem lösen (ein zweischichtiges Netzwerk kann das [2]).

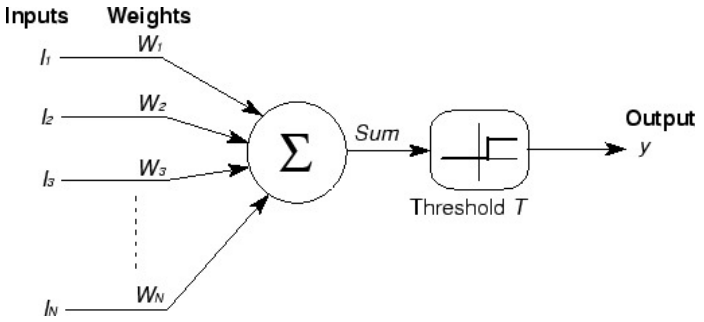


FIG. 2.2: Das Perzeptron[3]

2.2 Hebbsche Lernregel mit dem Perzeptron

Im zeitdiskreten Fall sieht die Lernregel folgendermassen aus:

$$y = w.in \tag{2.2}$$
$$e = t - y \tag{2.3}$$
$$w+ = \eta \cdot e.in \tag{2.4}$$

a.b steht für das Skalarprodukt zweier Vektoren oder Matrix-Vektormultiplikation.

2.3 Ausgabefunktion

Die Ausgabefunktion beschreibt, wie ein Zellkörper im Allgemeinen nichtlinear auf die Aufladung reagiert. Für die Ausgabefunktion wird oft die Sigmoidfunktion gewählt

$$f(x) = \frac{1}{1 + e^{-ax}}$$

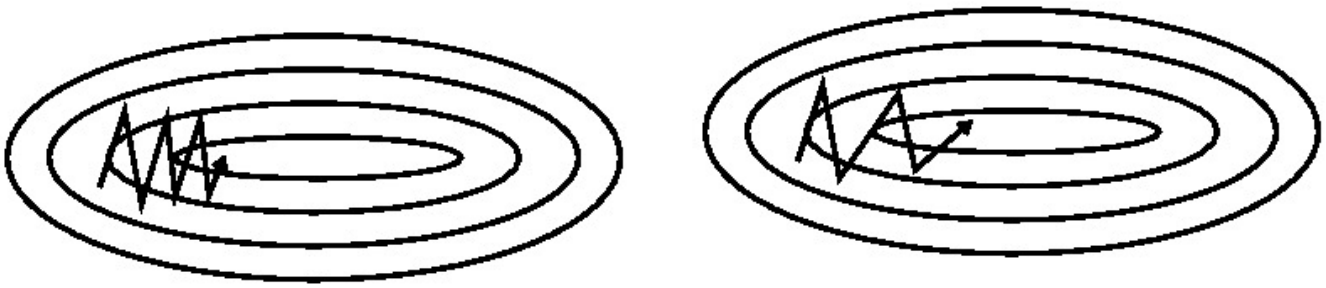
Für grosse *a* konvergiert sie zur klassischen Heavysidefunktion. Für kleine *a* wird der Übergang flacher. Die Sigmoid-

funktion ist monoton steigend, beschränkt und die Ableitung lässt sich durch kombinieren der Sigmoidfunktion selber schreiben:

$$f'(x) = a \cdot f(x)(1 - f(x))$$

2.4 Momentum Term

Bei *Momentum Term* handelt es sich um eine Erweiterung der Hebbschen Lernregel. Es geht darum, dass man beim ändern der Gewichte noch ein Schritt aus der Vergangenheit mitnimmt und dadurch meist schneller zum Minimum konvergiert.



(A) A tiger

FIG. 2.3: Ohne momentum term (links) und mit momentum term (rechts)[4]

$$\Delta w_{ij}(t) = \eta \cdot e.in + m\Delta w_{ij}(t-1)$$

wobei m ein neuer globaler Parameter ist, der durch *Trial and Error* bestimmt werden muss. Der *Momentum Term* addiert einen Bruchteil des vorherigen Gewichte-Update zum aktuellen. Wenn der Gradient in die gleiche Richtung zeigt, vergrössert das so die Schrittgrösse und man konvergiert schneller zum minimum.

2.5 Neuronale Netzwerke

Ein neuronales Netzwerk besteht aus einem Netzwerk von künstlichen Neuronen. Insgesamt handelt es sich um eine Abstraktion von Informationsverarbeitung aber es hat seinen Ursprung in der Biologie und dem Gehirn. Heutzutage werden aber neuronale Netzwerke weniger für das Nachbilden biologischer neuronaler Netze, was eher Gegenstand der *Computational Neuroscience* ist.

3 Anwendungen

Künstliche neuronale Netzwerke sind besonders für Anwendungen geeignet wo wenig oder kein explizites (systematisches Wissen) über das zu lösende Problem notwendig ist. Dazu gehören zum Beispiel

- Texterkennung

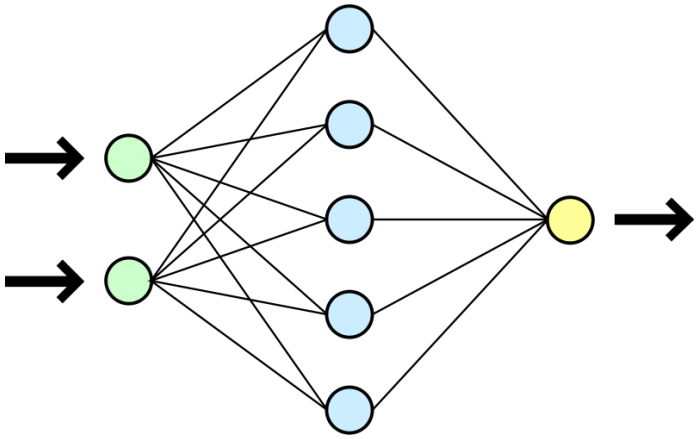


FIG. 2.4: Schematisches neuronales Netzwerk mit 2 Eingängen, einer versteckten Schicht und einem Ausgang[5]

- Bilderkennung
- Gesichtserkennung

bei denen einige Hunderttausend bis Millionen Bildpunkte in eine im Vergleich geringe Anzahl von erlaubten Ereignissen überführt werden müssen.

4 Veränderung der Parameter

Bei den durchgeführten Simulationen wurden folgende Parameter verändert und dann verglichen.

- η - Die *Lernrate*
- m - Der *'Momentum Term'*
- *hidnum* - Anzahl versteckter Neuronen im ersten *Hidden Layer*
- *hidhidnum* - Anzahl versteckter Neuronen im zweiten *Hidden Layer* (falls vorhanden)

5 Messungen und Resultate

In diesem Abschnitt werden die gefundenen Resultate diskutiert. Folgende Formen sollte das Neuronale Netzwerk erkennen:

5.1 Symbolerkennung mit einem Hidden Layer - zwei mögliche Outputs

5.1.1 Lernrate

Die *Lernrate* bestimmt wie stark die Gewichte angepasst werden. Grundsätzlich je grösser die *Lernrate*, desto schneller sollte man das Minimum finden. Es kann aber auch kontraproduktiv wirken insofern, dass man nicht beliebig nah ans

Minimum kommt, weil man durch die grosse *Lernrate* immer darüber hinaus schiesst.

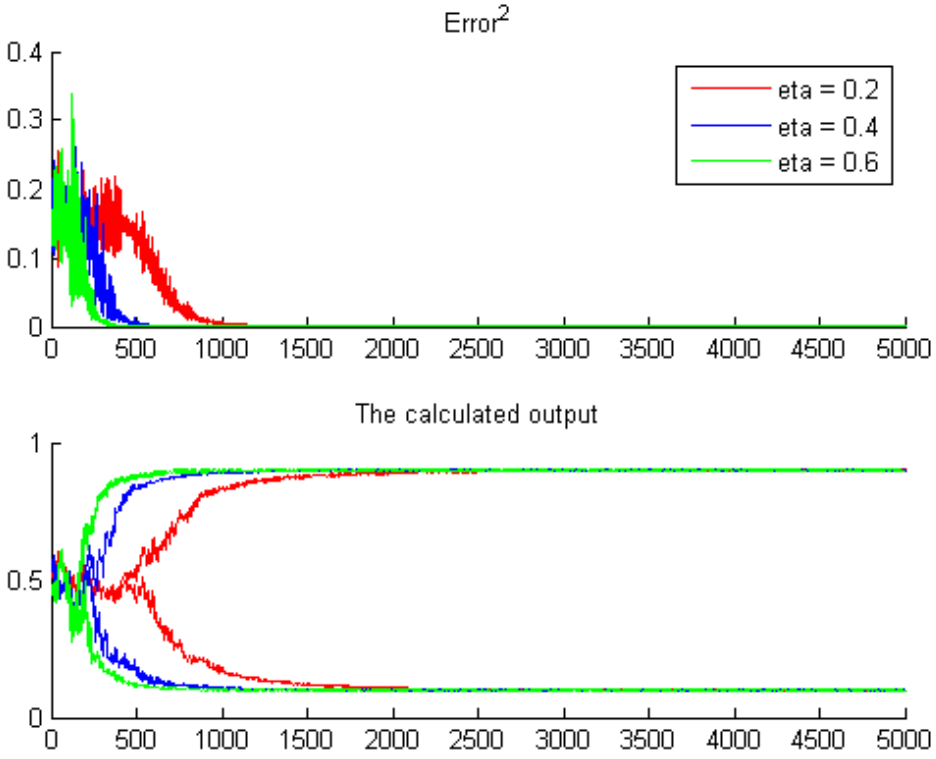


FIG. 5.5: Lernphase des Netzwerks mit verschiedenen η's

Hier sieht man gut, dass eine Vergrösserung der *Lernrate* von 0.2 auf 0.4 zu einem schnelleren Lernen führt. Jedoch der Sprung von 0.4 auf 0.6 weniger effizient ist, als der Sprung vorher. Aber 0.6 scheint in diesem Fall ein guter Wert zu sein, aber auch mit 0.4 funktioniert es gut.

Die anderen Parameter sind fixiert.

- $hidnum = 6$
- $m = 0.2$

5.1.2 Momentum Term

Der *Momentum Term* bestimmt wie viel wir vom vorherigen Schritt in den nächsten mitnehmen. In folgenden Beispiel hat der *Momentum Term* keinen grossen Einfluss. Das System lernt mit oder ohne *Momentum Term* fast gleich schnell und man könnte deshalb in diesem Beispiel auf den *Momentum Term* verzichten.

Die anderen Parameter sind fixiert.

- $hidnum = 6$
- $\eta = 0.2$

5.1.3 Versteckte Neuronen

Versteckte Neuronen sind solche Neuronen, deren Output man von aussen nicht sieht und ihren Output nur an die nächste Schicht weitergeben (in diesem Beispiel an das Output-Neuron).

Man sieht, dass die Anzahl versteckter Neuronen einen grossen Einfluss auf die Lerngeschwindigkeit des Systems hat, aber auch hier wieder ist der Sprung bei der zweiten Erhöhung (von 4 bis 6) nicht so gross wie beim ersten (von 2 bis 4).

Die anderen Parameter sind fixiert.

- $m = 0.2$
- $\eta = 0.2$

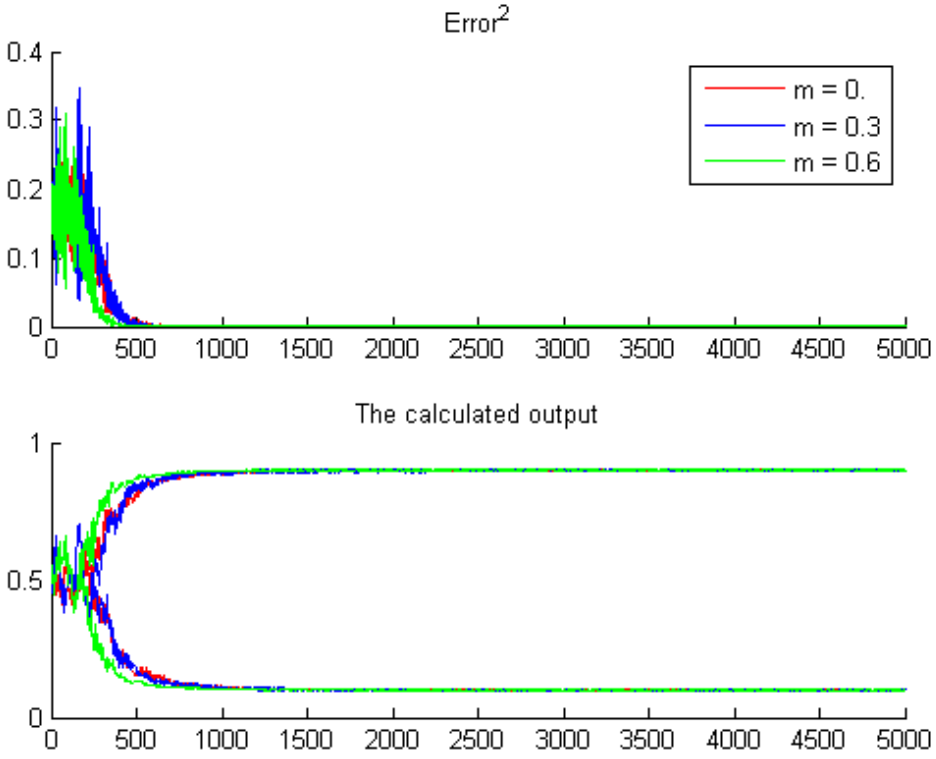


FIG. 5.6: Lernphase des Netzwerks mit verschiedenen m's

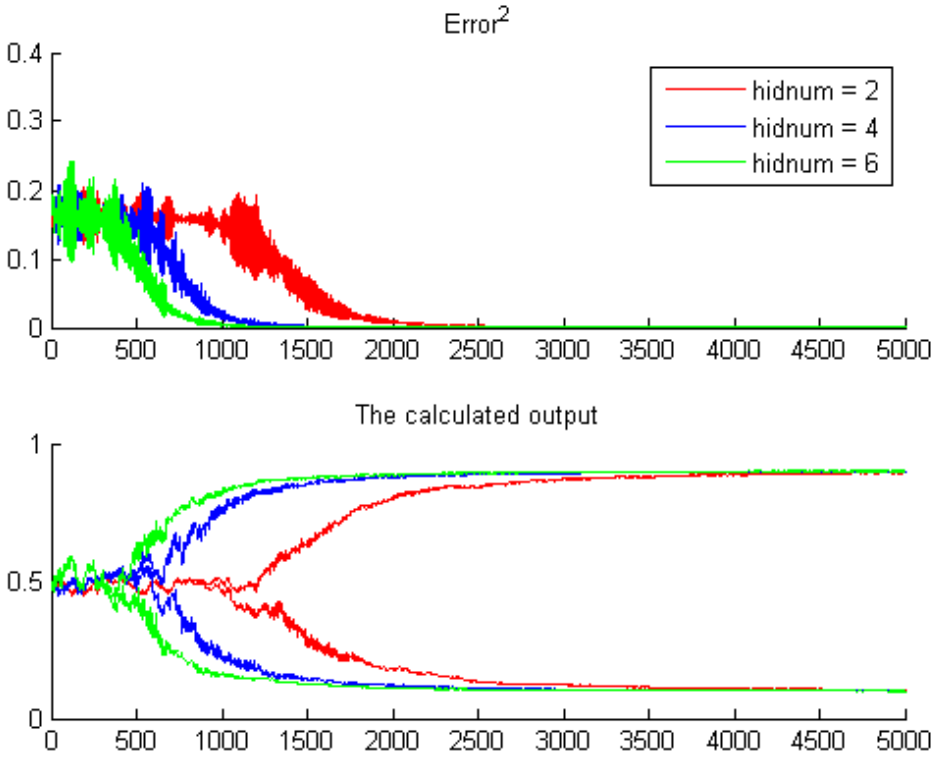


FIG. 5.7: Lernphase des Netzwerks mit verschiedenen *hidnum*'s

5.2 Symbolerkennung mit zwei Hidden Layer - zwei mögliche Outputs

5.2.1 Lernrate

Beim zweischichtigen neuronalen Netzwerk sieht man, wenn man eine zu hohe *Lernrate* wählt, dass der Effekt negativ sein kann. Bei $\eta = 0.6$ schafft es das Netzwerk nicht, für den zweiten Wert den richtigen Output zu lernen und der Fehler wie man sieht, springt immer hin und her und wird nicht kleiner.

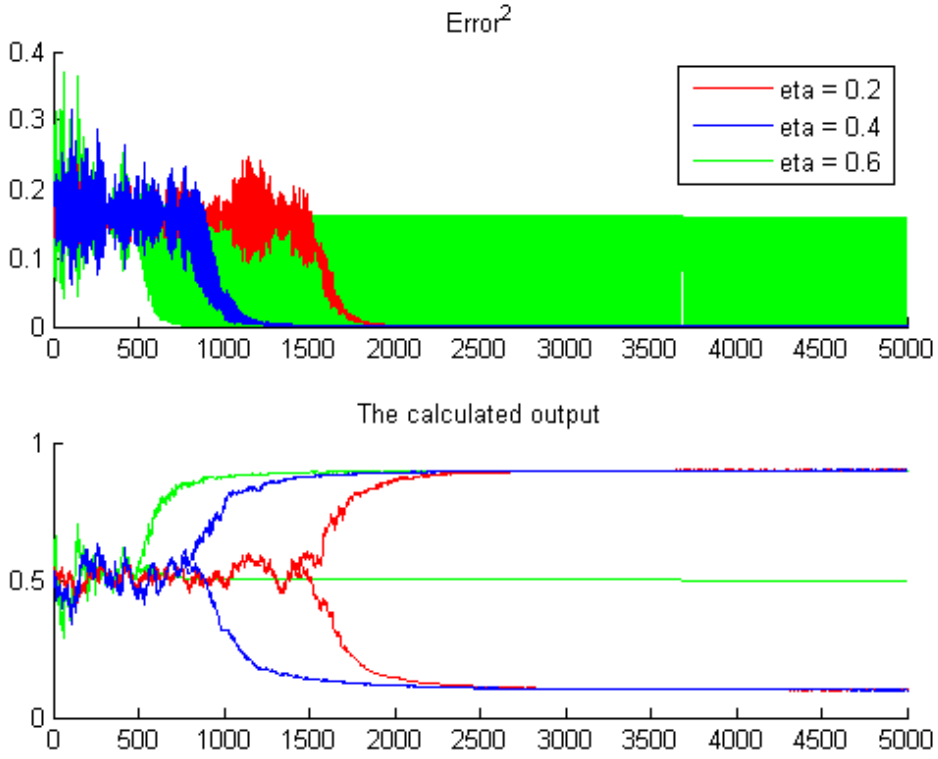


FIG. 5.8: Lernphase des Netzwerks mit verschiedenen η's

5.2.2 Momentum Term

Der *Momentum Term* steigert in einem ersten Schritt die Lerngeschwindigkeit erheblich (richtige Outputs schon nach ~1700 Schritten anstatt nach ~2200 Schritten. Der nächste Sprung hat wiederum eine kleinere Wirkung, aber die Zielwerte werden mit guter Sicherheit gelernt.

5.2.3 Versteckte Neuronen (1. Schicht)

Die 1. versteckte Neuronenschicht scheint extrem wichtig zu sein. Bei nur 2 versteckten Neuronen in der 1. Schicht, schafft es das Netzwerk überhaupt nicht, die Zielwerte zu lernen sondern pendelt die ganze Lernperiode in der Mitte hin und her. Bei 4 versteckten Neuronen scheint das Netzwerk nach über 4000 Schritten in die richtige Richtung zu gehen, ob es aber wirklich die Zielwerte lernen kann müsste überprüft werden. Einzig bei 6 versteckten Neuronen in der 1. Schicht, scheint das Netzwerk in der Lage zu sein, die Zielwerte richtig zu lernen und das auch relativ schnell (< 2000 Schritte).

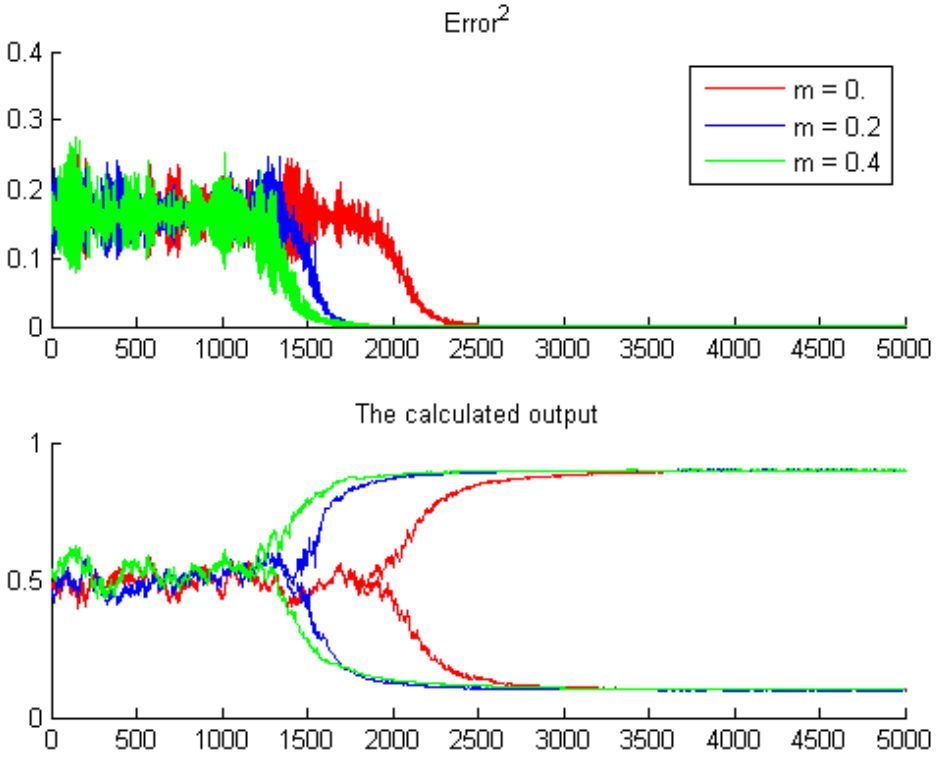


FIG. 5.9: Lernphase des Netzwerks mit verschiedenen m's

5.2.4 Versteckte Neuronen (2. Schicht)

Die zweite versteckte Schicht scheint nicht so anfällig wie die erste zu sein. Hier finden wir bei allen Werten innerhalb der Lernperiode, dass das Netzwerk die richtigen Zielwerte gelernt hat. Klar ist aber, dass es nur mit 2 versteckten Neuronen in der 2. Schicht mit Abstand am längsten dauert (über 4000 Schritte gegenüber $<$ 2500 mit 4 und mehr versteckten Neuronen).

5.3 Symbolerkennung mit drei möglichen Outputs

Als letztes habe ich noch versucht, das Netzwerk 3 anstatt von 2 Mustern erkennen zu lassen (ich habe auch noch 4 versucht, aber dazu fand ich kein passendes Set von Parametern, so dass das Netzwerk die Zielwerte richtig lernen konnte.

Zusätzlich zu den ersten beiden Mustern kommt hier noch ein drittes hinzu. Die ersten beiden Muster hatten Zielwerte 0.1 und 0.6. Das dritte Muster hat den Wert 0.6

Die Parameter für dieses Problem waren folgende:

- $\eta = 0.3$
- $m = 0.3$
- $hidnum = 24$

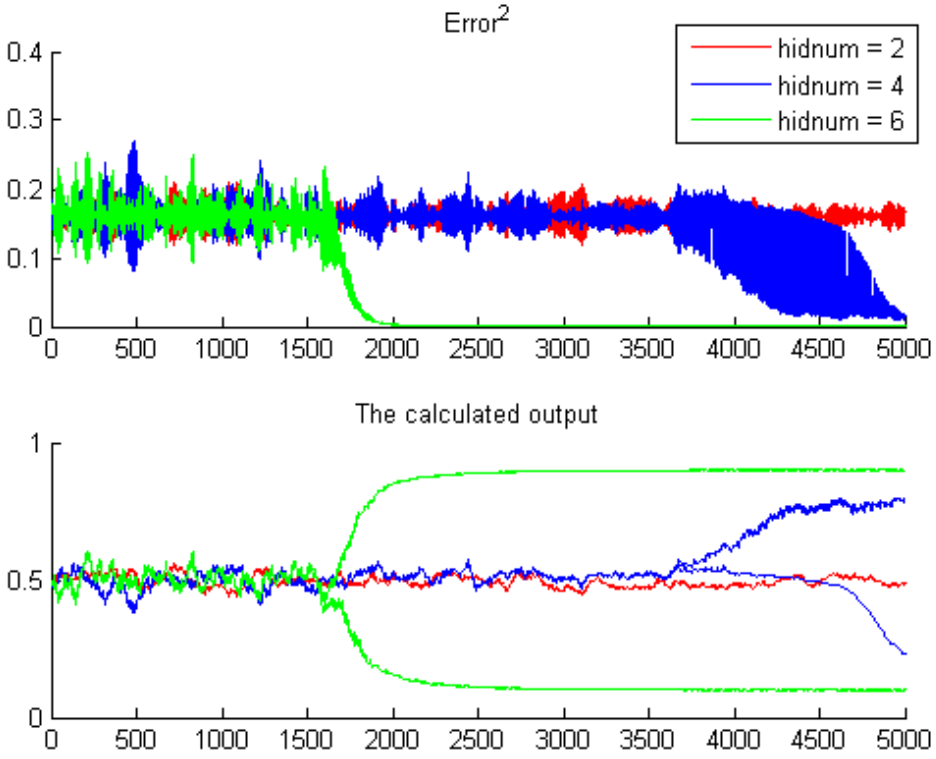


FIG. 5.10: Lernphase des Netzwerks mit verschiedenen *hidnum*'s

- $hidhidnum = 12$

5.4 Schlussfolgerungen

Wir haben hier nur jeweils die Leistung des Netzwerkes unter Veränderung jeweils eines Parameters angeschaut. Es ist aber durchaus denkbar, dass auf den ersten Blick schlechte Parameterwahl sich verbessern könnte, wenn man einen zweiten Parameter weiter anpasst.

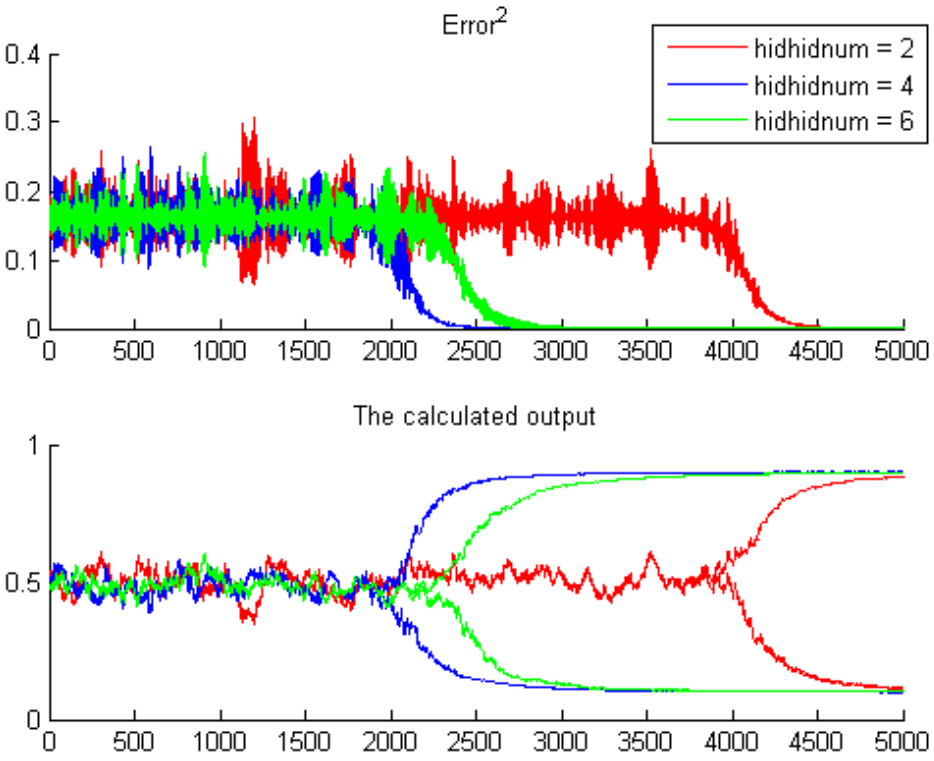


FIG. 5.11: Lernphase des Netzwerks mit verschiedenen *hidhidnum*'s

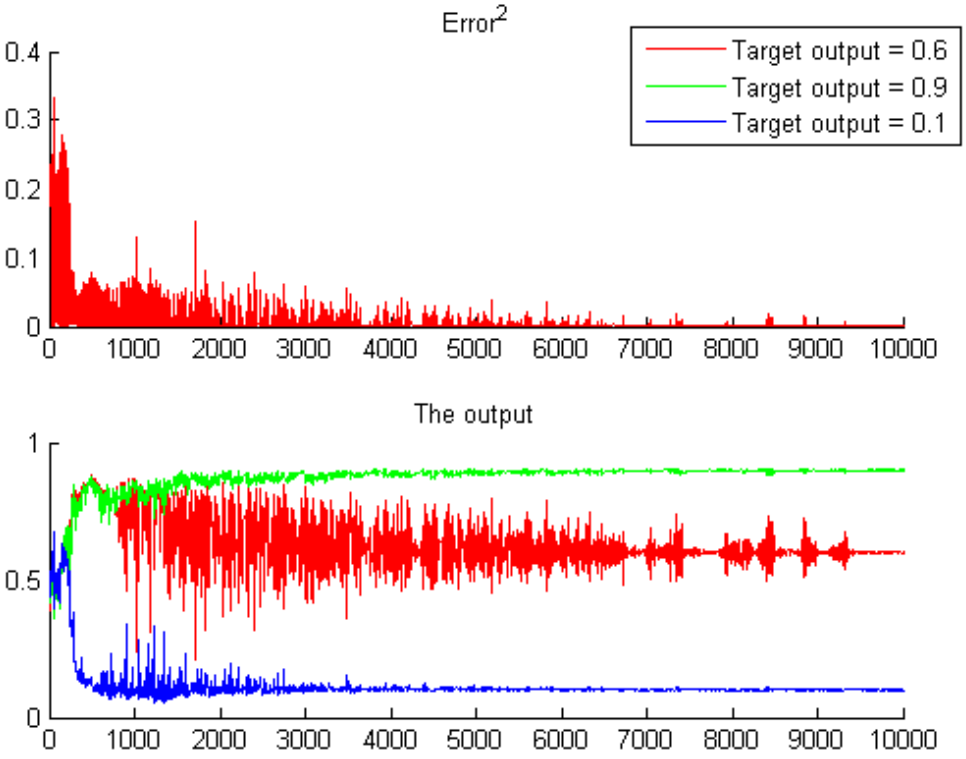


FIG. 5.12: Lernphase des Netzwerks mit verschiedenen *hidhidnum*'s

6 Code

6.1 Eine versteckte Schicht

```
unsigned int innum = 9, hidnum = 6, outnum = 1; // innum = number of inputs, hidnum = number of hidden neurons, outnum = number of outputs
double m = 0.6; //momentum term
matrix<double> wh (hidnum, innum), wo (outnum, hidnum), wh_old(hidnum, innum,0), wo_old(outnum, hidnum,0); //weight matrices for the hidden layer and the output layer
initWeights(wh); initWeights(wo);
double eta = 0.4; //learning rate
std::ofstream errorfile, outputfile;
errorfile.open ("set3_error.txt");
outputfile.open ("set3_output.txt");



for (unsigned int i = 0; i<5000; i++) {
   Paar* cur = paarList[Tools::returnRandomI(0,7)]; //randomly pick a input/output pair
   vector<double> xx = prod(wh,cur->input); //determine the output of the hidden layer
   vector<double> outhid(hidnum); // initialize vector for hidden layer output
   sigmoid(xx,outhid); // have to program this way because microsoft compiler complains if I pass a boost
   vector<double> out(outnum);
   vector<double> yy = prod(wo,outhid);
   sigmoid(yy,out); // out is usually of length 1 because the final output is just from one neuron
   vector<double> curout(1,cur->output);
   vector<double> e = vector<double>(curout - out); // e is also of length one
   vector<double> outdelta(1,e[0]*out[0]*(1-out[0]));
   vector<double> onevec(hidnum,1);
   vector<double> hiddelta = inner_prod(outhid,(onevec-outhid))*prod(trans(wo),outdelta);

   wo += eta*outer_prod(outdelta,outhid) + m*wo_old;       //momentum term added
   wh += eta*outer_prod(hiddelta,cur->input) + m*wh_old;   //momentum term added

   wo_old = eta*outer_prod(outdelta,outhid);   //save delta w for next iteration (momentum term)
   wh_old = eta*outer_prod(hiddelta,cur->input);

   double oout = out[0];
   errorfile  << "e^2:\t"  << inner_prod(e,e) << std::endl;
   outputfile <<  "out:\t" << oout     << "\t" << cur->output  << std::endl;

}
```

6.2 Zwei versteckte Schichten

```
unsigned int innum = 9, hidnum = 6, hidhidnum = 6, outnum = 1; // innum = number of inputs, hidnum = number of hidden neurons, outnum = number of outputs
   double m = 0.2; //momentum term
   matrix<double> wh (hidnum, hidhidnum), whh (hidhidnum, innum), wo (outnum, hidnum), wh_old(hidnum, hidhidnum,0), whh_old(hidhidnum, innum,0), wo_old(outnum, hidnum,0); //weight matrices for the hidden layer and the output layer
   initWeights(wh); initWeights(wo); initWeights(whh);
   double eta = 0.6, a = 1; //learning rate
   std::ofstream errorfile, outputfile;
   errorfile.open  ("set3_error.txt");
   outputfile.open ("set3_output.txt");


```

```
11
12    for (unsigned int i = 0; i<5000; i++) {
13       Paar* cur = paarList[Tools::returnRandomI(0,7)]; //randomly pick a input/output pair
14       vector<double> zz = prod(whh,cur->input);
15       vector<double> outhidhid(hidhidnum);
16       sigmoid(zz,outhidhid);
17       vector<double> xx = prod(wh,outhidhid); //determine the output of the hidden layer
18       vector<double> outhid(hidnum); // initialize vector for hidden layer output
19       sigmoid(xx,outhid); // have to program this way because microsoft compiler complains if I pass a ↩
             boost
20       vector<double> yy = prod(wo,outhid);
21       vector<double> out(outnum);
22       sigmoid(yy,out); // out is usually of length 1 because the final output is just from one neuron
23       vector<double> curout(1,cur->output);
24       vector<double> e = vector<double>(curout - out); // e is also of length one
25       vector<double> outdelta(1,e[0]*out[0]*(1-out[0]));
26       vector<double> onevech(hidnum,1);
27       vector<double> hiddelta = inner_prod(outhid,(onevech-outhid))*prod(trans(wo),outdelta);
28       vector<double> onevechh(hidhidnum,1);
29       vector<double> hidhiddelta = inner_prod(outhidhid,(onevechh-outhidhid))*prod(trans(wh),hiddelta);
30
31       wo += eta*outer_prod(outdelta,outhid) + m*wo_old;        //momentum term added
32       wh += eta*outer_prod(hiddelta,outhidhid) + m*wh_old;    //momentum term added
33       whh+= eta*outer_prod(hidhiddelta, cur->input) + m*whh_old;
34
35       wo_old = eta*outer_prod(outdelta,outhid);                    //save delta w for next iteration (↩
             momentum term)
36       wh_old = eta*outer_prod(hiddelta,outhidhid);
37       whh_old = eta*outer_prod(hidhiddelta, cur->input);
38
39       double oout = out[0];
40       errorfile  << "e^2:\t"  << inner_prod(e,e) << std::endl;
41       outputfile <<  "calc out:\t" << oout     << "\t" << cur->output  << std::endl;
42
43    }
```

6.3 Zwei versteckte Schichten mit 3 möglichen Outputs

```
1  void Tetris_2hlayers() {
2     int N = 9;
3     Paar p0(N), p1(N), p2(N), p3(N), p4(N), p5(N), p6(N), p7(N), p8(N), p9(N), p10(N), p11(N), p12(N), p13↩
          (N); //p0-3 are L's, p4-7 are Squares's, N is the length of the input, output length is always one
4
5
6     // intermediate vectors - easier to assign a list of values. they then get copied to boost vectors
7     std::vector<double> vvL0, vvL1, vvL2, vvL3, vvS0, vvS1, vvS2, vvS3, vvZ0, vvZ1, vvZ2, vvZ3, vvI0, vvI1↩
          ;
8     vvL0 += 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.9, 0.9, 0.1;
9     vvL1 += 0.1, 0.1, 0.1, 0.1, 0.1, 0.9, 0.9, 0.9, 0.9;
10    vvL2 += 0.1, 0.9, 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.9;
11    vvL3 += 0.9, 0.9, 0.9, 0.9, 0.1, 0.1, 0.1, 0.1, 0.1;
12
13    vvS0 += 0.1, 0.1, 0.1, 0.9, 0.9, 0.1, 0.9, 0.9, 0.1;
14    vvS1 += 0.1, 0.1, 0.1, 0.1, 0.9, 0.9, 0.1, 0.9, 0.9;
15    vvS2 += 0.1, 0.9, 0.9, 0.1, 0.9, 0.9, 0.1, 0.1, 0.1;
16    vvS3 += 0.9, 0.9, 0.1, 0.9, 0.9, 0.1, 0.1, 0.1, 0.1;
17
18    vvZ0 += 0.1, 0.9, 0.1, 0.9, 0.9, 0.1, 0.9, 0.1, 0.1;
19    vvZ1 += 0.1, 0.1, 0.1, 0.9, 0.9, 0.1, 0.1, 0.9, 0.9;
20    vvZ2 += 0.1, 0.1, 0.9, 0.1, 0.9, 0.9, 0.1, 0.9, 0.1;
21    vvZ3 += 0.9, 0.9, 0.1, 0.1, 0.9, 0.9, 0.1, 0.1, 0.1;
22
23    vvI0 += 0.1, 0.9, 0.1, 0.1, 0.9, 0.1, 0.1, 0.9, 0.1;
```

```
24    vvI1 += 0.1, 0.1, 0.1, 0.9, 0.9, 0.9, 0.1, 0.1, 0.1;
25
26    copyToBoost(vvI0,p12.input); p12.output = 0.6;
27    copyToBoost(vvI1,p13.input); p13.output = 0.6;
28
29    copyToBoost(vvZ0,p8.input); p8.output = 0.4;
30    copyToBoost(vvZ1,p9.input); p9.output = 0.4;
31    copyToBoost(vvZ2,p10.input); p10.output = 0.4;
32    copyToBoost(vvZ3,p11.input); p11.output = 0.4;
33
34    copyToBoost(vvL0,p0.input); p0.output = 0.1;
35    copyToBoost(vvL1,p1.input); p1.output = 0.1;
36    copyToBoost(vvL2,p2.input); p2.output = 0.1;
37    copyToBoost(vvL3,p3.input); p3.output = 0.1;
38
39    copyToBoost(vvS0,p4.input); p4.output = 0.9;
40    copyToBoost(vvS1,p5.input); p5.output = 0.9;
41    copyToBoost(vvS2,p6.input); p6.output = 0.9;
42    copyToBoost(vvS3,p7.input); p7.output = 0.9;
43
44
45    std::vector<Paar*> paarList;
46
47    paarList.push_back(&p0);
48    paarList.push_back(&p1);
49    paarList.push_back(&p2);
50    paarList.push_back(&p3);
51    paarList.push_back(&p4);
52    paarList.push_back(&p5);
53    paarList.push_back(&p6);
54    paarList.push_back(&p7);
55    /* paarList.push_back(&p8);
56    paarList.push_back(&p9);
57    paarList.push_back(&p10);
58    paarList.push_back(&p11); */
59    paarList.push_back(&p12);
60    paarList.push_back(&p13);
61
62    unsigned int innum = 9, hidnum = 24, hidhidnum = 12, outnum = 1; // innum = number of inputs, hidnum =↩
          number of hidden neurons, outnum = number of outputs
63    double m = 0.3; //momentum term
64    matrix<double> wh (hidnum, hidhidnum), whh (hidhidnum, innum), wo (outnum, hidnum), wh_old(hidnum, ↩
          hidhidnum,0), whh_old(hidhidnum, innum,0), wo_old(outnum, hidnum,0); //weight matrices for the ↩
          hidden layer and the output layer
65    initWeights(wh); initWeights(wo); initWeights(whh);
66    double eta = 0.3, a = 1; //learning rate
67    std::ofstream errorfile, outputfile;
68    errorfile.open  ("set1_error.txt");
69    outputfile.open ("set1_output.txt");
70
71
72
73    for (unsigned int i = 0; i<30000; i++) {
74       Paar* cur = paarList[Tools::returnRandomI(0,9)]; //randomly pick a input/output pair
75       vector<double> zz = prod(whh,cur->input);
76       vector<double> outhidhid(hidhidnum);
77       sigmoid(zz,outhidhid);
78       vector<double> xx = prod(wh,outhidhid); //determine the output of the hidden layer
79       vector<double> outhid(hidnum); // initialize vector for hidden layer output
80       sigmoid(xx,outhid); // have to program this way because microsoft compiler complains if I pass a ↩
             boost
81       vector<double> yy = prod(wo,outhid);
82       vector<double> out(outnum);
83       sigmoid(yy,out); // out is usually of length 1 because the final output is just from one neuron
84       vector<double> curout(1,cur->output);
85       vector<double> e = vector<double>(curout - out); // e is also of length one
86       vector<double> outdelta(1,e[0]*out[0]*(1-out[0]));
87       vector<double> onevech(hidnum,1);
```

```
88      vector<double> hiddelta = inner_prod(outhid,(onevech-outhid))*prod(trans(wo),outdelta);
89      vector<double> onevechh(hidhidnum,1);
90      vector<double> hidhiddelta = inner_prod(outhidhid,(onevechh-outhidhid))*prod(trans(wh),hiddelta);
91
92      wo += eta*outer_prod(outdelta,outhid) + m*wo_old;       //momentum term added
93      wh += eta*outer_prod(hiddelta,outhidhid) + m*wh_old;    //momentum term added
94      whh+= eta*outer_prod(hidhiddelta, cur->input) + m*whh_old;
95
96      wo_old = eta*outer_prod(outdelta,outhid);                   //save delta w for next iteration (↩
            momentum term)
97      wh_old = eta*outer_prod(hiddelta,outhidhid);
98      whh_old = eta*outer_prod(hidhiddelta, cur->input);
99
100     double oout = out[0];
101     errorfile  << "e^2:\t"  << inner_prod(e,e) << std::endl;
102     outputfile <<  "calc out:\t" << oout     << "\t" << cur->output  << std::endl;
103     cur = NULL;
104     delete cur;
105
106  }
```

6.4 Tools.h

```
1  /* A General library of often used functions such as random numbers, waiting for enter */
2
3  #include <random>
4
5  class Tools {
6
7  public:
8     static double returnRandomD(double start, double end) {
9        std::random_device rd;
10       std::mt19937 gen(rd());
11       std::uniform_real_distribution<> dis(start,end);
12       return dis(gen);
13    }
14
15    static int returnRandomI(int start, int end) {
16       std::random_device rd;
17       std::mt19937 gen(rd());
18       std::uniform_int_distribution<> dis(start,end);
19       return dis(gen);
20    }
21
22    static void PressEnterToContinue() {
23       int c;
24       printf( "Press ENTER to continue... " );
25       fflush( stdout );
26       do c = getchar(); while ((c != '\n') && (c != EOF));
27   }
28 };
```

6.5 Paar.h

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <boost/numeric/ublas/matrix.hpp>
4  #include <boost/numeric/ublas/vector.hpp>
5  #include <boost/numeric/ublas/io.hpp>
```

```
6  #include <fstream>
7
8  class Paar {
9  public:
10    Paar(int n) { input.resize(n);
11    }
12    boost::numeric::ublas::vector< double > input;
13    double  output;
14 private:
15
16 };
```

7 Referenzen

[1] wikipedia.org, "McCulloch-Pitts-Zelle." http://de.wikipedia.org/wiki/McCulloch-Pitts-Zelle.

[2] R. Rojas, *Neural Networks - A Systematic Introduction*. Springer.

[3] K. Kawaguchi, "A multithreaded software model for backpropagation neural network applications." http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html.

[4] G. Orr, "Momentum and Learning Rate Adaptation." http://www.willamette.edu/~gorr/classes/cs449/momrate.html, 1999.

[5] wikipedia.org, "Artificial neural network." http://commons.wikimedia.org/wiki/File:Neural_network.svg?uselang=de.