

# Mustererkennung: Übungsblatt 7

von

Naja v. Schmude (4127652), Lisa Dohrmann (4130066), Adrian Neumann (4140810)

## Aufgabe 1

### Code

Es ist die Aufgabe, die Ziffern mit Hilfe eines neuronalen Netzes zu Klassifizieren. Die Schwierigkeit besteht darin, das Netz so zu trainieren, dass es nicht einfach auswendig lernt und auch nicht zu schnell wieder vergisst. Zum Trainieren benutzen wir den Backpropagation-Algorithmus, den wir auf ein 3-schichtiges Netz anwenden.

Im wesentlichen berechnet uns der Algorithmus die Korrekturen aller Gewichte, damit der Fehler zum gesetzten Ziel minimal wird. Dies erledigt der Algorithmus in zwei Schritten:

**Feedforward** Ein Beispiel wird in das Netz geworfen und durchläuft es. Der Output kann dann mit dem Ziel verglichen werden (durch euklidischen Abstand), was zu dem Gesamtfehler führt.

**Backpropagation** Das Netz wird rückwärts betrieben, dadurch werden dann die Korrekturen ermittelt.

Wie wir in der Vorlesung gesehen haben, kann man diese Schritte nur mit Hilfe von Operationen auf Matrizen lösen. Das lässt sich dann natürlich auch sehr komfortabel in Code umsetzen:

```

1  private Matrix[] backprop(Digit d) {
2      Matrix[] deltas = new Matrix[2];
3
4      // Target
5      Matrix t = new Matrix(10, 1);
6      t.set(d.getNumber(), 0, 1);
7
8      // ##### Feedforward #####
9
10     // Output 1. o1 ist ein Zeilenvektor mit neuronsHiddenLayer
11     // Einträgen
12     // o11 ist ein Zeilenvektor mit neuronsHiddenLayer+1 Einträgen
13     Matrix o1 = sigmoidMatrix(d.digitToMatrix().transpose().times(w1));
14     Matrix o11 = new Matrix(1, neuronsHiddenLayer + 1);
15     o11.setMatrix(0, 0, 0, o1.getColumnDimension() - 1, o1);
16     o11.set(0, o1.getColumnDimension(), 1);
17
18     // Output 2. o2 ist ein Zeilenvektor mit 10 Einträgen
19     Matrix o2 = sigmoidMatrix(o11.times(w2));
20
21     // Ableitungen 1. d1 ist eine neuronHiddenLayer x neuronHiddenLayer
22     // Diagonalmatrix
23     Matrix d1 = new Matrix(neuronsHiddenLayer, neuronsHiddenLayer);
24     for (int i = 0; i < neuronsHiddenLayer; i++) {
25         double oi1 = o1.get(0, i);
26         d1.set(i, i, oi1 * (1 + oi1));
27     }
28
29     // Ableitungen 2. d2 ist eine 10 x 10 Diagonalmatrix
30     Matrix d2 = new Matrix(10, 10);
31     for (int i = 0; i < 10; i++) {
32         double oi2 = o2.get(0, i);
33         d2.set(i, i, oi2 * (1 + oi2));
34     }
35

```

```

36 // Fehler. Spaltenvektor mit 10 Einträgen
37 Matrix e = new Matrix(10, 1);
38 for (int i = 0; i < 10; i++) {
39     e.set(i, 0, Math.pow(o2.get(0, i) - t.get(i, 0), 2) / 2);
40 }
41 double error = 0;
42 for (int i = 0; i < 10; i++) {
43     error += e.get(i, 0);
44 }
45 // System.out.println("Fehler: " + error);
46 rb.add(error);
47
48 // Fehlerableitung. Spaltenvektor mit 10 Einträgen
49 Matrix epsilon = o2.transpose().minus(t);
50
51 // ##### Backpropagation #####
52 // w21 = w2 ohne letzte Zeile
53 Matrix w21 = w2.getMatrix(0, neuronsHiddenLayer - 1, 0, 9);
54 Matrix delta2 = d2.times(epsilon);
55 Matrix delta1 = d1.times(w21).times(delta2);
56 Matrix deltaW2 = delta2.times(o11).transpose();
57 Matrix deltaW1 = delta1.times(d.digitToMatrix().transpose())
58     .transpose();
59
60 deltas[0] = deltaW1;
61 deltas[1] = deltaW2;
62
63 return deltas;
64 }

```

Diese Methode berechnet also die Änderungen an den beiden Gewichtsmatrizen  $W_1$  und  $W_2$  für die Eingabe von  $d$ .  $d$  ist dabei schon um eine Dimension erweitert. Es werden folgenden Rechnungen durchgeführt:

$$o_1 = \sigma(d^T \cdot W_1) \text{ Ausgabe Hidden-Layer}$$

$o_{11}$  ist  $o_1$  mit zusätzlicher Dimension

$$o_2 = \sigma(o_{11} \cdot W_2) \text{ Ausgabe des Output-Layers}$$

Gleichzeitig werden die Ableitungen berechnet. Dabei enthält die Matrix  $d_1$  die Ableitung  $(o_1)' = o_1(1 - o_1)$  auf der Diagonalen für jeden Eintrag des Zeilenvektors  $o_1$ .  $d_2$  gilt analog, eben bloß für  $o_2$ . Jetzt muss noch der Fehler berechnet werden.  $e$  ist ein Spaltenvektor, in dessen Komponenten jeweils  $(o_2 - t)^2$  steht. Der Gesamtfehler **error** ist dann die Summe über alle Zeilen von  $e$ .

Für den Backpropagation-Schritt brauchen wir dann folgende Formeln:

$$\epsilon = o_2^T - t \text{ die Fehlerableitung}$$

$$\delta_2 = d_2 \cdot \epsilon$$

$$\delta_1 = d_1 \cdot W_2 \cdot \delta_2 \text{ wobei } W_2 \text{ die letzte Dimension fehlt}$$

Die Korrekturen ergeben sich dann als

$$\Delta W_1 = \delta_1 \cdot d^T$$

$$\Delta W_2 = \delta_2 \cdot o_{11}^T$$

Dabei wird allerdings hier noch nicht die Lernkonstante  $\lambda$  berücksichtigt, da es so einfacher ist, diese durch verschiedenen Verfahren (z.B. Silva & Almeida) nachträglich zu berechnen und einzubinden.

Zur Verarbeitung von die beiden Deltas, wird dann ein Array von Matrizen zurückgegeben, die an erster Stelle eben **deltaW1** und an zweiter Stelle **deltaW2** enthält.

Diese Methode berücksichtigt nun ja nur jeweils ein Datum, daher muss sie zum Trainieren mehrfach ausgeführt werden. Im folgenden zeigen wir nun die Einbettung des Backpropagation in ein Online- und ein Offline-Verfahren.

```

1 public void onlineTraining() {
2     Random rand = new Random();

```

```

3  double u = 1.001;
4  double d = 0.999;
5  boolean go = true;
6
7  // Matrizen der Gewichtsdelatas W1 und W2 der vorhergehenden Iteration
8  Matrix[] deltas1 = {
9      new Matrix(w1.getRowDimension(), w1.getColumnDimension()),
10     new Matrix(w2.getRowDimension(), w2.getColumnDimension()) };
11 // Matrizen der Gewichtsdelatas W1 und W2 der aktuellen Iteration
12 Matrix[] deltas = {
13     new Matrix(w1.getRowDimension(), w1.getColumnDimension()),
14     new Matrix(w2.getRowDimension(), w2.getColumnDimension()) };
15
16 // solange wir ncoh nciht gut genug sind ...
17 while (go) {
18     // Gammas initialisieren für W1 und W2
19     Matrix gamma1 = new Matrix(w1.getRowDimension(), w1
20         .getColumnDimension(), 0.01);
21     Matrix gamma2 = new Matrix(w2.getRowDimension(), w2
22         .getColumnDimension(), 0.01);
23
24     // zufällig 1000 Beispiele wählen und für die Backprop durchführen
25     // und dann korrigieren entsprechend dem aktuellen gamma
26     for (int n = 0; n < 1000; n++) {
27         Digit t = trainingsDigits.get(rand.nextInt(trainingsDigits
28             .size()));
29         deltas = backprop(t);
30
31         // Für alle W1_{i,j} gamma berechnen und multiplizieren
32         for (int i = 0; i < w1.getRowDimension(); i++) {
33             for (int j = 0; j < w1.getColumnDimension(); j++) {
34                 // Vorzeichen hat sich nicht verändert, also gamma
35                 // vergrößern
36                 if (deltas1[0].get(i, j) * deltas[0].get(i, j) > 0) {
37                     gamma1.set(i, j, gamma1.get(i, j) * u);
38                 }
39                 // Vorzeichen hat sich verändert, also gamma verkleinern
40                 else {
41                     gamma1.set(i, j, gamma1.get(i, j) * d);
42                 }
43                 deltas[0].set(i, j, deltas[0].get(i, j)
44                     * (-gamma1.get(i, j)));
45             }
46         }
47         // Für alle W2_{i,j} gamma berechnen und multiplizieren
48         for (int i = 0; i < w2.getRowDimension(); i++) {
49             for (int j = 0; j < w2.getColumnDimension(); j++) {
50                 // Vorzeichen hat sich nicht verändert, also gamma
51                 // vergrößern
52                 if (deltas1[1].get(i, j) * deltas[1].get(i, j) > 0) {
53                     gamma2.set(i, j, gamma2.get(i, j) * u);
54                 }
55                 // Vorzeichen hat sich verändert, also gamma verkleinern
56                 else {
57                     gamma2.set(i, j, gamma2.get(i, j) * d);
58                 }
59                 deltas[1].set(i, j, deltas[1].get(i, j)
60                     * (-gamma2.get(i, j)));
61             }
62         }
63
64         // Ws verändern
65         w1.plusEquals(deltas[0]);
66         w2.plusEquals(deltas[1]);
67         deltas1 = deltas;
68     }
69     // Ausgabe der durchschnittlichen Fehler ...
70     System.out.println(rb.avg());
71
72     go = false;
73     // gucken, ob wir genug schon richtig klassifizieren. Wenn ja, dann
74     // brechen wir ab

```

```

75     for (int i = 0; i < 50; i++) {
76         Digit t = trainingsDigits.get(rand.nextInt(trainingsDigits
77             .size()));
78         if (classify(t) != t.getNumber()) {
79             go = true;
80         }
81     }
82 }
83 }

```

Das Online-Verfahren nimmt sich zufällig Beispiele aus der Trainingsmenge, berechnet die Korrekturen mit **backprop** und multipliziert dann die Werte mit  $-\gamma$ . Dabei wird das  $\gamma$  mittels Silva & Almeida dynamisch korrigiert, weswegen die Methode auch sehr aufgebauscht wirkt. Für jede Stelle  $(i, j)$  der beiden Gewichtsmatrizen  $W_1$  und  $W_2$  muss nämlich ein eigener  $\gamma$ -Wert hinhalten, die wiederum in Matrizen **gamma1** und **gamma2** gehalten sind. Zusätzlich braucht man noch die Korrekturen vom vorhergehenden Durchlauf, um zu schauen, ob die Korrekturen noch in die selbe Richtung gehen, oder nicht. Wenn sie nämlich noch das selbe Vorzeichen haben, dann können wir unser  $\gamma$  mit  $u$  multiplizieren, wir machen also größere Schritte. Anderenfalls verkleinern wir es durch eine Multiplikation mit  $d$ . Nachdem wir alle Komponenten der Korrekturmatriizen mit den entsprechenden  $\gamma$ s multipliziert haben, können wir jetzt  $W_1$  und  $W_2$  korrigieren, in dem wir die Korrekturen drauf addieren. Das machen wir zunächst 1000 mal und klassifizieren dann 50 zufällig gewählte Ziffern, um zu sehen, ob wir schon gut genug trainiert haben. Wenn wir alle 50 richtig erkennen, brechen wir ab und sind fertig. Die Gewichtsmatrizen stehen dann fest in unserem Objekt und wir können was auch immer wir wollen klassifizieren.

Die Klassifizierung sieht übrigens so aus, dass man das Netz vorwärts betreibt, und anstatt einen Fehler zu berechnen, zu prüfen, an welchem Ausgang die Ausgabe am größten ist, und die Stelle zurück zuliefern.

```

1  public int classify(Digit d) {
2      // Output 1. o1 ist ein Zeilenvektor mit neuronsHiddenLayer
3      // Einträgen
4      // o11 ist ein Zeilenvektor mit neuronsHiddenLayer+1 Einträgen
5      Matrix o1 = sigmoidMatrix(d.digitToMatrix().transpose().times(w1));
6      Matrix o11 = new Matrix(1, w2.getRowDimension());
7      o11.setMatrix(0, 0, 0, o1.getColumnDimension() - 1, o1);
8      o11.set(0, o1.getColumnDimension(), 1);
9
10     // Output 2. o2 ist ein Zeilenvektor mit 10 Einträgen
11     Matrix o2 = sigmoidMatrix(o11.times(w2));
12     double[] o2A = o2.getColumnPackedCopy();
13     int max = 0;
14     double maxValue = Double.MIN_VALUE;
15     for (int i = 0; i < o2A.length; i++) {
16         if (o2A[i] > maxValue) {
17             maxValue = o2A[i];
18             max = i;
19         }
20     }
21     return max;
22 }

```

Das Offline-Verfahren funktioniert im Prinzip genauso, bloß dass wir zunächst für alle Trainingsdaten die Korrekturen aufsummieren (dabei wird jede Einzelkorrektur wie oben durch Silva & Almeida beeinflusst. Wenn wir dann alle Korrekturen beisammen haben, wird der ganze Batzen auf die Gewichte raufaddiert. Dann testen wir wieder, ob wir schon gut genug sind, wenn ja brechen wir ab.

```

1  public void offlineTraining() {
2      Random rand = new Random();
3      double u = 1.0002;
4      double d = 0.9998;
5      boolean go = true;
6
7      // Matrizen der Gewichtsdelatas W1 und W2 der vorhergehenden Iteration
8      Matrix[] deltas1 = {
9          new Matrix(w1.getRowDimension(), w1.getColumnDimension()),
10         new Matrix(w2.getRowDimension(), w2.getColumnDimension()) };

```

```

11 // Matrizen der Gewichtsdelatas W1 und W2 der aktuellen Iteration
12 Matrix[] deltas = {
13     new Matrix(w1.getRowDimension(), w1.getColumnDimension()),
14     new Matrix(w2.getRowDimension(), w2.getColumnDimension()) };
15 // Matrizen der Gewichtsdelatas W1 und W2 in der Summe über alle
16 // Iterationen
17 Matrix[] deltasSum = {
18     new Matrix(w1.getRowDimension(), w1.getColumnDimension()),
19     new Matrix(w2.getRowDimension(), w2.getColumnDimension()) };
20
21 Matrix gamma1 = new Matrix(w1.getRowDimension(), w1
22     .getColumnDimension(), 0.00001);
23 Matrix gamma2 = new Matrix(w2.getRowDimension(), w2
24     .getColumnDimension(), 0.00001);
25
26 // Solange wir noch nicht fertig sind
27 while (go) {
28     // für alle Trainingsdaten
29     for (Digit t : trainingsDigits) {
30         // Backprop durchführen
31         deltas = backprop(t);
32
33         // Gammas berechnen für W1 und mit deltas[0] multiplizieren
34         for (int i = 0; i < w1.getRowDimension(); i++) {
35             for (int j = 0; j < w1.getColumnDimension(); j++) {
36                 // Wir sind in der gleichen Richtung, gamma kann größer werden
37                 if (deltas1[0].get(i, j) * deltas[0].get(i, j) > 0) {
38                     gamma1.set(i, j, gamma1.get(i, j) * u);
39                 // Andere Richtung, gamma kleiner machen
40                 } else {
41                     gamma1.set(i, j, gamma1.get(i, j) * d);
42                 }
43
44                 deltas[0].set(i, j, deltas[0].get(i, j)
45                     * (-gamma1.get(i, j)));
46             }
47         }
48         // Gammas berechnen für W2 und mit deltas[1] multiplizieren
49         for (int i = 0; i < w2.getRowDimension(); i++) {
50             for (int j = 0; j < w2.getColumnDimension(); j++) {
51                 // Wir sind in der gleichen Richtung, gamma kann größer werden
52                 if (deltas1[1].get(i, j) * deltas[1].get(i, j) > 0) {
53                     gamma2.set(i, j, gamma2.get(i, j) * u);
54                 // Andere Richtung, gamma kleiner machen
55                 } else {
56                     gamma2.set(i, j, gamma2.get(i, j) * d);
57                 }
58                 deltas[1].set(i, j, deltas[1].get(i, j)
59                     * (-gamma2.get(i, j)));
60             }
61         }
62
63         // Deltas aufsummieren
64         deltasSum[0].plusEquals(deltas[0]);
65         deltasSum[1].plusEquals(deltas[1]);
66         deltas1 = deltas;
67     }
68     // Jetzt erst, nachdem alle Trainingsdaten verarbeitet, Gewichte
69     // verändern
70     w1.plusEquals(deltasSum[0]);
71     w2.plusEquals(deltasSum[1]);
72
73     System.out.println(rb.avg());
74
75     go = false;
76     // Gucken, ob wir schon genug richtig klassifizieren ...
77     for (int i = 0; i < 10; i++) {
78         Digit t = trainingsDigits.get(rand.nextInt(trainingsDigits
79             .size()));
80         if (classify(t) != t.getNumber()) {
81             go = true;
82         }
83     }

```

```

82 |         }
83 |     }
84 | }

```

## Auswertung

Bei anfänglichen  $\gamma$ s von 0,01 und  $u = 1,001$ ,  $d = 0,999$  kam man beim Online-Verfahren auf folgende Werte (gemittelt über 10 Durchläufe):

Anzahl Neuronen im Hidden-Layer	benötigte Iterationen, bis Abbruch	Erkennungsrate
15	467	80%
30	90	85%
45	89	86%

Die Anzahl der Iterationen gibt dabei die Anzahl der Durchläufe der **while**-Schleife in der Online-Verfahren-Methode an.

Die hohe Iterationsanzahl bei 15 Neuronen erklärt sich dadurch, dass man mit 15 Neuronen nur wesentlich weniger speichern und merken kann als mit mehr Neuronen. Daher braucht es hier wesentlich länger, um die Gewichte so gut zu trainieren, dass 50 erstmal richtig klassifiziert werden, bevor das Trainieren abbricht. Bei 30 und 45 Neuronen dauert es dann ungefähr gleichlang, von der Anzahl der Iterationen her. Natürlich ist das mit 30 wesentlich flotter, da hier die Matrizen kleiner sind. Und so viel mehr Erkennungsrate bringt das auch nicht.

Leider hat die Theorie für die Offline-Variante nicht hingehauen. Es tritt folgendes Problem auf, dass er anfangs noch ganz gut lernt und der Fehler stetig kleiner wird, bis er bei knapp 0,4 liegt. Dann wird er wieder größer und hält sich konstant um 0,5. Bei diesem Prozess ist auch zu beobachten, dass sich der Algorithmus immer mehr nur auf eine Ziffer spezialisiert, und dann immer, egal wie viel korrigiert wird, z.B. 9 ausspuckt. Auch durch einen Crossvalidation-Ansatz, bei dem nicht die ganze Trainingsmenge benutzt wird, traten gleiche Erscheinungen auf. Selbiges passierte auch, wenn man die  $\gamma$ s konstant lässt. Es scheint, als falle der Algorithmus dort jedes mal in ein lokales Minimum oder irgendwas merkwürdiges.