Wissensbasierte Systeme

Zusammenfassung v1.0

Kälin Thomas, Abteilung I
FS 08

1. EINFÜHRUNG IN DIE KÜNSTLICHE INTELLIGENZ 4
2. ANALYTISCH LÖSBARE OPTIMIERUNGSAUFGABEN 4
2.1. Die Rolle der Simulation und der Modelle 4
2.2. Grafische Methoden und Spieltheorie 4
2.2.1. Gerichtete Graphen / Ränge von Knoten (Sprage / Grundy) 4
2.2.2. NIM-Spiel 4
2.3. Matrizendarstellung 4
2.4. Eindimensionale Optimierung in R^n 5
2.5. Die Methode der Lagrange'schen Multiplikatoren 5
2.6. Mehrdimensionale Optimierung in R^n 5
2.6.1. Gradientenmethode 5
2.6.2. Gradientenabstieg 5
2.7. Lineare Optimierung 5
2.7.1. Simplex-Verfahren 5
2.8. Nichttriviale Lösungsmengen 5
3. NEURONALE NETZE 6
3.1. Biologische und technische Neuronale Netze 6
3.2. Biophysiknahe Neuronenmodellierung 6
3.2.1. Die Ionenpumpe 6
3.3. Klassische künstliche Neuronen und neuronale Netze 6
3.3.1. Outputfunktion 6
3.3.2. Die Rolle der Gewichte 6
3.3.3. Lernvorgang 7
3.3.4. Beispiel: Perzeptron lernt AND (ohne Feuerschwelle) 7
3.3.5. Grenzen des elementaren Neurons 7
3.4. Netze von Perzeptronen 7
3.4.1. Werbos-Konstruktion 7
3.4.2. Mehrschichtige Neuronen 8
3.4.3. Beispiel: Backpropagation-Netzwerk 8
3.4.4. Empfehlungen zu Neuronalen Netzen 9
3.5. Diskretes Hopfield Netzwerke 9
3.5.1. Probleme mit dem diskreten Hopfield Netzwerk 10
3.5.2. Beispiel: Diskretes Hopfield Netzwerk 10
3.6. Stochastisches diskretes Hopfield Netzwerk 10
3.6.1. Beispiel: Stochastisches diskretes Hopfield Netzwerk 10
3.7. Vergleich: Überwachtes und unüberwachtes Lernen 11
3.8. Selbstorganisierte Kohonennetze 11
3.8.1. Beispiel: Kohonen-Netz für das TS-Problem 11
4. GENETISCHE ALGORITHMEN 12
4.1. Genetische Algorithmen 12
4.1.1. Grundidee 12
4.1.2. Erweiterungen 12
4.1.3. Beispiel: Genetischer Algorithmus 13
4.2. Genetische Programmierung 14
4.2.1. Grundidee 14
4.2.2. Beispiel: Genetische Programmierung 14
5. CLUSTERING-GRUPPIERVERFAHREN 15
5.1. Klassen von Clusteringverfahren 16
5.2. Element-Distanzmasse 16
5.3. Klassen-Distanzmasse 16
5.3.1. Gebräuchliche Distanzmasse 16
5.4. Klassisches CV: K-Means 16
5.4.1. Beispiel: K-Means 16
5.5. Klassisches CV: Ward-Clustering 17
5.6. Autonome CV: Potts-Spin 17
5.6.1. Beispiel: Potts-Spin 18
5.7. Autonome CV: SSC-Clustering 18

6. VERSCHIEDENES 18
6.1. Travelling Salesman Problem (TSP) 18
6.2. Prüfungsfragen 19

1. EINFÜHRUNG IN DIE KÜNSTLICHE INTELLIGENZ

Bei den Anwendungsgebieten der KI handelt es sich um Themengebiete, die mit Methoden der ,,herkömmlichen" Informatik schlecht erschliessbar sind und daher bisher weit gehend dem Menschen vorbehalten bleiben.

BEISPIELE: Spiele spielen (Schach, Dame), Robotik (Steuerung), Expertensyssteme (Nachbildung von Spezialwissen), automatisches Programmieren, Sprache verstehen, Wahrnehmung (Nachbildung)

2. ANALYTISCH LÖSBARE OPTIMIERUNGSAUFGABEN

2.1. Die Rolle der Simulation und der Modelle

- Wir haben typischerweise ein Objekt oder Phänomen, das es zu beschreiben gilt. Dazu bildet man ein Modell des Systems. Für jedes Phänomen gibt es dabei eine optimale Beschreibungsebene. Weil jedes Phänomen mit jedem anderen verbunden ist, kann allerdings das gesamte System nie vollumfänglich beschrieben werden (Gödel). Was man dabei wegglässt, fasst man i.A. als Rauschen auf.
- Ein Modell ist gut, wenn es bei seiner Simulation das Phänomen bis auf ein kleines Rauschen zu reproduzieren vermag.
- Die Simulation trägt zum Verständnis bei, da sie beim Erstellen einer Hyptose hilft.
- Objekt -> Modell -> Simulation (+ Rauschen)

2.2. Grafische Methoden und Spieltheorie

2.2.1. Gerichtete Graphen / Ränge von Knoten (Sprage / Grundy)

- Sprague und Grundy leiteten ein system. Verfahren zur Bestimmung der Gewinnpositionen her.
- Ein gerichteter Graph ist ein Gebilde bestehend aus Knoten und gerichteten Kanten (Pfeile), welche von Knoten zu Knoten führen. In unserem Zusammenhang sind Pfeile Züge und Knoten Spielstände.
- Der Rang eines Knotens ist die kleinste Zahl aus N0, die nicht bereits durch den Rang eines direkten Nachfolgers besetzt ist. Endknoten (=Verlierpositionen) haben den Rang 0.

4	5	3	2	7
3	4	5	6	2
2	0	1	5	3
1	2	0	4	5
0	1	2	3	4

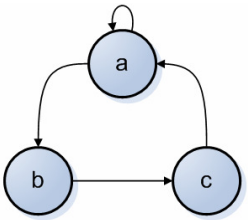
- Beispiel: „Treib die Dame in die Ecke"
- Wer nicht mehr ziehen kann, hat verloren
- Horizontale / Vertikale / Diagonale-Züge erlaubt
- Wenn ich auf einer 0 bin, kann ich niemals auf eine weitere 0 ziehen.
- Es gibt immer einen Zug auf eine 0.

2.2.2. NIM-Spiel

- Bekanntes Spiel: Wer das letzte Streichholz nehmen muss hat verloren. Dieses Spiel ist äquivalent zum Damenspiel, kann also bei 2 Haufen mit einer 2D-Darstellung aufgezeichnet werden.
- Verallgemeinerung mit N-Haufen: Wir befinden uns genau dann auf einer Gewinnposition, wenn die NIM-Summe (=XOR-Bildung) aller Haufen 0 ergibt. Es gibt stets einen legalen Zug aus einer Verliererposition in eine Gewinn-Position.

```
H1     13 Hölzer        1101            H1      4 Hölzer        0100
H2     12 Hölzer        1100            H2     12 Hölzer        1100
H3      8 Hölzer        1000            H3      8 Hölzer        1000
                        ----                                    ----
NIM-Summe               1001 = 9                                0000 = 0
```

2.3. Matrizendarstellung



$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

- Matrizen können auf effiziente Weise Spiele auf Z^n kodieren.
- Eine reguläre Matrix erlaubt das erreichen aller Knoten.
- Eine Adjazenzmatrix hat nur Einträge mit 0 und 1.
- Eine symmetrische Matrix ist ungerichtet.
- Zyklen sind nicht erwünscht (Endlosschleifen)

2.4. Eindimensionale Optimierung in R^n

- **DIFFERENTIALRECHNUNG:** Ableitung der Funktion muss 0 ergeben (`f'(x) = 0`).
- **NUMERISCHER ANSATZ:** wir gehen die Funktion hinunter, bis sie wieder ansteigt.

2.5. Die Methode der Lagrange'schen Multiplikatoren

- Hierbei wird ein zweidimensionales Problem gelöst. Wir besitzen dabei eine zu maximierende Funktion (=`f`) sowie die Nebenbedingung (=`g`).
- Aus diesen Beiden funktionen leitet man die Funktion `F(x,y) = f + λg` her.
- Ziel: Reduzierung des Rechenaufwands durch Gleichungssysteme (anstatt Gleichungen).

```
Beispiel:    f(x,y) = x^2 + y^2          g(x) = 3x-5

             1) F(x,y) = f + λg = x^2 + y^2 + λ(3x-5)
             2) 3x Ableiten von F (nach x, y und λ)
             3) Entstandenes Gleichungssystem aus Ableitungen lösen
```

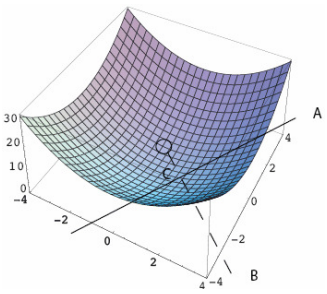
2.6. Mehrdimensionale Optimierung in R^n

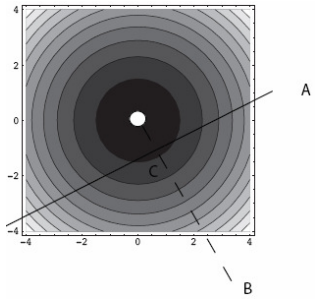
2.6.1. Gradientenmethode

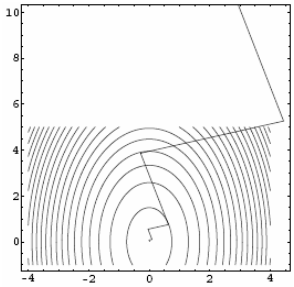
(kapier ich nicht, ist aber wohl auch überflüssig… ☺)

2.6.2. Gradientenabstieg

- In diesem Fall ist die Ursprungsfunktion nicht als Ganzes bekannt. Wir konzentrieren uns deshalb auf lokale Optimierungen.
- Das Verfahren kann so beschrieben werden: wir beginnen an einem Startpunkt (=A) und bewegen uns solange in die Richtung mit dem grössten Abstieg, bis der Punkt mit dem kleinsten Abstand (=B) erreicht wurde. Dieser wird im nächsten Schritt wieder als Anfangspunkt genommen.
- Abbruch sobald der Abstand kleiner als ein vorgegebener Wert ist.
- Problematisch sind Situationen, bei denen mehr als ein Minimum vorhanden sind („Start auf Berggipfel"). Oder man befindet sich bereits in einem Tal (aus diesem kommen wir nicht mehr raus, da immer nur nach Abstiegen gesucht wird!).

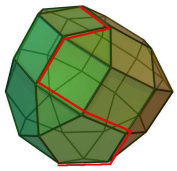






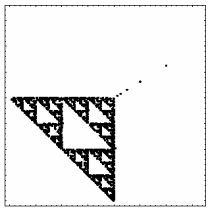
2.7. Lineare Optimierung

2.7.1. Simplex-Verfahren



Ein Simplex ist ein mehrdimensionales Polygon mit beliebigen Flächen und Kanten. Auf dem Simplex wird in optimaler Folge die Lösung optimiert, indem man sich längs der Kanten in Richtung der zu optimierenden Grösse bewegt (selbes Verfahren wie beim Gradientenabstieg!). Die optimale Lösung liegt auf einer Ecke.

2.8. Nichttriviale Lösungsmengen



Die Implementationen des Chaos-Spiels zeigen, dass man, anstatt ein einziges Optimum zu finden, sich bei diesem Spiel auf einem seltsamen Objekt als Spielmenge/Spiellösung bewegt, und nie zur Ruhe kommt. In diesem Sinne ist die asymptotische Struktur die Lösungsmenge des Problems.

3. NEURONALE NETZE

3.1. Biologische und technische Neuronale Netze

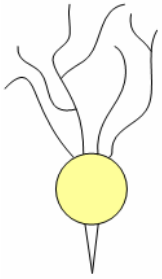
- Biologische NN unterliegen dem Prinzip der Evolution. Die Elemente des Hirns sind stark rauschbehaftet und arbeiten stark parallel.
- Das „Programm" des Hirns wird langsam aber ständig umgeschrieben und erneuert, indem Verbindungen auf- und abgebaut, bzw. verstärkt und geschwächt werden.
- Die einzelnen berechnenden Elemente des Hirns werden **NEURONEN** genannt. Es gibt anregende und hemmende Neuronen (im Verhältnis 0.8 / 1.0)

	COMPUTER	KORTEX
PROZESSOR	10^10 Hz	10^2Hz
SIGNALGESCHWINDIGKEIT	10^8 m/s	1 m/s
ANZAHL PROZESSOREN	1	10^11
OPERATION	Sequenziell	Parallel
STRUKTUR	Programm & Daten	Verbindungen & Neuronenform
PROGRAMMIERUNG	Extern	Selbstprogrammierend

3.2. Biophysiknahe Neuronenmodellierung

- Viele Biologen gehen oft von Poisson-Verteilten Feuerereignissen aus. Dies ist aber eine Vereinfachung. In der Natur zeigen eine Vielzahl von Neuronen allgemeinere Feuerstatistiken, die üblicherweise mit „Heaviside Schwellenfunktionen" beschrieben werden.

3.2.1. Die Ionenpumpe

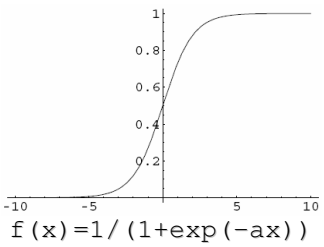


- Die **DENDRITEN** (Antennen) empfangen Signale von anderen Neuronen
- In der **SOMA** (Zellkern) werden die eintreffenden synaptischen Beiträge summiert. Sind diese gross genug, so wird ein Feuerereignis (Spike) erzeugt.
- Über das **AXON** wird dieser Ausschlag an andere angeschlossene Neuronen weitergeleitet.
- **HODGKIN UND HUXLEY** haben 1963 den Nobelpreis in Biologie für ein elektrisches Modell der Spikeerzeugung erhalten. Dieses Modell zeigte, dass sich Neuronen auf einer Grenzzykluslösung befinden. Nach dem Aufladevorgang wird ab einer bestimmten Schwelle „gefeuert".

3.3. Klassische künstliche Neuronen und neuronale Netze

- Künstliche neuronale Netze sind eine sehr starke Vereinfachung von biologischen neuronalen Netzen.
- Ein Neuron hat viele Inputs, im Allgemeinen aber nur einen Output. Der Gesamtinput zu einem Neuron wird durch eine gewichtete Summierung der einzelnen Eingänge von angenommen.
- Der Entladevorgang wird mit guter Genauigkeit als Exponentialabstieg beschrieben.
- Neuronen können sich selber „aufladen", also Synapsen (Verbindungen) auf sich selber haben.

3.3.1. Outputfunktion



- Die Outputfunktion beschreibt, wie der Zellkörper nichtlinear auf die Aufladung reagiert. Im Normalfall wird an Stelle der Heaviside Funktion eine **SIGMOIDFUNKTION** gewählt (siehe Bild), weil diese besonders einfach differenzierbar ist.
- Der Faktor a in der Sigmoidfunktion beschreibt die Steigung der Sigmoidfunktion (grosses a = grosse Steigung)
- Üblicherweise wird zur bestimmung der aktuellen Ladung über alle Eingänge summiert. Dabei wird der interne Verlust des Neutrons als 0. Eingang (negativ) gezählt, um die Summierung zu vereinfachen.

3.3.2. Die Rolle der Gewichte

- Jeder Input des Neurons hat eine eigene Gewichtung. Diese Gewichtung bestimmt, wie „gut" der betreffende Eingang ist.
- Mittels Skalarprodukt zwischen Eingangswerten (=Muster zur Erkennung) und den Gewichten (=Aktuell gelerntes Muster) wird ein Wert errechnet, welcher anschliessend zur Korrektur der Werte verwendet wird.

- Die Gewichte schwanken zwischen den Werten +1 (anregend) und -1 (hemmend) und werden beim Lernvorgang angepasst (siehe nächstes Kapitel).

3.3.3. Lernvorgang

- Lernen bedeutet das Finden von optimalen Gewichten.
- Lokales Optimieren (einzelne Gewichte anpassen) erzeugt bessere Resultate als globales Optimieren (alle Gewichte gleichzeitig anpassen).
- Führt die Aktivität einer Synapse zum Feuern eines Zielneurons, so wird dieser Eingang verstärkt, also dessen Gewicht erhöht.
- Wir betrachten beim **ÜBERWACHTEN LERNEN** den zeitdiskreten Fall, wa dem Gradientenabstiegs-verfahren entspricht (Annäherung an Optimum). Der Lernvorgang sieht schematisch wie folgt aus:

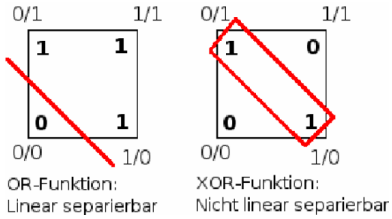
```
y = w.in;          Voraussage = Gewichte.Inputs
e = t-y;           Fehler = Vorgabe - Voraussage
w += n*e.in;       Gewichte += Korrekturfaktor*Fehler.Inputs
```

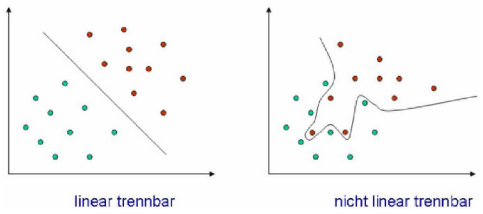
3.3.4. Beispiel: Perzeptron lernt AND (ohne Feuerschwelle)

```
w=Table[Random[],{i,1,2}];                              //Zufällige Gewichte
eta=0.0005;                                             //Korrekturfaktor
k=1;                                                    //Laufindex
ioPaar={{{0,0},0},{{0,1},0},{{1,0},0},{{1,1},1}};      //UND-Funktion

Errorliste=Table[
 {in, t}=ioPaar[[Random[Integer,{1,4}]]];               //Zufälliger Input
 y=w.in;                                                //Voraussage
 e=t-y;                                                 //Fehler berechnen
 w+=eta e in;                                           //Korrektur Gewichte
 k++;
 {e,w},{i,1,50000}];                                    //Werte archivieren

Do[Print[w.ioPaar[[i,1]]],{i,1,4}]                      //Resultate prüfen
0.
0.335786
0.328558
0.664344
```

3.3.5. Grenzen des elementaren Neurons

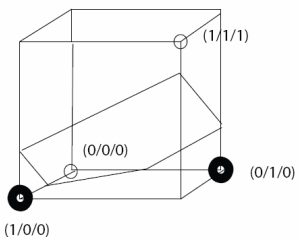
- Das Perzeptron kann die AND-Funktion lernen, nicht aber die XOR-Funktion. Das liegt daran, dass die obige Summierung (`w1*i1+w2*i2-`Φ) eine Geradenfunktion definiert, wobei Φ (= interner Verlust) einfach ein vertikaler Verschiebungsfaktor ist.
- Diese Geradenfunktion entspricht gleichzeitig auch der Feuerschwelle. Da eine Geradenfunktion nur lineare separierbare Probleme löst (z.B. die AND-Funktion), scheitert das Perzeptron an der XOR-Funktion.





3.4. Netze von Perzeptronen

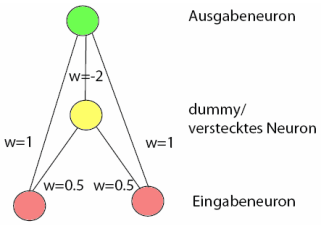
3.4.1. Werbos-Konstruktion

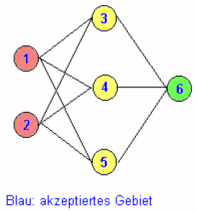


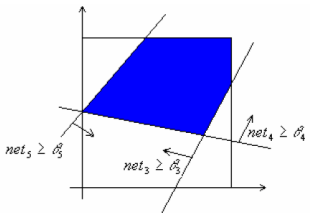
- Die Lösung des Problems besteht darin, Netze von Perzeptronen zu bilden. Es ist eine alte Beobachtung, dass Probleme in einer höheren Dimension einfacher werden.
- Führt man einen von den bestehenden Eingängen abhängigen Dummy-Input ein, so entsteht aus der 2D-Trennung ein dreidimensionaler Raum: `i3 = i1 * i2`.
- Mit diesem 3. Eingang kann nun auch XOR gelernt werden!

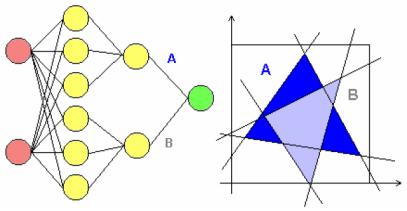
3.4.2. Mehrschichtige Neuronen

- In der Werbos-Konstruktion entspricht das Dummy-Neuron einer „versteckten" Schicht (ohne direkten Input).
- Mehr als zwei versteckte Schichten bringen nichts Neues, da über zwei verschiedene Schichten beliebig komplexe Feuerschwellen („Feuerflächen") definiert werden können.









3.4.3. Beispiel: Backpropagation-Netzwerk

- Der Lernvorgang, bzw. die Fehlerrückverfolgung, bei mehrschichtigen NN wird Backpropagation-Algoritmus genannt.
- Nachfolgendes Beispiel zeigt ein Netzwerk, welches die Buchstaben T und C voneinander unterscheiden sollen. Dabei wird ein Netzwerk mit zwei verborgenen Schichten verwendet.

```
1) Definition der Buchstaben T und C (Vorgabe des Lehrers)
ioPaar={     {{0.9,0.9,0.9,0.9,0.1,0.1,0.9,0.9,0.9},{0.1}},
             {{0.9,0.9,0.9,0.1,0.9,0.1,0.1,0.9,0.1},{0.9}},
             … weitere Buchstaben aus Platzgründen weggelassen …
             {{0.9,0.1,0.1,0.9,0.9,0.9,0.9,0.1,0.1},{0.9}}};

2) Initialisierung aller nötigen Werte
iter = 50000;                    //50'000 Iterationen für Lernvorgang
inzahl = 9;                 //Input-Neuronen
hidhidzahl = 3;             //Neuronen in 2.versteckter Schicht
hidzahl = 3;                //Neuronen in 1.versteckter Schicht
outzahl = 1;                //Neuronen auf Ausgabeebene

3) Gewichts-Tabellen erstellen (zufällige Werte)
wh  = Table[Table[Random[Real,{-0.1,0.1}],{hidhidzahl}],{hidzahl}];
whh = Table[Table[Random[Real,{-0.1,0.1}],{inzahl}],{hidhidzahl}];
wo  = Table[Table[Random[Real,{-0.1,0.1}],{hidzahl}],{outzahl}];

4) Ausgabefunktion definieren
sigmoid[x_]:=1/(1+Exp[-a x]);          //Ausgabefunktion (Sigmoid)
a = 0.5;                               //Steilheit der Sigmoid-Funktion
eta = 0.5;                             //Korrekturfaktor

5) Lernvorgang mit Backpropagation
Errorliste = Table[
 {in, t} = ioPaar[[Random[Integer, {1, 8}]]];    //Zufälliger Buchstaben

 outhidhid = sigmoid[whh.in];          //Ausgabe aus 1. versteckter Schicht
 outhid = sigmoid[wh.outhidhid];       //Ausgabe aus 2. versteckter Schicht
 out = sigmoid[wo.outhid];             //Ausgabe aus Ausgabeschicht

 e = t - out;                          //Fehler berechnen (Vorgabe - Ausgabe)
 outdelta = e out(1 - out);            //Fehler der Ausgabeschicht bestimmen
 hiddelta = outhid(1 - outhid) Transpose[wo].outdelta; //Fehler der 1.v.S.
 hidhiddelta = outhidhid(1 - outhidhid) Transpose[wh].hiddelta;   //F.2.v.S

 wo += eta Outer[Times, outdelta, outhid];       //Gewichte korrigieren
 wh += eta Outer[Times, hiddelta, outhidhid];    //Gewichte korrigieren
 whh += eta Outer[Times, hidhiddelta, in];       //Gewichte korrigieren

 {e.e, wo, wh}, {k, 1, iter}];        //Werte merken und iterieren
```

```
6) Unterscheidungsfunktion (Unterscheidung der Ausgabe-Werte)
Ans={"C", "none","T"};
decision[x_] := Print[Ans[[Which[(y=x/.x->x[[1]])<0.2,2,
 (y=x/.x->x[[1]])>0.2&& (y=x/.x->x[[1]])\[LessEqual]0.4,
 aa=1, (y=x/.x->x[[1]])>0.4&& (y=x/.x->x[[1]])<0.6,
 aa=2, (y=x/.x->x[[1]])>0.6 && (y=x/.x->x[[1]])\[LessEqual]0.8,
 aa=3, (y=x/.x->x[[1]])>0.8,2]]]]

7) Netzwerk testen mit allen Vorgaben
Do[
 in=ioPaar[[i,1]];                    //Buchstabe nehmen
 outhidhid=sigmoid[whh.in];           //Ausgabe aus 1. versteckter Schicht
 outhid=sigmoid[wh.outhidhid];        //Ausgabe aus 2. versteckter Schicht
 out=sigmoid[wo.outhid];              //Ausgabe aus Ausgabeschicht
 decision[out],                       //Unterscheidung der Ausgabe (T || C?)
 {i,1,8}]
```

3.4.4. Empfehlungen zu Neuronalen Netzen

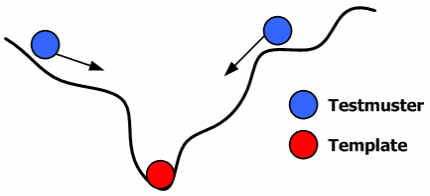
- Bei zu vielen versteckten Neuronen bekämpfen diese sich gegenseitig, das Netz konvergiert folglich sehr schlecht oder gar nicht.
- Nach einem erfolgreichen Lernvorgang sollten die Anzahl Neuronen in den versteckten Schichten stufenweise reduziert werden („Backward Methode").
- Ein zu **HOHES ETA** (Korrekturfaktor, Lernrate) kann auch zu Nicht-Konvergenz führen. Ein **KLEINES ETA** funktioniert zwar, allerdings kann es ziemlich lange dauern, bis das Netz konvergiert.
- Zu wenige Schichten führt zu einem grossen Trainingsfehler, zu viele Schichten zu einem grossen Verallgemeinerungsfehler

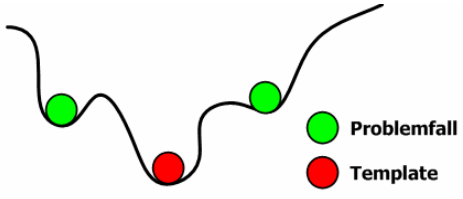
3.5. Diskretes Hopfield Netzwerke

- Bis anhin hatten wir immer die richtige Antwort vorgegeben und diese beim Lernverfahren berücksichtig (**ÜBERWACHTES LERNEN**). Bei den Netzwerken in diesem Kapitel ist kein „Lehrer" vorhanden, weshalb von **UNÜBERWACHTEM LERNEN** gesprochen wird. Man nennt diese Netzwerke auch **PROBABILISTISCHE NETZWERKE**.
- Ein Hopfield-Netzwerk ist ein Beispiel für ein **ASSOZIATIVES NETZWERK**. In wird mit sog. Prototypen („Templates") gearbeitet, wobei jedes Template durch seine **SELBSTKORRELATION** charakterisiert wird.
- Diese Selbstkorrelation wird durch das direkte Produkt eines Templates mit sich selber (Kreuzprodukt) und einer anschliessend Summation über die daraus entstandene Matrix bestimmt. Nachfolgendes Beispiel soll dies verdeutlichen:

```
Template = { { 1, -1,  1,  1},
             { 1,  1, -1, -1} }
Matrix = { {  2,  0,  0,  0},  //1*1+1*1 , 1*-1+1*1, 1*1+1*-1, 1*1+1*-1
           {  0,  2, -2, -2},  //-1*1+1*1, -1*-1+1*1, …
           {  0, -2,  2,  2},
           {  0, -2,  2,  2} }
```

- Die **KORRELATIONSMATRIX** ist immer symmetrisch. Auf der Symmetrieachse wird immer der maximale Wert erreicht, da jedes Template mit sich selber optimal korreliert.
- Eine eigentliche **LERNPHASE** ist nicht vorhanden, die diese Korrelationsmatrix direkt aus den vorgegebenen Templates (Zielmuster) berechnet werden kann.
- Beim **BESTIMMEN DES ERGEBNIS** wird das eingegebene Muster iterativ solange angepasst, bis es mit einem der vorgegebenen Templates übereinspricht. Man kann sich das bildlich so vorstellen, dass eine Kugel in eine Hügellandschaft geworfen wird und solange zum Weiterrollen animiert wird, bis einer der Zielpunkte erreicht wurde.





3.5.1. Probleme mit dem diskreten Hopfield Netzwerk

- Es kann sein, dass wir in einem lokalen Minimum steckenbleiben ohne das eigentliche Minimum zu erreichen. Die Lösung für dieses Problem ist das stochastische Hopfield Netzwerk (nächstes Kapitel).
- Es kann sein, dass das Netz nicht konvergiert und ständig zwischen zwei Zuständen hin und her pendelt.
- Mit jedem Muster wird auch das genau gegenteilige Muster erkannt.

3.5.2. Beispiel: Diskretes Hopfield Netzwerk

```
1) Templates erzeugen (Zielmuster)
Trainingsmuster = Table[Table[Random[Integer, {0, 1}], {10}], {3}];

2) Korrelationsmatrix bestimmen
w = Apply[Plus, Map[Outer[Times, #, #] &, Trainingsmuster]];

3) Energiefunktion festlegen (Misst die Korrelation mit Matrix)
EnergieHop[x_, w_] := -0.5 x.w.x;

4) Ausgabefunktion festlegen (Ändert die Eingabe in Richtung der Templates)
OutFunktion[x_]:=Which[x<0,-1,x>0,1,x\[Equal]0,0]
SetAttributes[OutFunktion,Listable]

5) Testmuster vorgeben und in Richtung Templates anpassen
x={0,0,1,-1,-1,-8,-1,-1,-1,-1};      //Testmuster, wird angepasst
Do[y=OutFunktion[w.x];Print[y," ",EnergieHop[y,w]];x=y,{i,1,10}]
{1,-1,-1,1,0,-1,-1,1,-1,-1} -28.
{1,-1,-1,1,-1,-1,1,1,-1,-1} -39.     //Übereinstimmung bereits erreicht
{1,-1,-1,1,-1,-1,1,1,-1,-1} -39.

6) Suchen des passenden Musters
Trainingsmuster.x
{2,10,2}                             //Muster 2 (10 Elemente identisch)
```

3.6. Stochastisches diskretes Hopfield Netzwerk

- Wie bei den „Problemen" genannt wurde kann es beim diskreten Hopfield Netzwerk passieren, dass wir in einem lokalen Minimum stecken bleiben.
- Das stochastische DPN verwendet als Lösung für dieses Problem einen Zufallsfaktor. Analog zur **TEMPERATUR** werden auch hier zu beginn bei „hohen Temperaturen" starke zufällige Störungen („Rütteln") simuliert, wodurch die sitzengebliebenen Muster aus dem Tal springen sollen. Mit fortlaufender Dauer „kühlt" die Temperatur ab – es wird nur noch wenig gerüttelt.
- Es werden ausserdem bei der Ausgabefunktion nicht mehr länger alle Stellen geändert (**SYNCHRON**), sondern immer nur noch eine (**ASYNCHRON**).

3.6.1. Beispiel: Stochastisches diskretes Hopfield Netzwerk

```
1) Templates erzeugen (Zielmuster)
Trainingsmuster= 2Table[Table[Random[Integer,{0,1}],{10}],{3}]-1;

2) Korrelationsmatrix bestimmen
W = Apply[Plus,Map[Outer[Times,#,#]&, Trainingsmuster]];

3) Asynchrone Ausgabefunktion mit Temperatur-Einfluss definieren
OutFunktion[x_] := Which[
 z = Random[];
 z < 1/(1 + Exp[-net/beta]), 1,
 z >= 1/(1 + Exp[-net/beta]),
 Which[net > 0, 1, net < 0, -1, net == 0, Invector[[indx]]]]

4) Startwerte festlegen
anzruns = 100;                       //100 Durchläufe
anzneuronen = 10;                    //10 Stellen im Trainingsmuster
beta = 50;                           //Starttemperatur
Invector= {-1,-1,1,1,1,1,1,1,1,1};   //Testmuster
```

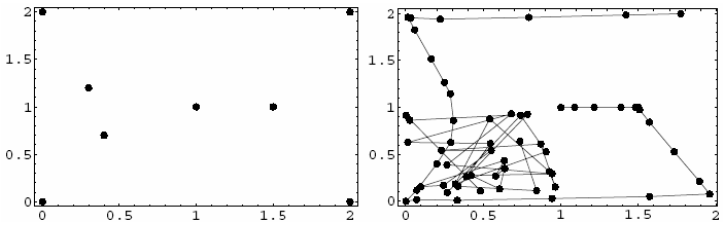
```
5) Testmuster in Richtung Templates anpassen (inkl. Abkühlung)
Do[
 beta-=5;                 //Abkühlung simulieren
                          //Alternative: beta -= betatot/(anztemp + 1);

 Do[
       indx=Random[Integer,{1,anzneuronen}];     //Zufällige Position
       net=Invector.w[[indx]];
       Invector[[indx]] = OutFunktion[Invector[[indx]]];
 Print[Invector],{j,1,anzruns}],
{i,1,9}]
```

3.7. Vergleich: Überwachtes und unüberwachtes Lernen

	ÜBERWACHTES LERNEN	**UNÜBERWACHTES LERNEN**
STRATEGIE	Finden von optimalen Gewichten	Testmuster an Template anpassen
LERNPHASE	Grosse Loop (Gewichte finden)	Kleine Loop (Korrelations-Matrix)
ERGEBNIS BESTIMMEN	Kleine Loop (Durchrechnen)	Grosse Loop (Template suchen)

3.8. Selbstorganisierte Kohonennetze

- Kohonennetze sind ausgebuffte neuronale Netze, welche auf dem Prinzip des Siegers in einem Wettbewerb beruhen. Es handelt sich um ein **UNÜBERWACHTES LERNEN**.
- **VORTEILE:** selbstorganisiert, schnell und einfach.
- Es handelt sich hierbei um ein „unnatürliches" NN, da bei diesen Netzen jedes Neuron Einfluss auf die anderen nehmen kann (ist in der Natur nicht so, dort haben die Nachbar mehr Einfluss als andere Neuronen).
- **GRUNDIDEE:** wir werfen an zufälligen Positionen Neuronen auf eine 2D-Karte. Die einzelnen Neuronen sind untereinander mit einer Art Gummiband verbunden. Die Neuronen bewegen sich in der „Lernphase" zu den Zielpunkten, wobei sie sich gegenseitig über das Gummiband beeinflussen. Dabei gilt: je weiter weg die Neuronen sind, desto weniger werden sie beeinflusst.



3.8.1. Beispiel: Kohonen-Netz für das TS-Problem

```
1) Ausgangslage) definieren
xi = {{0,0},{0,2},{2,2},{2,0},{1,1},{0.4,0.7},{0.3,1.2},{1.5,1}}; //Städte
Nneur = 30;                          //Anzahl Neuronen
Nit = 1000;                          //Anzahl Iterationen
Eta = 0.8;                           //Änderungsrate

2) Zufällige Gewichte erzeugen (Anziehung der Neuronen)
neck=Length[xi];
w=Table[{Random[],Random[]},{j,1,Nneur}];

3) Distanzfunktion erstellen (Abstand von Stadt)
dist[k_]:=Module[{},((gsi[[1]]-w[[k,1]])^2+(gsi[[2]]-w[[k,2]])^2)^(1/2)];

4) Verbessern der Gewichte
Do[
 gsi=xi[[Random[Integer,{1,neck}]]];          //Zufällige Stadt
 d=30000;                                     //Grosse Distanz

 Do[a=dist[k];If[a<d,d=a; j=k,],{k,1,Nneur}];  //Nächstes Neuron suchen

 w[[j]]+=eta (gsi-w[[j]]);                    //Ne. gegen Stadt schieben
```

```
 //Nachbarschaft der Stadt verändern (Gummiband-Effekt)
 Do[k=j-1;If[k>0,w[[k]]+=eta (gsi-w[[k]])*(1/2)^l,],{l,1,neck}];
 Do[k=j+1;If[k<Nneur+1,w[[k]]+=eta (gsi-w[[k]])*(1/2)^l,],{l,1,4}],{jj,1,
       Nit}]

5) Endresultat ausgeben (grafisch)
B = ListPlot[      w, PlotJoined->True, Axes->None, Frame->True,
                   PlotStyle->PointSize[0.02]];
```

4. GENETISCHE ALGORITHMEN

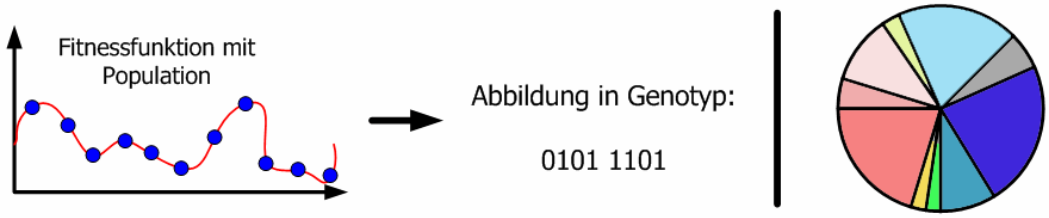
- Die klassischen NN sind mathematische Annäherungen an die Biologie. Die im letzten Kapitel vorgestellten Kohonennetze und die genetischen Algorithmen lösen bestimmte Probleme besser, sind jedoch beide nur eine durch die Biologie motiviert Annäherung.

4.1. Genetische Algorithmen

- Genetische Algorithmen gründen auf der Formalisierung der wichtigsten Prozesse und Begriffe, welche der Evolution zu Grunde liegen: **KREUZUNG**, **MUTATION** und **FITNESS**.
- Das **HAUPTPROBLEM** bei den GA ist: wie modelliere ich mein Problem in Form von Chromosomen?

4.1.1. Grundidee

- Wir haben eine zu optimierende 2D-Funktion mit vielen lokalen Minima und Maxima. Unser Ziel ist es, durch kluge Positionierung von Elementen alle Maxima abzudecken. Dabei soll die Abdeckung natürlich um das optimale Maximum herum am dichtesten sein.
- Die Elemente werden **POPULATION** genannt. Zu Beginn werden diese rein zufällig auf der sog. **FITNESSFUNKTION** positioniert. Der Y-Wert eines „Lebewesens" wird dabei **PHÄNOTYP** genannt. Da der Wert eine beliebige Zahl sein kann wird dieser in einen Binärstring mit dem Namen **GENOTYPEN** abgebildet.
- Über den Genotypen wird also jedem Lebewesen eine **FITNESS** zugewiesen. Ein höherer Wert entspricht also einfach einer höheren Fitness. Die **GESAMTFITNESS** einer Population ist die Summation über die einzelnen Fitnesswerte der Bewohner. Die einzelnen Fitnesswerte werden bildlich nun auf einem **GLÜCKSRAD** positioniert: fittere Elemente erhalten dabei mehr Fläche und werden bei der Auslosung bevorzugt. Dennoch hat jedes Element die Chance, irgendwenn ausgewählt zu werden.



- Mit Hilfe des Glücksrades werden nun zufällig zwei **ELTERN**-Elemente ausgewählt (je höher die Fitness desto höher die Wahrscheinlichkeit gewählt zu werden). Diese beiden Eltern erfahren nun eine **KREUZUNG**: Ihre Genotypen werden an einer zufälligen Stelle getrennt und zu zwei neuen **KIND**-Elementen zusammengesetzt. Die Eltern „sterben" bei diesem Prozess übrigens.

```
Elternteil 1 = 01/011101     =>    Kind 1 = 01110011
Elternteil 2 = 11/110011     =>    Kind 2 = 11011101
```

- Damit sich die Bevölkerung auch ein wenig weiterentwickelt kommt nun noch das Element der **MUTATION** zum Zug: bei den neu erzeugten Kindern kann ein zufälliges Bit des Genotyps invertiert werden. Dies geschieht allerdings sehr selten (Richtwert: 10% aller Fälle).

4.1.2. Erweiterungen

- Beim so genannten **ELITE-PROZESS** wird das fitteste Individuum der alten Generation in die neue Generation überführt.
- Es gibt auch eine Variante, in denen beim Kreuzungsvorgang mit einer gewissen Wahrscheinlichkeit gar nichts gemacht wird (Kinder = Eltern). Richtwert für die WKeit ist 20%.

4.1.3. Beispiel: Genetischer Algorithmus

```
1) Fitnessfunktion festlegen (Maximum dieser Funktion wird gesucht!)
fitnessFunction1[x_,y_]:=Sin[ Pi x]Sin[ Pi y];

2) Kreuzungsfunktion
doSingleCrossover[{string1_,string2_}] := Module[{stle,cut,temp1,temp2},
 If[   Random[] < crossoverRate,     //Kreuzung wenn Random < Kreuzungsrate
       stle=Length[string1];                                  //Länge des Str
       cut = Random[Integer,{0,stle}];                        //Schnittpunkt
       temp1 = Join[Take[string1,cut],Drop[string2,cut]];     //Kind 1
       temp2 = Join[Take[string2,cut],Drop[string1,cut]];     //Kind 2
       {temp1,temp2},{string1,string2}
 ]
]

3) Mutationsfunktion (Jedes einzelne Bit wird mit einer WKeit invertiert)
doMutation[string_] := Module[{tempstring,i},
 tempstring=string;
 Do[
       If[   Random[]<mutationRate,
             tempstring[[i]]=1-tempstring[[i]]
       ],{i,2 stringGeneLength}
 ];
 tempstring
]

4) Glücksrad-Funktionen (Kummulierte Fitness, Auswahl der Eltern)
doCumSumOfFitness := Module[{temp},                    //Summierung
 temp=0.0;
 Table[temp+=popFitness[[i]],{i,populationSize}]
]

doSingleSelection := Module[{rfitness,ind},            //Ein Individuum
 rfitness = Random[Real,{0,cumFitness[[populationSize]]}];
 ind = 1;
 While[rfitness>cumFitness[[ind]],ind++];
 ind--
]

selectPair := Module[{ind1,ind2},                      //Eltern wählen
 ind1=doSingleSelection;
 While[(ind2=doSingleSelection)\[Equal]ind1];
 {ind1,ind2}
]

5) Phänotyp zu Genotyp (und umgekehrt)
floatToBinary[x_] := Module[{temp},
 temp=RealDigits[x,2];
 Take[Join[Table[0,{-temp[[2]]}],temp[[1]],Table[0,{stringGeneLength}]],
       stringGeneLength]]/;
 x<1
]

binaryStringToTwoFloats[string_] := {
 N[Sum[string[[i]]*2^(-i),{i,1,Length[string]/2}]],
 N[Sum[string[[i]]*2^(-i+Length[string]/2),{i,Length[string]/2+1,
 Length[string]}]]
}
```

```
6) Nächste Generation mit Elite-Prozess bestimmen
updateGenerationSync := Module[{parentsid,children,ip,bestInd},
 If[   elitism,                                    //Elite-Individuum
       bestInd = Flatten[ Take[popStrings,
             Flatten[First[Position[popFitness,Max[popFitness]]]]]]
 ];

 parentsid={};                                     //Eltern wählen
 Do[AppendTo[parentsid,selectPair],{populationSize/2}];

 children={};                                      //Kinder erzeugen
 Do[   AppendTo[children,
       doSingleCrossover[{popStrings[[parentsid[[ip,1]]]],
       popStrings[[parentsid[[ip,2]]]]}]],{ip,populationSize/2}];

 popStrings=Flatten[children,1];
 popStrings=Map[doMutation,popStrings];

 If[elitism,popStrings[[Random[Integer,{1,populationSize}]]]=bestInd];
 popFitness = Table[fitnessFunction[(floatList=
                  binaryStringToTwoFloats[popStrings[[i]]])[[1]],
                  floatList[[2]]],{i,populationSize}];

 cumFitness=doCumSumOfFitness;                     //Neue Gesamtfitness
];
```

4.2. Genetische Programmierung

- Genetische Algorithmen kann man selbst wieder auf Programme anwenden, wenn man ihren Zweck definiert hat. Das automatische Erzeugen von Programmen nach diesem Muster wird genetische Programmierung genannt.
- Für die genetische Programmierung muss darauf geachtet werden, dass alle Ausdrücke für alle möglichen Werte definiert sind (Beispiel: Abbruchbedingungen).

4.2.1. Grundidee

- Unser Beispiel soll ein Robotersteuerungsprogramm in einem zweidimensionalen zellulären erzeugen. Der Roboter soll dabei vom Startpunkt zu einer Wand laufen und dann der Wand entlang einmal im Kreis herum.
- Als Grundpopulation wird mit zufällig erzeugten Programmen aus Funktionen, Konstanten und sensorischen Daten gearbeitet.
- Die Fitness wird gemessen, indem die Anzahl besuchten „Wandfelder" gezählt werden. Im Optimalfall werden dabei 28 Wandzellen gezählt, im schlechtesten Fall 0.
- Es wird mit einem abgeänderten Eliteprozess gearbeitet: die besten 10% der Programme gehen direkt in die neue Generation über. Die restlichen Programme der neuen Generation entstehen durch Kreuzung.

4.2.2. Beispiel: Genetische Programmierung

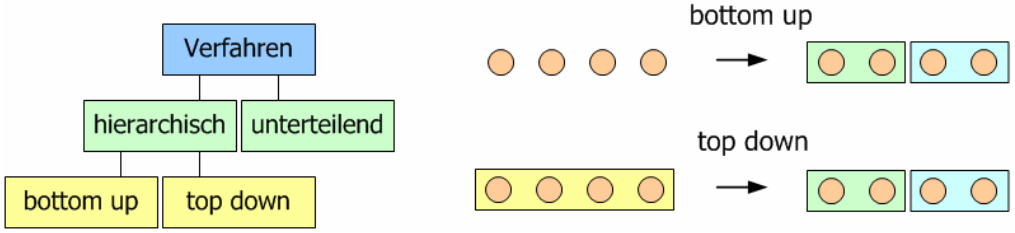
```
1) Raum und Wände festlegen
bound={      {1,1},{2,1},{3,1},{4,1},{5,1},{6,1},{7,1},{8,1},{8,2},{8,3},
             {8,4},{8,5},{8,6},{8,7},{8,8},{7,8},{6,8},{5,8},{4,8},{3,8},
             {2,8},{1,8},{1,7},{1,6},{1,5},{1,4},{1,3},{1,2}};
occupied[x_,y_] = x<1 || x>8 || y<1 || y>8;      //Wände festlegen

2) Operatoren, Sensor-Inputs und Bewegungen festlegen
randomp := Block[{r = Random[Integer, {1, 26}]},
 Which[
       r <= 3, If[randomp, Evaluate[randomp], Evaluate[randomp]],
       r <= 6, And[randomp, randomp],
       r <= 9, Or[randomp, randomp],
       r <= 12, Not[randomp],
       r == 13, e,
       r == 14, se,
```

```
       … noch mehr Sensor Inputs …
       r == 21, east,
       … noch mehr Richtungen …
       r == 25, True,
       r == 26, False]]

3) Grundpopulation erstellen
Do[program[i]=randomp,{i,1000}]

4) Fitnessfunktion festlegen (5 x Prüfen der Programmqualität)
estimate[p_] := Block[{fitness = 0},
 Do[   Block[{arrived = {},
       x = Random[Integer, {1, 8}], y = Random[Integer, {1, 8}]},

       While[FreeQ[arrived, {x, y}],
             arrived = Prepend[arrived, {x, y}];

       program[p] /. {              //Testen des Programms
       e :> occupied[x + 1, y],
       se :> occupied[x + 1, y - 1],
       s :> occupied[x, y - 1], sw :> occupied[x - 1, y - 1],
       w :> occupied[x - 1, y], nw :> occupied[x - 1, y + 1],
       n :> occupied[x, y + 1], ne :> occupied[x + 1, y + 1],
       east :> (If[! occupied[x + 1, y], x++]; Continue[]),
       south :> (If[! occupied[x, y - 1], y--]; Continue[]),
       west :> (If[! occupied[x - 1, y], x--]; Continue[]),
       north :> (If[! occupied[x, y + 1], y++]; Continue[])}];

       fitness += Length[Intersection[arrived, bound]]],
 {5}];

 fitness           //Fitness zurückgeben
]

5) Elite-Prozess (10% direkt übernehmen, 7 zufällig)
tour := Block[{group=Table[Random[Integer,{1,1000}],{7}]},
       group[[Position[evalu[[group]],Max[evalu[[group]]]][[1,1]]]]]

6) Kreuzungs-Funktion (neue Generation erzeugen)
crossover:= Block[{father=program[tour],mother=program[tour],fasub,mosub},
 fasub=Position[father,_,Heads\[Rule]False];
 mosub=Position[mother,_,Heads\[Rule]False];
 ReplacePart[mother,
 Extract[father,fasub[[Random[Integer,{1,Length[fasub]}]]]]/.{}
       \[Rule]father,mosub[[Random[Integer,{1,Length[mosub]}]]]]]

7) Programm starten bis perfektes Programm gefunden wurde (5x28 = 140)
While[ Max[evalu=Table[estimate[i],{i,1000}]]<140,
       Do[nextgen[i]=program[tour],{i,100}];
       Do[nextgen[i]=crossover,{i,101,1000}];
       Do[program[i]=nextgen[i],{i,1000}]
]
```

5. CLUSTERING-GRUPPIERVERFAHREN

- Mit Clustering wird der Vorgang bezeichnet, bei dem Objekte, welche durch Daten ausgedrückt sind, in Mengen gruppiert werden, so dass alle Mitglieder einer **GRUPPE (ODER KLASSE)** eine gewisse Eigenschaft (nach der man gruppiert) in verstärkten Masse teilen.
- Clustering wird eingesetzt für: statistische Datenanalyse, Bildanalyse, Data Mining und Mustererkennung.
- Der Clusteringprozess ist der menschlichen Wahrnehmung oder dem Erfassen nachgebildet.
- Clustering kann als ein sehr **ALLGEMEINES OPTIMIERUNGSVERFAHREN** angesehen werden, welches auf einer diskreten Menge von Elementen stattfindet.

5.1. Klassen von Clusteringverfahren



5.2. Element-Distanzmasse

- Das Mass, mit dem der **UNTERSCHIED ZWISCHEN ZWEI OBJEKTEN** gemessen wird, ist das Element-Distanzmass. Eine kleine Distanz entspricht dabei einer grossen Ähnlichkeit.
- Gebräuchliche Distanzmasse: euklidsche Distanz, Maximumnorm, Tanimoto Distanz, Ott-Tanimoto Distanz
- Ein Distanzmass muss zuerst unbedingt auf eine Skala genormt werden, um Einflüsse von grossen Elementen zu vermeiden.

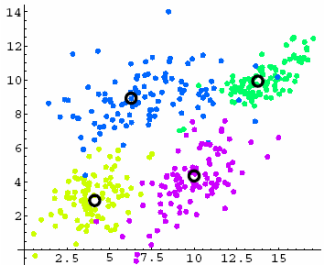
5.3. Klassen-Distanzmasse

- Das Mass, mit dem der **UNTERSCHIED ZWISCHEN ZWEI KLASSEN** (Cluster) gemessen wird, ist das Klassen-Distanzmass.
- Jede Zusammenführung von zwei Clustern gibt einen grössen Klassenabstand zwischen den restlichen Clustern. Man entscheidet, den Vorgang abzuschliessen, wenn die Cluster genügend Abstand (=Unterschied) haben.

5.3.1. Gebräuchliche Distanzmasse

- Minimaler Abstand zweier Elemente aus beiden Clustern
- Maximaler Abstand zweier Element aus beiden Clustern
- Durschnittlicher Abstand aller Elementpaare aus den beiden Clustern
- Abstand der Mittelwerte der beiden Cluster

5.4. Klassisches CV: K-Means



- K-Means ist ein klassisches CV. Es ordnet jeden Punkt demjenigen Cluster zu, dessen Mittelpunkt ihm am nächsten liegt. Dies beinhaltet, dass eine Anzahl Cluster und ihre Mittelpunkte als Ausgangslage vorgegeben werden muss.
- Es muss nicht jede Durchführung des Verfahrens dasselbe Ereignis liefern, was eine **GROSSE SCHWÄCHE** bedeutet. Auch muss das Verfahren nicht unbedingt konvergieren.

- **WEITERER PROBLEMFALL:** Sind die Elemente eines Clusters weit auseinander (viele Elemente links, viele Elemente rechts), so wird der Mittelpunkt dort positioniert, wo eigentlich gar keine Elemente liegen. Dadurch werden verständlicherweise völlig andere Elemente angezogen, als zum Cluster gehören würden.

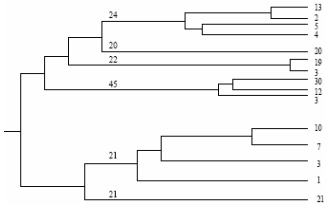
5.4.1. Beispiel: K-Means

- Folgendes Codefragment ist aus Platzgründen auf das wichtigste reduziert worden: es gibt also keine Output-Funktionen oder ähnliches.

```
1) Erzeugen der Ausgangsdaten (Anzahl Cluster, Rohelemente)
numCenters = 4;                                    //Anzahl Zentren
centri = {{4,3}, {14,10}, {7,9}, {10,4}};          //Koordinaten der Zentren
sigma = {{{2,1},{1,2}}, {{2,1},{1,1}}, {{6,1},{1,2}}, {{3,2},{2,3}}};
mdist=Table[MultinormalDistribution[centri[[i]],sigma[[i]]],{i,1,Length[cen
tri]}];
m={95,100,105,95};
numData=Plus@@m;
```

```
2) Methoden für K-Means Verfahren definieren
theNorm[x_, y_] := Sqrt[(x - y).(x - y)]          //Skalarprodukt

attributeToCenter[item_, centers_] :=             //Zu welchem Cluster?
 Sort[Thread[{Range[1, Length[centers]],
 theNorm[item, #] & /@ centers}], #1[[2]] <= #2[[2]] &][[1, 1]]

updateCenter[c_, data_] :=                        //Zentrum neu berechnen
 Block[{patterns, np}, patterns =
 Part[data, Select[Range[1, Length[data]], attributedLabel[[#]] == c &]];
 np = Length[patterns];
 Plus @@ patterns/np];

findCluster[numData_,numCenters_] :=              //Cluster suchen
 (tmp = Thread[{Range[1, numData], attributedLabel}];
 Return[Table[Select[tmp, #[[2]] == kk &][[All, 1]], {kk,
 1,numCenters}]];);

findOccurrence[lltarg_, numCenters_] := Sort[Thread[{Range[1, numCenters],
 Table[Length[Select[lltarg, # == i &]], {i, 1, numCenters}]}],
 #1[[2]] >= #2[[2]] &][[1, 1]]

kmeans[data_, numCenters_, tol_, gr_] :=          //K-Means Verfahren
 Block[{i, ind, clusters, centers, oldCenters, initialcenters},
 numData = Length[data];
 clusters = Table[0, {numCenters}];
 centers = Table[data[[Random[Integer, {1, numData}]]], {i, 1,numCenters}];
 initialcenters = centers;
 If[gr == True, Show[grPtColori, drawCenter[centers]]];
 error = 1;

3) Berechnen bis Fehler kleiner als Toleranz ist
While[error > tol,attributedLabel=attributeToCenter[#, centers] & /@data;
 If[(comp = Complement[Range[1,numCenters], Union[attributedLabel]]) != {},
       Table[attributedLabel[[Random[Integer,{1,numData}]]] = comp[[ind]],
             {ind, 1, Length[comp]}];];
       oldCenters = centers;
       clusters = findCluster[numData, numCenters];
       centers = updateCenter[#, data] & /@ Range[1, numCenters];
       error = Max[Abs[oldCenters - centers]];
       If[gr == True, Show[grPtColori, drawCenter[centers]]];];
       mappa = findOccurrence[targetLabel[[#]], numCenters] & /@ clusters;
       nerr = (Plus @@ (Min[#, 1] & /@
       Abs[mappa[[attributedLabel]] - targetLabel]))/numData // N;
       Return[{centers, nerr}];
];
```

5.5. Klassisches CV: Ward-Clustering



- Hierbei handelt es sich um ein hierarchisches bottom-up Clustering, startet also mit Clustern aus einzelnen Objekten.
- Die beiden ähnlichsten Cluster werden jeweils zu einem grössen Cluster zusammengelegt. So entsteht nach und nach ein binärer Baum, auch **DENDOGRAMM** genannt.
- Es wird eine vorgabe gemacht, wie viele Cluster schlussendlich enstehen sollen.

5.6. Autonome CV: Potts-Spin

- Bei den autononomen CV wird nicht vorgegeben, wie viele Cluster vorhanden sind, da dies in realistischen Anwendungen gerade das ist, was man herausfinden möchte.
- Dieses Verfahren bleibt normalerweise bei tiefen Temperaturen in einem lokalen Minimum stecken und die Cluster nehmen jeweils eine Farbe an.

5.6.1. Beispiel: Potts-Spin

```
1) Verfahrensparameter
r = 4;                          //Kopplungsradius
A = Table[, {i, 1, r}];
T = 0.00002;                    //Temperatur
rep = 10;                       //Durchläufe

2) Zufällige Initialisierung von 3 Clustern mit „Rauschen"
l = Union[   Table[{10,8}+2.3{Random[],Random[]},{i,1,20}],
             Table[{17,12}+2.2{Random[],Random[]},{i,1,20}],
             Table[{2,5}+2{Random[],Random[]},{i,1,20}],
             Table[{20Random[],18Random[]},{i,1,25}]];

3) Berechnung der Kopplungs-Matrix J
eukl[k_,l_]:=Sqrt[(k[[1]]-l[[1]])^2+(k[[2]]-l[[2]])^2];
abst[l1_,l2_]:=Table[N[eukl[l1[[i]],l2[[j]]]],{i,1,Length[l1]},{j,1,Length[
 l2]}]
a=abst[l,l];
ball[l_,r_]:=Table[(Order[l[[i]][[j]],r]+1)/2,{j,1,Length[l]},{i,1,Length[l
 ]}]-IdentityMatrix[Length[l]];
B=ball[a,r];
con1[b_]:=Fold[Plus,0,Fold[Plus,0,b]];
adurch[ball_,abst_]:=Fold[Plus,0,Fold[Plus,0,ball*abst]]/con1[ball]
J[ball_,abst_]:=1/con1[ball]*ball*Exp[-(abst)^2/(2*adurch[ball,a]^2)]
j1=J[B,a];

4) Zufällige Anfangswerte für Spins (4 Möglichkeiten)
spin = Table[IntegerPart[4*Random[]], {x, 1, Length[l]}];

5) Energiefunktion
e1[j_, spin_] := Fold[Plus, 0, J[B, a][[j]]
(1-Table[KroneckerDelta[(spin-Table[spin[[j]],{l1,1,Length[spin]}])[[i]]],
{i, 1,Length[spin]}])];

6) Aufdatieren des Spins
spinneu[] := IntegerPart[4*Random[]];

eneu[j_,neuspin_,spin_]:= Fold[Plus,0,J[B,a][[j]](1-Table[KroneckerDelta[
 (spin-Table[neuspin,{l1,1,Length[spin]}])[[i]]],
 {i,1, Length[spin]}])];

mont1[k_,spin_,T_,neuspin_]:= If[((eneu[k,neuspin,spin]-e1[k,spin])<0) ||
 (Random[]<Exp[-(eneu[k,neuspin,spin]-e1[k,spin])/T]),1,0];

mont3[T_]:=
 Table[If[mont1[Mod[i,Length[spin]]+1,spin,T,neuspin=spinneu[]]\[Equal]1,
 spin=ReplacePart[spin,neuspin,Mod[i,Length[spin]]+1],spin=spin],
 {i,1,rep*Length[l]}];
```

5.7. Autonome CV: SSC-Clustering

- Weiterentwicklung des Potts-Spin CV, welcher allen anderen bekannten Ansätzen überlegen ist.
- In diesem Verfahren werden die Klumpen, welche unter dem Einfluss der Wechselwirkung zwischen den Spin sich bilden, durch Erhöhung der Temperatur aufgebrochen.

6. VERSCHIEDENES

6.1. Travelling Salesman Problem (TSP)

- Die Leistung eines **HOPFIELD NETZWERKS** ist enttäuschend (kein kürzester Pfad gefunden). Ein wenig bessere Resultate erhält man mit den asynchron aufdatieren Netzwerken (nur eine Stelle). Ziemlich komplexe Implementierung.

- Das so genannte **MONTE CARLO** Verfahren erreicht sehr gute Resultate. Wurde im Unterricht aber nicht näher behandelt. Sehr komplexe Implementierung.
- Die **KOHONEN-NETZE** erreichen beachtlich gute Resultate und sind besonders einfach zu implementieren. Hauptproblem: optimale Anzahl an Neuronen (nicht zu viel und nicht zu wenig, Empfehlung: 2 x Anzahl Städte)
- Auch mit Hilfe eines **GENETISCHEN ALGORITHMUS** kann das Problem gelöst werden. Die dabei erzeugten Resultate sind als sehr gut zu bewerten, die Implementierung jedoch ist ziemlich aufwendig. Die Codierung der Individuen (Problem) entspricht dabei der Reihenfolge der Städte, die Fitnessfunktion der total benötigten Distanz zwischen allen Städten. Das Hauptproblem liegt bei der Kreuzung der Eltern, da so doppelte Städte enthalten sein könnten. Die Lösung liegt darin, dass nach der Kreuzung die doppelten Städte gelöscht werden und die restlichen Elemente des zweiten Elternteils einfach nachgeschoben werden.

6.2. Prüfungsfragen

Welche Hauptklassen von Neuronmodellen gibt es?
- Hodgkin-Huxley: trögt den molekularen Prozessen, welche in die Spikeerzeugung eingehen, detailliert Rechnung. Eletr. Schwingkreis.
- Fitzhugh-Nagumo: Ein Neuronmodell, welches sich auf die Natur des Grenz-zykluses zurückzieht
- Morris-Lecar: Sehr viel realistischer, sogar als das von Hodgkin-Huxley.
- Perzeptron: Summierung der Inputs, Spike bei Übertreten eines Wertes
- McCulloch-Pitts: Perzeptron mit Signum / Heaviside Aktivierungsfunktion

Nennen Sie die 2 einfachsten Fälle von nicht-linear trennbaren Funktionen!
XOR und NOT XOR

Nennen Sie 3 Möglichkeiten, nicht-linear trennbare Datenmengen durch Neuronenmodelle zu trennen!
Werbos-Konstruktion, mehrschichtige NN, ???

Woher kommt die Bezeichnung Backpropagation-Netzwerk?
Was bewirken die versteckten Schichten? Wieviele sollte man wählen?
- Propagation d. Fehlerkorrekturen von der Outputschicht an vers. Schichten
- Ermöglicht das Lernen von nicht-linear trennbaren Funktionen
- Mehr als zwei Schichten verbessern das Resultat nicht.

Erklären Sie die Eigenschaft „Generalisierungsfähigkeit der NN".
NN sind fähig, auch auf einen nicht-perfekten Input eine Aussage zu machen.

Gegeben sei ein BP-NW. Diskutieren Sie, was richtig ist: „Die Anzahl der Parameter (=Gewichte) der Neuronen soll [grösser/gleich/kleiner] als die Anzahl Muster sein."
Schüttel schrieb "Grösser", mit dem Verweis auf die Tatsache, dass dadurch die Kapazität des Netzes grosser werde. Antwort war aber FALSCH!

Meine Vermutung: "Kleiner", da z.B. mit 2 Inputs binär 4 Muster kodiert werden können. Kleinere Netze konvergieren schneller!

Geben Sie 3 wesentliche Schritte bei einem Clusteringprozess an!
Passende Codierung finden, Distanzmass auf Skala normieren (kein Übergewicht bei unterschiedlichen Gewichtungsskalen), CV wählen

Wie lernt ein Hopfield-Netzwerk?
Eigentlich gar nicht, da die Korrelationsmatrix direkt berechnet wird. Es wird zum Minimum konvergiert.

Wie wird eine Feuerschwelle implementiert?
Es muss einfach ein weiterer Input mit dem Wert 1 eingefügt werden!

```
ioPaar = {{{0,0,1},0}, {{0,1,1},0}, {{1,0,1},0},{{1,1,1},1}};     //AND
```