Vorlesung Neuronale Netze - Zusammenfassung

erstellt von Christoph Tornau

Erstellungsdatum dieser Version: 9. August 2003

Inhaltsverzeichnis

1 Einführung **5**
1.1 Was sind neuronale Netze? 5
1.2 Biologische Motivation 5
1.3 Eigenschaften biologischer Systeme, die man gerne übernehmen möchte 6
1.4 Merkmale biologischer Netze, welche übernommen wurden 6
1.5 Hauptkategorien 6
1.6 Drei Arten des Lernens 7
1.7 Einsatzbereich neuronaler Netze 7

2 ADALINE **8**

3 Perzeptron **9**
3.1 Aufbau 9
3.2 Lineare Separierbarkeit 9
3.3 XOR-Problem 10

4 MLP (Multilayer-Perzeptron) **10**
4.1 Einzelnes Neuron 10
4.1.1 Aufbau 11
4.1.2 Aktivierungsfunktionen (Transferfunktionen) 12
4.1.3 Umrechnung Tangenshyperbolicus in Fermifunktion 14
4.2 Multilayer Perzeptron 14
4.2.1 Aufbau 14
4.2.2 Nomenklatur 15
4.2.3 Hintondiagramm 16
4.2.4 MLPs mit linearen Kennlinien lassen sich durch Matrixmultiplikation ausdrücken 16

5 Lernen **17**
5.1 MLPs als universelle Funktionsapproximatoren 17
5.2 Lernen an Beispielen 17
5.3 Ziel des Lernens 17
5.4 Hebbsche Lernregel 18
5.5 Delta-Regel (Widrow-Hoff-Regel) 18
5.6 Backpropagation of Error 18
5.6.1 Idee 18
5.6.2 Herleitung 19
5.6.3 Algorithmus 20
5.6.4 Single-Step und Batch-Learning 20
5.6.5 Synchrone und asynchrone Aktivierung 21
5.6.6 Einstellen der Lernrate 21
5.6.7 Probleme und deren Lösungen 22
5.7 Lernkurve 23
5.8 Hinzunahme von Neuronen 23
5.9 Codierung des Outputs 23
5.10 Momentum-Term 24
5.11 Verfahren höherer Ordnung 24
5.11.1 Quickprop 24
5.11.2 Zwei Gewichte 25

5.11.3 Analytische Methode 25
5.12 Optimal Brain Demage 25

6 RBF-Netze 25
6.1 Aufbau 26
6.2 RBF-Netze sind unverselle Funktionsapproximatoren 27
6.3 Einstellen der Gewichte 28
6.4 Anpassen bzw. Wahl der Zentren und Breiten 29
6.4.1 Zentren 29
6.4.2 Breiten 31
6.5 Probleme 31
6.6 Der entscheidende Unterschied zwischen RBF-Netzen und MLPs . . . 32

7 SOMs (Self Organizing Maps) 32
7.1 Idee 32
7.2 Abstand 32
7.3 Nachbarschaftsfunktion am RBF-Netz visualisiert 33
7.4 Lernregel 33
7.4.1 Lernregel 33
7.4.2 Nachbarschaftsfunktion 33
7.4.3 Veränderung der Lernrate mit der Zeit 34
7.5 SOMs visualisiert 34
7.6 Anwendungen von SOMs 35

8 Neuronales Gas 36
8.1 Aufbau 36
8.2 Vorteile 36
8.3 Die drei Phasen des Neuronalen Gases im Ring 36

9 LVQ (Learning Vector Quantisation) 37
9.1 Prinzip 37
9.2 LVQ (LVQ1) 37
9.3 LVQ2 38
9.4 OLVQ 38

10 Hopfield Netze 38
10.1 Aufbau 39
10.2 Lernen der Muster 39
10.3 Asynchrone und synchrone Aktivierung 40
10.4 Aufnahmekapazität 40
10.5 Energieoberfläche 40
10.6 Konvergenz 40
10.7 Unsymmetrische Gewichte 40
10.8 Spezialhardware 41

11 Jordan & Elmannetze 41
11.1 Aufbau 41
11.2 Lernen 41

12 Reinforcement Learning (Bestärkendes Lernen) 42
12.1 Idee bzw. Aufgabe 42
12.1.1 Agent in Environment 42

12.1.2 Agent lernt eine Policy π . . . 42
12.1.3 Unterschied Zustand - Situation . . . 43
12.1.4 Reward und Return . . . 43
12.1.5 Sonderfälle des Rewards . . . 43
12.1.6 Beispiele . . . 44
12.2 Value Function . . . 44
12.2.1 State Value Function V^π . . . 44
12.2.2 Action Value Function Q^π . . . 44
12.3 Greedy-Policy vs. Optimal Policy . . . 45
12.4 Lernen der Value Function (Temporal Difference Learning) . . . 45
12.4.1 Was passiert? . . . 45
12.4.2 State Value Function V^π . . . 46
12.4.3 Action Value Funktion Q^π . . . 46
12.4.4 Neuronales Netz zur Bestimmung der Value function eines Teilbaumes . . . 46
12.5 Reinforcment Learning mit (adaptive) Critic . . . 46
12.6 SARO (Sensor Driven Random Optimisation) . . . 47
12.7 ASE - ACE . . . 47

13 SVMs (Support Vector Machines) 49
13.1 Idee . . . 49
13.2 Ziel . . . 50
13.3 VC-Dimension . . . 50
13.4 Risiko . . . 50
13.5 Gute Generalisierung . . . 51
13.6 Linear separierbare Probleme . . . 52
13.6.1 Linear separierende Ebene mit Korridor . . . 52
13.6.2 Decision Function . . . 53
13.6.3 La Grange . . . 53
13.7 Feature Space . . . 54
13.8 Kernel functions . . . 55

1 Einführung

1.1 Was sind neuronale Netze?

Zur Zeit gibt es keine saubere Definition von neuronalen Netzen. Es gibt einen ganzen „Zoo” von Neuronalen Netzen. Im Prinzip ist ein neuronales Netz ein Netzwerk aus vielen kleinen Einheiten, welche hochgradig verknüpft sind. Die einzelnen Einheiten sind adaptiv, d.h. sie lernen. Alle diese Einheiten – meistens Neuronen genannt – erledigen simple Aufgaben. Das neuronale Netz insgesamt aber erledigt eine komplexe Aufgabe.

1.2 Biologische Motivation

Neuronale Netze sind biologisch motiviert. Unser Nervensystem, ganz besonders unser Gehirn, und auch das Nervensystem anderer Tiere ist sehr leistungsfähig. Wir sind in der Lage zu lernen und das Gelernte anzuwenden und sogar auf unbekannte Aufgaben eine kreative Lösung zu finden. Computer sind hiervon weit entfernt und werden wohl auch immer stumpfe Rechenmaschinen ohne jegliche Kreativität bleiben, da der Mensch mehr als die Summe seiner elektrischen Impulse ist. Dennoch kann man einige Prinzipien aus der Biologie in die Informatik übertragen. Dies tut man aus zwei Gründen:

- Mit Hilfe der Computer lassen sich biologische Effekte so näher untersuchen. Biologen interessieren sich für neuronale Netze, weil sie so Beobachtungen, die sie in der Biologie gemacht haben, nachbilden können und an neuronalen Netzen erklären können. Dies soll uns jedoch nicht weiter interessieren. Wir wollen mehr in den Informatikaspekt der neuronalen Netze einsteigen.

- Mit Hilfe der aus der Biologie übertragenen Funktionsweise lassen sich Methoden implementieren, die ganz neue Funktionen von Computern ermöglichen.

Die Nervenzelle ist nicht komplett erforscht. Das, was aber schon erforscht ist, ist schon sehr komplex. Nervenzellen bestehen aus[1]

- *Dendriten:* Dies sind Verästelungen, welche elektrische Impulse von anderen Zellen sammeln. (auch Dendritenbaum genannt)

- *Synapsen:* Synapsen befinden sich an den Dendriten, aber auch an anderen Teilen der Nervenzelle. An die Synapsen sind andere Nervenzellen angekoppelt. Über den synaptischen Spalt werden Botenstoffe übertragen, die zur Auslösung eines elektischen Signals in der Zelle führen. Die Vorgänge in den Synapsen sind komplex. Eine Nervenzelle kann bis zu 150000 Synapsen haben (die Nervenzellen im Kleinhirn).

- *Zellkern und Zellkörper:* Hier findet sich das Erbgut (keine Zelle ohne Zellkern). Desweiteren finden in der Zelle Vorgänge zur Versorgung der Zelle mit Nährstoffen statt.

- *Axon (Nervenfaser):* Weiterleitung des elektrischen Signals an andere Zellen. Das Axon ist wieder an die Synapsen der Nachfolgezellen angeschlossen.

[1] Dies ist ein sehr grober Überblick über die Nervenzelle

Weiterhin gibt es Forschungsarbeiten darüber, dass Synapsen, wenn sie aktiviert worden sind, nicht so schnell wieder aktiviert werden können. Die Botenstoffe müssen erst neu an ihren Platz kommen. Die Weiterleitung des Signals über das Axon ist interessant. Es ist unklar, ob eine einzelne Nervenzelle nun nur ein Signal weiterleitet oder mehr Signale weiterleitet. Wieviel Bit überhaupt eine Nervenzelle hat, ist unklar, wenn man überhaupt eine Bitzahl für eine Nervenzelle festlegen kann.
Zur Darstellung im Computer abstrahiert man stark. Im Computer besteht ein **Neuron** (meistens) nur noch aus:

- Gewichten an den Verbindungen von anderen Neuronen oder der Eingabe und ein Gewicht als Schwellwert.
- Summe
- Aktivierungsfunktion und Ausgabe

Dazu später mehr.

1.3 Eigenschaften biologischer Systeme, die man gerne übernehmen möchte

- Fehlertoleranz
- Generalisierungsfähigkeit
- Lernen
- Selbstorganistation
- ...

1.4 Merkmale biologischer Netze, welche übernommen wurden

- identisch aufgebaute Einheiten (Neuronen), die lernfähig sind.
- hochgradige Verknüpfung zwischen den Neuronen
- jedes Neuron: Eingabevektor, gewichtete Summe, nicht-lineare Kennlinie, skalarer Output (mehr nicht)
- Lernen an Beispielen bzw. am Fehler auf Beispielen

1.5 Hauptkategorien

Mustererkennung: Das neuronale Netz lernt verschiedene Muster. Nach dem Lernen wird dem Netz ein Muster dargeboten. Es entscheidet, zu welcher Musterklasse es gehört. Zum Beispiel: Texterkennung. Es werden viele 'A's gelernt. Dann werden alle 'A's als 'A's erkannt.

$$\mathbb{B}^n, \mathbb{R}^n \longrightarrow \mathbb{B}^m$$

Funktionsapproximation: Das Netz bekommt verschiedene Stützstellen einer Funktion präsentiert. Nach dem Lernen kann es dann Funktionswerte einer Funktion berechnen. Zum Beispiel: Vorhersage von Börsenkursen[2].

$$\mathbb{B}^n, \mathbb{R}^n \longrightarrow \mathbb{R}^m$$

[2] Von vielen werden neuronale Netze als eine Methode zur Vorhersage von Börsenkursen genannt. Es waren Systeme im Einsatz, die aus den Börsenkursen selbst Vorhersagen getroffen habe. Diese

1.6 Drei Arten des Lernens

	Deutsch	Englisch	Beschreibung
Teacher	Überwachtes Lernen	Supervised	Es gibt einen desired Output, welchen das Netz lernen soll.
Reward	Bestärkendes Lernen	Reinforcment Learning	Es gibt nur eine Belohnung, die sagt, ob die Aktion gut oder schlecht war.
-	Unüberwachtes Lernen	Unsupervised	Es gibt gar keinen Output mehr. Wir können die Muster nur zu Clustern zusammenfassen.

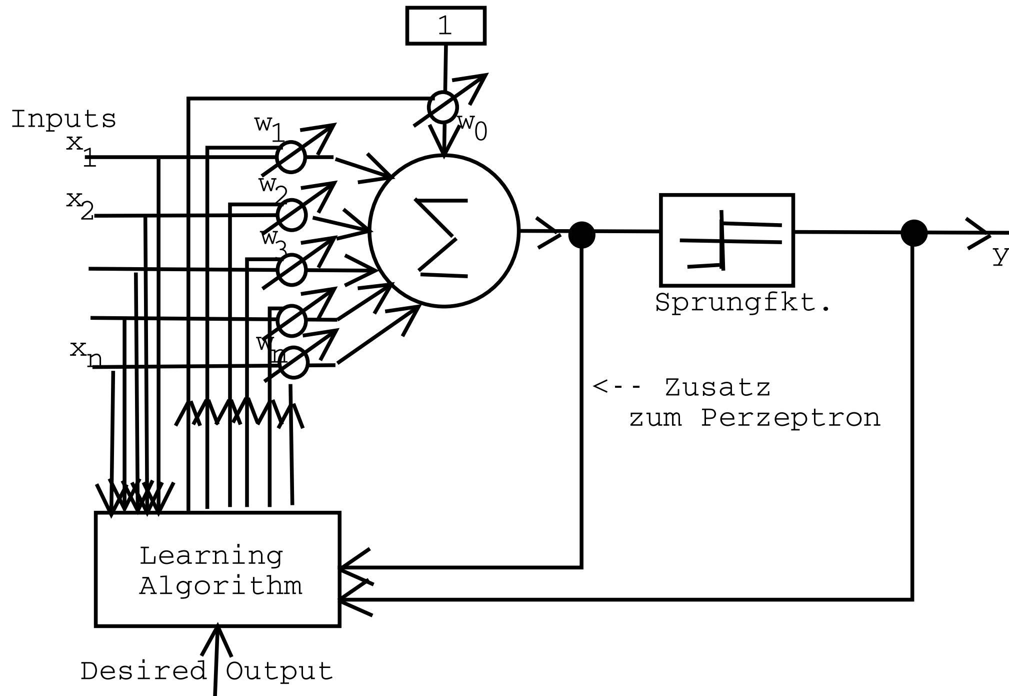
1.7 Einsatzbereich neuronaler Netze

Neuronale Netze werden heutzutage in vielen Bereichen unseres Lebens eingesetzt. Häufig sind sie dabei jedoch nur Teile eines komplexen Systems. Das ein Gerät/System ein Neuronales Netz benutzt, wird jedoch häufig nicht genannt, da mit dem Begriff neuronalem Netz negative Assoziationen bei den Kunden auslöst. Neuronale Netze werden eingesetzt in

- Bildverarbeitung
- Schrifterkennung
- Spracherkennung
- Robotersteuerung
- Regelungsaufgaben (z.B. Waschmaschine)
- ...

funktionieren aber nicht mehr, da sich der Markt diesen Systemen angepaßt hat (Rückkopplung). Denkbar sind Systeme, die aus zusätzlichen Daten Ergebnisse errechnen, wie zum Beispiel dem Wetter oder den Tageszeitungen. Es ist wichtig, welche Eingabedaten man dem Neuronalen Netz anbietet. Nichtrelavante Eingabedaten sollten ausgeblendet werden.

2 ADALINE



Die Stärke des ADALINE gegenüber einem einzelnen Perzeptron-Neuron ist es, dass schon vor der Sprungfunktion das Signal abgefangen wird und somit ein besseres Einstellen der Gewichte möglich ist. Der Output nach der Sprungfunktion wird benutzt, um ein Flag zu geben, ob das ADALINE nun lernen soll oder nicht. Der Learning Algorithm stellt mit Hilfe der Hebbschen Lernregel

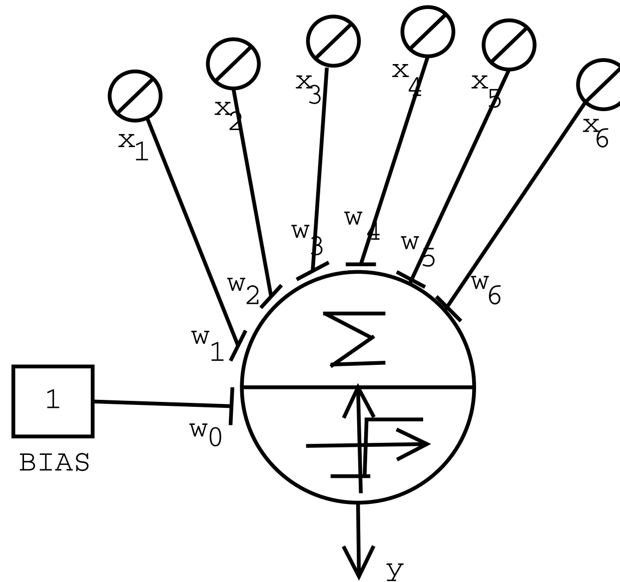
$$\Delta w_i = \eta x_i y$$

die Gewichte ein.

Das ADALINE war in Hardware mit physikalischen Schaltern, Potiometern und analogem Messgerät implementiert. Also ein Analogcomputer.

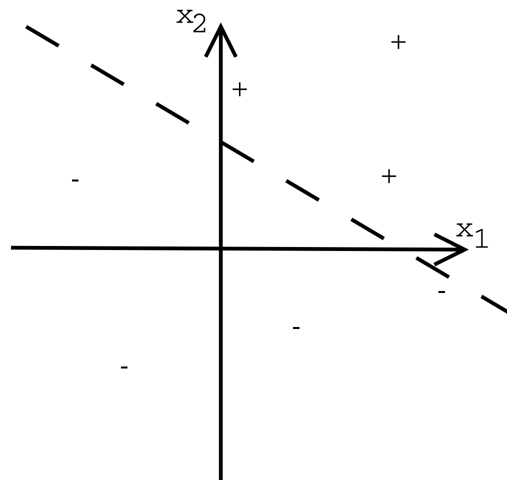
3 Perzeptron

3.1 Aufbau



3.2 Lineare Separierbarkeit

Ein einzelnes Perzeptron teilt den Eingangsraum mit einer Ebene in zwei Teile. Im zweidimensionalen sind diese Ebenen Geraden, im dreidimensionalen Flächen usw. Beispiel für zweidimensionalen Eingangsraum:



Wir können diese Ebene lernen lassen. Wenn der Output des Perzeptrons nicht dem Desired Output entspricht, können wir die Gewichte lernen lassen, indem wir ein Δw_i aufaddieren:

$$\Delta w_i = D(x) \cdot x_i$$

Im zweidimensinalen hat die separierende Gerade folgende Formel:

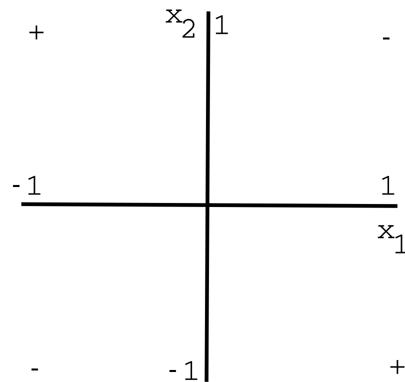
$$x_1 \cdot w_1 + x_2 \cdot w_2 + w_0 = 0$$

Wir können diese Formel nach x_2 umstellen und erhalten eine „normale" Geradengleichung

$$x_2 = \frac{-w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

3.3 XOR-Problem

Es gibt Mustermengen, die sich durch eine Ebene nicht teilen lassen. Beispielsweise das XOR-Problem im Zweidimensionalen. Im Zweidimensionalen wäre die Ebene eine Gerade:



Wir finden keine Gerade, die diese Punkte in die $+$ und die $-$-Klasse unterteilen kann. In Gleichungen aufgeschrieben ist folgendes Gleichungssystem unlösbar:

$$\begin{aligned} w_1 0 + w_2 0 &= 0 \\ w_1 0 + w_2 1 &= 1 \\ w_1 1 + w_2 0 &= 1 \\ w_1 1 + w_2 1 &= 0 \end{aligned}$$

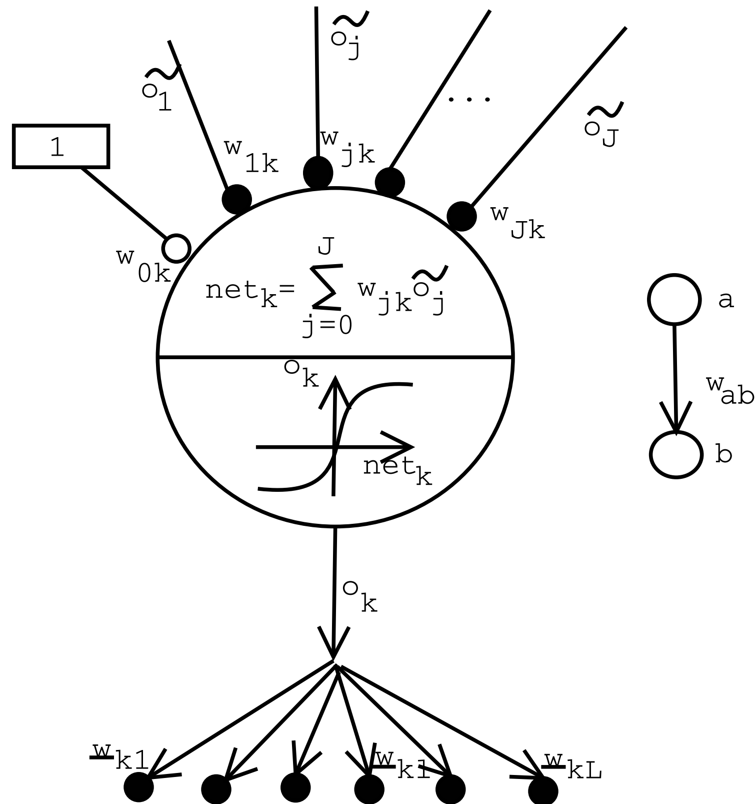
Im zweidimensionalem haben wir 2 Funktionen, die wir nicht lineare separieren können. In höheren Dimensionen steigt jedoch der Prozentsatz der nicht linear separierbaren Funktionen stark an.

4 MLP (Multilayer-Perzeptron)

4.1 Einzelnes Neuron

Das MLP besteht aus vielen einzelnen Neuronen. MLP steht für Multilayerperzeptron. Häufig wird ein solches neuronales Netz jedoch auch als Perzeptron bezeichnet, so dass Verwechslungsgefahr zu wirklich einzelnen Perzeptrons besteht.

4.1.1 Aufbau



Das Perzeptron k besteht von oben nach unten

- aus J Eingängen. Jeder Eingang hat einen Wert mit $\tilde{o}_j$ bezeichnet,
- einem Eingang $\tilde{o}_0$, welcher immer 1 ist. Dies wird für den Schwellwert (BIAS) benötigt.
- $J+1$ Gewichten. Die Zählung beginnt bei dem Gewicht für den Schwellwert mit 0 und läuft bis J. Die Gewichte werden angegeben mit $w_{\text{von,nach}}$. Siehe dazu auch rechts stehendes kleines Bild. Die Gewichte vom vorhergehenden Neuron oder von der Eingabe werden mit ausgefüllten Kreisen angegeben. Das eine Gewicht für den Schwellwert (BIAS) hat einen unausgefüllten Kreis.
- der Summe net_k. Summiert werden die Produkte aus den Eingaben und den Gewichten. (incl. BIAS-Gewicht mit 1-Eingabe)
- der Aktivierungsfunktion (auch Transferfunktion, Schwellfunktion, Neuronenfunktion ... genannt) f_k. Die Eingabe dieser Funktion ist die Summe net_k.
- der Ausgabe o_k. Sie wird berechnet durch

$$o_k = f_k(net_k) = f_k\left(\sum_{j=0}^{J} w_{jk}\tilde{o}_j\right).$$

- die Gewichte der nächsten Schicht werden mit w_{kl} bezeichnet.

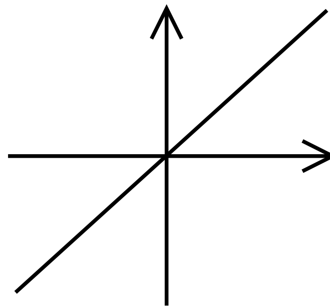
4.1.2 Aktivierungsfunktionen (Transferfunktionen)

Es gibt verschiedene Aktivierungsfunktionen, die eingesetzt werden können, um die Netzsumme in die Ausgabe zu verrechnen. Die Aktivierungsfunktion sollte

- streng monoton steigend
- differenzierbar (einige Funktionen (Sprungfunktion) sind nicht differenzierbar; sie funktionieren so nicht im Backpropagation-Verfahren)
- nach oben und nach unten beschränkt (die lineare Funktion ist nicht beschränkt)
- Werte zwischen -1 und 1 oder 0 und 1 (die lineare Funktion liefert keine Werte dazwischen, sondern auch viel größere bzw. kleinere Werte)
- Wendepunkt bei $z = 0$

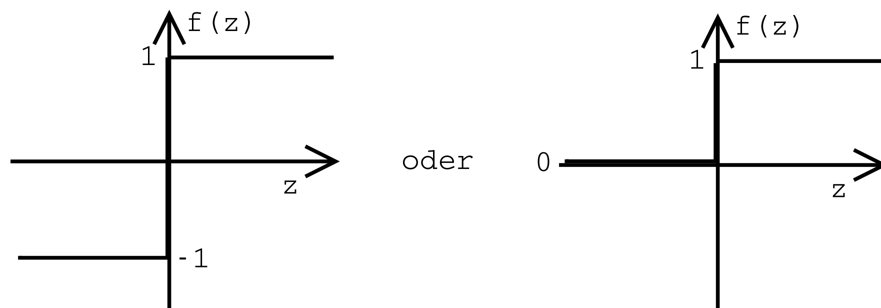
Folgende Aktivierungsfunktionen sind die am häufigsten benutzten[3]:

- *Lineare Kennlinie (Identität)*: $f_k(z) = z$



- *Sprungfunktion (Stufenfunktion, Heoviside Funktion)*:

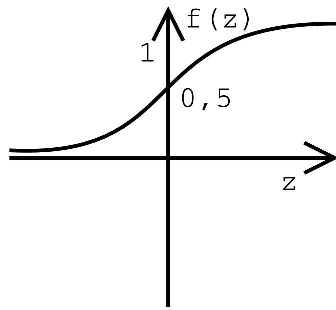
$$f_k(z) = \begin{cases} +1 & z \geq 0 \\ -1 & z < 0 \end{cases} \text{ oder } f_k(z) = \begin{cases} +1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$



- *Fermifunktion (Logistische Funktion, eine sigmoide Funktion)*:

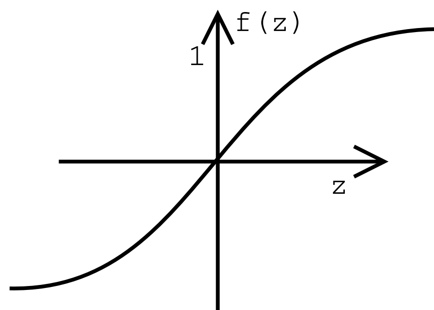
$$f(z) = \frac{1}{1 + e^{-z}}$$

[3]Graphen sind nur Skizzen

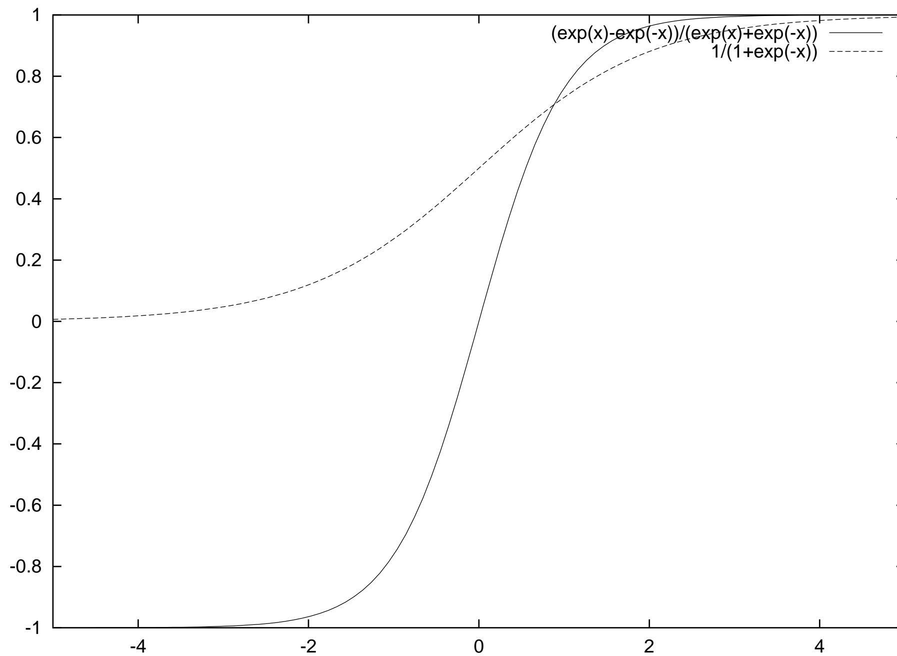


- *Tangenshyperbolikus*:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Tangentshyperbolicus und Fermifunktion durch Gnuplot dargestellt:



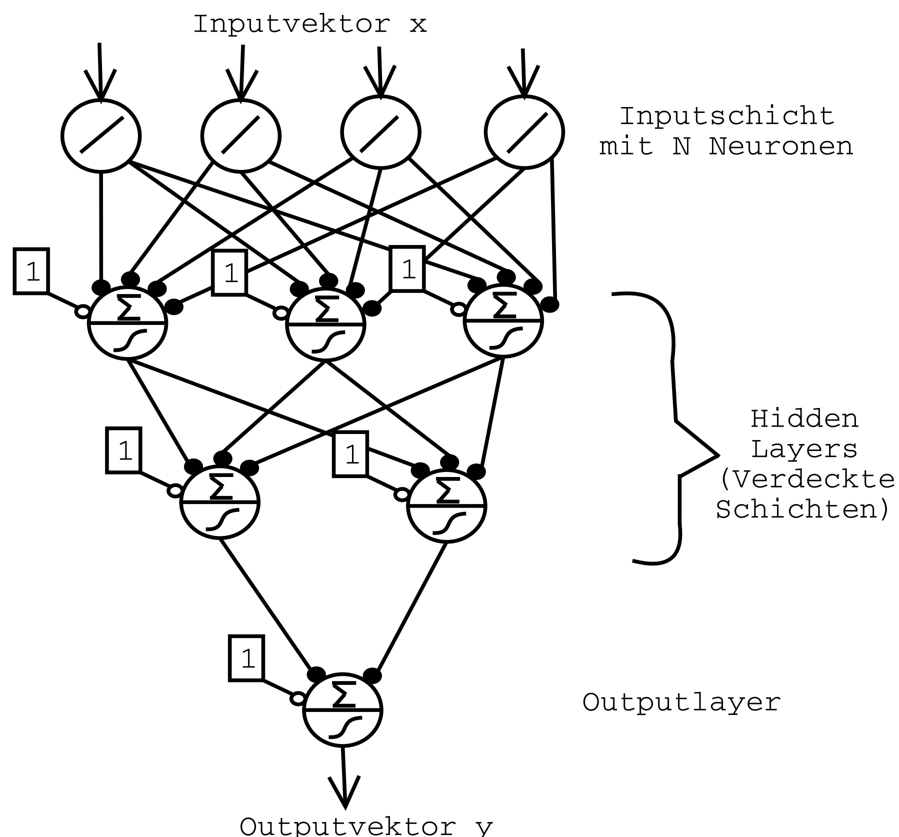
4.1.3 Umrechnung Tangenshyperbolicus in Fermifunktion

Der Tangenshyperbolicus läßt sich in die Fermifunktion umrechnen. Hierbei benutzen wir $e^z = \frac{1}{e^{-z}}$

$$
\begin{aligned}
\tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\
&= \frac{\frac{1}{e^{-z}} - e^{-z}}{\frac{1}{e^{-z}} + e^{-z}} \\
&= \frac{\frac{1}{e^{-z}}}{\frac{1}{e^{-z}} + e^{-z}} - \frac{e^{-z}}{\frac{1}{e^{-z}} + e^{-z}} \\
&= \frac{1}{\frac{1}{e^{-z}}e^{-z} + e^{-z}e^{-z}} - \frac{e^{-z} + \frac{1}{e^{-z}} - \frac{1}{e^{-z}}}{\frac{1}{e^{-z}} + e^{-z}} \\
&= \frac{1}{1 + e^{-2z}} - \frac{e^{-z} + \frac{1}{e^{-z}}}{\frac{1}{e^{-z}} + e^{-z}} + \frac{\frac{1}{e^{-z}}}{\frac{1}{e^{-z}} + e^{-z}} \\
&= \frac{1}{1 + e^{-2z}} - 1 + \frac{1}{1 + e^{-2z}} \\
&= 2\frac{1}{1 + e^{-2z}} - 1 \\
&= 2f(z) - 1 \text{ ,wobei f die Fermifunktion}
\end{aligned}
$$

4.2 Multilayer Perzeptron

4.2.1 Aufbau



- Die Neuronen der einzelnen Schichten sind bei MLPs vollverknüpft. Dabei gibt es nur Vorwärtsverknüpfungen (Feed forward net). Es werden keine Schichten übersprungen.

In der Literatur (z.B. Zell) findet man neuronale Netze von unten nach oben gezeichnet. Wir zeichnen hier von oben nach unten.

- Die Indizes der Neuronen von links nach rechts der einzelnen Schichten laufen immer mit Kleinbuchstaben. Der größte Indize wird mit dem jeweiligen Großbuchstaben bezeichnet. Z.B. Inputschicht: $1, 2, \dots, n, \dots, N$
- Die Inputschicht wird mit N bezeichnet. Die Outputschicht mit M. Das Hiddenlayer mit H. Gibt es weitere Hiddenlayer, wie hier, werden diese mit Buchstaben nach H gekennzeichnet, also hier dritte Schicht von oben G.
- Input und Output sind Vektoren. Diese werden manchmal durch einen Unterstrich, manchmal durch einen Pfeil darüber, manchmal auch gar nicht gekennzeichnet.

$$x \in \mathbb{R}^N \quad y \in \mathbb{R}^M \quad \text{oder auch} \quad x \in \mathbb{B}^N \quad y \in \mathbb{B}^M$$

- Jede Schicht hat eine Gewichtsmatrix. Diese hat die Dimension

$$(X + 1, Y)$$

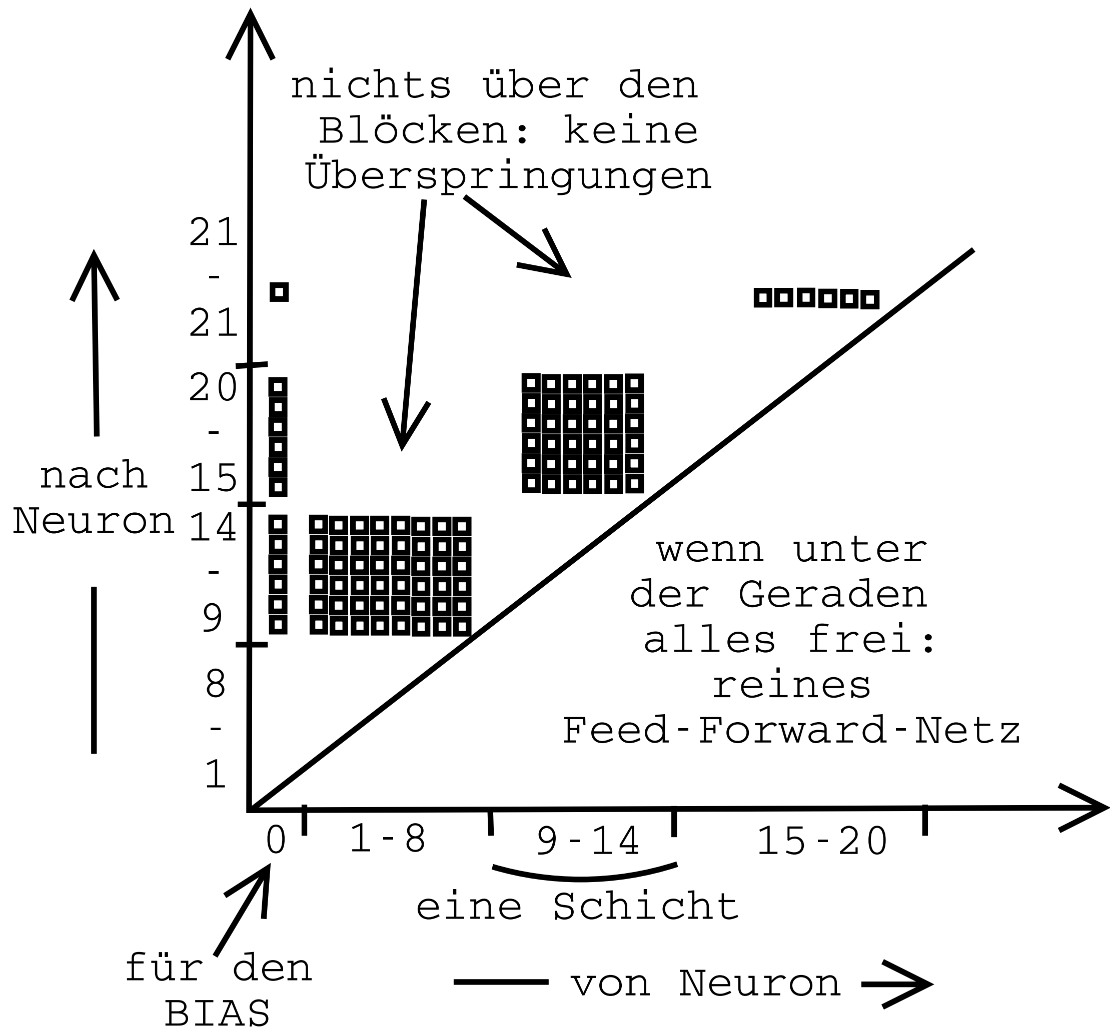
wobei X die Anzahl der Neuronen der darüberliegenden Schicht ist (also der Schicht von der die Schicht die Signale bekommt) und Y die Anzahl der Neurone der Schicht ist. Jedes Neuron der Schicht bekommt also eine Spalte der Matrix zugeteilt, in welche es seine Gewichte abspeichert.
Hinweis: Wollen wir eine Schicht mit linearer Kennlinie mit Hilfe der Matrixmultiplikation berechnen, also die Gewichtsmatrix G mit dem Vektor v multiplizieren, so müssen wir die Matrix genau andersherum definieren: Eine Zeile ist nun für jedes Neuron reserviert. An forderster Stelle (also die gesamte erste Spalte der Matrix) steht der BIAS. Wir müssen den Vektor v umschreiben, indem wir die erste Komponente 1 setzen und alle weiteren Komponenten nach unten verschieben. Führen wir die Matrixmultiplikation so durch, so erhalten wir als Ergebnis den Outputvektor der gesamten Schicht.

4.2.2 Nomenklatur

Obiges Netz stellt ein vierschichtiges oder dreistufiges Netz da. Bei „Stufennomenklatur" wird die Eingabeschicht nicht als Stufe mitgezählt, weil diese sowieso nur die Eingaben weiterleitet.
Die Anzahl der Neuronen wird von der Inputschicht an hintereinander geschrieben. Obiges Netz ist z.B. ein $4 - 3 - 2 - 1-$MLP.
Man kann desweiteren hinter jede Nummer eine Zahl schreiben, die angibt, welche Kennlinie die Neuronen der jeweiligen Schicht haben. Für obiges Netz z.B. $4L - 3T - 2T - 1T-$MLP.

T	Tangenshyperbolicus
L	Lineare Kennlinie
F	Fermi Funktion

4.2.3 Hintondiagramm



Die einzelnen Kästchen variieren in Farbe und Größe. Große Kästen können z.B. hohe Beträge der Gewichte veranschaulichen. Schwarze Kästen negative Gewichte. Nicht ausgefüllte positive. Es sind auch Schattierungen der Kästen möglich.
Hintondiagramme eignen sich hervorragend um den Status eines neuronalen Netzes abzulesen. In Hintondiagrammen bilden sich bei einigen Lernaufgaben für neuronale Netze Muster. Weiterhin kann man gucken, wo eventuell Verbindungen gelöscht werden bzw. neu hinzugefügt werden.

4.2.4 MLPs mit linearen Kennlinien lassen sich durch Matrixmultiplikation ausdrücken

Den Output einer Schicht mit linearen Kennlinien kann man durch Vektor-Matrix-Multiplikation berechnen. Haben nun mehrere Schichten hintereinander lineare Kennlinien, so lassen sich all diese Schichten zu einer Schicht zusammenfassen, indem wir eine Matrixmultiplikation über die Gewichtsmatrixen durchführen.

5 Lernen

5.1 MLPs als universelle Funktionsapproximatoren

- MLPs sind unversale Funktionsapproximatoren. D.h., dass sie beliebige Muster und Funktionen darstellen können. Wieviele Neuronen nun für ein bestimmtes Problem begnötigt werden ist nicht klar. Gibt man einem neuronalen Netz zuviele Neurone, lernt es sehr schnell auswendig und generalisiert nicht. Hat es zu wenige, wird das Problem nicht ordentlich gelöst. Die Anzahl der Neurone in der Hidden Schicht eines MLPs ist maßgeblich für die Mächtigkeit eines MLPs. Neuronale Netze mit mehr als einer Hiddenschicht sind leistungsfähiger als die mit einer Schicht. Sie sind jedoch auch langsamer in der Berechnung, da durch mehr Neuronen zurückpropagiert werden muss. Einige Probleme, die sich mit einer Hidden-Schicht nicht lösen lassen, lassen sich mit zwei Schichten lösen. Wieder andere lassen sich besser lösen. Wieder andere lassen sich auch mit beidem nicht lösen.

- Das ein MLP eine Funktion darstellen kann, heißt noch lange nicht, dass es sie auch lernen kann. Nur einzelne Perzeptrons können alles lernen, was sie repräsentieren können[4]. Es kann tiefe Täler in der Fehleroberfläche geben, in welche wir aber leider nie hineinspringen. In der Praxis interessieren wir uns mehr für die Lernbarkeit von Funktionen.

5.2 Lernen an Beispielen

Lernen anhand von Beispielen. Es wird gibt eine Menge von Beispielsmustern[5] zu denen jeweils ein *Desired Output*[6] vorliegt:

$$(\underline{x}, \underline{D}(\underline{x})) \quad \text{oder} \quad (\underline{x}, \underline{Y}(\underline{x}))$$

$$\underline{x} \in \mathbb{R}^N \quad \text{und} \quad \underline{Y}(\underline{x}), \underline{D}(\underline{x}) \in \mathbb{R}^M$$

Diese Daten heißen Trainingsmuster oder Lernmuster für die Mustererkennung. Trainingsdaten oder Lerndaten für die Funktionsapproximation und auch für die Mustererkennung.

5.3 Ziel des Lernens

Lernen – oder zunächsteinmal das Verhalten des Netzes verändern – können wir, indem wir an den Gewichten irgendwie drehen[7]. Ziel des Lernens ist es, den Fehler zwischen dem Desired Output und dem tatsächlichen Output für alle Eingaben möglichst klein zu machen. Es gibt

- den Fehler für ein Trainingsmuster: Der Fehler für ein Trainingsmuster p ist definiert als

$$^{p}E = \frac{1}{2} \sum_{k=1}^{K} \left(d_k(^{p}\underline{x}) - Y_k(^{p}\underline{x})\right)^2 .$$

[4] Berühmter Rosenblatt-Beweis

[5] bzw. für Funktionsapproximation auch Beispielsdaten genannt

[6] desired: erwünscht

[7] Man muss sich die Gewichte als Potimeter, z.B. als Lautstärken(dreh)knopf beim Autoradio vorstellen. Unser Ziel ist es nun, dass wir die einzelnen Knöpfe so drehen, dass das Netz für Daten mit unbekannter Ausgabe eine gute Ausgabe erstellt.

Um dieses Trainingsmuster zu lernen, stellen wir die Gewichte so ein, dass genau die richtige Ausgabe herauskommt. Das ist relativ einfach lösbar und funktioniert immer.

- den Fehler für alle Trainingsmuster zusammen: Dieser Fehler ist definiert als die Summe aller Fehler der Trainingsmuster

  $$E = \sum_{p=1}^{P} {}^{p}E.$$

  Wir möchten gerne für alle Beispiele zusammen die Gewichte so einstellen, dass immer die richtige Ausgabe herauskommt, wenn wir ein Beispiel anlegen. Das ist sehr viel schwieriger und vielleicht unlösbar[8].

Was wir wollen ist allerdings, dass auch unbekannte Daten richtig klassifiziert werden. Dies ist ein schwieriges Problem der Neuronalen Netze. Unter gewissen Umständen lernen die Neuronalen Netze die Trainingsdaten auswendig. Auf unbekannten Daten versagen sie völlig. Wir möchten gerne eine **Generalisierung** erreichen. Dazu später mehr.

5.4 Hebbsche Lernregel

Die Hebbsche Lernregel läßt sich wie folgt umgangssprachlich ausdrücken:

Wenn eine Zelle j eine Eingabe von Zelle i erhält und beide gleichermaßen aktiv sind, wird das dazwischenliegende Gewicht erhöht, d.h. deren Verbindung verstärkt.

Als Formel:

$$\Delta w_{ij} = \eta \tilde{o}_i a_j$$

Hierbei ist η die Lernrate, $\tilde{o}_i$ der Output von Neuron i und a_j die Aktivierung von Neuron j, wobei typischerweise für die Aktivierung die Werte $-1, +1$ vorliegen.

5.5 Delta-Regel (Widrow-Hoff-Regel)

Die Delta-Regel (oder auch Widrow-Hoff-Regel) ist von der hebbschen Lernregel abgeleitet und ist ein Spezialfall des Backpropagationverfahrens. Sie ist nur für MLPs mit einer Schicht benutztbar, bzw. für die Ausgabeschicht eines MLPs. Sie lautet:

$$\Delta w_{ik} = \eta(d_k - o_k)\tilde{o}_i$$

5.6 Backpropagation of Error

5.6.1 Idee

Bei Netzen mit mehr Schichten wird man vor das Problem gestellt, dass man keinen direkten Fehler für Neurone der Hiddenschicht oder Schichten darüber bestimmen kann. Lange Zeit konnten deshalb mehrschichtige Netze nicht lernen. Die Backpropagation-Regel hebt jedoch dieses Dilemma auf. Sie propagiert den Fehler zurück bis zur Eingabeschicht.

Die Backpropagationregel führt einen Gradientenabstieg auf der Fehleroberfläche durch.

[8] Die Lösbarkeit hängt von der Anzahl der Neuronen im Hidden Layer ab und davon ab, ob die Trainingsdaten in sich selbst konsistent sind. Wenn für ein und dasselbe Beispiel zwei verschiedene Outputs existieren, sind die Daten nicht konsistent.

5.6.2 Herleitung

Der Gradient ist die Richtung des steilsten Anstieges auf der Fehleroberfläche. Es ist ein Vektor, der uns sagt, wo es am steilsten hoch geht. Wir möchten jedoch den Fehler minimieren. Deshalb suchen wir den steilsten Abstieg. Dies ist meistens der negative Gradient. Dies muss jedoch nicht so sein. Mit dem negativen Gradienten liegen wir jedoch meistens richtig. Wir bestimmen die Gewichtsänderung Δw_{ik} für das Gewicht w_{ik}:

$$\Delta w_{ik} \sim - \bigtriangledown {}^{p}E = -\frac{\partial {}^{p}E(w_{ik})}{\partial w_{ik}}$$

Wir führen einen Proportionalitätsfaktor η, auch Lernrate genannt, ein:

$$\Delta w_{ik} = -\eta \underbrace{\frac{\partial {}^{p}E(w_{ik})}{\partial w_{ik}}}_{1}$$

Weiterrechnen bei 1 mit mehrdimensionaler Kettenregel[9]

$$\frac{\partial E(w_{ik})}{\partial w_{ik}} = \frac{\partial E(net_k)}{\partial net_k} \cdot \frac{\partial net_k(w_{ik})}{\partial w_{ik}}, \text{da } E(w_{ik}) = E(net_k(w_{ik}))$$

$$= \underbrace{\frac{\partial E}{\partial net_k}}_{2} \cdot \underbrace{\frac{\partial net_k}{\partial w_{ik}}}_{3}$$

Aus 3 folgt

$$\frac{\partial net_k}{\partial w_{ik}} = \frac{\partial \left(\sum_{j=0}^{J} w_{jk} \cdot \tilde{o}_j \right)}{\partial w_{ik}} = \sum_{j=0}^{J} \underbrace{\frac{w_{jk} \cdot \tilde{o}_j}{\partial w_{ik}}}_{\substack{j \neq i \;\Rightarrow\; 0 \\ j = i \;\Rightarrow\; 1 \cdot \tilde{o}_j}} = \tilde{o}_j$$

2 ist definiert als $-\delta_k$:

$$\delta_k = \frac{\partial E}{\partial net_k} = \underbrace{\frac{\partial E}{\partial o_k}}_{4} \cdot \underbrace{\frac{\partial o_k}{\partial net_k}}_{5}$$

Term 5 ist abhängig von der gewählten Kennlinie des Neurons

$$\frac{\partial o_k}{\partial net_k} = \frac{\partial f(net_k)}{\partial net_k} = f'(net_k)$$

$$= \begin{cases} 1 & \text{für lineare Kennlinie} \\ 1 - \tanh^2(net_k) & \text{für Tangentshyperbolikus} \\ f(net_k)(1 - f(net_k)) & \text{für Fermifunktion} \end{cases}$$

Term 4 ist abhängig von der Position des Neurons im MLP (entweder Outputlayer oder Hiddenlayer). Für das Outputlayer ist dieser Term

$$\frac{\partial E}{\partial o_k} = \frac{\partial}{\partial o_k} \underbrace{\frac{1}{2}\sum_{k=1}^{K}(d_k - o_k)^2}_{\text{Definition des Fehlers}} = \frac{1}{2}2(d_k - o_k)(-1) = -(d_k - o_k)$$

[9]Ab dieser Stelle lassen wir das p oben weg. Wir rechnen jedoch weiter nur auf einem Beispiel.

Dies führt zur Delta-Regel.

Für die darüberliegenden Layer

$$\frac{\partial E}{\partial o_k} = \frac{\partial E(\underline{net_l})}{\partial o_k} = -\sum_{l=1}^{L} \underbrace{\left(-\frac{\partial E}{\partial \underline{net_l}}\right)}_{\delta_l} \cdot \underbrace{\frac{\partial \underline{net_l}}{\partial o_k}}_{\underline{w_{kl}}}$$

Somit ergibt sich für Backpropagation **insgesamt**

$${}^p\Delta w_{ik} = \eta\, {}^p\delta_k\, {}^p\tilde{o}_j$$

Für **Outputlayer**

$${}^p\delta_k = ({}^pd_k - {}^po_k) \cdot f_k'({}^pnet_k)$$

Für **Hidden und Inputlayer**[10]

$${}^p\delta_k = \sum_{l=1}^{L} ({}^p\delta_l \cdot \underline{w_{kl}}) \cdot f_k'({}^pnet_k)$$

5.6.3 Algorithmus

0: **Initialisieren** der Gewichte auf zufällige Werte
1: **Trainingsmuster wählen**
2: **Vorwärtsschritt** durchführen (den Output des neuronalen Netzes berechnen) Dabei die Aktivierungen der einzelnen Neuronen abspeichern, da wir die für die Berechnung der Ableitung der Kennlinie benötigen.
3: **Desired Output** mit dem tatsächlichen Output vergleichen und dadurch alle δ für die Outputschicht berechnen. Daraus und aus der Ableitung Kennlinie berechnen der **Gewichtsänderungen Δw in der Outputschicht.**
4: **Backpropagation** in darüberliegende Schichten durch Berechnen aller δ aus Gewichten der nachfolgenden Schicht und deren verschiedenen δ. Wieder Bestimmung der **Gewichtsänderungen** Δw aus diesen Werten und mit Hilfe der Kennlinie.
5: **Update** der Gewichte $w = w + \Delta w$
6: Wenn nicht zufrieden, noch einmal bei 1 anfangen

5.6.4 Single-Step und Batch-Learning

- Das mathematisch korrektere Verfahren ist das **Batchlearning**. Beim Batchlearning werden alle Muster dem Netz nacheinander zugeführt und für jedes Muster die Gewichtsänderungen festgestellt. Nachdem alle Muster dran waren, werden die Gewichtsänderung für ein Gewicht zusammenaddiert und auf das aktuelle Gewicht draufaddiert:

$$w_{ik} = w_{ik} + \sum_{p=1}^{P} {}^p\Delta w_{ik}$$

  Dieses Verfahren verschlingt viel Rechenzeit.

[10] Das Inputlayer muss sowieso nicht gelernt werden, da wir alles, was wir im Inputlayer machen auch im Hiddenlayer machen können. Das Inputlayer dient nur zur Weiterleitung.

- Das schnellere Verfahren ist das **Single-Step** Verfahren. Es wird auch **stochastisches Gradientenverfahren** genannt. Hierbei berechnen wir den Fehler für ein Muster und aktualisieren nach der Hochpropagierung die Gewichte sofort. Verwirrend ist, dass nicht alle Muster dieselbe Fehleroberfläche haben[11]. Dieses Verfahren hat die Idee, dass alle Muster den Fehler in einer gemeinsamen Senke wohl schon minimieren werden. Wegen dieser ungefähren wagen Behauptung ist es mathematisch ungenau. Die Praxis aber zeigt, dass dies genau der Fall ist. Deshalb kommt dieses Verfahren, vor allen Dingen wegen der erhöhten Geschwindigkeit, in Anwendungen häufig zum Einsatz.

5.6.5 Synchrone und asynchrone Aktivierung

- Bei **synchroner Aktivierung** werden alle Neuronen gleichzeitig auf neue Outputwerte gesetzt. Dies kann man zum Beispiel in Spezialhardware erreichen oder man speichert die neue Ausgabe zwischen, wobei aber für jedes Neuron ein weiterer Speicherplatz verbraucht wird.

- Bei **asynchroner Aktivierung** wird jedes Neuron nach seiner Berechnung sofort auf die neue Ausgabe gesetzt. Bei der asynchronen Aktivierung muss man sich überlegen, wie man die Neuronen, die neu aktiviert werden, aussucht. Es gibt

  - feste Reihenfolge: Es ist nur eine Reihenfolge der Neuronen möglich. Nicht vorteilhaft, da es sehr schnell zu Oszillationen auf der wechselnden Fehleroberfläche kommen kann.
  - zufällige Auswahl: Man weiß nicht, wann eine Phase zuende ist. Es gibt eine kleine Wahrscheinlichkeit, mit welcher ein Neuron nie upgedatet wird.
  - zufällige Permutation: Hier wird jedes Neuron upgedatet, allerdings ist die Reihenfolge zufällig gewählt. Wohl am vorteilhaftesten.
  - topologische Sortierung: Bei einfachen Feed-Forward-Netzes – wie das MLP – können wir eine topologische Sortierung wählen. Wir rechnen die Neuronen der einzelnen Schichten neu. Wenn eine Schicht fertig ist, kommt von oben nach unten die nächste Schicht dran.

5.6.6 Einstellen der Lernrate

Gute Werte für η sind $0,01 \leq \eta \leq 0,5$. Typisch für η ist $\eta = 0,1$.

- Eine zu hohe Lernrate verursacht, dass wir über Schluchten hinwegspringen und dass wir Oszillation in Schluchten haben.

- Eine zu kleine Lernrate verursacht, dass wir auf Plateaus verschmachten und endlos lange lernen. Des weiteren muss man bei zu kleiner Lernrate generell lange lernen.

Wir können, nachdem wir grobes Lernen mit einer hohen Lernrate gemacht haben, auf eine kleinere Lernrate herunterwechseln. Damit können wir noch etwas besser werden.

- Allerdings sollte man die Lernrate nicht vom Fehler abhängig einstellen, da es so sein kann, dass wir auf der Fehleroberfläche auf einem Plato stecken bleiben.

[11] Denn sonst wären sie ja gleich.

- Man sollte sie auch nicht von der Zeit abhängig machen. Es kann so sein, dass die Lernkurve super aussieht, wir aber in Wirklichkeit irgendwo auf halber Strecke liegen bleiben, obwohl wir noch weiterlernen könnten.

Was wir machen können, ist zu einem festen, gut gewähltem Zeitpunkt die Lernrate um z.B. $\frac{1}{10}$ zu verkleinern. In der Lernkurve werden hier Zacken sichtbar. Der Fehler wird noch einmal geringer.

Für **Neuronen mit linearer Kennlinie benötigen wir generell eine kleinere Lernrate** als für solche mit sigmoider Kennlinie:

- Bei einer sigmoiden Kennlinie können wir Pech haben und wir sind mit den vorinitialierierten Werten genau auf der falschen Seite. Beim Drehen an den Gewichten ändert sich der Fehler nur sehr langsam. Haben wir dazu noch eine kleine Lernrate, braucht Lernen ewig. (siehe dazu auch Flat-Spot-Elimination-Verfahren)

- Bei einer linearen Kennlinie können wir auf dieser sehr schnell hin und her rutschen. Um dies nicht allzu heftig zu tun, wählen wir eine kleinere Lernrate.

5.6.7 Probleme und deren Lösungen

- *Laufen in ein Nebenminimum*: Dies ist ein großes Problem neuronaler Netze. Es kann sein, dass auf ein Nebenminimum hin optimiert wird, statt auf das globale Minimum zu optimieren.
  - Abhilfe verschafft hier, dass man das Netzwerk mit mehreren unterschiedlichen zufällig erzeugten Startwerten lernen läßt. Liefern alle dasselbe Ergebnis, so ist dies höchstwahrscheinlich das globale Minimum.
  - Weiterhin kann man **simulated annealing** anwenden. Hierbei wird eine Temperatur definiert, die langsam abkühlt und die Lernrate steuert. Zu Beginn werden noch große Sprünge über die gesamte Fehleroberfläche erlaubt, so dass Nebenminima übersprungen werden. Wenn das Lernen weiter fortgeschritten ist, wird immer weiter abgekühlt, bis das Netz letztendlich in das globale Minimum läuft. Wichtig hierbei ist, dass man die Lernrate nicht abhängig vom Fehler oder der Gewichtsänderung macht.

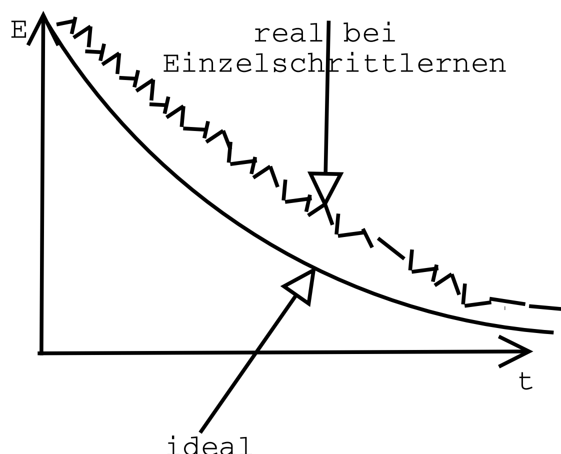
- *Parallele Änderung der Gewichte*: Wir müssen die Gewichte mit unterschiedlichen zufällig gewählten Werten initialisieren. Diese sollten in der Nähe von 0 liegen, damit die Sigmoide Kennlinie sich für eine Seite entscheiden kann, ohne endlos zu rechnen. Diese Werte müssen des weiteren verschieden voneinander sein. Wenn sie nicht verschieden sind, dann passiert es, dass das Backpropagation-Verfahren auf allen Neuronen einer Schicht das Gleiche macht. **Symmetrie Breaking**

- *Flat Spot Elimination*: Wenn wir weit links oder weit rechts auf der Kennlinie einer sigmoiden Funktion sind, dann brauchen wir endlos lange, um daraus wieder herauszukommen. Wenn wir bei der Initialisierung gerade auf der falschen Seite angefangen haben, dann haben wir Pech gehabt. Abhilfe verschafft hier die Flat Spot Elimination. Es wird auf die Ableitung der Kennlinie ein Wert aufaddiert, welcher die Kennlinie kippt, so dass auch weit Links und weit Rechts gelernt wird.

- *Oszillation*: Das Netzwerk kann in Schluchten anfangen zu oszillieren, d.h. es springt immer von einer Seite zur anderen Seite der Schlucht, nimmt aber nie das Minimum an. Dies kann man verhindern, indem man immer eine andere zufällige Permutation der Trainingsmuster verwendet. Benutzt man immer die gleiche Sequenz, kann es sein, dass das Netzwerk immer dasselbe macht.
- *Steckenbleiben auf flachen Plateaus*: In Minima und in Maxima ist der Gradient 0. Wenn wir ein flaches Plateau haben, wo der Gradient nur einen kleinen Betrag hat, kann es sein, dass man auf diesem Plateau steckenbleibt.

Für Oszillation und Steckenbleiben auf flachen Plateaus schafft der Momentum[12]-Term Abhilfe.

5.7 Lernkurve



Haben wir eine logarithmische y-Achse, so ist die ideale Lernkurve linear. Setzen wir die Lernrate während des Lernes kleiner, so kann man bei logarithmischer y-Achse Stufen in der Lernkurve entdecken.

5.8 Hinzunahme von Neuronen

Die Frage nach der Anzahl der Neuronen im Hidden Layer ist eine schwierig zu beantwortende Frage. Hier hilft nur Erfahrung oder Ausprobieren. Wählen wir im Hidden Layer zu wenige Neuronen kann das Netz vielleicht die Aufgabe nicht lernen und es kommt Müll heraus. Wählen wir zu viele Neuronen generalisiert das Netz schlecht oder gar nicht und lernt nur auswendig. Es gibt lange Lernzeiten und viele Nebenminima.

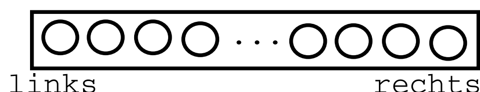
Am Besten fährt man, indem man am Anfang wenige Neuronen nimmt und nach und nach Neurone hinzufügt. Bei wenigen Neuronen stellt sich der Misserfolg schon nach kurzer Rechenzeit ein, so dass man schnell reagieren kann.

5.9 Codierung des Outputs

Für viele Anwendungen ist es sinnvoll, die Outputs geschicht zu kodieren.

[12]Momentum – Schwung

- Ein Netzwerk, welches überprüft, ob ein Staubsauger defekt ist, sollte nicht nur ein Outputneuron haben, welches aktiv oder inaktiv ist. Es gibt dann einen scharfen Wert, wo gesagt wird, dass er defekt oder okay ist. Besser ist es jedoch einen Bereich zu haben, in welchem man sagt, dass der Staubsauger vielleicht defekt, vielleicht in Ordnung ist. Gelöst wird dies mit 2 Neuronen. Ist eines von ihnen an, so sagt dass, dass der Staubsauer defekt oder okay ist. Ist keins oder beide an, so sagt dass, dass das Neuronale Netz es nicht weiß.
- **Place Coding**: Bei einer Fahrzeugsteuerung sagt nicht ein Neuron, wo das Lenkrad hinzusteuern ist, sondern ein ganzes Array von Neuronen. Es wird dahin gesteuert, wo die Wolke der aktivierten Neuronen ist[13].



5.10 Momentum-Term

Der Momentum-Term löst das Oszillieren in steilen Schluchten und das Stehenbleiben auf flachen Plateaus. Der Momentumterm addiert einen Teil der vorherigen Gewichtsänderung zu der aktuellen Gewichtsänderung hinzu:

$$\Delta w_{ij}(t+1) = \eta \delta_j \tilde{o}_i + \alpha \Delta w_{ij}(t)$$

Dabei kann α zwischen $0 \leq \alpha < 1$ liegen. $0,1 \leq \alpha \leq 0,5$ ist aber typisch.
Der Momentum-Term verursacht

- *Beschleunigung auf flachen Plateaus*: Es wird sozusagen Gas gegeben. Der Gradientenvektor hat immer dasselbe vorzeichen, so dass nach und nach sich ziemlich viel zusammenaddiert.
- *Unterdrückung der Oszillation in Schluchten*: Bei Oszillation wechselt der Gradientenvektor immer das Vorzeichen. In einem solchen Fall verkleinert der Momentum-Term die Gewichtsänderung und wir springen wahrscheinlich direkt in die Schlucht.

Den Momentum-Term nur bei Mustererkennung anwenden, weil wir hier eine scharfe Fehleroberfläche haben (wegen der Klassifizierung der Muster). Bei der weichen Fehleroberfläche der Funktionsapproximation ist der Momentum-Term eher kontraproduktiv. Wenn wir einen Momentumterm benutzen, können wir die Lernrate η erhöhen.

5.11 Verfahren höherer Ordnung

5.11.1 Quickprop

Quickprop nutzt aus, dass die Fehleroberfläche quadratisch ist[14]. Nach einem Backprop-Schritt – wobei wir uns den Fehler merken, der vor diesem Schritt aktuell war – können wir mit Hilfe des neuen Fehlers das Minimum der Parabel berechnen und direkt ins Minimum springen.

Allerdings haben wir mehrere Fälle, wo wir Quickprop nicht anwenden können:

[13] Wenn es links und rechts eine Wolke gibt, heißt dass wahrscheinlich, dass es in der Mitte einen Baum gibt, der tunlichst nicht angesteuert werden sollte. Mittelwertbildung ist also schlecht

[14] Sie ist quadratisch wegen der Definition des Fehlers

- Parabel ist falsch herum. Ihr Extrempunkt, in welchen wir hineinspringen würden, ist ein Maximum.
- über steile Schluchten wird hinweggesprungen
- ...

Backprop wird dann angewandt, wenn wir kein Quickprop anwenden können. Wollen wir all diese Fallunterscheidungen machen, lohnt sich Quickprop gar nicht mehr und wir machen lieber Backprop.

5.11.2 Zwei Gewichte

Wir springen gleich mit zwei Gewichten ins Minimum. Wir benötigen die Hessische Matrix. Diese ist relativ gross.

5.11.3 Analytische Methode

Wir können mit Hilfe einer Funktionalmatrix (gigantisch gross) das Problem sofort ausrechnen. Dies können wir mit dem Levenberg-Marquardt-Verfahren.

5.12 Optimal Brain Demage

Problem: Netze mit wenigen Neuronen können Probleme manchmal nicht lernen. Netze mit zuvielen Neuronen lernen auswendig und generalisieren nicht. Optimal Brain Damage ist ein Verfahren Gewichte und Neuronen wegzustreichen. Dabei wird mit einem Term gelernt, der aus zwei sich wiederstrebenden Zielen besteht

$$\underbrace{\frac{1}{2}\sum(t-o)^2}_{\text{Fehler ist zu minimieren}} \qquad + \qquad \underbrace{\gamma\sum_w(w)}_{\text{Gewichte gegen 0}}$$

Durch das Bestreben den Fehler zu minimieren, werden die Gewichte von 0 weggedrückt. Einer der beiden Terme ist nun stärker. Wenn das Gewicht tatsächlich benötigt wird, wird es nicht 0 werden. Wenn das Gewicht nicht benötigt wird, wird es gegen 0 tendieren und es kann gelöscht werden. Ein Neuron mit allen Gewichten auf 0 kann gelöscht werden.
Die Lernregel wäre

$$\Delta w_{ij} = \eta\delta_j\tilde{o}_i - \epsilon w_{ij}$$

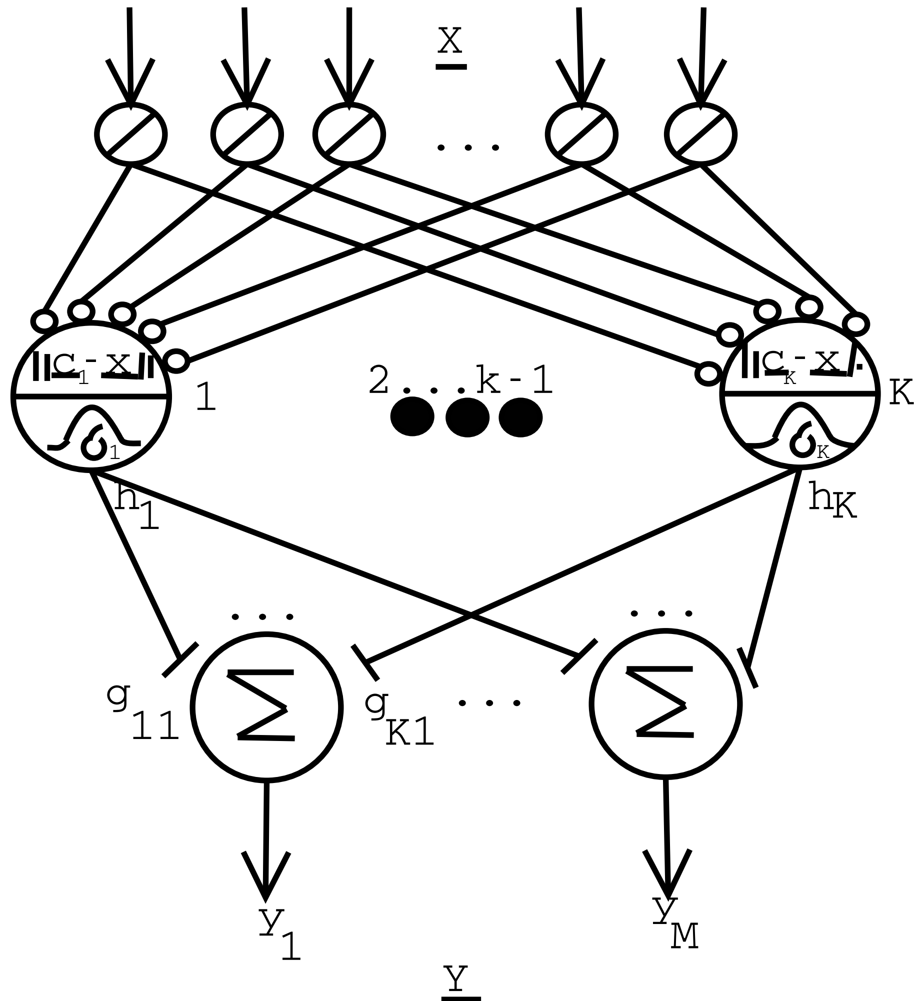
oder

$$\Delta w_{ij} = \eta\delta_j\tilde{o}_i \text{ und } \Delta w_{ij} = \Delta w_{ij}(1-\epsilon)$$

6 RBF-Netze

RBF-Netze (ausgeschrieben: Radial Basis Funktionen-Netze) sind eine andere Herangehensweise an das Problem der Mustererkennung und Funktionsapproximation.

6.1 Aufbau



- $\underline{X} \in \mathbb{R}^N$, $\underline{Y} \in \mathbb{R}^M$
- Das RBF-Netz besteht aus einer Inputschicht mit N Neuronen, einer Hiddenschicht mit K Neuronen und einer Outputschicht mit M Neuronen. Somit klassifiziertes N-demenisonale Muster gemäß eines M-dimensionalen Outputraums.
  - *Inputschicht:* Die Inputschicht ist eine reine Weiterleitung. Jedes Neuron verteilt seinen Wert (siehe Identität in den Neuronen) an alle Neuronen der Hiddenschicht
  - *Hiddenschicht:* In der Hiddenschicht wird in jedem Neuron der Abstand zwischen der Eingabe und dem Zentrum c mit Hilfe einer Norm gebildet:

    $$||\underline{c}_k - \underline{x}||$$

    Danach wird dieser Abstand z.B. durch eine Gauss-Kennlinie geschickt:

    $$h_k = h_k(||\underline{c}_k - \underline{x}||) = \exp\left(-\frac{1}{2\sigma^2}||\underline{c}_k - \underline{x}||^2\right)$$

    Die Gausskennlinie – wie oben geschrieben – ist abhängig von einem Parameter σ der die Breite der Glocke angibt. Dieses σ muss für jedes Zentrum gewählt werden.

Die Gaussglocke verursacht, dass das Neuron, wenn Zentrum und Muster nahe beieinander liegen, sehr viel stärker als die lineare Kennlinie reagiert, wenn die Zentrum und Muster weiter auseinander liegen, sehr viel schwächer.

- *Outputschicht:* In der Outputschicht wird eine gewichtete Summe gebildet. Für das erste Neuron der Outputschicht wäre dies also

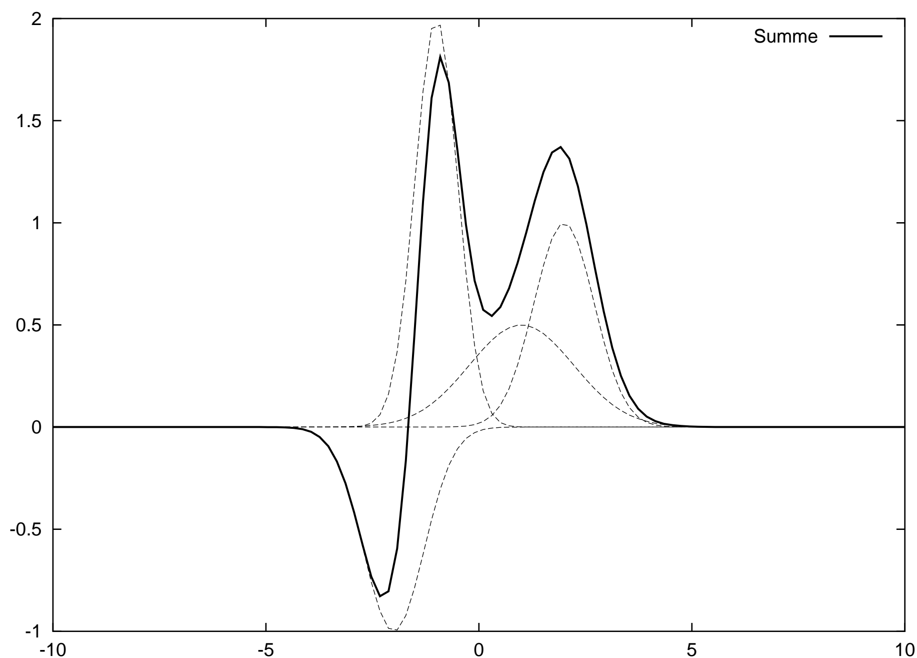
$$y_1 = \sum_{k=1}^{K} g_{k1} h_k$$

Die Neuronen der Outputschicht, sowie auch alle anderen Neuronen, haben keinen BIAS.

- Die für das Lernen einzustellenden Werte, die wir nun haben:
  1. **die Zentren**
  2. **die Breiten** σ
  3. **die Gewichte** g

6.2 RBF-Netze sind unverselle Funktionsapproximatoren

RBF-Netze sind universale Funktionsapproximatoren für Funktionen $\mathbb{R}^N \longrightarrow \mathbb{R}^M$. Sie setzen Funktionen als Überlagerung von Radialen[15] zusammen:



Hier sehen wir einen großen Vorteil von RBF-Netzen. Während MLPs in den Bereichen, wo keine Stützstellen für die Funktion vorhanden sind, irgendetwas machen können, interpoliert das RBF zwischen den Stützstellen, so dass das RBF keinesfalls hier ganz extreme Werte ausgeben wird.

[15] z.B. Gaussglocken

6.3 Einstellen der Gewichte

Angenommen wir hätten die Zentren und die Breiten der Gaussglocken schon. Wir wollen nun „nur noch" die Gewichte g_k an unserem Outputneuron einstellen, so dass die Zielfunktion approximiert wird. Wir machen dies nur an einem Outputneuron. Die anderen werden analog eingestellt. Die Anzahl der Neurone ist K, die Anzahl der Trainingsbeispiele ist P:

- $K > P$: Dieser Fall ist uninteressant. Wir können die Gewichte auf viele unterschiedliche Arten einstellen. Dieser Fall ist auch selten, da wir im Allgemeinen mehr Trainingsmuster haben, als Neuronen.
- $K = P$: Für den Output des Netzes haben wir folgende P Gleichungen

$$\begin{aligned} {}^{p=1}y &= \sum_{k=1}^{K} g_k h_k \left(||{}^{p=1}\underline{x} - \underline{c}_k|| \right) \\ {}^{p=2}y &= \sum_{k=1}^{K} g_k h_k \left(||{}^{p=2}\underline{x} - \underline{c}_k|| \right) \\ &\dots \\ {}^{p=P}y &= \sum_{k=1}^{K} g_k h_k \left(||{}^{p=P}\underline{x} - \underline{c}_k|| \right) \end{aligned}$$

  In Matrixschreibweise ergibt dies

$$\underline{Y} = \underbrace{\underline{\underline{H}}}_{P \times K} \underline{G}$$

  Wir stellen dieses Gleichungssystem um, indem wir die Abbildung $\underline{\underline{H}}$ invertieren und erhalten für G ($D = Y$ mit D für Desired Output)

$$\underbrace{\underline{G}}_{K \times 1} = \underbrace{\underline{\underline{H}}^{-1}}_{K \times P} \cdot \underbrace{\underline{D}}_{P \times 1}$$

  Dass die inverse Matrix existiert wird vorausgesetzt. Praktisch gesehen wird auch keine Matrixinversion durchgeführt, sondern es werden andere Verfahren zum Lösen großer Gleichungssysteme eingesetzt.

- $K < P$: Der eigentlich interessante Fall: Es gibt viel mehr Muster als Neurone. Es gibt mehrere Möglichkeiten die Gewichte anzupassen:

  – Wir stellen wieder folgende Gleichung auf

$$\underline{Y} = \underbrace{\underline{\underline{H}}}_{P \times K} \underline{G}$$

  Nun haben wir aber ein Problem beim Invertieren, dass wir die Matrix H nicht invertieren können, da sie nicht quadratisch ist[16]. Wir weichen aus auf eine Pseudoinverse, die **Moore-Penrose Pseudoinverse**, die berechnet wird durch

$$H^{+} = \left(H^{T} \cdot H \right)^{-1} \cdot H^{T}$$

[16] Wir erinnern uns, nur quadratische Matrizen lassen sich invertieren, haben also Chancen als lineare Abbildungen Isomorphismen zu sein. Es gibt des weiteren auch quadratische Matritzen, die sich nicht invertieren lassen.

In Dimensionen

$$\underbrace{\underbrace{(K \times P \cdot P \times K)}_{K \times K}{}^{-1} \cdot (K \times P)}_{K \times P}$$

Die Moore-Penrose Pseudoinverse wird die Fehlerquadrate minimieren.
Also haben wir wieder

$$G = H^{+}D$$

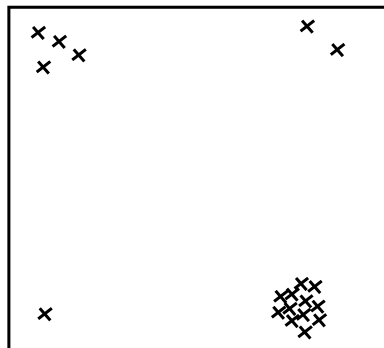
Hinweis: Die Berechnung mit der Moore-Penrose Pseudoinverse wird praktisch nicht durchgeführt, da diese Matritzen sehr hohe Dimensionen haben, wenn wir viele Trainingsmuster haben.

– Wir können auch den Gewichtsvektor G durch Lernen an Beispielen finden. Ein Gradientenabstieg ist möglich. Wir benutzen die δ-Regel. Kein Backpropagation, da wir nur in der Outputschicht lernen müssen.

6.4 Anpassen bzw. Wahl der Zentren und Breiten

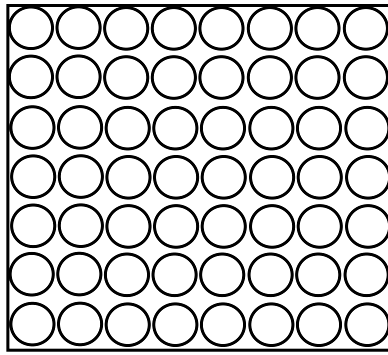
6.4.1 Zentren

Die Wahl der Zentren gestaltet sich sehr schwierig. Sie ist abhängig von der Beschaffenheit des Eingangsraumes.
Häufen sich die Eingabebeispiele an bestimmten Stellen in diesem Raum, so ist es sinnvoll hier die Zentren zu plazieren. Wie ein solcher Eingangsraum (hier im Zweidimensionalen) beschaffen sein kann, zeigt untenstehendes Bild.



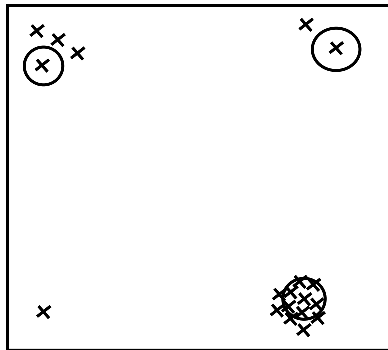
—

Wir können nun den Eingangsraum mit Zentren parkettieren, d.h. wir legen ihn gleichmäßig mit Zentren aus. Unten in der Grafik sehen wir feste Grenzen für die Gaussglocken. Zu beachten ist aber, dass die Gaussglocken, die wir hier von oben sehen, keine festen Grenzen haben. Geht man auf höhere Dimensionen, so wird die Parkettierung mit Gaussglocken immer schwieriger. Die Gaussglocken-Anzahl nimmt enorm zu, weshalb dieses Verfahren ein schlechtes Verfahren ist.



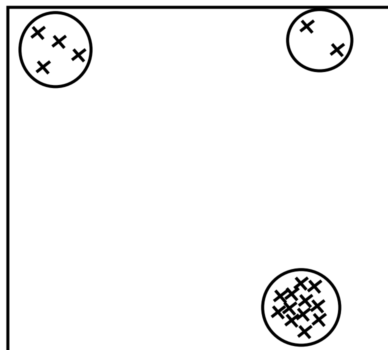
—

Wir können auch Zentren durch den Zufall aus den vorhandenen Trainingsdaten auswählen. Das ist zwar leicht zu implementieren, meistens klappt diese zufällige Auswahl jedoch sehr schlecht.



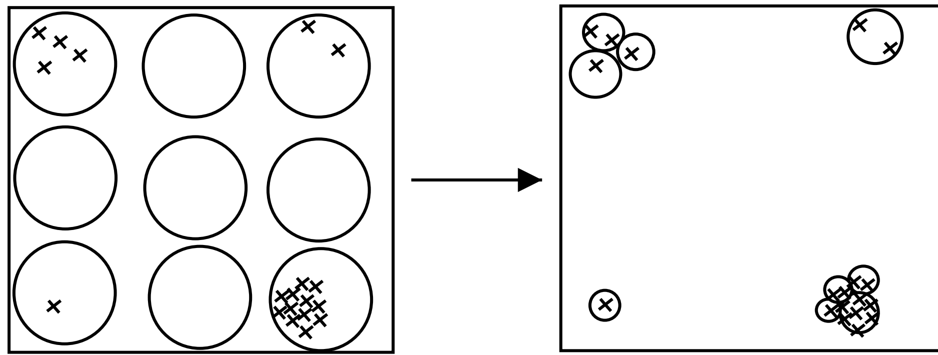
—

Wir können vorher eine Clusteranalyse machen. Die Cluster-Analyse hat jedoch den Nachteil, dass man am Anfang nicht weiß, wieviele Cluster man hat, was man aber dem Verfahren sagen muss. Auch auf höherdimensinalen Räumen versagt das Clustering-Verfahren.



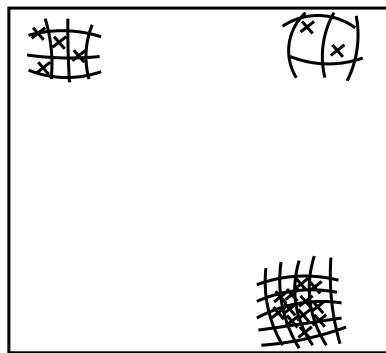
———

Wir können die Zentren mit Hilfe von LVQ (Lernende Vector Quantisation) an den richtigen Ort schieben.



—

Oder wir können SOMs (Self organizing maps), Neuronales Gas oder m-SOMs anwenden. Unten sind m-SOMs gezeigt.



———

Ein Gradientenabstieg kann auch gemacht werden, ist aber ungenauer und instabil.

6.4.2 Breiten

Wir können die Breiten σ mit Hilfe einer Heuristik berechnen. Entweder

$$\sigma = \frac{d}{\sqrt{2K}}$$

oder

$$\sigma = \frac{1}{3}d$$

wobei d der Abstand.

6.5 Probleme

Hohe Eingangsdimensionen sind sehr **problematisch**. Wir benötigen dafür sehr viele Zentren bzw. Neurone in der Hidden Schicht, da in höherdimensionalen Räumen die Weite des Raumes immer größer wird. Dies führt zu extrem hochdimensinalen Matritzen, sie sich schwer rechnen lassen oder zu extrem vielen Gewichten an den Outputneuronen, die wir einstellen müssen.

Hohe Ausgangsdimensionen sind **unproblematisch**, da nur eine Summe gebildet werden muss, bzw. die Gewichte mit der Delta-Regel eingestellt werden müssen.

6.6 Der entscheidende Unterschied zwischen RBF-Netzen und MLPs

Bei MLPs wird eine Hyperebene oder mehrere Hyperebenen durch den Eingangsvektorraum gelegt. Bei RBF-Netzen werden im Raum Gaussglocken plaziert, die deren Umgebung klassifizieren.

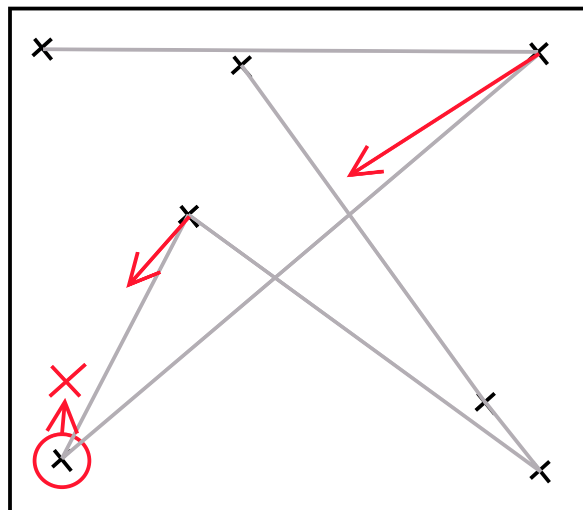
7 SOMs (Self Organizing Maps)

7.1 Idee

SOMs können für RBF-Netze zum Lernen der Zentren verwandt werden. SOMs haben ein **unüberwachtes Lernen**, d.h. es gibt **keinen Teacher / Desired Output**. Interessant ist die Struktur des Netzes bzw. der Ort der Zentren, nicht, was das Netz ausgibt.

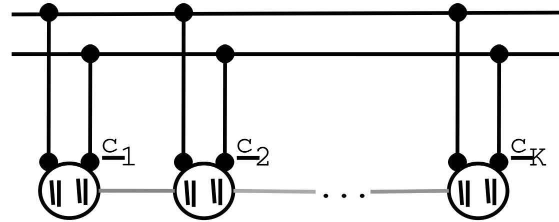
7.2 Abstand

Auf dem Eingaberaum gibt es eine Abstandsfunktion. Es wird jeweils ein Beispiel (Stimulus) eingegeben und der Abstand zu allen vorhandenen Zentren berechnet. Dasjenige Zentrum, welches am nächsten an dem Beispiel dran ist, ist der Gewinner und wird aktiviert (**Winner Takes All**). Eventuell werden aufgrund der Nachbarschaftsfunktion benachbarte Neuronen zusätzlich in geringerem Maße mit aktiviert.



Die Nachbarschaftsfunktion – hier eine eindimensionale Kette – ist grau angedeutet. Das rote Kreuz ist der Stimulus. Die roten Pfeile verdeutlichen, wie sich die einzelnen Zentren bewegen. Das rot umkreiste Zentrum ist der „Winner". Hierbei ist bemerkenswert, dass sich die weiter entfernten Zentren mehr bewegen als das nahe Zentrum (der Gewinner). Die Zentren überholen jedoch das nahe Zentrum nicht.

7.3 Nachbarschaftsfunktion am RBF-Netz visualisiert



Die Nachbarschaftsfunktion ist die graue Linie, eine Kette.

7.4 Lernregel

7.4.1 Lernregel

$$\Delta c_k = \eta(t) \cdot h(i,k,t) \cdot ({}^p\underline{x} - \underline{c}_k)$$

$$c_k + = \Delta c_k$$

Dabei ist

Δc_k	Die Änderung für das Zentrum k (ein Vektor!!)
$\eta(t)$	die Lernrate in Abhängigkeit von der Zeit
$h(i,k,t)$	die Nachbarschaftsfunktion. i das Winner-Neuron, k das aktuelle Neuron und t die Zeit
$({}^p\underline{x} - \underline{c}_k)$	der Abstand vom Stimulus (als Vektor!!)

7.4.2 Nachbarschaftsfunktion

Die Nachbarschaftsfunktion ist zum Beispiel die Gaussglocke

$$h(i,k,t) = \frac{1}{\sigma(t)\sqrt{2\pi}} \cdot exp\left(-\frac{||\underline{c}_i - \underline{c}_k||^2}{2\sigma^2(t)}\right)$$

Andere wären

- Eine andere Gaussglocke: $e^{\left(-\frac{z}{d(t)}\right)^2}$
- Ein Zylinder: $\begin{cases} 1 & z < d(t) \\ 0 & \text{sonst} \end{cases}$
- Ein Kegel: $1 - \frac{z}{d}$
- Eine Halbwelle: $\begin{cases} \cos(\frac{z}{d}\frac{\pi}{2}) & z < d(t) \\ 0 & \text{sonst} \end{cases}$
- Ein Mexican Hat

7.4.3 Veränderung der Lernrate mit der Zeit

Es ist sinnvoll mit der Zeit das Netz erstarren zu lassen, indem wir die Lernrate oder die Breite der Gausskurve heruntersetzen. Lassen wir das Netz erstarren, stabilisiert sich das Gitter.

Für die Lernrate η:

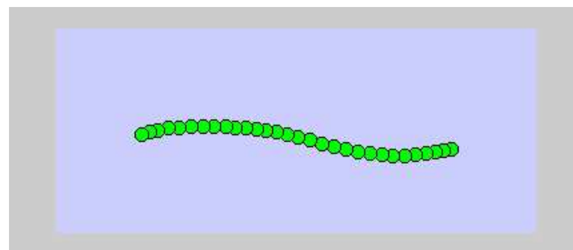
$$\eta(t) = \eta_{Start} \cdot \left(\frac{\eta_{Ende}}{\eta_{Start}}\right)^{\frac{t}{t_{max}}}$$

Für die Breite σ:

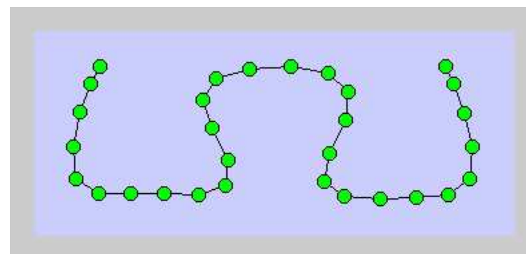
$$\sigma(t) = \sigma_{Start} \cdot \left(\frac{\sigma_{Ende}}{\sigma_{Start}}\right)^{\frac{t}{t_{max}}}$$

7.5 SOMs visualisiert

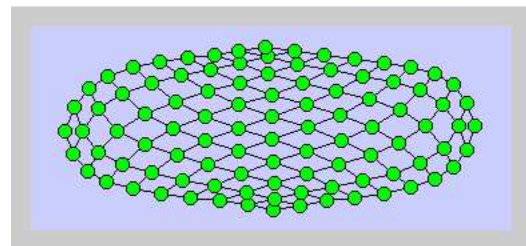
Ein eindimensionales Gitter, also eine Kette, fügt sich in ein zweidimensionales Rechteck. Stimuli kommen nur aus diesem Rechteck.
Nach ca. 400 Schritten



Bis zum Ende gelaufen:



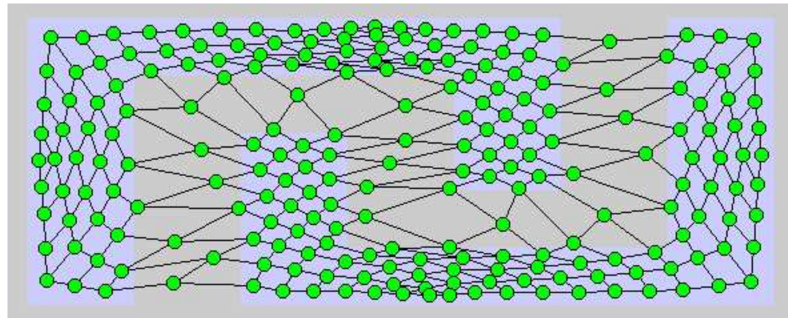
Ein zweidimensionales Gitter, also ein Netz fügt sich in ein zweidimensionales Rechteck.



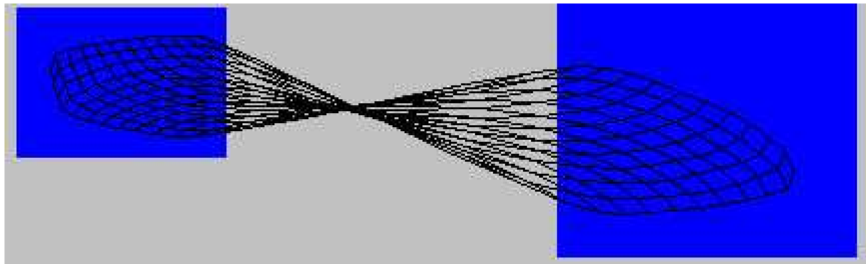
Hier wird ein Problem klar: Die Randneuronen werden nicht so wie die mittleren Neuronen gezogen, da bei ihnen einige Nachbarn fehlen. Hier muss die Lernrate angepaßt werden.

Probleme treten auf, wenn die Eingaben disjunkt sind. Es gibt also zwei (oder mehr) geometrische Figuren im Raum. Es liegen Neuronen an Stellen, wo überhaupt gar

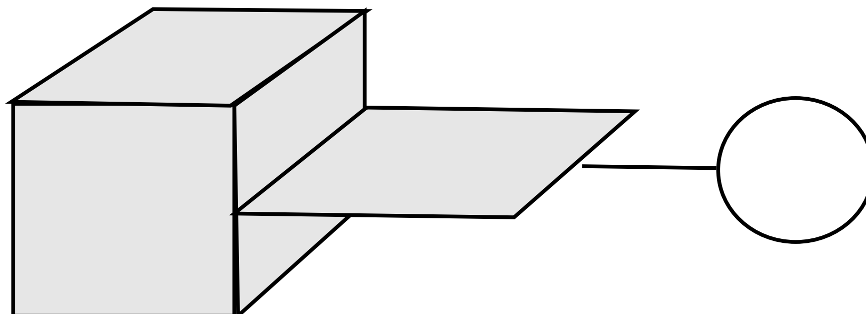
keine Eingaben stattfinden. Diese Neuronen sind sozusagen verschenkt. An Stellen, wo Eingaben stattfinden, häufen sich allerdings die Neuronen:



Bei mehrdimensionaler Gitterstruktur können schnell Verknotungen stattfinden, die sich nicht mehr lösen lassen.



Auf auf unterschiedlich-dimensionalen Strukturen wie folgender versagen SOMs



7.6 Anwendungen von SOMs

- SOMs sind erfolgreich in der Spracherkennung angewandt worden. Mit Hilfe eines zweidimensionalen Gitters werden Laute auf einem zweidimensionalen Eingangsraum abgedeckt. Ein gesprochenes Wort springt dann durch dieses Gitter durch. Durch die Neuronen, die es dabei aktiviert, kann man dass Wort dann bestimmen.

- SOMs kann man auch dazu benutzen, um Publikationen zu klassifizieren. Diese kommen aus einem hochdimensionalen Eingangsraum herein. Man legt eine zweidimensionale Gitterstruktur durch das SOM in diesen Raum hinein. Bestimmte Publikationen (z.B. über Programmiersprachen) landen in einer Ecke.

8 Neuronales Gas

8.1 Aufbau

Neuronales Gas sind SOMs ohne Gitterstruktur. Die Gitterstruktur wird ad hoc gebildet, indem man eine für jeden Lernschritt eine nach Abstand sortierte Liste der Neuronen erstellt und die jeweils nächsten Neuronen mitlernen lässt.

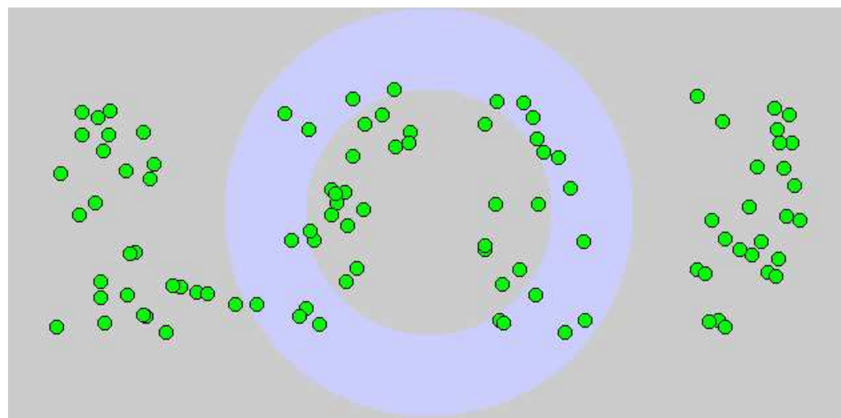
8.2 Vorteile

Der enorme Vorteil des Neuronalen Gases ist es, dass wir keine Verknotungen mehr erhalten. Es werden auch alle Neuronen in einen Bereich gezogen, in welchem auch Eingangssignale vorliegen. Der Nachteil ist natürlich, dass man kein Gitter mehr hat.

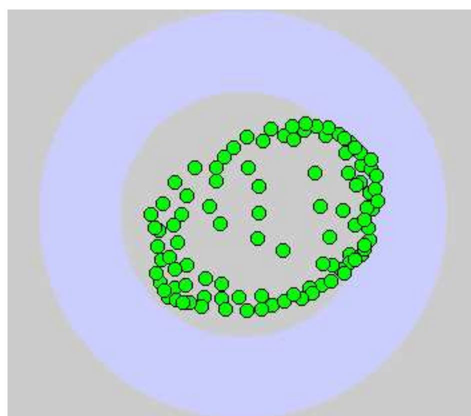
8.3 Die drei Phasen des Neuronalen Gases im Ring

Diese Phasen lassen sich auch an anderen Strukturen beobachten, aber am Ring besonders schön.

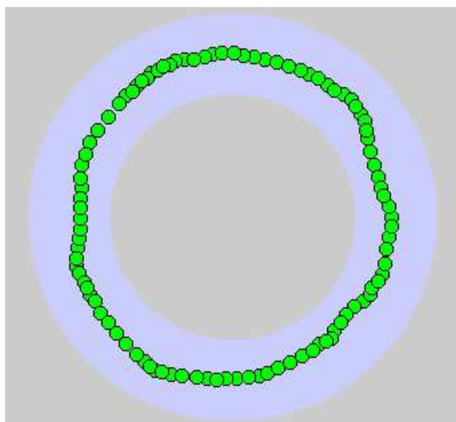
0.Phase: Initialisierung



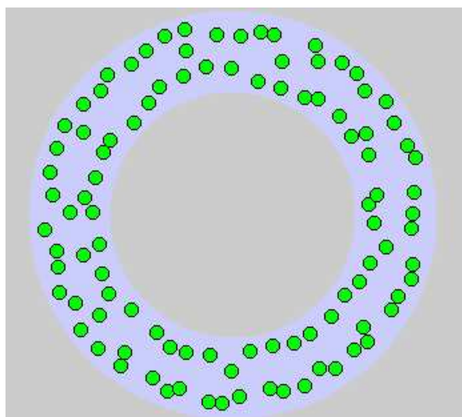
1.Phase: Alle Neuronen werden in den Kern gezogen. Dies ist das Mittel aller Änderungsvektoren



2.Phase: Es geht in den Ring. Hier ist der Fehler kleiner



3.Phase: Der Fehler wird weiter minimiert, indem die Kette im Ring aufbricht



9 LVQ (Learning Vector Quantisation)

9.1 Prinzip

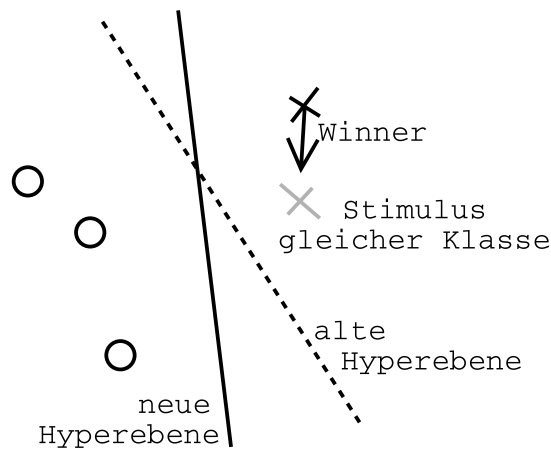
Learning Vector Quantisation ist ein supervised Lernverfahren, also mit Teacher. Es werden sogenannte Codebook-Vektoren angepaßt, so dass diese die Muster möglichst gut repräsentieren. Jeder Codebook-Vektor wird dabei schon vor dem durchführen des Lernens einer Klasse von Output zugeteilt.

9.2 LVQ (LVQ1)

Bei LVQ1 wird ein Stimulus x gewählt und es wird das Winner-Neuron, d.h. das nächstliegende Neuron bzw. Codebook-Vektor, bestimmt. Dieses Winner-Neuron c_k wird dann verschoben und zwar auf folgende Art und Weise:

$$\Delta c_k = \alpha(x - c_k) \quad \text{, wenn } x \text{ und } c_k \text{ zur gleichen Klasse gehören.}$$
$$\Delta c_k = -\alpha(x - c_k) \quad \text{, wenn } x \text{ und } c_k \text{ zu unterschiedlichen Klassen gehören.}$$

Das Gewinnerneuron wird also in Richtung Stimulus gezogen, wenn Stimulus und Neuron zur gleichen Klassen gehören, weggedrückt, wenn sie zu unterschiedlichen Klassen gehören. Die separierende Hyperebene verschiebt sich so.



9.3 LVQ2

LVQ2 ermittelt einen zweiten Gewinner, der nach dem ersten Gewinner noch mitlernt. Es wird nur gelernt, wenn x zur Klasse vom zweiten Gewinner gehört und nicht zur Klasse des ersten Gewinners.

Winner i	Gewinner
Second j	zweiter Gewinner, x gehört zu dieser Klasse

Der Gewinner wird weggedrückt, der zweite Gewinner angezogen:

$$\begin{aligned}\Delta c_i &= -\alpha(x - c_i) \\ \Delta c_j &= \alpha(x - c_j)\end{aligned}$$

9.4 OLVQ

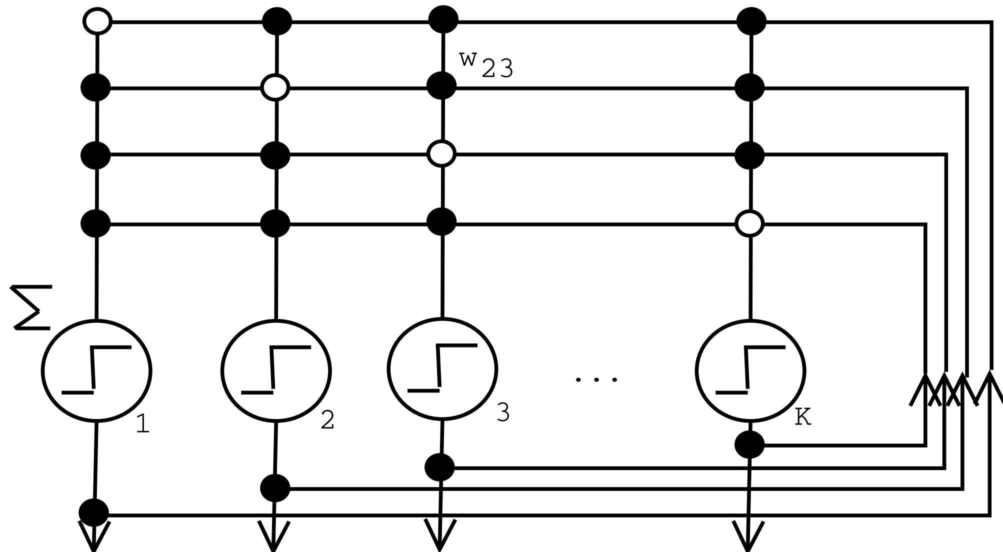
LVQ mit einer „optimised learning rate". Die Lernrate wird tiefpassgefiltert. Dies verhindert, dass, wenn die Zentren in der falschen Klasse landen, die Zentren hin und her geschleudert werden. So erzeugen wir langfristig eine schnellere Konvergenz. Für jedes Zentrum einzelne Lernrate

$$\text{Lernrate } \eta_k = \frac{\eta_k|_{t-1}}{1 + \eta_k|_{t-1}}$$

10 Hopfield Netze

Hopfield Netze sind von Herrn Hopfield erfunden worden. Sie tragen seinen Namen und haben nichts mit „hüpfenden Netzen" zu tun. Motiviert werden diese Netze durch Vorgänge in der Physik. Wir haben zum Beispiel in Kristallen ähnliche Strukturen.

10.1 Aufbau



- Hopfield-Netze sind vollverknüpfte Netze (jeder mit jedem) über symmetrische Verbindungen.
- Dabei ist jedes Neuron nicht mit sich selbst verknüpft. Die Zustände der Neuronen können entweder $+1$ oder -1 sein.
- Die Ausgabe ist in dem Zustand der einzelnen Neuronen gespeichert, wie hier mit Ausgabepfeilen angedeutet, und von der Gestalt $\{+1, -1\}^K$.
- Von jedem Neuron zu jedem Neuron gibt es eine symmetrische Verbindung mit Gewicht. Die Gewichte können wir in einer Gewichtsmatrix abspeichern. Da die Neuronen nicht mit sich selbst verknüpft sind hat diese Matrix auf der Diagonalen nur Nullen.
- Der Zustand x_k eines jeden Neurons k ist das Vorzeichen der gewichteten Summe über alle anderen Neuronen, wenn die Summe 0 ist, der alte Zustand

  $$x_k(t) = f\left(\sum_j w_{jk} x_j(t-1)\right)$$

  wobei

  $$f(z) = \begin{cases} +1 & z > 0 \\ x_k(t-1) & z = 0 \\ -1 & z < 0 \end{cases}$$

10.2 Lernen der Muster

Wir lernen die Muster, indem wir „Dellen" in die Gewichtsmatrix drücken. Die Lernregel ist motiviert, von der Hebbschen Lernregel. Wenn zwei ein Muster an zwei Stellen denselben Wert hat, erhöhe das Gewicht, wenn es unterschiedliche Werte gibt, vermindere es. Da wir nur Gewichte -1,$+1$ haben, können wir dies einfach durch Multiplikation tun:

y_i	y_j	$y_i \cdot y_j$
-1	1	-1
-1	-1	1
1	1	1

Somit ist die Gewichtsmatrix für ein Muster

$$^p(w_{ij}) = {}^p y_i \cdot {}^p y_j$$

und für alle Muster

$$(w_{ij}) = \sum_p^P {}^p(w_{ij}) = \sum_p^P {}^p y_i \cdot {}^p y_j$$

10.3 Asynchrone und synchrone Aktivierung

- Bei der **synchronen** Aktivierung werden alle Neurone aufeinmal in ihren neuen Zustand gebracht. Diese Aktiveriungsregel ist für mathematische Beweise besser zu handhaben, hat aber den Nachteil, dass Hopfieldnetze eventuell oszillieren[17].
- Bei der **asynchronen** Aktivierung wird der neue Zustand eines Neurons berechnet und dieses Neuron sofort auf den neuen Zustand gesetzt. Bringt Geschwindigkeitsvorteile. Oszilliert weniger (s.o.). Mathematisch unkorrekter (s.o.).

10.4 Aufnahmekapazität

Die Anzahl der stabil hinterher rekonstruierbaren Muster, die wir in dem Netz speichern können, ist arg begrenzt. Wenn die Muster orthogonal (also optimal) sind, ist die Speicherkapazität bei K Neuronen

$$0,14 \cdot K$$

10.5 Energieoberfläche

Die Energie eines Hopfieldnetzes zum Zeitpunkt t ist definiert durch

$$E(t) = -\frac{1}{2}\sum_i \sum_j w_{ij} x_i(t) x_j(t) + \underbrace{\sum_j \theta_j x_j(t)}_{\text{optional (1)}} - \underbrace{\sum_j in_j x_j(t)}_{\text{selten (2)}}$$

(1) θ_i ist ein Schwellwert und verschieb die Symmetrie der Muster. Muster mit wenigen Einsen kann man so in Richtung Eins verschieben.

(2) in_j ist eine Eingabe für das Netzwerk. Es wird ein Eingabemuster als Suchmuster angelegt. Zu einem ähnlichen Muster soll das Netz konvergieren.

10.6 Konvergenz

Man kann mit Hilfe der Energiefunktion zeigen, dass ein solches Hopfieldnetz wirklich konvergiert.

10.7 Unsymmetrische Gewichte

Autoassoziator: symmetrische Gewichte
Heteroassoziator: unsymmetrische Gewichte

[17] Ein Neuron wird dauernd gezwungen von $+1$ auf -1 zu flippen und umgekehrt.

10.8 Spezialhardware

- Für Hopfieldnetze gibt es Spezialhardware. Wir haben eine hohe Parallelisierung, deshalb können wir Höchst-Parallel-Rechner sehr gut einsetzen. Problem dabei ist, dass wir einen sehr großen Datenbus benötigen, da wir für sehr viele Neuronen Ausgangsleitungen haben. Glücklicherweise sind nur wenige Neuronen „on” (sparse loaded). Wir können erhelblich Datenbusbreite sparen, indem wir nur die Adressen der jeweiligen Neuronen, die „on” sind, übertragen.

- Optische Hardware: Mit Hilfe von zwei Linsen, eine zum Aufsplitten verteilen der Werte des Vektors und eine zum Zusammenaddieren und einer dazwischengeschalteten, an unterschiedlichen Punkten verschieden durchlässigen, Glasplatte als Matrix können wir ein das Hopfieldnetz optisch updaten. Der Ausgabelichtvektor kommt als Eingabelichtvektor immer wieder herein, bis wir den stabilen Zustand erreicht haben. Die optische Hardware arbeitet sehr schnell.

11 Jordan & Elmannetze

11.1 Aufbau

- Beim **Jordan**-Netz wird der Output des Netzes als Input wieder eingegeben. Der Output wird in sogenannte **Kontextzellen** eingegeben, die zur Inputschicht dazukommen. Beim Jordannetz sind die Kontextzellen mit sich selbst verknüpft.

- Beim **Elman**-Netz wird der Output der verdeckten Schicht wieder als Input wieder eingegeben[18]. Eine Rückkopplung der Kontextneurone findet nicht statt.

11.2 Lernen

Beim Lernen tritt das Problem auf, dass Backpropagation in eine Endlosschleife gerät, da wir endlos im Netz zurückpropagieren müssen. Wir können das auf zwei Arten verhindern:

- **Unfolding in Time**: Wir gehen die Endlosschleife nur eine bestimmte Zeit und propagieren nur eine bestimmte Zeit den Fehler zurück. Nach ungefähr 10 Unfolding-In-Time-Schritten ist es sowieso sinnlos weiter zu propagieren, da wir an die Rechengenauigkeit unseres Rechners angekommen sind[19].

- **Teacher forcing**: Bei Jordannetzen können wir während des Lernens die Rückkopplung einfach abschalten (**open loop**). Beim Abfragen (auch bei der Erzeugnung des nächsten Outputs) des Netzes schalten wir sie wieder ein (**closed loop** oder **rekurrent**). Bei der Erzeugnung des nächsten Outputs setzen wir die Kontextneurone auf den gewünschten Output des Schrittes davor[20].

[18] Die Neuronen der Hidden-Schicht haben also über die Kontextzellen eine Verknüpfung auf sich selbst.

[19] Unser δ wird immer kleiner.

[20] Wir warten also nicht lange, bis das Netz sich für das angelegte Beispiel endlich stabilisiert hat und sparen so Zeit.

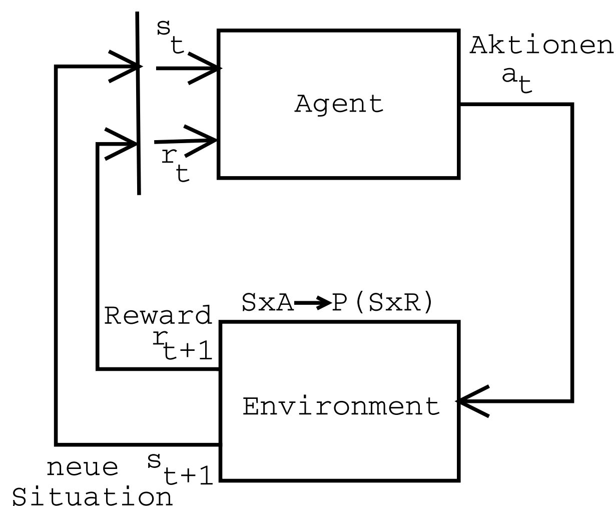
12 Reinforcement Learning (Bestärkendes Lernen)

Beim Reinforcement Learning (Übersetzt: Bestärkendes Lernen) gibt es anders als bei MLPs und RBF-Netzen keinen Teacher mehr, sondern nur noch einen Reward, der uns sagt, ob wir gut oder schlecht waren. Dies ist ein Mittelding zwischen Teacher und gar nichts, wie zum Beispiel bei SOMs und Neuronalem Gas.

12.1 Idee bzw. Aufgabe

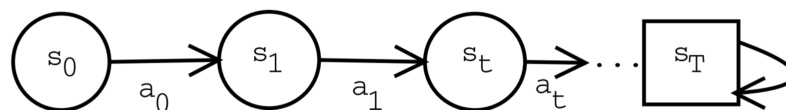
12.1.1 Agent in Environment

Reinforcement Learning wird auf einen **Agenten** angewendet, der auf ein **dynamisches Environment (System)** Einfluss ausübt und in diesem Enviroment bestimmte Aufgaben ausführen muss. Der Agent bildet also die momentane Situation $s \in S$ des Enviroments auf eine **Aktion** $a \in A$ ab. Für jede Aktion bekommt der Agent einen **Reward** $r \in \mathbb{R}$, der seine Aktion bewertet, aber nicht sagt, was er falsch gemacht hat oder was er gut gemacht hat. Beim Reinforcement Learning ist es das Ziel einen möglichst hohen Reward über eine lange Zeit zu erwirtschaften[21].



Dabei kann es für ein und dieselbe Aktion in dem gleichen Zustand mehrere unterschiedliche Rewards geben. In dem Environment ist der Zufall mit enthalten.

Die Kette von Situation/Aktion kann wie folgt dargestellt werden



12.1.2 Agent lernt eine Policy π

Beim Reinforcement Learning lernt der Agent eine Policy π, die ihm sagt, auf welche Art und Weise er auf die jeweils aktuelle Situation reagieren soll, um an sein Ziel zu kommen.

Es gibt zwei verschiedene Arten von Policys

[21] Dies ist kein Widersprung zum Erfüllen der Aufgaben, wenn man auf das Erfüllen der Aufgaben einen Reward aussetzt, bzw. das nicht Erfüllen der Aufgaben einen negativen Reward.

- **open loop policy, plan, Steuerung**: Wir machen uns zu Beginn einen Plan und führen diesen bis zum Ende aus.
- **closed loop policy, reactive plan, Kontrolle**: Die Umgebung übt während des Ausführen des Planes Einfluss auf den Plan aus.

12.1.3 Unterschied Zustand - Situation

- Ein **Zustand** eines Systems ist die Abspeicherung aller Variablen, die das System hat. Aus diesen Variablen läßt sich der Zustand komplett rekonstruieren. Ist es möglich Zustände eines Systems in jedem Schritt zu bekommen, so können wir die Policy mit Marcow Decision Problems (MDPs) komplett berechnen (im Prinzip).
- Bei den *meisten* Systemen kommen wir jedoch an alle Variablen gar nicht dran, weshalb die unvollständige Variablemenge nur eine **Situation** ist.

12.1.4 Reward und Return

Für eine Aktion zum Zeitpunkt t bekommt der Agent einen **Reward** r_t. Die Summe der zukünftigen Rewards ist der **Return** R_T

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T$$

Den exakten Return können wir jedoch nicht berechnen. Wir können nur einen Erwartungswert berechnen. Dies können wir auf zwei verschiedene Art und Weise

- Wir können die Aktionen zeitlich unterteilen in **Trials, Episodes**. Wir können den Erwartungswert des Returns einer Episode ermitteln.

  $$R_t = r_{t+1} + r_{t+2} + \ldots + r_T$$

  Wenn die Episode unendlich lang ist, dann hat dies keinen Sinn, da der Return unendlich wird.
- Wir können ein **discounting** einführen, um mit **delayed rewards**, zu denen wir nicht wissen, wann diese genau auftauchen, umzugehen. Wir gewichten in ferner Zukunft liegende Rewards weniger, als Rewards, die wir schnell erhalten. Wir wählen ein $0 \leq \gamma < 1$ und $T = \infty$ und berechnen diesen Reward wie folgt

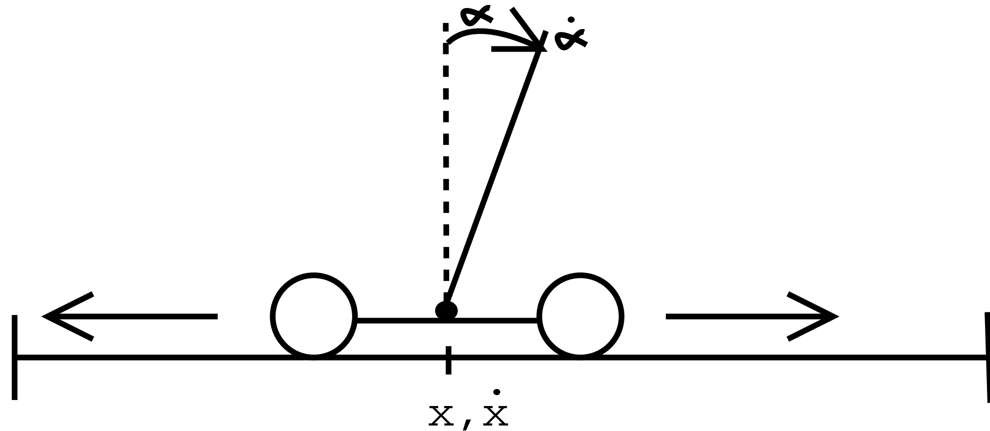
  $$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} = r_{t+1} + \gamma^1 r_{t+1+1} + \gamma^2 r_{t+1+2} + \ldots$$

12.1.5 Sonderfälle des Rewards

- **Pure delayed reward**: Ganz lange Zeit bleibt der Reward 0. Erst am Ziel wird ein positiver oder negativer Reward gegeben. *Kann zum Beispiel ein Spiel sein, wo erst am Ende dann feststeht, wer gewonnen hat.*
- **Jede Aktion kostet**: Jedes Mal, wenn der Agent eine Aktion macht, wird er bestraft: $r_t = -1$. Erst wenn der Agent am Ziel ankommt, wird er belohnt. Wenn nun zusätzlich in jedem Zeitschritt eine Aktion erzwungen wird, wird der Agent versuchen mit möglichst wenig Schritten am Ziel anzukommen.
- **Pure negative reward**: Es wird immer ein Reward 0 gegeben. Nur wenn der Agent etwas schlechtes macht, bzw. in einer schlechten Situation ist, gibt es einen negativen Reward.

12.1.6 Beispiele

- **TD-Gammon**: Reinforcementlearning zum Spielen von Backgammon. Anwendung von **Pure delayed reward**.
- **Auto in Senke**: Ein Auto befindet sich in einer Senke. Unser Ziel ist es, dass es aus der Senke herauskommt. Das Auto hat zu wenig Motorleistung um mit einem Anlauf aus der Senke herauszufahren. Wir bestrafen aber das Auto immer, wenn es sich in der Senke befindet. Erst am Ziel setzen wir den Reward auf 0: $r_z = 0$. (**2.Sonderfall**) Dieses System wird sich langsam hochschaukeln.
- **Pole Balancer**:



Der Polebalancer besteht aus

- einem **Wagen**, der nach links und rechts per **Bang-Bang-Control** beschleunigt werden kann. Dabei muss er immer in eine Richtung beschleunigt werden. Er kann nicht einfach stehen gelassen werden.
- Der Wagen hat eine Position x auf der Schiene mit einer Geschwindigkeit $\dot{x}$.
- An den Seiten der Schiene, auf der der Wagen steht, befinden sich Begrenzungen.
- Auf dem Wagen befindet sich ein **Pole** an einem Gelenk. Dieser hat einen Winkel α zur Senkrechten und eine Geschwindigkeit $\dot{\alpha}$.

Ziel das Pole Balancers ist es, den Stab gerade zu halten. Wir können dies mit Reinforcmentlearning mit einem **pure negativ reward** tun. Wir brauchen, um den Pole-Balancer zu steuern exakt ein Neuron. Um ihn zu trainieren, ein weiteres.

12.2 Value Function

12.2.1 State Value Function V^π

Die *State Value Function* bewertet eine Situation des Systems. Es wird bewertet, was wir aufgrund der Policy π aus diesem Status für ein Return haben können.

12.2.2 Action Value Function Q^π

Die *Aktion Value Function* bewertet eine Aktion des Agentens.

12.3 Greedy-Policy vs. Optimal Policy

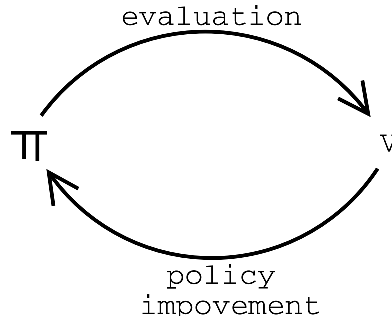
Unser Ziel ist es eine Optimal Policy zu finden, die unsere Aufgabe optimal löst.

- Bei einer **Greedy-Policy** nehmen wir die erstbeste Lösung. Selbst wenn diese Lösung besonders schlecht ist, wird diese Lösung dann immer genommen. Wenn wir die Greedy-Policy immer verwenden, haben wir eine reine **Exploitation (Ausnutzung)**.
  Beispiel für eine Greedypolicy: Wir haben einmal einen Weg zur Mensa gefunden, der zwar einmal um die ganze Stadt führt, durch den wir aber tatsächlich an der Mensa ankommen. Es gibt einen viel besseren Weg, nämlich direkt in die Mensa, welche das Nachbargebäude ist. Aber diesen nehmen wir nie und suchen wir auch nie, wenn wir eine Greedypolicy anwenden.

- Bei einer **Optimal Policy** versuchen wir den besten Weg zu finden. Dies ist eine **Exploration (Erforschung)**. Auch wenn uns die Valuefunction sagt, dass wir den Wert für eine Aktion/Situation maximieren, da wir hier schon einen Weg gefunden haben, probieren wir einmal eine andere Entscheidung aus. Um diese Entscheidung konstruktiv zu treffen, können wir den **Baum der Value-Function** entweder mit einer **Tiefensuche** oder **Breitensuche** explorieren. Schon gehabte Situationen/Aktionen können wir dabei abschneiden, da deren Ergebnisse bekannt sind.

12.4 Lernen der Value Function (Temporal Difference Learning)

Wir lernen die Value-Function durch Ausprobieren der Policy. Wir lernen also aus Erfahrung.



12.4.1 Was passiert?

Von einer Situation können wir in mehrere Unter-Situationen geraten. Von diesen Unter-Situationen können wir in weitere Unter-Situationen geraten. Wir haben einen **Baum**. Allerdings kann es in diesem Baum Verknüpfungen zwischen den einzelnen Ästen geben, da wir aus zwei verschiedenen Situationen in dieselbe Situation kommen können.
Normalerweise gibt es erst am Ende des Baumes in einem Blatt einen Reward (pure delayed reward). Die Value-Function **propagiert** nun den Wert der Blätter in die Knoten des Baumes hoch, so dass wir für jeden Knoten (Situation) einen Wert bekommen. Teilbäume von Knoten, deren Nachfolgesituationen wir alle schon kennen, weil wir sie schon „erfahren" haben, können wir abschneiden und durch den entsprechenden Wert der value function ersetzen.

Für die action value function stehen in den Knoten die Rewards für die Aktionen. Situation ist durch Aktion im obigen Text zu tauschen.

12.4.2 State Value Function V^π

$$_{neu}V(s_t) = {}_{alt}V(s_t) + \alpha[\underbrace{r_{t+1}}_{\text{empfangener Reward}} + \gamma\ {}_{alt}V(s_{t+1}) - \underbrace{{}_{alt}V(s_t)}_{\text{teilweise wieder abziehen}}]$$

12.4.3 Action Value Funktion Q^π

$$_{neu}Q(s_t, a) = {}_{alt}Q(s_t, a) + \alpha[r_{t+1} + \gamma \underbrace{\max_a {}_{alt}Q(s_{t+1}, a)}_{\text{Greedy-Policy}} - {}_{alt}Q(s_t, a)]$$

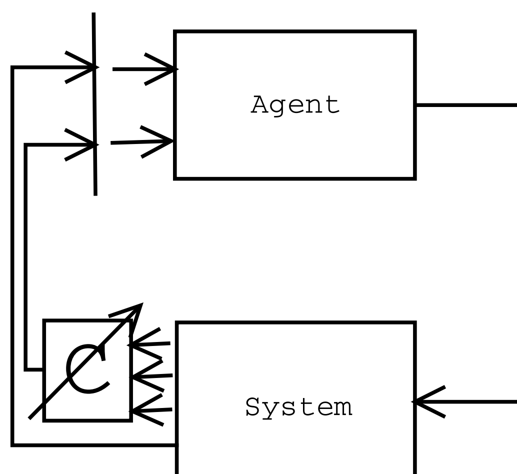
Die Action Value Function ist schneller als die State Value Function. Generell müssen wir aber im Hinterkopf behalten, dass Reinforcement Learning sehr langsam ist. Zur Zeit ist es aber wieder im Kommen, da wir immer größere Datenmengen haben.

12.4.4 Neuronales Netz zur Bestimmung der Value function eines Teilbaumes

Wir können auch ein neuronales Netz zur Bestimmung des Wertes eines Teilbaums einsetzen. Wir trainieren dieses Netz mit den Werten, die wir durch das Reinforcement Learning gewonnen haben.

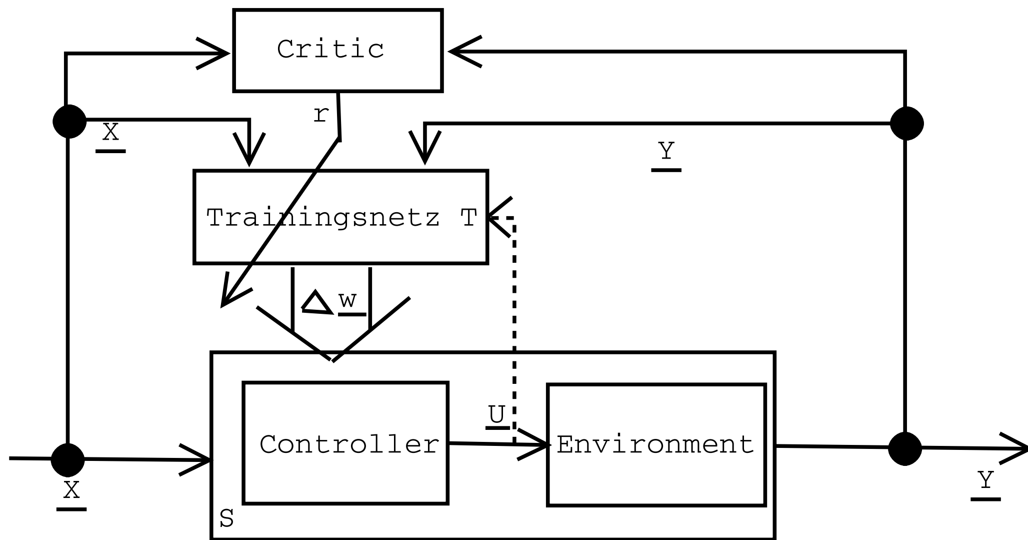
12.5 Reinforcment Learning mit (adaptive) Critic

Dies ist eine Unterart des RLs. Der Critic gibt einen Ersatz für das Reinforcmentsignal, indem er das System auswertet. Der Critic kann Adaptive sein, d.h. er lernt selbst noch.



12.6 SARO (Sensor Driven Random Optimisation)

Wir haben unter Umständen sehr komplizierte **Controller** (Agenten) mit vielen Parametern. Dieser Agent ist in diesem Falle ein neuronales Netz, z.B. zwei-Schicht Perzeptron. Ein neuronales Netz, das **Trainingsnetz**, **stellt die Parameter des Controllers ein**. Problem ist, dass wir für das Trainingsnetz gar keinen Teacher haben und so gar kein Backpropagation machen können. Wir lernen mit einem Critic. Dieser vergleicht die beiden letzten Rewards und sagt dem Trainingsnetz, dass es die neue Änderung lernen soll, wenn der Reward größer wurde. **Eine Änderung erstellen wir dadurch, indem wir ganz einfach Würfeln (SARO)**.



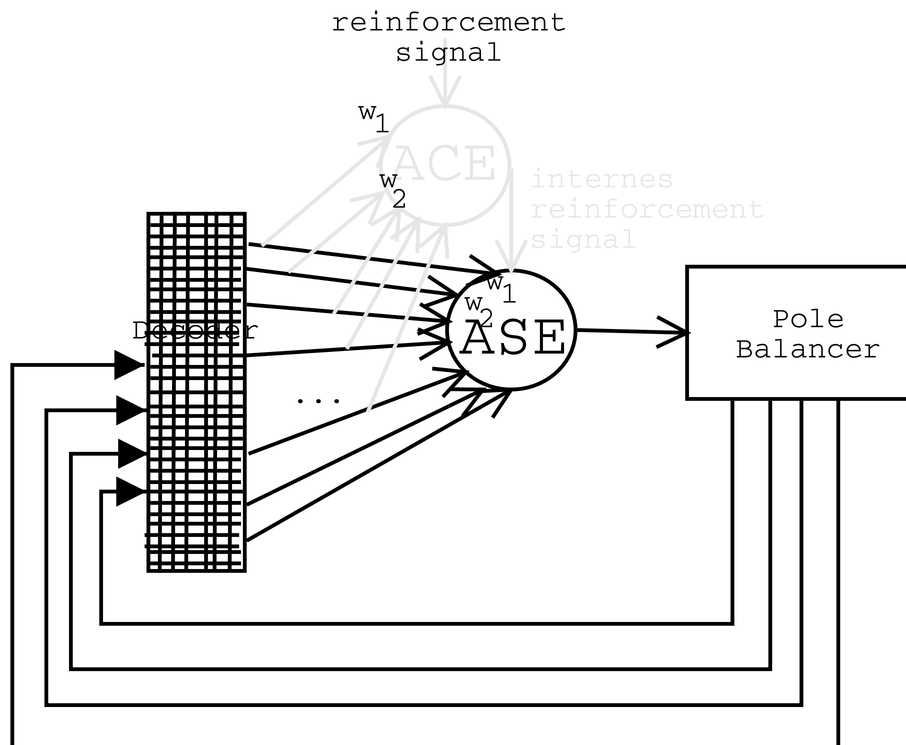
1. Output des Netzes $\underline{Y}$ erstellen
2. Δw durch die aktuellen Gewichtswerte des Trainingsnetzes berechnen
3. Diese Gewichtswerte auf die Gewichtswerte des Controllers aufaddieren:
$$w = w + \Delta w$$
4. Einen weiteren Output erstellen $\underline{Y}'$
5. Critic vergleicht die beiden Outputs und bewertet: Ist die Gewichtsänderung Δw gut, dann $r = 1$, schlecht, dann $r = -1$
6.
   - $r > 0$: Es war die richtige Richtung. T lernt mit modifizierter δ-Regel, indem wir alle Gewichte in die vom Δw vorgeschlagene Richtung drehen.
   - $r < 0$: Es wird verworfen. Wir drehen an allen Gewichten etwas zurück. Das reicht uns jedoch noch nicht, sondern wir addieren auf die Gewichtsmatrix eine zufällige Gewichtsmatrix mit kleinen Werten auf, um durch Würfeln rechtwinklig in dem Raum der Gewichte für den Controller abzubiegen.
7. Vergessen, d.h. Gewichte Richtung 0 drücken $\underline{\underline{G}} = (1 - \theta) \cdot \underline{\underline{G}}$

12.7 ASE - ACE

Das von Barto, Sutton und Anderson[22] vorgeschlagene System zur Steuerung der Bang-Bang-Kontrolle eines Pole-Balancers, besteht aus vier Elementen:

[22] Barto, Sutton und Anderson, *Neuronlike Adaptive Elements that can solve Difficult Learning Controll Problems*, IEEE transaction On Systems, Man and Cybernetics Vol. 13 pp. 834-846

1. Den **Pole-Balancer** selbst als **Enviroment**. Er exisitiert nicht real, sondern wird durch ein weiteres Programm simuliert.
2. Dem **Decoder**. Für die Eingabe in den Decoder werden die einzelnen Zustände des Pole-Bancers quantisiert. Der Decoder unterteilt sich in Boxen. Jede Box enthält einen möglichen Zustand des Systems. Die Aufgabe des Decoders ist es nun, jedem Zustand einer Box zuzuteilen. Dabei wird „winner-takes-all" angewendet. Eine Box wird aktiviert, alle anderen Boxen sind inaktiv.
3. Dem **ASE**.
4. und dem **ACE** (optional).



Der ACE ist grau hinterlegt, weil er optional ist. Ohne ACE ist das Reinforcmentsignal direkt.

Beschreibung von ASE und ACE:

- *ASE (Associative Search Element(Verbindendes Such Element))* Jede Box x_i des Decoders hat eine Verbindung zum ASE über ein Gewicht w_i. Weiterhin erhält das ASE das Reinforcement-Signal. Dies ist entweder positiv, wenn der Stab steht, oder negativ, wenn der Stab gefallen ist.
  Der Output y zum Zeitpunkt t wird nun bestimmt durch

  $$y(t) = f\left(\sum_{i=1}^{n} w_i(t)x_i(t) + noise(t)\right)$$

  Dabei ist f eine Schwellfunktion, eine sigmoide Funktion oder die Identität. $noise$ ist eine Zufallsvariable, normalverteilt um 0 mit der Varianz σ^2. Die Dichte dieser Normalverteilung hat die Formel ($\mu = 0$)

  $$f_{normal}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Die $noise-$Funktion verursacht, dass mit einer gewissen Wahrscheinlichkeit doch der andere Output, als durch die Gewichte bestimmt, gewählt wird. Dabei ist die Wahrscheinlichkeit, dass der andere Output gewählt wird, umso niedriger, je größer der Betrag des jeweiligen Gewichtes ist. Selbst wenn der Pole-Balancer gar nicht läuft und kein Eingabevektor anliegt, erzeugt das ASE so einen Output. Gelernt werden die Gewichte mit folgender Lernregel:

$$w_i(t+1) = w_i(t) + \underbrace{\alpha r(t) e_i(t)}_{\Delta w_i(t)}$$

Dabei ist

- α eine Konstante, die bestimmt, wie wechselwillig das Gewicht ist.
- $r(t)$ das Reinforcementsignal.
- e_i ein Wert der angibt, wie stark am jeweiligen Gewicht gedreht werden soll. Dieser Wert wird auf 1 gesetzt, wenn eine Aktion für den Pole-Balancer durch dieses Gewicht zum Zeitpunkt t stattgefunden hat. Dann sinkt dieser Wert gegen 0 mit der Zeit. Er zeigt an, wie weit in der Vergangenheit diese Verbindung aktiviert wurde, so dass die richtige Verbindung „bestraft" werden kann.

- *ACE (Adaptive Critic Element (Anpassbares kritisierendes Element))* Das ACE bekommt die Ausgabe des Decoders und erzeugt das Reinforcement-Signal für das ASE. Das ACE sagt vorraus, wie „gut" der momentane Zustand des Poles ist und gibt dies an das ASE weiter. Das ACE lernt auch, wenn der Pole nicht umfällt. Diese Vorhersage wird bestimmt durch

$$p(t) = \sum_{u=1}^{n} v_i(t) x_i(t).$$

Das Update der Gewichte v_i (eine Value Function) geschieht folgerndermaßen

$$v_i(t+1) = v_i(t) + \beta[r(t) - \gamma p(t) + p(t-1)]\overline{x}_i(t)$$

Dabei ist

- β eine Konstante, die bestimmt, wie wechselwillig das Gewicht ist.
- $r(t)$ das Reinforcementsignal.
- $\overline{x}_i(t)$ der „Trace" eines der Signale aus dem Decoder, ähnlich zu e_i.
- γ eine Rate, in welcher das externe Reinforcement genutzt werden soll.

Das Verfahren mit ASE und ACE ist gegenüber dem Box-Verfahren erheblich besser. Wo das Box-Verfahren nicht gut lernt, lernt das ASE/ACE-Verfahren schnell, so dass der Stab lange oben bleibt.

13 SVMs (Support Vector Machines)

13.1 Idee

Wir haben mehrere Verfahren zur Mustererkennung und Funktionsapproximation kennengelernt. Die Frage ist nun, was diese Verfahren gemeinsam haben? Die Frage ist

auch, ob wir eine generelle Überklasse bilden können, die alle Verfahren zur Mustererkennung und Funktionsapproximation beinhaltet. Die Support Vector Machines sind da ein Ansatz. Sie fassen mit Hilfe der Kernelfunctions für einige Neuronale Netze einige Neuronalen Netze zusammen.

Support Vector Machines kommen aus dem Gebiet des **Machine Learnings** und der **Statistical Learning Theory**.

13.2 Ziel

Unser Ziel ist es, wie bei Neuronalen Netzen, eine Mustererkennung zu machen. Bei Support Vector Machines fragen wir nach der Funktion f_α die unsere Beispiele alle richtig klassifiziert:

$$f_\alpha : \mathbb{R}^n \to \mathbb{B}^1 \;\; f_\alpha(x) = y$$

Dabei ist α ein Index auf dem Funktionenraum.

13.3 VC-Dimension

Die VC-Dimension einer Funktion f_α ist definiert als die Menge an Punkten, die diese Funktion in jeder beliebigen Lage auf dem Eingangsraum separieren kann.
Beispiele:

- Eine linear separierende Ebene hat die VC-Dimension von 2. Man kann 3 Punkte auf einer Geraden anordnen. Dann ist es unmöglich eine Gerade zu finden, die den mittleren Punkt in die eine Klasse und die beiden äußeren Punkte in andere Klassen einteilt. Mache nehmen diesen Fall aus, da er ein entarteter Fall ist und dies, abgesehen von Drehungen, auch nur ein Fall ist. Die VC-Dimension ist dann 3.
- Eine Tabelle, die jedem Beispiel einen Wert zuordnet, hat die Anzahl der Zeilen als VC-Dimension.
- Polynome beliebigen Gerades haben die VC-Dimension von ∞.

13.4 Risiko

Wir ziehen aus einer Menge von Mustern l Muster, die einer Zufallsverteilung $p(x, y)$ gehorchen. Wir können nun ein Risiko formulieren, dass von einer Funktion f_α diese Mustermenge nicht spariert werden kann:

$$R(\alpha) = \int \frac{1}{2} \underbrace{|f_\alpha(x) - y|}_{\substack{> 0 \\ \text{wenn falscher} \\ \text{Output}}} dP(x, y)$$

Wir können dieses Integral leider nicht berechnen. Aber wir können das impirische Risiko für l Muster berechnen

$$E_\alpha = \frac{1}{l} \sum_{i=1}^{l} \frac{1}{2} |f_\alpha(x_i) - y_i|$$

Jedoch repräsentiert E_α nicht immer $R(\alpha)$. Es kann sein, dass wir unsere Funktion f so schlecht gewählt haben, dass gerade die gewählten Muster auf sie passen und keinen Fehler haben (das ist zum Beispiel bei einer Tabelle der Fall) und das emprische Risiko 0 ist, aber das reale Risiko sehr hoch. Deshalb können wir das reale Risiko vom empirischen Risiko nur mit einem Vertrauensterm ableiten:

$$\underbrace{R(\alpha)}_{\text{Risiko}} \leq \underbrace{E_\alpha}_{\text{Lernfehler}} + \underbrace{\Theta(h,l,\eta)}_{\text{Vertrauensterm}}$$

Der Vertrauensterm ist

$$\Theta(h,l,\eta) = \sqrt{\frac{h \cdot (\log \frac{2l}{h} + 1) - \log(\frac{\eta}{4})}{l}}$$

- Die Gleichung für das empirische Risiko gilt mit einer Wahrscheinlichkeit von $1 - \eta$.

  Wenn wir mit einer hohen Wahrscheinlichkeit sagen wollen, dass die Risikogleichung richtig ist, dann erhalten wir einen hohen Wert für den Vertrauesterm z.B.

  $$\text{Risk} \leq 1 + 100000.$$

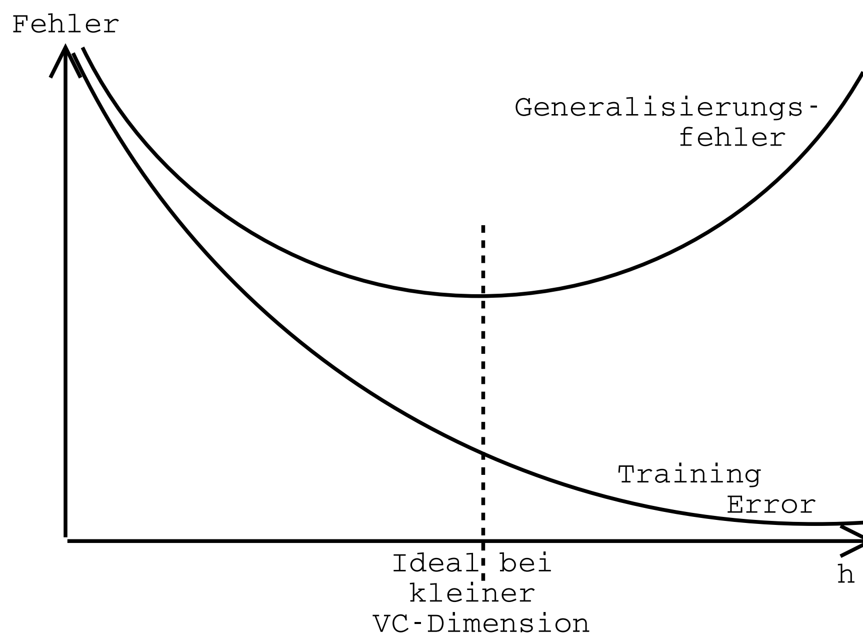
  Damit können wir allerdings nichts anfangen. Wenn wir auf der anderen Seite mit einer kleinen Wahrscheinlichkeit sagen wollen, dass die Gleichung auch wirklich richtig ist, wird der Vertrauensterm klein. Das nützt aber gar nichts, weil die Gleichung dann nichts mehr aussagt.

- l ist die Anzahl der Muster. Um den Vertrauensterm zu verkleinern können wir die Anzahl der Muster groß machen.

- h ist die VC-Dimension. Wir müssen die VC-Dimension klein machen, um den Vertrauensterm klein zu halten.

13.5 Gute Generalisierung

Eine gute Generalisierung ist erwünscht. Große VC-Dimensionen generalisieren schlecht:



Die VC-Dimension ist nur ein Konzept. Hier sehen wir, dass die VC-Dimension sehr gut funktioniert. Andere Beschreibungsmöglichkeiten wären

$$\underbrace{H^{\nearrow}(l)}_{\text{VC-Entropie}} \leq \underbrace{H^{\nearrow}_{Ann}}_{\text{Annealed VC-Entropie}} \leq \underbrace{G^{\nearrow}(\lambda)}_{\text{Grow function}} \leq \underbrace{h \cdot \left(\log \frac{2l}{h} + 1\right)}_{\text{VC-Dimension}}$$

13.6 Linear separierbare Probleme

13.6.1 Linear separierende Ebene mit Korridor

Wir können einen Vektorraum durch eine Hyperebene eine Dimension tiefer in zwei Teile teilen. Linear sparierbare Probleme lassen sich durch eine Hyperebene in zwei Klassen teilen. Eine solche Hyperebene läßt sich definieren durch

$$\{\vec{z}\,|(\vec{w} \cdot \vec{z}) + b = 0\}$$

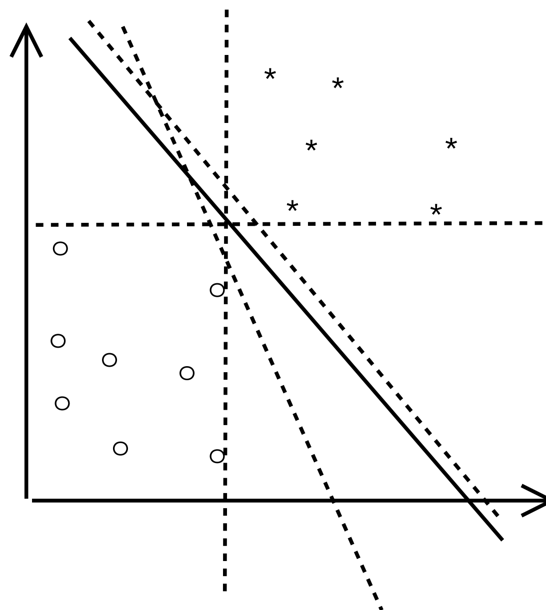
Die eine Halbebene ist dann definiert durch

$$\{\vec{z}\,|(\vec{w} \cdot \vec{z}) + b > 0\}$$

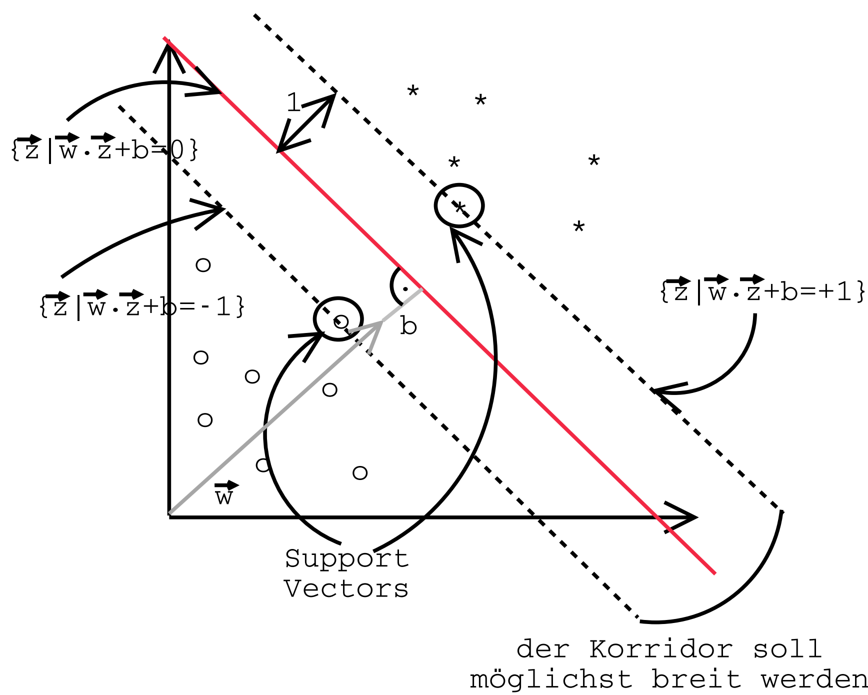
die andere durch

$$\{\vec{z}\,|(\vec{w} \cdot \vec{z}) + b < 0\}$$

Für viele separierbare Probleme können wir eine Hyperebene auf verschiedene Art und Weise erstellen



Wir sehen jedoch, dass viele Ebenen nicht so klug separieren. Wenn wir gerade an einen Rand die Ebene legen, dann können leicht Muster auftauchen, die die Ebene überschreiten. Wir wollen deshalb eine Ebene mit einem Korridor drumherum erstellen



Wir können $\vec{w}$ und b so einstellen, dass alle Punkte außerhalb des Korridors liegen und die Support Vectors genau auf der Grenze:

$$\min_{i=1...l} |(\vec{w}\vec{x}_i) + b| = 1$$

Seien $\vec{z_1}$ und $\vec{z_2}$ die Support Vectors, dann gilt

$$\begin{array}{c|l} & (\vec{w}\vec{z_1}) + b = +1 \\ - & (\vec{w}\vec{z_2}) + b = -1 \\ \hline & \vec{w}(\vec{z_1} - \vec{z_2}) = 2 \end{array}$$

Wir normieren

$$\frac{\vec{w}}{||\vec{w}||}(\vec{z_1} - \vec{z_2}) = \frac{2}{||\vec{w}||}$$

und erhalten ein passendes $\vec{w}$

$$\frac{\vec{w}}{||\vec{w}||}(\vec{z_1} - \vec{z_2}) - \frac{2}{||\vec{w}||} = 0$$

13.6.2 Decision Function

Unsere Entscheidungsfunktion ist nun

$$f_{\vec{w},b} = \underbrace{sgn}_{\text{Vorzeichen}} ((\vec{w} \cdot \vec{x}_i) + b) = y_i$$

13.6.3 La Grange

Wir versuchen zwei Ziele zu erfüllen

1. gute Generalisierung (**Extremum**)

$$\frac{1}{2}||\vec{w}||^2 \text{ minimieren}$$

2. alle Beispiele richtig klassifizieren (**Nebenbedingung**)

$$y_i((\vec{w} \cdot \vec{x}_i) + b) \geq 1$$

Dies können wir mit dem La-Grange-Ansatz. Die La-Grange-Funktion L hat die Form

$$L(\vec{w}, b, \vec{\alpha}) = \underbrace{\frac{1}{2}||w||^2}_{\text{Extremwert}} \underbrace{- \sum_{i=1}^{l} \alpha_i (y_i((\vec{w} \cdot \vec{x}_i) + b) - 1)}_{\text{Nebenbedingung}}$$

Nach Einsetzen und Auflösen erhalten wir die sogenannte Duale Form

$$D(\vec{\alpha}) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j)$$

Die Decision Function wäre dann

$$f(\vec{z}) = sgn \left(\sum_{i=1}^{l} \alpha_i y_i (\vec{z} \cdot \vec{x}_i) + b \right)$$

13.7 Feature Space

Was aber tun, wenn das Problem nicht linear separierbar ist? **Wir betten das Problem in einen hochdimensionalen Feature-Space ein.** Dies tun wir mit einer Funktion ϕ:

$$\phi : G \to F$$
$$\phi : \mathbb{R}^N \to \mathbb{R}^M$$

Die Dimension M des Feature Space ist sehr hoch:

$$M = \frac{(N + d - 1)!}{d!(N - 1)!}$$

wobei d die Ordnung und N die Dimension des Eingangsraumes. Bei $d = 5$ und $N = 256$ beispielsweise $M \approx 10^{10}$.

Problem ist nun, dass wir die großen Skalarprodukte, die nun entstehen gar nicht berechnen können:

$$D(\vec{\alpha}) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j (\underbrace{\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)}_{\text{Skalarprodukt}})$$

$$f(\vec{z}) = sgn \left(\sum_{i=1}^{l} \alpha_i y_i (\underbrace{\phi(\vec{z}) \cdot \phi(\vec{x}_i)}_{\text{Skalarprodukt}}) + b \right)$$

13.8 Kernel functions

Aber wir können sogenannte Kernel-Functions benutzen. Diese müssen die Mercer-Bedingung erfüllen. Mit der Kernelfunction können wir die Duale Form und die Decision Function dann doch berechnen:
Maximiere

$$D(\vec{\alpha}) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j \, k(\vec{x}_i, \vec{x}_j)$$

Decision Function ist

$$f(\vec{z}) = sgn \left(\sum_{i=1}^{l} \alpha_i y_i \, k(\vec{z}, \vec{x}_i) + b \right)$$

Beispiele für Kernel Functions:

- Polynom Klassifikation

$$k(\vec{z}, \vec{x}_i) = (\vec{z} \cdot \vec{x}_i)^d$$

- RBF

$$k(\vec{z}, \vec{x}_i) = \exp(-||\vec{z} - \vec{x}_i||^2 / c)$$

- MLP

$$k(\vec{z}, \vec{x}_i) = \tanh(\vec{g}(\vec{z} \cdot \vec{x}_i) + s)$$