Seminar Neuronale Netze

Kohonenalgorithmen

Marco Vieth, 16.02.1995
ali@uni-paderborn.de

Inhaltsverzeichnis

1 Überblick

Woher weiß das Sehzentrum im Gehirn, aus welchen Bereichen der Netzhaut die Bildinformationen stammen? Gibt es im Gehirn ein Abbild der Körperoberfläche, um aktive Sinneszellen zu lokalisieren? Wie wird das in einzelne Frequenzen zerlegte Geräusch vom Ohr zum Gehirn übertragen? Diese Fragen kann ich zwar auch nicht beantworten, sie führen aber mitten in den Bereich der sogenannten **„feature maps“** und der **topologieerhaltenden Abbildungen**. Ich werde den **Kohonenalgorithmus** vorstellen und zum Schluß noch auf eine neue Lösungsmöglichkeit für ein wohlbekanntes Problem eingehen, dem **Problem des Handlungsreisenden**. Doch zuvor einige Vorbemerkungen über unüberwachtes Lernen.

2 Unüberwachtes Lernen (unsupervised learning)

Was ist überhaupt ein Lernverfahren bei neuronalen Netzen? Das „Wissen“ eines neuronalen Netzes ist ja bekanntlich in den Gewichten zwischen den Neuronen gespeichert. Am Anfang haben wir ein „dummes“ Netz, bei dem die Gewichte irgendwie initialisiert wurden, so daß sie bestimmt nicht optimal für die vorgesehene Anwendung sind.

Def. 2.1 *Ein Lernverfahren ist eine Strategie, wie man optimale Gewichte für ein neuronales Netzwerk bestimmen kann.*

Neben dem überwachten Lernen, bei dem neben den Eingaben auch die Ausgaben vorgegeben werden und ein „Lehrer“ kontrolliert, ob das Netz auch auf diese Ausgaben kommt, gibt es auch das unüberwachte Lernen.

Def. 2.2 *Ein unüberwacht lernendes Netz ist ein selbstorganisierendes Netz, bei dem der Lernerfolg nicht überwacht wird.*

Selbstorganisation meint: Das Netz erkennt selbständig Muster, Besonderheiten, und Regelmäßigkeiten in den Eingaben und ordnet entsprechende Ausgabeneuronen zu. Das unüberwachte Lernen funktioniert meistens in zwei Phasen:

1. Lernphase: Das Netz bekommt die ihm unbekannten Eingaben und paßt die Gewichte automatisch durch Selbstorganisation an.
2. Anwendungsphase: Ohne weiteres Lernen kann das „trainierte“ Netz z.B. neue Eingaben kategorisieren.

Beispiel 2.3 *(Schrifterkennung mittels Computer): Der Computer wird zuerst mit der Handschrift einer Person „trainiert“ und versucht dann, neue (undeutlich geschriebene) Zeichen zu erkennen.*

Bei diesem Lerntyp können die Ausgabeneuronen z.B. auch einen Binärcode ausgeben, indem sie eine 01-Folge mit mehreren 1en ausgeben, um die Eingabe zu charakterisieren. Bei dem folgenden Typ ist das nicht mehr möglich.

2.1 Konkurrierendes Lernen (competitive learning)

Def. 2.4 *Ein konkurrierend lernendes Netz ist ein unüberwachtes Netz, bei dem nur ein Neuron feuern darf und den Gewichtsupdate bekommt. Dieses Neuron wird Gewinner genannt.*

Das konkurrierende Lernen ist also ein Spezialfall des unüberwachten Lernens. Die Ausgabeneuronen liegen im Wettstreit darum, wer feuern darf. Nur eines wird sich durchsetzen und bekommt als Belohnung den Gewichtsupdate. Es gibt also einen **„winner-take-all“-**Charakter (es kann nur einen geben).
Was kann man damit machen? Solche Netze eignen sich besonders gut zum Kategorisieren von Eingaben, wobei die Klasseneinteilung mittels Selbstorganisation gefunden wird. Bei „ähnlichen“ Eingaben feuert dasselbe Ausgabeneuron. Eine wichtige Anwendung ist die sogenannte **„vector quantization“** (clustering), bei der Eingabevektoren durch die Nummer des Ausgabeneurons repräsentiert werden. Doch dazu gleich mehr. Vorher wollen wir das Modell eines konkurrierenden Netzes konkretisieren.

2.1.1 Aufbau eines konkurrierenden Netzes

- Eingabeschicht vollständig verbunden mit Ausgabeschicht; trotzdem einschichtig
- feed-forward
- reelwertiger Eingabevektor $\vec{x} = x_1 \dots x_J$, diskreter Ausgabevektor $\vec{y} = y_1 \dots y_N$, Gewichte des i-ten (Ausgabe-)Neurons: $\vec{w_i} = w_{i1} \dots w_{iN}$

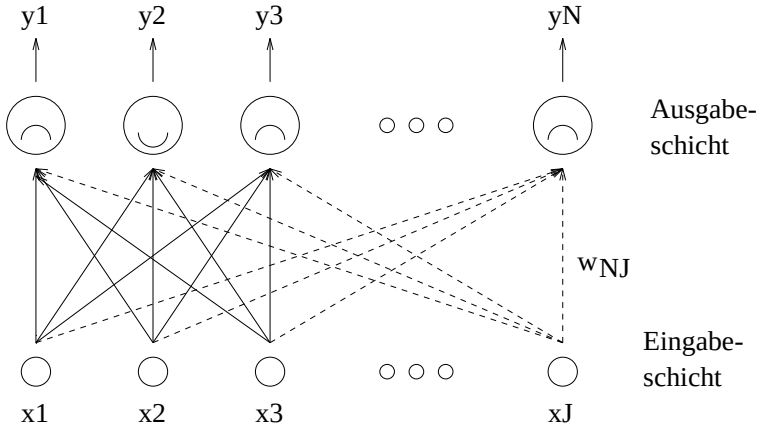

Abbildung 1: Modell eines konkurrierenden Netzes

Abbildung 1 zeigt den typischen Aufbau eines konkurrierenden Netzes. Wie üblich, läuft der Informationsfluß von unten nach oben. Die Eingabeneuronen sind so bedeutungslos, daß sie bei der Zählung der Schichten nicht mitgezählt werden.

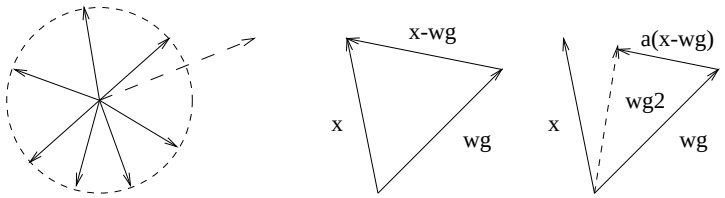

Abbildung 2: a) normierte Gewichte b,c) Gewichtsupdate

2.1.2 Ablauf des konkurrierenden Lernens

1. Gewinner bestimmen: Das Neuron g mit der größten Aktivierung $y_g = \vec{w_g} \cdot \vec{x}$
2. Gewichtsupdate nur für den Gewinner

Gewinner bestimmen: Präsentation des Eingabevektors $\vec{x} = x_1 \dots x_J$. Neuron g ist der Gewinner, falls $\forall i : \vec{w_g} \cdot \vec{x} \geq \vec{w_i} \cdot \vec{x}$. Falls die Gewichte normiert sind (d.h. $\forall i : \|\vec{w_i}\| = 1$), kann man diesen Schritt auch so ausdrücken, daß der Gewichtsvektor $\vec{w_g}$ des Gewinners der Eingabe $\vec{x}$ am nächsten kommt:

$$\forall i : \|\vec{w_g} - \vec{x}\| \leq \|\vec{w_i} - \vec{x}\| \tag{1}$$

Normiert heißt, die Gewichtsvektoren befinden sich innerhalb des Einheitskreises (bzw. der -kugel), vgl. Abbildung 2a.
Gewichtsupdate: Im Gegensatz zum Hopfield-Netz erfolgt bei diesem Typ der Gewichtsupdate nur für den Gewinner. Ziel ist es, den Gewichtsvektor $\vec{w_g}$ näher an den Eingabevektor

$\vec{x}$ heranzurücken, damit der Gewinner in Zukunft bei ähnlichen Eingaben mit höherer Wahrscheinlichkeit gewinnt. Das leistet die Standard-Lernregel des konkurrierenden Lernens:

$$\Delta\vec{w}_g \quad = \quad \alpha(\vec{x} - \vec{w}_g) \tag{2}$$

Wie funktioniert diese Lernregel? Anwenden auf $\vec{w}_g$ ergibt: $\vec{w}_g + \Delta\vec{w}_g = \vec{w}_g + \alpha(\vec{x} - \vec{w}_g) = \alpha\vec{x} + (1-\alpha)\vec{w}_g$ mit $0 \leq \alpha \leq 1$. Der Update besteht also aus dem gewichteten Mittel zwischen (oder der konvexen Linearkombination von) dem alten Wert $\vec{w}_g$ und dem Eingabevektor $\vec{x}$. Abbildung 2bc veranschaulicht, daß der Endpunkt des neuen $\vec{w}_g$ auf der Strecke zwischen diesen beiden Vektoren liegt.

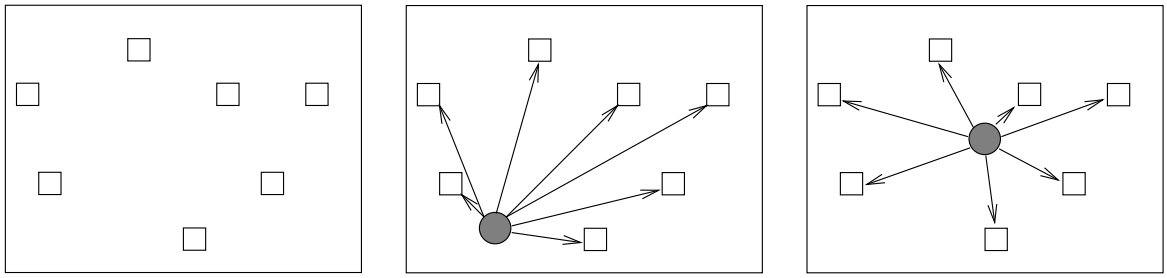

Abbildung 3: a) Eingaben b) Energie nicht minimal c) Energie minimal

2.1.3 Mathematische Analyse mittels Energiefunktion

Wann hat das Netz genug gelernt? Dazu muß es einen Gleichgewichtszustand erreicht haben, bei dem die folgende Energiefunktion minimal ist:

$$E = \frac{1}{2}\sum_{i,j,k} M_i(k)(x_j(k) - w_{ij})^2 = \frac{1}{2}\sum_{j,k}(x_j(k) - w_{gj})^2 \tag{3}$$

Hierbei ist $\vec{x}(k)$ die Eingabe zum diskreten Zeitpunkt k. $M_i(k) = \{1 \text{ für } i = g(k),\ 0 \text{ sonst}\}$ ist die „cluster membership matrix“, die angibt, ob bei Eingabe $\vec{x}(k)$ Neuron i der Gewinner ist. Es werden alle $\|\vec{x} - \vec{w}_g\|$ summiert, also die Vektorlängen zwischen dem Gewinner und allen möglichen (gleichverteilten) Eingaben. Zur Erläuterung betrachte Abbildung 3: Die kleinen Quadrate in 3a sind einige Eingaben. In 3b hat der Gewinner eine schlechte Position eingenommen, bei der die Vektorlängen noch recht groß sind. In 3c hat er die Mitte des Rechtecks eingenommen, eine für Gleichverteilung optimale Position.
Wie kann man die Energiefunktion interpretieren? Man kann sich eine Gebirgslandschaft vorstellen, auf der eine Kugel herumrollt. Diese Kugel wird bestrebt sein, in einem tiefen Tal liegenzubleiben. Wie schnell wird sie im Mittel ein Tal erreichen? Dazu bilden wir die partielle Ableitung der Energiefunktion nach w_{ij}:

$$\frac{\partial E}{\partial w_{ij}} = -\sum_k M_i(k)(x_j(k) - w_{ij})$$

Dieses ist die Standardregel (2), summiert über alle Eingaben $\vec{x}(k)$, für die i der Gewinner ist. Die mittlere Änderungsrate der Energiefunktion setzen wir als (EW=Erwartungswert):

$$EW(\Delta w_{ij}) = -\alpha\frac{\partial E}{\partial w_{ij}}$$

Dies ist die mittlere Geschwindigkeit, mit der die Kugel (oder der Skifahrer) den Berg hinunterrollt. Dabei wird über alle möglichen Richtungen gemittelt (hier eben Gleichverteilung). Im Durchschnitt verringert die Standardregel die Energiefunktion (bei kleinem α), bis ein lokales Minimum erreicht wird. Es gibt aber keine Garantie dafür, daß ein globales Minimum (das tiefste Tal) erreicht wird.

2.1.4 Wichtige Anwendung: „vector quantization“

„Vector quantization“ ist die wichtigste Anwendung des konkurrierenden Lernens. Dabei werden die Eingabevektoren $\vec{x}(k)$ in Klassen kategorisiert und durch den Index der Klasse repräsentiert. Dieses Verfahren eignet sich gut zur Datenkompression, denn es braucht nicht mehr der „große“ Vektor selber übertragen oder gespeichert werden sondern nur dessen Index. In einem Kodebuch kann man dann den Repräsentations-Vektor nachschlagen. Beim unserem Netz gibt der Gewinner die Klasse an.

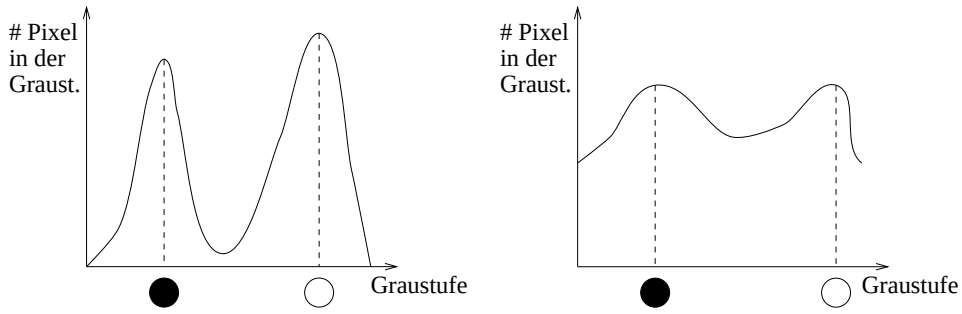

Abbildung 4: Graustufenspektrum: a) Foto 1, b) Foto 2

Beispiel 2.5 *Bildumwandlung: Wir haben ein Bild in Graustufen vorliegen, und wollen es nach schwarzweiß umwandeln. Dazu legen wir einen Schwellwert fest und färben jedes Pixel schwarz, wenn die Graustufe darunterliegt, andernfalls weiß. Problematisch ist die Wahl eines vernünftigen Schwellwertes, er muß für jedes Bild anders gewählt werden. Mit „vector quantization“: Wie in Abbildung 4 suchen wir zwei Maxima im „Graustufenspektrum“ des Bildes und nennen das eine schwarz, das andere weiß. Eine Graustufe wird dann durch das am nächsten liegende Maxima repräsentiert.*

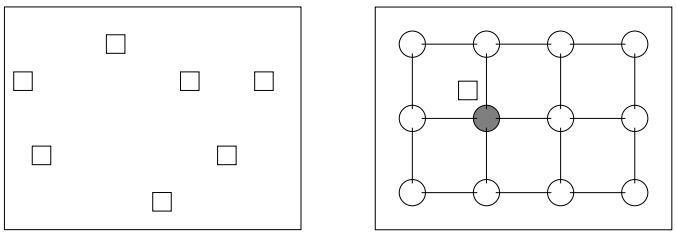

Abbildung 5: Kohonens Map Beispiel: a) gleichverteilte Eingaben, b) nächstes Neuron feuert

3 Kohonens Algorithmus

3.1 Topologieerhaltende Abbildungen und „feature maps“

Betrachten wir zuerst das Beispiel in Abbildung 5. In 5a haben wir wieder realwertige, gleichverteilte Eingaben. In 5b sind die Gewichte eines Netzes dargestellt, das wir jetzt kennenlernen werden. Bei einer Eingabe feuert das am nächsten gelegene Neuron.

Def. 3.1 *Eine topologieerhaltende Abbildung f ist eine Abbildung, bei der „Nachbarschaftsverhältnisse“ erhalten bleiben. Seien A, B zwei Mengen mit Distanzfunktionen d_A bzw. d_B. Für $f : A \to B$ gilt dann: $\forall a, b, c \epsilon A : d_A(a, b) \leq d_A(a, c) \Rightarrow d_B(f(a), f(b)) \leq d_B(f(a), f(c))$.*

Beispiel 3.2 *Vergleich: Zwei Nachbarn sind sich nicht „ganz grün“. Als sie wegen Hochwasser evakuiert werden, hoffen sie, in den neuen Unterkünften endlich einmal Ruhe zu haben. Doch die Evakuierungs-Behörde hat zur Umquartierung eine topologieerhaltende Abbildung benutzt, so sind sie natürlich wieder Nachbarn, jetzt sogar Türnachbarn.*

Bemerkung 3.3 Zurück zur Mathematik: Es gibt z.B. keine topologieerhaltende Abbildung von einem Kreis auf eine Gerade, irgendwo muß man den Kreis aufschneiden und Nachbarn werden getrennt.

Satz 3.4 *Jede stetige Abbildung ist topologieerhaltend aber nicht umgekehrt.*

Beweis: Übungsaufgabe.

Def. 3.5 *Eine „feature map“ ist ein Netz mit einer topologieerhaltenden Abbildung.*

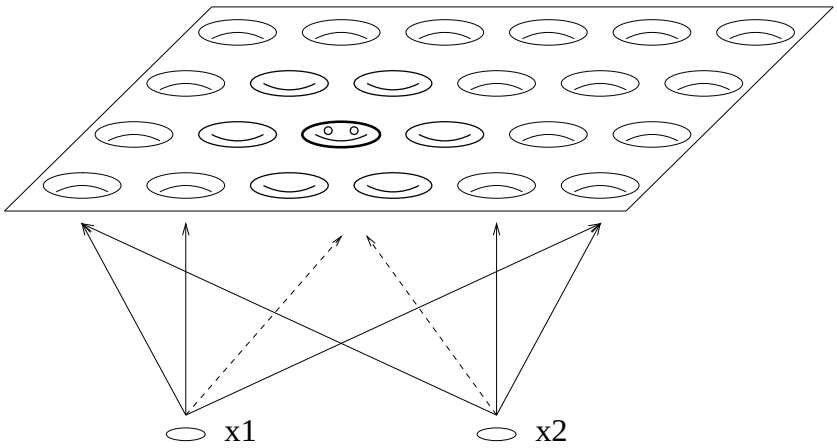

Abbildung 6: Aufbau einer „feature map“

Der Aufbau einer „feature map“ ist fast identisch mit dem eines normalen konkurrierenden Netzes. Nur: Wie in Abbildung 6 ordnen wir die Neuronen aber in einem Array an, so daß die Position des i-ten Neurons z.B. eindeutig durch $\vec{p_i} = (p_{i1}, p_{i2})$ (Zeile, Spalte) festgelegt ist. Auf dieser **geometrischen Anordnung** nehmen wir als Distanzfunktion die euklidische Distanz. Für eine „feature map“ gilt folgendes: Seien $\vec{x}(1)$, $\vec{x}(2)$ zwei Eingabevektoren aus dem Eingaberaum, $\vec{p_g}(1)$, $\vec{p_g}(2)$ die Positionen der gewinnenden Neuronen aus dem Positionsraum. Dann soll gelten: $\vec{x}(1)$, $\vec{x}(2)$ „ähnlicher“ $\Rightarrow$ $\vec{p_g}(1)$, $\vec{p_g}(2)$ „näher zusammen“.

3.2 Kohonens Algorithmus zum Erzeugen einer „feature map“

Der Algorithmus von Teuvo Kohonen verwandelt ein Netz in eine „feature map“. Es gibt viele Variationen, deshalb ist auch von Kohonenalgorithmen die Rede. Der Ablauf ist nahezu identisch mit dem des konkurrierenden Lernens:

1. Gewinner bestimmen: Wieder Neuron g mit $\forall i : \|\vec{w_g} - \vec{x}\| \leq \|\vec{w_i} - \vec{x}\|$.
2. Gewichtsupdate: Für den Gewinner **und seine Nachbarschaft!**

Gewichtsupdate: Die Nachbarschaft bilden alle Neuronen in einem bestimmten Radius um den Gewinner, wie z.B. in Abbildung 6 zu sehen war. Außerdem sollte die Größe des Updates kontinuierlich von der Nähe zum Gewinner abhängig sein. Sei $\vec{p_i}$ wieder die Position von Neuron i. Als Lernregel nehmen wir die Standard-Lernregel des konkurrierenden Lernens (2), erweitert um eine Nachbarschaftsfunktion $N(i, g)$, die 1 ist für $i = g$ und abfällt, wenn der Abstand $\|\vec{p_i} - \vec{p_g}\|$ größer wird.

$$\Delta\vec{w_i} \quad = \quad \alpha N(i, g)(\vec{x} - \vec{w_i}) \tag{4}$$

Die Gewichte des Gewinners und seiner nahen Nachbarn werden verändert, die Gewichte weiter entfernt liegender Nachbarn ändern sich kaum. Für $i = g$ ergibt sich wieder die Standard-Lernregel (2): $\Delta w_{gj} = \alpha(x_j - w_{gj})$. Eine gute Wahl für N ist $N(i, g) = \exp\left(-\frac{\|\vec{p_i} - \vec{p_g}\|^2}{2\sigma^2}\right)$

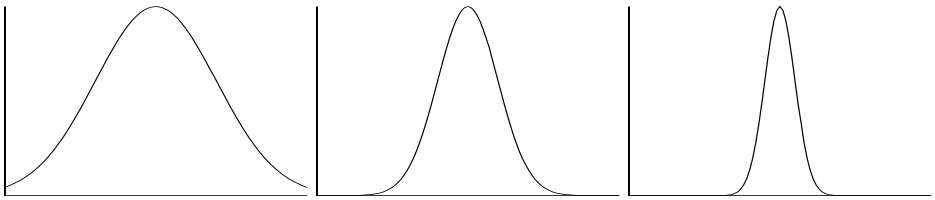

Abbildung 7: Nachbarschaftsfunktion bei σ = 4,2,1

mit σ als „Einflußparameter“ auf die Nachbarschaft. Abbildung 7 zeigt die Auswirkungen verschiedener σ in einem Bereich von $-10 \ldots 10$ für die Differenz $\|\vec{p_i} - \vec{p_g}\|$. Man sieht sofort: Je kleiner σ, desto kleiner die Nachbarschaft.

Der Algorithmus ist 2-fach zeitabhängig: N und α sollten zeitabhängig sein, also vom diskreten Zeitpunkt k abhängen ($N^k(i, g)$ bzw. $\sigma(k)$ und $\alpha(k)$). Werden diese Parameter am Anfang groß gewählt und im Laufe des Lernens immer kleiner, dann kann sich die „feature map“ zuerst schnell aufbauen und später verfeinern. Die ganze Kunst liegt darin, diese Parameter vernünftig zu wählen.

Wie man sich das vorstellen kann: Elastisches Netzes im Eingaberaum, welches den Eingaben so gut wie möglich entsprechen will. Es hat die Topologie des Ausgabearrays (z.B. 2-dim) und die Punkte des Netzes haben die Gewichte als Koordinaten.

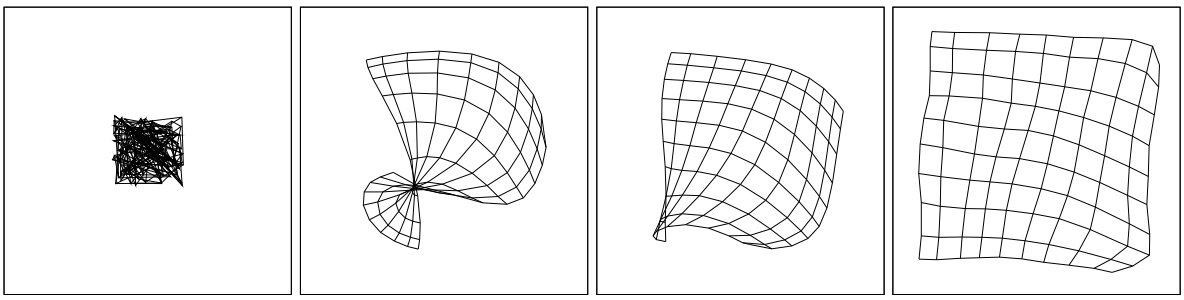

Abbildung 8: Die Entstehung einer „feature map“

3.3 Veranschaulichung

Ich wollte wissen, ob die schönen Bildchen aus dem Buch wirklich völlig automatisch entstehen, wenn man den Algorithmus in dieser Form verwendet. Und es klappt wirklich, wie die folgenden Abbildungen zeigen (sie sind alle mit Kohonens Algorithmus entstanden) ...
Abbildung 8 zeigt die Entwicklung einer „feature map“. Zugrunde liegt ein 10x10-Netz mit zwei Eingaben x_1, x_2 (also 2-dim $\rightarrow$ 2-dim). Wieder sind die Eingaben zufällig, gleichverteilt. Dargestellt sind die Gewichte (w_{i1}, w_{i2}), mit Nachbargewichten durch Linien verbunden. Somit kann man gut verfolgen, wie sich die Gewichte in der vorgegebenen topologischen Struktur der Neuronen anordnen. In 8a wurden die Gewichte zufällig in einem Quadrat der Seitenlänge 0.5 um den Punkt $(0.5, 0.5)$ initialisiert. 8b zeigt das Netz nach etwa etwa 100 Lernschritten (auf einem 486/66 in weniger als einer Sekunde berechnet). usw.

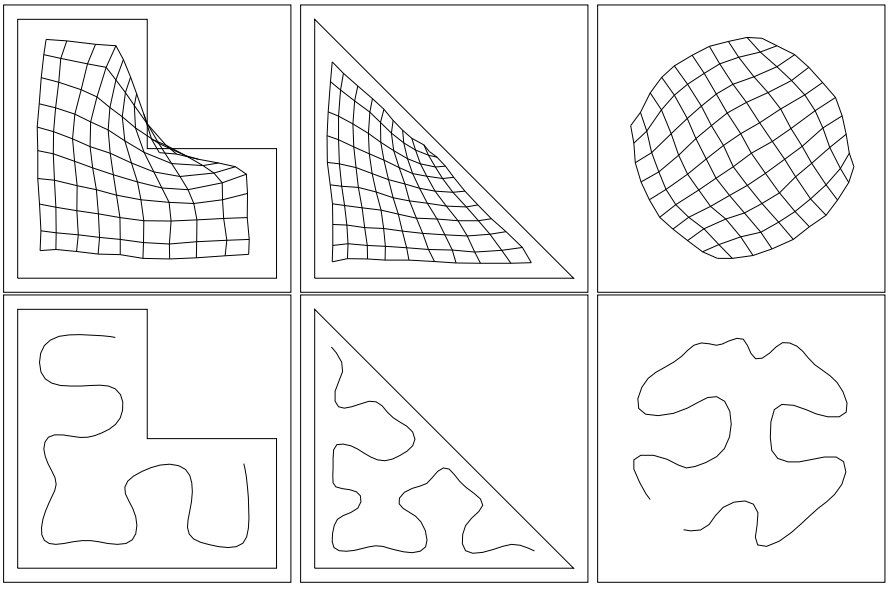

Abbildung 9: „feature maps“ bei anderen Eingaberegionen; oben 2-dim, unten 1-dim

Andere Eingaberegionen: Abbildung 9 zeigt fertige „feature maps“ bei anderen Eingaberegionen. Die gleichverteilten Eingaben stammen also aus den Bereichen L, Dreieck und Kreis. In der oberen Reihe ist das Ausgabearray wieder 2-dim. Darunter wurden die Neuronen in einem 1-dim Array angeordnet. In diesem Fall findet eine **Dimensionsreduktion** statt. Auch hier versucht die „feature map“, den Eingaben möglichst gut zu entsprechen. Wird die Nachbarschaft am Anfang groß genug gewählt, dann verschwinden alle Überschneidungen und es entsteht eine Peanokurve.

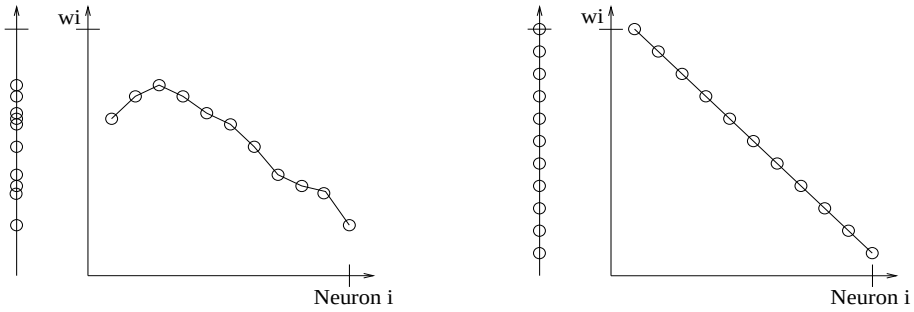

Abbildung 10: „feature map“ 1-dim nach 1-dim, mit Projektion

Abbildungen 1-dim → 1-dim sind natürlich auch möglich. Das Netz bekommt die (gleichverteilte) Eingabe x und die Neuronen sind in einem 1-dim Array angeordnet. In Abbildung 10 wurden die Gewichte über den Neuronen aufgetragen. In 10a ist noch ein Knick zu sehen. Die Projektion der Gewichte zeigt, daß diese noch nicht gleichverteilt sind. In 10b sind die Gewichte dann gleichverteilt. Genausogut hätte sich das Netz für eine steigende Gewichtsfolge entscheiden können.

3.4 Mathematische Analyse

Die meisten Untersuchungen von Kohonens „feature map“ konnten nur für den 1-dim Fall durchgeführt werden. Die Energiefunktion ist fast dieselbe wie beim einfachen konkurrierenden Netz (3). Nur die Nachbarschaft ist jetzt größer, so daß nicht nur die Vektoren zwischen dem Gewinner und allen Eingaben summiert werden, sondern auch diejenigen zwischen den Nachbarn des Gewinners und allen Eingaben:

$$E = \frac{1}{2}\sum_{i,j,k,l} M_l(k)N(i,l)(x_j(k)-w_{ij})^2 = \frac{1}{2}\sum_{i,j,k} N(i,g(k))(x_j(k)-w_{ij})^2 \qquad (5)$$

Bezeichnungen wie in Abschnitt 2.1.3. Partielle Ableitung der Energiefunktion ergibt wieder die mittlere Änderungsrate der Energiefunktion:

$$EW(\Delta w_{ij}) = -\alpha\frac{\partial E}{\partial w_{ij}} = \alpha\sum_{k,l} M_i(k)N(i,l)(x_j(k)-w_{ij}) = \alpha\sum_{k} N(i,g(k))(x_j(k)-w_{ij})$$

Dieses ist Kohonens Lernregel (4), summiert über alle Eingaben $\vec{x}(k)$. Wieder verringert die Lernregel die Kostenfunktion (bei kleinem α), bis ein lokales Minimum erreicht wird.

Wann befinden sich die Neuronen in einer stabilen Konfiguration, d.h. wann haben ihre Gewichte optimale Positionen erreicht? Um solche Gleichgewichtszustände zu finden, setzen wir $EW(\Delta w_{ij}) = 0$, also (ohne α):

$$0 = \sum_k N(i, g(k))(x_j(k) - w_{ij})$$

Da sich diese Gleichung nicht einfach lösen läßt, approximieren wir sie (im 1-dim Fall) durch:

$$0 = \int N(\vec{p} - \vec{p}_g(x))(x - w(\vec{p}))P(x)dx$$

Hierbei ist $P(x)$ die Wahrscheinlichkeitsdichte der Eingabe x. Die Summe über k wurde durch $\int P(x)dx$ ersetzt, so daß jetzt beliebinge Eingabeverteilungen möglich sind. Außerdem wurde der Index i durch den Positionsvektor $\vec{p}$ ersetzt. Die Lösung dieser Gleichung zeigt, daß die Gewichte am Ende des Lernens wirklich steigen oder fallen müssen (wie in Abbildung 10). Außerdem folgt:

Satz 3.6 *(„unfaire" Dichte) Bei Kohonens Algorithmus kann man nur eine Neuronendichte von $P(x)^{\frac{2}{3}}$ erwarten (anstatt der idealen Dichte $P(x)$).*

Kohonens Algorithmus tendiert also dazu, in höher wahrscheinliche Bereiche wenig Neuronen und in weniger wahrscheinliche Bereiche viele Neuronen „hinzuschicken".

3.4.1 Von Schleifen und Knicken

Oft treten während des Lernens Schleifen oder Knicke auf. Im 1-dim Fall natürlich nur Knicke, wie z.B. in Abbildung 10. Was ist überhaupt ein Knick?

Def. 3.7 *Ein Knick ist eine Grenze zwischen monotonen Regionen.*

Wie lange kann es dauern, bis solche Knicke verschwinden? Dazu müssen wir monotone Regionen untersuchen.

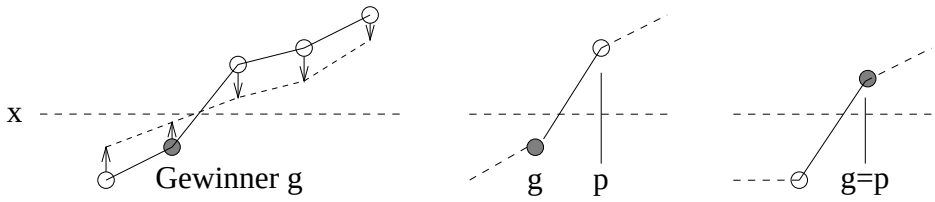

Abbildung 11: Eine monotone Folge bleibt nach Update erhalten

Lemma 3.8 *(Monotone Regionen) Eine monoton steigende (fallende) Sequenz von Gewichten bleibt auch nach jedem Update monoton steigend (fallend).*

Beweis: Leider kann dies nur für den Fall gezeigt werden, daß keine Knicke vorhanden sind, also die gesamte Gewichtsfolge monoton ist. O.B.d.A. sei die Folge (w_n) wie in Abbildung 11a monoton steigend, d.h. $\forall i : w_i(k-1) \geq w_{i-1}(k-1)$, oder äquivalent: $\forall i : w_i(k-1) - w_{i-1}(k-1) \geq 0$ (in diesem 1-dim Fall können wir natürlich auf die Vektorschreibweise verzichten). Z.z.:

$\forall i : w_i(k) \geq w_{i-1}(k)$, oder äquivalent: $\forall i : w_i(k) - w_{i-1}(k) \geq 0$. Voraussetzung: $N(i,g)$ ist monoton steigend für $i < g$ und monoton fallend für $i \geq g$. Betrachte die Lernregel aus (4): $w_i(k) = w_i(k-1) + \alpha N(i,g)(x - w_i(k-1))$. Diese läßt sich umschreiben:

$$\begin{aligned} w_i(k) - x &= w_i(k-1) + \alpha N(i,g)(x - w_i(k-1)) - x \\ &= (w_i(k-1) - x) - \alpha N(i,g)(w_i(k-1) - x) \\ &= (1 - \alpha N(i,g))(w_i(k-1) - x) \end{aligned}$$

Sei x eine beliebige Eingabe, i ein beliebiger Index. Sei $N(i-1,g) = N(i,g) + d$ für ein geeignetes $-1 \leq d \leq 1$. Wir betrachten die Differenz $(w_i(k) - x) - (w_{i-1}(k) - x)$:

$$\begin{aligned} w_i(k) - w_{i-1}(k) &= (1 - \alpha N(i,g))(w_i(k-1) - x) - (1 - \alpha N(i-1,g))(w_{i-1}(k-1) - x) \\ &= (1 - \alpha N(i,g))(w_i(k-1) - x) - (1 - \alpha(N(i,g) + d))(w_{i-1}(k-1) - x) \\ &= (1 - \alpha N(i,g))(w_i(k-1) - w_{i-1}(k-1)) + \alpha d(w_{i-1}(k-1) - x) \\ &\geq \alpha d(w_{i-1}(k-1) - x) =: H \end{aligned}$$

Bleibt z.z.: $H \geq 0$. Sei p der Index, ab dem $(w_{i-1}(k-1) - x) \geq 0$ ist, d.h. $(w_{i-1}(k-1) - x) \leq 0$ für $i - 1 < p$ und ≥ 0 für $i - 1 \geq p$. Da das Gewicht des Gewinners am nächsten bei x liegt, gilt $p = g$ oder $p = g + 1$ wie in Abbildung 11b und 11c. Wir nehmen an, daß $p = g$ ist und unterscheiden zwei Fälle.

1. $i \leq p$: Aus der Voraussetzung von N folgt: $N(i-1,g) \leq N(i,g)$, d.h. $d \leq 0$. Und aus der Definition von p folgt: $w_{i-1}(k-1) - x \leq 0$, da $i - 1 \leq p$. Somit ist $H \geq 0$.
2. $i \geq p - 1$: Aus der Voraussetzung von N folgt: $N(i-1,g) \geq N(i,g)$, d.h. $d \geq 0$. Und aus der Definition von p folgt: $w_{i-1}(k-1) - x \geq 0$, da $i - 1 \geq p$. Somit ist $H \geq 0$.

Somit ist die Monotonie gezeigt. □

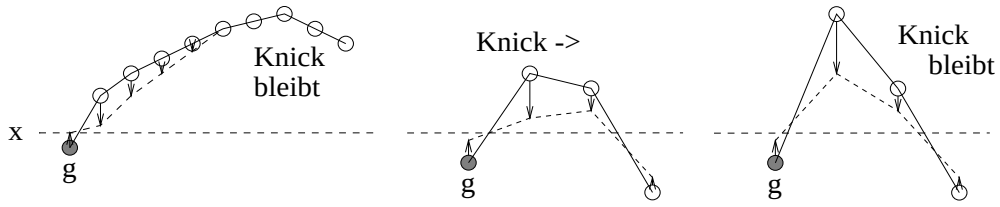

Abbildung 12: Knick bewegt sich, bewegt sich nicht, ...

Was bedeutet das für Knicke? Wenn wir einmal eine monotone Folge haben, dann bleibt sie monoton, d.h. es entstehen keine neuen Knicke. Intuitiv ist klar, daß Knicke sich aufheben können, wenn sich zwei begegnen (wie zwei entgegengesetzte Wellen, die sich überlagern). Wir betrachten nur den Fall eines einzelnen Knicks. Dieser muß zum linken oder rechten Rand wandern, um zu verschwinden. Wie lange wird das dauern? Abbildung 12a zeigt eine Situation, in der der Knick weit vom Gewinner entfernt ist. Wegen der dort kleinen Nachbarschaftsfunktion bewegt er sich nicht. In 12b und 12c ist der Gewinner nah genug. Der Knick bewegt sich nur, wenn er nicht zu stark ist. Zumindest wird er aber abgeflacht, wenn er sich nicht bewegt. Daraus schließen wir:

Lemma 3.9 *Wenn der Gewinner in der Nähe ist, bewegt sich der Knick im Mittel um einen Schritt nach links oder rechts.*

Wir setzen voraussetzen, daß die Nachbarschaftsfunktion N symmetrisch ist (d.h. $N(i,j) = N(j,i)$) und nehmen an, daß sich der Knick mit der gleichen Wahrscheinlichkeit $\frac{1}{2}$ in eine der Richtungen bewegt. Er führt einen sogenannten symmetrischen „random walk“ aus.

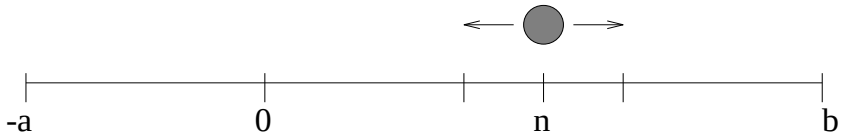

Abbildung 13: Teilchen beim „random walk“

Einschub („random walk“): Um das ein wenig zu beleuchten, betrachten wir eine Strecke mit den Endpunkten $-a$ und b, auf der ein Teilchen den „random walk“ ausführt. Sei S_n die Anzahl der Schritte, um aus Position n nach $-a$ oder b zu gelangen. $E(S_n)$ ist dann die mittlere Anzahl der Schritte (Erwartungswert). Diese Anzahl läßt sich folgendermaßen auf die Nachbarpositionen zurückführen:

$$E(S_n) = 1 + \frac{1}{2}E(S_{n+1}) + \frac{1}{2}E(S_{n-1})$$

Die allgemeine Lösung dieser Gleichung ist $E(S_n) = -n^2 + c_1 n + c_2$ mit Konstanten c_1 und c_2 (einfach durch Einsetzen überprüfen). Mit $S_{-a} = S_b = 0$ lassen sich diese Konstanten aus dem Gleichungssystem bestimmen: $c_1 = b - a$ und $c_2 = ab$. Es folgt: $E(S_n) = -n^2 + (b-a)n + ab = (n+a)(b-n)$. Aus der mittleren Position 0 benötigt das Teilchen also im Schnitt $E(S_0) = ab$ Schritte. (Siehe [BLO94].)

Zurück zu unserem Knick: Bei $N = a + b$ Neuronen benötigt der Knick ab Schritte, um zum Rand zu gelangen, was einer Größenordnung von $\Theta(N^2)$ entspricht. Wie schon angedeutet, bewegt er sich aber nur, wenn der Gewinner in der Nähe ist, was bei N Neuronen mit Wahrscheinlichkeit $\frac{1}{N}$ passiert (die Anzahl der Nachbarn des Gewinners kann als Konstante vernachlässigt werden). Somit ergibt sich:

Satz 3.10 *Bei N Neuronen braucht ein Knick ungefähr N^3 Schritte, um zum Rand zu gelangen und zu verschwinden.*

4 Anwendung: Problem des Handlungsreisenden

Neben vielen anderen Ansätzen zur Lösung des Problemes des Handlungsreisenden (TSP) läßt sich auch die „feature map“ dazu verwenden. Gegeben: J Städte durch ihre Koordinaten.

Def. 4.1 *Eine Tour ist ein Linienzug durch alle Punkte, bei der Anfang und Ende verbunden sind (Ring). Beim TSP suchen wir die kürzeste Tour.*

Eine Tour kann man als Abbildung auffassen: 2-dim $\rightarrow$ 1-dim Ring. Wir „füttern“ Kohonens

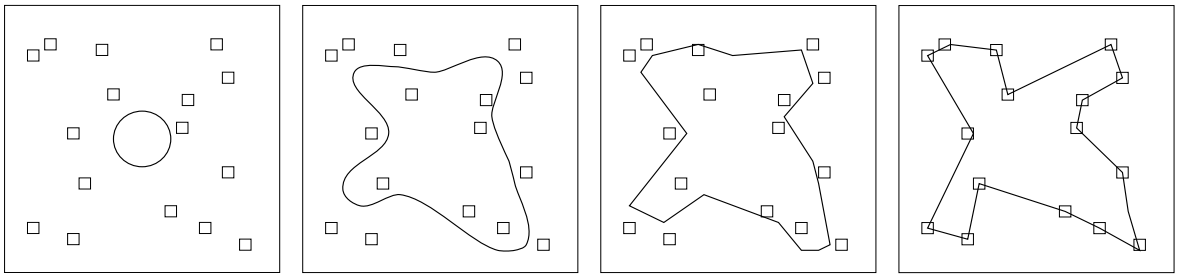

Abbildung 14: Der Handlungsreisende

Algorithmus mit den Koordinaten (x, y) der Städte und hoffen, daß nach dem Lernen die Gewichte gleich den Positionen der Städten sind oder wenigstens nahe an ihnen vorbeigehen. Dieses kann man auch mit der Energiefunktion begründen: Die Neuronengewichte müssen zu den Städten wandern, da dadurch die Vektoren zwischen Gewichten und Eingaben kurz werden, also auch die Energiefunktion minimal.
Bei vernünftiger Wahl der Lernparameter erhält man normalerweise eine Tour, die sich nicht überschneidet, da sich nicht benachbarte Neuronen gegenseitig „abstoßen“. Vergleiche dazu auch den 1-dim Fall in der Abbildungen 9 aus dem letzten Kapitel. Dieses liefert schon recht gute Ergebnisse. Eine Verbesserung kann noch erreicht werden durch eine Modifikation der Lernregel, um Nachbarstädte explizit zu bevorzugen.

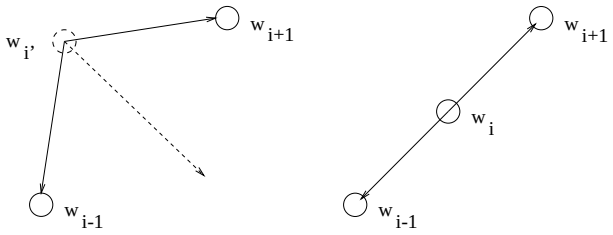

Abbildung 15: Die ideale Position von w_i

4.1 Modifizierte Lernregel

Man kann sich einen elastischen Gummiring vorstellen, der einerseits zu den einzelnen Städten gezogen wird und andererseits so klein wie möglich bleibt. Abbildung 14 zeigt eine solche Tour. Dieses leistet die folgende Lernregel:

$$\Delta\vec{w}_i = \alpha \left(\sum_k N^k(i)(\vec{x}(k) - \vec{w}_i) + \kappa((\vec{w}_{i+1} - \vec{w}_i) + (\vec{w}_{i-1} - \vec{w}_i)) \right)$$

Hier ist $\vec{w}_i$ ein Punkt auf dem Gummiring und $\vec{x}(k)$ die Position der Stadt k. Der erste Term in der Summe zieht jeden Punkt $\vec{w}_i$ mit Faktor $N^k(i)$ zur Stadt $\vec{x}(k)$, der zweite Term hält Nachbarpunkte auf dem Ring mit Faktor κ zusammen. Abbildung 15 soll die Funktion

des zweiten Terms erläutern. Man stelle sich ein Gummiband vor, das den „Punkt“ $\vec{w}_{i'}$ mit seinen Nachbarpunkten verbindet. Lassen wir $\vec{w}_{i'}$ los, wird er in Position $\vec{w}_i$ gezogen, bei der das Gummi am wenigsten beansprucht wird (nach einigem Hin- und Herschwingen). Mathematisch ist die Summe der beiden Vektoren $\vec{w}_{i+1} - \vec{w}_i$ und $w_{i-1} - \vec{w}_i$ nur für diese Position minimal (d.h. 0), in anderen Fällen wird der resultierende Kraftvektor mit in die Lernregel einfließen.

Wahl der Nachbarschaftsfunktion: wir machen die Nachbarschaftsfunktion nur von der Distanz zwischen $\vec{w}_i$ und $\vec{x}(k)$ abhängig: $N^k(i) = \frac{\tilde{N}^k(i)}{\sum_j \tilde{N}^k(j)}$ mit $\tilde{N}^k(i) = \exp\left(-\frac{\|\vec{x}(k)-\vec{w}_i\|^2}{2\sigma^2}\right)$.
Der modifizierte Kohonenalgorithmus liefert für das TSP recht gute Ergebnisse, die ziemlich nahe am Optimum liegen.

Literatur

[ALE90] Igor Aleksander, Helen Morton: An Introduction to Neuronal Computing, Chapman and Hall, 1990

[BLO94] Gunnar Blom, Lars Holst, Dennis Sandell: Problems and Snapshots from the World of Probability, Springer-Verlag, 1994

[HER91] J.Hertz, A.Krogh, R.G.Palmer: Introduction to the theory of neuronal computation, Addison-Wesley, 1991

[KOP94] Kopka, Helmut: LaTeX- Einführung, Addison-Wesley, 1994

[RIG94] Gerhard Rigoll: Neuronale Netze: eine Einführung für Ingenieure, Informatiker und Naturwissenschaftler – Rennigen-Malmsheim: expert-Verl., 1994