Machine Learning for Visual Computing

Zusammenfassung

Ryus

October 30, 2018

Im Lernprozess für die Lehrveranstaltung "Machine Learning for Visual Computing" an der TU Wien habe ich dieses Dokument geschrieben. Ursprünglich als Zusammenfassung gedacht, ist es letztendlich vom Ausmaß her etwas zu einer Art Skriptum ausgeartet. Da ich mir vorstellen kann, dass andere Studierende auch davon profitieren können, stelle ich es hier zur Verfügung. Die Wahrscheinlichkeit ist hoch, dass es einige Tippfehler gibt und ich gebe auch keine Korrektheitsgarantie. Nicht alle Kapitel sind hier vertreten und an manchen Stellen wird auf die Folien verwiesen. Der Inhalt basiert auf dem Stoff vom Wintersemester 2017. Rückmeldungen an suchmich at gmx.at.

Contents

1 Einführung **4**
- 1.1 Polynomial Curve Fitting . . . 4
  - 1.1.1 Minimierung einer Fehlerfunktion . . . 4
  - 1.1.2 Overfitting . . . 5
  - 1.1.3 Regularization . . . 5
- 1.2 Lernkategorien . . . 5
- 1.3 Neuron Models . . . 6
- 1.4 Perceptron . . . 7
  - 1.4.1 Geometrische Interpretation . . . 7
  - 1.4.2 Homogene Erweiterung . . . 8
  - 1.4.3 Perceptron-Online-Training . . . 9
- 1.5 Gradient Descent . . . 10
  - 1.5.1 Perceptron-Batch-Training . . . 10
- 1.6 Linear Regression / Linear Filtering . . . 11
  - 1.6.1 Gradient Descent im Linear Filtering . . . 11
  - 1.6.2 LMS-Regel . . . 12
  - 1.6.3 Direktes Lösen mittels Pseudo-Inverse . . . 13
  - 1.6.4 Bias in LMS-Problemen . . . 13

1.7 Linear Basis Function Models . . . 16
1.7.1 LMS-Lösungen . . . 16
1.8 Netwon-Verfahren . . . 17

2 Multi-Layer Perceptrons 17
2.1 Klassifizierung mittels MLPs . . . 18
2.1.1 Ausdruckskraft von 2-Layer-Networks . . . 18
2.2 Multi-Layer Network Training (Backpropagation) . . . 19
2.2.1 Fehlerableitung nach Hidden-To-Output-Gewichte . . . 20
2.2.2 Fehlerableitung nach Input-To-Hidden-Gewichte . . . 21
2.2.3 Stochastic Backpropagation Algorithm . . . 22

3 Complexity Control 23
3.1 Overfitting und Underfitting . . . 23
3.2 Bias und Varianz . . . 24
3.2.1 Bias-Variance Decomposition . . . 24
3.2.2 Bias . . . 25
3.2.3 Varianz . . . 26
3.2.4 Bias-Variance-Dilemma . . . 26
3.3 Curse of Dimensionality . . . 27
3.3.1 Network Complexity . . . 28
3.3.2 Overfitting Vermeiden . . . 28
3.4 VC-Theorie . . . 29
3.4.1 VC-Dimension . . . 29
3.4.2 Empirisches und echtes Risiko . . . 30

4 Radial Basis Function Networks 31
4.1 Grundlagen . . . 31
4.2 Vereinfachte Notation . . . 32
4.3 Training . . . 32
4.4 Exakte Interpolation . . . 33
4.5 Standard RBF-Networks . . . 33
4.6 Überwachtes Training . . . 34
4.7 RBF-Networks vs. MLPs . . . 35

5 Support Vector Machines 35
5.1 Das primale Optimisierungsproblem . . . 35
5.2 Lagrang'sche Multiplikatoren . . . 36
5.3 Das Duale Optimisierungsproblem . . . 37
5.4 Ergebnisberechnung . . . 39
5.5 Kernel Trick . . . 39
5.6 VC-Dimension . . . 40
5.7 Soft Margin . . . 41

6 Principal Component Analysis **41**
6.1 Notation . . . 42
6.2 Statistische Grundlagen . . . 42
6.3 Subspace Projection . . . 43
6.4 Fehlerminimierung . . . 44
6.5 PCA-Schritte . . . 45
6.6 Principal Component Neural Networks . . . 45

7 Associative Memories & Hopfield Networks **45**
7.1 Einführung . . . 45
7.2 Hamming Distanz . . . 46
7.3 Hopfield Networks . . . 46
7.4 Network Training . . . 46
7.5 46

8 Competitive Learning **47**
8.1 Clustering and Vector Quantization . . . 47
8.2 Neural Gas . . . 48
8.3 Self-Organizing Maps . . . 49

9 Bayessian Regression **49**
9.1 Maximum Likelihood und Maximum Posterior . . . 49
9.2 Predictive Distribution . . . 51

10 EM & Gaussian Mixture Models **52**
10.1 K-Means . . . 52
10.2 Gaussian Mixture Models . . . 52
10.3 General EM . . . 53

1 Einführung

Wir haben ein System, welches Eingabewerte $\mathbf{x}$ erhält und aus diesen den Wert $\mathbf{t} = \mathbf{y}(\mathbf{x}) + \epsilon$ erzeugt. Das ϵ ist ein Fehler, denn wir gehen davon aus, dass die beobachteten Daten verrauscht sind. Diese $\mathbf{t}$-Werte können wir beobachten und anhand der $(\mathbf{x}, \mathbf{t})$-Paare eine Maschine bauen, welche das Verhalten nachstellt. Die Ergebnisse der Maschine werden als $\hat{\mathbf{y}}(\mathbf{x})$ bezeichnet. Das Finden der Funktion $\hat{\mathbf{y}}$ von den n $(\mathbf{x}, \mathbf{t})$-Paaren wird als Supervised Machine Learning bezeichnet. Die Paare werden auch als Trainingsdaten bezeichnet. Der Algorithmus, der diesen Lernprozess durchführt, wird auch als Trainingsalgorithmus bezeichnet.

Wichtig ist hierbei, dass das ML-System auch für neue $\mathbf{x}$-Werte sinnvolle Lösungen generiert. Das Dach bei $\hat{\mathbf{y}}$ wird häufig weggelassen, wenn im Kontext klar ist, dass von einer trainierten Funktion die Rede ist. Meistens basiert diese Funktion noch zusätzlich auf einem Gewichtsvektor $\mathbf{w}$, welcher im Trainingsprozess angepasst wird.

1.1 Polynomial Curve Fitting

Für den Fall, dass wir einen skalaren Input-Wert haben und einen skalaren Output haben, so ist ein mögliches Modell, von einem Polynom Mten[1] Grades auszugehen:

$$\mathbf{y}(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x_j \tag{1}$$

1.1.1 Minimierung einer Fehlerfunktion

Ziel des Lernprozesses ist es nun, jenen Gewichtsvektor $\mathbf{w}$ zu finden, welcher für eine Polynom-Funktion sorgt, die am besten auf die Trainingsdaten passt. Dazu brauchen wir ein Maß, mit welchem wir messen können, wie geeignet ein Gewichtsvektor ist. Eine Möglichkeit dafür ist zum Beispiel die *Sum-of-Squares Error Function*, die uns den halben[2] durchschnittlichen quadratischen Fehler zwischen Trainingsdaten und Funktion angibt:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2 \tag{2}$$

Eine Alternative ist auch der Root-Mean-Square (RMS) Error. Dieser hat den Vorteil, dass er unabhängig von der Anzahl der Daten ist, und sich so Fehler bei verschieden großen Datensets vergleichen lassen. Durch die Wurzel wird außerdem sichergestellt, dass der Wert in der gleichen Dimension wie die beobachteten t-Werte liegt:

$$E_{RMS} = \sqrt{2E(\mathbf{w})/N} \tag{3}$$

[1] M muss zu Beginn ausgewählt werden

[2] halb deshalb, weil das beim Ableiten für Vorteile sorgt und konstante Faktoren keinen Unterschied im Ergebnis machen

1.1.2 Overfitting

Durch Minimisierung dieser Funktion erhalten wir das Polynom mit dem kleinsten Fehler. Der "optimale" Gewichtsvektor wird mit $\mathbf{x}^*$ bezeichnet. Dies funktioniert gut, wenn wir das M gut wählen. Wenn wir es jedoch zu hoch wählen, zum Beispiel gleich $n-1$, so tritt ein Phänomen namens *Overfitting* auf. Die erzeugte Funktion geht genau durch alle Trainingsdaten durch, daher ist der Fehler 0. Allerdings oszilliert sie an anderen Stellen sehr stark und liefert Werte, die keine gute Verallgemeinerung mehr sind. Dies zeigt sich auch durch Vergleich von Testfehler und Trainingsfehler. Während der Fehler auf den Trainingsdaten mit steigendem M immer kleiner wird, steigt der Fehler auf den Trainingsdaten. Daher ist es oft sinnvoller, einen kleinen Fehler in Kauf zu nehmen (insbesondere auch, da die Trainingsdaten sowieso verrauscht sind).

Was lässt sich nun tun, um Overfitting zu vermeiden? Wir können die Modelkomplexität reduzieren: In diesem Fall bedeutet das, das M zu verringern. Wir können aber auch einfach mehr Trainingsdaten nehmen. Eine Kombination von $M = 9$ und $n = 10$ liefert zum Beispiel ein sehr über-angepasstes Resultat, aber $M = 9$ und $n = 100$ liefert eine sehr gute Annäherung an die tatsächlichen Werte.

1.1.3 Regularization

Eine weitere Vorgehensweise gegen Overfitting ist *Regularization*. Ein häufiger Hinweis auf Overfitting ist die Größenordnung der Werte des Gewichtsvektoren. Sind diese sehr groß, herrscht die Gefahr, dass wir Overfitting betreiben. Bei Regularization werden daher Gewichtsvektoren mit hohen Beträgen vermieden. Dies geschieht, in dem zur Fehlerfunktion ein Penalty Term hinzugefügt wird:

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}||\mathbf{w}||^2 \tag{4}$$

Der Parameter λ muss entsprechend gewählt werden. Ein zu großer Parameter sorgt dafür, dass das Overfitting so stark vermeidet wird, dass die Funktion fast nur mehr eine Grade ist und auch nicht zu gebrauchen ist. Ein guter Wert sorgt für eine gute Anpassung an die tatsächliche Kurve. Abbildung 1 demonstriert dies an einem Beispiel. In diesem Beispiel zeigt sich, dass bei zu geringem λ der Trainingsfehler zwar niedrig ist, aber der Testfehler sehr hoch, sprich: Overfitting. Erhöht man das λ, so sind beide Fehler in einem guten Niveau. Wird das λ jedoch zu hoch, so gibt es Underfitting und weder Trainings- noch Testfehler sind zumutbar.

1.2 Lernkategorien

Es gibt verschiedene Arten von Lernalgorithmen: Error-Correction Learning, Hebbian Learning, Competitive LEarning, Memory-Based Learning und Blotzmann Learning.

Außerdem gibt es drei große Lernparadigmen:

- Supervised (Überwacht): Lernen anhand von Beobachtungen von Eingabe- und Ausgabewerten

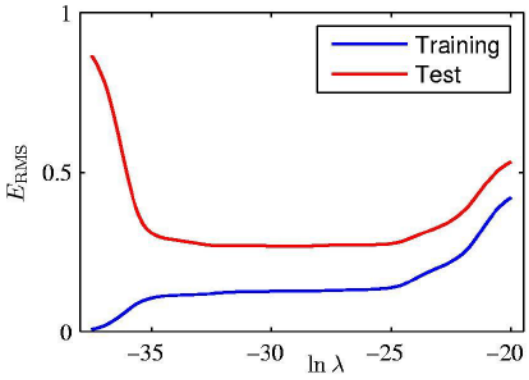


Figure 1: E_RMS vs $ln\lambda$

- Unsupervised (Unüberwacht): Lernen ohne Eingabe- und Ausgabepaare. Es gibt einfach nur Inputdaten und es wird automatisch versucht eine sinnvolle Output-Zuordnung zu finden, z.B. durch Clustering.
- Reinforcement Learning: Lernen anhand von Fehlern.

Aufgaben, die durch Lernen durchgeführt werden können:

- Pattern Association
- Pattern Classification: Einen einzelnen diskreten Ausgabewert den Eingabedaten zuweisen (z.B. ein Bild stellt einen Hund oder eine Katze dar)
- Regression: Stetige Funktionenapproximation. z.B. Polynomial Curve Fitting
- Clustering und Vector Quantization: Unüberwachtes Zuweisen von Labels anhand von automatischer Cluster-Findung
- Density Estimation
- Control

1.3 Neuron Models

Neuronale Modelle bestehen aus einer Verarbeitungseinheit, welche als Input einen d-dimensionalen $\mathbf{x}$-Vektor erhalten. Die Werte x_i dieses Vektors können von anderen Neuronen stammen oder von außen kommen. Zusätzlich wird jedem x_i ein Gewicht w_i zugeordnet (Stärke der zugehörigen Synapse). Gewichts- und Inputvektor haben also die gleiche Dimension. Der Output (auch genannt: Aktivierung) hängt von Inputvektor, Gewichtsvektor und einem weiteren Parameter namens Bias (manchmal als Teil des Gewichtsvektors modelliert) ab.

In der üblichsten Form berechnet sich die Aktivierung eines Neurons wie folgt: Zunächst wird der Netto Input berechnet, indem jeder Input-Wert mit dem zugehörigen Gewicht multipliziert wird:

$$net = \sum_{i=1}^{d} x_i w_i = \mathbf{w}^T \mathbf{x} \tag{5}$$

Als nächstes wird das Aktivierungspotential berechnet, in dem der Bias vom Netto Input abgezogen wird:

$$ap = net - \theta \tag{6}$$

Der Output ("Aktivierung") berechnet sich dann, in dem dieses Aktivierungspotential einer Aktivierungsfunktion g übergeben wird. Diese werden normalerweise so gewählt, dass sie monoton, nichtlinear und beschränkt[3] sind. Simple Funktionen sind zum Beispiel die Treshold Funktion, die Werte kleiner 0 auf 0 mappt und größergleich 0 auf 1, oder die Signum Function, die Werte kleiner 0 auf -1 mappt und Werte größergleich 0 auf +1. Häufig verwendet wird aber die logistische Funktion mit Parameter s (Gleichung 7), da sie differenzierbar ist. Eine weitere Option ist der Tangens Hyperbolicus.

$$g(a) = \frac{1}{1 + e^{-sa}} \tag{7}$$

1.4 Perceptron

Eine der ersten neuronalen Modelle ist Rosenblatt's Perceptron (1958). Dieses nutzt die Signum-Funktion als Aktivierungsfunktion. Rosenblatt schlug außerdem einen Trainigsalgorithmen für das Perceptron vor.

Das Perceptron kann zur binären Klassifikation genutzt werden. In diesem Fall sind unsere Trainingsdaten N d-dimensionale Vektoren mit Target-Werten +1, falls sie zur Klasse w_1 gehören oder -1, falls sie zur Klasse w_2 gehören. Unser System soll nun in der Lage sein, anhand dieser Trainingsdaten die Klasse neuer Daten zu identifizieren. Unser Ziel ist also Werte für $\mathbf{w}$ und θ zu finden, so dass $sgn(\mathbf{w}^T \mathbf{x}_i - \theta) = t_i$. In diesem Fall wollen wir also unseren Trainingsfehler tatsächlich auf 0 kriegen. Die Gefahr des Overfittings wird hier meist ignoriert, ist aber auch vernachlässigbar, da das Perceptron sowieso von einer linearen Separierbarkeit der Daten ausgeht und daher sowieso nicht sehr flexibel ist.

Das Perceptron trennt den Raum durch eine Hyperebene in zwei Hälften (Klasse 1 und Klasse 2) trennen, lässt sich das Perceptron nur auf Probleme anwenden, die linear separierbar sind. Das XOR-Problem lässt sich damit zB nicht lösen.

1.4.1 Geometrische Interpretation

Die Gleichung $\mathbf{w}^T \mathbf{x} = \theta$ definiert eine $d-1$-dimensionale Hyperebene, die die beiden Klassen trennt. Das Perceptron ist also nur sinnvoll anwendbar, wenn die Daten linear separierbar sind. Die Hyperebene wird auch als *decision boundary (Entscheidungsgrenze)* bezeichnet und die beiden Regionen zu beiden Seiten als *decision regions*. Wichtig

[3] ausgenommen z.B. RELU

hier ist die geometrische Interpretation der Parameter: Der Gewichtsvektor w ist der Normalvektor der Ebene und gibt damit deren Orientierung an. Der Bias θ definiert die Position der Hyperebene: Die Normaldistanz zwischen Hyperebene und Ursprung ist durch $\frac{\theta}{||\mathbf{w}||}$ definiert. Die Distanz zwischen einem Punkt und der Hyperebene lässt sich durch $\frac{\mathbf{w}^T\mathbf{x}-\theta}{||\mathbf{w}||}$ berechnen (Vorzeichen gibt die Seite an).

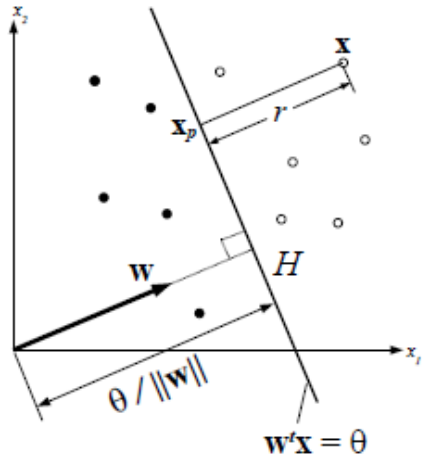


Figure 2: Geometrische Verhältnisse einer Hyperebene

$\mathbf{w}$ und θ können mit einem gleichen Skalar multipliziert werden, ohne dass sich die Hyperebene ändert. Dies macht geometrisch Sinn: Die Richtung des Normalvektors $\mathbf{w}$ bleibt gleich und auch die Normaldistanz zum Ursprung $\theta/||\mathbf{w}||$ bleibt unverändert, da sich der neue Faktor wegkürzt. Auch in Anbetracht der Tatsache, dass wir $sgn(\mathbf{w}^T\mathbf{x}-\theta)$ berechnen, ist klar, dass die Klassifizierung gleich bleibt, da wir uns nur das Vorzeichen anschauen. Als *kanonische Repräsentation* wird jene Parameterbelegung bezeichnet, für welche die Länge des Gewichtsvektors gleich 1 ist. Diese kann einfach erhalten werden, in dem man Gewichtsvektor und Bias durch die Länge des Gewichtsvektors dividiert. In diesem Fall ist das Aktivierungspotential $\mathbf{w}^T\mathbf{x} - \theta$ gleich der Distanz zur Entscheidungsgrenze.

1.4.2 Homogene Erweiterung

Häufig werden Gewichtsvektor und Bias mittels homogenen Koordinaten zusammengelegt. Der erweiterten Vektoren lauten dann:

$$\mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \dots \\ x_d \end{pmatrix}, \mathbf{w} = \begin{pmatrix} -\theta \\ w_1 \\ w_2 \\ \dots \\ w_d \end{pmatrix} \tag{8}$$

Durch diese Konvention wird vieles vereinfacht, das Aktivierungspotential lässt sich so z.B. einfach berechnen durch: $\mathbf{w}^T\mathbf{x}$. Die geometrische Sicht verändert sich nun auch: Wir sind nun in $d+1$ Dimensionen. Alle Input-Vektoren werden in diesen höherdimensionalen Raum projiziert (mit fixer homogener Koordinate 1) und durch diesen Raum wird wieder eine Hyperebene gelegt, welche allerdings fix durch den Ursprung geht.

1.4.3 Perceptron-Online-Training

Nun zum Trainingsalgorithmus: Eine erste wichtige Erkenntnis, die wir benötigen ist, dass wir $sgn(\mathbf{w}^T\mathbf{x}_i) = t_i$ prüfen können, in dem wir $\mathbf{w}^T\mathbf{x}_i t_i > 0$ prüfen. Die Idee des Algorithmus ist es, dass wenn ein Vektor durch den aktuellen Gewichtsvektor falsch klassifiziert wird ($\mathbf{w}^T\mathbf{x}_i t_i <= 0$), dann wird ein Vielfaches von $\mathbf{x}_i t_i$ zu $\mathbf{w}$ dazu addiert. Der konstante Faktor mit dem dieser Wert multipliziert wird, wird mit γ bezeichnet und heißt *Lernrate*. Diese Prozedur muss im Wesentlichen so lange wiederholt werden, bis alle Daten korrekt klassifiziert sind.

1. Initialize w, γ
2. **do**
3. **for** $i = 1$ **to** N
4. **if** $\mathrm{w}^T(\mathrm{x}_i t_i) \leq 0$ (misclassified ith pattern)
5. $\mathrm{w} \leftarrow \mathrm{w} + \gamma \mathrm{x}_i t_i$
6. **end if**
7. **end for**
8. **until** all patterns correctly classified

Figure 3: Perceptron-Trainingsalgorithmus. Schritte 3-7 werden auch als Epoche bezeichnet

Dieser Algorithmus ist ein *Online*-Algorithmus, weil er immer nur ein Datum des Datensatzes gleichzeitig anschaut und nicht alle Fehler gleichzeitig berücksichtigt.

Der initiale Gewichtsvektor ist meistens $\mathbf{0}$, so dass der finale Gewichtsvektor eine Linearkombination der falsch klassifizierten Trainingsdaten ist. In diesem Fall sorgt die Lernrate γ nur für eine Skalierung des Ergebnisses und hat daher keinen Effekt auf die Entscheidungsgrenze. Der Wert ist daher beliebig wählbar und wird häufig auf 1 gesetzt.

Sofern die Daten linear separierbar sind, wird der Algorithmus sicher determinieren, auch wenn es sehr lange dauern kann (Es gibt einen Beweis dafür, der in diesem Dokument ausgelassen ist). Sind die Daten nicht separierbar, so determiniert der Algorithmus nicht. Es ist jedoch möglich eine obere Schranke für die maximale Anzahl an nötigen Schritten zu berechnen.

Der *geometric margin (Geometrischer Abstand)* ist definiert als:

$$gm(\mathbf{x}_i) = \frac{\mathbf{w}^{*T}(\mathbf{x}_i t_i)}{||\mathbf{w}^*||} \tag{9}$$

Dieser gibt die Distanz des iten Datums zur Hyperebene. Sofern das Datum richtig klassifiziert ist, ist der Wert positiv. Der kleinste geometrische Abstand zur Hyperebene (im Bezug auf die Trainingsdaten) bestimmt den geometrischen Abstand (margin) der Hyperebene $gm(\mathbf{w}^*)$.

Das Risiko des Fehlklassifizierens wird meistens durch Mitteln der Klassifizierungsfehler eines Testdatensets festgestellt.

1.5 Gradient Descent

Gradient Descent ist eine simple und häufig angewandte Technik für numerische Optimisierung und Netzwerktraining. Sie lässt sich anwenden, wenn wir eine Kostenfunktion $E(\mathbf{w})$ haben, welche bezüglich des Parameters minimiert werden soll. Es lässt sich der Gradient $\nabla_{\mathbf{w}} E(\mathbf{w})$ (Vektor der partiellen Ableitungen) berechnen. Dieser zeigt in die Richtung des stärksten Anstiegs. Dementsprechend zeigt $-\nabla E$ in die Richtung des stärksten Falles.

Für die Minimierung der Kosten können wir also wie folgt vorgehen: Sei $\mathbf{w}(t)$ der Parameter zur Zeit t, so können wir eine Update-Regel definieren, die einfach den Parameter in Richtung des stärksten Abstieges verändert:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma \nabla E(\mathbf{w}(t)) \tag{10}$$

Auf diese Art nähren wir uns einem Minimum an. γ ist hier wiederum eine Lernrate. Anders als beim Perceptron ist hier die Wahl der Lernrate nicht egal. Ist γ zu klein, dauert das Training sehr lange, ist γ zu groß, überspringen wir möglicherweise das Minimum.

1.5.1 Perceptron-Batch-Training

Angenommen man wolle einen Gradient Descent Algorithmus fürs Perceptron entfernen: Welche Kostenfunktion soll genutzt werden. Eine Möglichkeit wäre die Anzahl an falsch klassifizierten Mustern im Trainingsset, allerdings ist diese Funktion stückweise linear und kann daher nicht für den Gradient Descent genutzt werden, da der Gradient fast überall null ist. Eine bessere Wahl ist:

$$E(\mathbf{w}) = -\sum_{\mathbf{x}_i \in \mathcal{F}} \mathbf{w}^T \mathbf{x}_i t_i \tag{11}$$

$\mathcal{F}$ ist hierbei die Menge aller Training-Daten die falsch klassifiziert wurden. Wie beim Perceptron beschrieben, ist $\mathbf{w}^T \mathbf{x}_i t_i$ genau dann negativ, wenn das Muster falsch klassifiziert wurde. Außerdem steigt der Ausdruck proportional mit der Distanz von der Entscheidungsgrenze. Daher ist der Gesamtausdruck nicht negativ und erhält das Minimum 0 genau dann, wenn alle Muster korrekt klassifiziert werden.

Also berechnen wir den Gradienten:

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = -\sum_{\mathbf{x}_i \in \mathcal{F}} \mathbf{x}_i t_i \tag{12}$$

Daher:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma \sum_{\mathbf{x}_i \in \mathcal{F}} \mathbf{x}_i t_i \tag{13}$$

Die so entstehende Regel ist ähnlich dem Online Trainingsalgorithmus des Perceptrons. In diesem wird ein falsch klassifiziertes Muster hergenommen und $\mathbf{x}_i t_i$ zum Gewichtsvektor addiert. In diesem Algorithmus jedoch werden alle falsch klassifizierten Muster berücksichtigt. Dieser Ansatz nennt sich *batch learning*.

1.6 Linear Regression / Linear Filtering

In diesem Kapitel werden *linear units* untersucht. Das heißt (neuronale) Einheiten, deren Output einfach der Netto Input ist:

$$o(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \tag{14}$$

Homogene Koordinaten dürfen hier vorkommen. Anders als beim Perceptron kann der Output hier jeden Wert aus $\mathbb{R}$ annehmen. Netzwerke aus linearen Einheiten (*lineare Netzwerke*) können verwendet werden, um relle Funktionen zu approximieren. Das Training eines linearen Netzwerks wird als *linear filtering* oder *linear regression* bezeichnet.

Sei $\mathbf{X} \in \mathbf{R}^{(d+1)\times N}$ die Matrix deren Spalten die Input-Vektoren sind und $\mathbf{t} \in \mathbf{R}^N$ ist der Zeilenvektor der die Target-Werte enthält. So wollen wir die Gewichte finden, so dass:

$$\mathbf{w}^T \mathbf{x}_i = t_i, \quad 1 \leq i \leq N \tag{15}$$

Was sich aus beschreiben lässt als:

$$\mathbf{w}^T \mathbf{X} = \mathbf{X}^T \mathbf{w} = \mathbf{t} \tag{16}$$

Dieses System kann eigentlich nur gelöst werden, wenn $N \leq d+1$, was dann aber meist zu Overfitting führt. Meistens wird N allerdings viel größer sein als d.

Das Problem, den besten $\mathbf{w}$-Vektor zu finden, entspricht dem aus der Statistik bekannten Problem des Findens der kleinsten-Quadrate-Regressionsgerade (least mean squares), nur verallgemeinert auf mehrere Dimensionen.

1.6.1 Gradient Descent im Linear Filtering

Um so eine Einheit nun zu trainieren, benötigen wir wieder eine Fehlerfunktion. Auch hier bietet sich der *sum of squares error (SSE)* an:

$$E(\mathbf{w}, \mathbf{X}) = \frac{1}{2} \sum_{i=1}^{N} (t_i - o(\mathbf{x}_i))^2 = \frac{1}{2} (\mathbf{t} - \mathbf{o})(\mathbf{t} - \mathbf{o})^T \tag{17}$$

wobei

$$\mathbf{o} = (o(\mathbf{x}_1), \ldots, o(\mathbf{x}_N))) \tag{18}$$

Vorteile dieser Funktion: Sie ist differenzierbar, nicht-negativ, quadratisch in $\mathbf{w}$ und konvex (daher ist der Gradient eine lineare Funktion).

Nun können wir Gradient Descent anwenden, um den SSE zu minimieren. Dafür schreiben wir die SSE an als:

$$E(\mathbf{w}, \mathbf{X}) = \sum_{i=1}^{N} E(\mathbf{w}, \mathbf{x}_i) \tag{19}$$

$$E(\mathbf{w}, \mathbf{x}_i) = \frac{1}{2}(t_i - \mathbf{w}^T \mathbf{x}_i)^2 \tag{20}$$

Berechnen wir nun den Gradienten von $E(\mathbf{w}, \mathbf{X})$, reicht es einfach die Gradienten von $E(\mathbf{w}, \mathbf{x}_i)$ zu addieren. Berechnen wir also die Ableitung nach dem j-ten Gewicht:

$$\begin{aligned} \frac{\partial}{\partial w_j} E(\mathbf{w}, \mathbf{x}_i) &= \frac{1}{2} \frac{\partial}{\partial w_j} (t_i - \mathbf{w}^T \mathbf{x}_i)^2 \\ = \frac{1}{2} \frac{\partial}{\partial w_j} (t_i - \sum_{k=0}^{d} w_k x_{ki})^2 &= \frac{1}{2} \frac{\partial}{\partial w_j} (t_i - \sum_{k=0}^{d} w_k x_{ki})(-x_{ji}) \\ &= -(t_i - o_i) x_{ji} \end{aligned} \tag{21}$$

Daher lautet der Gradient (also der Vektor aller partiellen Ableitungen):

$$\nabla_{\mathbf{w}} E(\mathbf{w}, \mathbf{x}_i) = -(t_i - o_i)(x_{1i}, \ldots, x_{di})^T = -(t_i - o_i)\mathbf{x}_i \tag{22}$$

Daher:

$$\begin{aligned} \nabla_{\mathbf{w}} E(\mathbf{w}, \mathbf{X}) &= \sum_{i=1}^{N} \nabla_{\mathbf{w}} E(\mathbf{w}, \mathbf{x}_i \\ = -\sum_{i=1}^{N} (t_i - o_i)\mathbf{x}_i &= -\mathbf{X}(\mathbf{t}^T - \mathbf{o}^T) \\ = \mathbf{X}(\mathbf{X}^T \mathbf{w} - \mathbf{t}^T) &= \mathbf{X}\mathbf{X}^T \mathbf{w} - \mathbf{X}\mathbf{t}^T \end{aligned} \tag{23}$$

1.6.2 LMS-Regel

Wenn wir nun ähnlich wie beim Perceptron eine Gewichts-Update-Regel anhand einzelner Trainingsmuster aufstellen erhalten wir die sogenannte *least mean square (LMS)*-Regel, bei welcher der Gradient für ein Trainingsmuster vom Gewichtsvektor abgezogen wird:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma(t_i - o_i)\mathbf{x}_i \tag{24}$$

Alternativ zu dieser Online-Variante, gibt es auch eine Batchvariante in dem der vollständige Gradient genutzt wird.

Wenn der LMS-Algorithmus konvergiert, konvergiert er zur optimalen Lösung $\mathbf{w}^*$ (da die SSE-Funktion konvex ist, gibt es auch nur ein Minimum!). Damit der Algorithmus konvergiert, muss die Lernrate γ ausreichend klein gewählt werden. Zu großes γ

kann dafür sorgen, dass der Algorithmus nicht konvergiert oder neue Inputs so stark ins Gewicht fallen, dass die alten aus dem "Gedächtnis" verloren gehen. Um einige dieser Probleme in den Griff zu kriegen, wird häufig eine zeit-abhängig Lernrate $\gamma(t)$ verwendet, wie z.B.:

$$\gamma(t) = \frac{c}{t}, \quad c \in \mathbf{R}^+ \tag{25}$$

1.6.3 Direktes Lösen mittels Pseudo-Inverse

Wie gesagt: Die SSE-Funktion besitzt in diesem Fall nur ein einziges Minimum, da sie konvex ist. Konvex bedeutet, dass jede Linie, die man durch zwei Punkte auf der Kurve legt, nicht die Kurve an anderen Stellen schneidet. Konvexe Funktionen haben entweder ein Minimum oder mehrere Extremwerte, welche dann aber alle als Tal nebeneinander liegen. Das heißt, wir können das Minimum auch einfach finden, in dem wir die erste Ableitung der Fehlerfunktion gleich 0 setzen:

$$\nabla_{\mathbf{w}} E(\mathbf{w}^*) = \mathbf{0} \tag{26}$$

Dadurch (Herleitung ausgelassen) lässt sich die optimale Lösung wie folgt ausdrücken:

$$\mathbf{w}^* = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{t}^T \tag{27}$$

$(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}$ ist bekannt als die *Pseudo-Inverse*.

1.6.4 Bias in LMS-Problemen

Bisher haben wir in diesem Kapitel den Bias als homogene Koordinate des Gewichtsvektors modelliert. Um mehr Einsicht in den Prozess zu erhalten, betrachten wir die Gradientenberechnung noch einmal, aber diesmal mit explizitem Bias ohne homogene Koordinaten.

$$SSE = \frac{1}{2}\sum_{i=1}^{N}(t_i - (\mathbf{w}^T\mathbf{x}_i + w_0))^2 \tag{28}$$

Der Vektor $\mathbf{w}$ enthält hier kein w_0 mehr. Dieses ist nun als Bias ($w_0 = -\theta$) explizit herausgehoben. Berechnen wir nun die Ableitung nach w_0 und setzen diese gleich null (für die Suche nach dem Extremwert):

$$\frac{\partial}{\partial w_0} SSE = \sum_{i=1}^{N}(t_i - (\mathbf{w}^T\mathbf{x}_i + w_0)) = 0 \tag{29}$$

Durch Aufspalten der Summe kommen wir weiter auf folgendes:

$$\begin{aligned} &\sum_{i=1}^{N}(t_i - (\mathbf{w}^T\mathbf{x}_i + w_0)) \\ &= \sum_{i=1}^{N} t_i - \sum_{i=1}^{N}(\mathbf{w}^T\mathbf{x}_i) - Nw_0 = 0 \end{aligned} \tag{30}$$

Und so weiters:

$$\sum_{i=1}^{N} t_i - \mathbf{w}^T \sum_{i=1}^{N}(\mathbf{x}_i) = Nw_0 \tag{31}$$

Dividieren wir nun die Gleichung durch N, so erscheinen die Mittelwerte von t und x in der Gleichung:

$$\hat{m}_t - \mathbf{w}^T\hat{\mathbf{m}}_{\mathbf{x}} = w_0 \tag{32}$$

Oder auch anders geschrieben:

$$\mathbf{w}^T\hat{\mathbf{m}}_{\mathbf{x}} + w_0 = \hat{m}_t \tag{33}$$

Sprich: Setzen wir den durchschnittlichen Input-Vektor aus dem Trainingsset in unsere lineare Einheit ein, so erhalten wir den durchschnittlichen Targetwert. Das macht ja auch irgendwie Sinn!

Daraus schließen wir: Der Bias sorgt für die Differenz zwischen Mittelwert der Target-Werte und Mittelwert der $\mathbf{w}^T$-Mappings der Input-Werte. Ohne Bias wären diese gleich! Die Regressionsfläche passiert dementsprechend auch den Punkt $(\hat{m}_{\mathbf{x}}, \hat{m}_t)$.

Würden wir in diesem Fall einfach die Pseudo-Inverse vom nicht-homogen-erweitertem Datenset berechnen, so würde die Hyperebene durch den Ursprung gehen (denn $\mathbf{w}^T\mathbf{0} = 0$, anders als bei homogenen Koordinaten wo w_0 herauskommen würde). Das heißt, unser Ergebnis wäre nicht besonders gut.

Wie können wir nun den Gewichtsvektor und den Bias berechnen, wenn wir nicht homogene Koordinaten verwenden? Wir können damit anfangen, dass wir die Gleichung $w_0 = \hat{m}_t - \mathbf{w}^T\hat{\mathbf{m}}_{\mathbf{x}}$ in die SSE-Kostenfunktion einsetzen:

$$
\begin{aligned}
SSE &= \frac{1}{2}\sum_{i=1}^{N}(t_i - (\mathbf{w}^T\mathbf{x}_i + w_0))^2 \\
SSE &= \frac{1}{2}\sum_{i=1}^{N}(t_i - (\mathbf{w}^T\mathbf{x}_i + \hat{m}_t - \mathbf{w}^T\hat{\mathbf{m}}_\mathbf{x}))^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}(t_i - \hat{m}_t - \mathbf{w}^T\mathbf{x}_i + \mathbf{w}^T\hat{\mathbf{m}}_\mathbf{x})^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}((t_i - \hat{m}_t) - \mathbf{w}^T(\mathbf{x}_i - \hat{\mathbf{m}}_\mathbf{x}))^2 \\
&= \frac{1}{2}\sum_{i=1}^{N}(\tilde{t}_i - \mathbf{w}^T\tilde{\mathbf{x}}_i)^2
\end{aligned}
\tag{34}
$$

Das heißt, wenn wir t_i und $\mathbf{x}_i$ durch ihre mittelwertnormalisierten[4] Gegenstücke $\tilde{t}_i$ und $\tilde{\mathbf{x}}_i$ ersetzen, so wird das Problem rein linear und wir brauchen keinen Bias mehr! Daher ist Mittelwertnormalisierung ein häufigiger Vorverarbeitungsschritt. Wenn wir die $\tilde{t}_i$ und $\tilde{\mathbf{x}}_i$-Werten zusammenfassen zu $\tilde{\mathbf{X}}$ und $\tilde{\mathbf{t}}$, so können wir die Lösung des Problems wieder mit der Pseudo-Inversen berechnen:

$$
\mathbf{w}^* = (\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T)^-1\tilde{\mathbf{X}}\tilde{\mathbf{t}}^T \tag{35}
$$

Alternativ kann man diese Formel auch aufschreiben als:

$$
\mathbf{w}^* = \hat{\mathbf{C}}_{\mathbf{XX}}^{-1}\hat{\mathbf{C}}_{\mathbf{xt}} \tag{36}
$$

wobei

$$
\begin{aligned}
\hat{\mathbf{C}}_{\mathbf{XX}} &= \frac{1}{N-1}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T \\
\hat{\mathbf{C}}_{\mathbf{Xt}} &= \frac{1}{N-1}\tilde{\mathbf{X}}\tilde{\mathbf{t}}^T
\end{aligned}
\tag{37}
$$

Diese Einheiten sind bekannt als die Stichprobenkovarianzmatrizen. Für $N \to \infty$ gehen diese gegen die tatsächliche Kovarianz der zu Grunde liegenden Verteilung und $\mathbf{w}^*$ konvergiert gegen den sogenannten *Wiener Filter*.

Das entspricht übrigens dem Standard-Regressionsmodell welches man meist in typischen Statistikbüchern findet!

Zusammenfassend: Wir können beim Least Squares Problem auf zwei Arten vorgehen: Wir können entweder Input und Gewichtsvektor in homogenen Koordinaten angeben und mittels der Pseudo-Inversen den Gewichtsvektor (inkl. Bias) berechnen. Oder wir können keine homogenen Koordinaten angeben und die Pseudo-Inverse nach Abziehen der Durchschnittswerte bilden. Im Anschluss können wir entweder neue Inputs ebenfalls mittelwertnormalisieren oder wir können den Bias explizit berechnen.

[4] sprich: Mittelwert abgezogen

1.7 Linear Basis Function Models

Eine Erweiterung der simplen linearen Regression stellen die *Linear Basis Function Models* dar. Bei diesen wird der d-dimensionale Input-Vektor $\mathbf{x}$ durch m[5] vordefinierte nicht notwendigerweise lineare Funktionen $\phi_j(\mathbf{x})$ - so genannte *Basisfunktionen* - ersetzt. Definiert man $\phi_0(\mathbf{x}) = 1$, so lässt sich der Output beschreiben als:

$$o(\mathbf{x}) = \sum_{i=0}^{m} w_i \phi_i(\mathbf{x}) \tag{38}$$

Ein Beispiel dafür, was sich damit modellieren lässt ist quadratische Regression. Diese ließe sich theoretisch definieren durch (höhere Polynome ließen sich analog definieren):

$$o(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i + \sum_{i=1}^{d} \sum_{j=1}^{d} w_{ij} x_i x_j \tag{39}$$

Durch das Verwenden von $d^2 + d$ Basis-Funktionen kann dies als Linear Basis Function Model dargestellt werden. In der Klassifizierung (z.B. durch Tresholding des Outputs) werden solche Funktionen auch oft als *Generalized Linear Discriminant Functions* bezeichnet. LBMFs können auch sehr nützlich sein um einen Raum in einen höheren Raum zu projizieren, wo sich ein Problem z.B. linear lösen lässt, was im ursprünglichen Raum nicht der Fall war.

Es kann aber passieren, dass man sehr viele Basisfunktionen benötigt, daher muss man aufpassen, dass man genug Trainingsdaten besitzt, damit es nicht zu Overfitting kommt.

1.7.1 LMS-Lösungen

An dieser Stelle sei erwähnt, dass die im vorherigen Abschnitt gezeigten Lösungen sehr ähnlich in LBFMs übernommen werden können.

Die LMS-Online-Updateregel wird zu (oder so):

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \gamma(t_i - o_i)\phi_i \tag{40}$$

Definieren wir eine Matrix $\mathbf{\Phi}$:

$$\Phi = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix} \tag{41}$$

so können wir auch die Pseudo-Inverse definieren:

$$\Phi^\dagger \equiv (\Phi^T \Phi)^{-1} \Phi^T \tag{42}$$

[5] auch genannt $\hat{d}$

1.8 Netwon-Verfahren

Das Newton-Verfahren ist ein Verfahren mit welchem die Nullstellen einer differenzierbaren Funktion gefunden werden können. Dieses kann beim Trainieren von Neuronalen Netzwerken eingesetzt werden, wo wir nach Nullstellen der Ableitung (Gradient) der Fehler-Funktion suchen, um Extremwerte (insb. Minima) zu finden. Haben wir eine initiale Schätzung x_0 so können wir diese verfeinern durch:

$$x_1 = -\frac{f(x_0)}{f'(x_0)} + x_0 \tag{43}$$

Diesen Schritt können wir wiederholen um gegen die Nullstelle zu konvergieren (Die Methode konvergiert nicht immer, aber meistens).

Im Machine Learning haben wir es jedoch meist nicht mit simplen zweidimensionalen Funktionen zu tun, sondern mit Vektoren. Wir definieren daher die Hesse-Matrix, welche die zweiten Ableitungen der Fehlerfunktion beinhaltet:

$$\mathbf{H}(\mathbf{w}) = \left(\frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j} \right)_{1 \leq i,j \leq d} \tag{44}$$

$$\mathbf{H}_{|\mathbf{w}(t)} = \mathbf{H}(\mathbf{w})_{\mathbf{w}=\mathbf{w}(t)} \tag{45}$$

Sofern E zweimal stetig differenzierbar ist, ist die Hesse-Matrix symmetrisch.

Das Newton-Verfahren zweiter Ordnung lautet:

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)} \tag{46}$$

Angewendet auf unseren Fehlerfunktionsfall:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \mathbf{H}^{-1}_{|\mathbf{w}(t)} \nabla E_{|\mathbf{w}(t)} \tag{47}$$

Im linearen LMS-Problem kommt hier bereits nach einem Schritt das Ergebnis der Pseudo-Inverse heraus. Das Newton-Verfahren konvergiert hier also bereits nach einem Schritt.

2 Multi-Layer Perceptrons

Bis jetzt haben wir nur Single-Layer "Networks" besprochen wie etwa das Perceptron oder lineare Regression. Diese sind in ihrer Modellierungsfähigkeit sehr limitiert, da nur lineare Funktionen abgebildet werden können. Eine Alternative wären die besprochenen Linear Basis Function Models, allerdings gibt es hier keine Möglichkeit, passende Basisfunktionen automatisch herauszufinden. Hier tritt das Multi-Layer Perceptron (MLP) - das klassische Neuronale Netzwerk - in Kraft! In dem wir mehrere Neuronen in verschiedene Layer aufteilen und die Outputs der Neuronen eines Layers mit den Inputs der Neuronen des nächsten Layers verbinden, können wir kompliziertere Funktionen

darstellen. Die einzelnen Neuronen können unterschiedliche Aktivierungsfunktionen besitzen. Für Regression ist es z.B. üblich, dass die Neuronen im Output Layer lineare Aktivierungsfunktionen haben. Für Klassifizierung dagegen werden im Output Layer meist Sigmoid Funktionen verwendet.

2.1 Klassifizierung mittels MLPs

Angenommen wir haben ein Klassifizierungsproblem mit M möglichen Klassen. In diesem Fall wird der Target-Wert als Vektor $\mathbf{t}_l$ für den Input $\mathbf{x}_l$ beschrieben, bei welchem die $k$te Komponente genau dann eins ist, wenn das Muster zur $k$ten Klasse gehört, und ansonsten null.

Der Output des MLPs wird dann so interpretiert, dass das Datum zu jener Klasse gehört, die den höchsten Wert im Output-Vektor hat.

2.1.1 Ausdruckskraft von 2-Layer-Networks

MLPs gehören zu den sogenannten *Feed-Forward Networks*, da sie Werte nie zurückschicken, sondern immer nur geradeaus weiter. In der folgenden Diskussion wird von Two-Layer-MLPs (sprich Input, Hidden und Output. "Two" weil es zwei Gewichts-Ebenen gibt) ausgegangen. Dabei werden die Indizes i für Input Units verwendet, j für Hidden Units und k für Output units. Die Aktivierung des $j$ten Hidden Units ist y_j, die Aktivierung des $i$ten Input-Units x_i und die Aktivierung des $k$ten Output Units o_k. w_{ji} beschreibt das Gewicht der Kante vom Input-Unit x_i zum Hidden-Unit y_j. Aktivierungsfunktion von Hidden- und Output-Units ist meistens gleich.

Der Output eines 2L-MLP kann als Vektor $\mathbf{o} = \begin{pmatrix} o_1 & o_2 & \dots & o_c \end{pmatrix}^T$ beschrieben werden, wobei (ohne homogene Vektoren)

$$o_k = g(\sum_{j=1}^{n} w_{kj} g(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}) + w_{k0}) \tag{48}$$

Es wurde gezeigt, dass es für jede Funktion von n Variablen (wobei jede Variable zwischen 0 und 1 liegt) zwei Funktionen Ξ_j und $\Psi_i j$) gibt, so dass:

$$f(\mathbf{x}) = \sum_{j=1}^{2n+1} \Xi_j \left(\sum_{i=1}^{d} \Psi_{ij}(x_i) \right) \tag{49}$$

Two-Layer-MLPs sind von dieser Form, wenn man die Identität als Output-Aktivierungsfunktion nutzt und definiert:

$$\Psi_{ij}(x) = w_{ji} x + w_{j0} \tag{50}$$

$$\Xi_j(x) = \begin{cases} w_{kj} g(x) + w_{k0}, & j \leq n \\ 0, & j > n \end{cases} \tag{51}$$

Dennoch hat dieses Theorem nur begrenzte Relevanz für Neuronale Netzwerke, denn die tatsächlich idealen Funktionen Ξ_j und $\Psi_i j$ können nicht einfach gefunden werden (hängen ja von f ab) und sind oft auch nicht stetig, was allerdings vorteilhaft wäre.

Allerdings ist bekannt, dass zwei-Layer-MLPs dennoch jede beliebige Funktion beliebig genau approximieren können. Dies ist möglich, da sich mittels den Units Gaussian Bumps erzeugen lassen können, und durch Addition von Gaussians kann man beliebige Funktionen approximieren. Es kann jedoch sein, dass sehr viele Neuronen dafür nötig sind.

2.2 Multi-Layer Network Training (Backpropagation)

Wir wollen nun herausfinden, wie wir die Gewichte eines MLP trainieren können. Der Trainingsalgorithmus basiert wieder auf Gradient Descent und der SSE-Kostenfunktion.

Diesmal ist die Kostenfunktion nicht notwendigerweise konvex, daher können wir keine geschlossene Lösung berechnen. Wir könnten die LMS-Regel nutzen um die Gewichte des letzten Layers zu setzen, allerdings könnten damit die Gewicht der Hidden Layers nicht gesetzt werden (*credit assignment problem*). Es gibt jedoch eine Lösung:

Betrachten wir mal die Fehlerfunktion. Diese addiert nun alle Distanzen zwischen Target- und Output-Werten auf:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{l=1}^{N}\sum_{k=1}^{c}(t_{kl} - o_k(\mathbf{x}_l))^2 = \frac{1}{2}\sum_{l=1}^{N}||\mathbf{t}_l - \mathbf{o}_l||^2 \tag{52}$$

Wenn wir den Fehler für ein konkretes Muster extra darstellen, können wir den Fehler auch schreiben lassen als:

$$\begin{aligned} E(\mathbf{w}) &= \sum_{l=1}^{N} E_l(\mathbf{w}) \\ E_l(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_{kl} - o_k(\mathbf{x}_l))^2 &= \frac{1}{2}||\mathbf{t}_l - \mathbf{o}_l||^2 \end{aligned} \tag{53}$$

Wir verwenden nun die Gradient Descent Regel zum Trainieren unserer Gewichte. Zur Erinnerung, die Regel lautet:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma\nabla E_l(\mathbf{w}(t)) \tag{54}$$

An dieser Stelle eine Erinnerung an die Kettenregel: Wollen wir eine Funktion $f(g(x))$ ableiten, so ist das Ergebnis $f'(g(x))g'(x)$. Oder, in Leibniz-Schreibweise:

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)}\frac{dg(x)}{dx} \tag{55}$$

In dieser Schreibweise wirkt die Kettenregel formal grade zu trivial, weil der Bruch einfach mit $dg(x)$ erweitert wird. Theoretisch hätten wir auch mit einer anderen Funktion, die in der Formel vorkommt erweitern können. In dieser Schreibweise ist übrigens auch die Interpretation klarer: Die linke Seite beschreibt, wie stark sich $f(g(x))$ im Verhältnis zu x ändert (im Limes), wenn wir x ganz leicht verändern. Und hier steht nun, dass

das gleich ist der Veränderung von $f(g(x))$ wenn wir $g(x)$ leicht verändern mal der Veränderung von $g(x)$, wenn wir x ganz leicht verändern.

Wir wollen nun den Gradienten, also die partiellen Ableitungen berechnen. Dabei gibt es einerseits die Hidden-To-Output-Gewichte (w_kj) nach denen wir ableiten müssen, und andererseits die Input-To-Hidden-Gewichte (w_ji).

2.2.1 Fehlerableitung nach Hidden-To-Output-Gewichte

Beginnen wir mit den Hidden-To-Output-Gewichten w_kj (vom $j$ten Hidden zum $k$ten Output). Hierzu verwenden wir die Kettenregel, wie oben beschrieben:

$$\frac{\partial E_l}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \frac{1}{2} \sum_{m=1}^{c} (t_{ml} - o_m(\mathbf{x}_l))^2 = \frac{\partial}{\partial w_{kj}} \frac{1}{2} \sum_{m=1}^{c} (t_{ml} - g(ap_m))^2 \tag{56}$$

ap_m ist hier das Aktivierungspotential (sprich der Wert der in die Aktivierungsfunktion geht - also die Outputs des vorherigen Layers mal den entsprechenden Gewichten minus Bias) des kten Output-Units. ap_m ist selber auch von dem Gewicht abhängig, was wir beim Ableiten mit der Kettenregel berücksichtigen müssen. Wir kommen auf:

$$\frac{\partial}{\partial w_{kj}} \frac{1}{2} \sum_{m=1}^{c} (t_{mj} - g(ap_m))^2 = (t_{kl} - g(ap_k)) g'(ap_k) ap'_k \tag{57}$$

Wir wissen, dass das Aktivierungspotential ap_k gleich ist dem Input des $k$ten hidden Units multipliziert mit den entsprechenden Gewichten (- bias, wir gehen davon aus, dass das im homogenen Vektor inkludiert ist):

$$ap_k = \mathbf{w}_k^T \mathbf{y} = \sum_{j=0}^{n} y_j w_{kj} \tag{58}$$

Leiten wir dies ab, so erhalten wir:

$$\frac{\partial ap_k}{\partial w_{kj}} = y_j \tag{59}$$

Daher lautet also die gesamte Ableitung:

$$(t_{kl} - g(ap_k)) g'(ap_k) y_j \tag{60}$$

Der Ausdruck

$$\delta_k = -\frac{\partial E_l}{\partial ap_k} = (t_{kl} - g(ap_k)) g'(ap_k) \tag{61}$$

wird außerdem als die "Empfindlichkeit" des $k$ten Output-Units und gibt an, wie stark sich der Fehler verändert, wenn sich das Aktivierungspotential verändert.

Also lautet die komplette Ableitung:

$$\frac{\partial E_l}{\partial w_{kj}} = -\delta_k y_j \tag{62}$$

(In den Vorlesungsfolien wird hier eine etwas längere Herleitung verwendet, welche Tricks der Leibniz-Schreibweise nutzt. Ich halte diese aber für etwas unklarer und verwirrender.)

Das heißt unsre Gewichts-Update-Regel für die Hidden-To-Output-Gewichte wird zu:

$$w_{kj}(t+1) = w_{kj}(t) + \gamma\delta_k y_j \tag{63}$$

2.2.2 Fehlerableitung nach Input-To-Hidden-Gewichte

Was wir nun als nächstes brauchen sind die partiellen Ableitungen für die Input-To-Hidden-Gewichte w_{ji}. Dies läuft ähnlich ab wie im vorherigen Abschnitt, geht aber etwas tiefer:

$$\frac{\partial E_l}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \frac{1}{2} \sum_{k=1}^{c} (t_{kl} - g(ap_k))^2 = \sum_{k=1}^{c} (t_{kl} - g(ap_k)) g'(ap_k) ap'_k \tag{64}$$

Hier können wir wieder einen Teil durch δ_k ersetzen:

$$\sum_{k=1}^{c} (t_{kl} - g(ap_k)) g'(ap_k) ap'_k = \sum_{k=1}^{c} \delta_k ap'_k \tag{65}$$

Wir müssen jedoch noch die Ableitung ap'_k berechnen. Diese ist (im Gegensatz zu vorher) nicht y_j, weil wir nicht nach w_kj (Hidden-To-Output) sondern nach w_ij (Input-To-Hidden) ableiten:

$$ap_k = \sum_{j=0}^{n} y_j w_{kj} \tag{66}$$

$$\frac{\partial ap_k}{\partial w_{ji}} = y'_j w_{kj} \tag{67}$$

Nun müssen wir weiters y'_j ableiten:

$$y_j = g\left(\sum_{i=1}^{d} w_{ji} x_i \right) \tag{68}$$

$$\frac{\partial y_j}{\partial w_j i} = g'\left(\sum_{i=1}^{d} w_{ji} x_i \right) x_i = g'(ap_j) x_i \tag{69}$$

Wir kommen also insgesamt auf:

$$\frac{\partial E_l}{\partial w_{ji}} = \sum_{k=1}^{c} \delta_k w_{kj} g'(ap_j) x_i = x_i g'(ap_j) \sum_{k=1}^{c} \delta_k w_{kj} = -x_i \delta_j \tag{70}$$

wobei

$$\delta_j = -\frac{\partial E_l}{\partial ap_j} = g'(ap_j) \sum_{k=1}^{c} w_{kj} \delta_k \tag{71}$$

δ_j ist die Empfindlichkeit des $j$ten hidden Units. Bei mehr Layern können die weiteren Empfindlichkeiten analog berechnet werden und daher auch die weiteren partiellen Ableitung ebenfalls analog.

Ableitung für die Hidden-To-Output-Gewichte war:

$$\frac{\partial E_l}{\partial w_{kj}} = -\delta_k y_j \tag{72}$$

Die Ableitung für die Input-To-Hidden-Gewichte lautet sehr ähnlich:

$$\frac{\partial E_l}{\partial w_{ji}} = -\delta_k x_j \tag{73}$$

Das heißt wir können auch für diese Gewichte nun eine Update-Regel definieren:

$$w_{ji}(t+1) = w_{ji}(t) + \gamma \delta_j x_i \tag{74}$$

2.2.3 Stochastic Backpropagation Algorithm

Wir können also den Backpropagation Algorithmus definieren:

1. Initialize $n, \theta, \mathbf{w}, \gamma$
2. **do**
3. $\quad t \leftarrow t + 1$
4. $\quad \{\mathbf{x}_t, \mathbf{t}_t\} \leftarrow$ randomly chosen training pattern $\in \mathcal{S}_{Tr}$
5. $\quad w_{ji} \leftarrow w_{ji} + \gamma \delta_j x_i; w_{kj} \leftarrow w_{kj} + \gamma \delta_k y_j$
6. **until** $\|\nabla E(\mathbf{w})\| < \theta$

Figure 4: Der Backpropagation Algorithmus

Zeile 5 wird für alle Gewichte ausgeführt. Da die Empfindlichkeiten evaluiert werden müssen und diese von den Aktivierungspotentialen abhängen, müssen immer vor der δ-Evaluierung (sogenannter Backward Pass) die Aktivierungen berechnet werden (Forward Pass). Der Algorithmus läuft so lange, bis ein ausreichend kleiner Fehler θ erreicht wird.

Nutzt man übrigens als Aktivierungsfunktion die Sigmoid Function:

$$g(ap) = \frac{1}{1 + e^{-ap}} \tag{75}$$

so lautet die Ableitung:

$$g'(ap) = g(ap)(1 - g(ap)) \tag{76}$$

Backpropagation kann außerdem bei beliebigen Feed-Forward-NNs genutzt werden, nicht nur bei MLPs und es ist auch möglich mit anderen Kostenfunktionen.

3 Complexity Control

3.1 Overfitting und Underfitting

Neuronale Netzwerke können als parametrisiertes Model $\hat{f}_{\mathbf{w}}$ von der tatsächlichen Funktion f verstanden werden. Das Ziel des Trainierens ist es, die Parameter $\mathbf{w}$ so zu setzen, dass die Abweichung zwischen den Werten von f und $\hat{f}_{\mathbf{w}}$ - genannt: *overall risk* - minimiert wird. Dieses overall risk lässt sich mathematisch beschreiben als:

$$\mathbb{E}_{\mathbf{x}}[(\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}))^2] = \int (\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}))^2 p(\mathbf{x}) d\mathbf{x} \tag{77}$$

Eine Erinnerung hier an die Statistik. $\mathbb{E}$ ist der Erwartungswert einer Verteilung, also sozusagen das Mittel über alle überabzählbar unendlich vielen Möglichkeiten, gewichtet nach ihrer Wahrscheinlichkeit. Laut dem LotUS wird der Erwartungswert einer Transformation einer statistischen Größe berechnet durch: $\mathbb{E}(g(X)) = \int_{-\inf}^{inf} g(x) f_X(x) dx$. Die Notation oben ist ein bisschen anders und informeller, da nicht zwischen der statistischen Größe X und ihrer Realisierung x unterschieden wird. Außerdem fehlen die Grenzen des Intervalls, da die Integrationsvariable hier ein Vektor ist. Es wird einfach über den gesamten Vektorraum definiert (Es handelt sich hierbei also *nicht* um ein unbestimmtes Integral!!). $p(\mathbf{x})$ bezeichnet die Dichtefunktion der Verteilung von $\mathbf{x}$.

Die ganze Funktion ist so etwas wie der komplette SSE-Fehler und ist in der Praxis natürlich nicht berechenbar, da f nicht in geschlossener Form verfügbar ist, sondern nur einzelne Samples t beobachtet werden. Diese sind außerdem in der Praxis verrauscht:

$$t = f(\mathbf{x}) + \epsilon, \quad \mathbb{E}[\epsilon] = 0 \tag{78}$$

Der Erwartungswert des Rauschens wird also normalerweise als null angenommen, so dass die Trainingsdaten "im Mittel stimmen". Das Vorhandensein des Rauschens macht es schwieriger, die Modelparameter zu schätzen. In der Praxis haben wir zwei wichtige Ziele:

1. Expressivität: Das Model sollte flexibel und expressiv genug sein, um die zugrunde liegende Target-Funktion f zu repräsentieren.

2. Generalisierbarkeit: Das Model sollte gut generalisierbar sein. Sprich: Egal, welche Samples wir von der zugrunde liegenden Verteilung nehmen, wir sollten so oder so ein ähnliches Ergebnis erhalten, welches nicht so sehr von den konkreten Daten abhängt. Uns geht es also darum, das overall risk (meist gemessen durch den Testdatenfehler) zu minimieren anstatt den Trainingsfehler.

Zu wenig Expressivität führt zu Underfitting (Modell ist zu allgemein, passt nicht mehr auf die Daten, Hoher Trainingsfehler). Das passiert wenn man ein zu schwaches Modell wählt, zum Beispiel lineare Regression, um eine quadratische Funktion zu modellieren, oder ein Perceptron fürs XOR-Problem.

Zu wenig Generalisierbarkeit führt zu Overfitting (Modell passt nur auf die aktuellen Daten, lässt sich nicht verallgemeinern, Niedriger Trainingsfehler, aber meist hoher Testfehler bzw. hohes Overall Risk). Das passiert bei zu flexiblen Modellen, die den Noise in den Daten zu stark berücksichtigen.

Theoretisch könnten beide Probleme gelöst werden, in dem man ein extrem flexibles Modell wählt (für die Expressivität) und unendlich viel Trainingsdaten zur Verfügung stellt (für die Generalisierbareit). In der Praxis muss die Flexibilität des Models der Größe des Trainingssets angepasst werden. Diesen Prozess bezeichnet man als *model selection* oder *complexity control*.

Komplexitätskontrolle kann realisiert werden in dem ein vorhandenes Wissen über den Problembereich berücksichtigt wird (z.B. "Die Funktion soll differenzierbar sein"). Dies wird als *Regularisierung* bezeichnet. Parameter, die steuern, wie stark diese Annahmen sich auswirken, werden als *Regularisierungsparameter* bezeichnet.

Wir haben bereits in der Polynomiellen Regression Beispiele gesehen, in welchen sich ein zu flexibles Modell (Polynom hohen Grades) schlecht auswirkt, in dem es zu Overfitting kommt. Schon leichte Änderungen an den Daten können dann auch schon zu komplett anderen Ergebnisfunktionen führen.

Auf der anderen Seite gibt es auch zu wenig flexible Modelle (Polynom niedrigen Grades oder gar eine Gerade). Eine leichte Änderung der Trainingsdaten wirkt sich hier zwar kaum aus (starke Generalisierbarkeit), aber die Funktion selbst ist eine sehr schlechte Approximation der Target-Funktion. Es kommt also zu Underfitting.

3.2 Bias und Varianz

3.2.1 Bias-Variance Decomposition

Wollen wir die Effektivität eines ML-Modells beschreiben, sind die Begriffe Bias und Varianz sehr wichtig. Um diese zu verstehen bilden wir nun wieder den mittleren Fehler unseres Modells, diesmal aber gemittelt über alle möglichen Trainings-Datensätze $\mathcal{D}$ anstatt über alle möglichen Input-Daten.

$$\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}))^2] \tag{79}$$

Dieses Maß allein gäbe schon eine gewisse Information über die Eignung eines Modells, denn der Fehler sollte ja möglichst klein sein. Doch man kann diesen Fehler noch genauer aufspalten:

Aus der Statistik kennen wir den Verschiebungssatz zur Berechnung der Varianz einer Verteilung:

$$\begin{aligned} \mathbb{V}(X) &= \mathbb{E}(X^2) - (\mathbb{E}(X))^2 \\ \mathbb{E}(X^2) &= (\mathbb{E}(X))^2 + \mathbb{V}(X) \end{aligned} \tag{80}$$

Angewendet auf den mittleren Fehler erhalten wir:

$$\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}))^2] = (\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})])^2 + \mathbb{V}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})] \tag{81}$$

Wir wissen nun weiters, dass die Varianz die mittlere quadratische Abweichung vom Mittelwert ist:

$$\mathbb{V}(X) = \mathbb{E}[(X - \mathbb{E}(X))^2] \tag{82}$$

Angewendet auf diesen Fall, können wir also schreiben:

$$\mathbb{V}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})] = \mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})]\right)^2\right] \tag{83}$$

Da wir die Erwartungswerte über allen möglichen Datensätzen bilden, ist $f(\mathbf{x})$ selbst eine Konstante und kann aus dem Erwartungswert herausgehoben werden, da dieser linear ist:

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})]\right)^2\right] &= \mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x})] + f(\mathbf{x})\right)^2\right] \\ &= \mathbb{E}_{\mathcal{D}}\left[\left(\hat{f}_{\mathbf{w}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x})]\right)^2\right]\end{aligned} \tag{84}$$

Wir kommen so also auf folgende Formel (Bias-Variance-Decomposition):

$$\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}))^2] = (\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})])^2 + \mathbb{E}_{\mathcal{D}}[(\hat{f}_{\mathbf{w}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x})])^2] \tag{85}$$

Diese beiden Summanden nennen sich $bias^2$ und *variance* und entsprechen den bereits vorhin erwähnten - mit Overfitting und Underfitting verwandten - Konzepten, die gleich noch näher erläutert werden.

Diese können wir uns beide noch näher anschauen, um die Konzepte zu verstehen:

3.2.2 Bias

$$bias^2 = (\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x})])^2 \tag{86}$$

Der Bias ist die erwartete Differenz zwischen Target-Funktion und unserer Schätzfunktion. Ist der Bias hoch, heißt das, dass wir uns stark verschätzen. Das heißt unser Modell liefert oft Werte, die total daneben sind. Das entspricht dem Fall des Underfittings. Ist der Bias dagegen niedrig, verschätzt sich die Schätzfunktion im Mittel nur sehr gering und sie passt gut auf die Daten.

Achtung: Nur weil die Differenz im Mittel also gering sein mag, heißt das nicht, dass wir uns nicht dennoch stark verschätzen können. Da das ^{2} außerhalb des Erwartungswertes ist, mitteln wir hier über positive und negative Differenzen, die sich ausgleichen können!

3.2.3 Varianz

$$variance = \mathbb{E}_{\mathcal{D}}[(\hat{f}_{\mathbf{w}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x})])^2] \tag{87}$$

Die Varianz gibt an, was die mittlere Differenz zwischen Schätzfunktion und dem Mittelwert der Schätzfunktion über alle Datensätze hinweg ist. Das bedeutet: Ist die Varianz hoch, so ist die Differenz zwischen Schätzfunktion und mittlerer Schätzfunktion sehr hoch, das heißt, der Datensatz beeinflusst das Ergebnis stark. Dies ist genau der Fall bei Overfitting. Ist die Varianz dagegen niedrig, so ist die mittlere Abweichung vom Mittelwert über alle Datensätze sehr gering, das heißt die Auswahl des Datensatzes spielt keine große Rolle.

3.2.4 Bias-Variance-Dilemma

Idealerweise würde man einen niedrigen Bias und eine niedrige Varianz wollen. Das Problem ist jedoch, dass diese gegenläufig sind.

$$\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathbf{w}}(\mathbf{x}) - f(\mathbf{x}))^2] = bias^2 + variance \tag{88}$$

Durch diese Darstellung wird auch klar, dass diese für einen gegebenen mittleren Fehler gegenläufig sind: Steigt der Bias, so sinkt die Varianz. Steigt die Varianz, so muss der Bias kleiner werden. Sie sind die zwei Teile des Gesamtfehlers.

Es geht also immer darum, ein gutes Mittel zu finden. Haben wir einen niedrigen Bias, so macht unsere Schätzfunktion zwar im Mittel kaum einen Fehler, doch die hohe Varianz sorgt dafür, dass die Schätzfunktion stark vom Datensatz abhängt und die Funktion daher nicht verallgemeinerbar ist. Haben wir eine niedrige Varianz, so hängt unsere Funktion nicht besonders von der Wahl des Trainings-Datensatzes ab und ist damit gut verallgemeinerbar, allerdings machen wir im Mittel einen großes Schätzfehler.

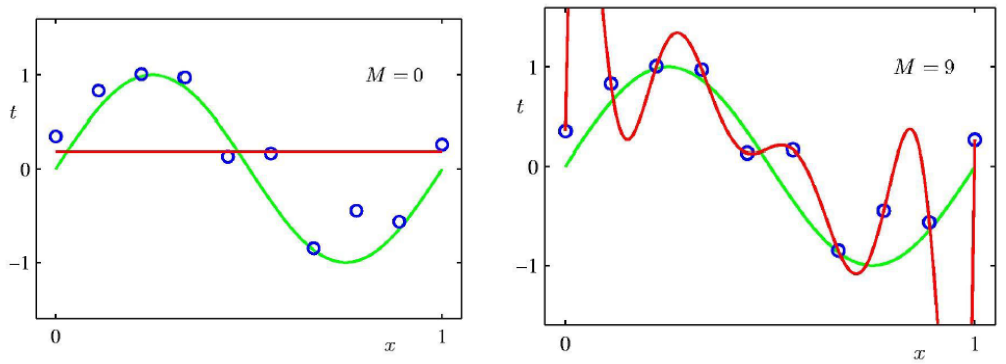
Hier sind nochmal die wichtigsten Zusammenhänge zusammengefasst:

Overfitting:

- Hohe Varianz → stark von Datensatz abhängig
- Niedriger Bias → niedriger Fehler im Mittel
- Modell ist zu expressiv/flexibel
- Modell ist nicht generalisierbar
- Modell hat einen niedrigen Trainingsfehler
- Modell berücksichtigt Noise zu stark
- Leichte Änderungen im Noise führen zu großen Änderungen in der Funktion
- Target-Funktion wird schlecht approximiert

Underfitting:

- Niedrige Varianz → nicht von Datensatz abhängig
- Hoher Bias → großer Fehler im Mittel
- Modell ist nicht expressiv/flexibel genug
- Modell ist generalisierbar
- Modell hat einen hohen Testfehler
- Modell berücksichtigt Noise nicht
- Leichte Änderungen am Noise ändern die Funktion nicht stark
- Target-Funktion wird schlecht approximiert



(a) Underfitting - Geringe Varianz, Hoher Bias

(b) Overfitting - Hohe Varianz, Geringer Bias

Figure 5: Underfitting vs. Overfitting

Indem man ein Modell in unterschiedlichen Komplexitäten testen, kann man MSE (mean-squared error), Bias und Varianz jeweils abschätzen und so die geeignetste Modellkomplexität finden.

3.3 Curse of Dimensionality

Der Fluch der Dimensionalität besagt, dass die Modellkomplexität exponentiell steigt, umso mehr Dimensionen hinzugefügt werden.

Angenommen unser ML-System hat eindimensionale Inputs und wir haben 10 gleichmäßig verteilte Trainingsdaten auf einer Linie der Länge l. Wenn wir eine weitere Dimension hinzufügen, so brauchen wir nun 100 Trainingsdaten um die gleiche Sampling-Rate entlang jeder Dimension beizubehalten. Für jede weitere Dimensionen, steigt die Anzahl der nötigen Samples noch weiter exponentiell an, in diesem Fall nach 10^d.

3.3.1 Network Complexity

Die Komplexität eines neuronalen Netzwerkes hängt ab von der Anzahl der verstellbaren Parameter / Gewichte. Die Komplexität steigt mit der Anzahl an Dimensionen. Um noch gute Lösungen bei hochdimensionalen Inputs zu garantieren, bräuchten wir extrem viele (und unrealistisch viel) Trainingsdaten. Dieses Problem kann jedoch überwunden werden, in dem man zwei Eigenschaften von echten Daten nutzt:

- Intrinsische Dimensionalität: Die Komponenten eines Input-Vektors sind meistens korreliert
- Differenzierbarkeit des Outputs: Der Output wird sich meist nicht beliebig verändern, sondern folgt einer differenzierbaren Funktion

3.3.2 Overfitting Vermeiden

Wenn sich die Dimensionalität der Trainingsvektoren erhöht, gibt es meist zwei Möglichkeiten, um Overfitting zu vermeiden: Zusätzliche Einschränkungen einfügen (Reglarisierung etc.) oder die Dimensionalität reduzieren (wird später besprochen).

Im Fall von Neuronalen Netzwerken wird die Komplexität durch die Anzahl der Hidden Units kontrolliert. Es gibt vier Techniken, die für die optimale Modellkomplexität sorgen können: Cross-Validation, Regularisierung, Network Pruning, Early Stopping Rules

Cross-Validation In der Cross-Validation werden die Trainingsdaten in zwei Submengen aufgespalten: Das *Estimation subset*, welches für das eigentliche Training verwendet wird und das *Validation subset*, welches benutzt wird um den Fehler abzuschätzen. Wir können so unterschiedliche Modelle ausprobieren und prüfen, welches zum minimalen Validation-Fehler führt.

Eine Variante davon ist die k-Fold Cross Validation. In dieser teilen wir das Set in k Untermengen auf. Wir trainieren dann das Modell $k$mal mit jeweils $k-1$ dieser Mengen (immer andere Auswahl) als Estimation Subset. Die übrig gebliebene $k$te Menge wird als Validation Subset verwendet. Mit Hilfe dieser wird der Mean Squared Error berechnet und über alle Durchläufe gemittelt. So erhält man für jedes Modell einen mittleren MSE und kann auswählen, welches Modell man nutzt.

Regularisierung Das Bestrafen von hohen Gewichtsvektoren kann genauso auch bei Neuronalen Netzen verwendet werden. Die Gewichtsfunktion ändert sich so zu:

$$E_\lambda(\mathbf{w}, \mathbf{X}) = E(\mathbf{w}, \mathbf{X}) + \lambda E_r(\mathbf{w}) \tag{89}$$

E_r kann einfach der Betrag des Vektors sein. Man könnte aber auch andere Penalty Terms verwenden.

Network Pruning Diese Methode wird nach dem Training angewandt. Es wird abgeschätzt, wie sehr es sich auswirkt, wenn manche Verbindungen zwischen Units entfernt werden.

Early stopping rules Je länger das Training des NN durchgeführt wird, umso komplexer wird das Mapping. Im Early Stopping wird Cross-Validation verwendet um festzustellen, wann das Training beendet werden kann.

3.4 VC-Theorie

3.4.1 VC-Dimension

Die *Vapnik-Cherveneksis(VC)-Dimension* macht eine Aussage über die Mächtigkeit eines Classifiers. Sie ist nicht immer feststellbar, ist aber prinzipiell wie folgt definiert:

Ein Classifier f mit Parameter $\mathbf{w}$ *zerschmettert* eine Menge von Datenpunkten $\{x_1, x_2...x_n\}$, genau dann wenn für *alle* möglichen Labels dieser Datenpunkte, es eine Parametrisierung $\mathbf{w}$ gibt, so dass die Datenpunkte vom System richtig klassifiziert werden. Die Kardinalität der größten Menge, die von dem Classifier zerschmettert werden kann, nennt sich die *VC-Dimension.*

Beispiele:

- f gibt einfach ein konstantes Label zurück. In diesem Fall kann f nicht mal einen einzigen Punkt zerschmettern, denn wir können den Punkt immer so labeln, dass er den anderen Wert zurückgibt. Daher ist die VC-Dimension 0.
- f ist ein Treshold auf den reellen Zahlen. x wird als 0 gelabelt, wenn $x < w$ und x wird als 1 gelabelt, wenn $x > w$. Ein einzelner Punkt kann nun durch Anpassen des w-Tresholds immer richtig klassifiziert werden. Bei zwei Punkten ist dies jedoch nicht mehr der Fall, da der Punkt mit dem kleineren x-Wert den Target-Wert 1 und der andere den Target-Wert 0 haben könnte. Unser Model geht aber immer davon aus, dass die 0-Werte links sind. Es gibt also keine zwei Punkte, die bei beliebigen Target-Werten immer richtig klassifiziert werden würden. Die Daher ist die VC-Dimension 1.
- f ist in der Lage, eine lineare Entscheidungsgrenze im 2-dimensionalen Raum zu ziehen (z.B. Perceptron). Ordnen wir drei Punkte in einem Dreieck an, so können wir, egal wie das Target-Labeling aussieht, immer die beiden Labels erfolgreich trennen. Die VC-Dimension ist daher mindestens drei. Die Tatsache, dass man auch drei Punkte in einer Linie anordnen könnte und es dann nicht mehr für jedes Labelling funktionieren täte, ist hier egal, da es nur für manche Anordnungen gehen muss. Bei vier Punkten würde es jedoch nicht mehr funktionieren (Satz von Radon), da man durchs Labeln immer so etwas wie das XOR-Problem erschaffen könnte. Daher ist die VC-Dimension 3.
- Die VC-Dimension von einem linearen Classifier im d-dimensionalen Raum ist generell $d + 1$.
- Die Funktion $f(x) = I(sin(wx) > 0)$,[6] hat zwar nur einen anpassbaren Parameter, hat aber unendliche VC-Dimension, da jede beliebige Menge von Datenpunkten korrekt klassifiziert werden kann.

[6] I ist die Indikator-Funktion und gibt 1 zurück, wenn der Inhalt wahr ist, sonst 0

3.4.2 Empirisches und echtes Risiko

Beschäftigen wir uns nun mit der Frage, was die Beziehung zwischen dem Trainingsfehler (empirisches Risiko) E_{emp} und der Erwartung (echtes Risiko) E_{true} ist.

$$E_{emp}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\left((\hat{y}(\mathbf{x}_i, \mathbf{w}) - t_i)^2\right) \tag{90}$$

$$E_{true}(\mathbf{w}) = \mathbb{E}_{\mathbf{x}}\left((\hat{y}(\mathbf{x}, \mathbf{w}) - y(\mathbf{x}))^2\right) \tag{91}$$

E_{emp} wird mittels eines Testsets meist als Schätzer von E_{true} verwendet. Dieses Prinzip nennt sich *Empirical Risk Minimization (EMP)*. Im Allgemeinen gilt, dass, umso mehr Trainingsdaten wir haben (höheres N), umso mehr nähert sich der empirische Fehler an den tatsächlichen Fehler an. In diesem Fall redet man von einem *konsistenten* Schätzer. Damit ein Schätzer konsistent sein kann, muss seine VC-Dimension endlich sein. Hat der Schätzer unendliche Dimension, so ist der empirische Fehler immer 0, während der echte Fehler durchaus hoch sein kann.

Es lässt sich nun folgende Beziehung zwischen den beiden Fehlern angeben:

$$E_{true}(\mathbf{w}) \leq E_{emp}(\mathbf{w}) + \sqrt{\frac{h(\ln\frac{2}{h}+1) - \ln\frac{\eta}{4}}{N}} \tag{92}$$

Hierbei ist h die VC-Dimension. η ist ein Konfidenzlevel. Aus dieser Formel können wir schließen: Ist h/N sehr groß - sprich, es gibt zu wenig Trainingsdaten für die Komplexität des Modells - so ist der Unterschied zwischen empirischem und tatsächlichem Risiko möglicherweise sehr hoch und ein reines ERM-Verfahren ist nicht ausreichend! In diesem Fall muss die Flexibilität des Modells den verfügbaren Daten angepasst werden (*Occam's Razor*-Prinzip: - die einfache Theorie den komplizierten vorziehen).

Verwenden wir ein komplexeres Model mit einer höheren VC-Dimension so wird das empirische Risiko - der Fehler auf den Trainings-Daten - normalerweise kleiner, da es sich mehr auf die Daten anpassen kann. Allerdings steigt so auch der zweite Ausdruck der obrigen Gleichung, und wir können uns also nicht mehr sicher sein. Das macht auch Sinn, wenn wir ans Overfitting denken: Das Model war da zu komplex, der tatsächliche Fehler sehr hoch, aber der Trainingsfehler war sehr klein.

Sofern die VC-Dimension eines Modelles endlich ist, approximiert das empirische Risiko das echten Risiko mit steigendem N. Das heißt, umso mehr Daten wir haben, umso mehr sagt der Trainingsfehler aus.

Der Begriff *Structural Risk Minimization* bezeichnet den Vorgang, die optimale Modellkomplexität zu finden, so dass das maximale Risiko minimal ist. Ist möglich z.B. über Regularisierung mittels Penalty Term.

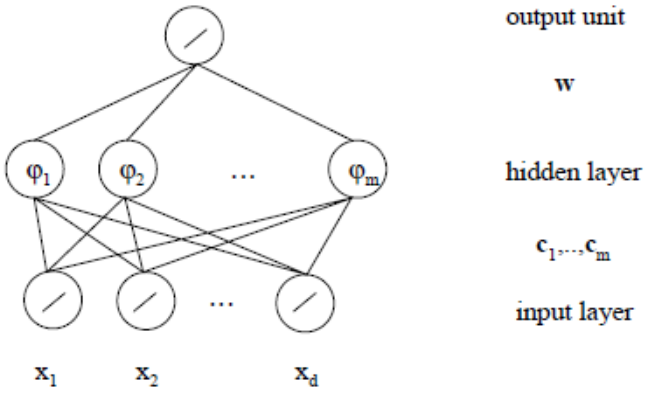


Figure 6: RBF-Netzwerk-Struktur

4 Radial Basis Function Networks

4.1 Grundlagen

Wir haben bereits verschiedene Ansätze gehört um mit nicht-linearen Problemen umzugehen. So z.B. Linear Basis Function Models. Dort wird das Originalsignal in einen anderen Raum gemappt und dort linear getrennt. Da die Basisfunktionen fixiert sind, sind diese Modelle aber eingeschränkt. Eine andere Möglichkeit sind Multi-Layer-Perceptren. Diese sind sehr mächtig, aber das Training kann sehr teuer sein und sie haben auch andere Probleme, etwa suboptimale Lösungen wegen lokaler Minima oder geringe Nachvollziehbarkeit.

Ein *Radial Basis Function (RBF)-Netzwerk* ist eine andere Möglichkeit, nicht-lineare Probleme zu lösen. Ein RBF-Netzwerk besitzt einen m-dimensionalen Gewichtswektor $\mathbf{w}$ und m Aktivierungsfunktionen $\varphi_i : \mathbb{R}^d \to \mathbb{R}$. Außerdem besitzen sie m Zentren $\mathbf{c}_i$. Den Output eines RBF-Netzwerks berechnet sich über die Formel:

$$o(\mathbf{x}) = \sum_{i=1}^{m} w_i \varphi_i(||\mathbf{x} - \mathbf{c}_i||) \tag{93}$$

$||\mathbf{x} - \mathbf{c}_i||$ wird oft auch als r bezeichnet.

Das Ganze kann man sich in einer neuronalen Netzwerkstruktur vorstellen, in welcher die d Input-Werte an m Hidden-Units gesendet werden, welche diesmal keine Linearkombination durchführen, sondern stattdessen die Differenz vom $i$ten Zentrum zum $i$ten Input berechnen und diesen mittels φ_i mappen ($\mathbf{c}_i$ kann als eine Art Kantengewicht interpretiert werden). Im Output-Layer gibt es dann ein "klassisches" Neuron, welches einfach eine Linearkombination mit seinem Gewichtsvektor durchführt. Die Struktur ist in Figur 6 dargestellt.

Zwei häufig genutzte Aktivierungsfunktionen sind:

$$\varphi_i(r) = \exp\left(-\frac{r^2}{2\sigma_i^2}\right) \tag{94}$$

$$\varphi_i(r) = \frac{1}{\sqrt{r^2 + c^2}} \tag{95}$$

σ und c sind zusätzliche fixe Parameter. Beide Funktionen haben eine Glockenform, sind positiv und gehen gegen 0 für $r \to \infty$. Wir gehen in weiterem nur von Gausschen Aktivierungsfunktionen aus (Formel 94).

Das $i$te Hidden Unit hat also genau dann eine hohe Aktivierung, wenn der Input-Vektor nahe dem $i$ten Zentrum ist, wobei Nähe anhand der Streuung σ definiert wird. Das so genannte *Receptive Field* einer RBF-(Hidden-)Einheit ist jener Bereich im Input Space, in welchem die Aktivierung "signifaknt" ist (also höher als ein Schwellwert α):

$$\mathcal{R}_i = \{\mathbf{x} \in \mathbb{R}^d : \varphi_i(||\mathbf{x} - \mathbf{c}_i||) \geq \alpha)\} \tag{96}$$

Aufgrund der Form der Aktivierungsfunktionen nehmen die Receiptive Fields die Form von Hyperkugeln an [7].

Der Output-Funktion des RBF-Netzwerkes ist also einfach die Summe von mehreren gewichteten Gaussian Bumps.

4.2 Vereinfachte Notation

Bisher haben die Aktivierungsfunktionen die Distanz zwischen $\mathbf{x}$ und $\mathbf{c}_i$ erhalten. Manchmal ist es jedoch nützlich, diese Distanzberechnung als Teil der Funktion zu modellieren:

$$\varphi_i(\mathbf{x}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{c}_i||}{2\sigma_i^2}\right) \tag{97}$$

So können wir eine Funktion $\varphi(\mathbf{x}) = (\varphi_1(\mathbf{x}), \ldots, \varphi_m(\mathbf{x}))^T$ definieren, um den Output eleganter zu notieren:

$$o(\mathbf{x}) = \mathbf{w}^T \varphi(\mathbf{x}) \tag{98}$$

Das Ganze kann so eigentlich als spezielles Linear Basis Function Model verstanden werden, nur dass die internen Parameter der Basisfunktionen ($\mathbf{c}_i$) sowie die Anzahl dieser (m) ebenfalls trainiert werden können.

Zusätzlich kann definiert werden: $\varphi(\mathbf{X}) = (\varphi(\mathbf{x}_1), \ldots, \varphi(\mathbf{x}_N))$

4.3 Training

Das Training von RBF-Netzwerken läuft über zwei Phasen: Die erste Phase bezieht sich auf den Hidden Layer. Hier werden die Anzahl der Zentren m, die Zentren $\mathbf{c}_i$ und die Streuungen σ_i festgelegt. Danach können die Ergebnisse von $\varphi(\mathbf{x})$ bereits leicht berechnet werden und die Parameter des Output-Layers können in der zweiten Phase einfach berechnet werden mittels Pseudo-Inverse oder LMS-Regel.

[7] Das Receiptive Field einer Sigmoid-Funktion wäre zum Beispiel ein Halbraum

4.4 Exakte Interpolation

Eine Art, die Parameter des Netzwerkes in der ersten Trainingsphase zu setzen ist wie folgt: $\mathbf{c}_i = \mathbf{x}_i$, wobei $\mathbf{x}_i$ die Trainingsdaten sind ($m = N$). Der Parameter σ ist für alle Einheiten gleich und wird manuell vom Netzwerkdesigner gesetzt. Die φ-Matrix lautet dann:

$$\varphi(\mathbf{X}) = \begin{pmatrix} \varphi_1(||\mathbf{x}_1 - \mathbf{x}_1||) & \cdots & \varphi_1(||\mathbf{x}_N - \mathbf{x}_1||) \\ \vdots & \ddots & \vdots \\ \varphi_N(||\mathbf{x}_1 - \mathbf{x}_N||) & \cdots & \varphi_N(||\mathbf{x}_N - \mathbf{x}_N||) \end{pmatrix} \tag{99}$$

Diese Trainingsart heißt Exakte Interpolation, da die Ergebnisfunktion durch alle Trainingsdaten gehen wird (Trainingsfehler ist 0).

In der zweiten Trainingsphase müssen wir also den Fehler: $(\varphi(\mathbf{X})^T\mathbf{w} - \mathbf{t}^T)^2$ minimieren. Im Allgemeinen Fall könnte man hier mit der Pseudo-Inversen arbeiten, doch bei exakter Interpolation ist dies nicht nötig: Sofern alle Trainingsmuster unterschiedlich sind und Gauss-Funktionen verwendet werden, so ist die Matrix regulär und kann invertiert werden. So können wir den optimalen Gewichtsvektor mit Hilfe dieser berechnen:

$$\mathbf{w}^* = (\varphi(\mathbf{X}^T))^{-1}t^T \tag{100}$$

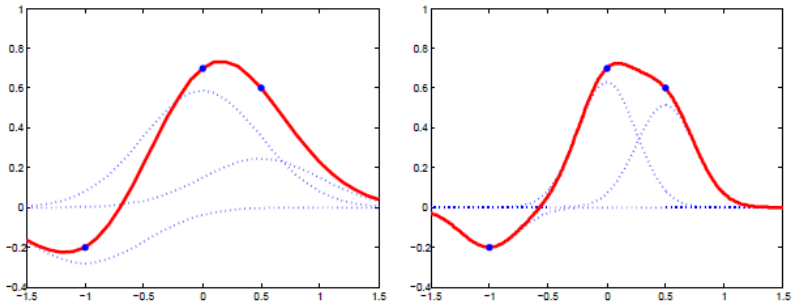


Figure 7: Ergebnis eines RBF-Netzwerks. Die blauen Punkte stellen die Trainingsdaten dar. Die blauen gestrichelten Kurven sind die einzelnen Aktivierungsfunktionen (Radial Basis Functions) φ_i, multipliziert mit dem jeweiligen Gewicht w_i. Die rote Kurve ist die Summe dieser und damit der Output des Netzwerks. In der linken Abbildung wurde $\sigma = 0.5$ verwendet, in der rechten $\sigma = 0.25$.

4.5 Standard RBF-Networks

Allerdings ist exakte Interpolation nicht immer der richtige Weg. Hat man viele Trainingsdaten, so hat man auch viele RBF-Units. Das Training kann so sehr aufwendig werden und es hat hohen Speicherbedarf. Außerdem neigt die Exakte Interpolation stark zum Overfitting! Daher werden in der Praxis meist Netzwerke genutzt, deren Anzahl an Units kleiner ist als die Anzahl der Trainingsdaten ($m < N$).

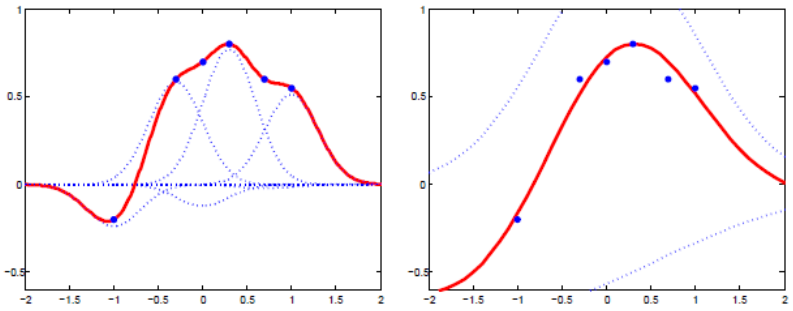


Figure 8: Links: Funktion die ermittelt wurde mittels Exakter Interpolation. Rechts: Funktion die ermittelt wurde mit weniger RBFs.

In allgemeinen Standard-RBF-Netzwerken gilt also meist $m < N$. Außerdem können die Zentren unterschiedlichen den Trainingsdaten sein und die Streuungen können verschieden sein. Es ist möglich einen Bias einzubauen, indem man ein dummy Hidden Unit einbaut mit $\varphi_0(\mathbf{x}) = 1$. Wie können wir in diesem Fall die Parameter $\mathbf{c}_i$, σ_i und m herausfinden?

Zunächst müssen wir die Zentren $\mathbf{c}_i$ herausfinden. Man könnte zum Beispiel ein Subset der Trainingsdaten als Zentren nehmen. Eine bessere Herangehensweise ist es jedoch, Clustering Algorithmen (siehe späteres Kapitel) zu nutzen, die versuchen ein relativ kleines Set an Prototyp-Vektoren zu bestimmen, die das Trainingsdatenset möglichst gut repräsentieren. Die meisten Clustering Algorithmen müssen die Anzahl an Prototypen (m) im Vorhinein wissen. Für manche Ansätze - wie etwa *Growing Neural Gas* - ist dies jedoch nicht notwendig. Ein mögliches Problem bei Clustering-Algorithmen ist jedoch, dass sie nur die empirische Verteilung der Trainingsdaten $\mathbf{X}$ berücksichtigen und nicht den Zusammenhang zwischen Input und Output.

Die Wahl der σ_i hat einen wesentlichen Einfluss auf das Ergebnis des RBF-Netzwerkes. Zu kleine Werte sorgen für "spiky" Funktionen, welche zu Overfitting neigen. Zu große Werte können dafür sorgen, dass die Approximation zu weich wird (*Oversmoothing*) und lokale Details verloren gehen. In der Praxis wird oft das gleiche σ für alle Einheiten genommen, und es basiert meist auf dem Abstand der Zentren, z.B. $\sigma = 2 * avgdist$, wobei $avgdist$ die durchschnittliche Distanz der Zentren bezeichnet.

4.6 Überwachtes Training

Alternativ kann ein RBF-Netzwerk auch mittels Gradient Descent trainiert werden. Durch manuelles Fixieren von m, der Wahl einer Fehlerfunktion, zB dem SSE, und dem Berechnen der Gradienten nach $\mathbf{w}$, $\mathbf{c}_i$ und σ_i, lässt sich ein Backpropagation-ähnlicher Algorithmus durchführen. Hier kann es aber - wie auch bei MLPs - zum Steckenbleiben in lokalen Minima kommen.

4.7 RBF-Networks vs. MLPs

Sowohl RBF-Netzwerke als auch MLPs sind universale Approximatoren: Das heißt, sofern sie genug Units haben, können sie jede beliebige Funktion beliebig genau approximieren.

RBF-Netzwerke haben immer einen Hidden Layer, MLPs können mehrere haben.

Der Netto-Input ist bei RBF-Ns Distanz-basiert, bei MLPs dagegen wird das Skalarprodukt verwendet.

RBF-Ns nutzen lokale Repräsentationen durch die Radial Receptive Fields, MLPs dagegen globale.

Das Training von RBF-Ns ist meist schneller als das von MLPs. Allerdings benötigen sie meist mehr hidden Units als MLPs.

Das Ergebnis von RBF-Netzwerken ist einfacher nachzuvollziehen als das von MLPs.

5 Support Vector Machines

Support Vector Machines (SVMs) sind ein mächtiges Klassifizierungs-Werkzeug, welches in seiner simplen Form lineare Entscheidungsgrenzen bildet, sich aber auch auf komplexere Modelle erweitern lässt und so zum Beispiel auch RBF-NNs oder MLPs darstellen kann. Mit SVMs lässt sich die ideale Generalisierungsfähigkeit finden, ohne Domänen-Wissen einfließen zu lassen!

5.1 Das primale Optimisierungsproblem

Wir kehren zurück zu dem Problem, dass wir eine lineare Entscheidungsgrenze zum Klassifizieren in zwei Klassen finden wollen. Haben wir N Trainingsdatenpunkte $(\mathbf{x}_i, t_i)$, so suchen wir einen Vektor $\mathbf{w}$, so dass: $t_i = sgn(\mathbf{w}^T\mathbf{x}_i + w_0)$ bzw. $(\mathbf{w}^T\mathbf{x} + w_0)t_i > 0$

Zur Erinnerung: Mittels $\frac{\mathbf{w}^T\mathbf{x}+w_0}{||\mathbf{w}||}$ können wir die Distanz zwischen einem Punkt $\mathbf{x}$ und der Entscheidungsgrenze (Hyperebene) berechnen. Der Term $\mathbf{w}^T\mathbf{x} + w_0$ wird oft auch als $d(\mathbf{x}_i)$ bezeichnet.

Wir haben bereits den Perceptron-Algorithmus kennengelernt, der uns solche Entscheidungsgrenze liefert. Theoretisch gibt es jedoch unendlich viele potentielle Entscheidungsgrenzen. Was die SVM uns liefern soll ist die idealste Entscheidungsgrenze mit der kleinsten Fehlerwahrscheinlichkeit. Dazu wird jene Entscheidungsgrenze genommen, die genau in der Mitte der beiden Muster-Gruppen liegt, oder anders gesprochen: Jene Entscheidungsgrenze mit dem größten Margin (der Margin ist der kleinste Abstand zum nächsten Muster aus dem Trainingsdatensatz).

Wenn wir einen Mindestmargin τ verlangen, so verlangen wir also für alle i:

$$\frac{d(\mathbf{x}_i)t_i}{||\mathbf{w}||} \geq \tau \tag{101}$$

Sprich: Die Distanz zwischen Hyperebene und jedem Punkt ist mindestens τ.

Diesen Margin wollen wir maximieren. Wir suchen jenen Gewichtsvektor $\mathbf{w}$ und Bias w_0, so dass das größte τ, welches die obrige Ungleichung erfüllt, maximal ist über alle $\mathbf{w}$.

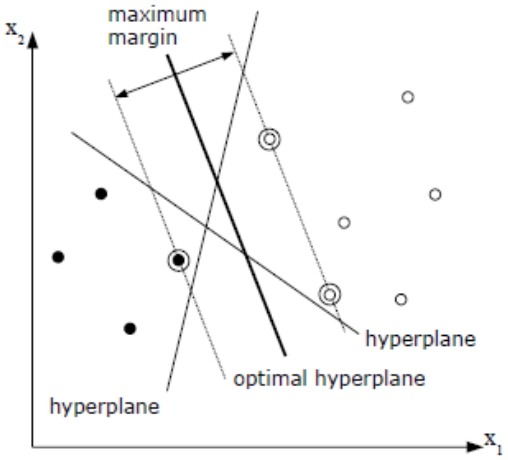


Figure 9: Beispiel für die optimale Entscheidungsgrenze

Allerdings ist so der ideale Gewichtsvektor noch nicht eindeutig bestimmt, denn $\mathbf{w}$ und w_0 können ja beliebig skaliert werden, ohne dass sich die Entscheidungsgrenze verändert. Wir wollen aber *eine* konkrete Lösung herausbekommen. Wir verlangen daher, dass der Wert von $d(\mathbf{x}_i)$ bei den nähesten Trainings-Punkten zur Hyperebene (den sogennanten Suport Vectoren) gleich 1 ist! Dies machen wir, indem wir setzen:

$$\tau = \frac{1}{||\mathbf{w}||} \tag{102}$$

Daraus folgt:

$$d(\mathbf{x}_i)t_i \geq 1 \tag{103}$$

Bekanntlich wollten den Margin maximieren. Durch Gleichung 102 bedeutet dies, dass wir nun die Vektornorm von $\mathbf{w}$ minimieren müssen.

Zusammengefasst: Wir suchen jenes $\mathbf{w}$, so dass $J(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||^2$ minimiert wird, mit der Einschränkung, dass $t_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1$ für alle i.

5.2 Lagrang'sche Multiplikatoren

Für die weitere Vorgehensweise benötigen wir Lagrang'sche Multiplikatoren. Dabei handelt es sich um eine Vorgehensweise zur Extremwertfindung unter Nebenbedingungen:

Angenommen wir haben eine Funktion $f(\mathbf{x})$ und wir suchen jenes $\mathbf{x}$, welches diese Funktion minimiert. Zusätzlich haben wir weitere m Bedingungen $h_i(\mathbf{x} = 0$, die dabei erfüllt bleiben müssen. In einfachen Fällen könnte man die h_i einfach in f einsetzen und dann einfach Minima von f suchen, doch im Allgemeinen ist dies nicht so einfach möglich.

Stattdessen kann man durch geometrische Überlegungen (siehe z.B. "Mathematik für Informatik" von Drmota et.al. Kapitel 6.3 oder Videos der Khan Academy) feststellen,

dass an einer korrekten Lösung die Gradienten von f und h_i parallel sein müssen (und nicht notwendigerweise gleich $\mathbf{0}$), so dass also für alle i gilt ein λ_i bzw. ein β_i gibt, so dass:

$$\nabla f(\mathbf{x}) = \lambda_i \nabla h_i(\mathbf{x}) = -\beta_i \nabla h_i(\mathbf{x}) \tag{104}$$

In der obrigen Gleichung wurden zwei verschiedene übliche Notationen verwendet, wir werden mit der β-Notation fortfahren. Wir definieren nun die Lagrange'sche Funktion:

$$L(\mathbf{x}, \beta) = f(\mathbf{x}) + \sum_{i=1}^{m} \beta_i h_i(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\beta}^T \mathbf{h}(\mathbf{x}) \tag{105}$$

Aufgrund Gleichung 104 können wir schließen, dass an einer korrekten Lösung der Gradient von L gleich $\mathbf{0}$ ist. Wir bilden also alle partielle Ableitungen nach x und β und setzen diese gleich 0, um eine Lösung zu finden[8].

Nun kann es auch sein, dass wir weitere Bedingungen haben, welche die Form $g_i(\mathbf{x}) \leq 0$ annehmen. In diesem Fall erweitern wir die Lagrange'sche Funktion:

$$L(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = f(\mathbf{x}) + \boldsymbol{\alpha}^T \mathbf{g}(\mathbf{x}) + \boldsymbol{\beta}^T \mathbf{h}(\mathbf{x}) \tag{106}$$

Sofern das Optimierungsproblem konvex ist und g_i und h_i affine Funktionen (also der Form $\mathbf{A}\mathbf{x} - \mathbf{b}$) sind, dann gibt es für eine optimale Lösung $\mathbf{x}^*$ die Vektoren $\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*$, so dass die partiellen Ableitungen nach $\mathbf{x}$ und $\boldsymbol{\beta}$ wieder $\mathbf{0}$ sind (nicht unbedingt die nach $\boldsymbol{\alpha}$, und so dass die folgenden (Un-)Gleichungen erfüllt werden:

$$\begin{aligned} \alpha_i^* g_i(\mathbf{x}) &= 0 \\ g_i(\mathbf{x}) &\leq 0 \\ \alpha_i^* &\geq 0 \end{aligned} \tag{107}$$

Beispiel: minimiere $f(x) = x^2$ unter der Bedingung $g(x) = x + 1 \geq 0$. Bilden wir die Lagrange'sche Funktion erhalten wir: $L(x, \alpha) = x^2 + \alpha(x + 1)$. Nun setzen wir die Ableitung 0: $2x + \alpha = 0 \implies x = -\alpha/2$. Wir sind also in der Lage, das x durch die α-Werte auszudrücken. Setzen wir diese Gleichung wieder in die Lagrange'sche Funktion ein erhalten wir: $\frac{\alpha^2}{4} + \alpha$. Indem wir das originale Problem nur durch die α ausdrücken, erhalten wir das so genannte *Duale Optimisierungsproblem*:

Maximiere $\frac{\alpha^2}{4} + \alpha$ unter der Bedingung $\alpha \geq 0$.

Das Lösen dieses dualen Problems ist oft einfacher als das Lösen des primalen Problems, führt aber letztendlich zur gleichen Lösung.

5.3 Das Duale Optimisierungsproblem

Kehren wir nun zurück zu dem SVM-Optimierungsproblem:

[8] Dies gilt, sofern L konvex ist

$$minimiere \frac{1}{2}||\mathbf{w}||^2, sodass : (\mathbf{w}^T\mathbf{x}_i + w_0)t_i \geq 1 \quad \forall i \tag{108}$$

Die Bedingung können wir auch schreiben als $-\mathbf{w}^T\mathbf{x}t_i - w_0t_i + 1 \leq 0$. Bilden wir nun, wie oben beschrieben die lagrange'sche Funktion (Minus wurde aus der Summe herausgehoben):

$$L(\mathbf{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i(\mathbf{w}^T\mathbf{x}_i t_i + w_0 t_i - 1) \tag{109}$$

Wir bilden nun die Ableitung der Lagrange'schen Funktion nach $\mathbf{w}$ und setzen diese gleich 0 [9]:

$$\begin{aligned} \mathbf{w} - \sum_{i=1}^{N} \alpha_i \mathbf{x}_i t_i = 0 \\ \implies \mathbf{w}^* = \sum_{i=1}^{N} \alpha_i^* \mathbf{x}_i t_i \end{aligned} \tag{110}$$

Wir sehen hier also, dass der optimale Gewichtsvektor als Linearkombination der Trainingsdaten ausgedrückt werden kann - eine Tatsache, die aus dem Perceptron-Algorithmus bereits bekannt ist.

Setzen wir nun weiters die Ableitung nach w_0 gleich 0:

$$\sum_{i=1}^{N} \alpha_i^* t_i = 0 \tag{111}$$

Wir wissen, dass einerseits die Ableitungen nach $\mathbf{w}$ und w_0 gleich 0 sein müssen, es müssen aber auch die Bedingungen 107 erfüllt sein müssen. Aus diesen Bedingungen können wir schließen, dass α_i nur dann größer 0 sein kann, wenn $g_i = 0$ (sonst würde die erste Bedingung nicht erfüllt sein). In unserem konkreten Fall bedeutet das, dass $alpha_i$ nur dann größer 0 sein kann, wenn $(\mathbf{w}^T\mathbf{x}_i + w_0)t_i = 1$. Das bedeutet dann, dass $\mathbf{x}$ genau am Margin liegt. Diese Vektoren sind die Support-Vektoren. Alle anderen αs sind gleich 0.

Wir können nun die Gleichungen 110 und 111 in unsere Lagrange'sche Funktion einsetzen[10] und erhalten so eine Gleichung die nur mehr $\boldsymbol{\alpha}$ enthält. Somit erhalten wir die Funktion des dualen Optimierungsproblems:

$$L(\boldsymbol{\alpha}) = -\frac{1}{2}\sum_{i,j=1}^{N} \alpha_i\alpha_j t_i t_j (\mathbf{x}_i^T\mathbf{x}_j) + \sum_{i=1}^{N} \alpha_i \tag{112}$$

[9] Wir nutzen hier die Tatsache, dass $||\mathbf{w}||^2 = \mathbf{w}\mathbf{w} = \mathbf{w}^2$

[10] Ich hätte hier gerne die genauen Schritte erklärt, leider konnte ich sie nicht nachvollziehen und hab auch nirgends eine genaue Erklärung gefunden

Diese Funktion muss nun maximiert werden unter den Bedingungen:

$$\sum_{i=1}^{N} \alpha_i t_i = 0, \alpha_i \geq 0 \tag{113}$$

Wie man solch ein Optimierungsproblem tatsächlich löst, ist ein eigenes Thema (*Quadratische Programmierung*). In der Praxis reicht es, eine passende mathematische Formulierung (wie wir sie hier haben) aufzustellen und diese an ein entsprechendes Framework (z.B. CVXOPT in Python oder quadprog in Matlab) zu übergeben. So können wir also die korrekten α-Werte berechnen.

5.4 Ergebnisberechnung

Wenn wir nun die idealen α_i^* berechnet haben, wie können wir dann den idealen Gewichtsvektor bestimmen? Wie bereits in Gleichung 110 festgestellt wurde, berechnet sich dieser einfach als:

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i^* \mathbf{x}_i t_i \tag{114}$$

Wir können die Entscheidungsfunktion auch direkt mittels der α_i ausgedrückt werden kann:

$$d(\mathbf{x}) = (\mathbf{w}^T \mathbf{x} + w_0) = \sum_{i=1}^{N} \alpha_i t_i \mathbf{x}_i^T \mathbf{x} + w_0 \tag{115}$$

Der optimale Bias berechnet sich aus:

$$w_0^* = t_s - \sum_{i=1}^{N} \alpha_i^* t_i \mathbf{x}_i^T \mathbf{x}_s \tag{116}$$

Wobei $(\mathbf{x}_s, t_s)$ ein beliebiger Support Vector ist.

5.5 Kernel Trick

Stellen wir uns vor, wir würden unsere Trainingsdaten zunächst mittels einer Funktion ϕ transformieren, bevor wir sie der SVM geben. Ein Beispiel dafür könnte eine Funktion $\phi : \mathbb{R}^2 \mapsto \mathbb{R}^6$ sein:

$$\phi(\mathbf{x}) = \begin{pmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \sqrt{2}x_1 x_2 \\ x_1^2 \\ x_2^2 \end{pmatrix} \tag{117}$$

Nun wird an verschiedenen Stellen im Trainings- bzw. Klassifikationsprodukt ein Skalarprodukt $\phi(\mathbf{x}_i)^T\phi(\bar{\mathbf{x}})$ gebildet. In diesem Fall lautet das:

$$\begin{aligned}\phi(\mathbf{x}_i)^T\phi(\bar{\mathbf{x}}) &= 1 + 2x_1\bar{x}_1 + 2x_2\bar{x}_2 + 2x_1x_2\bar{x}_1\bar{x}_2 + x_1^2\bar{x}_1^2 + x_2^2\bar{x}_2^2 \\ &= (x_1\bar{x}_1 + x_2\bar{x}_2 + 1)^2 = (\mathbf{x}^T\bar{\mathbf{x}} + 1)^2\end{aligned} \tag{118}$$

Wir können also dieses Skalarprodukt sehr einfach aus den beiden Input-$\mathbf{x}$-Vektoren berechnen. Indem wir die Formel $(\mathbf{x}^T\bar{\mathbf{x}} + 1)^2$ verwenden, sind wir schneller, als wenn wir die Vektoren erst in die Funktion einfügen und dann das innere Produkt berechnen. Solch eine Funktion wird als *Kernel Funktion* bezeichnet und wird meist mit $K(\mathbf{x}, \hat{\mathbf{x}}$ bezeichnet. Sie dient dazu, das Skalarprodukt nach dem Mapping direkt zu berechnen, ohne das Mapping selbst erst durchführen zu müssen.

Dies reduziert außerdem die Nachteile, die durch das Hinzufügen mehrerer Dimensionen hinzu kommt und es ermöglicht sogar, ein Mapping im unendlichdimensionalen Raum durchzuführen.

Unsere Entscheidungsfunktion wird so also zu:

$$d(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i t_i K(\mathbf{x}_i, \mathbf{x}) + w_0 \tag{119}$$

Häufig genutzte Kernels sind:

- Polynome des Grades d: $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + 1)^d$
- Radial Basis Function (unendlichdimensionales Mapping!): $K(\mathbf{x}, \mathbf{y}) = \exp(-\frac{||\mathbf{x}-\mathbf{y}||^2}{\sigma^2})$
- Multilayer Perceptrons: $K(\mathbf{x}, \mathbf{y}) = \tanh(v(\mathbf{x}^T\mathbf{y})+a)$ (v und a entsprechend gewählt)

Oft hört man auch den Begriff einer Kernel-Matrix. Dies ist einfach eine Matrix deren Elemente $k_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ sind.

Hiermit wachst die SVM zu einem sehr mächtigen Werkzeug, das beliebige Datensätze separieren kann - egal, ob diese im Originalraum linear separierbar waren oder nicht.

5.6 VC-Dimension

Der erwartete Fehler steht in Beziehung mit der erwarteten Anzahl an Support Vektoren:

$$\mathbb{E}(error) \leq \frac{\mathbb{E}(\#SV)}{N} \tag{120}$$

Umso weniger Support-Vektoren (minimal 2) und umso mehr Daten wir haben, umso besser können wir also den Fehler abschätzen.

Wenn wir außerdem einschränken, dass $||\mathbf{w}||^2 \leq c$ gilt, so können wir die VC dimension h abschätzen:

$$h \leq \min(r^2c, d) + 1 \tag{121}$$

r ist hier der Radius der kleinsten Hyperkugel, die alle Trainingsdaten enthält.

5.7 Soft Margin

Manchmal ist es nicht nötig, möglich oder sinnvoll die Trainingsdaten komplett zu separieren. Mit einem Soft Margin wird es ermöglicht, gewisse falsch klassifizierte Trainingsmuster zuzulassen. Dabei kann es auch passieren, dass Muster innerhalb des eigentlich als $\tau = \frac{1}{||\mathbf{w}||}$ definierten Margins liegen.

Um dies zu realisieren, fügen wir für jedes Trainingsmuster eine sogenannte *Slack*-Variable ξ_i ein und wir verlangen, dass gilt:

$$t_i d(\mathbf{x}_i) \geq 1 - \xi_i \tag{122}$$

Es gibt nun drei Fälle pro Trainingsvektor:

- Der Vektor ist außerhalb des Margins und ist korrekt klassifiziert. Also gilt $t_i d(\mathbf{x}_i) \geq 1$. In diesem Fall gilt $\xi_i = 0$.
- Der Vektor ist innerhalb des Margins und ist korrekt klassifiziert. Also gilt $0 \leq t_i d(\mathbf{x}_i) \leq 1$. Das bedeutet $0 < \xi_i \leq 1$
- Der Vektor ist falsch klassifiziert. Also gilt $t_i d(\mathbf{x}_i) < 0$. In diesem Fall ist $\xi_i > 1$.

Das Ziel unseres Optimierungsproblem ist es nun, einerseits den Margin zu maximieren und gleichzeitig die Anzahl an Punkten mit $\xi_i > 0$ - messbar durch $\sum_{i=1}^{N} \xi_i^p$ (p ist eine kleine positive Konstante, meistens 1) - zu minimieren unter Einhaltung der Ungleichung $t_i d(\mathbf{x}_i) \geq 1 - \xi_i$.

Auch hier ist es möglich das duale Problem zu bestimmen. Im dualen Problem wird folgende Funktion maximiert:

$$L(\boldsymbol{\alpha}) = -\frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j t_i t_j (\mathbf{x}_i^T \mathbf{x}_j) + \sum_{i=1}^{N} \alpha_i \tag{123}$$

Unter den Voraussetzungen:

$$\sum_{i=1}^{N} t_i \alpha_i = 0, 0 \leq \alpha_i \leq \frac{c}{N} \tag{124}$$

wobei c eine Konstante ist, mit der der Trade-Off zwischen Komplexität (siehe VC-Dimension) und Anteil an nichtseparierbaren Daten gewählt werden kann (beschränkt $||\mathbf{w}||^2$ nach oben).

6 Principal Component Analysis

PCA ist eine Methode für lineare Feature Extraktion, die auf Dimensionalitätsreduktion abzieht, in dem Korrelationen zwischen den Input-Komponenten gefunden werden.

Eine mögliche Anwendung von PCA ist, dass ein d-dimensionales Signal $\mathbf{x}$ als ein m-dimensionales Signal $\mathbf{y} = f(\mathbf{x})$ kodiert und übertragen werden soll, wobei $m < d$. Das Originalsignal soll dann (möglichst ohne Verlust) durch $g(\mathbf{y})$ wieder rekonstruiert werden können.

6.1 Notation

Die Notation in diesem Kapitel:

- $\mathbf{x}$: Eigentlicher Zufallsvektor ("Signal")
- x_i: Zufallsvariable. $i$te Komponente von $\mathbf{x}$
- $\mathbf{x}_i$: Eine Realisierung/Beobachtung des Zufallsvektors $\mathbf{x}$ mit Index i
- x_{ij}: $i$te Komponente von $\mathbf{x}_j$
- $\mathbf{X}$: Trainingsdaten-Matrix. Zweiter Index adressiert Trainingsdaten, erster Index die jeweilige Komponente. Enthält also die x_{ij}
- $\hat{\mathbf{w}}, \hat{\mathbf{C}}$: Schätzer
- $\tilde{x}$ (Mittelwert-) Normalisierte Daten
- ${}^r\mathbf{x}$ Rekonstruiertes Signal

6.2 Statistische Grundlagen

Signale (also Zufallsvektoren) können in der Statistik durch verschiedene Maße beschrieben werden. Ein wichtiges Merkmal ist der Mittelwert:

$$\mathbf{m} = [\mathbf{x}] = (\mathbb{E}[x_1], \ldots, \mathbb{E}[x_d])^T \tag{125}$$

Sowie die Kovarianz zwischen zwei Vektorkomponenten, mit welcher Hilfe wir herausfinden können, wie stark die beiden Komponenten miteinander korrelieren:

$$\mathbf{c}_{ij} = \mathbb{E}[(x_i - m_i)(x_j - m_j)^T] \tag{126}$$

Alle Kovarianzen können in der Kovarianzmatrix zusammengefasst werden:

$$\mathbf{C} = \begin{pmatrix} c_{11} & \cdots & c_{1d} \\ \vdots & \ddots & \vdots \\ c_{d1} & \cdots & c_{dd} \end{pmatrix} = \mathbb{E}[(\mathbf{x} - \mathbf{m})(\mathbf{x} - \mathbf{m})^T] \tag{127}$$

Der Korrelationskoeffizient r_{ij} gibt uns nun an, wie sehr die beiden Variablen miteinander korrelieren. Ist dieser ca. 1 oder -1, sind sie stark korreliert.

$$r_{ij} = \frac{c_{ij}}{\sqrt{c_{ii} c_{jj}}} \tag{128}$$

Normalerweise ist die echte Verteilung aber nicht bekannt, daher können diese Daten nur abgeschätzt werden. Ein Schätzer für den Mittelwert lautet:

$$\hat{\mathbf{m}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i \tag{129}$$

Um die Empirische Kovarianzmatrix (Schätzer für die echte) zu berechnen, mittelwertnormalisieren wir zunächst unser Datenset:

$$\tilde{X} = (\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_2) = (\mathbf{x}_1 - \hat{\mathbf{m}}, \ldots, \mathbf{x}_N - \hat{\mathbf{m}}) \tag{130}$$

Die Kovarianzen lassen sich nun wie folgt berechnen:

$$\hat{c}_{ij} = \frac{1}{N-1} \sum_{k=1}^{N} (x_{ki} - \hat{m}_i)(x_{kj} - \hat{m}_j) \tag{131}$$

(Achtung: Anders als vorhin behauptet, bezeichnet x_{ki} hier wohl die ite Komponente des kten Vektors.)

(Empirische) Kovarianzmatrizen sind symmetrischen und positiv semidefinit.

6.3 Subspace Projection

In PCA werden die Korrelationen untersucht, um die d Dimensionen (*Superficial Dimensionality*) auf m Dimensionen zu reduzieren (*Intrinsische Dimensionalität*). Dazu wird ein d-dimensionaler Vektor auf einen m-dimensionalen Vektor gemappt.

Wollen wir zum Beispiel alle 2-dimensionalen Punkte auf die Gerade, die durch den Vektor $(2,1)^T$ aufgespannt wird, wie in Abbildung 10 abbilden, können wir das machen in dem wir $y = \frac{(2,1)^T}{||(2,1)^T||}\mathbf{x}$ berechnen. Das Ergebnis ist ein 1-dimensionaler Vektor (also ein Skalar), der uns angibt, wo auf dieser Gerade sich die Projektion befindet. Rekonstruieren können wir dann durch ${}^r\mathbf{x} = \frac{(2,1)}{||(2,1)||}y$. Diese Rekonstruktion mappt von dem Skalar wieder in den ursprünglichen zweidimensionalen Raum, macht hierbei aber eine Rekonstruktionsfehler.

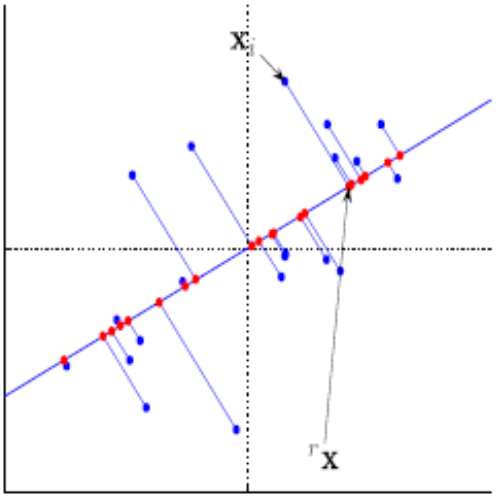


Figure 10: Projektionsbeispiel. Blaue Punkte sind Originaldaten, rote Punkte sind rekonstruiert

.

Sofern der Unterraum, in den wir projezieren eine orthonormale Basis $\mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_m)$ besitzt, so lässt sich die Projektion durch $\mathbf{y} = \mathbf{W}^T\mathbf{x}$ durchführen und die Rekonstruktion über ${}^r\mathbf{x} = \mathbf{W}\mathbf{y} = \mathbf{W}\mathbf{W}^T\mathbf{x}$

6.4 Fehlerminimierung

Wie entscheiden wir nun, in welchen Subraum wir projizieren? Idealerweise wollen wir Projektionen f, g die den Rekonstruktionsfehler minimieren:

$$L_{mse} = \mathbb{E}\left[\sum_{i=1}^{d}(\tilde{x}_i - g(f(\tilde{x}_i)))^2\right] \tag{132}$$

Diesen Fehler wollen wir minimieren. Man beachte, dass das hier ebenfalls eine Art Lernproblem ist. Da wir f und g aber alleine aufgrund der Verteilung der Trainingsdaten erlernen, handelt es sich bei PCA um ein unüberwachtes Lernverfahren.

Setzen wir einfach $d - m$ Variablen auf 0, so lautet der Fehler: $\mathbb{E}[\sum_{i=m+1}^{d}\tilde{x}_i^2]$. Für andere Fälle können wir den Fehler weiter umformen:

$$\begin{aligned}
&\mathbb{E}\left[\sum_{i=1}^{d}(\tilde{x}_i - g(f(\tilde{x}_i)))^2\right] = \\
&\sum_{i=1}^{d}\int(\tilde{x}_i - g(f(\tilde{x}_i)))^2 p(\tilde{x}_i)d\tilde{x}_i \\
&= \int ||\tilde{\mathbf{x}} - g(f(\tilde{\mathbf{x}}))||^2 p(\mathbf{x})d\tilde{\mathbf{x}} \\
&= \mathbb{E}[||\tilde{\mathbf{x}} - {}^r\tilde{\mathbf{x}}||^2]
\end{aligned} \tag{133}$$

Die Integrale die hier verwendet werden sind wieder bestimmte Integrale über den vollständigen entsprechenden Räumen.

Wenn wir f und g als lineare Funktionen annehmen (also der Form $\mathbf{w}^T\tilde{\mathbf{x}}$, wie oben beschrieben), so können wir weiters schreiben:

$$\mathbb{E}[||\tilde{\mathbf{x}} - \mathbf{w}\mathbf{w}^T\tilde{\mathbf{x}}||^2] = \mathbb{E}[\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} - y^2] \tag{134}$$

y ist gleich $\mathbf{w}^T\tilde{\mathbf{x}}$ (y wird auch die Hauptkomponente (Principal Component) genannt). Die genauen Umformungsschritte wurden hier ausgelassen. Das Minimieren des Rekonstruierungsfehlers gleich dem Maximieren der Varianz von y sowie ${}^r\tilde{\mathbf{x}}$.

Wir haben außerdem als Nebenbedingung, dass $||\mathbf{w}|| = \mathbf{w}^T\mathbf{w} = 1$ gilt. Wir können nun die Lagrange'sche Funktion (siehe 5.2) aufstellen:

$$L(\mathbf{w}, \lambda) = \mathbb{E}[\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} - (\mathbf{w}^T\tilde{\mathbf{x}})^2] - \lambda(\mathbf{w}^T\mathbf{w} - 1) \tag{135}$$

Durch Setzen der Ableitung nach $\mathbf{w}$ gleich 0 und weiteren Umformungen erhält man letztendlich:

$$\mathbf{C}\mathbf{w} = \lambda\mathbf{w} \tag{136}$$

Das bedeutet, der ideale Gewichtsvektor ist ein Eigenvektor der Kovarianzmatrix! Wir können also die Kovarianzmatrix mit der empirischen Kovarianzmatrix abschätzen und dann die Eigenvektoren dieser berechnen. Da sie reell und symmetrisch ist, sind auch die Eigenwerte reell. Da sie auch positiv semidefinit ist, sind die Eigenwerte außerdem nicht negativ.

Fun Fact: Ist $\mathbf{E}$ die Matrix, die als Spalten die Eigenvektoren enthält und Λ die Diagonalmatrix der Eigenwerte, so gilt: $\hat{\mathbf{C}} = \mathbf{E}\mathbf{\Lambda}\mathbf{E}^T$. Dies wird als *Eigenwertdekomposition* bezeichnet. Die Kovarianzmatrix von $\mathbf{y}$ ist übrigeng gleich $\mathbf{\Lambda}$, was gleich ist zu $\mathbf{E}^T\hat{\mathbf{C}}\mathbf{E}$

Komponenten mit geringer Varianz kann man auch eigentlich komplett entfernen. Diese enthalten nämlich kaum Information und sind dominiert von Noise.

6.5 PCA-Schritte

1. Mittelwert $\hat{\mathbf{m}}$ berechnen, sowie die mittelwertnormalisierte Trainingsmatrix $\tilde{\mathbf{X}}$ und die empirische Kovarianzmatrix $\hat{\mathbf{C}} = \frac{1}{N-1}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T$
2. Berechne $\mathbf{\Lambda}, \mathbf{E}$ für $\hat{\mathbf{C}}$.
3. Behalte die m Eigenvektoren mit den höchsten Eigenwerten. Die Matrix dieser nennt sich $\mathbf{E}_{[m]}$
4. Wenn wir nun ein neues Input-Muster $\mathbf{x}_{new}$ haben, berechnen wir die neue Repräsentation mittels: $\mathbf{y}_{new} = \mathbf{E}_{[m]}^T(\mathbf{x}_{new} - \hat{\mathbf{m}})$

Ein Beispiel für die Anwendung von PCA sind die *Eigenfaces*, bei welcher sich Bilder von Gesichter als Linearkombinationen von prototypischen Gesichtern, welche die Eigenvektoren einer Gesichter-Verteilung sind, darstellen lassen.

6.6 Principal Component Neural Networks

Siehe Folien.

7 Associative Memories & Hopfield Networks

7.1 Einführung

Associative Memories sind besondere Neurale Netzwerke, die N Input Muster $\mathbf{x}_i$ speichern. Wird dem Netzwerk ein neues Muster $\mathbf{y}$ gegeben, so liefert es als Ergebnis jenes $\mathbf{x}_i$, welches am ehesten $\mathbf{y}$ entspricht.

Dies kann zum Beispiel genutzt werden um Fehler in Binärbildern zu korrigieren.

7.2 Hamming Distanz

Angenommen wir haben nur binäre Trainings-Muster $\mathbf{x}_i \in \{+1, -1\}^d$. Das zu einem neuen Inputvektor $\mathbf{y}$ ähnlichste Muster, könnte zum Beispiel gefunden werden, in dem zu jedem Trainingsmuster die Hamming-Distanz berechnet wird. Die Hamming-Distanz gibt an, wie viele Komponenten in den beiden Vektoren anders sind.

7.3 Hopfield Networks

Eine andere Art, dies zu realisieren ist mithilfe eines Recurrent. Reccurent bedeutet, dass Neuron-Verbindungen nicht immer nur nach vorne gehen, sondern auch zu vorherige Stellen zurückkehren können. Bei RNNs ist es so, dass es immer Update-Regeln gibt, die pro Timestep entweder für alle oder nur für einzelne Neuronen durchgeführt werden. In dem konkreten Fall von Hopfield Networks, schaut diese Updateregel so aus:

$$s_k \leftarrow sgn(\sum_{j=1,\dots,d;j\neq k} w_{kj}s_j) = sgn(\mathbf{Ws}) = \mathbf{s} \tag{137}$$

Das heißt der Output jedes Neurons wirkt sich mit entsprechendem Gewicht w_{kj} auf jedes andere Neuron aus. Die Vereinfachung zur zweiten Schreibweise mit der Matrix $\mathbf{W}$, gilt nur, wenn die Diagonale dieser Matrix nur 0er enthält. Die einzige Ausnahme zu der obrigen Update-Regel tritt ein, falls das Signum 0 ist, dann wird s_k nicht verändert.

Die $\mathbf{s}$-Werte werden als der Zustand des Systems bezeichnet. Das Netzwerk funktioniert so, dass der $\mathbf{y}$-Wert als Zustand gesetzt wird und dann die Gewichte geupdadet werden. In einem Hopfield Network werden die Gewichte so gesetzt, dass der Zustand sich nach einigen Schritten zum entsprechenden ähnlichen Trainingsmuster $\mathbf{x}_i$ verändert und dann stabil bleibt.

Das System ist *stabil*, wenn sich der Status durch Updates nicht mehr verändert.

7.4 Network Training

Wir müssen dafür sorgen, dass für alle Trainingsdaten gilt: $sgn(\mathbf{W}\mathbf{x}_i) = \mathbf{x}_i$. Dies kann prinzipiell einfach ermöglicht werden, indem gewählt wird:

$$\mathbf{W} = \frac{1}{d}\mathbf{x}_i\mathbf{x}_i^T \tag{138}$$

Wobei d gleich dem Quadrat der Länge von $\mathbf{x}_i$ ist, also $\mathbf{x}_i^T\mathbf{x}_i$. Diese Wahl funktioniert, weil $sgn(\frac{1}{d}\mathbf{x}_i\mathbf{x}_i^T\mathbf{x}_i) = sgn(\frac{1}{d}\mathbf{x}_i d) = sgn(\mathbf{x}_i)$ gilt.

7.5 ...

Siehe Folien

8 Competitive Learning

8.1 Clustering and Vector Quantization

Clustering ist der Prozess, eine Datenmenge in Gruppen ähnlicher Daten (Clustern) zu unterteilen. Die Intra-Cluster-Varianz soll maximiert werden, während die Inter-Cluster-Varianz minimiert werden soll. Manchmal wird jeder Cluster durch einen Prototyp-Vektor repräsentiert. *Vector Quantization* dagegen bedeutet, einen gewissen Quantisierungsfehler zu reduzieren, wenn die Anzahl der Prototypvektoren fixiert ist. Bei diesen Verfahren gibt es keine Klassenlabels, es ist also ein unüberwachtes Lernverfahren.

Häufig werden dabei 1-Layer-Netzwerke (also alle Netzwerk-Inputs gehen in alle Netzwerk-Inputs) mit Radial Symmetrischen Aktivierungsfunktionen - wie bei RBF Networks - verwendet:

$$o_k(\mathbf{x}) = \phi_k(||\mathbf{x} - \mathbf{w}_k||) \tag{139}$$

Eine Wahl der Aktivierungsfunktion ist z.B. wieder die Gauss-Funktion. Die Units werden hier oft auch als "Grandmother Cells" bezeichnet.

Das "gewinnende" Unit $s(\mathbf{x})$, ist der Index des Units mit der maximalen Aktivierung. Sind alle σ der Gauss-Funktionen gleich, so ist der Index der maximalen Aktivierung gleich dem Index der minimalen Distanz zum entsprechenden Gewichtsvektor.

Die Voronoi Region V_k ist das Subset des Inputraums, für welches $\mathbf{w}_k$ der nähereste Referenzvektor ist. Haben alle Units das gleiche σ, so ist die Voronoi Region auch einfach gleich jenem Input-Subraum für welches k der Gewinner ist. Das σ kann sozusagen eine weitere Skalierung einfügen.

Diesbezüglich gibt es zwei wichtige Begriffe:

1. Voronoi-Tesselation: Aufteilung des Inputraums in m konvexe Voronoi Regionen, die durch m Referenzvektoren $\mathbf{w}_j$ festgelegt werden

2. Delaunay-Triangulation: Graph, den man erhält, wenn man die $\mathbf{w}_i$ als Knoten nimmt und zwei Knoten verbindet, wenn die entsprechenden Voronoi-Regionen aneinander grenzen.

Haben wir eine Menge an Trainingsvektoren $\mathcal{X}$ gegeben, so bezeichnet $\mathcal{X}_k$, all jene Trainingsvektoren, für die das $k$te Unit der Gewinner ist.

Wir wollen nun schauen, wie wir die Referenzvektoren bestimmen können. Wir beginnen mit einer Menge $\mathcal{W} = \{\mathbf{w}_1, \ldots, \mathbf{w}_m\}$ und führen nun eine Vector Quantization aus. Wir wollen also die durchschnittliche Distanz zwischen Inputvektoren $\mathbf{x}$ und Referenzvektoren $\mathbf{w}_{s(\mathbf{x})}$ berechnen. Empirisch lässt sich diese abschätzen über:

$$L(\mathcal{X}, \mathcal{W}) = \frac{1}{N} \sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{w}_{s(\mathbf{x}_i)}||^2 \tag{140}$$

Die Idee der "LBG-Prozedur" ist, jeden Referenzvektor zum Schwerpunkt seines Voronoi-Sets zu machen. Grund ist, dass wir die durchschnittliche Distanz zwischen Referenzvektoren und zugewiesenen Trainingsdaten $L(\mathcal{X}, \mathcal{W})$ minimieren wollen.

Praktisch lässt sich dies also so machen:

$$\mathbf{w}_k = \frac{1}{|\mathcal{X}_k|} \sum_{x_i \in \mathcal{X}_k} \mathbf{x}_i \tag{141}$$

Der Batch-VQ-LBG-Algorithmus initialisiert alle Referenzvektoren zufällig (oder anders) und weißt den Trainingsdaten den nähersten Referenzvektor zu. In einer Schleife werden dann alle Referenzvektoren entsprechend der obrigen Regel angepasst. Dies geschieht so lange, bis sich die durchschnittliche Distanz $L(\mathcal{X}, \mathcal{W})$ nicht mehr ändert.

Alternativ gibt es auch die Online-Variante. In dieser läuft eine Schleife eine vorgegebene Anzahl an Malen durch. In jedem Schleifendurchlauf wird ein Input-Vektor gewählt, das Winner Unit wird festgestellt und der Referenzvektor dieses wird in die Richtung des Input-Vektors bewegt nach der Regel: $\mathbf{w}_s \leftarrow \mathbf{w}_s + \gamma(t)(\mathbf{x}_i - \mathbf{w}_s)$. Die Lernrate $\gamma(t)$ kann entweder konstant gewählt sein (was dazu wirkt, dass der Einfluss von Vektoren sich über die Zeit exponentiell vermindert), oder z.B. harmonisch: $\gamma(t) = 1/t$ (was dazu führt, dass alle Vektoren gleichgewichtet sind). In diesem Fall wird der Algorithmus auch *k-means* genannt. Es gibt auch eine Variante, die ähnlich ist zu Simulated Annealing: $\gamma(t) = \gamma_0(\gamma_{end}/\gamma_0)^{(t/t_{max})}$. Dies ermöglicht das Entkommen aus lokalen Minima.

8.2 Neural Gas

Im Neural Gas haben wir wieder eine Menge von Gewichtsvektoren (Units) $\mathbf{w}_i$, die zur Vektor-Quantisierung verwendet werden können. Anders als bei obrigen Algorithmus werden jedoch Gewichtsvektoren in der Nähe in einem Anpassungsschritt mitangepasst. Der Trainingsalgorithmus schaut versimpelt wie folgt aus (pro Iteration):

1. Wähle einen zufälligen Input Vektor $\mathbf{x}_i$.
2. Finde das näheste Unit $s(\mathbf{x}_i)$
3. Sortiere alle Units nach Distanz zu $\mathbf{x}_i$.
4. Update alle Gewichtsvektoren nach der Regel (bewegt die Vektoren näher zu $\mathbf{x}_i$:

$$\delta \mathbf{w}_{s_r} = \gamma(t) h(t, r)(\mathbf{x}_i - \mathbf{w}_{s_r}) \tag{142}$$

   Wobei $\mathbf{w}_{s_r}$ der $r$te Gewichtsvektor in der nach Distanz sortieren Liste ist, γ eine Lernrate ist und h eine Funktion, die angibt, wie stark das Element, gegeben die Distanz zum Referenzvektor und der aktuellen Zeit, verschoben werden soll.

Es gibt noch Erweiterungen von Neural Gas wie Competitive Hebbian Learning oder Growing Neural Gas (mit variabler Unit-Anzahl). Siehe Folien.

8.3 Self-Organizing Maps

SOMs funktionieren ähnlich wie Neural Gas, nur ist eine Topologie zwischen den Units definiert, z.B. eine Gitter-Struktur. Anstatt dass im Update-Schritt andere Units nach Distanz der Gewichtsvektoren angepasst werden, wird hier die Entfernung auf dieser Gitter-Struktur verwendet. Der Algorithmus sieht wie folgt aus (pro Iteration):

1. Wähle einen zufälligen Input Vektor $\mathbf{x}$.
2. Finde das näheste Unit $s(\mathbf{x})$.
3. Verschiebe den Gewichtsvektor dieses Units näher zum Input Vektor. Verschiebe ebenfalls die Nachbarn (und die Nachbarn der Nachbarn...) näher zum Input Vektor aber gewichtet mit der Funktion h, die mit fortschreitender Nachbarschaft einen kleineren Wert liefert. Darüber hinaus gibt es auch noch eine Lernrate γ. Die h-Funktion wird während des Lernprozesses normalerweise auch immer kleiner.

9 Bayessian Regression

In diesem Kapitel wird mit $\mathcal{N}(x|\mu, \sigma)$ die Dichtefunktion einer (multivariaten) Normalverteilung mit Mittelwert μ und Varianz $sigma$ bezeichnet.

9.1 Maximum Likelihood und Maximum Posterior

Kehren wir zurück zu dem Thema der Regression. Wir gehen ja davon aus, dass unsere Target-Werte mit Noise behaftet sind: $t = y(\mathbf{x}, \mathbf{w}) + \epsilon$. Es wird normalerweise davon ausgegangen, dass dieser Noise Gaussian ist. Der Mittelwert dieses Gaussian Noise ist der tatsächliche unverrauschte Wert $\mu = y(\mathbf{x}, \mathbf{w})$ und die Streuung/Varianz wird mit $\sigma = \beta^{-1}$ bezeichnet (Umso größer β, umso weniger fällt das Rauschen auf). Wir können also zusammenfassen:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) \tag{143}$$

Sprich: Die Wahrscheinlichkeit, bei einer Beobachtung den Wert t zu erhalten (gegeben $\mathbf{x}, \mathbf{w}$ und β) ist normalverteilt. Man beachte, dass wir im Normalfall die Parameter der Verteilung (y und β) nicht kennen!

Angenommen wir haben nun ein ganzes Trainingsset $(\mathbf{X}, \mathbf{t})$, so gilt:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}) \tag{144}$$

Also: Die Wahrscheinlichkeit den Target-Vektor $\mathbf{t}$ für bestimmte Daten zu erhalten, ist gleich dem Produkt der Wahrscheinlichkeiten die einzelnen ts zu erhalten. Gehen wir davon aus, dass sich die zu Grunde liegende Funktion als LBFM realisieren lässt, so können wir dies auch schreiben als:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|, \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}) \tag{145}$$

Diese Wahrscheinlichkeit in Abhängigkeit von $\mathbf{w}$ wird auch als *Likelihood* bezeichnet.

Unser Ziel im Machine Learning ist ja, den tatsächlichen Wert von $\mathbf{w}$ herauszufinden. Wir werden nun in weiterem den aus der Statistik bekannten Ansatz der *Maximum Likelihood* nutzen, um jenes $\mathbf{w}$ zu finden, welches die maximale Likelihood hat. Dazu werden wir als erstes die Log-Likelihood bilden. Da der Logarithmus eine monotone Funktion ist, verändert er Minima und Maxima nicht und die Funktion wird einfacher zum Handhaben:

$$\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \sum_{n=1}^{N} \ln \mathcal{N}(t|, \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}) = \frac{N}{2} \ln \beta - \frac{N}{2} \ln 2\pi - \beta \frac{1}{2} \sum_{n=1}^{N} (t_n - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n))^2 \tag{146}$$

Die Summe am Ende dieser Gleichung entspricht dem SSE! Wir erkennen also, dass das Maximieren der Likelihood gleich ist dem Minimieren des SSE, was eine Variante ist, die wir bereits probiert haben und die leider sehr anfällig für Overfitting sein kann! Wir brauchen also was besseres!

Wir wollen nun einen Bayes'schen Ansatz wählen. Hierzu eine Erinnerung an das Bayes-Theorem:

$$P(w|t) = \frac{p(t|w)P(w)}{p(x)} \tag{147}$$

Die einzelnen Ausdrücke in dieser Gleichung werden auch bezeichnet als:

$$posterior = \frac{likelihood \times prior}{evidence} \tag{148}$$

Unsere Gleichung oben kann als Likelihood dienen. Wir wollen nun den Posterior herausfinden.

Überlegung wir uns mal eine mögliche Prior-Verteilung über $\mathbf{w}$ ohne ein gegebenes Trainingsset. Man könnte z.B. einfach von einer Verteilung um $\mathbf{0}$ ausgehen mit einer gegebenen Varianz α^{-1}:

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) \tag{149}$$

Nun haben wir also eine Likelihood und einen Prior. Da der Evidence konstant ist, brauchen wir diesen nicht unbedingt weiter berücksichtigen, wir stellen fest, dass:

$$posterior \propto likelihood \times prior \tag{150}$$

(Das Symbol $\propto$ bedeutet "proportional zu", also nur um einen konstanten Faktor (dem Evidence) verschieden). In diesem Fall bedeutet das:

$$p(\mathbf{w}|\mathbf{X}, \mathbf{t}, \alpha, \beta) \propto p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) p(\mathbf{w}|\alpha) \tag{151}$$

Dies können wir nutzen, um eine Formel für den Posterior zu finden. Indem wir unsere Formeln für Prior und Likelihood einsetzen und den Logarithmus ziehen erhalten wir nach einigen Umformungen:

$$\ln p(\mathbf{w}|\mathbf{X}, \mathbf{t}, \alpha, \beta) = -\frac{\beta}{2}\sum_{n=1}^{N}(t_n - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_n))^2 - \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + C \tag{152}$$

Das C am Ende ist eine Konstante, die durch den nicht weiter spezifizierten Evidence zustande kommt und ist im Wesentlichen irrelevant, da wir die Wahrscheinlichkeit bezüglich $\mathbf{w}$ maximieren wollen. Der Ausdruck links ist wieder SSE und der Ausdruck rechts entspricht einem Peanlty-Term, wie wir ihn in der Regularisierung verwenden. So schließt sich der Kreis! Wir sind wieder bei der Strukturellen Risikominimierung zur Overfitting-Vermeidung angelangt!

Wir können übrigens sowohl das Maximum Likelihood (ML) als auch das Maximum Posterior (MAP) Problem in geschlossener Form lösen. Indem wir die Gradienten der Wahrscheinlichkeitsfunktionen berechnen und diese 0 setzen erhalten wir:

$$\begin{aligned} \mathbf{w}_{ML} &= (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{t} \\ \mathbf{w}_{MAP} &= (\frac{\alpha}{\beta}\mathbf{I} + \mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{t} \end{aligned} \tag{153}$$

Die erste Variante entspricht mal wieder der Multiplikation mit der Pseudoinversen (Transponierungen sind hier umgekehrt als im ersten Kapitel, da die Vektoren hier Zeilen sind und nicht Spalten).

Das Ganze lässt sich noch verallgemeinern. Wir sind jetzt von einem Prior $\mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I})$ ausgegangen. Im Allgemeinen könnten wir aber auch eine andere Entscheidung treffen, allgemein formuliert: $\mathcal{N}(\mathbf{w}|\mathbf{m}_0, \mathbf{S}_0)$. Ist das der Fall so erhalten wir letztendlich den Posterior:

$$p(\mathbf{w}|\mathbf{t}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N) \tag{154}$$

wobei gilt, dass:

$$\begin{aligned} \mathbf{m}_N &= \mathbf{S}_N(\mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\mathbf{\Phi}^T\mathbf{t}) \\ \mathbf{S}_N^{-1} &= \mathbf{S}_0^{-1} + \beta\mathbf{\Phi}^T\mathbf{\Phi} \end{aligned} \tag{155}$$

9.2 Predictive Distribution

In der Praxis sind wir eigentlich nicht unbedingt an dem tatsächlichen Wert von $\mathbf{w}$ interessiert, sondern wir wollen das t für neue $\mathbf{x}$-Werte vorhersagen. Dies können wir theoretisch auch machen, ohne den Gewichtsvektor selber zu berechnen. Stattdessen können wir über alle möglichen Gewichtsvektoren integrieren und die daraus resultierenden Wahrscheinlichkeiten, t vorherzusagen, zu mitteln:

$$p(t|\mathbf{t}) = \int p(t|\mathbf{w})p(\mathbf{w}|\mathbf{t})d\mathbf{w} \tag{156}$$

Hierbei ist t der Target-Wert, dessen Wahrscheinlichkeit wir ermitteln wollen. Wir integrieren über alle möglichen Gewichtsvektoren und multiplizieren dabei jeweils die

Wahrscheinlichkeit, dass t bei jenem Gewichtsvektor vorhergesagt wird mit der Wahrscheinlichkeit, dass $\mathbf{w}$ überhaupt als Gewichtsvektor gewählt wird, wenn wir den gegebenen $\mathbf{t}$-Vektor fürs Trainingsset haben.

Das Ergebnis des Integrals lässt sich auch wieder als Normalverteilung beschreiben.

10 EM & Gaussian Mixture Models

10.1 K-Means

Kehren wir zurück zum k-means-Clustering. Die Kostenfunktion, die dort minimiert wird, wird oft geschrieben als:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||\mathbf{x}_n - \mu_k||^2 \tag{157}$$

r_{nk} ist 1, falls $k = \text{argmin}_j ||\mathbf{x}_n - \boldsymbol{\mu}_j||^2$, sonst 0. Wir addieren also alle quadratischen Distanzen zwischen den Trainingsvektoren und den zugewiesenen Clusterpunkten auf.

Fixieren wir die Zuweisungen r_{nk} so können wir eine Lösung in geschlossener Form herausfinden, indem wir den Gradienten nach μ_k gleich 0 setzen:

$$2 \sum_{n=1}^{N} r_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k) = 0 \tag{158}$$

Dies lässt sich leicht auf $\boldsymbol{\mu}_k$ umformen:

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk} \mathbf{x}_n}{\sum_n r_{nk}} \tag{159}$$

Sprich: Es wird einfach der Mittelwert der dem Cluster zugewiesenen Vektoren gebildet.

K-Means hat nun zwei Phasen, die immer wiederholt werden. Zuerst kommt der E-Step: In diesem wird der Fehler J minimiert, in dem die r_{nk} angepasst werden. Sprich: Die Trainingsdaten werden den jeweils nähersten Clusterpunkten zugeordnet. Die zweite Phase nennt sich M-Step: In diesem wird der Fehler J minimiert, in dem die $\boldsymbol{\mu}_k$ angepasst werden. Sprich, es werden die Clusterpunkte aktualisiert, in dem die Mittelwerte der zugeordneten Punkte berechnet werden.

Das ist einfach eine andere Art, die Funktionsweise von k-Means zu berechnen.

10.2 Gaussian Mixture Models

Ein Gaussian Mixture Model ist eine Dichtefunktion, die aus mehreren Gauss-Funktionen zusammengesetzt ist:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{160}$$

Die π_k liegen im Intervall $[0, 1]$ und ihre Summe ist 1.

Wir gehen nun davon aus, dass wir K Normalverteilungen übereinander haben und wir wissen wollen, aus welcher der Verteilungen unsere Beobachtung kam.

Der Vektor $\mathbf{z}$ enthält K Komponenten, wobei $K-1$ 0 sind und eine ist 1. Der Index der 1er-Komponente gibt an, zu welcher Normalverteilung der Wert $\mathbf{x}$ gehört.

Die Dichteverteilung von $\mathbf{x}$ ist also gleich der Normalverteilung, die durch den $\mathbf{z}$-Vektor beschrieben wird:

$$p(\mathbf{x}|z_k = 1) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{161}$$

Wie finden wir nun heraus, wie hoch die Wahrscheinlichkeit ist, dass ein gegebener Datenpunkt $\mathbf{x}$ zur Verteilung k gehört? Dies lässt sich mit dem Bayes-Theorem realisieren:

$$p(z_k = 1|\mathbf{x}) = \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(\mathbf{x}|z_j = 1)} = \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} =: \gamma(z_k) \tag{162}$$

Wollen wir ein GMM nun auf Daten trainieren, also die nötigen $\pi, \boldsymbol{\mu}, \boldsymbol{\Sigma}$-Werte herausfinden, so lässt sich dies wieder mit einem Maximum Likelihood Ansatz realisieren.

Die Likelihood lautet in diesem Fall:

$$p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{n=1}^{N} \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{y}_k, \boldsymbol{\Sigma}_k) \right\} \tag{163}$$

Mittels Bilden der Log-Likelihood, Ableitun und Null-Setzen lassen sich so wieder Werte für die Parameter in geschlossener Form finden (siehe Folien). Alle Parameter hängen von den aktuellen Werten für $\gamma(z_{nk})$ ab.

Der Expectation-Maximization-Algorithmus für GMMs schaut also wie folgt aus:

1. Initialisiere die $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$ und π_k und evaluiere den Initialwert der Log-Likelihood.
2. E-Step: Evaluiere die $\gamma(z_{nk})$, also die Wahrscheinlichkeit, dass $z_k = 1$, gegeben $\mathbf{x}_n$.
3. M-Step: Die Parameter anhand der neuen γ-Werte neu ermitteln.
4. Log-Likelihood evaluieren und auf Konvergenz prüfen. Falls das Konvergenzkriterium noch nicht erfüllt ist, wiederhole ab Schritt 2.

10.3 General EM

Wir haben jetzt das Expectation-Maximization-Prinzip einmal bei K-Means und einmal bei GMMs betrachtet. Die Vorgehensweise lässt sich jedoch auch allgemein formulieren.

Wir haben ein Modell mit Parameter $\boldsymbol{\theta}$, Beobachtungen $\mathbf{X}$ und Zwischenvariablen $\mathbf{Z}$ (z.B. in K-Means die Zuweisungen r oder in GMMs die γ-Werte). Das Ziel ist die Likelihood $p(\mathbf{X}|\boldsymbol{\theta})$ zu maximieren.

1. Initiiere $\boldsymbol{\theta}$

2. E-Step: Evaluiere $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$
3. M-Step: Aktualisiere $\boldsymbol{\theta}$
4. Prüfe auf Konvergenz. Falls noch nicht vorhanden, wiederhole ab Schritt 2.