Institut für
Technische Informatik und
Kommunikationsnetze

Personenerkennung anhand des Sprachsignals

Andreas Ochsner

Semesterarbeit SA-2003.03

Wintersemester 2002/2003

Institut für Technische Informatik
und Kommunukationsnetze

Betreuer: Dr. B. Pfister und R. Beutler

Verantwortlicher: Prof. Dr. L. Thiele

Inhaltsverzeichnis

Zusammenfassung

Ein automatisches Sprecherverifikationverfahren soll in der Lage sein, mit grosser Zuverlässigkeit die behauptete Identität einer Person zu überprüfen. Das bisher am TIK entwickelte textabhängige Verfahren tut dies recht gut mittels Zeitnormalisation (dynamic time warping) und Messung der euklidischen cepstralen Distanz.

Da die Zeitnormalisation gut funktioniert, ist davon auszugehen, dass das Verfahren vorwiegend durch ein alternatives Distanzmass verbessert werden kann. Der Hypothese, dass ein geeignet trainiertes neuronales Netz ein besseres Entscheidungskriterium liefern kann, geht diese Arbeit nach. Zu diesem Zweck wird zunächst nach einem möglichst geeignetem Netz gesucht, dessen Performanz dann anhand des Fisher-Kriteriums mit dem bisherigen Verfahren verglichen wird.

1 Übersicht über die Sprechererkennung

Unter Sprechererkennung versteht man die Aufgabe, die Identität eines unbekannten Sprechers anhand seiner Stimme zu bestimmen (Sprecheridentifikation) oder zu überprüfen (Sprecherverifikation). Dies ist möglich, weil das Sprachsignal nicht nur eine Aussage (abstrakte Information im Sinne einer Phonemfolge), sondern auch Informationen über den Sprecher enthält. Das Sprachsignal kann somit als Produkt eines mehr oder weniger deterministischen Sprechprozesses betrachtet werden, welcher u.a. beeinflusst wird durch:

- die gemachte Aussage
- die Physiologie des Sprechapparates
- die Sprechgewohnheiten (Dialekt, Sprechrhythmus usw.)
- die Sprechsituation
- die momentane Stimmung
- den Übertragungsweg des Sprachsignals

Alle diese Komponenten schlagen sich in einer sehr komplexen und schwierig beschreibbaren Art und Weise im Sprachsignal nieder und lassen sich deshalb auch kaum separieren und zur Sprecherbestimmung nutzen.

Der obige Ansatz, das Sprachsignal als Ausgangssignal aus einem gesteuerten Sprechprozess zu betrachten, ist jedoch nützlich, um sich darüber Klarheit zu verschaffen, welche Steuerkomponenten für eine Sprecherverifikation brauchbar sind. Sicher helfen stochastische Elemente, wie z.B. die momentane Stimmung oder der Übertragungsweg des Signals wenig und in den meisten Fällen stören sie sogar den Erkennungsprozess. Dagegen sind z.B. aus physiologischen Eigenschaften oder längerfristig konstanten Sprechgewohnheiten ins Sprachsignal einfliessende Merkmale wichtig für die Sprecherunterscheidung.

Die bekannten Ansätze zur Lösung des Sprecherverifikationsproblems sind sehr zahlreich und lassen sich grob in zwei Klassen unterteilen, statistische (textunabhängige) und dynamische (textabhängige) Verfahren.

1.1 Statistische Methoden

Die statistischen Methoden extrahieren Merkmale aus dem Sprachsignal und vergleichen diese mit den über einen Sprecher abgespeicherten Referenzdaten. Das Besondere dieser Merkmale ist, dass sie zeitlich unabhängig sind, also eine statistische Aussage über das Sprachsignal liefern. Als Merkmale kommen etwa in Frage:

- Langzeitspektrum
- Varianz des Kurzzeitspektrums

- Mittelwert und Varianz der Sprachgrundfrequenz
- Varianz der Signalintensität
- akkumulierter Fehler bei der Codierung mit Vektorquantisierung

In der Regel werden mehrere Merkmale gleichzeitig verwendet, um die Entscheidungssicherheit zu erhöhen.

Die statistischen Verfahren sind textunabhängig, d.h. es ist möglich, zur Gewinnung der Referenzdaten und bei der Verifikation Sprachsignale unterschiedlichen Inhaltes zu verwenden. Damit aber der sprachliche Inhalt überhaupt ausgemittelt und eine akzeptable Zuverlässigkeit erreicht werden kann, werden lange Signale benötigt (5-10 Sekunden für die Verifikation, über eine Minute zur Bildung der Referenzdaten). Idealerweise sollten die Sprachproben auch lautlich möglichst ausgewogen sein.

1.2 Dynamische Methoden

Bei dynamischen Verfahren findet die Klassifizierung aufgrund der sprecherspezifischen Unterschiede einander entsprechender phonetischer Ereignisse der Referenzprobe einerseits und der unbekannten Sprachprobe anderseits statt. Damit diese textbezogenen Unterschiede überhaupt erfasst werden können, muss die textspezifische, zeitabhängige Information in der Darstellung der Sprachsignale erhalten bleiben.

Die erfolgreichsten Lösungsansätze gehen von der Darstellung des zeitlichen Verlaufes der spektralen Eigenschaften, typischerweise in Form von Konturen von LPC-Vektoren oder von spektralen Leistungsdichtevektoren aus. Seltener werden auch Stimmbandgrundfrezuenzkonturen oder Signalleistungskonturen eingesetzt.

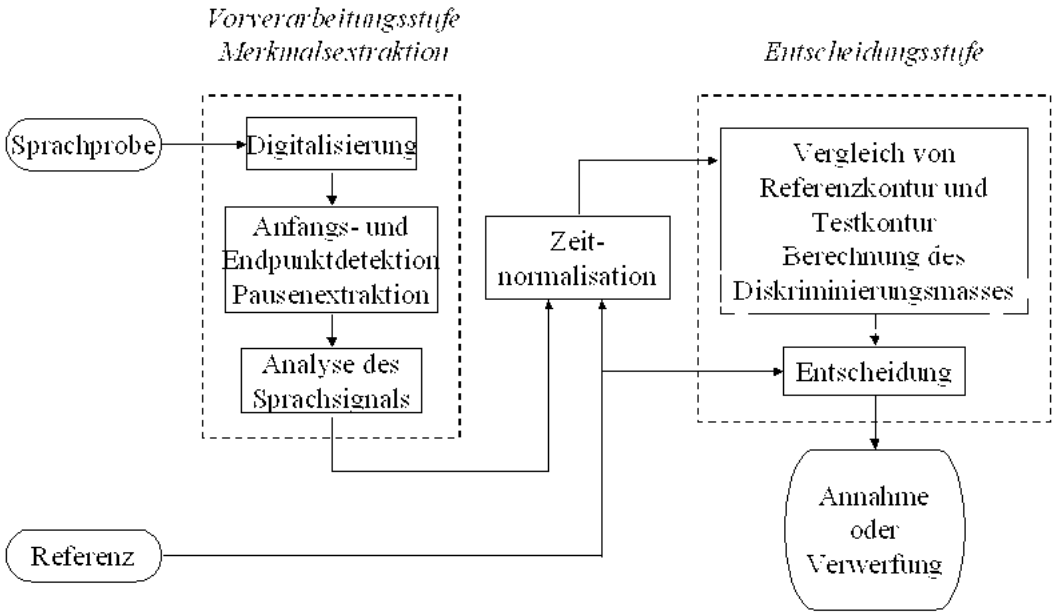
Die so gewonnenen Konturen, Test- und Referenzkontur, werden dann zeitnormalisiert, d.h. sich entsprechende phonetische Ereignisse werden lokalisiert und einander gegenüber gestellt. In einem ersten primitiven Lösungsversuch wurde die Normalisation einfach durch lineare Kompression respektive Streckung der Zeitachsen der Konturen realisiert (linear time warping), was verständlicherweise zu nicht sehr guten Ergebnissen geführt hat. Erst später wurden die sogenannten DTW-Algorithmen (dynamic time warping) eingeführt, welche eine nichtlineare Verzerrung der Zeitachsen erlauben. Damit wurde es möglich, die Zeitnormalisation mit ausreichender Genauigkeit durchzuführen, was entscheidend für den Erfolg der dynamischen Verfahren ist.

Der Vorteil der Textabhängigkeit ist, dass eine zuverlässige Klassifizierung bereits mit sehr kurzen Signalen (typischerweise 2 bis 3 Sekunden) möglich ist. Dies wird erkauft durch einen erhöhten Rechenaufwand für die Zeitnormalisation.

Figur 1 zeigt ein Blockschaltbild für textabhängige Verfahren gezeigt.

1.3 Das am TIK entwickelte Verfahren

Das bisher am TIK entwickelte Verfahren beruht auf einer Zeitnormalisation und einer Distanzmessung, welche beide das euklidische cepstrale Distanzmass (EC-Distanzmass)



Figur 1: *Blockschaltbild textabhängiger Verfahren*

benutzen. Die Cepstren können via LPC-Methode oder via Fouriertransformation ermittelt werden.

Da das Verfahren hauptsächlich auf Sprachsignale angewandt wird, die über das Telefon übertragen worden sind, muss die Übertragunscharakteristik der verschiedenen Leitungswege berücksichtigt werden. Ein elegantes Vorgehen zur Kompensation dieser spektralen Verformung ist die Subtraktion des cepstralen Mittelwert-Vektors.

Zudem weisen die über das Telefon übertragenen Sprachsignale je nach Übertragungsweg unterschiedliche Bandbreiten auf. Garantiert übertragen wird nur zwischen 300 Hz und 3400 Hz, weshalb die Signale vor der Berechnung der Cepstren durch ein Bandbegrenzunsfilter zurechtgeschnitten werden.

Das am TIK entwickelte Sprecheridentifikationsverfahren hat viele Optionen und Parameter, die anhand einer möglichst grossen Menge von Paaren von Sprachproben optimiert werden müssen. Ist von jedem Paar bekannt, ob beide Signale von derselben Person gesprochen worden sind oder nicht, dann kann aufgrund der Verteilung der Distanzen bei gleichem Sprecher (Eigendistanzen d_s) und der Verteilung der Distanzen bei verschiedenen Sprechern (Kreuzdistanzen d_c) die Güte des Verfahrens beurteilt werden. Als Mass für die Diskriminierung wird das Fisher-Kriterium eingesetzt:

$$F = \sqrt{\frac{(\overline{d_s} - \overline{d_c})^2}{\sigma_s^2 + \sigma_c^2}} \tag{1}$$

2 Problemstellung

Bei dem oben beschriebenen Verfahren wird sowohl für die Zeitnormalisierung als auch für die Messung der sprecherspezifischen Unterschiede das EC-Distanzmass verwendet, obwohl verschiedene Ziele verfolgt werden:

- Die Zeitnormalisation soll die Zeitskala der Sprachmuster so verzerren, dass gleiche Laute einander zugeordnet werden, soll also hinsichtlich Lautunterscheidung eine hohe Diskrimationsfähigkeit haben.
- Für die Srecherunterscheidung wäre aber ein Mass von Vorteil, welches so lautunabhängig wie möglich sprecherspezifische Unterschiede erfassen könnte.

Da die Zeitnormalisation mit dem EC-Mass gut funktioniert, soll nun für die Sprecherdiskriminierung ein anderes Mass gesucht werden. Wie die Experimente in [4] gezeigt haben, ist die Anwendung eines neuronalen Netzes ein viel versprechender Weg. Dabei geht es um die folgenden Probleme:

- Merkmalsextraktion: Es ist sicher nicht optimal, an die Eingänge des neuronalen Netzes direkt die Abtastwerte zu legen. Wird ein Signal auch nur um einen Wert verschoben, ändern sich alle Eingänge, obwohl das Signal praktisch noch dasselbe ist. Die Klassifizierung sollte also möglichst invariant unter kleinen zeitlichen Transformationen sein. Das Problem invarianter Klassifikation kann grundsätzlich auf drei Arten angegangen werden:

  1. Trainieren durch Beispiele. Das Trainingsset sollte eine genügende Anzahl Beispiele der möglichen Transformationen beinhalten.
  2. In einer Vorverarbeitung werden Merkmale extrahiert, welche selbst invariant gegenüber den Transformationen sind.
  3. Die Invarianzen werden in die Netzwerkstruktur selber eingebaut. Zum Beispiel *shared weights*.

  Der Nachteil der ersten Methode ist die Ineffizienz, da sie ein erheblich vergrössertes Trainingsset und ein komplexeres Netz benötigt. Auch wird ein solchermassen trainiertes Netz transformierte Daten nur approximativ erkennen können und ist beschränkt auf die Transformationen, welche in den Trainingsdaten berücksichtigt wurden. Die dritte Methode wurde in dieser Arbeit nicht verfolgt. Für die zweite Methode sollen also Merkmale extrahiert werden, welche bezüglich kleiner zeitlicher Verschiebungen unempfindlich sind. Eine durch Literatur [4] nahegelegte Wahl ist eine Anzahl Autokorrelations-Koeffizienten.

- Netztopologie: Das Netz hat grundsätzlich die Aufgabe, einen Inputvektor, welcher aus zwei zu vergleichenden Sprachproben ermittelt wird, einer von zwei Klassen zuzuordnen:

  1. Proben wurden vom selben Sprecher gesprochen

2. Proben stammen von verschiedenen Sprechern

Da ein 3-Schicht-Perzeptron (zumindest theoretisch) in der Lage ist, jedes Klassifizierungsproblem zu lösen (siehe Figure 3), wurde diese Topologie gewählt. Für jede Klasse wurde ein Ausgang vorgesehen, der entsprechend "hoch" ist, wenn das Netz den Input dieser Klasse zuordnet.

- Referenzbildung: Da die Sprachproben eines Sprechers i.a. eine beträchtliche Variabilität aufweisen, ist es sinnvoll, aus mehreren Proben ein Referenzmuster zu ermitteln. Dieser Ansatz wurde aber hier nicht verfolgt.

3 Grundlagen und Materialien

3.1 Theorie der neuronalen Netze

Eine erste Teilaufgabe war die Einarbeitung in die Theorie der neuronalen Netze. Eine kurze Zusammenfassung davon soll der nächste Abschnitt zeigen.

Wie der Name schon andeutet, basieren die künstlichen neuronalen Netze auf unserem heutigen Verständnis der Prozesse in biologischem Nervengewebe. Dabei werden Nervenzellen und ihre Verbindungen in einfache mathematische Modelle abstrahiert.

Das Grundmodell eines Neurons stützt sich im Wesentlichen auf die Vereinfachungen von McCulloch und Pitts aus dem Jahre 1943, die ein Neuron als eine Art Addierer mit Schwellwert betrachteten. Die Verbindungen (Synapsen) eines Neurons nehmen Aktivierungen x_i mit bestimmten Stärken w_i von anderen Neuronen auf, summieren diese und lassen dann am Ausgang y (Axon) des Neurons eine Aktivität entstehen, sofern die Summe einen Schwellwert s überschritten hat.

Ein wenig verallgemeinert lässt sich das durch die folgende Gleichung beschreiben:

$$y = f(\sum_{i=1}^{N} w_i x_i - s) \tag{2}$$

wobei f die Aktivierungsfunktion genannt wird. Im einfachsten Fall ist dies die Heaviside-Funktion, da diese jedoch nicht differenzierbar ist (was für gewisse Lernalgorithmen wichtig ist), werden meistens weichere Übergänge modelliert, z.B. mittels Tanhyp oder Arctan.

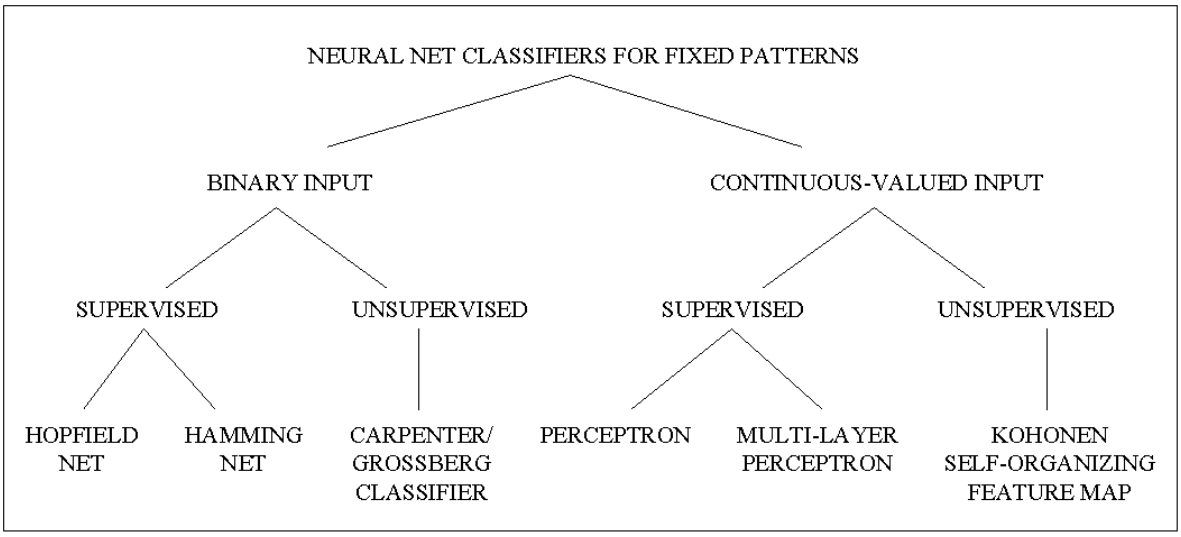
Neuronale Netze werden spezifiziert durch die Netztopologie, die Charakteristik der einzelnen Neuronen und die Trainings- oder Lernregeln. Diese Regeln bestimmen die anfänglichen Gewichte w_i und wie diese adaptiert werden sollen, während das Netz optimiert wird.

Die Vorteile von Neuronalen Netzen liegen nicht nur in der Geschwindigkeit, welche durch die massive Parallelität erreicht werden kann, sondern auch in einer grösseren Robustheit und Fehlertoleranz. Im Gegensatz zu sequentiellen von Neumann Computern gibt es hier viel mehr rechnende Einheiten mit primär lokalen Verbindungen. Der Ausfall von einigen wenigen stört die allgemeine Performanz meist nicht signifikant. Ein weiterer Vorteil ist die Fähigkeit, sich anzupassen und kontinuierlich zu lernen.

3.1.1 Eine kleine Taxonomie

In Figur 2 werden sechs wichtige Neuronale Netze gezeigt, welche für die Klassifizierung von statischen Mustern verwendet werden können. Zunächst wird unterschieden zwischen binärem und kontinuierlichem Input. Beim Trainieren von Netzen kann zwischen überwachtem und nichtüberwachtem Lernen unterschieden werden. Überwachung heisst dabei, dass dem Netz während der Trainingsphase die korrekte Klassifizierung vorgegeben wird. Der Fehler, den das Netz zunächst macht, wird für das Anpassen der Gewichte benutzt

und wird hoffentlich immer kleiner. Nicht überwachte Netze werden verwendet zur Vektorquantifizierung und zum Clustern von Daten. Diese Netze finden eventuell in den Daten vorhandene Strukturen selbständig.



Figur 2: *Taxonomie wichtiger Klassifizierungsnetze*

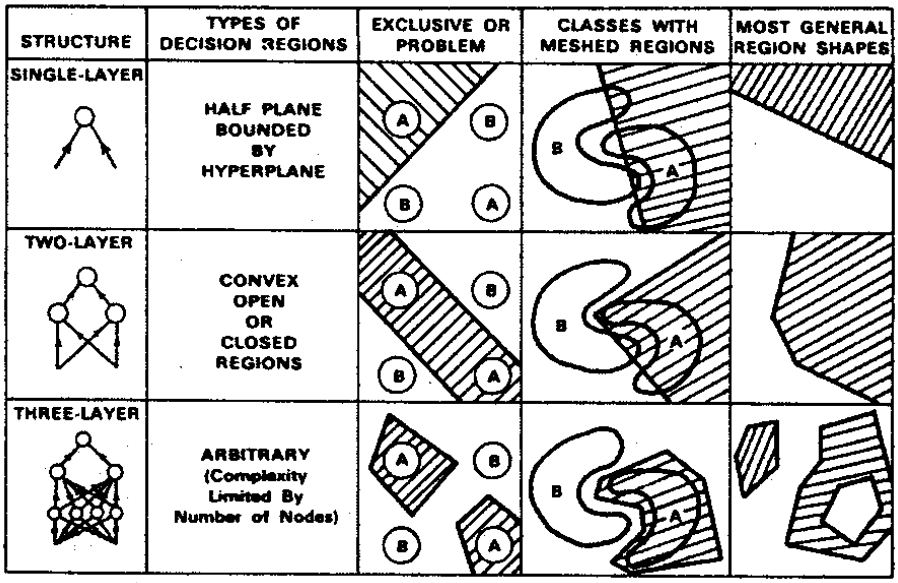
3.1.2 Multi-Layer Perzeptron

Das in dieser Arbeit benutzte Modell ist das mehrschichtige Perzeptron. Dies ist ein feedforward Netz mit einer oder mehreren Schichten von Knoten zwischen Input und Output. Erst in den 80er Jahren wurden effiziente Trainingsalgorithmen entwickelt, was diese Netze in den Bereich des Interesses rückte. Obwohl nicht bewiesen werden kann (anders als bei einschichtigen Perzeptrons), dass diese Algorithmen konvergieren, wurden sie seither bei vielen interessanten Problemen erfolgreich eingesetzt.

Die Klassifizierungsgebiete, welche die verschiedenen Perzeptrons (mit einer Heaviside-Aktivierungsfunktion) auflösen können, sind in Figur 3 gezeigt. Die zweite Spalte beschreibt die Typen von Entscheidungsregionen, die von den verschiedenen Netzen geformt werden können. Die beiden nächsten Spalten zeigen Beispiele, wie zwei gegebene Probleme gelöst werden können. In der letzen Spalte schliesslich sind die allgemeinsten Entscheidungsgrenzen illustriert, die gebildet werden können.

Es lässt sich beweisen, dass nicht mehr als drei Schichten nötig sind, um beliebig komplexe Regionen zu klassifizieren. Es ist aber nicht ausgeschlossen, dass mehr Schichten, je nach Problemstellung, nützlich sein können.

Trainiert werden diese Netze mit dem Back-Propagation Algorithmus, welcher in Figur 4 gezeigt ist.



Figur 3: *Entscheidungsgebiete*

The back-propagation training algorithm is an iterative gradient algorithm designed to minimize the mean square error between the actual output of a multilayer feed-forward perceptron and the desired output. It requires continuous differentiable non-linearities. The following assumes a sigmoid logistic non-linearity is used where the function $f(\alpha)$ in Fig. 1 is

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

Step 1. Initialize Weights and Offsets

Set all weights and node offsets to small random values.

Step 2. Present Input and Desired Outputs

Present a continuous valued input vector x_0, $x_1, \ldots x_{N-1}$ and specify the desired outputs d_0, $d_1, \ldots d_{M-1}$. If the net is used as a classifier then all desired outputs are typically set to zero except for that corresponding to the class the input is from. That desired output is 1. The input could be new on each trial or samples from a training set could be presented cyclically until weights stabilize.

Step 3. Calculate Actual Outputs

Use the sigmoid nonlinearity from above and formulas as in Fig. 15 to calculate outputs y_0, $y_1 \ldots y_{M-1}$.

Step 4. Adapt Weights

Use a recursive algorithm starting at the output nodes and working back to the first hidden layer. Adjust weights by

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_j x'_i$$

In this equation $w_{ij}(t)$ is the weight from hidden node i or from an input to node j at time t, x'_i is either the output of node i or is an input, η is a gain term, and δ_j is an error term for node j. If node j is an output node, then

$$\delta_j = y_j(1 - y_j)(d_j - y_j),$$

where d_j is the desired output of node j and y_j is the actual output.

If node j is an internal hidden node, then

$$\delta_j = x'_j(1 - x'_j)\sum_k \delta_k w_{jk},$$

where k is over all nodes in the layers above node j. Internal node thresholds are adapted in a similar manner by assuming they are connection weights on links from auxiliary constant-valued inputs. Convergence is sometimes faster if a momentum term is added and weight changes are smoothed by

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_j x'_i + \alpha(w_{ij}(t) - w_{ij}(t-1)),$$

where $0 < \alpha < 1$.

Step 5. Repeat by Going to Step 2

Figur 4: *Der Back-Propagation Trainings Algorithmus*

3.2 Materialien

3.2.1 Sprachproben

An Material standen mir 344 mit 8 kHz gesampelte Sprachproben zur Verfügung. Jede Probe enthielt ungefähr 20 Worte und war mit einem Labelfile versehen, welches neben andern Informationen für jedes Wort die Anfangsposition und die Länge enthielt.

3.2.2 Experimentiersystem

In Form eines Matlab-Programmes war ein Experimentiersystem vorhanden, welches das am TIK entwickelte Verfahren realisierte. Damit war es möglich, aufgrund der cepstralen Distanz die Verteilung der Eigen- und Kreuzdistanzen zu ermitteln und damit das Verfahren zu beurteilen. Mit zahlreichen Optionen und Parametern konnte man sowohl die Auswahl der Sprecher wie auch Berechnungsmethoden, den Detailgrad der Ausgabe und vieles mehr bestimmen.

3.2.3 NICO

Die Beschreibung des NICO Toolkits überlasse ich dem Autor gleich selber [6]:

> "The NICO Toolkit is an artificial neural network toolkit designed and optimized for speech technology applications. It is easy to construct neural networks with both recurrent connections and/or time-delay windows to capture temporal features. The network topology is very flexible – any number of layers is allowed, and layers can be arbitrarily connected. Powerful tools for sparse connectivity are also included. Tools for extracting input-features from the speech signal are also part of the toolkit, as well as tools for computing target values from many common phonetic label-file formats."

4 Arbeitskonzept

Das Ziel meiner Arbeit soll die Bestätigung der folgenden Hypothese sein: mittels eines geeignet trainierten künstlichen neuronalen Netzes lässt sich ein besseres Mass für den Abstand zwischen Sprachproben finden als das EC-Distanzmass.

Nach einer ersten Einarbeitung in die verschiedenen Tools und Techniken entschied ich mich für den folgenden grundsätzlichen Versuchsaufbau:

- Zur Merkmalsextraktion soll mir das bestehende Matlab-System dienen, welches mir zu ausgewählten Sprecherpaaren die Zeitnormalisation der einzelnen Worte durchführt. Aus den so ermittelten sich jeweils möglichst gut entsprechenden Analysefenstern werden dann eine Anzahl Autokorrelations-Koeffizienten berechnet, welche in eine Datei geschrieben werden. Diese soll der Input für ein neuronales Netz werden. Gleichzeitig wird in einer anderen Datei die Information festgehalten, ob das nun die Koeffizienten des gleichen Sprechers oder von zwei verschiedenen Sprechern seien. Dies soll der gewünschte Output sein, der für das Training oder die Beurteilung der Generalisierungsfähigkeit benötigt wird.

- Als nächstes werden eine Anzahl Netze konstruiert und mit Trainingsvektoren gefüttert. Nach jeder Trainingsepoche wird der Generalisierungsfehler, also der Fehler, den die Netze auf einer Menge von Validierungsvektoren (Inputvektoren, die im Trainingsset nicht enthalten sind), festgestellt und verglichen. Es soll so ein möglichst geeignetes Netz für diese Klassifizierungsaufgabe gefunden werden.

- In einem grösseren Experiment sollen dann die gewonnenen Erkenntnisse angewendet werden, um ein Netz zu trainieren, welches dann mit dem ursprünglichen Verfahren verglichen werden kann. Dieser Vergleich geschieht anhand des Fisher-Kriteriums (1).

5 Vorversuche

> "In most situations, there is no way to determine the best number of hidden units without training several networks and estimating the generalization error on each." [9]

5.1 Proof Of Concept

Nach diesen ermutigenden Worten wollte ich zuerst ein Gefühl für das Problem entwickeln und startete einen Versuch mit einem sehr kleinen Trainingset. Dies war als "Proof of Concept" gedacht (funktioniert die Merkmalsextraktion, können diese so gewonnenen Vektoren überhaupt gelernt werden) und war auf Anhieb ziemlich vielversprechend. Das Trainingsset wurde sehr schnell gelernt und es war sogar eine Verallgemeinerungsfähigkeit auf ein paar wenige andere Daten feststellbar. Das Ganze war informell gedacht und wurde nicht dokumentiert.

Als nächstes sollte untersucht werden, wie sensibel der ganze Prozess auf einzelne Parameter reagiert, insbesondere auf Lernparameter und die Grösse des Netzes. Da einzelne Versuche mit zunehmender Grösse des Netzes und zunehmender Menge von Inputvektoren sehr schnell sehr viel Zeit beanspruchten, dachte ich zunächst, mit wenigen Daten vielleicht eine Systematik erkennen zu können, die sich möglicherweise auf grössere Probleme verallgemeinern liessen. Ich begnügte mich also mit 1000 Trainings- und ebensovielen Testvektoren und trainierte die Netze 200 Epochen lang. Dabei ging ich bei den meisten Parametern von den Default-Werten im NICO-Toolkit aus.

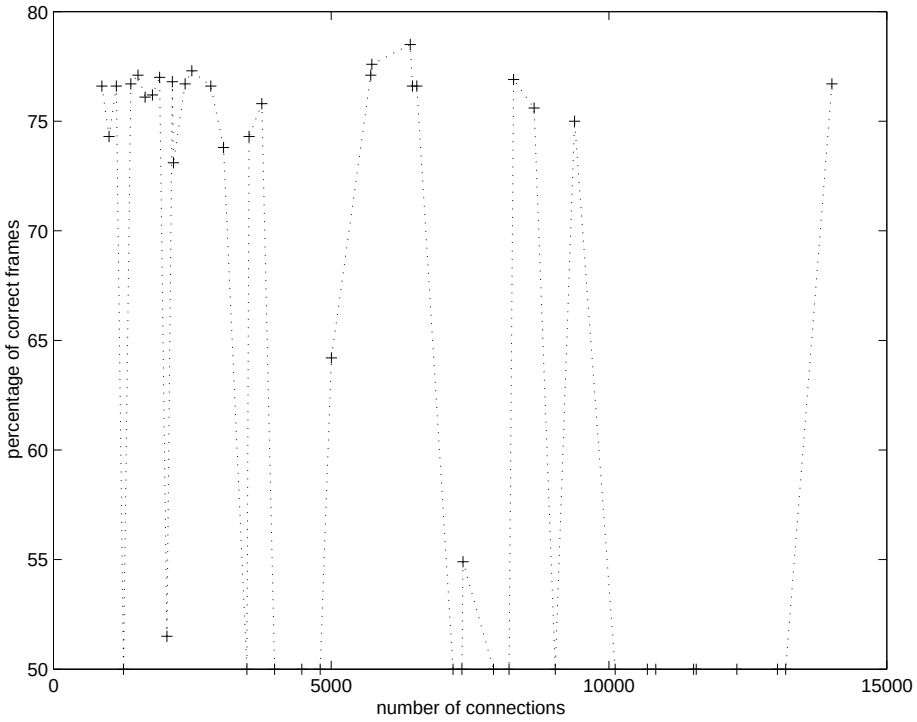
5.2 Netzgrösse

Nach den jeweils 200 Epochen des Trainings benutze ich für das Testen der trainierten Netze die NICO-Funktion *CResult*, welche auf dem Validierungsset eine Konfusionsmatrix berechnet und die Prozentzahl der korrekt erkannten Inputvektoren trug ich in der Figur 5 gegen die Anzahl Verbindungen auf. Ein systematisches Verhalten war nicht zu erkennen.

5.3 Untersuchung des Lernfaktors

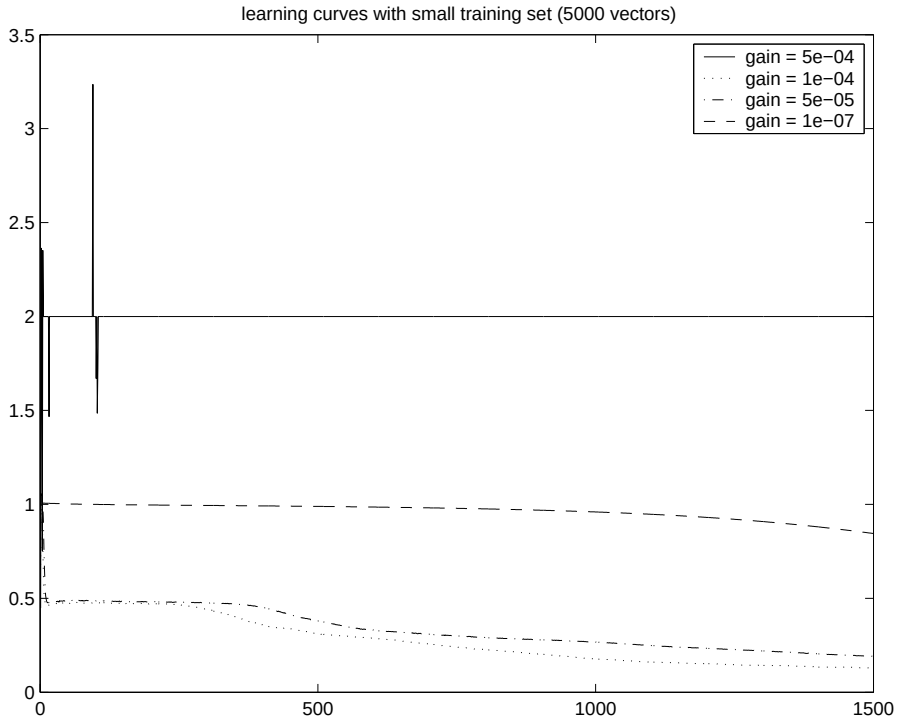
Als für das Training entscheidend erwies sich jedoch der Lernfaktor, den ich in der Folge als erstes genauer unter die Lupe nehmen wollte.

Salopp formuliert, bestimmt der Lernfaktor, mit welcher Schrittgrösse wir durch die Fehlerlandschaft schreiten auf der Suche nach einem Minimum. Sind diese Schritte zu gross, werden wir enge Täler in der Landschaft einfach überschreiten und eventuell gar nicht zu einem Minimum hinabsteigen können. Sind sie zu klein, kommen wir fast nicht vom Fleck und wir werden lange (zu lange vielleicht) ein Netz trainieren müssen, bis der Fehler, den es macht, ein minimaler wird. Es zeigte sich, dass ein Faktor von 1e-04 in diesem Fall das beste Resultat lieferte. Diesen Faktor behielt ich nun bei, auch wenn das natürlich eine etwas grobe Verallgemeinerung darstellt. Er könnte ja (und wird



Figur 5: *Generalisierungsfähigkeit verschiedener Netze*

es wahrscheinlich auch) von der Menge der Trainingsvektoren, speziellen Eigenschaften derselben, Komplexität des verwendeten Netzes, der zufälligen Wahl der Anfangsgewichte, der Häufigkeit der Gewichtsanpassung und vielen anderen Dingen abhängen. Der Grund für diese Entscheidung war, die Anzahl der freien Parameter meiner Versuche ein wenig einzuschränken, die Rechtfertigung war die Vermutung, dass sich die Fehlerlandschaften der verschiedenen Versuche zumindest ähnlich sind.



Figur 6: *Lernkurven zu verschiedenen Lernfaktoren*

6 Versuchsorganisation

Da insbesondere die Merkmalsextraktion sehr rechenintensiv war und mir schon jetzt die Zeit davonlief, unterteilte ich jetzt den Versuchsablauf in 4 Teilprozesse:

- Merkmalsextraktion
- Zusammenstellen von Trainings-, Validierungs- und Testdateien
- Bauen und Trainieren
- Analyse der trainierten Netze

Die einzelnen Prozess lassen sich unabhängig voneinander bearbeiten, wobei natürlich zuerst mindestens einmal Merkmale extrahiert worden sein müssen, bevor die Vektoren zusammengestellt werden können, was wiederum vor dem Training zu geschehen hat, u.s.w. Es ist aber so z.B. nicht nötig, jedesmal mit grossem Aufwand neue Merkmale zu extrahieren, wenn man eine andere Trainingsvektormenge zusammenstellen will. Die einzelnen Prozesse werden im Folgenden kurz beschrieben:

6.1 Merkmalsextraktion

Die Merkmalsextraktion wurde dahingehend verändert, dass nicht mehr direkt eine Inputdatei geschrieben wird, sondern die Autokorrelations-Koeffizienten in einem Verzeichnissystem abgelegt wurden, unterteilt nach Eigen-/Kreuzdistanz, Sprecherpaar und Segment. Mit Segment wird hier ein einzelnes Wort bezeichnet. (Am Ende erwies sich diese Unterteilung als zu fein, da ich gar nicht mehr dazu kam, verschiedene Trainingsdateien zusammen zu stellen und zu testen, um vielleicht unterschiedliche Performanz bei verschiedenen Wörtern oder Ähnliches zu untersuchen.)

6.2 Zusammenstellen von Dateien

Nun war es möglich, sehr schnell und flexibel mit einigen Unix-Befehlen neue Trainings- und Testdateien zusammenzustellen.

6.3 Bauen und Trainieren

Das Bauen und Trainieren von Netzen geschieht mittels Skripten, welche auch in Logfiles die Testergebnisse aufzeichnen.

6.4 Analyse der trainierten Netze

Die so generierten Logfiles wurden dann mit Hilfe von Matlab analysiert und graphisch dargestellt.

7 Test: Verschiedene Sprechergruppen

Um eine repräsentativere Aussage machen zu können, konstruierte ich nun eine schwierigere Situation. Die Trainings- und Validierungsvektoren entnahm ich verschiedenen Sprechergruppen. Zudem wählte ich eine ziemlich grosse Anzahl Gewichte. Ich wollte so den Punkt des weiter unten besprochenen "overfittings" finden, den Punkt, wo das eigentlich zu grosse Netz die Trainingsdaten auswendig gelernt hat und immer schlechter generalisieren kann.

7.1 Setup

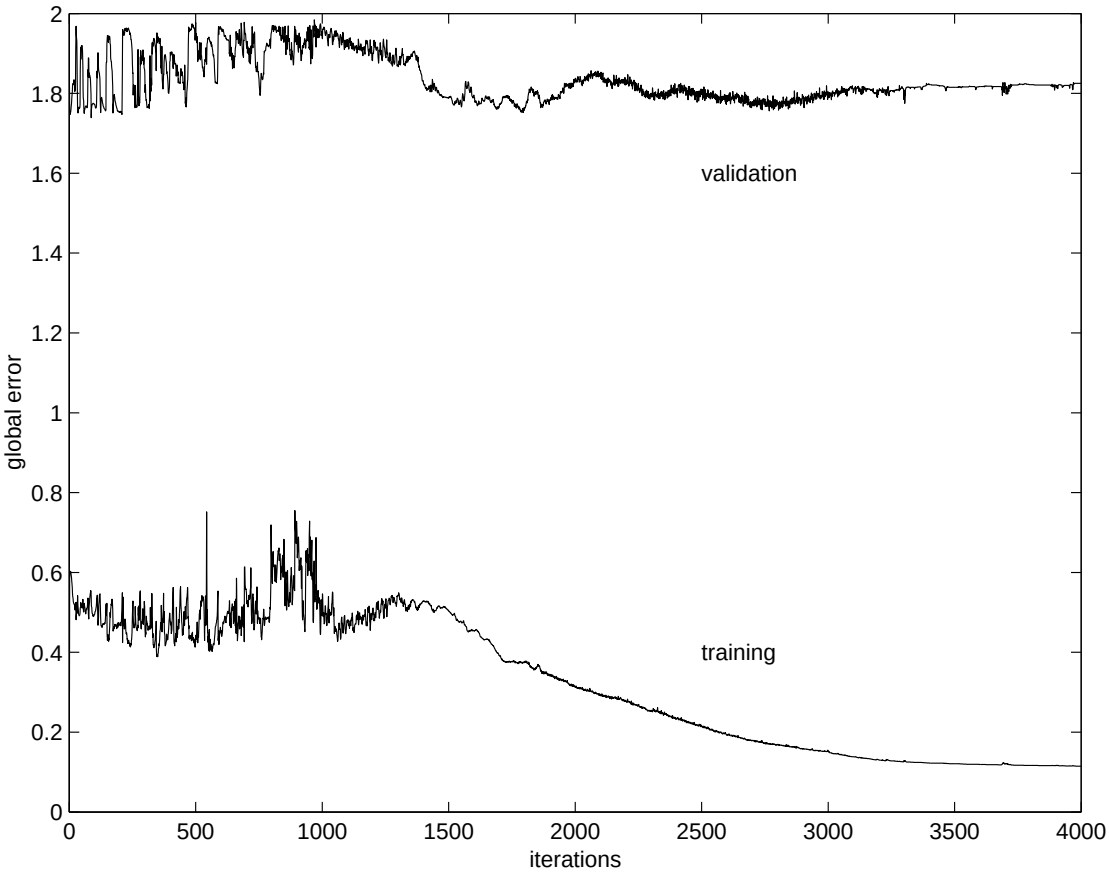
Trainingset	
Sprecher	[101 102 104 105]
Anzahl Vektoren	18000
Validationset	
Sprecher	[106 108 110 115 168]
Anzahl Vektoren	11000
Netzparameter	
Topologie	3-layer Perceptron
Anzahl Hidden Units in Layer 1	80
Anzahl Hidden Units in Layer 2	80
Anzahl Gewichte	9922
Trainingsparameter	
Update Methode	nach 15-30 Vektoren
Anzahl Epochen	4000
Lernfaktor	1e-04

7.2 Ergebnis

Die Figur 7 zeigt die Entwicklung des Fehlers auf dem Trainings- sowie auf dem Validierungsset. Wie man aus dem qualitativen Verlauf beider Kurven sofort sehen kann, hat das Netz zwar die Trainingsdaten einigermassen gelernt, hat aber praktisch keine Fortschritte bei der Generalisierung des Gelernten auf die Validierungsdaten gemacht. Ein "overfitting" war aber auch nicht erkennbar.

Der grosse quantitative Unterschied ist auf die Verwendung des Validierungsparameters zurückzuführen: während auf dem Trainingset der globale Fehler berechnet wurde, wurde für die Validierung ein Klassifikationsnetz angenommen und die Performanz so berechnet. Dabei ging aus dem NICO Manual [6] nicht klar hervor, was für ein Fehler denn per Default gerechnet wird. In weiteren Versuche wurde die Validierung dem Training angeglichen und eine spezifische Fehlerfunktion (mean square error) angegeben.

Der grosse absolute Wert des Validierungsfehlers warf aber einige Fragen an NICO auf, welche im nächsten Abschnitt untersucht werden sollen.



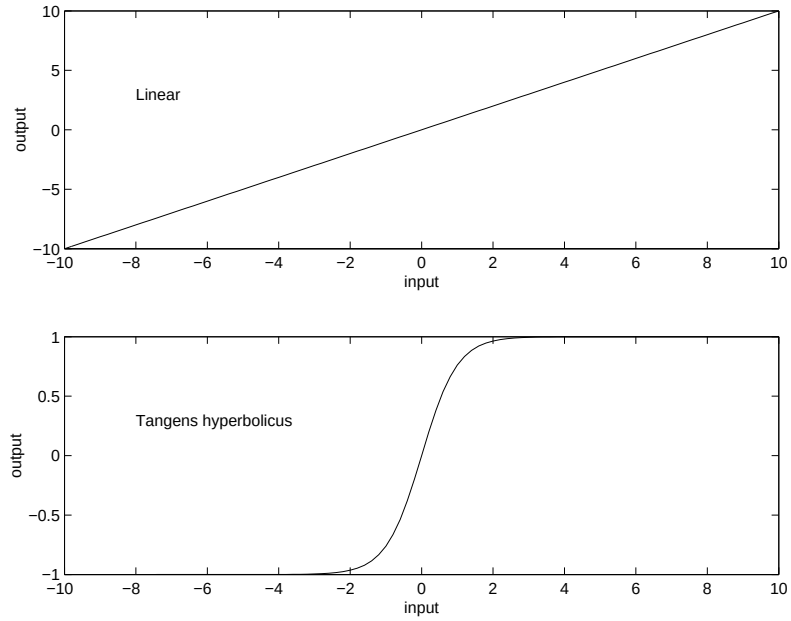
Figur 7: *Entwicklung des globalen Fehlers*

8 Was tut eigentlich NICO?

Um die genaue Funktionsweise der mit NICO gebauten Netze zu untersuchen, wurden eine Reihe Tests an sehr einfachen Netzen gemacht.

8.1 Aktivierungsfunktion

Zuerst überprüfte ich die Aktivierungsfunktion eines einzelnen Neurons. Das Ergebnis (Figur 8) entsprach den Erwartungen für den linearen Fall. Auch beim Tangens Hyperbolicus zeigte sich die korrekte Funktion, ich hatte aber irgendwie erwartet, der Wertebereich würde auf [0,1] normalisiert, was nicht der Fall war.



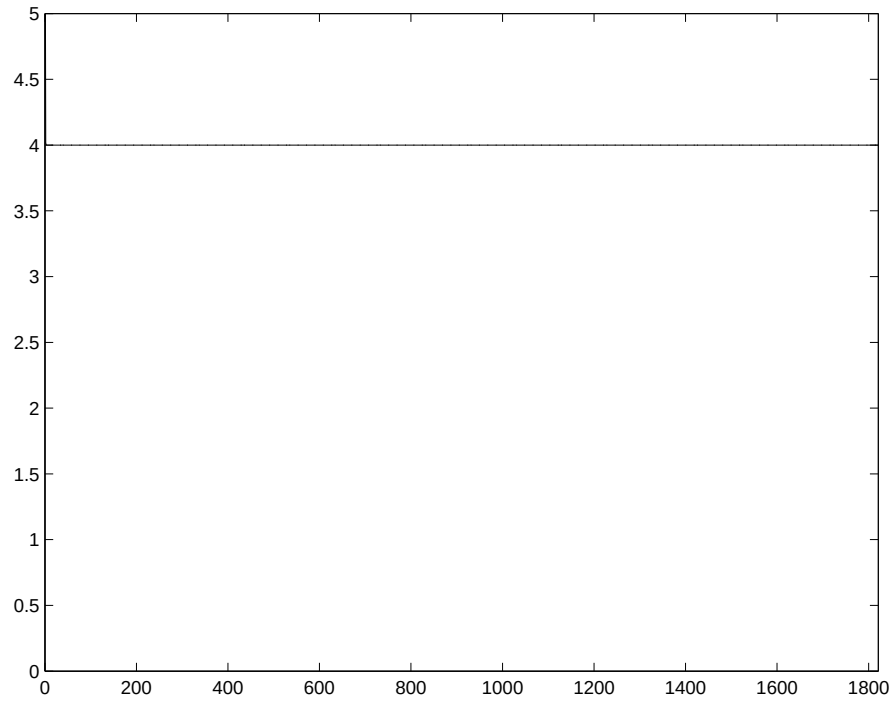
Figur 8: *Aktivierungsfunktionen eines Neurons*

Zu untersuchen war nun, ob das Anhängen eines Output-Streams diese Normalisierung zur Folge hatte, oder ob der gewünschte Output gar im Bereich [-1,1] formuliert werden müsste. Das Resultat war: ein Stream änderte gar nichts. Der Output war derselbe.

8.2 Gewünschter Output im Intervall [-1,1]?

Ich versuchte nun, das Sprechererkennungsproblem "symmetrisch" zu formulieren: 1 für Eigendistanz, -1 für Kreuzdistanz. In der Dokumentation von NICO konnte ich zwar wenig Anhaltspunkte dafür finden, ob das eine sinnvolle Idee war. Die dort beschriebenen Beispiele waren alle mit einem gewünschten Output in [0,1] beschrieben. Ich wollte es aber trotzdem ausprobieren. Mit deutlichem Ergebnis (Figur 9, dasselbe Experiment wie 7, mit dem anders formulierten gewünschten Output, das ich allerdings früher abbrach).

Auch ein überschaubareres Problem wie das XOR-Problem, das ich auch auf diese Weise formulierte, konnte so nicht mehr gelernt werden.



Figur 9: *Globaler Fehler bei symmetrischer Formulierung*

8.3 Fehlerrechnung

Ein weiterer Punkt, den ich klären wollte, war die Angabe des Fehlers in der Log-Datei. Es war mir nicht klar, wie derart grosse Werte zustande kamen.

Ich wählte verschiedene Fehlerfunktionen auf den Output-Neuronen eines sehr einfachen Netzes, das ich von Hand rechnen konnte und verglich die Resultate mit meinen Erwartungen. Diese stimmten nur im linearen Fall überein. Da ich aber nicht mehr zu viel Zeit verlieren wollte, und die anderen Experimente mittlerweile plausiblere Ergebnisse zeigten, ging ich dem nicht weiter nach.

8.4 Fazit dieser Untersuchung

Im Folgenden vertraute ich dann doch darauf, dass das NICO-Toolkit etwas vernünftiges mache. Um jedoch ein wenig mehr Kontrolle zu haben, setzte ich nun gewisse Parameter explizit. Die *hidden units* sollen die Tanh-Aktivierungsfunktion haben und die Fehlerfunktion der Output-Neuronen soll *mean square error* sein.

Ich schenkte dem absoluten Wert des globalen Fehlers, dessen Sinn ich nicht genau eruieren konnte, weniger Beachtung als dem qualitativen Verlauf der Lernkurven. Für

einen quantitativen Vergleich mit dem EC-Distanzmass spielt dieser Wert sowieso keine Rolle. Dafür werde ich ja dann das Fisher-Kriterium verwenden.

9 Test: Gleiche Sprechergruppe

Für die nächsten Versuche reduzierte ich die Schwierigkeit des zu lernenden Problems wieder. Test- sowie Validierungsvektoren stammen jetzt wieder von der gleichen Sprechergruppe, waren aber natürlich unterschiedlichen Sprachereignissen entnommen.

Trainingset	
Sprecher	[102 116 123 127]
Anzahl Vektoren	8400
Validationset	
Sprecher	[102 116 123 127]
Anzahl Vektoren	8400

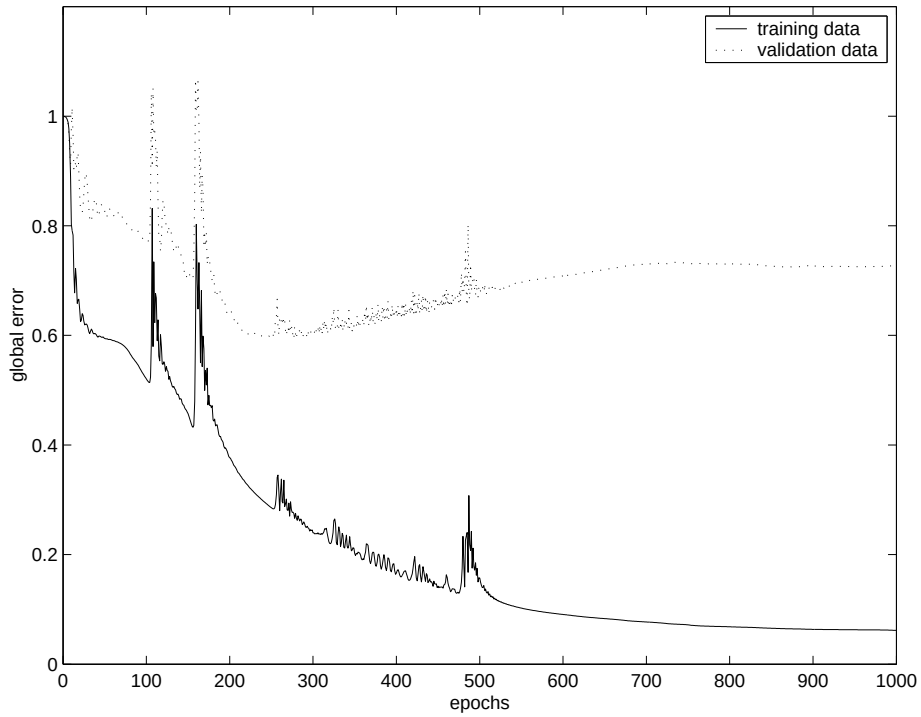
Auch die Grösse des Netzes reduzierte ich und versuchte es dieses Mal mit einem Anpassen der Gewichte nach einer ganzen Epoche.

Netzparameter	
Topologie	3-layer Perceptron
Anzahl Hidden Units in Layer 1	60
Anzahl Hidden Units in Layer 2	18
Anzahl Gewichte	3596
Fehlerfunktion output	mean square error
Trainingsparameter	
Update Methode	Epoche
Anzahl Epochen	2200
Lernfaktor	1e-04
Momentum	0.9

Figur 10 zeigt das Ergebnis. Endlich ein Verlauf, der irgendwie Sinn machte! Dieses Mal ist deutlich erkennbar, dass nach ca 230 Epochen der Generalisierungsfehler wieder zunimmt. Ein Zeichen für *overfitting*, was ich so früh eigentlich nicht erwartet hätte nach den vorherigen Versuchen. Das erreichte Fehlerminimum auf dem Validierungsset, welches ich dann im Folgenden mit verschiedenen Methoden zu verbessern suchte, war 0.5934 nach der 277. Epoche.

Zum Vergleich habe ich dasselbe Experiment noch mit einer anderen Update Methode durchgeführt. Dieses Mal wurde nach jedem Vektor die Gewichtsanpassung vorgenommen, wobei der Momentum Term, der bei so kurzen Änderungsintervallen wenig Sinn macht, auf 0 gesetzt wurde.

Trainingsparameter	
Update Methode	nach jedem Vektor
Anzahl Epochen	1600
Lernfaktor	1e-04
Momentum	0.0



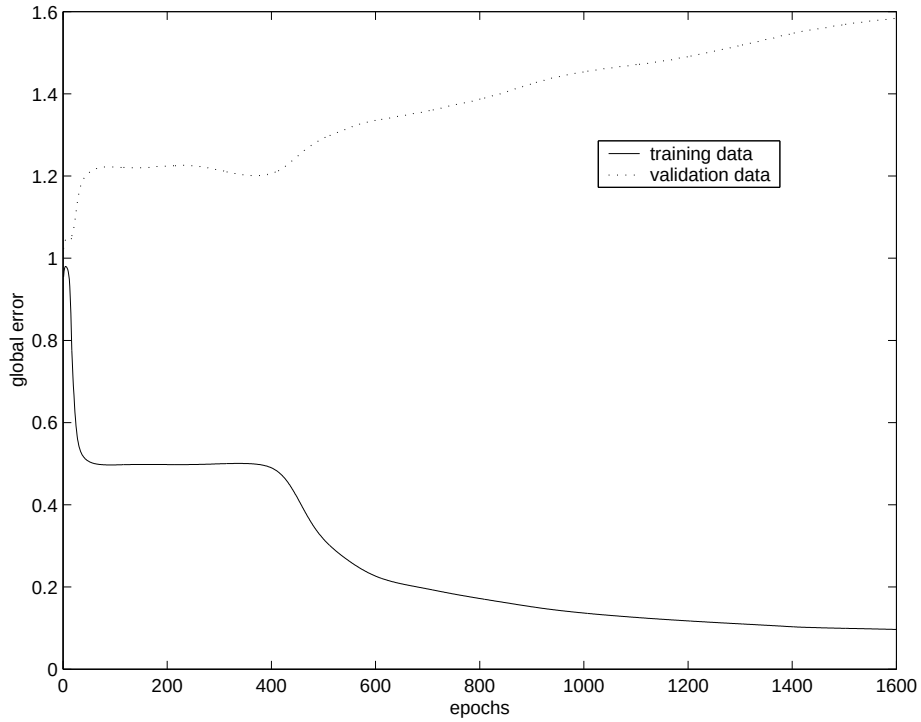
Figur 10: *Entwicklung des Fehlers bei epochalem Update der Gewichte*

Das Resultat war zunächst überraschend viel schlechter, eigentlich ziemlich sinnlos (Figur 11). Der Fehler bei der Validierung nahm augenblicklich zu, das Netz entwickelte keinerlei Generalisierungsfähigkeit.

Es kann aber so verstanden werden: Da jetzt nicht mehr 8400 Fehler, die Fehler einer ganzen Epoche, aufsummiert wurden, hatte der einzelne Fehler wahrscheinlich viel zu viel Gewicht. Ich reduzierte also den Lernfaktor also um drei Grössenordnungen. Da sich so jedoch nach 50 Epochen noch nichts verändert hatte, erhöhte ich den Faktor wieder auf 1e-06 und Figur 12 zeigt das Resultat.

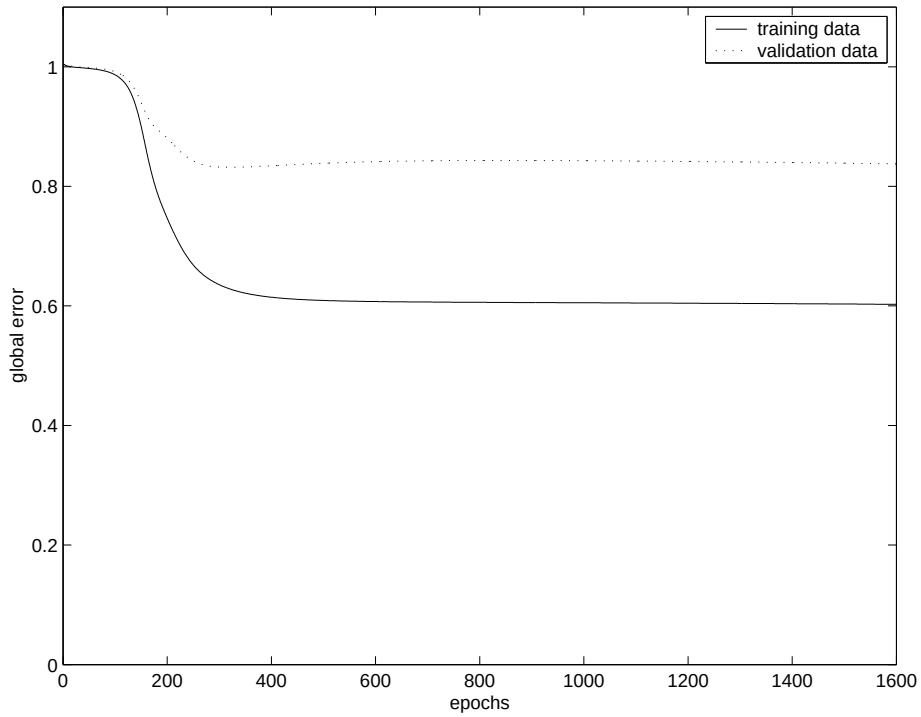
Trainingsparameter	
Update Methode	nach jedem Vektor
Anzahl Epochen	1600
Lernfaktor	1e-06
Momentum	0.0

Der augenfälligste Unterschied ist der viel glattere Verlauf der Lernkurven. Das lässt sich durchaus verstehen, wenn man bedenkt, dass wesentlich kleinere Gewichtsanpassungen pro Schritt vorgenommen werden. Der Zustand des Netzes wird sich also niemals abrupt ändern, wie dies beim Zurückpropagieren eines Fehlers einer ganzen Epoche schon mal vorkommen kann. Die Trainingskurve schien auch monoton zu sein, bei der Validierungskurve war zuerst eine monotone Verbesserung, danach eine monotone Verschlechterung zu beobachten, was die Bestimmung eines Minimums erleichtern und ein *early stopping* (siehe unten) möglich machen könnte. Nur war die Performanz im Ganzen viel

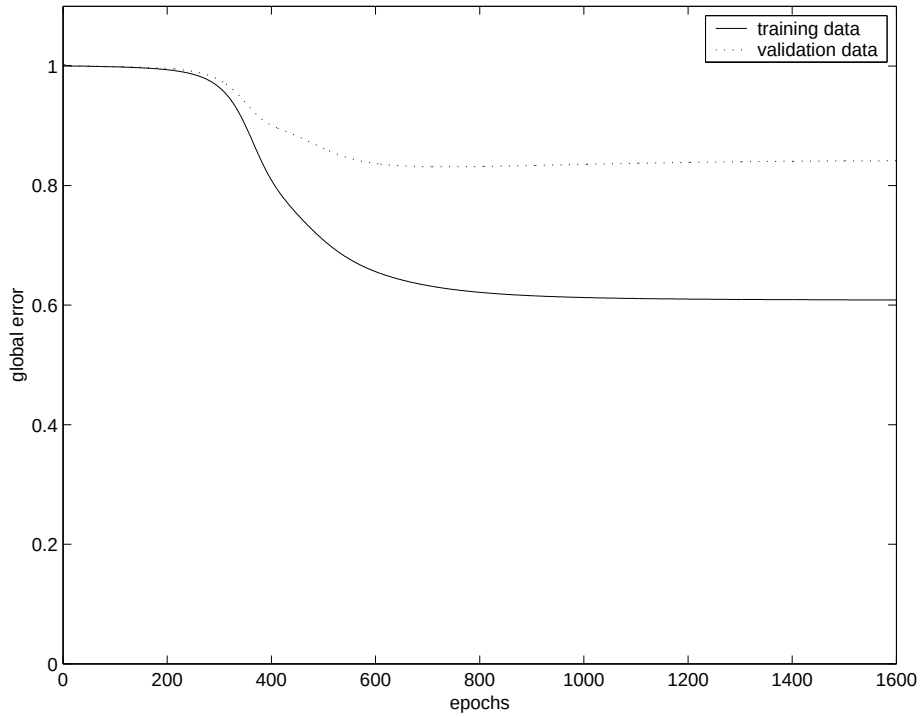


Figur 11: *Globaler Fehler bei Update der Gewichte nach jedem Vektor, Lernrate 1e-04*

schlechter. Das erreichte Minimum der Verallgemeinerung war 8.315 im Vergleich zu den epochalen 0.5934. (Auch eine weitere Erhöhung der Lernrate auf 5e-06 brachte keine Verbesserung, wie Figur 13 zeigt). Für die weiteren Versuche behielt ich darum die Methode der epochalen Gewichtsanpassung bei.



Figur 12: *Globaler Fehler bei Update der Gewichte nach jedem Vektor, Lernrate 1e-06*



Figur 13: *Globaler Fehler bei Update der Gewichte nach jedem Vektor, Lernrate 5e-06*

10 Zwischenüberlegung: The Curse of Dimensionality

10.1 Idee: Weniger kann mehr sein

Die Anzahl der Autokorrelations-Koeffizienten hatte ich im Vergleich zur Literatur ([4]) auf 20 erhöht in der selbstverständlichen Annahme, dass mehr Information die Leistung eines Klassifizierungsnetzes verbessern würde. Das folgende Zitat stellt dies aber in Frage:

> "We begin by dividing each of the input variables into a number of intervals, so that the value of a variable can be specified approximately by saying in which interval it lies. This leads to a division of the whole input space into a large number of boxes or cells. Each of the training examples corresponds to a point in one of the cells, and carries an associated value of the output-variable y. If we are given a new point in the input space, we can determine a corresponding value for y by finding which cell the point falls in, and then returning the average value of y for all of the training points which lie in that cell. By increasing the number of divisions along each axis we could increase the precision with which the input variables can be specified. There is, however, a major problem. If each input variable is divided into M divisions, then the total number of cells is M^d and this grows *exponentially* with the dimensionality of the input space. Since each cell must contain at least one data point, this implies that the quantity of training data needed to specify the mapping also grows exponentially. This phenomenon has been termed the *curse of dimensionality* (Bellman, 1961). If we are forced to work with a limited quantity of data, as we are in practice, then increasing the dimensionality of the space rapidly leads to the point where the data is very sparse, in which case it provides a very poor representation of the mapping." [7]

Natürlich ist diese Technik, den Input-Raum in Zellen aufzuteilen, ein sehr ineffizienter Weg, eine multivariate nicht-lineare Funktion darzustellen. Er berücksichtigt wichtige Eigenschaften realer Daten überhaupt nicht: so sind üblicherweise Input-Variablen auf irgendeine Art korreliert, so dass sie nicht über den ganzen Input-Raum verstreut sind, sondern auf einen niedriger dimensionalen Subraum beschränkt sind (*intrinsic dimensionality*). Weiter ändern die Werte der Inputvariablen für die meisten Probleme von Interesse von einer Region des Input-Raumes zur nächsten nicht in beliebiger Weise, so dass etwas Ähnliches wie Interpolation benutzt werden kann, wo keine Daten vorhanden sind.

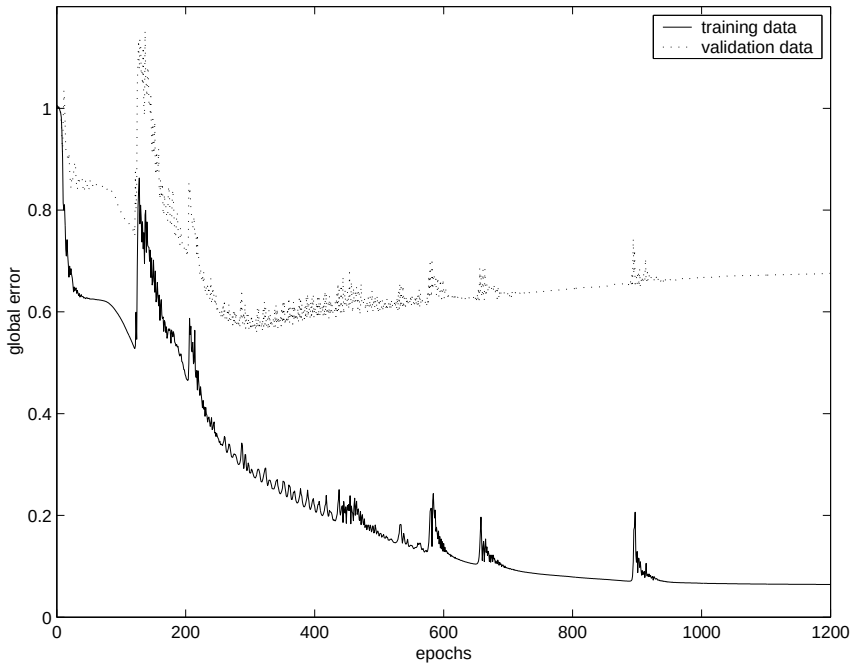
Trotzdem kann in vielen Fällen die Reduktion der Inputdimension für eine gegebene Menge Inputvektoren zu einer besseren Performanz führen. Die fixe Menge Daten kann die Abbildung in niedriger dimensionierten Räumen besser spezifizieren und dies kompensiert den Verlust der zur Verfügung stehenden Information.

10.2 Ergebnis

Um die Hypothese zu prüfen, dass eine Reduktion der Dimension des Inputraumes eine Verbesserung der Generalisierung bewirken könnte, habe ich die Zahl der Autokorrelationskoeffizienten auf die Hälfte reduziert. Damit das Verhältnis zwischen freien Netzparametern und Anzahl Inputvektoren in etwa dasselbe sei, erhöhte ich die Anzahl der *hidden units*. Die Trainings- und Validationsdaten waren selbstverständlich die gleichen wie vorher.

Netzparameter	
Topologie	3-layer Perceptron
Anzahl Hidden Units in Layer 1	78
Anzahl Hidden Units in Layer 2	24
Anzahl Gewichte	3584
Fehlerfunktion output	mean square error
Trainingsparameter	
Update Methode	Epoche
Anzahl Epochen	2200
Lernfaktor	1e-04
Momentum	0.9

Als Ergebnis war wirklich eine Reduktion des Fehlers auf dem Validierungsset zu beobachten, während das Lernverhalten in etwa gleich blieb (Figure 14). Das erreichte Minimum war 0.5609 nach Epoche 310, eine Verbesserung um 5.48%.



Figur 14: *Entwicklung des Fehlers bei Reduktion der Inputdimension*

11 Zwischenüberlegung: Lernen und Generalisieren

Das Ziel eines Netztrainings ist nicht, eine exakte Darstellung der Trainingsdaten zu lernen, sondern ein statistisches Modell des Prozesses zu bilden, der diese Daten generiert hat. Nur so ist es überhaupt möglich, zu generalisieren, also gute Voraussagen zu Daten zu machen, die noch unbekannt sind. Das Modell soll dabei so einfach wie möglich sein (Ockham's Razor: "One should not increase, beyond what is necessary, the number of entities required to explain anything.").

Eine simple Analogie mag das noch verdeutlichen: wollen wir z.B. drei Punkte, welche auf einer Geraden liegen sollten, aber mit je einem zufälligen Fehler versehen sind, mit einem Polynom interpolieren, so gibt es selbstverständlich eine Parabel, auf welcher alle drei Punkte liegen, welche aber dem eigentlichen linearen Entstehungsprozess überhaupt nicht angemessen ist. Die Parabel ist zu komplex, zu flexibel, hat zu viele freie Parameter für dieses Problem.

Um also die beste Verallgemeinerungsfähigkeit zu erreichen, müssen wir die Komplexität des beschreibenden Modells irgendwie kontrollieren und optimieren. Bei neuronalen Netzen gibt es hierzu zwei prinzipielle Ansätze ([7]): *Strukturelle Stabilisierung* und *Regularisierung*. Bei der strukturellen Stabilisierung wird die Komplexität über die Anzahl adaptiver Parameter kontrolliert, die Regularisierung führt einen Strafterm ein, der zur Fehlerfunktion addiert wird. In den nächsten zwei Abschnitte werden diese Ansätze genauer beschrieben.

11.1 Strukturelle Stabilisierung

Der einfachste Weg, die Komlexität eines künstlichen neuronalen Netzes zu optimieren wäre eine erschöpfende Suche durch eine bestimmte Klasse von Architekturen. Wir könnten z.B. die Klasse der 3-Schichten-Perzeptrons wählen, von denen wir wissen, dass sie prinzipiell beliebig komplexe Regionen zu klassifizieren im Stande sind. Dann bliebe nur noch die Anzahl M der *hidden units* zu spezifizieren übrig. Wir würden dann eine Menge Netze mit verschiedenem M trainieren und dasjenige nehmen, welches nach unseren Kriterien das Beste ist. Dieser von mir schon versuchte Ansatz hat folgende Probleme: erstens ist er sehr rechenintensiv. Zweitens hängen von M vielleicht noch andere Parameter ab (Lernrate, Updatemethode, Momentum, Anzahl Lernepochen ...), so dass die Suche entlang dieser einen Dimension nicht viel bringt.

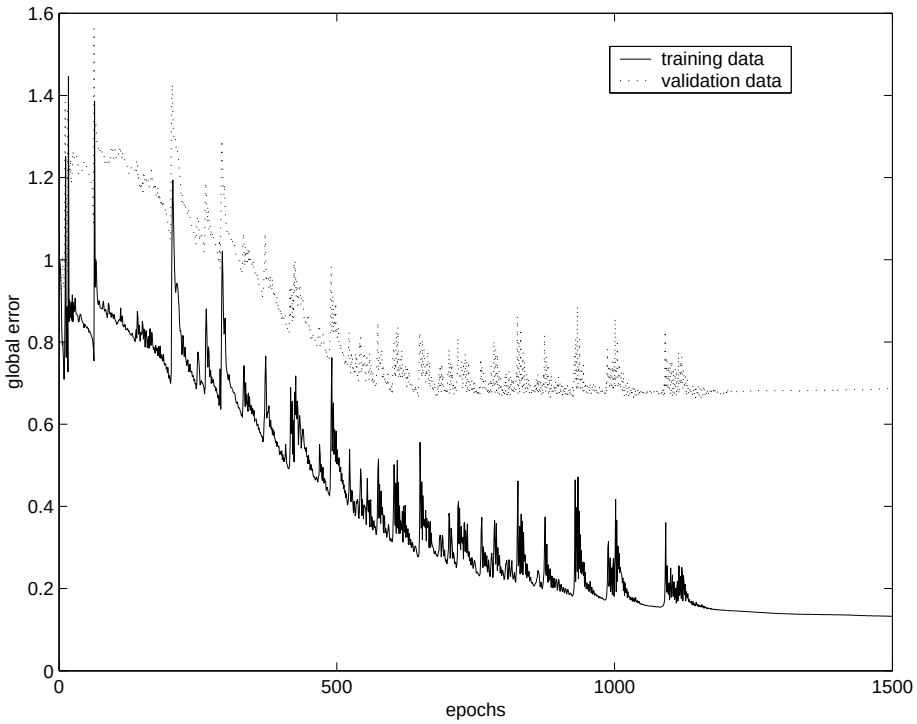
Ein weiterer Nachteil ist, dass sehr viele verschiedene Netze trainiert werden müssen. Dies könnten wir umgehen, indem wir mit einem kleinen Netz beginnen und während des Trainings neue Einheiten und Gewichte hinzufügen. Oder wir beginnen mit einem relativ grossen Netz und entfernen sukzessive Gewichte oder ganze Neuronen. Diese Technik wird *pruning* genannt und das NICO-Toolkit stellt sie zur Verfügung.

11.1.1 Pruning mit Grenzwert

Beim *pruning* gibt es mehrere Entscheidungen zu treffen, etwa wie viel Training zwischen den einzelnen *prunings* gemacht werden soll, wieviele Gewichte herausgeschnitten werden

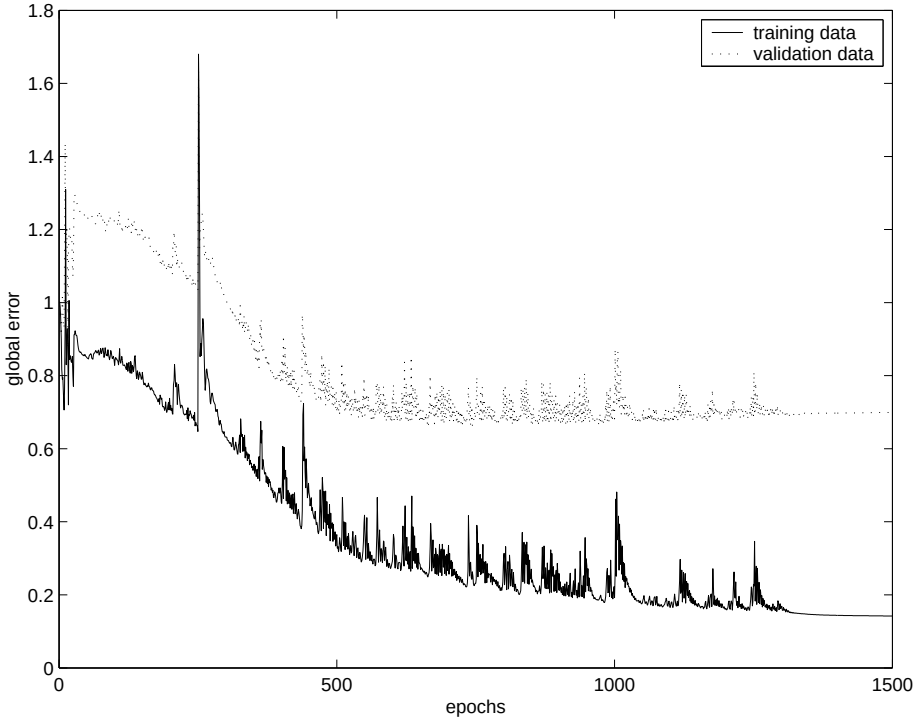
sollen und so weiter. Üblicherweise geschieht das heuristisch. Die wichtigste Frage aber ist, welche Gewichte man entfernen soll. Wir brauchen also irgendein Mass für die Wichtigkeit der Gewichte.

Das einfachste Konzept ist: kleine Gewichte sind weniger wichtig als grosse. Dafür gibt es eigentlich wenig theoretische Motivation und diese Art der Gewichtsreduktion sei gemäss Literatur [7] in der Praxis auch nicht sehr erfolgreich. Ich habe das trotzdem probiert: aus einem anfänglich sehr grossen Netz wurden nach 250 Trainingsepochen (ungefähr dort nahm der Fehler auf den Validierungsdaten jeweils wieder zu) die Verbindungen herausgeschnitten, welche einen bestimmten Grenzwert nicht überschritten. Training und Pruning wurden 6 Mal wiederholt. Der ganze Versuch wurde mit 3 verschiedenen Grenzwerten gemacht (Figuren 15 bis 18).

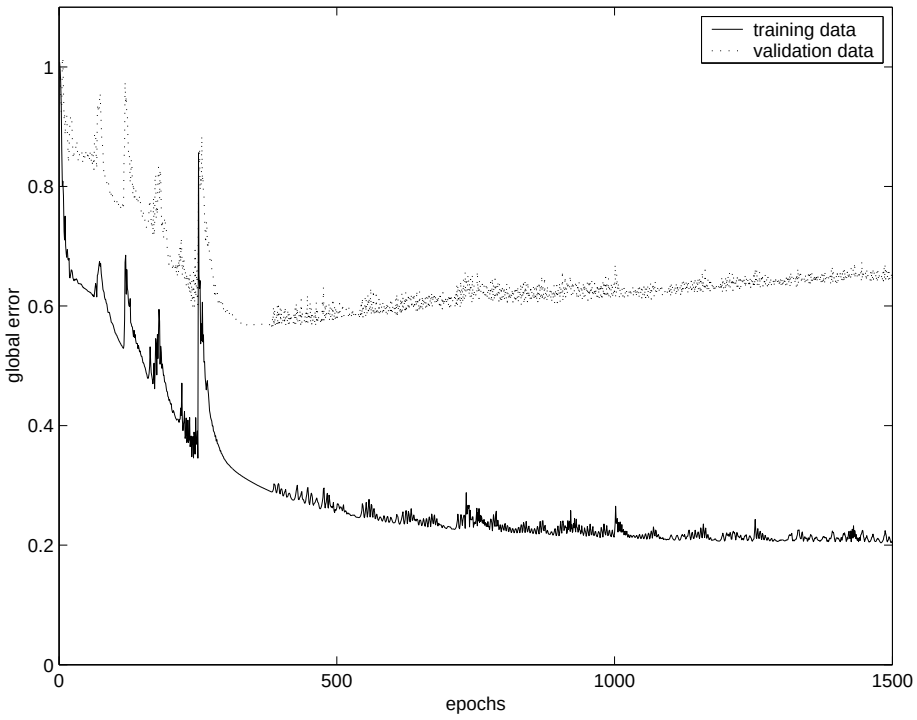


Figur 15: *Pruning mit Grenzwert 0.05*

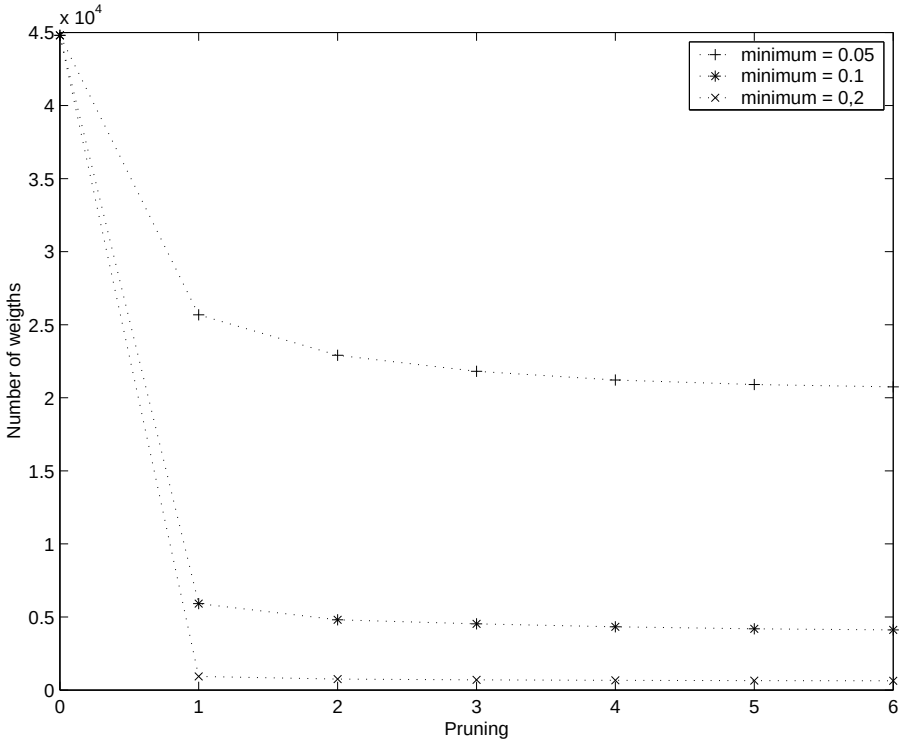
Das erste Pruning nach 250 Lernepochen hatte deutlich die grösste Auswirkung. Am besten schnitt das Netz mit dem grössten Grenzwert ab. Es hatte nach dem ersten Pruning nur noch 931 Verbindungen und erreichte nach 385 Iterationen einen Validierungsfehler von 0.5703. Dies ist noch nicht besser als das vorherige Netz, aber es sind deutlich weniger Gewichte.



Figur 16: *Pruning mit Grenzwert 0.1*



Figur 17: *Pruning mit Grenzwert 0.2*

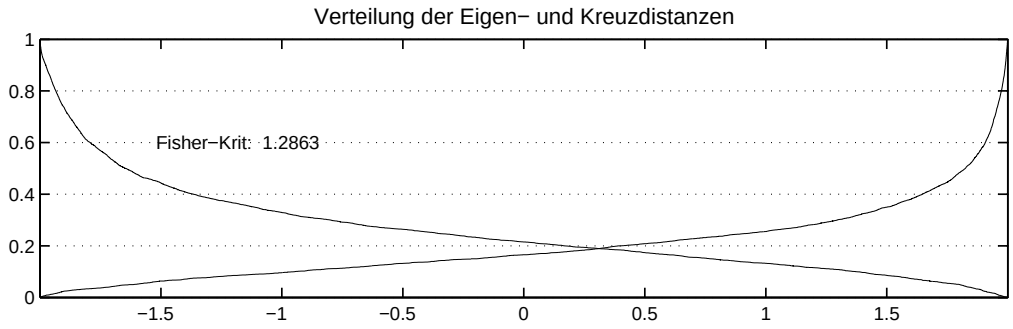


Figur 18: *Entwicklung der Anzahl Parameter bei verschiedenen Grenzwerten*

Dieses Netz habe ich dann noch eingehender untersucht. Eine Konfusionsmatrix, berechnet auf den Validierungsdaten, sieht so aus:

```
CResult -S -c prune1.rtdnn validation_20
NICO -- Frame-Level Classification Statistics

Number of Frames: 8400
Frames Correct: 81.0 %

Class       Freq    Correct[%]    False Alarms[%]
self        4200       78.5       16.5
cross       4200       83.5       21.5

Confusion Matrix
                c
          s     r
          e     o
          l     s
          f     s
  self 78.5 21.5
 cross 16.5 83.5
--
```

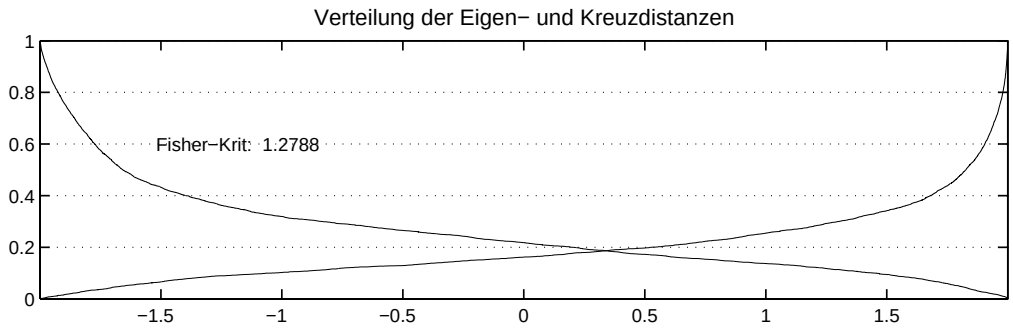
Die Validierungsvektoren lieferten einen Output, dessen Verteilung ich in Figur 19 geplottet habe. Dazu muss noch folgendes gesagt werden: Mein Klassifizierungsnetz hatte ja zwei Ausgänge, der erste sollte "hoch" sein, wenn die Eingänge vom selben Sprecher, der zweite dann, wenn sie von zwei verschiedenen Sprechern stammen. Um nun ein vergleichbares Abstandsmass zu erhalten, das dann klein ist, wenn die Proben vom selben Sprecher sind, subtrahierte ich den ersten Ausgang vom zweiten. Da die Ausgänge im Intervall [-1,1] liegen, ist im besten Fall bei gleichem Sprecher Ausgang eins 1 und Ausgang zwei -1, d.h. die erwähnte Differenz -2 (bei verschiedenen Sprechern entsprechend +2). Das Fisher-Kriterium wurde auch berechnet und ergab einen ganz ansprechenden Wert.



Figur 19: *Verteilungen und Fisher-Kriterium beim besten Pruning (Grenzwert 0.2 nach 385 Epochen)*

Nun wollte ich das noch mit einem ähnlich grossen Netz (962 Gewichte), welches gleich lange trainiert wurde vergleichen. Insgeheim hoffte ich natürlich, dass dieses viel schlechter abschneide und sich die Optimierungstechnik *pruning* bewähre. Enttäuschenderweise war das Ergebnis aber beinahe gleich (Figur 20). Warum einfach, wenn es auch kompliziert geht...

```
CResult -S -c small.rtdnn validation_20
NICO -- Frame-Level Classification Statistics

Number of Frames: 8400
Frames Correct: 81.0 %

Class       Freq    Correct[%]    False Alarms[%]
self        4200       78.2       16.2
cross       4200       83.8       21.8

Confusion Matrix
               c
          s    r
          e    o
          l    s
          f    s
  self 78.2 21.8
 cross 16.2 83.8
--
```



Figur 20: *Verteilungen und Fisher-Kriterium bei 965 Gewichten*

11.1.2 Pruning "2nd order mode"

Da ja das Training als Minimierung einer Fehlerfunktion definiert ist, ist es nur natürlich, die selbe Fehlerfunktion zu verwenden, um die Bedeutung der Gewichte abzuschätzen. Wir können das Mass der Wichtigkeit eines Gewichtes also definieren als die Änderung der Fehlerfunktion, welche aus der Eliminierung dieses Gewichtes resultiert. Dies kann auf direktem Weg berechnet werden, was aber sehr rechenintensiv ist. Eine bessere Methode soll hier nur angedeutet werden.

Wir betrachten dazu die Änderung der Fehlerfunktion bei einer Änderung des Gewichtes w_i um δw_i:

$$\delta E = \sum_i \frac{\partial E}{\partial w_i} \delta w_i + \frac{1}{2} \sum_i \sum_j H_{ij} \delta w_i \delta w_j + O(\delta w^3) \tag{3}$$

wobei H_{ij} die Elemente der Hessischen Matrix sind:

$$H_{ij} = \frac{\partial^2 E}{\delta w_i \delta w_j} \tag{4}$$

Der erste Term in (3) verschwindet, wenn das Training konvergiert. Und wenn wir die Hessische Matrix durch weglassen der Nicht-Diagonalelemente approximieren und die Terme höherer Ordnung vernachlässigen, reduziert (3) auf:

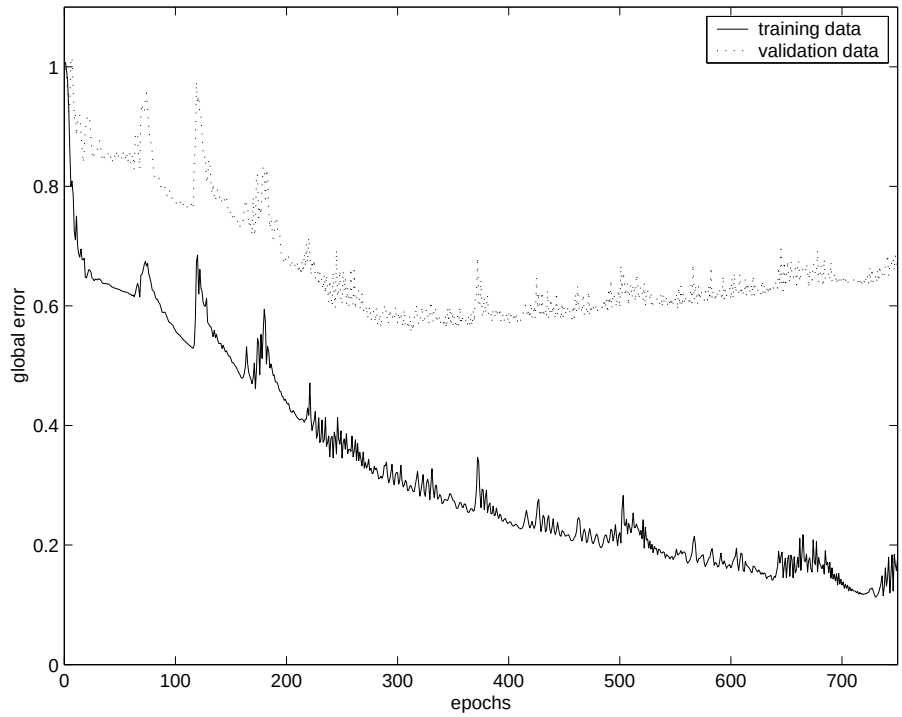
$$\delta E = \frac{1}{2} \sum_i H_{ii} \delta w_i^2 \tag{5}$$

Wenn nun ein Gewicht w_i auf Null gesetzt wird, ist die Änderung in der Fehlerfunktion näherungsweise durch (5) gegeben mit $\delta w_i = w_i$ und das Mass für die Bedeutung von w_i ist gegeben durch $H_{ii} w_i^2 / 2$.

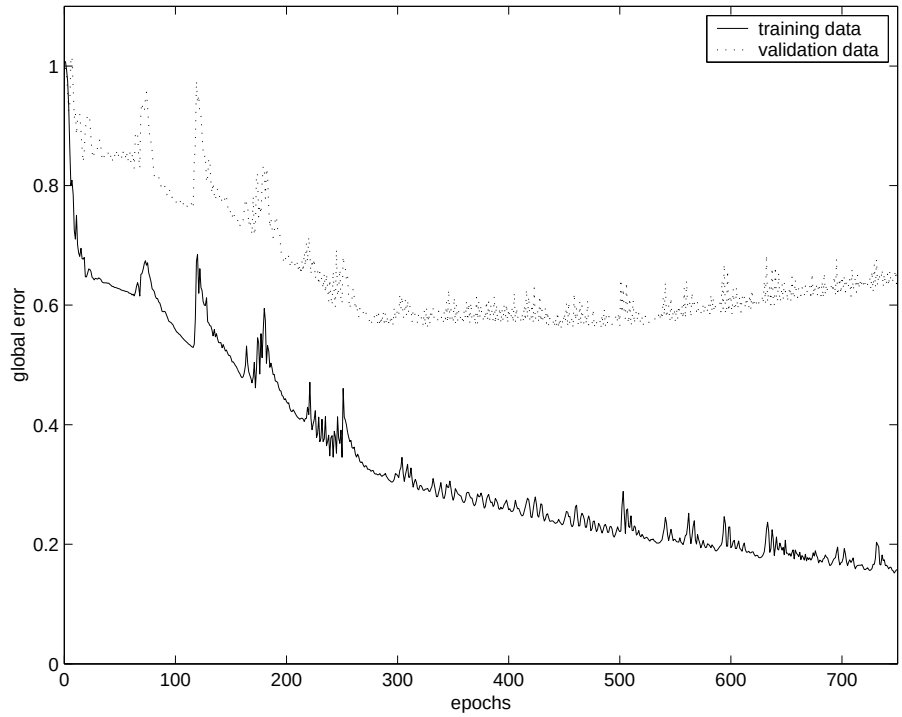
Dieser Ansatz wurde *optimal brain damage* genannt (Le Cun *et al.*, 1990). Lässt man die Approximation der Hessischen durch eine Diagonalmatrix weg (die oft nicht sehr gut ist), kommt man zu einer etwas komplizierteren Methode, genannt *optimal brain surgeon.*

NICO stellt eine Methode zur Verfügung, die dort *pruning 2nd order mode* genannt wird. Es geht aus der Dokumentation nicht hervor, welche der obigen Methoden gemeint ist, ich vermute aber, es ist die rechenintensive direkte Methode. Ich habe das auch ausprobiert, wobei hier wieder ein Paramter mehr dazu kommt, für den ich einen Wert aus der Luft greifen muss: "the maximum change in promille of the total error that are tolerated for removing a connection".

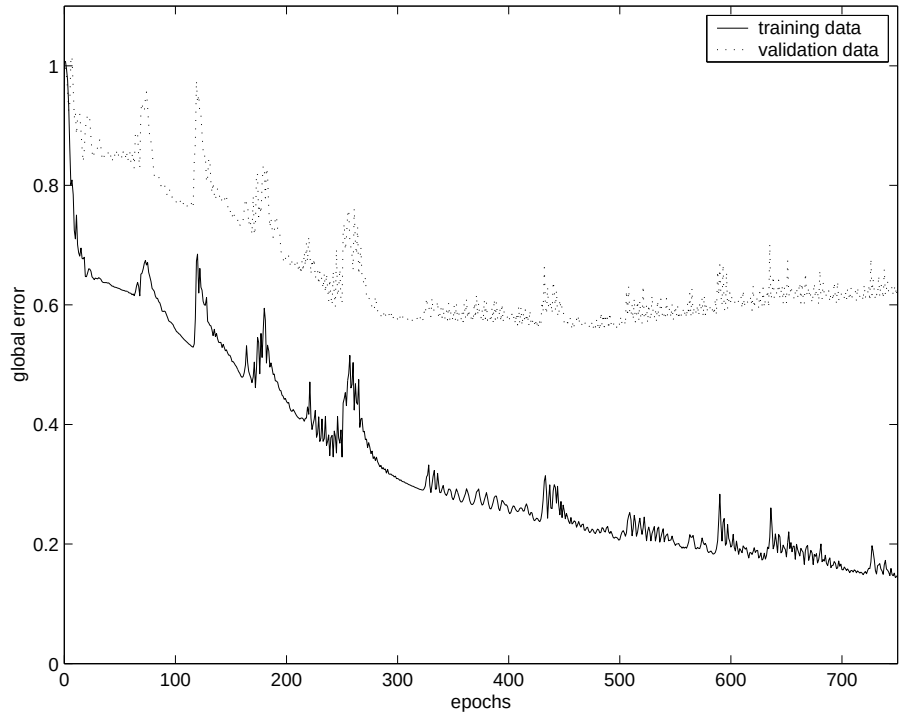
Figuren 21 und 22 zeigen Lernkurven zu zwei verschiedenen Parametern, einmal 10 Promille, einmal 100 Promille Fehlertoleranz. Die Ableitungen wurden auf den Fehlern zu den Validierungsvektoren gerechnet. Auch hier liegen die erreichten Fehlerminima um 0.56. Figur 23 ist das Resultat, wenn die Fehler der Trainingsvektoren zur Berechnung der zweiten Ableitung benutzt werden. Die Entwicklung der Anzahl Gewichte ist in Figur 24 gezeigt.



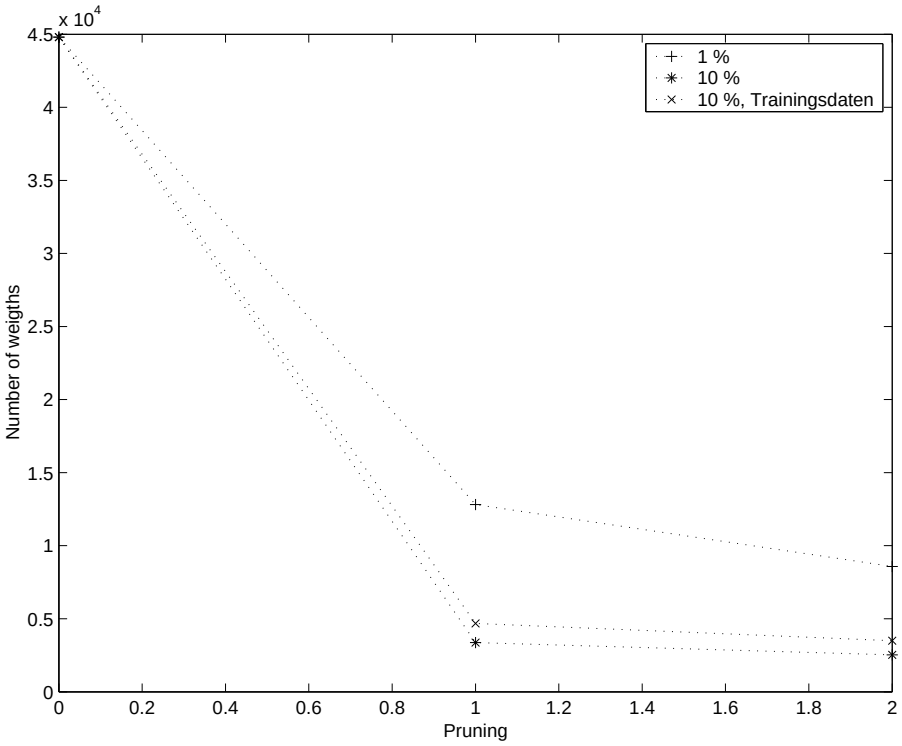
Figur 21: *2nd Order Pruning mit 1% Fehlertoleranz*



Figur 22: *2nd Order Pruning mit 10% Fehlertoleranz*



Figur 23: *2nd Order Pruning mit 10% Fehlertoleranz, Fehler der Trainingsvektoren*



Figur 24: *Gewichtsreduktion bei 2nd Order Pruning*

11.2 Regularisierung

Das überwachte Lernen in einem künstlichen neuronalen Netz bedeutet ja nichts anderes, als die Parameter (die Gewichte im Netz) zu finden, welche eine geeignet definierte Fehlerfunktion minimieren. Diese Fehlerfunktion beschreibt die Abweichung des Outputs des Netzes vom gewünschten Output. Das Problem, bekannt unter dem Namen *credit assignment problem*, ist, diejenigen *hidden units* zu identifizieren und anzupassen, welche einen auftretenden Fehler verursacht haben. Die Lösung ist, differenzierbare Aktivierungs- und Fehlerfunktionen zu wählen. Dann wird nämlich der Fehler eine differenzierbare Funktion von den Gewichten, und das Berechnen der Ableitungen nach den Gewichten kann benutzt werden, diejenigen Werte zu finden, welche die Fehlerfunktion minimieren. Der Algorithmus zur Berechnung der Ableitungen wird dann eben *back-propagation* genannt.

Die Idee der Regularisierung ist nun, dieser Fehlerfunktion noch einen Strafterm Ω hinzu zu fügen:

$$\overline{E} = E + \nu * \Omega \tag{6}$$

Dabei is ν ein Parameter, welcher den Einfluss dieses Terms kontrolliert und Ω eine differenzierbare Funktion der Gewichte. Somit haben wir die Möglichkeit, neben einer möglichst guten Anpassung an die Trainingsdaten, andere Anforderungen an die durch das neuronale Netz beschriebene Abbildung zu stellen und auf diese Weise einem *overfitting* entgegenzuwirken.

Generell lassen sich überangepasste Lösungen charakterisieren durch unruhig verlaufende Abbildungen mit viel Struktur und grosse Werte der Ableitungen, viel Krümmung. Beschreibt also der Strafterm diese hohe Strukturierung in irgendeiner Weise, so begünstigt die Minimierung desselben einen glatteren Verlauf der Abbildung und verhindert ein *overfitting*, oder schränkt es zumindest ein. Da die zweite Ableitung die Krümmung beschreibt, wäre zum Beispiel folgendes denkbar:

$$\Omega = \frac{1}{2} \sum_n \sum_i \sum_k \left(\frac{\partial^2 y_k^n}{\partial x_i^2}\right)^2 \tag{7}$$

wobei die Indices über alle Trainingsbeispiele (n), alle Outputvariablen (k) und alle Gewichte (i) laufen.

11.2.1 Weight Decay

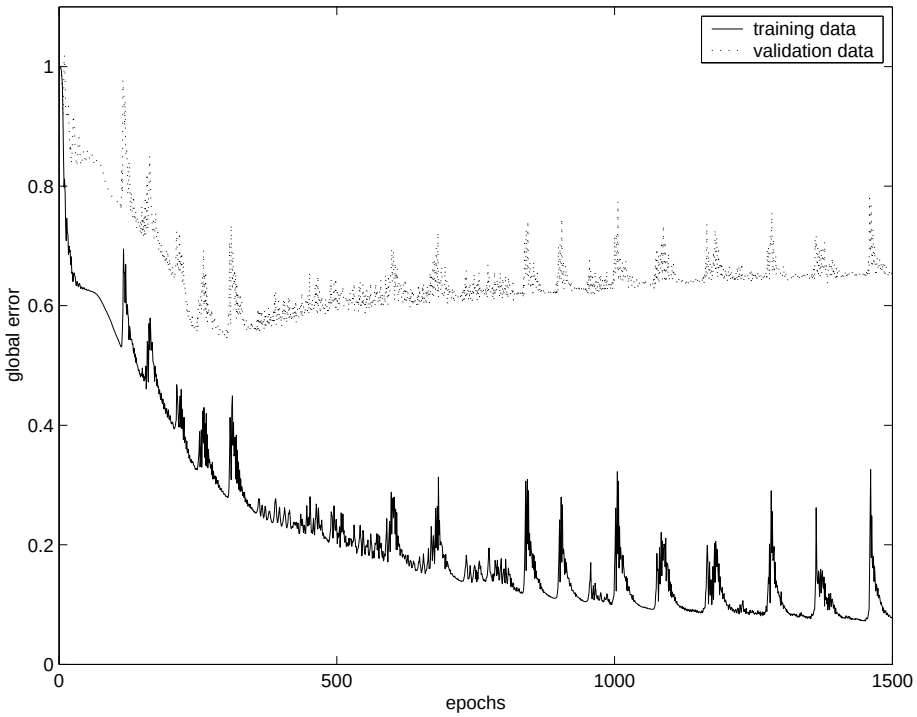
Eine viel einfachere Regularisiersmethode, ist der sogenannte *weight decay*. Der Regularisierungsterm besteht aus der Summe der Quadrate der adaptiven Parameter:

$$\Omega = \frac{1}{2} \sum_i w_i^2 \tag{8}$$

wobei die Summe über alle Gewichte geht. Es wurde empirisch festgestellt, dass eine Regularisierung in dieser Form die Verallgemeinerungsfähigkeit eines Netzes signifikant

erhöhen kann ([8]). Eine heuristische Rechtfertigung kann folgendermassen gegeben werden: Wir wissen, dass relativ grosse Werte nötig sind für ein *overfitting*. Für kleine Werte der Gewichte ist das Mapping eines Multilayer-Perzeptrons in erster Näherung linear, da die zentrale Region einer Tanh-Aktivierungsfunktion approximiert werden kann durch eine lineare Transformation. Durch den obigen Regularisierungsterm werden nun kleine Gewichte begünstigt, da grosse Gewichte einen grossen Fehlerterm ergeben.

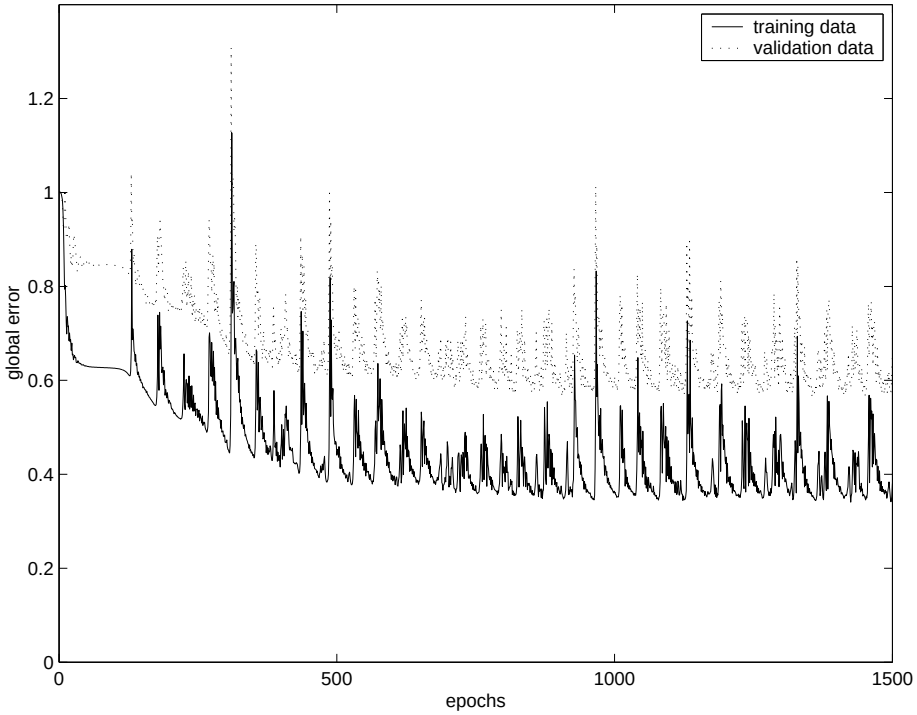
Auch zu dieser Technik habe ich zwei Versuche gemacht, wobei ich den Regularisierungsfaktor (ν in Gleichung 6) variiert habe. Bei einem grossen Faktor nehmen die Oszillationen der Lernkurve sowohl auf den Trainings- wie auf den Validierungsvektoren stark zu. Hier könnte vielleicht eine andere Methode der Gewichtsanpassung (nach jedem Vektor zum Beispiel) Abhilfe schaffen. Auch diese Fehlerminima bei der Verallgemeinerung sind von der gleichen Grössenordnung wie die bisherigen.



Figur 25: *Weight Decay mit Faktor 0.001*

11.2.2 Early Stopping

Wie wir in beinahe allen Grafiken gesehen haben, ist die Lernkurve in Hinblick auf die Trainingsvektoren eine monoton fallende Kurve (sieht man von dem unruhigen Verlauf mit vielen lokalen Minima und Maxima ab), während die Fehlerkurve der Validierungsvektoren so etwas wie ein globales Fehlerminimum erreicht und dann wieder ansteigt. Wir könnten nun eigentlich das Training bei diesem Minimum stoppen, da das Netz hier seine grösste Verallgemeinerungsfähigkeit hat. Dies wird *early stopping* genannt. Es kann gezeigt werden, dass im Falle einer quadratischen Fehlerfunktion *early stopping* einen ähnlichen Effekt hat wie Regularisierung mit einem einfachen *weight decay* Term, weshalb ich



Figur 26: *Weight Decay mit Faktor 0.02*

diese Technik in diesem Kapitel erwähne.

Praktisch stellt sich hier aber das folgende Problem: wie erkenne ich, bei dem unruhigen Verlauf der Kurven aufgrund des epochalen Aufdatierens, ein globales Minimum rechtzeitig, um das Training zu stoppen?

Eigentlich müsste ich nach jedem Trainingsschritt den Zustand des Netzes speichern, um dann im Nachhinein das beste auszuwählen. Da mir der Aufwand zu gross schien, habe ich das nicht verfolgt.

Eine andere Idee ist, ein Training durchzuführen, die Iteration mit dem kleinsten Fehler zu identifizieren und dann das Experiment bis dorthin zu wiederholen. Das ist auch nicht so einfach möglich. Einerseits werden die Gewichte zufällig initialisiert (dem kann man natürlich abhelfen, indem man das Netz einmal generiert und dann für jedes Training einfach kopiert), andererseits scheint auch im Verlaufe des Trainings mit NICO der Zufall eine Rolle zu spielen. Es ist mir jedenfalls nie gelungen, ein Training exakt zu wiederholen.

12 Übersicht über das Erreichte

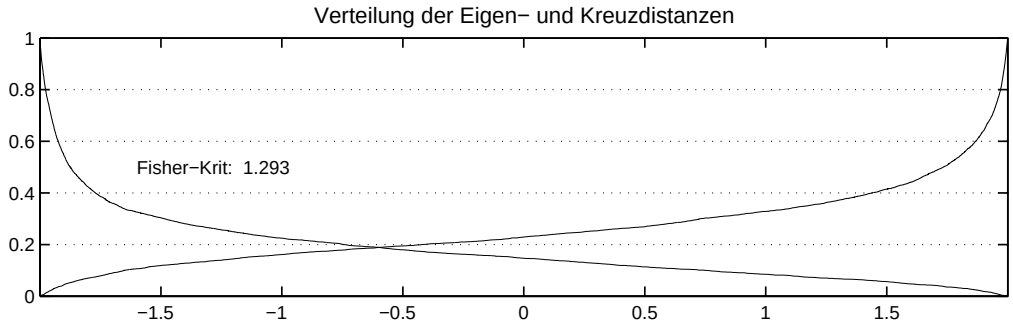
In einer tabellarischen Übersicht sollen nun die Testresultate noch einmal zusammengefasst werden. Wie erwähnt hatte ich den Netzzustand zu den erreichten Fehlerminima nicht speichern können. Deshalb war es mir nicht möglich, zu jedem Test das Fisherkriterium zu berechnen.

Experiment	erreichtes Minimum	nach Epoche
Gleiche Sprecher, 40 AK	0.5934	277
Gleiche Sprecher, 20 AK	0.5609	310
Pruning, w=0.05	0.6639	1025
Pruning, w=0.1	0.6622	744
Pruning, w=0.2	0.5652	385
2nd Order Pruning, 1%	0.5565	312
2nd Order Pruning, 10%	0.5611	447
2nd Order Pruning, 10%, Trainingdata	0.5569	482
Weight Decay, ν=0.001	0.5467	302
Weight Decay, ν=0.02	0.5638	1356

Um jetzt noch einen Vergleich der Performanz des Ansatzes mit neuronalen Netzen mit dem bisherigen EC-Distanzmass zu bekommen, versuchte ich zum Schluss mit einer Kombination der verschiedenen Techniken ein möglichst gutes Netz zu trainieren. Dazu wurde das Skipt auf der folgenden Seite verwendet:

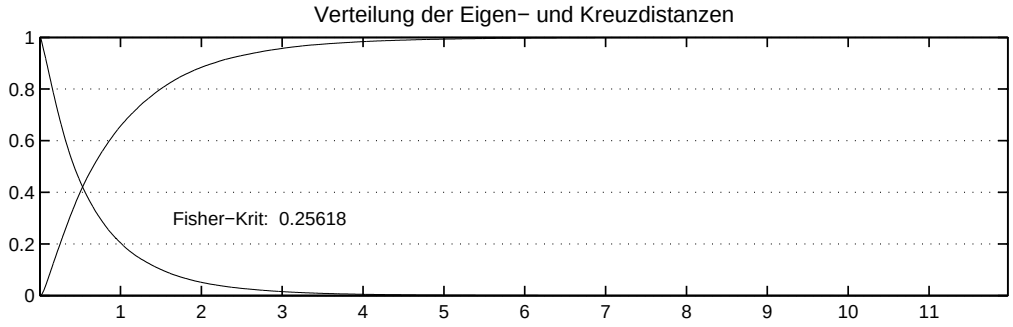
```
#!/bin/sh -v
NAME=finale
NET=$NAME.rtdnn
LOG=$NAME.log

INPDIM=20
HIDDENSIZE1=60
HIDDENSIZE2=18
OUTDIM=2

CreateNet $NET $NET

AddGroup input $NET
AddUnit -i -u $INPDIM input $NET

AddGroup hidden1 $NET
AddUnit -u $HIDDENSIZE1 hidden1 $NET

AddGroup hidden2 $NET
AddUnit -u $HIDDENSIZE2 hidden2 $NET

AddGroup output $NET
AddUnit -o -S names output $NET

SetType -t    hidden1 $NET
SetType -t    hidden2 $NET
SetType -O L2 output  $NET

AddStream -x input  -d data/ -F ascii $INPDIM r INPUT  $NET
AddStream -x output -d data/ -F ascii $OUTDIM t OUTPUT $NET

LinkGroup INPUT  input $NET
LinkGroup OUTPUT output $NET

Connect input   hidden1 $NET
Connect hidden1 hidden2 $NET
Connect hidden2 output  $NET

BackProp -S -d -p $LOG -g1e-04 -w 0.001 -v validation
                              OUTPUT -E -T 3 -i280 $NET training

Prune -g 10 -w 0.1 -S $NET training_20

BackProp -S -d -p 1$LOG -g1e-04 -w 0.001 -v validation
                               OUTPUT -E -T 3 -i100 $NET training
Display $NET
```

Der globale Fehler am Ende des Trainings betrug 0.5784. Die Verteilung der Eigen- und Kreuzdistanzen und das Fisher-Kriterium zeigt Figur 27.



Figur 27: *Verteilungen und Fisher-Kriterium für das Neuronale Netz*

Man vergleiche damit Figur 28, dieselbe Grafik für das EC-Distanzmass.



Figur 28: *Verteilungen und Fisher-Kriterium für das EC-Distanzmass*

13 Fazit

Die aufgestellte Hypothese, dass künstliche neuronale Netze ein besseres Distanzmass liefern könnten, konnte mit mehreren Beispielen belegt werden. Die von den neuronalen Netzen erreichten Werte für das Fisher-Kriterium überstiegen bei weitem diejenigen, die vom EC-Distanzmass auf der Ebene von Analysefenstern erreicht wurden (Figuren 27 und 28). Es ist also ein vielversprechender Weg.

Das Finden der optimalen Struktur eines künstlichen neuronalen Netzes erwies sich aber als schwieriger als erwartet. Die Anzahl Variablen bei Konstruktion und Training von neuronalen Netzen ist beträchtlich und das Optimieren so zeitaufwändig, dass man auf gute Heuristiken oder Optimierungstechniken angewiesen ist. Es existieren zahlreiche Tipps und Daumenregeln, von denen ich einige ergebnislos getestet habe. Die Techniken der strukturellen Stabilisierung und Regularisierung habe ich erst nach langer Suche gefunden, so dass sie nicht mehr ausführlich ausprobiert werden konnten.

Ein guter Weg wäre vielleicht, eine systematischere Untersuchung einer bestimmten Optimierungstechnik zu verfolgen.

In dieser Arbeit konnte ich die zwei anderen Probleme der Aufgabenstellung (Merkmalsextraktion und Referenzbildung) aus zeitlichen Gründen gar nicht angehen. Da wäre aber sicher auch noch viel Optimierungspotential verborgen.

Literatur

[1] A. E. Rosenberg. Automatic speaker verification: A review. Proceedings of the IEEE, 64(4): 475-487, April 1976.

[2] C. Bernasconi. Beiträge zum Problem der textabhängigen Sprecherverifikation. Diss. Nr. 8742, Institut für Elektronik, ETH Zürich, Dezember 1988.

[3] B. Pfister und H.-P. Hutter. Sprachverarbeitung I. Vorlesungsskript für das Wintersemester 2001/2002, Departement ITET, ETH Zürich, 2001.

[4] B. Pfister. Sprecherverifikation mit einem neuronalen Netz. AGEN-Mitteilungen Nr. 52, November 1990.

[5] R. Lippmann. An introduction to computing with neural nets. IEEE ASSP Magazine, 4(2): 4-22, April 1987

[6] N. Strom. The NICO Toolkit for Artificial Neural Networks, 1996. (online manual http://www.tik.ee.ethz.ch/internal/SEPP/nico-1.1-be).

[7] C.M. Bishop. Neural Networks for Pattern Recognition. Oxford University Press, 1995.

[8] G.E. Hinton. Learning translation invariant recognition in massively parallel networks. In J. W. de Bakker, A. J. Nijman and P. C. Treleaven (Eds.), Proceedings PARLE Conference on Parallel Architectures and Languages Europe, pp. 1-13. Springer-Verlag Berlin, 1987.

[9] `ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hl`

Anhang A

Vom Institut abgegebene Aufgabenstellung