Selbstlernende Systeme und ihr Optimierungsverhalten auf Fitnesslandschaften mit Anwendungen in der Biotechnologie

Dissertation

zur Erlangung des

akademischen Grades

Doktor rerum naturalium

an der Formal- und Naturwissenschaftlichen

Fakultät der Universität Wien

vorgelegt von

Dipl. Phys. Dirk Tomandl

am

Institut für Theoretische Chemie und Strahlenchemie

Wien, im Jänner 1999

1.Begutachter Prof. Dr. Peter Schuster

2.Begutachter Prof. Dr. Dieter Flamm

... für die Katz' *(Minna, Martha & Gerda)* ...

Danksagung

Die vorliegende Dissertation wurde durch die Forschungsvorhaben BEO 0310665, BEO 0310701, BEO 0310713 und BEO 0311766 des Bundesministeriums für Bildung und Forschung (BMBF) sowie durch die Evotec Biosystems GmbH (Hamburg) und die Fa. Merck KGaA (Darmstadt) unterstützt.

Mein besonderer Dank geht an Prof. Peter Schuster, der unsere Arbeitsgruppe und diese Arbeit mit unter seine Obhut nahm.

Ein herzliches Dankeschön möchte ich Andreas Schober widmen, der zusammen mit Marcel Thürk dieses spannende und interessante Thema vorschlug. Neben viel Hektik herrschte immer eine freundschaftliche, menschliche und konstruktive Atmosphäre.

Dank geht an Gregor „Marvin“ Schlingloff für seinen konsequenten, erfrischenden Pessimismus.

Unermüdliche Diskussionen, abenteuerliche Aktienspekulationen und viel Spaß verbanden (verbinden) mich mit Rüdiger Dietrich.

Keinesfalls dürfen hier Andreas Schwienhorst, Michael Gebinoga und Joachim Schmidt-Brauns vergessen werden.

Thomas Opitz, Stephan Diekmann und Doris Schuster soll an dieser Stelle (aus unterschiedlichen Gründen) für die absurden Seiten des Lebens gedankt werden.

Meinen „neuen“ Merck–Kollegen, Holger Deppe, Beate Diefenbach, Alexander Gross und unserem (Schla-)Wiener Stefan Wuchty danke ich für viel „Schmäh“ und die, unserem Projekt entgegengebrachte Energie.

Zuguterletzt möchte ich meinen Eltern für alles danken, vor allem, da ich in der Diplomarbeit vergaß, sie zu erwähnen.

Dorits Lebensfreude, Geduld, Gehalt und Korrekturvorschläge („Mein Gott, ist das langweilig...“) waren unverzichtbar für mich und halfen über so manches Motivationstal hinweg.

Vorbemerkung

Die vorliegende Arbeit entstand in der Arbeitsgruppe „Systemintegration“ von Andreas Schober. Die Gruppe war von Mitte 1994 bis Mitte 1997 in die Abteilung „Molekulare Evolutionsbiologie“ von Prof. Peter Schuster des Instituts für Molekulare Biotechnologie Jena e.V. (IMB) eingegliedert. Nach der Rückkehr von Prof. Schuster nach Wien siedelte die Arbeitsgruppe „Systemintegration“ zum Institut für physikalische Hochtechnologie Jena e.V. (IPHT) in die Abteilung „Mikrosystemtechnik“ über. Seit Anfang 1998 befindet sich die Arbeitsgruppe, bedingt durch die Kooperation mit der Fa. Merck KGaA im Rahmen des „NanoSynTest“–Projekts, überwiegend in Darmstadt.

Abstract

In this thesis optimization algorithms were developed which are suitable for applications in biotechnology. The algorithms are based on a *generalized frame for optimization* („**G**eneral **O**ptimization **S**hell“ — GOS), into which arbitrary optimization methods can be integrated. Direct applications of the GOS are the so-called **Doping**–algorithm and — as the main part of the thesis — the so-called **TST**–algorithm.

The *Doping–algorithm* is a modified genetic algorithm. It allows to include additional knowledge for the design of usefully reduced peptide- and protein libraries. The algorithm supports evolutionary protein design.

Conventional optimization procedures usually presuppose an explicit analytic fitness function. In contrast the *TST–algorithm* tries to optimize on fitness landscapes with an unknown fitness function or fitness values too expensive to calculate. The goal of the algorithm is to find a tractable and efficient compromise between search time and quality of solution. „Good“ optima, not necessarily the global one, are approached by means of a sufficiently low number of samples. The algorithm only requires a unique sequence–fitness mapping though the fitness values may be disturbed by statistical errors.
The TST–algorithm was first implemented as a prototype, then generalized and analyzed. A central module of the TST–algorithm is the so-called „General Regression Neural Network“ for which new, fast training procedures were developed.
During the development the algorithm was tested exclusively at computer-generated examples. However, the vision is to apply the TST–algorithm to experimental problems, such as optimization of binding constants of peptides to a given target.

Zusammenfassung

In der vorliegenden Arbeit wurden Optimieralgorithmen entwickelt, die sich u.a. für den Einsatz in der Biotechnologie eignen. Basis ist ein *verallgemeinerter Rahmen für Optimierverfahren* („**G**eneral **O**ptimization **S**hell“ — GOS), in den sich beliebige Optimiermethoden integrieren lassen. Konkrete Anwendungen der GOS sind zum einen der sog. **Doping**–Algorithmus, zum anderen — als Kern der Dissertation — der sog. **TST**–Algorithmus.

Der *Doping–Algorithmus* als Variante eines genetischen Algorithmus ermöglicht den Einsatz von Zusatzwissen beim Design sinnvoll reduzierter Peptid- und Proteinbibliotheken. Der Algorithmus unterstützt damit evolutionäres Proteindesign.

Herkömmliche Optimierverfahren setzen in der Regel eine explizite, analytische Fitnessfunktion voraus. Im Gegensatz dazu versucht der hier vorgestellte TST–Algorithmus, auf Fitnesslandschaften, deren Fitnessfunktion unbekannt ist bzw. deren Fitnesswerte nur mit großem Aufwand zu ermitteln sind, zu optimieren. Ziel des Algorithmus ist ein steuerbarer, effizienter Kompromiß zwischen Suchaufwand und Qualität der Lösungen. „Gute“ Optima, nicht unbedingt das globale, werden mit einer möglichst geringen Anzahl von Stichproben/Sequenzen gefunden. Einzige Voraussetzung ist eine eindeutige Sequenz–Fitnesszuordnung, wobei der Fitnesswert dabei auch durch statistische Fehler verrauscht sein kann.
Der *TST–Algorithmus* wurde im Rahmen dieser Dissertation zunächst als Prototyp implementiert, dann verallgemeinert und analysiert. Ein zentrales Modul des TST–Algorithmus ist das sog. „General Regression Neural Network“. Hierfür wurden neue, schnelle Trainingsverfahren entwickelt.
In dieser Arbeit wurde der Algorithmus ausschließlich an computer-generierten Beispielen getestet. Das Ziel ist aber auch der Einsatz des TST–Algorithmus auf experimentelle Probleme. Ein Beispiel aus der Praxis wäre das Optimieren von Peptiden auf Bindung an ein Target: Als Sequenz dient die Aminosäuresequenz des Peptids; die Fitness wäre dann die physicochemisch eindeutig zu messende Bindungskonstante des Moleküls.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In der modernen Biotechnologie ist das Design von Molekülen mit maßgeschneiderten Eigenschaften eine der großen Herausforderungen. Das dabei eingesetzte Methodenspektrum teilt sich in zwei grundsätzlich verschiedene Prinzipien auf: Zum einen das sog. „rationale Design“, bei dem versucht wird, mittels immer detaillierterer Modelle der Struktur-Aktivitäts–Beziehung neue Moleküle zu generieren. Auf der anderen Seite stehen die Methoden des „irrationalen Designs“, die durch die Imitierung von Grundzügen der biologischen Evolution — wie Mutation und Selektion — Molekülpopulationen optimieren. Eine wichtige Rolle spielen bei beiden Methodenklassen die Informationen durch Massenscreening (HTS) von Molekülen.

Die vorliegende Arbeit entstand in der Arbeitsgruppe „Systemintegration“ von Andreas Schober. Die Arbeitsgruppe entwickelt im Rahmen des „NanoSynTest“–Projekts in Kooperation mit der Fa. Merck KGaA eine Maschine (weiter), die im Nanolitermaßstab sowohl synthetisieren als auch die Effizienz der Moleküle durch Assays detektieren kann [96, 97, 98]. Es wird dabei auf Ergebnisse des Vorgängerprojekts aufgebaut. Die Synthese findet auf Polystyrol-Perlen in etwa 5000 Kammern auf einem Silizium-Wafer mit 4 inch Durchmesser statt. Die Kammern werden mit Pipettierrobotern befüllt, die nach dem Tintenstrahldrucker-Prinzip arbeiten. Die Detektion erfolgt optisch durch gekühlte, hochempfindliche CCD-Kameras.
Die Maschine wird neben der Synthese auch zum Massenscreening vorhandener Bibliotheken auf eine vorgegebene Funktion eingesetzt werden. Ziel des Ansatzes ist einerseits, durch Miniaturisierung und Automatisierung den Probendurchsatz zu erhöhen, andererseits durch Optimierung und geschickte Integration die Effizienz des Moleküldesigns zu erhöhen sowie die Anzahl Verfahrensschritte zu reduzieren.

In der vorliegenden Arbeit wurden Optimieralgorithmen entwickelt, die diesen

Ansatz unterstützen und sich für den Einsatz in der Biotechnologie eignen. Alle Algorithmen wurde unter der Prämisse entwickelt, den experimentellen Aufwand zu minimieren. Außerdem sollen sie für den Benutzer als „black box"–Verfahren arbeiten, d.h. es sind keine algorithmus-spezifischen Parameter einzustellen. Die Algorithmen reagieren auf die wenigen vom Benutzer festzulegenden Parameter robust und vorhersagbar.

Der **Doping–Algorithmus** ermöglicht, die Anzahl der Aminosäuren für eine Codon-Position einer Proteinbibliothek durch den Einsatz von Zusatzwissen zu reduzieren. Der Algorithmus berechnet für jede Codon–Position Nukleotid–Mischungen, die beim Einsatz in einem DNS–Synthesizer einen Pool an DNS–Sequenzen erzeugen, deren zugehörige Proteine den vom Experimentator vorgegebenen Randbedingungen genügen. Frank Wirsching, Rüdiger Dietrich und Andreas Schwienhorst setzten den Doping–Algorithmus bereits erfolgreich ein [130].

Der nach den Erfindern Dirk **T**omandl, Andreas **S**chober, Marcel **T**hürk benannte **TST–Algorithmus** ist ein neuer Typ an Optimierverfahren, der durch statistische Analyse der Abbildung zwischen bekannten Daten und ihren Fitnesswerten neue Datenpunkte mit einer möglichst hohen Fitness vorschlägt. Der Algorithmus versucht, mit einer möglichst geringen Anzahl an Stichproben aus dem Suchraum (lokale) Optima mit möglichst hoher Fitness zu detektieren. Tests mit computer-generierten Beispielen waren sehr erfolgversprechend. Zukünftig soll der TST–Algorithmus iterativ mit der Maschine zur Molekülоptimierung eingesetzt werden. Konventionelle Optimierverfahren sind dafür ungeeignet, da sie entweder eine spezielle Datenstruktur voraussetzen oder zu viele Stichproben benötigen. Im Extremfall entspricht jeder Stichprobe eine chemische Synthese.
Der TST–Algorithmus versucht, mit einem Minimum an *modellbehafteter* Information zu optimieren. Rationales Moleküldesign geht von Modellen der Struktur-Aktivitäts–Beziehung aus. Damit lassen sich präzise Vorhersagen berechnen, die allerdings aufgrund der Unzulänglichkeiten der Modelle häufig fehlerhaft sind. Der TST–Algorithmus arbeitet nach dem umgekehrten Ansatz: *Ohne* Modell, nur mittels statistischer Analyse der Struktur-Aktivitäts–Beziehung Vorhersagen zu treffen, die wesentlich unpräziser sind, aber mit höherer Wahrscheinlichkeit richtig.

Ein wichtiger Bestandteil des TST–Algorithmus ist das „**General Regression Neural Network**" (GRNN) von Donald Specht. Das GRNN ist ein nichtlineares Regressionsverfahren und wird mit neuen, hier entwickelten Trainingsverfahren eingestellt. Es approximiert einen Trainingsdatensatz mit vergleichbarer Genauigkeit wie hoch optimierte Backpropagation–Netze, ohne deren Nachteile aufzuweisen.

Das GRNN erwies sich als leistungfähiges Werkzeug, das das Potential besitzt, in Zukunft die mit vielen Nachteilen behafteten Backpropagation–Netze abzulösen.

Die Dissertation setzt beim Leser nur geringe Vorkenntnisse voraus; alle zum grundlegenden Verständnis notwendigen Informationen sind in den jeweiligen Kapiteln und Anhängen erklärt. Der Autor verfolgte die Absicht, daß auch eher experimentell orientierte, an praktischer Anwendung interessierte Leser die wesentlichen Aussagen erfassen können.

Die Arbeit ist in voneinander weitgehend unabhängige Kapitel aufgeteilt. Deshalb besitzt jedes Kapitel eine eigene Einleitung sowie Zusammenfassung und Ausblick.

Zunächst wird in Kapitel 2 der „General Optimization Shell“ (GOS) vorgestellt, in den sich Optimiermethoden, wie der Doping–Algorithmus oder der TST–Algorithmus standardisiert integrieren lassen.

In Kapitel 3 wird der Doping–Algorithmus eingehend beschrieben. Eine dazu passende Zusammenfassung der Grundlagen von Proteinen und Proteindesign findet sich in Anhang C.

Das General Regression Neural Network (GRNN) mit neuen, eigens entwickelten Trainingsverfahren wird in Kapitel 4 behandelt. Die tabellierten Ergebnisse des Tests des Netzes mit einem Benchmark–Test sind in Anhang A zusammengefaßt. Zum tieferen Verständnis des GRNN mag die Einführung in künstliche neuronale Netze in Anhang D dienen.

Kapitel 5 beschreibt den TST–Algorithmus und zeigt seine Leistungsfähigkeit anhand einiger Testbeispiele auf. Das Kapitel setzt Grundkenntnisse über das GRNN voraus.

Kapitel 2

General Optimization Shell — GOS

Thema des Kapitels ist ein *verallgemeinerter Rahmen für Optimierverfahren*, der sog. „**G**eneral **O**ptimization **S**hell“ — GOS. Der GOS soll als Grundlage bzw. als äußere Hülle für beliebige Optimieralgorithmen dienen.

2.1 Struktur des GOS

Der GOS ist für sich genommen *kein* eigener Optimieralgorithmus, sondern gibt lediglich die äußere Struktur für Optimierverfahren vor:

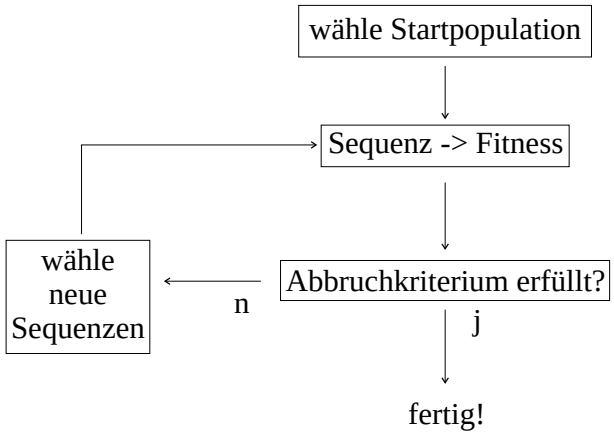

Abbildung 2.1: *Verallgemeinerter Rahmen für Optimierverfahren — GOS*

Wie in Abbildung 2.1 zu erkennen, ist der GOS zunächst nur eine Einschränkung der möglichen Struktur von Optimieralgorithmen. Der Vorteil des GOS liegt darin,

daß alle notwendigen Elemente eines Optimieralgorithmus nur einmal implementiert werden müssen und beliebig modular austauschbar sind. So kann die Abbildung *Sequenz* $\rightarrow$ *Fitness* ohne Schwierigkeiten ausgelagert werden, um sie experimentell durchzuführen. Zusätzlich stehen noch Routinen zur Verfügung, mit denen der zeitliche Verlauf der Optimierung sowie der zeitliche Verlauf der Zusammensetzung der Population untersucht werden kann.

Allen Optimierverfahren sind folgende Eigenschaften gemeinsam:

1. Jedes Optimierverfahren setzt eine eindeutige *Codierung* zwischen dem Problem und den das Problem charakterisierenden Parametern voraus. Bei vielen Optimierverfahren wird dieser Schritt nicht explizit gegangen, sondern lediglich mit einem Tupel reeller Parameter gearbeitet. Die Begriffe „Sequenz“, „String“, „Punkt“, „Chromosom“ oder „Individuum“ werden in dieser Arbeit synonym gebraucht und stehen für einen codierten Parametersatz.

   Der GOS läßt dem Benutzer völlige Freiheit in der Wahl der Codierung. Insbesondere müssen die Daten nicht sequenziell oder als Strings konstanter Länge vorliegen.

2. Den Strings bzw. den dort codierten Parametern muß ein reeller Funktionswert, die sog. „*Fitness*“ eindeutig zugeordnet werden können. Die Fitness- oder Gütefunktion ist ein numerisches Maß dafür, wie gut ein String das Problem löst. Ziel der Optimierung ist es, einen Parametersatz zu finden, dessen Fitnesswert je nach Aufgabenstellung minimal oder maximal ist und damit das gestellte Problem optimal löst.

   Für den GOS gilt o.B.d.A.[1] die Festlegung, daß der Wert der Fitnessfunktion umso *größer* sein muß, je *besser* das Problem durch den String löst wird. Der beste String hat demnach maximale Fitness; der GOS löst also Maximierungsprobleme. Ein Minimierungsproblem kann auf triviale Weise durch Multiplikation der Fitness mit dem Faktor -1 in ein Maximierungsproblem umgewandelt werden. Sinnvoll für den Optimiererfolg ist die Implementation einer streng monotonen Fitnessfunktion durch den Benutzer.

3. Jeder Optimieralgorithmus benötigt ein *Abbruchkriterium*. Übliche Kriterien sind das Überschreiten einer vorgegebenen Anzahl Iterationen oder das Unterschreiten der Fitnessänderungen unter einen Schwellwert.

[1] o.B.d.A. $\equiv$ **o**hne **B**eschränkung **d**er **A**llgemeinheit

Diese Kriterien sind beim GOS bereits voreingestellt, können aber jederzeit geändert werden.

Das grundlegende Schema des GOS ist in Abbildung 2.1 dargestellt. Die Module „*wähle Startpopulation*“, „*Sequenz → Fitness*“ und „*Abbruchkriterium erfüllt?*“ sind in ihrer Struktur in allen Algorithmen ähnlich aufgebaut. Der eigentliche Algorithmus des jeweiligen Verfahrens steckt im Modul „*wähle neue Sequenzen*“.

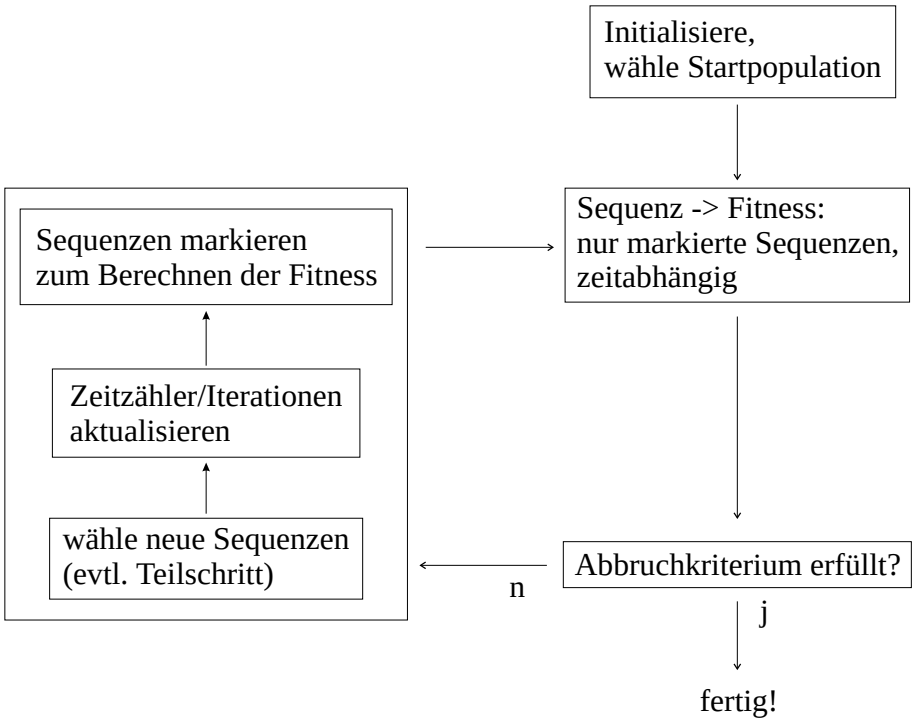

Abbildung 2.2: *Verallgemeinerter Rahmen für Optimierverfahren — Details*

In Abbildung 2.2 wird das verallgemeinerte Optimierschema etwas detaillierter vorgestellt:
Der GOS arbeitet mit einer Population variabler Größe. Jedes Mitglied der Population kann einen unterschiedlichen Typ (des Chromosoms) besitzen und mit seiner eigenen Fitnessfunktion arbeiten. Interne Zählvariablen erlauben die Darstellung einer individuellen Zeit. Damit können die Fitnessberechnungen asynchron erfolgen. Dies ist notwendig, falls eine einzelne Iteration des Optimierverfahrens mehrere Fitnessberechnungen als Zwischenschritte benötigt. Eine einzelne Iteration kann also unter Umständen mehrere Durchläufe des Zyklus in Abbildung 2.2 bedeuten. Durch die Einbeziehung einer individuellen Zeit könnten z.B. auch Populationsdynamiken studiert werden. Die Module des GOS bieten Unterstützung zur Verwaltung zeitabhängiger Fitnessberechnungen. Außerdem ist ein „caching“–Algorithmus implementiert, der — auf Wunsch — bereits berechnete Fitnesswerte zwischenpuffert.

Dies bietet, je nach Aufwand bei der Bestimmung der Fitness, enorme Geschwindigkeitsvorteile.

2.2 Prinzip der Implementation

Der GOS wurde als Klassenbibliothek[2] in C++ definiert, in die die konkreten Algorithmen standardisiert integriert werden können.

Das Grundprinzip soll hier als Liste dargestellt werden:

- `class_simple_individual`: Elementare Basisklasse, aus der sich die Population zusammensetzt. Die Klasse besteht aus einem Feld von `void`–Zeigern, die auf Bestandteile des Chromosoms des Individuums zeigen.

- `class_fitness`: Basisklasse für alle Fitnessfunktionen. Die Klasse bekommt als Argument einen Zeiger auf `class_simple_individual` und ist verantwortlich für alle Manipulationen des Chromosoms wie Speichermanagement, genetische Operatoren (Mutationen, Crossover etc.) und lokales Optimieren. Es gibt keinerlei Einschränkungen bezüglich der Struktur der Strings oder der Fitnessfunktion. Die Fitnessfunktion kann z.B. zeitabhängig oder gar durch ein Experiment zu bestimmen sein. Des weiteren ist ein Caching-Algorithmus implementiert, der auf Wunsch Strings-Fitness–Paare zwischenspeichert, für die die Fitness bereits bestimmt wurde. Bei einem erneuten Aufruf der Fitnessfunktion für diesen String muß der Fitnesswert nicht noch einmal berechnet werden, sondern wird aus dem Cache-Speicher gelesen. Für den TST–Algorithmus hat der Cache noch eine besondere Verwendung (siehe Kapitel 5.3.2).

- `class_individual`: Abgeleitet von `class_simple_individual`. Durch einen Zeiger auf `class_fitness` besitzt jedes Individuum seine eigene Fitnessfunktion. Für den Benutzer scheint es, als könne sich jedes Individuum selbst manipulieren, da in `class_fitness` alle Routinen zum Verwalten der Datenstrukturen abgelegt sind.

[2]Eine Bemerkung zum objektorientierten Programmieren in C++ :
Die Grundidee ist — als Erweiterung zum prozeduralen Programmieren — die Verschmelzung von Daten und Unterprogrammen zu Objekten oder Klassen. Entscheidend ist, daß analog zur Biologie, Nachfahren von Objekten definiert werden können. Die abgeleiteten Objekte übernehmen die Eigenschaften ihrer Vorfahren und können leicht um neue erweitert werden. Der Vorteil ist, daß gerade bei großen Projekten die Programme übersichtlicher und leichter umzustrukturieren sind.

- `class_population`: Verwaltet dynamisch eine Population aus Individuen und abgeleiteten Klassen von `class_individual`. Es ist möglich, eine Population aus völlig verschiedenen Individuen zusammenzusetzen, da jede Variable vom Typ `class_individual` ihre eigene Fitnessfunktion und eigenes Chromosom haben kann.

- `class_choose_new_individuals`: Diese Klasse entspricht dem Abschnitt „*wähle neue Sequenzen*“ in Abbildung 2.1. Sie enthält die konkrete Implementation des jeweiligen Optimieralgorithmus und wird von `class_general_optimize` benötigt. Die Klasse wählt (neue) Sequenzen aus, für die das Modul „*Sequenz* $\rightarrow$ *Fitness*“ die Fitness bestimmen soll. Außerdem muß sie eventuelle Zeitabhängigkeiten verwalten sowie die Anzahl Iterationen mitzählen.

- `class_general_optimize`: Diese Basisklasse ist die Umsetzung des GOS (siehe Abbildungen 2.1 und 2.2). Sie benötigt einen Zeiger auf eine vollständig initialisierte Population (`class_population`) sowie einen Zeiger auf eine Variable des Typs `class_choose_new_individuals`. Diese Klasse kann also in einer Population aus beliebigen Individuen mit einem beliebigen Optimierverfahren optimieren! Als konkrete Applikationen wurden eine eigene Variante eines genetischen Algorithmus und diverse lokale Optimieralgorithmen programmiert.

Mit der vorgestellten Konstruktion lassen sich alle Optimierverfahren implementieren, wie die Beispiele in Kapitel 2.3 demonstrieren werden.

2.3 Beispiele für Optimierverfahren

Mit der Vorstellung von Grundprinzipien einiger Optimierverfahren soll gezeigt werden, daß sich alle bekannten Klassen an Optimieralgorithmen problemlos in den Rahmen des GOS integrieren lassen.

Zur Vermeidung von Mißverständnissen:
In allen Listings dieser Arbeit in Pseudo-Programmcode werden nur die für den jeweiligen Algorithmus wesentlichen Bestandteile aufgeführt. Die Listings sind also als Skizze zu verstehen.

Alle dargestellten Algorithmen sind in der Lage, Minimierungs- und Maximierungsprobleme gleichermaßen zu lösen. Falls leichte Unterschiede zwischen beiden

Varianten bestehen, wird nur die Version für Maximierungsprobleme vorgestellt. Zum leichteren Verständnis der folgenden Beispiele soll hier die Struktur des GOS nach Abbildung 2.1 als Pseudo-Programmcode dargestellt werden:

```
Initialisiere Startpopulation;
do
    Berechnung der Fitness der (modifizierten) Sequenzen;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz;
    endif
while noch nicht fertig;
```

2.3.1 Lokale Optimieralgorithmen

Die üblichen *lokalen Optimiermethoden*, wie Gradientenverfahren, die Klasse der Conjugate Gradient Methoden [38, 84] (z.B. Levenberg-Marquardt–Verfahren) oder dem in Kapitel 3 eingesetzten downhill-simplex–Verfahren [76] etc. arbeiten mit sehr geringen Populationsgrößen von wenigen Sequenzen. Eine der Sequenzen ist dabei die eine zu optimierende Sequenz, die in jeder Iteration durch eine gleichgute oder bessere ersetzt wird. Die übrigen Sequenzen benötigt der jeweilige Algorithmus intern. Die Struktur lokaler Optimieralgorithmen läßt sich auf einfache Weise an die der GOS anpassen:

```
Initialisiere Startpopulation;
do
    Berechnung der Fitness der (modifizierten) Sequenzen;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz — Teilschritt;
    endif
while noch nicht fertig;
```

Im einfachsten Fall wählt der jeweilige Algorithmus als neue Sequenz eine bessere oder die beste Sequenz in unmittelbarer Umgebung der zu optimierenden Sequenz. Da die Auswahl der neuen Sequenz meist die Berechnung der Fitness von anderen (internen) Sequenzen voraussetzt, bedeutet dies, daß eine einzelne Iteration des Algorithmus aus mehreren Iterationen der GOS bestehen kann.

Das Modul „*wähle neue Sequenz*“ ist verantwortlich für die Verwaltung und Durchführung der Teilschritte einer Iteration und dem Übergang zur nächsten Iteration.

2.3.2 Evolution und evolutive Verfahren

Ein Grundverständnis der Frage, wie die Natur Optimierprobleme löst, ist erst seit CHARLES DARWIN's bahnbrechendem Werk „On the Origin of Species by Means of Natural Selection“ [15, 16, 17] von 1859 möglich. Danach haben sich alle Lebewesen über lange Zeiträume hinweg aus primitiveren Arten entwickelt. DARWIN erkannte als erster den zugrunde liegenden Mechanismus „Mutation und Selektion“.

Die Entwicklung der Evolutionstheorien durchlief mehrere Stufen: DARWIN ging von der „Arterhaltung“ aus. Der nächste Schritt durch R.A. FISHER [37] (1930) und G.C. WILLIAMS [129] (1966) betonte die Rolle des Individuums. Der aktuelle Erkenntnisstand des „egoistischen Gens“ wurde von W.D. HAMILTON [48, 49] (1964) und JOHN MAYNARD SMITH [111] (1972) erarbeitet und von RICHARD DAWKINS [20, 21] (1976) im gleichnamigen Buch massiv propagiert. Mit „Gen“ ist hier nicht ausschließlich das biologische Gen, das für ein Protein codiert, gemeint, sondern eine abstraktere, sequenzielle, kompakte Informationseinheit[3]. Unverzichtbare theoretische Basisarbeit leisteten in den 70er und 80er Jahren MANFRED EIGEN und PETER SCHUSTER mit der Entwicklung des „Hyperzyklus“ [29, 30, 31] und dem fundamentalen Konzept der „Quasispezies“ [27, 28]. MANFRED EIGEN gelang es auf molekularbiologischer Ebene, die DARWIN'schen Prinzipien quantitativ aus einem physikochemischen Modell abzuleiten und die Möglichkeit von Höherentwicklung und Evolution nachzuweisen. Ein weiterer Meilenstein ist die Theorie der „neutralen Evolution“ (1983) von MOTOO KIMURA [62].

Verglichen mit der Komplexität natürlicher Evolution und der zu ihrer Beschreibung notwendigen Theorien erscheinen die evolutiven Optimierverfahren vergleichsweise einfach. In der Regel ignorieren experimentelle wie theoretische evolutive Optimierverfahren beispielsweise diploide Chromosomen, die räumliche Verteilung von Populationen mit Nischenbildung oder die Zeitabhängigkeit der Fitness in der Natur. Zwei Spezies, die den gleichen Lebensraum beanspruchen, verändern durch ihre (gegenseitige) Anpassung die Fitness der jeweils anderen Spezies. Im Grunde reduzieren evolutive Optimieralgorithmen die biologische Evolution auf ein einfaches Gerippe. Nach KARL SIGMUND [107] (sinngemäß):

> Jedes System, das aus einer *Konstruktionseinheit* (z.B. Zellapparat) und einer *Instruktion*, die die Konstruktionseinheit steuert (z.B. Gen) be-

[3]Nach Meinung des Autors dieser Dissertation ist diese „Informationseinheit“ vergleichbar mit den Schemata von JOHN HOLLAND (siehe Gleichung (2.1) auf Seite 12).

steht, ist automatisch in der Lage, sich zu optimieren. Voraussetzung ist, daß die Instruktion zu kleinen Änderungen/Fehlern fähig ist.

In diesem Unterkapitel geht es um zwei theoretische Optimerverfahren („Genetische Algorithmen“ (Kapitel 2.3.2.1) und „Evolutionsstrategien“ (Kapitel 2.3.2.2)) und ein experimentelles Optimierverfahren („Phage Display“ (Kapitel 2.3.2.3)).

In den letzten Jahrzehnten entwickelten sich im wesentlichen zwei Schulen an Computersimulationen von Evolution zur Lösung von Optimieraufgaben. Auf der einen Seite steht die deutsche Schule der Evolutionsstrategien um INGO RECHENBERG, die die biologische Evolution nur als Richtschnur für die Entwicklung von Optimieralgorithmen benutzen will (Kapitel 2.3.2.2). Auf der anderen Seite steht die amerikanische Schule der genetischen Algorithmen um JOHN HOLLAND und DAVID GOLDBERG, die sich stärker für die Frage interessiert, wie es der Evolution gelingt, Information zu codieren, zu verarbeiten und über Generationen hinweg weiterzureichen (Kapitel 2.3.2.1). Interessanterweise ignorieren die Anhänger beider Schulen weitgehend die jeweiligen Resultate, obwohl neuere, unabhängige Entwicklungen versuchen, die Vorteile beider Verfahren zu kombinieren. Ein lesenswerte Übersicht über Genetische Algorithmen und Evolutionsstrategien sowie viele Anwendungsbeispiele enthält das Buch von EBERHARD SCHÖNEBURG [100].

Im direkten Vergleich [53] zeigen Evolutionsstrategien eine starke Neigung, rasch zu lokalen Optima zu konvergieren. Genetische Algorithmen finden häufiger das globale Optimum, konvergieren allerdings langsamer.

2.3.2.1 Genetische Algorithmen

Genetische Algorithmen wurden bereits 1975 von JOHN HOLLAND entwickelt [54]; ihren Durchbruch erfuhren sie 1989 durch das bekannte Buch [43] von DAVID GOLDBERG.

Ein genetischer Algorithmus imitiert Grundzüge biologischer Evolution und arbeitet deshalb typischerweise mit statistischen Populationsgrößen von einigen Dutzend bis mehreren tausend binären Sequenzen. Der Anwender muß das Problem in binäre Gene oder Chromosomen codieren. Dies erfordert eine explizite Abbildung von Genotyp auf Phänotyp, dessen Fitness dann bestimmt wird. Meist werden beide Schritte nicht getrennt behandelt, sondern erfolgen während der Fitnessberechnung. Die binäre Codierung kann zu bestimmten Artefakten („hamming cliff“–Problem [44]) führen. Beispielsweise ist die Zahl 7 binär durch den String `0111`

codiert. Hätte nun die Zahl 8 eine höhere Fitness, so müßten alle 4 Bits invertiert werden, da die Zahl 8 durch 1000 codiert wird. Durch die Benutzung des GRAY-Codes [43] kann dieses Problem entschärft werden. Eine weitere Schwierigkeit ist, daß die einzelnen Bitpositionen nicht gleichwertig sind. Die problemspezifische Codierung erfordert deshalb große Sorgfalt.

Hinter dem Abschnitt „*wähle neue Sequenzen*“ verbirgt sich hier:

- Dünne die Population „etwas“ (evtl. indirekt proportional zur Fitness) aus.
- Wende Operatoren (z.B. Mutation, Crossover, Inversion, Deletion etc.) auf die Sequenzen (evtl. proportional zur Fitness) zur Replikation an.

Ein genetischer Algorithmus sähe in der Struktur des GOS wie folgt aus:

```
Initialisiere Startpopulation;
do
    Berechnung der Fitness der (modifizierten) Sequenzen;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz:
            Population ausdünnen;
            Genetische Operatoren zur Replikation proportional zur Fitness;
            Adaption von Parametern des Algorithmus (z.B. Mutationrate);
    endif
while noch nicht fertig;
```

Der Anwender hat die Möglichkeit, Zusatzwissen über die Struktur des Suchraums in die Konstruktion angepaßter Operatoren einfließen zu lassen. Der entscheidende Unterschied zwischen genetischen Algorithmen/Evolutionsstrategien und anderen Optimierverfahren sind diejenigen genetischen Operatoren, die aus zwei oder mehr Sequenzen durch Rekombination zwei oder mehr Nachkommen produzieren.

Das Schema-Theorem von JOHN HOLLAND [54]

$$m(h,t+1) \geq m(h,t) \cdot \frac{f(h)}{\bar{f}} \cdot \left[1 - p_c \, \frac{L(h)}{n-1} - p_m \, o(h)\right] \tag{2.1}$$

mit
$m(h,t)$: Häufigkeit des Schemas h zur Zeit t
$f(h)$: durchschnittliche Fitness des Schemas h
$\bar{f}$: durchschnittliche Fitness der Population
n : Länge einer Sequenz
p_c : Crossover–Wahrscheinlichkeit

$$
\begin{aligned}
p_m &: \text{Mutations–Wahrscheinlichkeit} \\
o(h) &: \text{Ordnung des Schemas } h \\
L(h) &: \text{definierende Länge des Schemas } h
\end{aligned}
$$

besagt, daß Schemata h mit kleiner definierender Länge $L(h)$, niedriger Ordnung $o(h)$ und überdurchschnittlicher Fitness $f(h)$ mit größerer Wahrscheinlichkeit überleben als andere. Schemata sind in der ursprünglichen Formulierung Strings aus den Symbolen 0, 1 und # (# symbolisiert „don't care"), lassen sich aber auch bei anderen Arten der Codierung finden. Unter der Ordnung $o(h)$ eines Schemas h versteht man die Anzahl Positionen, die nicht das Platzhaltersymbol # enthalten, z.B. gilt $o(\#, 1, 0, \#, \#, 0, \#) = 3$. Die definierende Länge $L(h)$ ist der Abstand zwischen der ersten und der letzten von # verschiedenen Position in h. So gilt $L(\#, 1, 0, \#, \#, 0, \#) = 4$.

Indirekte Aussagen des Schema-Theorems (2.1) sind:

- Die Teile eines Chromosoms, die inhaltlich zusammenhängende Informationen codieren, sollten eng zusammenliegen und nicht getrennt codiert werden.
- Die für die Fitness eines Chromosoms maßgeblichen Segmente sollten möglichst kompakt codiert werden, damit überdurchschnittlich gute Chromosomen eine hohe Überlebenswahrscheinlichkeit erhalten.

Genetische Algorithmen benutzen also kompakte Schemata mit überdurchschnittlicher Fitness, die DAVID GOLDBERG „building blocks" nennt. Die Codierung des Problems sollte deshalb die Bildung von building blocks ermöglichen und erleichtern.

2.3.2.2 Evolutionsstrategien

Die *Evolutionsstrategien* basieren auf einem Modell der Evolution, das in den sechziger Jahren von INGO RECHENBERG an der Technischen Universität Berlin entwickelt wurde [86]. Wichtige Erweiterungen und Verbesserungen stammen von HANS-PAUL SCHWEFEL [102, 103].

In der GOS-Struktur sind Evolutionsstrategien identisch mit genetischen Algorithmen. Unterschiede bestehen in der Codierung, der Art der Mutationen und der Auswahl der Nachkommen. Aus der Sicht der Evolutionsstrategen läßt sich jedes Optimierproblem in Vektoren reeller Zahlen codieren, im Gegensatz zu den binären Strings genetischer Algorithmen. Der Vorteil ist neben der kompakten Form die

direkte Kopplung zwischen Sequenzen und Problemparametern. Es gibt im Wesentlichen zwei verschiedene Arten der Auswahl von Nachkommen (Notation nach SCHWEFEL [102]):

- $(\mu + \lambda)$-Evolutionsstrategie [86]:
  μ Eltern bekommen λ mutierte Nachkommen, die besten μ überleben. Es muß gelten: $\lambda \geq \mu \geq 1$. Da zur Auswahl der μ Besten jeweils die Eltern *und* die Nachkommen gemeinsam bewertet werden, kann sich die Qualität des Besten von Generation zu Generation nie verschlechtern. Diese Eigenschaft erhöht die Wahrscheinlichkeit der vorzeitigen Konvergenz zu einem lokalem Optimum. Deshalb hat HANS-PAUL SCHWEFEL in seiner Dissertation [102] die Komma-Notation (μ, λ) eingeführt.

- (μ, λ)-Evolutionsstrategie [102]:
  Im Gegensatz zur $(\mu+\lambda)$-Evolutionsstrategie werden die μ Besten nur noch aus den λ Nachkommen selektiert, die Eltern werden ignoriert. Hierbei wird die biologische Evolution besser modelliert, da es keine (höheren) unsterblichen Individuen gibt. Nun ist es aber möglich, daß sich die Fitness des Besten von einer Generation zur nächsten wieder verschlechtert. Gleichzeitig wird stärker global optimiert[4].

Spielt die Selektionsstrategie keine Rolle, so werden beide Varianten unter der Bezeichnung $(\mu\#\lambda)$-Evolutionsstrategie zusammengefaßt.

Wie bei genetischen Algorithmen existieren auch bei Evolutionsstrategien Variationen der sexuellen Rekombination: Die $(\mu/\rho\#\lambda)$-Evolutionsstrategie erzeugt zunächst λ Gruppen à ρ mutierte Nachkommen. In jeder Gruppe werden die Mitglieder miteinander rekombiniert und daraus ein Nachkomme ausgewählt.

Aus einem Elternteil wird ein mutierter Nachkomme dadurch erzeugt, indem der reelle Vektor des Elternteils mit einem Zufallsvektor addiert wird. Die Komponenten des Zufallsvektors sind unabhängige GAUSS-verteilte Zufallzahlen mit Mittelwert 0 und Standardabweichung σ. Jede Komponente kann dabei ihre eigene Standardabweichung haben [86].

[4]Diese häufige, etwas legere Formulierung sagt präzise aus, daß das Optimierverfahren mit höherer Wahrscheinlichkeit zum globalen Optimum konvergiert und seltener eines der lokalen Optima der Endpunkt der Optimierung ist.

2.3.2.3 Phage Display

Die *Phage Display*–Technik [70, 104, 110] ist das einzige rein experimentelle Optimierverfahren[5] in diesem Kapitel. Ziel des Verfahrens ist das Optimieren von Peptiden auf eine vorgegebene Funktion hin. Eine Übersicht über die Grundlagen von Proteinen und deren Optimierung kann in Anhang C nachgelesen werden.

Phage Display setzt einen vergleichsweise einfachen Algorithmus experimentell um. Kernpunkt ist die Genotyp-Phänotyp–Kopplung durch Phagen. Phagen sind virusähnliche Partikel, die im wesentlichen aus DNS und einer Proteinhülle bestehen. Im Genom des Phagen befindet sich eine zusätzliche, durch den Experimentator eingefügte DNS-Sequenz, deren zugeordnetes Protein an der Außenseite des Phagen exprimiert wird. Es wird mit einer sorgfältig zusammengestellten Startbibliothek an DNS-Sequenzen (siehe Kapitel 3) begonnen. Der Punkt „*Sequenz* $\rightarrow$ *Fitness*“ entspricht experimentell dem Einschleusen der Sequenzen in die Phagen, die die entsprechenden Proteine synthetisieren. Die Fitness der Proteine hängt von dem Optimierziel ab. Wird z.B. auf Bindung an einen Liganden optimiert, können die Phagen mit einer Säule nach der Bindungsstärke des Proteins selektiert werden. Nimmt der Experimentator die besten 10% aller Phagen, so ist dies äquivalent der Zuordnung des Fitnesswerts 1 zu den „besten“ DNS–Sequenzen. Alle anderen Sequenzen haben den Wert 0. Im Abschnitt „*wähle neue Sequenzen*“ werden die (besten) DNS–Sequenzen aus den Phagen extrahiert und durch fehlerhafte PCR (Punktmutationen) hochverstärkt.

In der Struktur der GOS nähme Phage Display folgende Form an:

```
Wähle Startbibliothek;
do
    Bestimmung der Fitness der Sequenzen:
        Einschleusen der Sequenzen in Phagen;
        Selektieren der „besten“ Protein-Phagen–Paare;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz:
            Extrahieren DNS aus den „besten“ Protein-Phagen–Paaren;
            Hochverstärken der DNS–Sequenzen durch fehlerhafte PCR;
    endif
while noch nicht fertig;
```

[5] Aus Platzgründen soll hier auf die Darstellung von „*Seriell Transfer*“ und „*SELEX*“ verzichtet werden. Das algorithmische Grundprinzip (verrauscht hochverstärken, dann die Besten selektieren) wird bei allen experimentellen Optimierverfahren iterativ angewandt.

2.3.3 Simulated Annealing

Simulated Annealing oder „simuliertes Ausglühen“ wurde 1983 von SCOTT KIRKPATRICK *et al.* vorgestellt [63] und basiert auf Ideen der statistischen Physik.

Simulated Annealing ist ein stochastisches Optimierverfahren. Neue Sequenzen entstehen aus den alten durch Verrauschen in Abhängigkeit von dem Systemparameter „Temperatur“ T. Ein Einzelschritt des Verfahrens für eine einzelne Sequenz fängt mit einer Kopie dieser Sequenz an, die stochastisch kleine Fehler besitzt. Anschließend wird der Wert ΔE :=Fitness(*neue Sequenz*)-Fitness(*alte Sequenz*) berechnet. Falls ΔE größer Null ist, also die neue Sequenz eine höhere Fitness als die alte besitzt, andernfalls mit der Wahrscheinlichkeit $e^{-\frac{\Delta E}{T}}$, wird die alte Sequenz durch die neue ersetzt. Simulated Annealing läßt sich wie folgt in die Struktur der GOS bringen:

```
Initialisiere Startpopulation;
wähle Starttemperatur T > 0;
verdoppelt_flag := false;
do
    Berechnung der Fitness der (modifizierten) Sequenzen;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz:
            if verdoppelt_flag = false
                then
                    verdopple jede Sequenz mit kleinen Fehlern;
                    verdoppelt_flag := true;
                else
                    forall kopierte Sequenzen
                        ΔE := Fitness(neue Sequenz)-Fitness(alte Sequenz);
                        if (ΔE ≥ 0) or ((ΔE < 0) and (random < e^(-ΔE/T)))
                            then lösche alte Sequenz;
                            else lösche kopierte Sequenz;
                        endif
                    endforall
                    verdoppelt_flag := false;
            endif
            erniedrige T;
    endif
while noch nicht fertig;
```

random steht hier für einen Pseudo-Zufallszahlengenerator, der gleichmäßig verteilte Werte im Intervall $[0;1]$ liefert. Die boolsche Variable `verdoppelt_flag` dient

dazu, zwischen den zwei Modi — verdopple Sequenzen und lösche Sequenzen — hin- und herzuschalten.

Je höher die Temperatur T ist, desto größer ist die Wahrscheinlichkeit, daß auch sehr schlechte Sequenzen übernommen werden. Dies ist in der Anfangsphase des Algorithmus sinnvoll, um überhaupt in die Nähe eines guten Optimums zu gelangen. In der Endphase muß die Temperatur verhältnismäßig klein sein, damit die gefundenen Optima nicht wieder verlassen werden. In den meisten Anwendungen wird der Erfolg des Algorithmus sehr empfindlich von dem zeitabhängigen Verlauf der Temperatur $T(t)$ beeinflußt. Häufige „annealing schedules“ sind ein linear abnehmender Verlauf, $T(t) \sim \frac{1}{t}$ oder auch $T(t) \sim \sin(t)$.

Ursprünglich ist der Algorithmus von SCOTT KIRKPATRICK für eine einzelne Sequenz formuliert worden. In der Praxis wird dagegen meist mit einer Population von Sequenzen gearbeitet, die unabhängig voneinander nach der Simulated Annealing–Regel optimiert werden. Dies ist bereits im GOS–Listing berücksichtigt. In Unterschied zu evolutiven Verfahren wechselwirken hier die Sequenzen nicht miteinander — Simulated Annealing könnte alternativ jeweils hintereinander mit jeder einzelnen Sequenz gestartet werden.

2.3.4 Threshold Accepting- und Sintflut–Algorithmus

GUNTER DUECK und TOBIAS SCHEUER entwickelten als Varianten von Simulated Annealing die Methode des Threshold Accepting [26] und den Sintflut–Algorithmus.

Threshold Accepting akzeptiert im Gegensatz zu Simulated Annealing *jede* kopierte Sequenz, deren Fitness nicht „wesentlich“ schlechter ist als die der alten Sequenz. Das folgende Listing ist nahezu identisch zum GOS-Listing von Simulated Annealing, deshalb werden nur die Unterschiede dargestellt:

```
...
wähle Start-Schwelle T > 0;
do
    ⋮
                        if ΔE > −T
                            then lösche alte Sequenz;
                            else lösche kopierte Sequenz;
                        endif
    ⋮
while noch nicht fertig;
```

Die Schwelle T wird in Laufe der Iterationen langsam erniedrigt. Im Fall $T = 0$ können keine schlechteren Sequenzen übernommen werden, der Algorithmus optimiert nur noch lokal.

Der *Sintflut–Algorithmus* ist noch einfacher aufgebaut:
Er akzeptiert jede kopierte Sequenz, solange ihr Fitnesswert einen *absoluten* Schwellwert T (den „Wasserstand“) nicht unterschreitet. Der Schwellwert wird langsam erhöht.

Trotz ihrer einfachen Struktur sind beide heuristischen Verfahren in der Praxis überraschend erfolgreich.

2.3.5 Tabu Search

Das Basiskonzept von Tabu Search stammt aus dem Jahr 1977 von FRED GLOVER [41, 42] und ist im Grunde eine Meta-Heuristik, die um eine weitere Heuristik (z.B. einem lokalen Suchalgorithmus) als eine Art Hülle gelegt wird, mit der Absicht zyklische Bewegungen des lokalen Algorithmus im Suchraum zu verhindern. Dies geschieht dadurch, daß neue Sequenzen bestraft werden oder gar verboten sind („Tabu-Liste“), die der lokale Algorithmus bereits bewertet hat. Tabu Search akzeptiert eine neue Sequenz mit schlechterer Fitness nur dann, wenn dadurch die Chance besteht, bereits begangene Pfade im Suchraum zu vermeiden. Damit wird die Wahrscheinlichkeit erhöht, daß neue Regionen des Suchraums untersucht werden. Als GOS-Listing ist Tabu Search nur eine geringe Erweiterung des ursprünglichen Zyklus:

```
Initialisiere Startpopulation;
do
    Berechnung der Fitness der (modifizierten) Sequenzen;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz:
            Teilschritt des jeweiligen Algorithmus;
            Tabu Search: Akzeptieren oder verbieten der neuen Sequenz;
    endif
while noch nicht fertig;
```

Die verschiedenen problemspezifischen Implementationen des Tabu Search unterscheiden sich in der Größe, Variabilität und Anpassungsfähigkeit des Tabu-Speichers.

2.3.6 Branch-and-Bound

Die Ursprünge des *Branch-and-Bound*–Verfahrens gehen auf eine Arbeit [13] der Autoren G.B. DANTZIG, D.R. FULKERSON und S.M. JOHNSON aus dem Jahr 1954 zurück. Die Autoren lösten damit ein Travelling-Salesman–Problem.

Wie der Name schon sagt, unterteilt sich die Technik des *Branch-and-Bound* in

- Branching (Verzweigung): Die zu lösende Aufgabe wird in zwei oder mehr disjunkte Teilaufgaben zerlegt, die keine gemeinsame Lösung besitzen. Dasselbe geschieht wiederum mit jeder entstehenden Teilaufgabe, so daß sich ein Baum aus Teilaufgaben ergibt. Zusammen haben die Teilaufgaben dieselbe Lösungsmenge wie die Ausgangsaufgabe.
- Bounding (Beschränkung): Wenn man sicher sein kann, daß die optimale Lösung einer Teilaufgabe nicht besser als eine schon bekannte Lösung ist, braucht die Teilaufgabe nicht weiter betrachtet zu werden. Man spricht davon, daß sie *ausgelotet* ist. Um dies festzustellen, schätzt man den höchstens erreichbaren Gewinn, eine *obere Schranke* für die betrachtete Teilaufgabe, ab.

Um diese Methode erfolgreich einsetzen zu können, ist eine detaillierte Kenntnis der Struktur des Suchraums erforderlich. Branch-and-Bound ist für jedes Problem gesondert zu entwickeln. Insbesondere gilt dies für die Teilschritte „Zerlegung in Teilaufgaben“, „Abschätzung des höchsten erreichbaren Gewinns“ und „Wahl des Wegs durch den Baum (eher horizontal oder vertikal)“.

Branch-and-Bound wird vor allem in den Bereichen Transport-, Touren-, Produktions-, Projekt- und Investitionsplanung eingesetzt.

Eng verwandt mit Branch-and-Bound ist *dynamisches Programmieren*, wie es z.B. im `Vienna RNA Package` [52] eingesetzt wird. Grundlage ist ebenfalls die Zerlegung einer Aufgabe in Teilaufgaben. Voraussetzung ist, daß sich eine optimale Lösung aus optimalen Teillösungen zusammensetzt. Diese Annahme führt zu Rekursions–Beziehungen der Teillösungen. Die eigentliche Lösung wird dann rekursiv „bottom–up“ aus den Teillösungen berechnet.

2.3.7 HOPFIELD–Netze und BOLTZMANN–Maschinen

Künstliche neuronale HOPFIELD–Netze von JOHN HOPFIELD sind in einer Variante als BOLTZMANN–Maschinen ebenfalls als Optimierverfahren einsetzbar. Bei-

spielsweise fand JOHN HOPFIELD einen Weg, mit HOPFIELD–Netzen Travelling-Salesman–Probleme zu lösen [56]. BOLTZMANN–Maschinen werden mit einen Verfahren trainiert, daß große Ähnlichkeit mit Simulated Annealing hat.

In den Anhängen D.2.7 und D.2.7.2 ab Seite 173 werden HOPFIELD–Netze hergeleitet sowie der Weg beschrieben, wie damit Optimierprobleme gelöst werden können.

2.3.8 Ant–System

Der *Ant*–Algorithmus [25] von MARCO DORIGO *et al.* ist ein neues (1996) und originelles Optimierverfahren, das durch die Funktionsweise von Ameisenstaaten inspiriert wurde.

Ein intensiv studiertes Problem in der Biologie war die Frage, wie einfach aufgebaute, fast blinde Ameisen in der Lage sein können, den kürzesten Weg zwischen Bau und Futterquelle zu finden. Eine zentrale, alles kontrollierende Intelligenz war nicht ausfindig zu machen, deshalb mußte die Lösung woanders zu finden sein. Der Schlüssel lag in einer *Pheromon-Spur*, die jede der Ameisen hinterläßt. Gleichzeitig bevorzugt eine Ameise diejenige Spur mit der höchsten Pheromon-Konzentration.

Es seien zwei verschieden lange Wege zwischen zwei Punkten gegeben. Anfänglich werden beide Wege mit der gleichen Wahrscheinlichkeit betreten. Da der kürzere Weg in der gleichen Zeit von mehr Ameisen zurückgelegt werden kann, wird sich dort mit der Zeit eine höhere Pheromon-Konzentration ausbilden, wodurch noch mehr Ameisen diesen Weg wählen werden. Diese positive Rückkopplung führt sehr schnell zur Konvergenz zum kürzeren der beiden Wege. Sind beide Wege gleich lang, so führen bereits minimale statistische Schwankungen in der Anzahl der Ameisen, die die beiden Wege zu Beginn betreten, zur Auswahl eines der Wege.

Der Ant–Algorithmus setzt diesem Mechanismus auf abstrakte Weise um. Insbesondere haben die künstlichen Ameisen Gedächtnis, sind nicht komplett blind und leben in einer Umgebung mit diskreten Zeitschritten. Das Optimierproblem muß immer in ein Wege-Auswahl–Problem umgeformt werden, d.h. als Graph repräsentierbar sein. Aus diesem Grund wurde der Ant–Algorithmus bis jetzt hauptsächlich auf Travelling-Salesman–Probleme und ähnliche eingesetzt. MARCO DORIGO zeigte im Computerexperiment, daß der Ant–Algorithmus eine vergleichbare Qualität an Lösungen wie Simulated Annealing erzielt.

2.3.9 TST–Algorithmus

Der Kapitel 5 vorgestellte *TST–Algorithmus* startet mit einer beliebig großen Startpopulation. Jedem Mitglied der Population wird zunächst seine Fitness zugeordnet. Anschließend wird die Abbildung $Sequenz \rightarrow Fitness$ statistisch analysiert. Auf der Basis der ermittelten Eigenschaften werden neue Sequenzen vorgeschlagen, die mit großer Wahrscheinlichkeit eine hohe Fitness besitzen werden.

```
Initialisiere Startpopulation;
do
    Bestimmung der Fitness der (neuen) Sequenzen;
    if Abbruchkriterium erfüllt
        then fertig!
        else wähle neue Sequenz:
            Statistische Analyse der Abbildung Sequenz → Fitness;
            Auswahl neuer Sequenzen auf der Basis der statistischen Eigenschaften;
    endif
while noch nicht fertig;
```

Eine detaillierte Darstellung des TST–Algorithmus sowie eine Reihe von Beispielen finden sich in Kapitel 5.

2.4 Testbeispiele für Optimierverfahren

An dieser Stelle sollen einige Testbeispiele vorgestellt werden, die entweder häufig in der Literatur Verwendung finden und/oder in dieser Arbeit eingesetzt wurden.

2.4.1 Reelle Funktionen — RASTRIGIN–Funktion

Viele Autoren testen ihre Optimieralgorithmen an Funktionen, deren Definitions- wie auch Wertebereich reell ist. Unter Umständen werden die Koordinaten in einen ganzzahligen Bereich wie [0;512] oder [0;65535] transformiert.

Als Beispiel dient die RASTRIGIN–Funktion in N Dimensionen:

$$Rastr(x_1, \ldots, x_N) := \sum_{i=1}^{N} \left[A \cos(2\pi x_i) - x_i^2 \right] - N \cdot A \tag{2.2}$$

In den meisten Simulationen wird die Konstante $A := 10$ gesetzt, so auch in der vorliegenden Arbeit. Für die Koordinaten gilt: $x_i \in [-a; +a]$, mit $a \in \mathbb{N}$. Die RASTRIGIN–Funktion hat damit $(2a)^N$ lokale Maxima. In allen Simulationen dieser

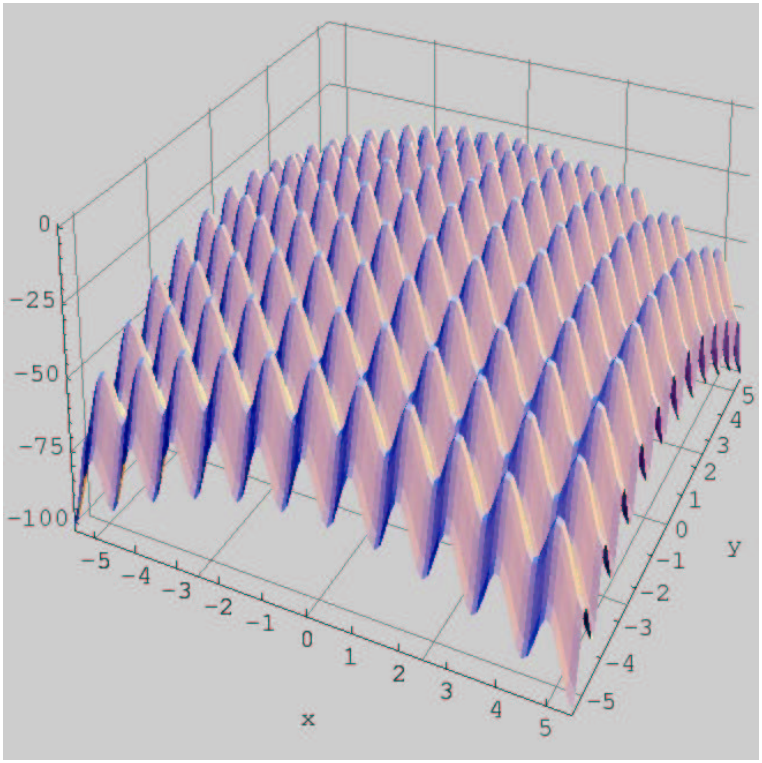

Abbildung 2.3: RASTRIGIN–*Funktion in zwei Dimensionen*

Arbeit wird $a = 5$ gesetzt. Das globale Maximum liegt an der Stelle $(0,\ldots,0)$ und hat den Wert 0.

Zur Veranschaulichung wird in Abbildung 2.3 die RASTRIGIN–Funktion in zwei Dimensionen dargestellt. Es ist deutlich zu erkennen, wie die umgedrehte Parabel von den Cosinus-Schwingungen überlagert wird, wodurch die Vielzahl lokaler Extrema entsteht. Die RASTRIGIN–Funktion gilt in der Literatur als schwieriges Optimierproblem.

2.4.2 Spinglas

Spingläser [118] sind spezielle Festkörper, die komplizierte magnetische Eigenschaften aufweisen. Man kann eine Art von Spinglas beispielsweise dadurch herstellen, daß man ein nichtmagnetisches Metall (wie Kupfer) mit wenigen Atomen dotiert, deren magnetische Momente von Null verschieden sind (wie Eisen). Damit wirkt etwa eine Hälfte der Atompaare ferromagnetisch und die andere antiferromagnetisch. In dieser Situation können sich die Spins nicht so einstellen, daß alle Wechselwirkungen gleichzeitig abgesättigt werden. Ein derartiges System bezeichnet man als *frustiert*. Gleichzeitig bedeutet dies, daß es für ein Spinglas mehrere tiefliegende

Energiezustände geben kann. Oder andersherum: Für einen gegebenen Energiezustand kann es mehrere Spineinstellungen geben.

Nach ROKHSAR *et al.* existieren bei N Spins etwa $e^{0.2N}$ lokale energetische Minima [88]. R. PALMER [80] definierte eine rauhe („rugged“) Fitnesslandschaft durch einen (mindestens) exponentiellen Anstieg der Anzahl lokaler Optima in Abhängigkeit von der Systemgröße N. Im Sinne dieser Definition sind Spinglas-Landschaften rauh und damit schwierige Probleme für Optimierverfahren.

Frustierte Ehefrauen stellen für Optimieralgorithmen schwierige Probleme dar, da es viele Extrema gibt, deren Fitness sehr ähnlich oder gar gleich groß ist. Viele praktische Optimierprobleme sind ebenfalls frustiert, wenn die Zwangsbedingungen nicht alle gleichzeitig erfüllt werden können und es deshalb viele erlaubte Lösungen ähnlicher Fitness gibt, die scheinbar nicht miteinander zusammenhängen.

Der Hamiltonian (die freie Spinglas–Energie in willkürlichen Einheiten) läßt sich durch die paarweise Wechselwirkung aller N Spins berechnen:

$$E(s_1, \ldots, s_N) := \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} W_{ij}\ s_i\ s_j \tag{2.3}$$

W_{ij} ist hierbei die symmetrische Wechselwirkungsmatrix mit $W_{ii} = 0$. Der Betrag eines Elements der Wechselwirkungsmatrix gibt die Stärke der Wechselwirkung an; ein positiver Zahlenwert entspricht ferromagnetischer, ein negativer antiferromagnetischer Wechselwirkung. Die Spins s_i können die Werte -1 und $+1$ annehmen. Da nach Gleichung (2.3) *alle* Spins paarweise miteinander interagieren können, entspricht dies einem Modell mit langreichweitiger Wechselwirkung.

Optimierziel ist dasjenige Tupel an Spins mit einer vorgegebenen, der minimalen oder der maximalen freien Energie. Für jeden beliebigen Satz an N Spins s_i gilt nach Gleichung (2.3):

$$E(s_1, \ldots, s_N) = E(-s_1, \ldots, -s_N) \tag{2.4}$$

Dies bedeutet, daß die Fitnesswerte für ein Tupel an Spins und sein inverses Tupel identisch sind. Dadurch existieren immer Paare an gleichberechtigten (lokalen und globalen) Optima. Außerdem halbiert sich die Größe des Suchraums von 2^N auf 2^{N-1} Sequenzen an Spins.

Im Computerexperiment wird die Wechselwirkungsmatrix meist mit gleichverteilten reellen Zufallszahlen aus dem Intervall $[-a; +a]$ besetzt. In dieser Arbeit wurde immer $a = 5$ gewählt.

2.4.3 HAMMING–Abstand

Der HAMMING–Abstand definiert sich als die Anzahl *unterschiedlicher* Positionen zwischen zwei Sequenzen $s^{(1)}$ und $s^{(2)}$:

$$d\left(s^{(1)}, s^{(2)}\right) := N - \sum_{i=1}^{N} \delta_{s_i^{(1)}, s_i^{(2)}} \tag{2.5}$$

$$\text{mit} \quad \delta_{i,j} := \begin{cases} 1, & \text{für } i = j \\ 0, & \text{für } i \neq j \end{cases}$$

Eine einfache Fitnesslandschaft läßt sich durch den HAMMING–Abstand zu einer vorgegebenen Referenzsequenz konstruieren. Gesucht ist die Zielsequenz mit dem maximalen HAMMING–Abstand zur Referenzsequenz.

Hat die Referenzsequenz beispielsweise an jeder Position den Wert 0 stehen, so ist jede Position der Zielsequenz mit 1 besetzt. Die auf diese Weise definierte Landschaft hat exakt ein globales Optimum und keine lokalen Extrema. Sie kann als erster, einfach zu lösender Test eines Optimierverfahrens dienen.

2.4.4 RNS-Sekundärstruktur

RNS-Moleküle formen in wässrigen Lösungen spontan dreidimensionale Strukturen, sofern strukturstabilisierende Kationen wie Mg^{2+} in der richtigen Konzentration vorliegen und pH-Wert und Temperatur im geeigneten Bereich liegen. Treibende Kraft ist die Bildung der WATSON-CRICK–Paare G≡C und A=U sowie G−U.

Da die Modelle zur Vorhersage der dreidimensionalen RNS-Struktur noch sehr unzuverlässig sind, beschränkt man sich bei theoretischen Untersuchungen meist auf die Sekundärstruktur der minimalen freien Energie. Die Sekundärstruktur erfaßt wesentliche Eigenschaften der räumlichen Struktur und dient daher den meisten Algorithmen zur Vorhersage der Tertiärstruktur als Grundlage. Eine wichtige Eigenschaft der RNS-Sekundärstrukturlandschaften ist die Redundanz der Sequenz → Struktur–Abbildung. Es existieren wesentlich weniger Strukturen als Sequenzen. Zudem verteilt sich die Anzahl der Strukturen nach dem ZIPF'schen Gesetz, d.h. es gibt einige Strukturen, die häufig und viele Strukturen, die selten vorkommen. Eine Übersicht findet sich in [101].

Es wurden bereits eine Reihe von Computerprogrammen publiziert, die aus der RNS-Sequenz die Sekundärstruktur mit der minimalen freien Energie vorhersagen. In dieser Arbeit wurde das `Vienna RNA Package` [52] aus der Wiener Gruppe um Prof. PETER SCHUSTER eingesetzt.

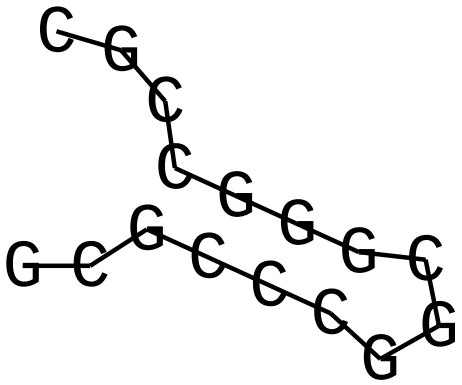

Abbildung 2.4: *Sekundärstruktur mit minimaler freier Energie der RNS-Sequenz* `GCGCCCGGCGGGCCGC`

In Abbildung 2.4 ist als Beispiel die Sekundärstruktur der minimalen freien Energie einer 16 Nukleotide kurzen RNS-Sequenz abgebildet.

Es lassen sich nun mehrere unterschiedliche Fitnesslandschaften definieren. Jeder Sequenz wird über den Umweg der Sekundärstruktur ein Fitnesswert zugeordnet:

- Als Fitness wird eine Funktion der minimalen freien Energie verwendet.
- Der Strukturabstand zwischen der Sekundärstruktur der Sequenz und einer Zielsekundärstruktur dient als Fitness.
- Einzelnen Strukturelementen, die mit bestimmten (biologischen/chemischen) Funktionen in Verbindung gebracht werden, können Bestrafungs- und Belohnungspunkte zugeordnet werden.

2.4.5 Travelling-Salesman

Es seien N Städte und eine Kostenfunktion $d(S_1, S_2)$ der Reise zwischen jedem Städtepaar (S_1, S_2) gegeben. Gesucht ist nun die Tour eines Geschäftsmanns, die unter minimalen Gesamtkosten

$$K(p) := \sum_{i=1}^{N-1} d\left(S_{p(i)}, S_{p(i+1)}\right) + d\left(S_{p(N)}, S_{p(1)}\right) \tag{2.6}$$

genau einmal durch jede Stadt läuft und zum Ausgangspunkt zurückkehrt. p ist eine Permutation der Indizes {1,2,...,N}, die die Reihenfolge der Städte codiert.

Es existieren exakt $\frac{(N-1)!}{2}$ verschiedene Touren, falls $d(S_1, S_2) = d(S_2, S_1)$ gilt. In diesem Fall spricht man vom *symmetrischen* Travelling-Salesman–Problem. Beim asymmetrischen Problem existieren $(N-1)!$ unterschiedliche Lösungen.

Das Travelling-Salesman–Problem gehört zur großen Klasse der NP-vollständigen Probleme, d.h. es existiert kein Algorithmus, der garantiert die exakte Lösung mit einem Rechenaufwand proportional zu einer Potenz von N findet.

In vielen praktischen Beispielen ist die Kostenfunktion einfach der EUCLID'sche Abstand $\sqrt{(x_1^{(1)} - x_1^{(2)})^2 + (x_2^{(1)} - x_2^{(2)})^2}$ zwischen je zwei Städten. Gesucht damit die *kürzeste* Tour durch alle Städte.

Kapitel 3

Der Doping–Algorithmus

In Kapitel 2 wurde ein Weg vorgestellt, beliebige Optimieralgorithmen in ein allgemeines Schema (GOS) unterordnen zu können. Als eine Anwendung der Klassenbibliothek des GOS wurde ein leicht modifizierter genetischer Algorithmus GALO (siehe Kapitel 3.2) implementiert. Mit diesem Werkzeug konnte in Zusammenarbeit mit Andreas Schwienhorst ein Problem aus dem Bereich der Biotechnologie erfolgreich gelöst werden [121].

Die Revolution der Molekularbiologie der letzten Jahre führte zu einer intensiven Untersuchung von Proteinen. Bislang mußte sich die Wissenschaft im wesentlichen auf die Entdeckung und Charakterisierung von natürlichen Proteinen beschränken. Viele dieser Proteine besitzen Funktionen, die von großem wirtschaftlichen Interesse sind. Die Proteine sind jedoch an ihre jeweilige Umgebung und ihre natürliche Aufgabe angepaßt. Da sich aber viele Bedingungen, unter denen Proteine Einsatz finden, von denen in der Natur unterscheiden, können nur wenige natürliche Proteine praktisch genutzt werden. Grundlagen von Proteinen und Methoden des Proteindesigns sind in Anhang C zusammengestellt.

Die Aufgabe des Doping–Algorithmus ist es, eine *gezielt eingeschränkte* Bibliothek an DNS–Sequenzen bzw. Proteinen zu generieren. Diesen Pool können evolutive Optimierverfahren wie Phage Display als Startbibliothek benutzen.

3.1 Problemstellung

Der Einsatz irrationaler, evolutiver Techniken entwickelt sich in der Biotechnologie zu einem wertvollen Werkzeug im Design von Polypeptiden und Proteinen mit

nützlichen, neuen Eigenschaften. Wie bei der Prinzipbeschreibung des Phage Display [70, 104, 110, 131] auf Seite 15 erwähnt, kommt der intelligenten Konstruktion der Startpopulation besondere Bedeutung zu. Ein möglicher, von RÜDIGER DIETRICH in der Gruppe von ANDREAS SCHWIENHORST verfolgter Ansatz ist es, Proteine aus funktionellen Modulen mittels „exon-shuffling“ zusammenzusetzen [24]. Homologe Rekombination wurde von WILLIAM P.C. STEMMER [119] entwickelt.

Ein anderer Ansatz ist es, zusätzliche Informationen zu nutzen, mit denen anstelle der 20 möglichen nur eine eingeschränkte Auswahl an Aminosäuren an jeder Proteinposition zugelassen wird. Als Zusatzinformationen können phylogenetische Daten, Strukturinformationen (Übersicht in [109]), Faltungsroutinen, Wildtypsequenzen [23, 72, 77], Translationseffizienz der mRNS und anderes Expertenwissen des Experimentators dienen. Beispielsweise wird man den Einbau von Prolin in α-Helizes vermeiden wollen. Ebenso sind STOP–Codone in der Praxis generell unerwünscht [68, 106]. Eine solche Auswahl an Aminosäuren bezeichnet man als Aminosäurebesetzung oder „Doping-Schema“.

Der hier vorgestellte Doping–Algorithmus berechnet die entsprechenden Nukleotidmischungen, mit denen die gewünschten Doping-Schemata konstruiert werden können. Er übersetzt für eine Aminosäureposition des Proteins die vom Benutzer gewünschte Mischung an Aminosäuren in Nukleotidmischungen zurück. Der Algorithmus liefert dabei für jede der drei Nukleotidpositionen im Codon eine eigene Mischung, also insgesamt drei verschiedene Mischungen der vier Mononukleotide. Werden diese Mischungen in einen DNS–Synthesizer eingesetzt, so entsteht ein Pool von DNS–Sequenzen, der für die gewünschte Aminosäuremischung codiert.

3.2 Der Algorithmus (GALO)

Der Optimieralgorithmus (GALO) besteht aus einer Kombinationen von **G**enetischem **A**lgorithmus [43, 54] (GA) und **L**okalem **O**ptimierverfahren, der „downhill-simplex“–Methode [76, 84]. Das downhill-simplex–Verfahren wurde gewählt, da es keine Gradienteninformation der zugrundeliegenden Landschaft benötigt. In der Literatur finden sich eine Reihe von Arbeiten (z.B. [79]), die den Nutzen einer solchen Kombination nahelegen. Die grundlegenden Eigenschaften genetischer Algorithmen wurden bereits in Kapitel 2.3.2.1 beschrieben, deshalb werden hier nur die Unterschiede zur Standard-Variante dargestellt.

Der grundsätzliche Ablauf des GALO–Algorithmus (Abbildung 3.1) sieht wie

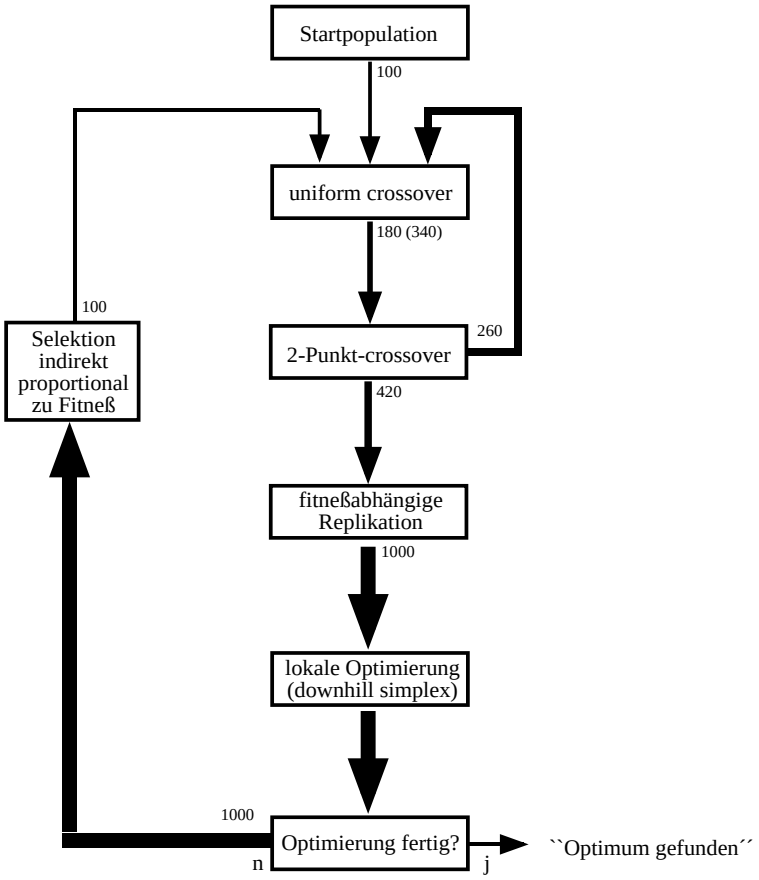

Abbildung 3.1: *Schema des Doping–Algorithmus*

folgt aus:

1. Beginne mit einer (kleinen) Population von zufällig ausgewählten Strings (100).

2. Wähle zwei Mitglieder der Population als Eltern und führe „uniform–crossover“ aus. Addiere die resultierenden „Kinder“ zur Population. Führe diesen Schritt 80 Male aus.

3. Wähle zwei Mitglieder der Population als Eltern und führe „Zwei-Punkt–crossover“ aus. Addiere die resultierenden „Kinder“ zur Population. Führe diesen Schritt 80 Male aus.

4. Wiederhole die Schritte 2 und 3 noch einmal.

5. Vermehre die Strings der Population proportional zu ihrer individuellen Fitness solange, bis die Population ihre maximale Größe (1000) erreicht. Laß die Vermehrung fehlerhaft mit einer individuellen Mutationrate sein.

6. Verwende die „downhill simplex–Methode“, um den jeweils fittesten String jeder Generation zu optimieren und ersetze mit dem gefundenen String den schlechtesten der Population.

7. Wenn das Ziel der Optimierung erreicht ist, „gebe Optimum aus“ und Ende des Algorithmus, andernfalls gehe zu Schritt 8.

8. Eliminiere solange Strings indirekt proportional zu ihrer Fitness, bis die (kleine) Startpopulationsgröße erreicht ist.

9. Gehe zu Schritt 2.

Insgesamt werden in jeder Iteration aus 100 Strings 1000 Strings gebildet. Von den 900 neuen Strings werden 320 durch crossover generiert, dies entspricht einer crossover-Rate von 35.5%. Die restlichen 580 Strings entstehen durch fehlerhafte Vermehrung proportional zur Fitness mit einer individuellen Mutationsrate nach Kapitel 3.5.1.

Um die frühzeitige Konvergenz in ein lokales Optimum zu erschweren, ist es für einen genetischen Algorithmus wichtig, in seiner Population eine gewisse Diversität beizubehalten. Aus diesem Grund wurden 10% der Strings der Startpopulation als „unsterblich“ festgelegt. Das Ziel der Optimierung kann auf viele Weisen definiert werden, da die zugrundeliegende Landschaft im allgemeinen unbekannt ist. Hier wurden drei Kriterien implementiert. Zum einen wird als optimale Lösung definiert, wenn der beste String für 10 Generationen unverändert bleibt, zum anderen, wenn ein bestimmter Anteil der Population den gleichen, besten String enthält. Außerdem wird der genetische Algorithmus nach einer bestimmten, vorzugebenden Anzahl Generationen beendet.

Im allgemeinen versucht ein genetischer Algorithmus ein (lokales) Optimum durch die Anwendung genetischer Operatoren auf eine Population von Strings und die Verstärkung mit einer Wahrscheinlichkeit proportional zu ihrer Fitness zu finden. Wie in den Kapiteln 2 und 2.3.2.1 ausgeführt, müssen bei praktischen Anwendungen genetischer Algorithmen drei wesentliche Probleme gelöst werden:

1. Codieren des Problems in Strings (Kapitel 3.3)

2. Fitnessfunktion der Strings (Kapitel 3.4 ab Seite 32)

3. Geeignete Operatoren für die Strings (Kapitel 3.5 ab Seite 36)

3.3 Codierung des Problems

Der Algorithmus behandelt ein einzelnes Codon während eines Laufes. Für jede der drei Codonpositionen müssen die vier Nukleotid-Konzentrationen[1] T_i, C_i, A_i, G_i $(i = 1, 2, 3)$, also $3 \cdot 4 = 12$ reelle Variablen optimiert werden. Da die Nukleotid-Konzentrationen jeder Codonposition normiert sind $(T_i + C_i + A_i + G_i = 1)$, reichen die Werte dreier Variablen z.B. T_i, C_i, A_i aus, um die vierte Variable G_i festzulegen. Die Anzahl der zu optimierenden Variablen pro Codonposition reduziert sich damit auf $3 \cdot 3 = 9$. Eine der Eigenschaften des genetischen Codes ist die Möglichkeit, alle 20 Aminosäuren durch Codone, deren dritte Position nur G und C enthält, darzustellen (siehe Tabelle C.1). Nutzt man diese Eigenschaft, so verbleiben $3+3+1 = 7$ Variablen.

$$Topf := (T_1, C_1, A_1, T_2, C_2, A_2, T_3, C_3, A_3) \tag{3.1}$$

oder

$$Topf := (T_1, C_1, A_1, T_2, C_2, A_2, C_3) \tag{3.2}$$

Nichtsdestotrotz existieren Aminosäuremischungen, für die bessere Lösungen durch eine Mischung aller vier Nukleotide an der dritten Position zu finden sind.

Bedingt durch die Eigenschaften des genetischen Codes können viele Aminosäuremischungen nicht durch eine Ein-Topf–Synthese generiert werden. Deshalb bietet der Algorithmus eine „split-and-mix"–Möglichkeit analog zum „mixed resin"–Ansatz [40] in der Peptidsynthese [65, 66, 105]. Die Grundidee ist, n voneinander unabhängige Ein-Topf–Synthesen durchzuführen und diese anschließend in den Mengenverhältnissen M_p $(p = 1, 2, \ldots, n)$ wieder zusammenzumischen. Im Extremfall wird jede Aminosäure in einem einzelnen Topf synthetisiert, die maximale Anzahl Synthesetöpfe ergibt sich damit zu $n_{max} = 20$.

Für n Synthesetöpfe ist ein String folgendermaßen definiert:

$$String := (\underbrace{M_1, \ldots, M_{n-1}}_{\text{Anteil}}, \underbrace{Topf_1, \ldots, Topf_n}_{n \text{ Synthesetöpfe}}) \tag{3.3}$$

Jede mögliche Lösung ist durch einen String bestehend aus $10n-1$ bzw. $8n-1$ reellen Zahlen codiert, je nachdem, ob an der dritten Codonposition A, T, C, G oder nur

[1] Da die Wahrscheinlichkeit des Antreffens eines Nukleotids proportional zu seiner Konzentration in der Mischung ist und die Nukleotid-Konzentrationen in den Bereich [0; 1] transformiert wurden, wird im folgenden alternativ auch von *Wahrscheinlichkeiten* gesprochen.

G, C zugelassen werden. Jeder String enthält noch drei zusätzliche Variablen, die die individuelle Mutationsrate für die Vermehrung festlegen.

In der Standard-Variante eines genetischen Algorithmus muß jede Variable binär codiert werden. Bei einer minimalen Auflösung von 16 Bits und beispielsweise $n = 3$ Töpfen ergäbe dies Strings der Länge $16 \cdot (10 \cdot 3 - 1 + 3) = 512$ Bits. Bei der maximalen Anzahl von $n = 20$ Töpfen wären die Strings 3232 Bits lang. Je länger Strings werden, desto langsamer konvergiert aus kombinatorischen Gründen die Population und umso leichter bleibt die Optimierung in lokalen Optima stecken. Ein weiterer Nachteil ist, daß die Bildung bestimmter Schemata erschwert wird, da die Grenzen zwischen den „natürlichen“ Blöcken — den reellen Variablen — verwischt werden. Andererseits konvergieren genetische Algorithmen in der Regel mit Strings aus Alphabeten niedriger Kardinalität am Besten („building block“–Hypothese [43, 44]), besitzen aber eine Neigung zu bestimmten Artefakten („hamming cliff“–Problem [44]). Eine etwas detailliertere Diskussion dieser Effekte, die sich aus dem Schema-Theorem ergeben, findet sich in Kapitel 2.3.2.1 und in [43, 54].

Da die direkte Codierung der Problemparameter (der Nukleotidkonzentrationen) auch aus Gründen der „Programmästhetik“ attraktiv ist, fiel die Entscheidung zugunsten reell-codierter Strings. Zur Vermeidung der gerade geschilderten Probleme, die ihre Ursache in extremer Kardinalität haben, wurden zwar reelle Strings, aber mit nur drei Nachkommastellen Genauigkeit verwendet. Ebenso beschleunigt der Einsatz lokaler Optimierung die Suche nach dem globalen Optimum.

3.4 Fitness eines Strings

Die Fitness eines Strings berechnet sich im Wesentlichen aus der Summe der Quadrate der Differenzen zwischen Ist- und Sollwert der Aminosäurekonzentrationen. Die STOP–Codone werden der Einfachheit halber als 21.Aminosäure gezählt.

Die Sollwerte der Aminosäurekonzentrationen sind vorgegeben — es fehlen also noch ihre Istwerte:

3.4.1 Umwandlung von Strings in normierte Wahrscheinlichkeiten

Wie bereits beschrieben, manipuliert der genetische Algorithmus die Strings beim Hochverstärken durch Punktmutationen und Crossover-Operatoren ohne Kenntnis

der internen Struktur der Strings. Die einzige automatisch erfüllte Randbedingung ist, daß keiner der Werte kleiner Null und größer Eins ist. Andere noch zu erfüllende Bedingungen sind, wie bereits erwähnt:

$$\sum_{p=1}^{n} M_p \stackrel{!}{=} 1 \tag{3.4}$$

$$T_i + C_i + A_i + G_i \stackrel{!}{=} 1 \ , \quad i = 1, 2, 3 \tag{3.5}$$

Das Normierprinzip ist in allen Fällen gleich und wird am Beispiel der Mengenverhältnisse M_p demonstriert: Es sei

$$s := \sum_{p=1}^{n-1} M_p \tag{3.6}$$

- Ist nun $s > 1$, werden die M_p $(p = 1, 2, \ldots, n-1)$ auf Eins durch $M_p \leftarrow \frac{M_p}{s}$ normiert und $M_n = 0$ gesetzt.
- Ist $s <= 1$, wird $M_n = 1 - s$ gesetzt, und die Bedingung (3.4) ist ebenfalls erfüllt.

3.4.2 Korrektur für reale Einbauwahrscheinlichkeit

In der Literatur gibt es widersprüchliche Angaben zu den Reaktionsraten der vier Standard-Phosphoramidite, wie sie in der automatischen DNS–Synthese verwendet werden. Einige Experimente deuten darauf hin, daß frisch zubereitete, equimolare Mischungen zu gleich häufigem Einbau der Nukleotide führen [60, 133]. Andere Forschungsgruppen [51, 58] beobachteten dagegen, daß das Phosphoramidit A häufiger als die übrigen Nukleotide C, G oder T eingebaut wird, während das *User Bulletin of Applied Biosystems* [1] das Phosphoramidit A als das am geringsten reaktive angibt. Aus derselben Quelle geht hervor, daß das Phosphoramidit G schneller als die anderen Nukleotide altert. Das Alter der Nukleotidmischungen ist also ein wichtiger Faktor für die Genauigkeit und Reproduzierbarkeit der DNS–Synthese. Außerdem scheinen die Reaktionsraten vom DNS–Synthesizer selbst abhängig zu sein.

Von JEREMY R. KNOWLES *et al.* werden in [51] folgende relative Wahrscheinlichkeiten (A auf Eins normiert) angegeben:

$$Korr_A = 1 \tag{3.7}$$

$$Korr_T = 0.602 \tag{3.8}$$

$$Korr_C = 0.565 \tag{3.9}$$

$$Korr_G = 0.528 \tag{3.10}$$

Sind die Wahrscheinlichkeiten T_i, C_i, A_i, G_i vorgegeben, müssen sie mit den entsprechenden Korrekturfaktoren (Gleichungen (3.7)–(3.10)) multipliziert werden. Die Summe $s = T_i + C_i + A_i + G_i$ muß mittels $T_i \leftarrow \frac{T_i}{s}$, $C_i \leftarrow \frac{C_i}{s}$, $A_i \leftarrow \frac{A_i}{s}$, $G_i \leftarrow \frac{G_i}{s}$ wieder auf Eins normiert werden.

Zusätzlich wurden Gewichtsfaktoren eingeführt, die eine gewünschte Codonusage berücksichtigen sollen. Experimente zur rekombinanten Protein Expression [5, 6, 33, 87, 92, 113] zeigen die Bedeutung der Codonusage. Standardgemäß sind diese Faktoren unter der Annahme, daß keine Korrekturen notwendig sind, auf Eins gesetzt.

3.4.3 Aminosäurebesetzung aus Nukleotidwahrscheinlichkeiten

Es seien die Wahrscheinlichkeiten T_i, C_i, A_i, G_i eines Codons gegeben. Die daraus resultierenden Auftretewahrscheinlichkeiten der Aminosäuren $P_{Aminos.}$ lassen sich leicht mit Hilfe eines Wahrscheinlichkeitsbaums und der Tabelle C.1 des genetischen Codes von Seite 152 berechnen:

$$P_{ALA} = G_1C_2T_3 + G_1C_2C_3 + G_1C_2A_3 + G_1C_2G_3 = G_1C_2 \tag{3.11}$$

$$\begin{aligned} P_{ARG} &= C_1G_2T_3 + C_1G_2C_3 + C_1G_2A_3 + C_1G_2G_3 + A_1G_2A_3 + A_1G_2G_3 \\ &= G_2(C_1 + A_1(A_3 + G_3)) \end{aligned} \tag{3.12}$$

$$P_{ASN} = A_1A_2T_3 + A_1A_2C_3 = A_1A_2(T_3 + C_3) \tag{3.13}$$

$$P_{ASP} = G_1A_2T_3 + G_1A_2C_3 = G_1A_2(T_3 + C_3) \tag{3.14}$$

$$P_{CYS} = T_1G_2T_3 + T_1G_2C_3 = T_1G_2(T_3 + C_3) \tag{3.15}$$

$$P_{GLU} = G_1A_2A_3 + G_1A_2G_3 = G_1A_2(A_3 + G_3) \tag{3.16}$$

$$P_{GLN} = C_1A_2A_3 + C_1A_2G_3 = C_1A_2(A_3 + G_3) \tag{3.17}$$

$$P_{GLY} = G_1G_2T_3 + G_1G_2C_3 + G_1G_2A_3 + G_1G_2G_3 = G_1G_2 \tag{3.18}$$

$$P_{HIS} = C_1A_2T_3 + C_1A_2C_3 = C_1A_2(T_3 + C_3) \tag{3.19}$$

$$P_{ILE} = A_1T_2T_3 + A_1T_2C_3 + A_1T_2A_3 = A_1T_2(1 - G_3) \tag{3.20}$$

$$\begin{aligned} P_{LEU} &= C_1T_2T_3 + C_1T_2C_3 + C_1T_2A_3 + C_1T_2G_3 + T_1T_2A_3 + T_1T_2G_3 \\ &= T_2(C_1 + T_1(A_3 + G_3)) \end{aligned} \tag{3.21}$$

$$P_{LYS} = A_1A_2A_3 + A_1A_2G_3 = A_1A_2(A_3 + G_3) \tag{3.22}$$

$$P_{MET} = A_1T_2G_3 \tag{3.23}$$

$$P_{PHE} = T_1T_2T_3 + T_1T_2C_3 = T_1T_2(T_3 + C_3) \tag{3.24}$$

$$P_{PRO} = C_1C_2T_3 + C_1C_2C_3 + C_1C_2A_3 + C_1C_2G_3 = C_1C_2 \tag{3.25}$$

$$\begin{aligned} P_{SER} &= T_1C_2T_3 + T_1C_2C_3 + T_1C_2A_3 + T_1C_2G_3 + A_1G_2T_3 + A_1G_2C_3 \\ &= T_1C_2 + A_1G_2(T_3 + C_3) \end{aligned} \tag{3.26}$$

$$P_{THR} = A_1C_2T_3 + A_1C_2C_3 + A_1C_2A_3 + A_1C_2G_3 = A_1C_2 \tag{3.27}$$

$$P_{TRP} = T_1G_2G_3 \tag{3.28}$$

$$P_{TYR} = T_1A_2T_3 + T_1A_2C_3 = T_1A_2(T_3 + C_3) \tag{3.29}$$

$$P_{VAL} = G_1T_2T_3 + G_1T_2C_3 + G_1T_2A_3 + G_1T_2G_3 = G_1T_2 \tag{3.30}$$

$$P_{END} = T_1G_2A_3 + T_1A_2A_3 + T_1A_2G_3 = T_1(G_2A_3 + A_2(A_3 + G_3)) \tag{3.31}$$

3.4.4 Fitness aus Aminosäurebesetzung

Der Experimentator gibt die Sollwahrscheinlichkeiten $S_{Aminos.}$ und Gewichtungsfaktoren $W_{Aminos.}$ aller 21 Aminosäuren vor.

Zur Berechnung der Fitness eines Strings muß der String in normierte Nukleotidwahrscheinlichkeiten und Mengenverhältnisse umgewandelt (Kapitel 3.4.1) und korrigiert (Kapitel 3.4.2) werden. Nach Bestimmung der Aminosäurebesetzungen (Kapitel 3.4.3) kann dem String die Fitness zugeordnet werden.

Die Fitnessfunktion ist negativ definiert, da der genetische Algorithmus auf maximale Werte optimiert, hier aber der Fehler der Aminosäurekonzentrationen minimiert werden soll.

$$F(String) := -\sum_{\mu=1}^{21} W_\mu \left[\sum_{p=1}^{n} M_p P_{p,\mu} - S_\mu\right]^2 \cdot \left(1 + \frac{0.1}{n}\sum_{p=1}^{n} \text{penalty}(M_p)\right) \tag{3.32}$$

$P_{p,\mu}$ entspricht dem berechneten Anteil der μ.Aminosäure im p.Synthesetopf (siehe Kapitel 3.4.3).

$\sum_{p=1}^{n} M_p P_{p,\mu}$ ist der *berechnete* Anteil der μ.Aminosäure im gesamten Synthesetopf.

S_μ ist der *gewünschte* Anteil der μ.Aminosäure im gesamten Synthesetopf.

W_μ ist der entsprechende Gewichtsfaktor für jede Aminosäure mit einem voreingestellten Wert von Eins. Mit den Gewichtungsfaktoren wird die Abweichung einzelner Aminosäuren vom Sollwert stärker betont. Wenn $W_{Aminos.}$ größer als Eins gesetzt ist, dann wird die betreffende Aminosäure genauer optimiert als die anderen, deren Fehler demzufolge etwas größer wird. Dies bietet sich beispielsweise für die STOP–Codone an, die am besten nie erscheinen sollen. Meist steigt dadurch allerdings der Betrag des Gesamtfehlers nach Gleichung (3.36) auf Seite 38 an.

DOUGLAS YOUVAN verwendete in [2] alternativ das Kriterium der Gruppen-Wahrscheinlichkeit $F = \prod_{\mu=1}^{21} P_\mu$. Deren Nachteil ist, daß damit nur Doping-Schemata mit gleichhäufigen Aminosäuren optimiert werden können.

Falls ein einzelner Synthesetopf nicht ausreicht, die gewünschte Aminosäurebesetzung in gewünschter Präzision herzustellen, muß eine Lösung mit der *minimalen* Anzahl von Synthesetöpfen gefunden werden, da der experimentelle Aufwand der „split-and-mix"–Technik mit steigender Anzahl von Töpfen anwächst. Ein Weg wäre es, den Doping–Algorithmus für eine steigende Anzahl von Synthesetöpfen zwischen 1 und 20 solange laufen zu lassen, bis ein optimaler Kompromiß zwischen Genauigkeit der Aminosäurebesetzung und experimentellem Aufwand erzielt ist. Dies ist algorithmisch unbefriedigend und erfordert einen hohen Rechenaufwand.

Der „penalty"–Term in Gleichung (3.32) ist der Versuch, bei vorgegebener maximaler Anzahl n von Synthesetöpfen Lösungen mit minimaler Anzahl zu bevorzugen. Die Idee ist es, Töpfe zu bevorzugen, deren prozentualer Anteil M_p am Gesamtmix entweder sehr klein oder sehr groß ist. Ein mittlerer Wert soll bestraft werden. Die Bestrafungsfunktion penalty(x) soll also für x-Werte um Null und Eins jeweils kleine Werte, für mittlere Argumente ungefähr Eins zurückliefern. In der aktuellen Version des Algorithmus ist die penalty–Funktion als nach unten geöffnete Parabel mit dem Scheitelpunkt (0.5, 1), die die geforderten Eigenschaften erfüllt, definiert:

$$\text{penalty}(x) := 4x(1-x) \ , \quad x \in [0;1] \tag{3.33}$$

$$\text{penalty}(0) = \text{penalty}(1) \ = \ 0 \tag{3.34}$$

$$\text{penalty}(0.5) \ = \ 1 \tag{3.35}$$

Die Funktion liefert im angegebenen Definitionsbereich Werte zwischen 0 und 1. Die maximale, durch den penalty–Term bedingte Reduktion der Fitness beträgt deshalb 10%.

3.5 Operatoren

Als genetische Operatoren wurden Mutation mit adaptiver Regelung der Mutationsrate sowie „Zwei-Punkt–Crossover" und „Uniform–Crossover" implementiert. Die Entwicklung weiterer, an das Problem speziell angepaßter Operatoren war nicht erforderlich.

3.5.1 Adaptive Mutationsrate

Ein Mutationsoperator für reelle Zahlen kann auf viele Weisen definiert werden. Die Implementation des GALO nutzt folgende Heuristik:

Der maximale Wertebereich der zu mutierenden reellen Zahl z wird in eine vorgegebene, feste Anzahl von Intervallen unterteilt. z liegt damit innerhalb eines der Intervalle. Ein *neues* Intervall wird mit der Wahrscheinlichkeit p_S gewählt; in diesem Fall bekommt z zufällig einen Wert aus diesem Intervall zugewiesen. Falls kein neues Intervall gewählt wurde, wird für z ein neuer Wert innerhalb des ursprünglichen Intervalls mit der Wahrscheinlichkeit p_V gewürfelt.

Da die Mutationsraten p_S und p_V ebenfalls reelle Zahlen sind, werden sie durch denselben Mechanismus unter Zuhilfenahme der „Mutationsrate der Mutationsraten“ p_M mutiert. Jeder einzelne String besitzt seinen eigenen Satz an Mutationsraten p_S, p_V und p_M. Damit können sich die individuellen Mutationsraten während der Optimierung selbst adaptieren.

Das Zeitverhalten des Mittelwertes $\frac{p_S+p_V}{2}$ der Mutationsraten jedes Mitglieds der Population eines typischen Programmlaufes wird in Abbildung 3.2 auf Seite 39 dargestellt.

3.5.2 Crossover

Beide Standardoperatoren „Zwei-Punkt–Crossover“ und „Uniform–Crossover“ bilden aus jeweils zwei Vorfahren zwei Nachkommen. Es zeigte sich, daß beide Operatoren ausreichend und hilfreich sind. Dies läßt darauf schließen, daß die zu erwartenden Schemata — die Synthesetöpfe — tatsächlich existieren.

Die Mutationsraten p_S, p_V und p_M der Nachkommen sind der Einfachheit halber als die Mittelwerte der beiden Eltern-Strings definiert.

3.6 Simulation statistischer Fehler

Die experimentelle Umsetzung der berechneten Dopings unterliegt statistischen Fehlern, wie beispielsweise Pipettierfehlern. Es ist deshalb wichtig, die Stabilität von optimierten Nukleotidmischungen in Abhängigkeit von (kleinen) Abweichungen der berechneten Werte zu untersuchen.

Die Annahme ist, daß die experimentell realisierten Aminosäurekonzentrationen eine GAUSS-Verteilung um die berechneten Konzentrationen formen. Als Maß für die Größe des statistischen Fehlers wird die Standardabweichung in der Simulation schrittweise von 0.5% bis 10% des jeweiligen theoretischen Wertes festgesetzt. Eine konstante Standardabweichung dagegen entspräche der Benutzung einer einzigen Pipette für alle Konzentrationen, die bei kleinen zu pipettierenden Mengen überproportional große Fehler hätte. Es wird hier also vorausgesetzt, daß der Experimentator für jeden Bereich die geeignete Pipette verwendet, der prozentuale Fehler immer gleich ist.

Für einen gegebenen Fehler (z.B. 0.5%) bestimmt die Simulation für jede berechnete Konzentration die zugehörige Standardabweichung. Anschließend wird für jede Konzentration ein neuer Wert entsprechend der GAUSS-Verteilung gewürfelt. Nachdem diese neuen Konzentrationen normiert wurden, kann die Aminosäureverteilung berechnet werden. Die daraus ableitbare *Summe der absoluten Fehler*

$$SAF(String) := \sum_{\mu=1}^{21} \left| \sum_{p=1}^{n} M_p P_{p,\mu} - S_\mu \right| \tag{3.36}$$

wird über 100 Programmläufe gemittelt. Falls zufällig bessere Lösungen als die eingangs berechnete gefunden werden, ersetzen diese die ursprüngliche. Diese Fehlersimulation kann deshalb auch als Qualitätskontrolle der Optimierung durch GALO sowie Feinkorrektur der Lösungen betrachtet werden.

3.7 Numerische Ergebnisse

Als erstes Testbeispiel wurde das Generieren einer equimolaren Mischung aller 20 Aminosäuren unter Vermeidung von STOP–Codonen gewählt [60].

Alle Berechnungen wurden auf einer SGI Indy R4400 (150 MHz) von Silicon Graphics durchführt. Die Ein-Topf–Variante (GALO1) benötigt etwa 2 Sekunden; die Drei-Topf–Version (GALO3) ist im Mittel nach etwa einer Minute beendet.

3.7.1 Performance des Algorithmus

Abbildung 3.2 stellt die typische Performance des genetischen Algorithmus mit und ohne lokale Optimierung einander gegenüber. Das Diagramm zeigt den zeitlichen Verlauf der Fitness (A,B) des besten Individuums und der mittleren Mutationsrate der gesamten Population (C,D).

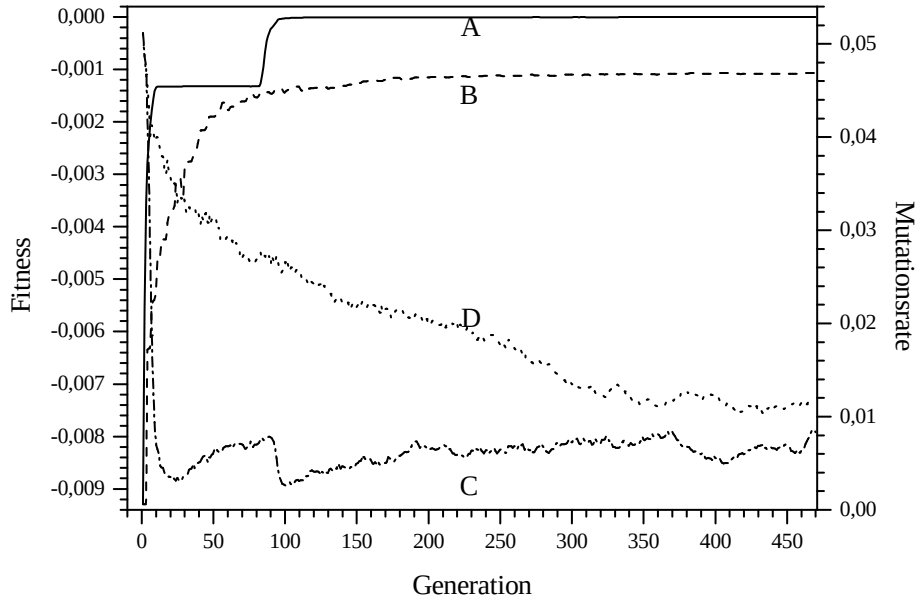

Abbildung 3.2: *Performance des Algorithmus am Beispiel der equimolaren Mischung aller 20 Aminosäuren.*

Ein Vergleich der zeitlichen Entwicklung der Fitness mit (A) und ohne (B) lokale Optimierung zeigt, daß die downhill-simplex–Methode eine wesentliche Verbesserung der Optimierung darstellt. Der genetische Algorithmus ohne lokales Optimieren benötigt deutlich mehr Generationen zum Erreichen des lokalen Optimums der Fitness -0.0013. In den weiteren Generationen gelingt es ihm (hier) nicht, das lokale Optimum zu verlassen.

Andererseits kann auf den genetischen Algorithmus nicht verzichtet werden, da sich das lokale Optimierverfahren alleine sehr schnell in einem lokalen Optimum fangen würde, wie die ersten Generationen in Abbildung 3.2 zeigen. In der Regel ist die absolute Fitness der Optima, die von dem genetischen Algorithmus mit und ohne lokales Optimieren gefunden werden, etwa gleich hoch, obwohl es Beispiele gibt, in denen GALO deutlich bessere Lösungen findet (siehe Abbildungen 3.2 und 3.3).

Der genetische Algorithmus zeigt eine deutliche Anpassung der mittleren Mutationsrate:
Mit steigender Anzahl Generationen sinkt die mittlere Mutationsrate der Population sowohl für den genetischen Algorithmus mit (C) als auch ohne (D) lokaler Optimierung. Für den genetischen Algorithmus mit lokaler Optimierung (C) kann ein erneuter Anstieg der Mutationsrate beobachtet werden, wenn sich die maximale Fitness einige Zeit lang nicht verbessert.

Je höher die Fitness eines Strings bereits ist, desto höher ist für ihn der Selekti-

onsdruck, die Mutationsraten p_S, p_V und p_M zu senken, damit die Nachkommen in seiner unmittelbarer Umgebung gebildet werden und das gefundene Optimum nicht sofort wieder verlassen. Dies erklärt den Verlauf von Kurve D, aber nicht, warum während der ersten 20 Generationen die mittlere Mutationsrate in Kurve C so viel schneller fällt, als in Kurve D. Der Grund ist, daß in jeder Generation der durch lokale Optimierung des besten Strings gefundene String automatisch eine vordefinierte minimale Mutationsrate zugewiesen bekommt, *falls* die lokale Optimierung eine Verbesserung der Fitness erbrachte. Solange die Population noch vom nächstgelegenen lokalen Optimum entfernt ist, bekommt in jeder Generation ein neuer String die minimale Mutationsrate zugewiesen — die mittlere Mutationsrate sinkt dadurch schnell. Die Mutationsraten der Kinder werden aus denen der Eltern gemittelt, d.h. die minimale Mutationsrate verteilt sich in der Population. Sie ist gleichzeitig eine untere Schranke für alle Mutationsraten, die von den mittleren Mutationsraten in Abbildung 3.2 nicht unterschritten werden können.

Sobald das lokale Optimum erreicht ist, bringt lokales Optimieren keinen Fortschritt mehr und es bekommt kein weiterer String mehr die minimale Mutationsrate zugewiesen. Die Anzahl Strings mit minimaler Mutationsrate nimmt durch Selektion wieder ab, und die mittlere Mutationsrate nimmt damit zu. Da ein String das lokale Optimum bereits erreicht hat, besteht aufgrund der Vermehrung proportional zur Fitness für die übrigen Strings der Population ein hoher Selektionsdruck, durch Erhöhung der Mutationsrate das Optimum schnell erreichen zu müssen.

3.7.2 Reproduzierbarkeit der Lösungen

Um die Reproduzierbarkeit des Optimierverfahrens sowie die Häufigkeit von Lösungen zu untersuchen, wurden am Beispiel der equimolaren Mischung aller 20 Aminosäuren 1000 unabhängige Programmläufe mit unterschiedlichen Startbedingungen durchgeführt.

Abbildung 3.3 zeigt die Verteilung der Lösungen in zwei verschiedenen Auflösungen. Im Allgemeinen verteilen sich die Lösungen auf wenige Cluster vergleichbarer Fitness bzw. Summe der absoluten Fehler SAF. Etwa 30% der Ergebnisse des GALO liegen im Cluster der optimalen Lösungen ($SAF \in [0\%; 1\%]$). Die Wahrscheinlichkeit für den reinen genetischen Algorithmus, Lösungen in demselben Intervall zu finden, ist nahezu Null.

Dies zeigt, daß die Kombination aus genetischem Algorithmus und lokalem Optimierverfahren zusammen mit dem in Kapitel 3.7.1 beschriebenen adaptiven Me-

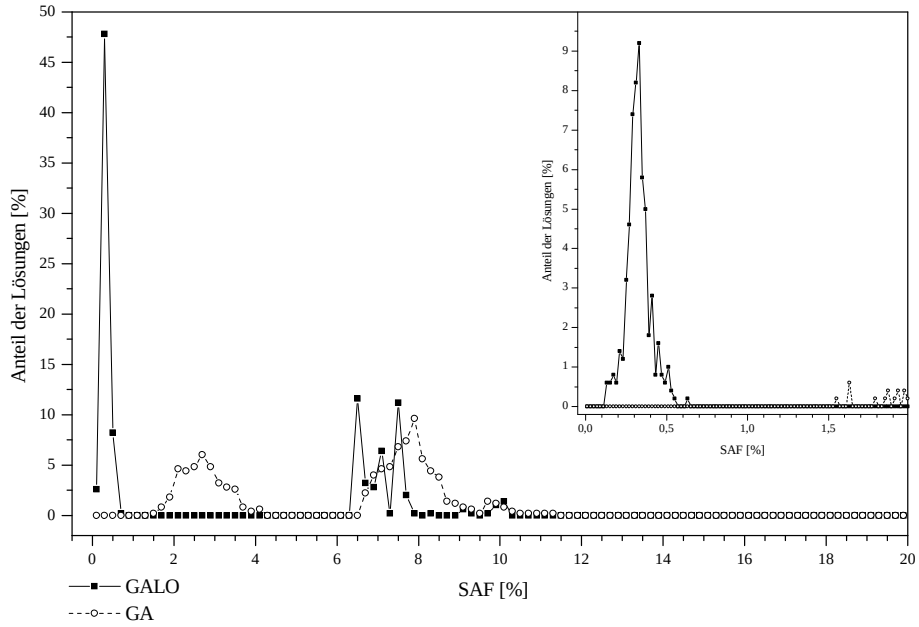

Abbildung 3.3: *Reproduzierbarkeit der Lösungen anhand des Beispiels der equimolaren Mischung aller 20 Aminosäuren.*

chanismus die Wahrscheinlichkeit erhöht, lokalen Optima zu entkommen und das globale Optimum zu erreichen.

3.7.3 Güte der Lösungen

Abbildung 3.4 vergleicht die besten Lösungen verschiedener Verfahren für das Problem der equimolaren Mischung aller 20 Aminosäuren unter Vermeidung von STOP–Codonen. Die verschiedenen Lösungen wurden durch den gängigen NN(G/C)–Ansatz ($SAF = 45.00\%$), eine intensive Computersuche durch alle möglichen Dopes mit einer Auflösung von 10% [2, 45] ($SAF = 32.40\%$), eine Ein-Topf– und eine Drei-Topf-„split-and-mix“–Variante des GALO ($SAF = 26.74\%$ bzw. $SAF = 0.19\%$) generiert.

Die verwendeten Methoden sind:

NN(G/C): An den ersten beiden Nukleotidpositionen des Codons werden alle vier Nukleotide T, C, A, G mit der gleichen Wahrscheinlichkeit 25% eingebaut. An der dritten Codonposition werden nur G und C zu gleichen Anteilen verwendet.

Aufwendige Computersuche (nach DOUGLAS YOUVAN [2]): Hierbei werden die Konzentrationen aller vier Nukleotide an den drei Positionen des Codons mit einer vorgegeben Genauigkeit durchgegangen, für jede Mischung die Aminosäureverteilung bestimmt und mit der vorgegebenen Verteilung verglichen. Wenn das Intervall

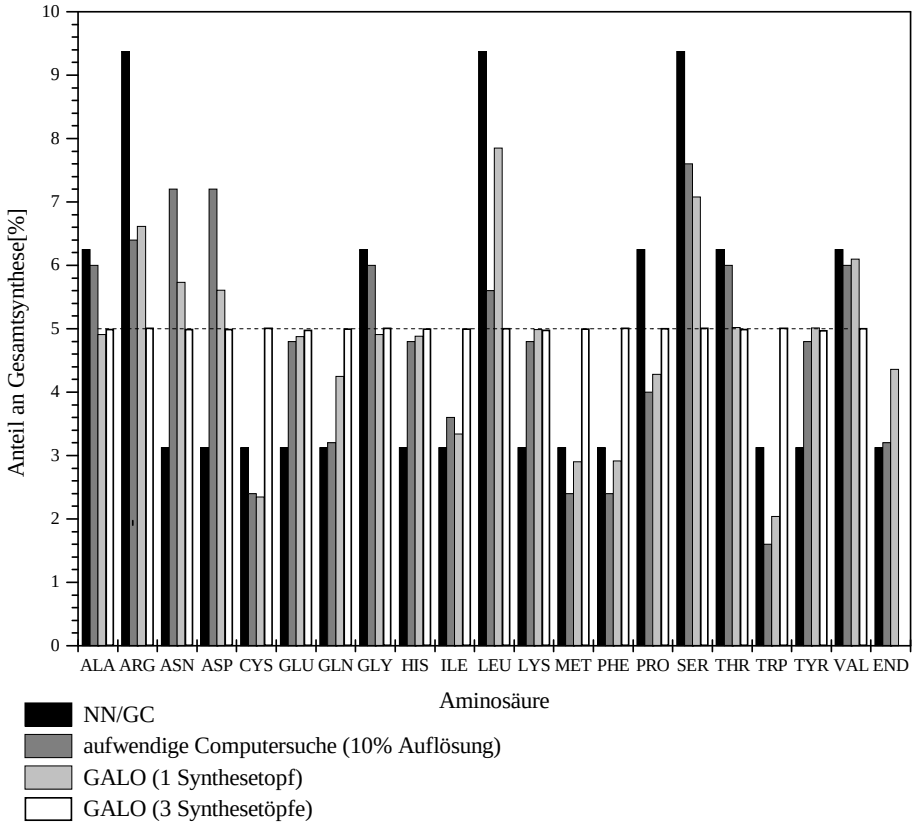

Abbildung 3.4: *Vergleich von Lösungen, die durch verschiedene Methoden gefunden wurden, am Beispiel der equimolaren Mischung aller 20 Aminosäuren.*

[0%;100%] für alle Nukleotide und alle Positionen in jeweils s Schritten abgetastet wird, so läßt sich unter Berücksichtigung der Normierbedingung zeigen, daß der Rechenaufwand von der Ordnung $O(s^9)$ ist: Es existieren genau $\left[\frac{(s+1)(s+2)(s+3)}{6}\right]^3 \approx \frac{s^9}{216}$ zu testende Nukleotidmischungen. Bei einer Schrittweite von 10% ($s = 10$) wären dies $286^3 \approx 2.3 \cdot 10^7$; bei 5% Genauigkeit ($s = 20$) bereits $1771^3 \approx 5.6 \cdot 10^9$ Möglichkeiten. Dies erklärt, warum YOUVAN mit nur 10% Genauigkeit und einem Synthesetopf arbeitete.

GALO1 und GALO3: Ein-Topf– und Drei-Topf-„split-and-mix“–Variante des GALO.

Der hier vorgestellte „split-and-mix“–Ansatz ist in der Lage, im Rahmen der vorgegebenen Genauigkeit (hier drei Nachkommastellen) nahezu perfekte Lösungen des gestellten Problems mit nur drei Synthesetöpfen zu finden. Es ist bemerkenswert, daß sogar die Ein-Topf–Variante des Algorithmus die anderen Verfahren aussticht.

Bedingt durch die Struktur des genetischen Codes sind einige Aminosäuren wie Arg, Leu und Ser, die durch jeweils sechs Codone repräsentiert werden, in der resultierenden Aminosäuremischung überproportional vertreten. Dieser Bias ist am

stärksten in der NN(G/C)–Bibliothek ausgeprägt und in der Drei-Topf–Variante des Algorithmus zu vernachlässigen. Nur in letzterem Ansatz wurden STOP–Codone nahezu vollständig vermieden. Alle anderen Verfahren produzieren jeweils etwa 3–4% STOP–Codone.

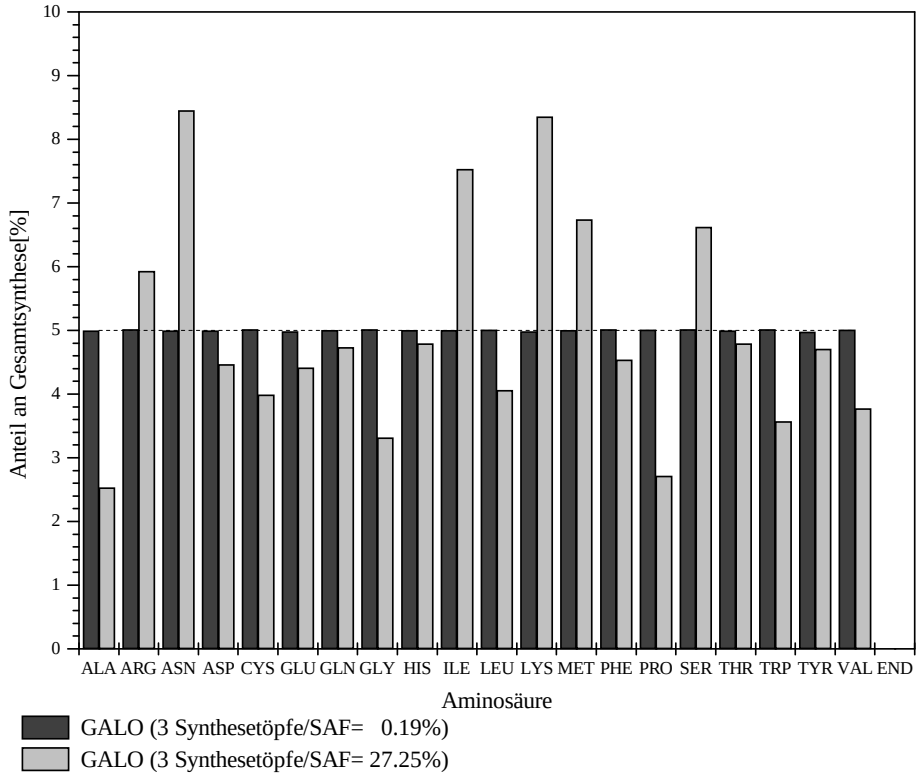

Abbildung 3.5: *Einfluß von Reaktionsraten der Nukleotide auf die errechnete Aminosäureverteilung.*

In Abbildung 3.5 wird der Einfluß von Reaktionsraten der Nukleotide auf die errechnete Aminosäureverteilung untersucht. Die schwarzen Balken stehen für die Lösung von GALO3 aus Abbildung 3.4 *ohne* Korrektur der Reaktionsraten, d.h. $Korr_A = Korr_T = Korr_C = Korr_G = 1$. Hätte man diese Nukleotidkonzentrationen in einen DNS–Synthesizer eingesetzt, der die Korrekturen (3.7)–(3.10) von Seite 33 nach JEREMY R. KNOWLES [51] benötigen würde, so ergäbe sich die Verteilung der Aminosäuren der grauen Balken mit einem absoluten Fehler von 27.25% anstelle von 0.19%.

Einzelne Aminosäuren verdoppeln bzw. halbieren fast ihre Konzentration in der Bibliothek. Dies zeigt, wie wichtig die Bestimmung der korrekten Reaktionsraten in der Praxis ist.

3.7.4 Stabilität der Lösungen

Da die Präparation der Nukleotidmischungen statistischen Fehlern unterliegt, wurde deren Einfluß auf die Zielzusammensetzung der Aminosäuren mit der Methode aus Kapitel 3.6 untersucht. Auch hier diente die equimolare Mischung aller 20 Aminosäuren als repräsentatives Beispiel. Für die Untersuchung wurden die in Kapitel 3.7.2 gefundenen Optima aus Abbildung 3.3 verwendet.

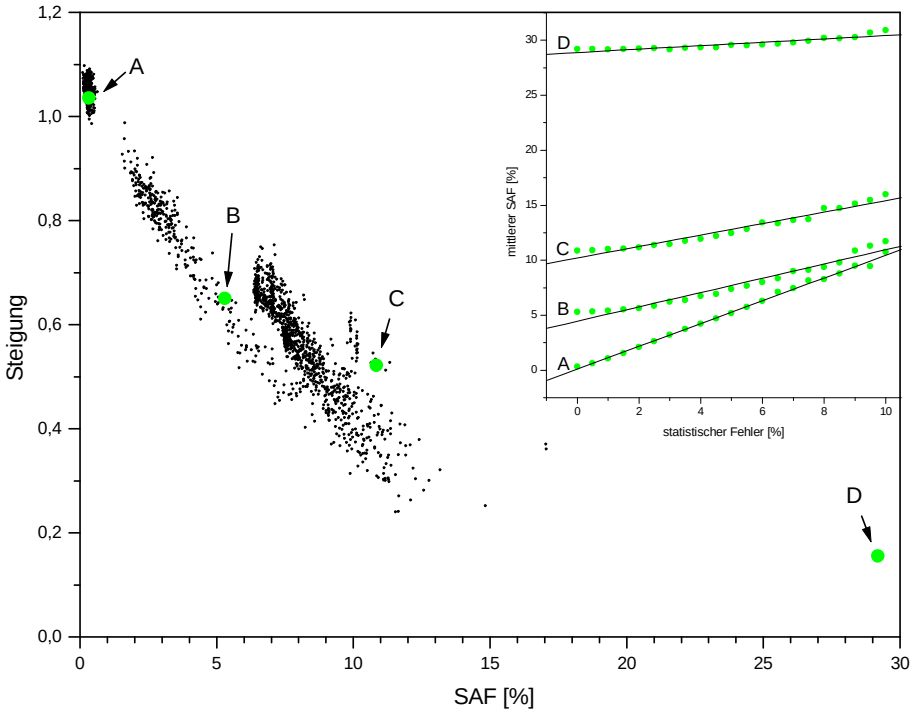

Abbildung 3.6: *Stabilität der Lösungen gegen experimentelle Fehler.*

In Abbildung 3.6 (Einschub) wird die Stabilität von vier ausgewählten Lösungen des Drei-Topf-„split-and-mix“–GALO gegenüber normalverteilten experimentellen Fehlern miteinander verglichen. Als Standardabweichung dient der statistische Fehler als prozentualer Anteil vom zu verrauschenden Wert. Der statistische Fehler wird nun in 0.5%-Schritten von 0.5% bis 10% durchvariiert und für 100 Programmläufe der Durchschnitt-SAF bestimmt.

Wie zu erwarten steigt der mittlere Fehler SAF der Zielzusammensetzung der Aminosäuren mit wachsendem statistischen Fehler der Nukleotidmischungen an. Näherungsweise steigt der SAF sogar linear an. Die dadurch definierbare Steigung kann als Maß für die Stabilität der Optima gegenüber statistischen Fehlern dienen. Je größer die Steigung, desto leichter wird das gefundene Optimum durch statistische Fehler verloren. Die Simulation zeigt, daß bessere Lösungen größere Steigungen besitzen, d.h. empfindlicher gegenüber kleinen Fehlern während der chemischen

DNS–Synthese sind. Dennoch scheinen die Optima breit genug zu sein, als daß sie durch experimentelle Fehler zu verlieren wären — dies ist für die Praxis wichtig.

Die Güte-Reihenfolge der Methoden, die in Abbildung 3.4 auf Seite 42 miteinander verglichen wurden, bleibt auch für vergleichsweise hohe statistische Fehler (10%) unverändert.

3.7.5 Hauptkomponentenanalyse der Lösungen

Für den Praktiker ist eine wichtige Frage, wie die Lösungen des Algorithmus im Konzentrationsraum verteilt sind. Mit anderen Worten: Existieren andere Kombinationen von Nukleotid-Konzentrationen, die ebenfalls die Wunschaminosäurebesetzung erzeugen, aber z.B. einfacher zu synthetisieren wären?

Wie in den Kapiteln zuvor dient auch hier die equimolare Mischung aller 20 Aminosäuren unter Vermeidung von STOP–Codonen als Beispiel. Die vom genetischen Algorithmus und vom GALO gefundenen 2000 Lösungen (siehe Abbildung 3.6) sowie 1000 zufällige (normierte) Nukleotid-Mischungen dienen als zu untersuchender Datensatz. Jede der Lösungen benötigt drei Synthesetöpfe. Daraus resultiert nach Kapitel 3.3 ein schwer zu veranschaulichender 29-dimensionaler Sequenzraum.

Die Hauptkomponentenanalyse [39, 84] transformiert das Koordinatensystem eines Datensatzes in ein neues, orthogonales Koordinatensystem gleicher Dimension. Die neuen Achsen werden durch Linearkombinationen der ursprünglichen Achsen konstruiert. Entscheidend ist dabei, daß die Reihenfolge der neuen Achsen abnehmende Anteile der Varianz der Datenpunkte erklärt: Die erste Achse zeigt in die Richtung maximaler Varianz, die zweite Achse in die Richtung der maximalen verbleibenden Varianz usw. In vielen praktischen Anwendungen kann bereits mit relativ wenigen Achsen, den sog. Hauptachsen oder Hauptkomponenten, ein Großteil der Varianz des Datensatzes erfaßt werden. Auf diese Weise können „wichtige“ Variablen von „unwesentlichen“ unterschieden werden. Außerdem kann ein Datensatz mittels Hauptkomponentenanalyse von statistischem Rauschen befreit werden, indem auf die Achsen verzichtet wird, die nur zu einem geringen Anteil der Gesamtvarianz beitragen.

Hier wurde die Hauptkomponentenanalyse nur zur Visualisierung des 29-dimensionalen Lösungsraums eingesetzt. In Abbildung 3.7 sind die insgesamt 3000 Mischungen im Koordinatensystem der ersten beiden Hauptkomponenten eingezeichnet. Jeder Mischung entspricht dabei ein einzelner Punkt. Der Wertebereich der

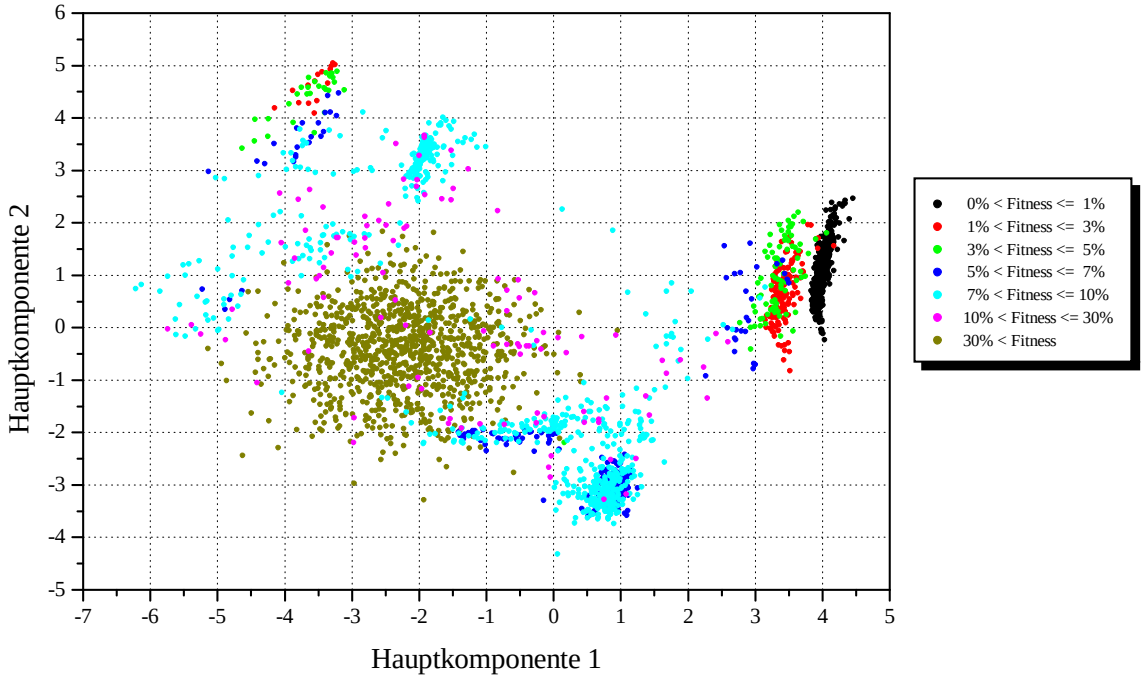

Abbildung 3.7: *Hauptkomponentenanalyse von 3000 Nukleotidmischungen: Verteilung der Nukleotidmischungen in den ersten beiden Hauptkomponenten.*

Summen der absoluten Fehler SAF wurde in sieben Klassen aufgeteilt. Jede Fehler-Klasse bekommt eine Farbe zugewiesen. Damit kann in Abbildung 3.7 sowohl die relative räumliche Verteilung der Mischungen zueinander als auch ihr Fehler abgelesen werden.

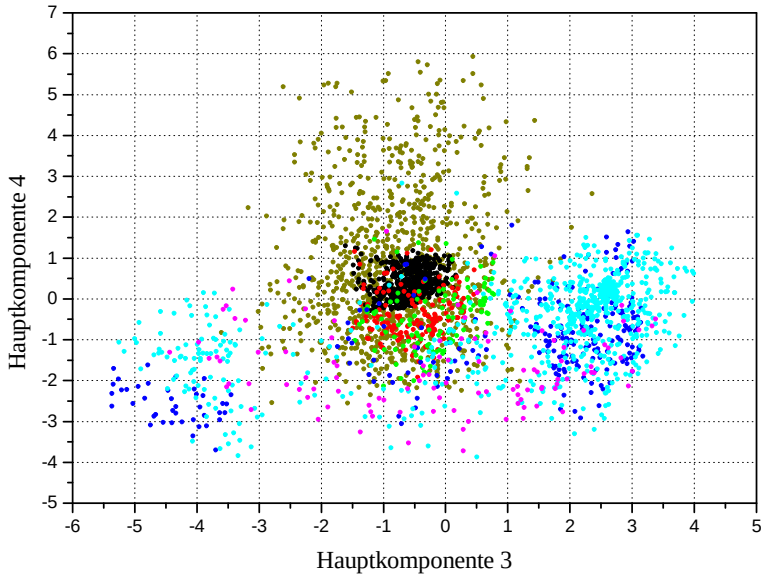

Abbildung 3.8: *Hauptkomponentenanalyse von 3000 Nukleotidmischungen: Verteilung der Nukleotidmischungen in der dritten und vierten Hauptkomponente.*

Da es möglich ist, daß zusammenhängende Cluster nach Hinzunahme weiterer Hauptkomponenten wieder separiert werden, wird derselbe Datensatz noch im Koordinatensystem der dritten und vierten Hauptkomponente aufgetragen. Abbildung 3.8 zeigt, daß bereits mit den ersten beiden Hauptkomponenten die Lösungen mit geringem Fehler ($SAF \leq 5\%$) sowie sehr schlechte Lösungen ($SAF > 30\%$) gut beschrieben werden. Lösungen mit mittlerem bis hohem Fehler ($5\% < SAF \leq 30\%$) zeigen eine Aufspaltung in der dritten Hauptkomponente.

Die Abbildungen 3.7 und 3.8 zeigen, daß die Hauptkomponentenanalyse Cluster an Mischungen mit typischen absoluten Fehlern identifiziert, obwohl der Fehler an keiner Stelle verwendet wird.

In beiden Diagrammen ist es leicht, die 2000 berechneten Lösungen von den 1000 zufälligen Mischungen durch die Farbgebung zu unterscheiden. Keine der Lösungen von GA und GALO hat einen *höheren* absoluten Fehler als 30%. Keine der zufälligen Mischungen hat einen *niedrigeren* absoluten Fehler als 30%. Deshalb sind nur die zufälligen Mischungen mit Dunkelgelb gekennzeichnet.

Die besten Lösungen (schwarz) belegen einen eng begrenzten Cluster, der von den übrigen Lösungen abgesetzt ist. Je größer der absolute Fehler ist (rot, grün, blau), auf desto mehr Cluster verteilen sich die Lösungen. Ab $SAF > 7\%$ (türkis) scheinen die einzelnen Cluster durch Grate miteinander verbunden zu sein. Ab $SAF > 10\%$ (voilett) sind keine offensichtlichen Cluster mehr zu erkennen.

Vergleicht man die Lage der Cluster mit der Verteilung der zufällig ausgewählten Mischungen, so sind die Cluster mit Fehlern $SAF < 10\%$ deutlich von den zufälligen Mischungen entfernt. Möglicherweise durchqueren einige Grate den Bereich der zufälligen Mischungen. Die lokalen Optima sind also auf nicht-zufällige Weise im Lösungsraum verteilt.

Da es nur einen einzigen Cluster mit Lösungen der besten Fitness gibt, scheint die Fitnesslandschaft des gestellten Problems ein deutlich ausgeprägtes globales Optimum zu besitzen. Dieses liegt an einer exponierten Stelle im Raum der Konzentrationen, weit entfernt von den anderen lokalen Optima. Insbesondere gibt es keine weiteren lokalen Optima mit vergleichbar guter Fitness.

Dies beantwortet die eingangs gestellte Frage:
Es gibt (zumindest für dieses Problem) nur eine einzige Kombination an Nukleotid-Konzentrationen, die das Doping-Schema optimal erfüllt. Der Experimentator hat also nicht die Wahl zwischen verschiedenen, gleich guten Lösungen.

3.8 Thioredoxin — ein weiteres Beispiel

Ein praktisches Beispiel soll die Anwendbarkeit des Doping–Algorithmus demonstrieren. Das Ziel ist hier eine Bibliothek an Thioredoxin-Molekülen, bei denen das aktive Zentrum von sechs Aminosäuren variiert wird. Der Rest des Moleküls besteht aus einer ausgewählten Wildtyp-Sequenz und wird hier nicht weiter berücksichtigt.

Proteinname	**Aminosäuren im aktiven Zentrum**						**Referenz**
Human Trx:	Trp	Cys	Gly	Pro	Cys	Lys	[Structure 2:503]
E.coli TrxA:	Trp	Cys	Gly	Pro	Cys	Lys	[Europ.J.Biochem. 6:475]
RsTrxA:	Trp	Cys	Gly	Pro	Cys	Lys	[PNAS 90:2179]
HelX:	Trp	Cys	Gly	His	Cys	Arg	[PNAS 90:2179]
RrTrxA:	Trp	Cys	Gly	Pro	Cys	Arg	[EMBO J. 12:3373]
CnTrx-c2:	Trp	Cys	Ala	Pro	Cys	Arg	[EMBO J. 12:3373]
BjORF132:	Trp	Cys	Val	Pro	Cys	His	[EMBO J. 12:3373]
SpTrxM:	Trp	Cys	Gly	Pro	Cys	Lys	[EMBO J. 12:3373]
BovPDI-A1:	Trp	Cys	Gly	His	Cys	Lys	[EMBO J. 12:3373]
BovPDI-A2:	Trp	Cys	Gly	His	Cys	Lys	[EMBO J. 12:3373]
Rat PDI C-term.:	Trp	Cys	Gly	His	Cys	Lys	[Nature 317:267]
Rat PDI N-term.:	Trp	Cys	Gly	His	Cys	Lys	[Nature 317:267]
TcpG:	Trp	Cys	Pro	His	Cys	Asn	[PNAS 90:2179]
Glutaredoxin (GRX):	Lys	Cys	Val	Tyr	Cys	Asp	[Structure 3:245]
Glutathion S-reductase (GST):	Val	Arg	Gly	Leu	Thr	His	[Structure 3:245]
DsbA (disulphide oxidant):	Phe	Cys	Pro	His	Cys	Tyr	[Structure 3:245]
Glutathion peroxidase:	Leu	Cso	Gly	Thr	Thr	Thr	[Europ.J.Biochem. 133:51]
Membran bound TlpA:	Trp	Cys	Val	Pro	Cys	Arg	[EMBO J. 12:3373]

Tabelle 3.1: *Aktive Zentren von Thioredoxin-ähnlichen Proteinen. Cso = Selenocystein — wird als Cystein und* nicht*, wie häufig unter Experimentatoren üblich, als 21.Aminosäure gezählt.*

Als erster Schritt werden die sechs Doping-Schemata durch ein Alignment der zur Klasse der Thioredoxin-ähnlichen Proteine in Tabelle 3.1 ermittelt. Anschließend berechnet der Doping–Algorithmus für jede Position des aktiven Zentrums die entsprechenden Nukleotid-Konzentrationen für jeweils jede der drei Codon-Positionen.

In den Tabellen 3.2 und 3.3 sind die Resultate der Berechnungen angegeben. Die Spalte „dope“ enthält das jeweilige Doping-Schema in %. Aus Platzgründen werden allerdings nicht die errechneten Nukleotidmischungen angegeben, sondern die zugehörige Aminosäureverteilung, aus der die Bibliothek besteht, wenn die Mischungen exakt umgesetzt werden. In den Spalten „GALOx“ stehen die Aminosäureverteilungen (in %), die sich ergeben, wenn die Nukleotidmischungen von GALO mit x Synthesetöpfen berechnet werden. Zur Erhöhung der Übersichtlichkeit sind die Werte der Doping-Schemata ungleich Null fett geschrieben.

Die Zeile SAF enthält die Summe der absoluten Fehler in % nach Gleichung (3.36). Nur eines der sechs Doping-Schemata (Position 2) ist mit einem einzigen Synthese-

Aminosäure	Position 1 [%]			Position 2 [%]		Position 3 [%]		
	dope	GALO1	GALO4	dope	GALO1	dope	GALO1	GALO2
ALA	0.00	0.10	0.00	0.00	0.00	**5.60**	**6.83**	**5.58**
ARG	0.00	2.97	0.01	**5.60**	**5.60**	0.00	2.02	0.00
ASN	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00
ASP	0.00	0.01	0.00	0.00	0.00	0.00	0.71	0.00
CYS	0.00	3.49	0.00	**94.40**	**94.40**	0.00	1.13	0.00
GLU	0.00	0.13	0.01	0.00	0.00	0.00	0.97	0.00
GLN	0.00	0.00	0.03	0.00	0.00	0.00	0.03	0.00
GLY	0.00	3.36	0.00	0.00	0.00	**66.60**	**67.41**	**66.63**
HIS	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00
ILE	0.00	0.02	0.01	0.00	0.00	0.00	0.00	0.00
LEU	**5.60**	**8.73**	**5.58**	0.00	0.00	0.00	0.93	0.00
LYS	**5.60**	**0.12**	**5.61**	0.00	0.00	0.00	0.00	0.00
MET	0.00	0.35	0.01	0.00	0.00	0.00	0.00	0.00
PHE	**5.60**	**0.41**	**5.58**	0.00	0.00	0.00	0.29	0.00
PRO	0.00	0.00	0.00	0.00	0.00	**11.10**	**0.20**	**11.09**
SER	0.00	2.45	0.00	0.00	0.00	0.00	0.27	0.00
THR	0.00	0.09	0.01	0.00	0.00	0.00	0.00	0.00
TRP	**72.00**	**74.13**	**71.98**	0.00	0.00	0.00	0.79	0.00
TYR	**5.60**	**0.15**	**5.58**	0.00	0.00	0.00	0.03	0.00
VAL	**5.60**	**0.40**	**5.58**	0.00	0.00	**16.70**	**17.58**	**16.70**
STOP	0.00	3.09	0.01	0.00	0.00	0.00	0.79	0.00
SAF		42.65	0.20		0.00		21.79	0.07

Tabelle 3.2: *Aminosäurebesetzungen von Thioredoxin-ähnlichen Proteinen — Positionen 1–3. In der Spalte „dope“ ist das Doping–Schema der jeweiligen Position angegeben.*

Aminosäure	Position 4 [%]			Position 5 [%]			Position 6 [%]		
	dope	GALO1	GALO2	dope	GALO1	GALO2	dope	GALO1	GALO2
ALA	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.34	0.00
ARG	0.00	0.00	0.00	0.00	1.37	0.00	**22.20**	**23.02**	**22.21**
ASN	0.00	2.68	0.00	0.00	0.01	0.00	**5.60**	**7.08**	**5.61**
ASP	0.00	0.00	0.00	0.00	0.02	0.00	**5.60**	**0.35**	**5.61**
CYS	0.00	0.00	0.00	**88.90**	**90.23**	**88.90**	0.00	0.13	0.00
GLU	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.27	0.00
GLN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.32	0.00
GLY	0.00	0.00	0.00	0.00	1.35	0.00	0.00	1.23	0.00
HIS	**38.90**	**38.85**	**38.76**	0.00	0.02	0.00	**11.10**	**0.52**	**11.10**
ILE	0.00	0.38	0.25	0.00	0.01	0.00	0.00	0.93	0.00
LEU	**5.60**	**5.49**	**5.44**	0.00	0.06	0.00	0.00	0.26	0.00
LYS	0.00	0.00	0.00	0.00	0.00	0.00	**44.30**	**45.36**	**44.27**
MET	0.00	0.00	0.25	0.00	0.00	0.00	0.00	1.48	0.00
PHE	0.00	0.33	0.20	0.00	1.41	0.00	0.00	0.01	0.00
PRO	**44.30**	**44.16**	**44.15**	0.00	0.01	0.00	0.00	0.49	0.00
SER	0.00	2.69	0.00	0.00	1.42	0.00	0.00	3.57	0.00
THR	**5.60**	**3.04**	**5.46**	**11.10**	**0.01**	**11.10**	**5.60**	**6.72**	**5.62**
TRP	0.00	0.00	0.00	0.00	1.30	0.00	0.00	0.58	0.00
TYR	**5.60**	**2.37**	**5.49**	0.00	1.41	0.00	**5.60**	**0.27**	**5.58**
VAL	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.12	0.00
STOP	0.00	0.00	0.00	0.00	1.34	0.00	0.00	1.98	0.00
SAF		12.17	1.40		22.19	0.00		42.31	0.10

Tabelle 3.3: *Aminosäurebesetzungen von Thioredoxin-ähnlichen Proteinen — Positionen 4–6. In der Spalte „dope“ ist das Doping–Schema der jeweiligen Position angegeben.*

topf exakt zu realisieren. Immerhin benötigen vier der sechs Schemata „nur“ zwei Synthesetöpfe zur perfekten[2] Aminosäureverteilung. Die erste Position ist eines der seltenen Beispiele, das nicht mit drei Synthesetöpfen auskommt.

3.9 ARG, GLN, LEU — ein analytisches Beispiel

Es gibt Doping-Schemata, die sich analytisch, d.h. ohne Einsatz des numerischen Doping–Algorithmus, lösen lassen. Dies soll an einem einfachen Beispiel demonstriert werden:

> Gesucht sind die drei Nukleotidmischungen, die in einer Ein-Topf–Reaktion für das Doping–Schema der Aminosäuren ARG, GLN, LEU in beliebigen gegebenen Konzentrationen S_{ARG}, S_{GLN} und S_{LEU} codieren. Die übrigen Aminosäuren sowie die STOP–Codone sollen verschwinden.

Experimentelle Untersuchungen [18, 19] von DAVIDSON *et al.* zeigten, daß Proteine, die sich ausschließlich aus den drei Aminosäuren ARG, GLN, LEU zusammensetzen, sich ähnlich wie natürliche Proteine falten. Die zugehörige Faltungslandschaft war Gegenstand theoretischer Untersuchungen [3].

Nach Kapitel 3.4.3 auf Seite 34 treten die drei Aminosäuren bei gegebenen Nukleotidkonzentrationen mit folgenden Wahrscheinlichkeiten auf:

$$P_{ARG} = G_2(C_1 + A_1(A_3 + G_3)) \tag{3.37}$$

$$P_{GLN} = C_1 A_2(A_3 + G_3) \tag{3.38}$$

$$P_{LEU} = T_2(C_1 + T_1(A_3 + G_3)) \tag{3.39}$$

Gesucht sind diejenigen Nukleotidkonzentrationen, mit denen die Beziehungen $P_{ARG} = S_{ARG}$, $P_{GLN} = S_{GLN}$ und $P_{LEU} = S_{LEU}$ erfüllt werden.

Zunächst ist offensichtlich, daß in den Gleichungen (3.37)–(3.39) die Variablen G_1, C_2, T_3 und C_3 nicht benutzt werden. Sie werden gleich Null gesetzt, da die Konzentrationen der verbleibenden Aminosäuren gleich Null sein sollen: $G_1 = C_2 = T_3 = C_3 := 0$. Zusammen mit der Normierbedingung $T_3 + C_3 + A_3 + G_3 = 1$ ergibt sich eine weitere Bedingung $A_3 + G_3 = 1$. Diese Bedingungen werden in die Gleichungen (3.11)–(3.31) eingesetzt. Somit verschwinden die Aminosäuren ALA, ASN,

[2] Im Rahmen der Rechengenauigkeit von drei Nachkommastellen.

ASP, CYS, GLU, GLY, HIS, PHE, PRO, SER, THR, TYR, VAL. Die Gleichungen (3.37)–(3.39) formen sich um zu:

$$S_{ARG} = (A_1 + C_1)\, G_2 \tag{3.40}$$

$$S_{GLN} = C_1 A_2 \tag{3.41}$$

$$S_{LEU} = (C_1 + T_1)\, T_2 \tag{3.42}$$

Es sind nun noch folgende Bedingungen zu erfüllen:

$$S_{ILE} \equiv 0 = A_1 T_2 A_3 \tag{3.43}$$

$$S_{LYS} \equiv 0 = A_1 A_2 \tag{3.44}$$

$$S_{MET} \equiv 0 = A_1 T_2 G_3 \tag{3.45}$$

$$S_{TRP} \equiv 0 = T_1 G_2 G_3 \tag{3.46}$$

$$S_{END} \equiv 0 = T_1 (A_2 + G_2 A_3) \tag{3.47}$$

Diese Bedingungen werden durch folgende voneinander unabhängige Lösungen erfüllt:

$$A_1 = T_1 = 0 \tag{3.48}$$

$$A_1 = G_3 = 0 \ \wedge \ A_2 + G_2 A_3 = 0 \tag{3.49}$$

$$A_1 = A_2 = G_2 = 0 \tag{3.50}$$

$$A_2 = A_3 = G_3 = 0 \tag{3.51}$$

$$T_1 = T_2 = A_2 = 0 \tag{3.52}$$

$$T_2 = A_2 = G_2 = 0 \tag{3.53}$$

Die Lösungen (3.50)–(3.53) stehen im Widerspruch zu den Gleichungen (3.40)–(3.42). Lösung (3.49) kann nicht für positive Konzentrationen erfüllt werden, damit verbleibt noch Lösung (3.48). Im Zusammenspiel mit der Normierbedingung $T_1 + C_1 + A_1 + G_1 = 1$ gilt für die Konzentration $C_1 = 1$. Wird dies alles in die Gleichungen (3.40)–(3.42) eingesetzt, so ergibt sich das einfache Ergebnis:

$$S_{ARG} = G_2 \tag{3.54}$$

$$S_{GLN} = A_2 \tag{3.55}$$

$$S_{LEU} = T_2 \tag{3.56}$$

In Tabelle 3.4 sind die gesuchten Nukleotidkonzentrationen für jede der drei Positionen im Codon aufgelistet:

Es existiert zwar genau eine Lösung, doch ist diese degeneriert, da unendlich viele zulässige Kombinationen an Konzentrationen A_3 und G_3 ($0 \leq A_3, G_3 \leq 1$) existieren. Es muß lediglich die Randbedingung $A_3 + G_3 = 1$ erfüllt sein. Die entsprechende Fitnesslandschaft hat demnach keine lokalen Optima, sondern einen Grat an

i	T_i	C_i	A_i	G_i
1	0	1	0	0
2	S_{LEU}	0	S_{GLN}	S_{ARG}
3	0	0	A_3	$1 - A_3$

Tabelle 3.4: *Nukleotidkonzentrationen für Doping–Schema ARG, GLN, LEU in beliebigen Konzentrationen.*

optimalen Lösungen. Die Freiheit in der Wahl der Konzentration A_3 erleichtert die Berücksichtigung einer eventuellen Codonusage. Im Falle $A_3 = 0$ nutzt die Bibliothek nur die Codone {CGG, CAG, CTG}. Die Codone {CGA, CAA, CTA} werden in Fall $A_3 = 1$ verwendet. Für Zwischenwerte von A_3 können beliebige Mischungen der Sets an Codonen eingesetzt werden.

3.10 Zusammenfassung

Die Methode des Dopings erhöht die Wahrscheinlichkeit, „positive“ Mutanten in einer Zufallsbibliothek durch Reduktion des Suchraums auf eine vielversprechende Untermenge an Sequenzen zu finden. Prinzipiell ist es die bessere Methode, gleich Trinukleotid-Phosphoramidite [11, 69, 78, 112, 124] zu verwenden, da damit unerwünschte Codone vollständig vermieden werden können. Gegenüber letztgenannten besitzt die vorgestellte Methode mehrere Vorteile:

- Trinukleotide sind noch nicht kommerziell verfügbar.
- Trinukleotide sind experimentell schwieriger zu behandeln als Mononukleotide.
- Trinukleotide sind teurer als Mononukleotide.
- Der Nachteil, daß in den meisten Fällen mit nur einem Synthesetopf auch andere Aminosäuren in der Bibliothek vorkommen, kann auch ein Vorteil sein, da der Suchraum etwas unschärfer eingegrenzt wird. Auf diese Weise können unvorhergesehene „positive“ Mutanten gefunden werden. Nach Meinung des Autors ist dies sogar der entscheidende Vorteil des Doping–Algorithmus, da damit auf einfache Weise eine Zufallsbibliothek mit „Bias“ generiert werden kann.

Der vorgestellte Algorithmus produziert Doping-Schemata mit einer Auflösung von 0.1%. Diese Auflösung liegt damit deutlich über der bereits existierender Metho-

den [2, 4, 45]. Die benötigte Computerzeit liegt je nach Anzahl gewünscher Synthesetöpfe auf handelsüblichen Pentium-PCs zwischen einigen Sekunden und wenigen Minuten. Im Gegensatz zu anderen Methoden läßt der Algorithmus Korrekturfaktoren für unterschiedliche Reaktionsraten der Nukleotide zu und ermöglicht das Bevorzugen bestimmter Codone.

Der Hauptnachteil des Algorithmus soll nicht verschwiegen werden — der Syntheseaufwand:
Pro Codon sind drei Nukleotidmischungen erforderlich. Falls n Synthesetöpfe nach dem split-and-mix–Verfahren notwendig werden, benötigt der Experimentator $3n$ Nukleotidmischungen.

Derzeit erreichen DNS-Synthesizer eine Synthesegenauigkeit von etwa 2.5%, sofern die Nukleotid-Mischungen extern zubereitet sind. Sowohl die Synthesegenauigkeit als auch die Herstellung der Nukleotid-Mischungen unterliegen experimentellen Fehlern. Der Doping–Algorithmus simuliert aus diesem Grund die Stabilität der gefundenen Optima gegenüber statistischen, normalverteilten Fehlern. Zumindest bei dem untersuchten Beispiel der equimolaren Mischung aller 20 Aminosäuren waren bessere Optima auch empfindlicher gegenüber simulierten Fehlern. Trotzdem war selbst bei den besten Optima die absolute Abweichung bei relevanten Fehlern bis 3% sehr gering.

Aus den Simulationen lassen sich einige (vorläufige) Aussagen über die Struktur der Fitness-Landschaft der equimolaren Mischung aller 20 Aminosäuren finden. Die lokalen Optima scheinen auf nicht-zufällige Weise im Raum der Konzentrationen verteilt zu sein. Das globale Optimum hingegen liegt entfernt von den übrigen Optima. Dies ist der Grund, warum GALO häufig den genetischen Algorithmus in der Qualität der gefundenen Lösungen übertrifft. Das lokale Optimierverfahren erhöht die Wahrscheinlichkeit, das isolierte globale Optimum zu finden. Die Simulation statistischer Fehler deutet zudem darauf hin, daß die gefundenen Optima eher „breit“ zu sein scheinen.

3.11 Ausblick

Der Doping–Algorithmus ist hier nur als vorläufiger Prototyp entwickelt worden. Deshalb sind eine Reihe an Untersuchungen und Erweiterungen denkbar:

- Wichtig ist als nächster Schritt die *experimentelle Überprüfung* des Doping–Algorithmus. Erste Versuche zum Design von Hirudin–Bibliotheken von FRANK

Wirsching, Rüdiger Dietrich und Andreas Schwienhorst waren erfolgversprechend [130].

- Bestimmung der präzisen, *relativen Reaktionsraten der einzelnen Nukleotide*, da hier große Uneinigkeit in der Literatur besteht. Vermutlich schwanken die Werte auch zwischen verschiedenen DNS–Synthesizern.
- Denkbar sind *sequenzabhängige Korrekturfaktoren* der Reaktionsraten.
- Zur Reduzierung des experimentellen Aufwands könnte mit *Linearkombinationen vorgefertigter Nukleotid-Mischungen* gearbeitet werden.
- Interessant wäre eine detailliertere *Untersuchung der Fitnesslandschaften* unterschiedlicher Doping-Schemata, insbesondere die Verteilung der Optima.
- Möglicherweise kann durch eine genauere *mathematische Analyse der Fitnesslandschaft* der Suchraum weiter eingeschränkt und strukturiert werden.
- Der experimentelle Aufwand bei einer größeren Anzahl von Synthesetöpfen legt die Entwicklung einer *automatisierten Misch- und Synthesestation* nahe.

Kapitel 4

General Regression Neural Network

Ein Hauptteil dieser Arbeit entfiel auf die Suche, Entwicklung und Implementation geeigneter statistischer Verfahren, die Anwendung im TST–Algorithmus (Kapitel 5) finden können. Neben der Implementation verschiedener Varianten von Backpropagation–Netzen lag der Schwerpunkt der Entwicklung in der Adaption des vergleichsweise unbekannten „General Regression Neural Network“ (GRNN) von Donald F. Specht [114].

In diesem Kapitel soll deshalb das „General Regression Neural Network“ (GRNN) als ein modernes und erfolgreiches nichtlineares Regressionsverfahren vorgestellt werden. Besonderes Gewicht wird dabei, neben der Herleitung der Grundlagen, auf die vom Autor entwickelten Trainingsverfahren gelegt. Zum Abschluß wird in Kapitel 4.10 die Güte des Netzes an ausgewählten Beispielen demonstriert und mit dem Stand der Technik verglichen.

4.1 Einleitung

Ein grundsätzliches Problem in der Wissenschaft ist die Bestimmung des funktionalen Zusammenhangs zwischen einer unabhängigen, eventuell verrauschten (vektoriellen) Variable $\mathbf{x}$ und einer abhängigen (vektoriellen) Variable $\mathbf{y}$. Typische Beispiele dafür sind:

- Messung einer Größe (z.B. Gasdruck) in Abhängigkeit von einem Parameter (z.B. Temperatur).

- Analyse von Zeitreihen, wie z.B. Aktienkursvorhersage in Abhängigkeit vom bereits bekannten Kurs.
- Bestimmung der Sekundär- oder Tertiärstruktur von Proteinen in Abhängigkeit von der Aminosäuresequenz.

In den beiden ersten Beispielen ist eine stetig differenzierbare Abbildung zwischen der unabhängigen Variablen $\mathbf{x}$ und der abhängigen Variablen y gesucht. Im letzten Beispiel hingegen eine unstetige Abbildung, da abhängig von der Aminosäuresequenz der Strukturtyp y einen diskreten Wert annimmt.

Es existieren mehrere Wege, um den Zusammenhang zwischen den abhängigen und den unabhängigen Variablen eines Problems zu ermitteln:

1. Steht ein Modell zur Verfügung, mit dem der Zusammenhang befriedigend beschrieben werden kann, so ist das Problem bereits gelöst. Gegebenenfalls muß das Modell iterativ mit Experimenten oder Simulationen verfeinert werden.

2. Häufig steht ein Modell bereit, in dem freie Parameter vorkommen. In der Regel werden diese freien Parameter numerisch so an bestehende Daten angepaßt, daß das Modell die Daten optimal erklärt, d.h. der Fehler zwischen Modell und Daten minimiert wird. Die einfachste Methode, freie Parameter in ein Modell zu integrieren, ist, wie in der klassischen Statistik [39, 84], Linearkombinationen von vorgegebenden Funktionen zur Vorhersage von y zu benutzen. Viele Probleme sind allerdings von nichtlinearer Natur und können mit linearen Modellen nur ungenügend beschrieben werden.

3. Bei vielen Problemen ist entweder gar kein Modell vorhanden, oder die Vorhersagen der bestehenden Modelle sind zu ungenau oder nicht allgemein genug. In solchen Fällen bietet sich die modellfreie Regression an, um den Zusammenhang zwischen der unabhängigen und der abhängigen Variablen wenigstens numerisch angeben zu können. Moderne Beispiele für nichtlineare Approximationsverfahren sind künstliche neuronale feedforward–Netze, wie das populäre Backpropagation–Netz.

   Klassifizieraufgaben werden meist dadurch gelöst, daß das Problem zunächst als Approximationsaufgabe behandelt wird. Aus den Regressionswerten muß dann durch ein geeignetes Kriterium wie einem Schwellwert oder dem maximalen Wert etc. die Klasse ermittelt werden. Dieses Verfahren ist nicht ganz unproblematisch, da die optimale Regressionsfunktion nicht automatisch zur optimalen Klassizifierung führen muß (siehe Kapitel 4.10, Abbildung 4.15).

Der TST–Algorithmus soll Probleme der letztgenannten Kategorie, bei denen kein Modell zur Vorhersage der Fitness benutzt wird, lösen, deshalb beschäftigen wir uns im weiteren Verlauf dieses Kapitels mit einem speziellen Verfahren zur modellfreien Regression — dem „General Regression Neural Network“ (GRNN).

Das GRNN wurde 1991 von DONALD F. SPECHT in [114] erstmals beschrieben. Obwohl in dieser bis dato relativ wenig beachteten Arbeit von einem neuronalen Netz gesprochen wird, ist das dort beschriebene Verfahren im Kern eine Approximationsformel, die eine Art gewichteten Mittelwert bildet. Die Formel läßt sich jedoch leicht parallelisieren und als neuronales Netz interpretieren (Kapitel 4.6). Der Vorteil dieser Sichtweise ist, daß das rechenintensive Abrufen von Informationen auf parallel arbeitender Hardware zu implementieren ist.

SPECHT zeigte in [114] an einigen Beispielen, daß sein „Netz“ mindestens die gleiche Generalisierungsgenauigkeit wie ein Backpropagation–Netz mit nur 1% der Anzahl Lernmuster erreicht. Der Hauptnachteil des GRNN ist der Rechenaufwand beim Training und Abrufen der gelernten Information im Vergleich zu feedforward–Netzen.

4.2 Regressionsformel

Die Regression einer abhängigen Variable $\mathbf{y}$ von einer unabhängigen Variablen $\mathbf{x}$ ist allgemein die Berechnung des *wahrscheinlichsten* Wertes für $\mathbf{y}$. Das GRNN approximiert unter Verwendung der Wahrscheinlichkeitsdichtefunktion $f(\mathbf{x}, \mathbf{y})$, die empirisch aus den beobachteten Daten durch Schätzung mittels der Methode der PARZEN–Fenster [81] ermittelt wird.

Im folgenden wird o.B.d.A. nur eine *skalare* abhängige Variable y angenommen. Die Erweiterung auf eine vektorielle Variable $\mathbf{y}$ ist trivial — es muß lediglich Gleichung (4.1) für jede Dimension von $\mathbf{y}$ parallel angewandt werden. Formuliert man das GRNN als künstliches neuronales Netz (siehe Abbildung 4.1 auf Seite 63), so ist für jede Dimension von $\mathbf{y}$ eine eigene Summations- und Ausgabeschicht zu implementieren.

Gegeben sei mit $f(\mathbf{x}, y)$ die kontinuierliche Wahrscheinlichkeitsdichte der vektoriellen Zufallsvariable $\mathbf{x}$ (p Dimensionen) und der skalaren Zufallsvariable y. Es seien $\mathbf{X}$ und Y gemessene Werte der Zufallsvariablen $\mathbf{x}$ und y. Der Erwartungswert $E[y|\mathbf{X}]$

bzw. die Regression $\hat{y}(\mathbf{X})$ von y für ein gegebenes $\mathbf{X}$ berechnet sich nach:

$$\hat{y}(\mathbf{X}) \equiv E[y|\mathbf{X}] := \frac{\int_{-\infty}^{+\infty} dy\ y \cdot f(\mathbf{X}, y)}{\int_{-\infty}^{+\infty} dy\ f(\mathbf{X}, y)} \tag{4.1}$$

Ist die Wahrscheinlichkeitsdichtefunktion $f(\mathbf{x}, y)$ bekannt, so kann der Erwartungswert von y nach Gleichung (4.1) berechnet werden, und das Regressionsproblem ist gelöst.

In der Praxis ist jedoch die „echte“ Wahrscheinlichkeitsdichtefunktion unbekannt — sie muß aus den verfügbaren Werten von $\mathbf{x}$ und y geschätzt werden.

4.3 Univariate Wahrscheinlichkeitsdichte nach PARZEN

PARZEN [81] stellte 1962 eine Methode vor, die univariate Wahrscheinlichkeitsdichte aus einer Zufallsstichprobe der Größe n abzuschätzen. Seine genäherte Dichtefunktion $\hat{f}(x)$ konvergiert asymptotisch mit wachsender Größe der Stichprobe gegen die wirkliche Wahrscheinlichkeitsdichte $f(x)$. Er benutzt eine Gewichts- oder Kernel–Funktion $W(d)$, die ihren maximalen Wert an der Stelle $d = 0$ hat und für wachsenden Betrag von d schnell gegen 0 konvergiert; $d \equiv d(X, X^i)$ ist hierbei die Distanz oder Metrik zwischen den Punkten X und X^i. Ein oberer Index steht in dieser Arbeit für die Nummer des Punktes aus der Stichprobe; ein unterer Index im mehrdimensionalen Fall für die Dimension. Die Kernel–Funktion muß folgenden Bedingungen genügen:

- Die Kernel–Funktion muß beschränkt sein:

$$\sup_d |W(d)| < \infty \tag{4.2}$$

- Die Kernel–Funktion muß für wachsenden Betrag des Arguments schnell gegen 0 konvergieren:

$$\int_{-\infty}^{+\infty} dx\ |W(x)| < \infty \tag{4.3}$$

$$\lim_{x \to \infty} |xW(x)| = 0 \tag{4.4}$$

- Die Kernel–Funktion muß normiert sein, damit die genäherte Wahrscheinlichkeitsdichtefunktion $\hat{f}(x)$ ebenfalls normiert ist:

$$\int_{-\infty}^{+\infty} dx\ |W(x)| = 1 \tag{4.5}$$

- Die Kernel–Funktion muß symmetrisch sein:

$$W(x) = W(-x) \tag{4.6}$$

Diese Bedingungen implizieren, daß die Kernel–Funktion in der Regel im groben einer Glockenkurve ähneln wird. Einige Beispiele für normierte Kernel–Funktionen demonstrieren dies:

$$\text{GAUSS'scher Kernel:} \qquad W(d,\sigma) := \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{d^2}{2\sigma^2}} \tag{4.7}$$

$$\text{GAUSS-ähnlicher Kernel:} \qquad W(d,\sigma) := \frac{1}{\pi\sigma\left[1+\left(\frac{d}{\sigma}\right)^2\right]} \tag{4.8}$$

$$\text{Kastenförmiger Kernel:} \qquad W(d,\sigma) := \begin{cases} \frac{1}{2\sigma} & , \quad |d| < \sigma \\ 0 & , \quad \text{sonst} \end{cases} \tag{4.9}$$

$$\text{Dreieckiger Kernel:} \qquad W(d,\sigma) := \begin{cases} \frac{1}{\sigma}\left(1-\left|\frac{d}{\sigma}\right|\right) & , \quad |d| < \sigma \\ 0 & , \quad \text{sonst} \end{cases} \tag{4.10}$$

In praktischen Anwendungen wird nahezu immer der GAUSS-förmige Kernel (4.7) verwendet. Dieser Kernel scheint für Regressions- wie auch Klassifizierungsaufgaben optimale Eigenschaften zu besitzen, setzt aber die wiederholte Berechnung der Exponentialfunktion voraus. Aus Performancegründen wurde deshalb vor allem in älteren Arbeiten der GAUSS-ähnliche Kernel (4.8) implementiert. Dieser Kernel erweist sich jedoch häufig wegen des schmalen Optimums aber breiten „Schwanzes“ als nur mäßig geeignet.

Die genäherte Dichtefunktion $\hat{f}(X)$ läßt sich im *eindimensionalen* Fall mit der Stichprobengröße n wie folgt schreiben:

$$\begin{aligned} \hat{f}(X) &:= \frac{1}{n}\sum_{i=1}^{n} W\left(\frac{d(X,X^i)}{\sigma^i}\right) \\ &= \frac{1}{n}\sum_{i=1}^{n} W\left(\frac{X-X^i}{\sigma^i}\right) \end{aligned} \tag{4.11}$$

Der Parameter $\sigma^i \geq 0$ ist, analog zur Standardabweichung bei der Normalverteilung, ein Maß für die „Breite“ der Kernel–Funktion für jedes Mitglied der Stichprobe.

4.4 Multivariate Wahrscheinlichkeitsdichte nach Cacoullos

Im Jahr 1966 erweiterte Cacoullos [7] Parzen's Methode auf den mehrdimensionalen Fall. Die multivariate genäherte Dichtefunktion

$$\hat{f}(\mathbf{X}) = \frac{1}{n}\sum_{i=1}^{n} W\left(\frac{X_1 - X_1^i}{\sigma_1^i}, \ldots, \frac{X_p - X_p^i}{\sigma_p^i}\right) \tag{4.12}$$

hat im allgemeinsten Fall für jede Variable $\mathbf{X^i}$ und jede der p Dimensionen ihren eigenen Skalierungsfaktor $\sigma_j^i \quad (1 \leq i \leq n,\ 1 \leq j \leq p)$.

Zur Vereinfachung nehmen wir an, daß die Skalierungsfaktoren für alle Dimensionen gleich sind:

$$\sigma^i \equiv \sigma_1^i = \ldots = \sigma_p^i, \quad \forall i \tag{4.13}$$

Durch Normieren der Variablen auf gleiche Variation in jeder Dimension läßt sich diese Forderung meist rechtfertigen. Es existieren aber Probleme (siehe Kapitel 4.10.1), für die diese Forderung eine zu starke Einschränkung bedeutet. Im Allgemeinen jedoch wird diese zusätzliche Vereinfachung in der Praxis kaum Probleme bereiten.

Die zweite, allgemein übliche Vereinfachung betrifft die Form des multivariaten Kernels als Produkt univariater Kernels:

$$W(\mathbf{X}) = \prod_{j=1}^{p} W(X_j) \tag{4.14}$$

Man beachte, daß für jede Dimension dieselbe univariate Kernel–Funktion angenommen wird.

4.5 Fundamentalgleichung des GRNN

Für den Einsatz im GRNN wird eine nochmals leicht veränderte Form des multivariaten Kernels angenommen:

$$W(\mathbf{X}, Y) = W(\mathbf{X}) \cdot W(Y) \tag{4.15}$$

Zusammengefaßt ergibt sich für die multivariate genäherte Dichtefunktion des GRNN:

$$\hat{f}(\mathbf{X}, Y) = \frac{1}{n}\sum_{i=1}^{n} W\left(\frac{d_x(\mathbf{X}, \mathbf{X}^i)}{\sigma^i}\right) \cdot W\left(\frac{d_y(Y, Y^i)}{\sigma^i}\right) \tag{4.16}$$

Hier wurde übrigens die vereinfachende Annahme getroffen, für jedes Mitglied der Stichprobe dieselbe Kernel–Funktion zu verwenden. Mit Gleichung (4.16) steht nun

das Werkzeug zur Vervollständigung des GRNN zur Verfügung. Setzt man die Gleichung in Gleichung (4.1) ein, so erhält man als Erwartungswert für y folgenden Ausdruck:

$$\hat{y}(\mathbf{X}) = \frac{\frac{1}{n}\sum_{i=1}^{n}\left[W\left(\frac{d_x(\mathbf{X},\mathbf{X}^i)}{\sigma^i}\right)\int_{-\infty}^{+\infty} dy\; y\cdot W\left(\frac{d_y(y,Y^i)}{\sigma^i}\right)\right]}{\frac{1}{n}\sum_{i=1}^{n}\left[W\left(\frac{d_x(\mathbf{X},\mathbf{X}^i)}{\sigma^i}\right)\int_{-\infty}^{+\infty} dy\; W\left(\frac{d_y(y,Y^i)}{\sigma^i}\right)\right]} \tag{4.17}$$

Das Integral über y im Nenner ist wegen der Normierbedingung (4.5) gleich Eins. Unter der Annahme, daß als eindimensionale Metrik $d_y(y,Y^i) := |y - Y^i|$ der Absolutbetrag der Differenz zwischen y und Y^i genommen wird, läßt sich das Integral im Zähler folgendermaßen umformen:

$$\begin{aligned}\int_{-\infty}^{+\infty} dy\; y\cdot W\left(\frac{d_y(y,Y^i)}{\sigma^i}\right) &= \underbrace{\int_{-\infty}^{+\infty} dy\;(y-Y^i)\cdot W\left(\frac{|y-Y^i|}{\sigma^i}\right)}_{0} + Y^i\underbrace{\int_{-\infty}^{+\infty} dy\; W\left(\frac{|y-Y^i|}{\sigma^i}\right)}_{1}\\ &= Y^i\end{aligned} \tag{4.18}$$

Das erste Integral auf der rechten Seite ist wegen der Spiegelsymmetrie (4.6) der Kernel–Funktion gleich Null, das Zweite wegen der Normierbedingung (4.5) gleich Eins. Ersetzt man die Integrale in Gleichung (4.17) entsprechend, so ergibt sich die „Fundamentalgleichung“ des General Regression Neural Network:

$$\hat{y}(\mathbf{X}) = \frac{\sum_{i=1}^{n} Y^i\cdot W\left(\frac{d_x(\mathbf{X},\mathbf{X}^i)}{\sigma^i}\right)}{\sum_{i=1}^{n} W\left(\frac{d_x(\mathbf{X},\mathbf{X}^i)}{\sigma^i}\right)} \tag{4.19}$$

Man beachte, daß bei dieser Herleitung in Gegensatz zu [71, 114] keine speziellen Kernel–Funktionen und Metriken Verwendung fanden.

Die Fundamentalgleichung (4.19) hat folgende Eigenschaften, die sich alle durch einfaches Einsetzen beweisen lassen:

1. Gleichung (4.19) bildet einen gewichteten Mittelwert:
   Jeder Punkt der Stichprobe wird mit seiner PARZEN'schen Kernel- oder Fenster–Funktion an der Stelle $\mathbf{X}$ gewichtet und dann aufsummiert.

2. Die Fitfunktion ist an jeder Stelle $\mathbf{X}$ des Definitionsbereichs definiert.

3. Die Fitfunktion ist stetig und stetig differenzierbar, solange kein σ^i gleich Null ist.

4. Die einzigen unbekannten und noch zu bestimmenden Parameter sind die n Skalierungsfaktoren σ^i. Siehe dazu Kapitel 4.9.

5. Besteht die Stichprobe nur aus einem einzelnen Punkt ($n = 1$), so nimmt der Erwartungswert $\hat{y}(\mathbf{X})$ für alle $\mathbf{X}$ den konstanten Wert Y^1 an.

6. Sind alle $Y^1 = \ldots = Y^n = Y$ der Stichprobe identisch, so nimmt der Erwartungswert $\hat{y}(\mathbf{X})$ für alle $\mathbf{X}$ und alle σ^i den konstanten Wert Y an.

7. Es gelte vorübergehend $\sigma^1 = \ldots = \sigma^n = \sigma$.

   - Limit $\sigma \to 0$:
     Aus dem kontinuierlichen wird ein stufenförmiger Fit. Der Erwartungswert $\hat{y}(\mathbf{X})$ nimmt den (nächstgelegenen) Wert Y^i an, für den die Distanz $d_x(\mathbf{X}, \mathbf{X}^i)$ minimal wird.
   - Limit $\sigma \to +\infty$:
     Der Erwartungswert $\hat{y}(\mathbf{X})$ entspricht dem (ungewichteten) Mittelwert aller Y^i.
   - Liegt $\mathbf{X}$ weit entfernt von den $\mathbf{X}^i$, so nimmt der Erwartungswert $\hat{y}(\mathbf{X})$ den nächstgelegenen Wert Y^i an.

8. Liegt $\mathbf{X}$ weit entfernt von den $\mathbf{X}^i$, so nimmt der Erwartungswert $\hat{y}(\mathbf{X})$ im Falle unterschiedlicher σ^i den Wert $Y^{i'}$ an, dessen $\sigma^{i'}$ am *größten* ist, also *nicht* den Wert des nächstgelegenen Y^i wie im Falle identischer σ^i. Insbesondere kann das GRNN dadurch nicht extrapolieren, sondern nur zwischen den bekannten Stützstellen approximieren. Diesen Nachteil teilt das GRNN mit feedforward–Netzen.

Mit GAUSS'schem Kernel und EUCLID'scher Metrik nimmt Gleichung (4.19) die bekannte, hier implementierte Form an:

$$\hat{y}(\mathbf{X}) = \frac{\sum_{i=1}^{n} Y^i \cdot \exp\left[-\frac{D_i^2}{2(\sigma^i)^2}\right]}{\sum_{i=1}^{n} \exp\left[-\frac{D_i^2}{2(\sigma^i)^2}\right]} \quad \text{, mit } D_i^2 := (\mathbf{X} - \mathbf{X^i})^T(\mathbf{X} - \mathbf{X^i}) \tag{4.20}$$

4.6 Das GRNN als neuronales Netz

Wie der Name dieses Approximationsverfahrens bereits impliziert, läßt sich die Approximationsformel (4.19) in die Struktur eines künstlichen neuronalen Netzes brin-

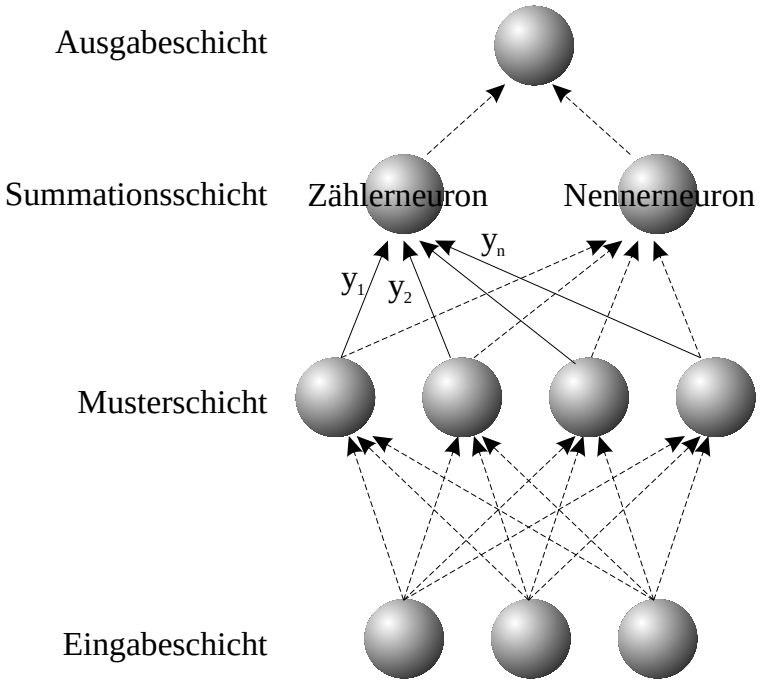

Abbildung 4.1: *General Regression Neural Network als neuronales Netz: Der Informationsfluß verläuft von unten nach oben. Die gestrichelten Pfeile symbolisieren Gewichtsfaktoren der Stärke Eins.*

gen. Das Hauptmotiv, eine bereits funktionierende Approximationsformel auf etwas künstlich anmutende Art und Weise in die Struktur eines neuronalen feedforward–Netzes zu bringen, ist die Möglichkeit der Implementation auf parallelen Prozessoren mit den damit verbundenen Geschwindigkeitsvorteilen. Es werden zwar viele $(p + n + 3)$ Neuronen/Prozessoren benötigt, die aber nur elementare Operationen durchführen müssen.

Der Informationsfluß des Netzes in Abbildung 4.1 verläuft von unten nach oben:

1. *Eingabeschicht*: Sie besteht aus p linearen Neuronen, an die jeder Eingabevektor $\mathbf{X}$ angelegt wird.

2. *Musterschicht*: Die Musterschicht hat genau n Neuronen, d.h. für jedes Muster aus dem Trainingsset ein eigenes Neuron. Die Musterschicht ist mit der Eingabeschicht durch Verbindungen der Stärke Eins vollständig verknüpft. Die Kernel–Funktion des i.Neurons berechnet erst die Distanz $d = d_x(\mathbf{X}, \mathbf{X}^i)$ und dann als Aktivierungsfunktion die Kernel–Funktion $W\left(\frac{d}{\sigma^i}\right)$.

3. *Summationsschicht*: Die Summationsschicht besteht aus je einem Neuron für Zähler und Nenner aus Gleichung (4.19). Die Verbindungen zwischen der Musterschicht und dem „Nennerneuron“ haben die Stärke Eins; die Verbindung

zwischen dem i.Musterneuron und dem „Zählerneuron“ hat den Wert der abhängigen Variable Y^i.

4. *Ausgabeschicht*: Das eine Ausgabeneuron ist mit der Summationsschicht durch Verbindungen der Stärke Eins verknüpft. Das Ausgabeneuron muß die Division der Ausgabewerte von Zähler- und Nennerneuron der Summationsschicht durchführen. Sieht man ein künstliches neuronales Netz als biologisches Modell an, so ist die Divisionaufgabe des Ausgabeneurons ein gewichtiger (der einzige) Einwand, da die Division im Gegensatz zu den Aktivierungsfunktionen üblicher künstlicher neuronaler Netze biologisch nur schwer zu motivieren ist.

Die Schemazeichnung 4.1 zeigt auf eindrückliche Weise die Vorteile des General Regression Neural Networks:

- Die Architektur ist vollständig determiniert.
- Alle Gewichtsfaktoren des Netzes sind vollständig festgelegt. Der Übersichtlichkeit halber sind alle Verbindungen der Stärke Eins durch gestrichelte Pfeile dargestellt.
- Die einzigen freien zu trainierenden Parameter sind die n Skalierungsfaktoren σ^i der Kernel–Funktion.

4.7 Arbeitsweise eines GRNN

Zum leichteren Verständnis soll hier an einem einfachen, eindimensionalen Beispiel die Arbeitsweise eines GRNN nach Gleichung (4.20) demonstriert werden.

In Abbildung 4.2 wurden die Trainingsdaten als Sterne, die Kernel–Funktionen als gestrichelte und die Approximationskurve als durchgezogene Linie eingezeichnet. Für die Standardabweichungen der Kernel–Funktionen wurde ein für alle Funktionen identischer Wert von $\sigma = 0.65$ gewählt. Es ist unmittelbar einsichtig, wie die Überlagerung der GAUSS–Funktionen nach Gleichung (4.20) die Approximationskurve in Abbildung 4.2 ergibt.

Werden die Breiten der Kernel–Funktionen falsch gewählt, so ist das Approximationsergebnis ungenügend:

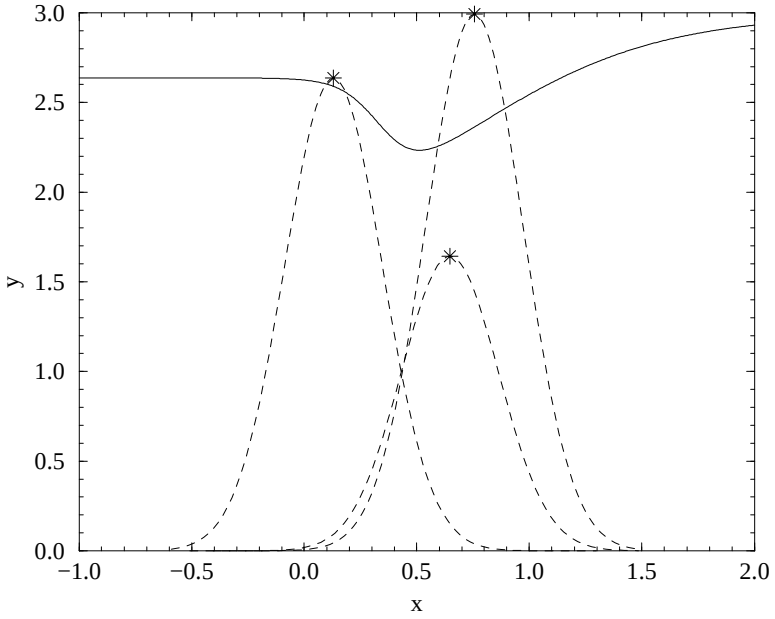

Abbildung 4.2: *Approximation durch ein GRNN*

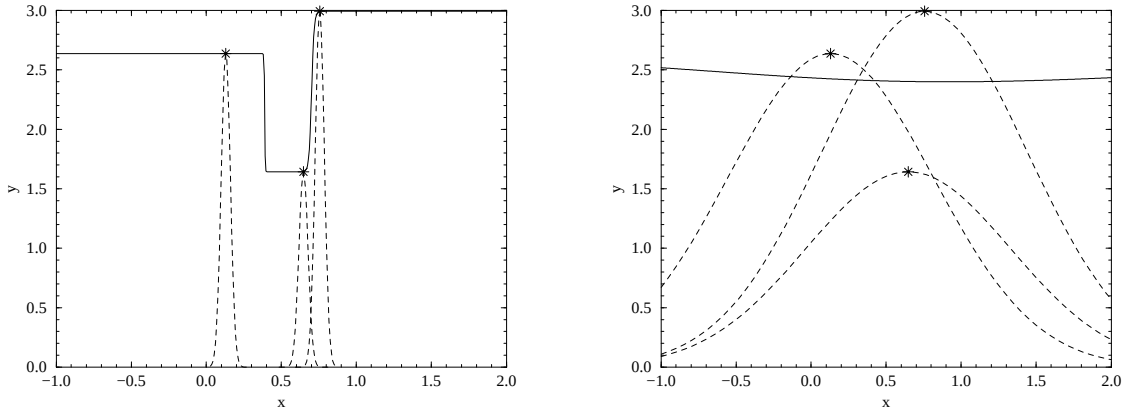

Abbildung 4.3: *Ungünstige Breiten der Kernel–Funktion*

1. Im Falle einer zu kleinen Parameterwahl neigt das GRNN — wie andere Netze auch — zum Überfitten, d.h. die Trainingsdaten werden im Wesentlichen auswendiggelernt und kaum verallgemeinert. Dieses Verhalten zeigt das linke Diagramm in Abbildung 4.3. Wenn sehr viele, unverrauschte Daten vorliegen, kann dies tolerabel sein. In der Regel ist allerdings der zu analysierende Datensatz zum einen im Vergleich zur Größe des Suchraums sehr klein und zum anderen fehlerhaft.

2. Werden die Kernel–Breiten jedoch zu groß gewählt, so glättet der Fit zu stark. Im Extremfall unendlich großer σ-Werte besteht die Fitfunktion für alle Elemente der Definitionsmenge aus dem Mittelwert der Trainingsdaten, wie das rechte Diagramm in Abbildung 4.3 veranschaulicht.

Nach Definition (4.19) ist das GRNN prinzipiell in der Lage, für jeden beliebigen Eingabevektor eine Ausgabe zu liefern. In der Praxis können dabei allerdings numerische Instabilitäten auftreten, weil die Fensterfunktionen — und damit der Nenner — für steigende Distanzen nach den Gleichungen (4.3) und (4.4) gegen Null konvergieren. Analytisch ist dies kein Problem, da der Zähler ebenfalls gegen Null strebt und ein endlicher Grenzwert existiert. Auf Seite 61 wurden bei der Auflistung der Eigenschaften von Fundamentalgleichung (4.19) bereits die Grenzwerte im Fall gleicher und unterschiedlicher Kernelbreiten angegeben.

Abbildung 4.4 zeigt das charakteristische Verhalten des GRNN für einen typischen Bereich der x-Werte. Hier ist die Kernel–Breite des mittleren Punktes größer als die der anderen.

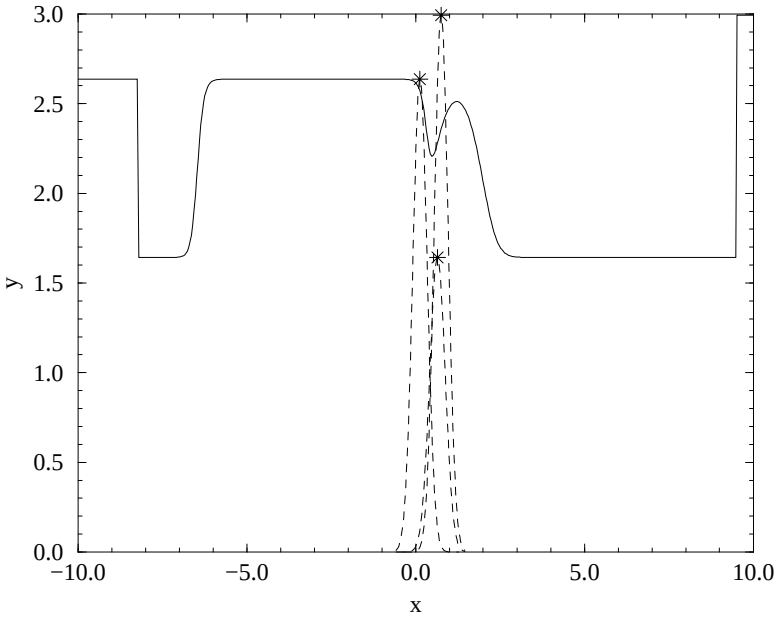

Abbildung 4.4: *Numerische Instabilitäten des GRNN*

1. Liegt x in einem engen Bereich um die Trainingsmuster (hier $x \in [-1; +1]$), liefern die meisten Kernelfunktionen einen Wert ungleich Null. Damit approximiert das GRNN erwartungsgemäß.

2. Etwas weiter entfernt, wenn nur noch wenige Kernelfunktionen — vor allem die des nächstgelegenen, dominierenden Musters — einen Wert ungleich Null liefern (hier $x \in [-6; -1]$), nimmt die Ausgabefunktion den y-Wert dieses Trainingsmusters an.

3. Noch weiter entfernt, wenn nur noch eine Kernelfunktion — die des Musters mit der größten Kernelbreite — einen Wert ungleich Null liefert (hier $x \in [-8; -6]$ und $x \in [+1; +9.5]$), nimmt die Ausgabefunktion den y-Wert dieses Trainingsmusters (hier des mittleren Punktes) an.

4. Ist x so groß, daß alle Kernelfunktionen einen Wert von Null liefern (hier $x < -8$ und $x > 9.5$), müßte laut Gleichung (4.19) Null durch Null dividiert werden. Aufgrund der analytisch bekannten Grenzwerte wird die Ausgabe des y-Wertes des nächstgelegenen Trainingsmusters erzwungen.

Diese Eigenschaften von General Regression Neural Networks können im Falle unterschiedlicher Kernelbreiten σ^i zu bestimmten Artefakten führen, wie Abbildung 4.5 demonstriert.

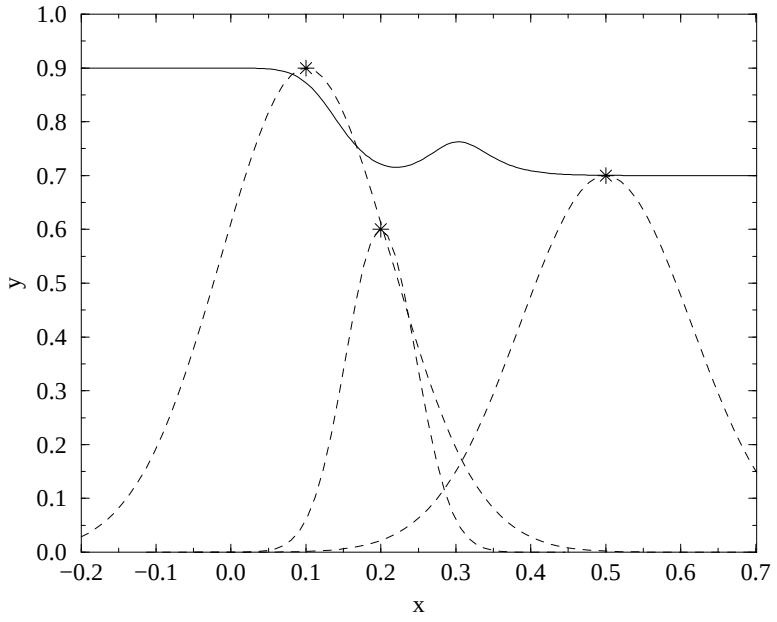

Abbildung 4.5: *Artefakte des GRNN. Im Falle stark unterschiedlicher Kernelbreiten σ^i sowie dünnverteilter Datensätze können künstliche Extrema durch das GRNN generiert werden.*

Der mittlere Punkt liegt näher am linken als am rechten Punkt und hat eine etwas kleinere Kernelbreite als die anderen Punkte. Im Bereich zwischen dem mittleren und dem rechten Punkt befindet sich ein Maximum, das durch die drei Trainingspunkte alleine nicht zu motivieren ist. Das Maximum entsteht durch die „Dominanz“ des linken Punkts an der Stelle $x \approx 0.3$: Durch die kleinere Kernelbreite des mittleren Punkts und dem größeren Funktionswert des linken Punkts überlagert der linke Punkt den Einfluß des mittleren.

Dieser Effekt entsteht im allgemeinen bei stark unterschiedlichen Kernelbreiten und ungleichmäßig verteilten Stützstellen. Je mehr Dimensionen die unabhängige Variable $\mathbf{x}$ hat, desto geringer ist die Bedeutung dieses Effektes. Der Grund ist, daß die Breite der Verteilung der Distanzen zweier zufällig ausgewählter Punkte mit zunehmender Anzahl Dimensionen immer schmaler wird, die Stützstellen also immer äquidistanter verteilt sind — näheres dazu in Anhang B.

4.8 Vergleich von GRNN und RBF–Netzen

Da sowohl das GRNN als auch Radiale-Basisfunktionen–Netze (RBF–Netze — siehe Anhang D.2.2.3) ihre Ausgabe durch Überlagerung von (radial–symmetrischen) Kernelfunktionen erzeugen, lohnt es sich, die Unterschiede zwischen beiden Netztypen genauer zu beleuchten.

Ein RBF–Netz verteilt eine vorzugebende Anzahl an Kernelfunktionen (meist GAUSS–Funktionen) im Trainingsraum und berechnet den Ausgabewert durch eine Linearkombination der Kernelfunktionen. Die Entwicklungsfaktoren (= Gewichte zwischen verdeckter Schicht und Ausgabeschicht) und die charakteristischen Parameter der Kernelfunktionen sind Gegenstand des Trainings. Die Anzahl und Lage der Kernelfunktionen kann, muß im Gegensatz zum GRNN aber nicht, mit den Trainingsdaten übereinstimmen. In der Standardvariante erfolgt das Training der Entwicklungsfaktoren durch das (evtl. approximative) Lösen eines linearen Gleichungssystems. Fortgeschrittene Trainingsverfahren versuchen, Anzahl, Lagen und Breiten der Kernelfunktionen sowie die Entwicklungsfaktoren zu optimieren.

Nach Kapitel 4.7 weist das GRNN jedem Trainingsmuster eine Kernelfunktion zu, deren Maximalwert an der Position des Trainingsmusters liegt. Auch hier wird die Ausgabe im Wesentlichen durch eine Linearkombination der Kernelfunktionen erreicht. Allerdings werden als Entwicklungsfaktoren die zu lernenden Y^i verwendet. Die Ausgabe wird durch Division mit der Summe aller Kernelfunktionen normiert.

Im Gegensatz zum RBF–Netz ist beim GRNN die Architektur zusammen mit den Gewichtsfaktoren genau definiert. Lediglich die charakteristischen Parameter der Kernelfunktionen (Kernelbreiten σ^i) müssen trainiert werden — dies ist Aufgabe des folgenden Kapitels.

4.9 Trainingsverfahren

Für die Anwendung künstlicher neuronaler feedforward–Netze ist die Konstruktion der geeigneten Netzwerkarchitektur und das Training ihrer freien Parameter ein wichtiges Forschungsthema [132]. Neben rein theoretischen Arbeiten wurde bereits viel Forschungs- und Entwicklungsarbeit zur Beschleunigung und Automatisierung des populären Backpropagation–Verfahrens geleistet; ein nennenswerter Teil der Veröffentlichungen [34, 91, 132] über neuronale Netze beschäftigt sich mit diesem Thema.

Für das GRNN existieren verschiedene Trainingsmethoden [8, 9, 10, 71, 115, 116, 117]: Meist wird dabei entweder nur mit einer einzelnen Kernelbreite σ für alle Trainingspunkte gearbeitet, oder es werden Varianten von Gradientenverfahren eingesetzt. Eine einzelne Kernelbreite führt nicht zur maximal möglichen Generalisierung, und Gradientenverfahren benötigen einen Rechenaufwand der Ordnung $O(n^3)$ bei n Trainingsmustern (siehe Seite 72). Deshalb wurden in dieser Arbeit spezielle, an das GRNN angepaßte Trainingsverfahren entwickelt.

Im folgenden werden nach den vorbereitenden Kapiteln 4.9.1 (Definition der Fehlerfunktion) und 4.9.2 (Bereich der Kernelbreiten) ein einfaches und schnelles Trainingsverfahren (Kapitel 4.9.3) sowie darauf aufbauend ein rechenintensives, aber präziseres Trainingsverfahren vorgestellt (Kapitel 4.9.4).

4.9.1 Fehlerfunktion

Trainingsmethoden gehören zur Klasse der Optimierverfahren. Wie bei allen Optimierverfahren muß deshalb auch bei Trainingsmethoden das Optimierkriterium definiert werden. Wir wollen uns an dieser Stelle auf Regressionsverfahren beschränken. Das naheliegendste Kriterium ist die Summe der Quadrate der Abweichung zwischen Ist- und Sollwert über *alle* Punkte[1]. Dieses Kriterium ist dann geeignet, wenn

[1]Die quadratische Abweichung zwischen Ist- und Sollwert ist nicht die einzig mögliche Fehlerfunktion. Sinnvoll kann auch beispielsweise der Korrelationskoeffizient zwischen Soll- und Istwert sein (siehe z.B. [36]).

dem Regressionsverfahren ein Modell zugrundeliegt, wie z.B. eine lineare Beziehung zwischen abhängiger und unabhängiger Variable — dem üblichen „least-squares"–Fit. Bei einem modellfreien Fitverfahren führt dieses Kriterium unweigerlich zum einfachen Auswendiglernen („overfitting") aller Punkte. Das Trainingsziel muß aber sein, aus den vorgegebenen Punkten die Trends zu extrahieren und Rauschen zu eliminieren, also generalisiert oder verallgemeinert zu lernen.

Als natürliche und in der Regel verwendete Erweiterung des least-squares–Kriteriums werden die vorgegebenen Punkte in ein „Trainingsset" und ein „Überprüfungsset" aufgeteilt — „crossvalidation" genannt. Zur Erhöhung der statistischen Signifikanz kann dies S-mal geschehen.

Häufig wird *vor* der Aufteilung in Trainings- und Validierungsset dem Datensatz ein Testset entnommen. Das Testset darf an keiner Stelle eines Trainingsverfahrens eingesetzt werden und dient der Überprüfung der Generalisierfähigkeit des fertig trainierten statistischen Verfahrens.

Für die Trainingsfehler E_T des Trainingssets und E_V des Überprüfungssets ergeben sich folgende Beziehungen:

$$E_T(p_1,\ldots,p_k) := \frac{1}{S\cdot(n-V)}\sum_{s=1}^{S}\sum_{i=1}^{n-V}\left[R_s\left(\mathbf{X}_{t_{i,s}},p_1,\ldots,p_k\right)-Y_{t_{i,s}}\right]^2 \quad (4.21)$$

$$E_V(p_1,\ldots,p_k) := \frac{1}{S\cdot V}\sum_{s=1}^{S}\sum_{j=1}^{V}\left[R_s\left(\mathbf{X}_{v_{j,s}},p_1,\ldots,p_k\right)-Y_{v_{j,s}}\right]^2 \quad (4.22)$$

mit
$R_s(\mathbf{X},p_1,\ldots,p_k)$: Regression von $\mathbf{X}$ auf der Basis des s.Trainingssets
$p_1,\ldots,p_k$: k freie Parameter der Regressionsfunktion R_s
S : Anzahl unterschiedlicher Setpaare
V : Größe jedes Überprüfungssets
$t_{i,s}$: indiziert das i.Element des s.Trainingssets
$v_{j,s}$: indiziert das j.Element des s.Überprüfungssets
Y_{index} : zu lernender Wert

Die Funktion $R_s(\mathbf{X},p_1,\ldots,p_k)$ beschreibt das eigentliche Regressionsverfahren, beispielsweise ein neuronales Netz oder aber auch Gleichung (4.19) des GRNN. Die k freien Parameter $p_1,\ldots,p_k$ des Regressionsverfahrens sind der eigentliche Gegenstand des Trainings. Sie müssen so optimiert werden, daß der Fehler E_V des *Überprüfungssets* minimiert wird. Bei künstlichen neuronalen Netzen entsprechen ihnen

die Gewichtsfaktoren der Verbindungen zwischen den Neuronen. Hier sind es die Kernelbreiten σ^i.

Im Kern ist das Optimierkriterium E_V identisch mit der Fehlerfunktion von Backpropagation–Netzen. Der einzige Unterschied ist, daß hier S Trainings-/Überprüfungssetpaare erlaubt sind. Ein feedforward–Netz kann nur mit einem einzigen Setpaar arbeiten: Mit dem Trainingsset wird trainiert, mit dem Überprüfungsset der Generalisierungsfehler (nicht Trainingsfehler!) überprüft. Ziel eines jeden Trainingsverfahrens ist die Minimierung des Generalisierungsfehlers. Für die üblichen feedforward–Netze sind mehrere Setpaare nutzlos, da jedem Trainingsset/Überprüfungssetpaar ein eigenes Netz zugeordnet werden müßte. Für das GRNN gilt dies nicht, da beliebig viele Setpaare mit Gleichung (4.19) verarbeitet werden können.

Implizite Annahme ist hier und im Folgenden, daß das Trainingsset und das Überprüfungsset *repräsentativ* für den vollständigen Datenraum sind. Nur mit dieser Voraussetzung führt ein Minimieren des Trainingsfehlers E_V zu besserer Generalisierung. Besitzen Trainingsset/Überprüfungsset einen „Bias“, so decken sie nur einen Teil des Datenraums ab. In solchen Fällen ist es möglich, daß die optimale Generalisierung eines unabhängigen Testsets bei einem größeren als dem minimalen Trainingsfehler erfolgt. Beispiele dazu finden sich in Kapitel 4.10.2 (z.B. Abbildung 4.13).

Man beachte, daß in die Berechnung von E_T und E_V nach Gleichung (4.22) die Fehler *aller* verfügbaren Datenpaare eingehen. Das Training findet demzufolge im „Batch–Mode“ statt. Im Gegensatz dazu existieren auch „online“–Trainingsverfahren, bei denen die zu trainierenden Parameter nach Präsentation eines jeden einzelnen Datenpaares aktualisiert werden. Diese Varianten finden hier keine Berücksichtigung.

In den meisten bekannten Trainingsverfahren werden die Trainingsfehler E_T und E_V auf eine *vorzugebende* Anzahl Nachkommastellen gerundet. Auf diese Weise wird implizit ein Abbruchkriterium für das Training definiert: Sobald E_T und/oder E_V ihren Wert nicht mehr verändern, hat das Verfahren konvergiert.

Donald F. Specht schlägt in [114] die sog. „holdout“– oder „leave-one-out“–Methode vor. Die vorgegebenen Punkte werden $S = n$ mal in Trainings- und Überprüfungsset aufgeteilt, deshalb kann die holdout–Methode als Extremfall von crossvalidation betrachtet werden. Das Überprüfungsset besteht aus jeweils einem einzelnen Punkt, das Trainingsset aus den verbleibenden Punkten. Auf diese Weise entspricht jedem Punkt genau einmal ein Überprüfungsset.

Die Berechnung des Regressionsfehlers E_V benötigt $S \cdot V$ Aufrufe von R_s. Für das GRNN muß bei jeder Regression R_s in Gleichung (4.19) $n - V$ mal die Kernelfunktion W aufgerufen werden. Die gesamte Anzahl Rechenschritte zur Bestimmung des Regressionsfehlers E_V beträgt damit für das GRNN $S \cdot V \cdot (n - V)$. Je nach Verteilung der Punkte auf die Trainings- und Überprüfungssets ergibt sich folgender Rechenaufwand (= Anzahl Aufrufe der Kernelfunktion W):

$$\begin{aligned} \text{holdout}: &\ n(n-1) \\ \text{crossvalidation}: &\ S \cdot \gamma(\gamma-1) \cdot n^2 \\ \text{mit} \quad \gamma: &\ \text{Anteil des Trainingssets an der Gesamtzahl von Punkten } (0 \leq \gamma \leq 1) \end{aligned}$$

In jedem Fall ist der Rechenaufwand zur Bestimmung des Trainingsfehlers von der Ordnung $O(n^2)$.

Vergleicht man den Rechenaufwand von holdout und crossvalidation, so ist der Aufwand für holdout dann geringer, wenn folgende Beziehung für die Anzahl an Setpaaren gilt:

$$S \geq \frac{n-1}{\gamma(\gamma-1)n} \quad \stackrel{n\to\infty}{\longrightarrow} \quad \frac{1}{\gamma(\gamma-1)} \tag{4.23}$$

Der Rechenaufwand ist für das holdout–Kriterium nur dann geringer als für crossvalidation, wenn $S \geq 4$ gilt, da der Ausdruck $\gamma(\gamma-1)$ für $\gamma = \frac{1}{2}$ maximal ist. Für andere Werte von γ können noch wesentlich mehr Setpaare verwendet werden, bis crossvalidation ungünstiger als holdout wird. Allerdings ist es aus statistischen Gründen sinnvoll, Trainings- und Überprüfungsset gleich groß, d.h. $\gamma = \frac{1}{2}$ zu wählen.

Bei vielen Versuchen erwies es sich als hilfreich, für Trainingsläufe mit crossvalidation eine möglichst große Anzahl Setpaare zu verwenden. Ab $S = 5$ oder mehr Setpaaren unterschieden sich die Lagen der Trainingsoptima kaum mehr voneinander und der Rechenaufwand ist dann für $\gamma = \frac{1}{2}$ vergleichbar mit holdout. Diese empirischen Ergebnisse sowie Aussagen aus der Literatur [71, 114] führten zur Entscheidung, die holdout–Methode als Fehlerfunktion zu übernehmen. Ein weiteres Argument zugunsten von holdout ist die Tatsache, daß die crossvalidation–Methode im Gegensatz zu holdout zwei benutzerdefinierte Parameter S und γ benötigt.

4.9.2 Bereich der Kernelbreiten

Bei verschiedenen Trainingsversuchen zeigte sich sehr schnell die Notwendigkeit, für die Kernelbreiten σ^i minimale und maximale Werte zu finden. Entscheidend ist dabei, zu garantieren, daß die ausgewählten Grenzen den vermutlich optimalen Wert

von σ^i umrahmen. Ziel war es zudem, daß die minimalen Werte für die Kernelbreiten möglichst nahe am Optimum liegen, so daß das rechenintensive Training vereinfacht wird. Die minimalen Kernelbreiten sollen so groß sein, daß das GRNN für keines der zur Verfügung stehenden Trainingsmuster numerische Instabilitäten (siehe Abbildung 4.4) aufweist.

Da es keine rein mathematisch ableitbaren Grenzen gibt, mußte mit Heuristiken gearbeitet werden. In dieser Arbeit wurden verschiedene empirische Regeln getestet. Die zwei sich am besten bewährenden sollen hier vorgestellt werden:

4.9.2.1 Heuristik I

Die eine Grundidee zur Bestimmung der unteren Grenze ist, die Kernelfunktion so schmal werden zu lassen, daß sie den nächsten benachbarten Punkt „gerade noch berührt“. Im Falle des GAUSS'schen Kernels heißt das:

$$Y^i \cdot \exp\left[-\frac{D^2_{min,i}}{2(\sigma^i_{min})^2}\right] = Y_B \tag{4.24}$$

mit $D^2_{min,i}$: Abstand zum nächstgelegenen Punkt

Y_B : Wert der Kernelfunktion an der Stelle des nächstgelegenen Punktes

Damit ergibt sich als untere Grenze der i.Kernelbreite:

$$\sigma^i_{min} = \sqrt{-\frac{D^2_{min,i}}{2\ln\left(\frac{Y_B}{Y^i}\right)}} \tag{4.25}$$

Analoge Überlegungen zum maximal erlaubten Wert der Kernelbreite — jeder Punkt soll gerade noch den am weitesten von ihm entfernten Punkt berühren dürfen — führen zur oberen Grenze der i.Kernelbreite:

$$\sigma^i_{max} = \sqrt{-\frac{D^2_{max,i}}{2\ln\left(\frac{Y_B}{Y^i}\right)}} \tag{4.26}$$

Die Stärke der „Berührung“ mit dem Nachbarpunkt steckt in der willkürlich zu wählenden Größe Y_B. Je kleiner der Wert ist, desto weniger werden sich die Kernelfunktionen zweier benachbarter Punkte überlappen. Der Wert ist so zu wählen, daß die Punkte einerseits (auf Wunsch) noch durch das Training voneinander zu trennen sind, andererseits der Fit auch im Falle minimaler Kernelbreiten noch „glatt“ bleibt (siehe Abbildung 4.3 auf Seite 65 — linkes Diagramm).

In Kapitel 4.4 wurde bereits angesprochen, daß die Variablen X_p^i auf gleiche Standardabweichung zu normieren sind. Für die abhängige Variable Y^i existiert eine solche Forderung nicht. Es ist für die Heuristik I allerdings notwendig, die Y^i in einen Zahlenbereich ohne Null zu skalieren, damit es nicht zu Problemen mit der Division durch Null in den Gleichungen (4.25) und (4.26) kommen kann.

4.9.2.2 Heuristik II

Im Eindimensionalen ist die einfachste Möglichkeit der Interpolation das Verbinden der jeweils benachbarten Punkte durch Geradenstücke. Die Idee ist nun, für jeweils zwei benachbarte Punkte die Kernelbreite so einzustellen, daß die Approximation des GRNN der Verbindungsgerade „möglichst nahe kommt". Konkret heißt dies, daß der Betrag des Gradienten der Regression des GRNN in der Mitte des Verbindungsvektors zwischen $\mathbf{X}^1$ und $\mathbf{X}^2$ mit dem Betrag der Steigung der Verbindungsgeraden gleichgesetzt und nach σ aufgelöst wird.

Die Regressionsformel des GRNN lautet in p Dimensionen nach Gleichung (4.19) für zwei Punkte:

$$\hat{y}(\mathbf{X}) = \frac{Y^1 \cdot W\left(\frac{d_x(\mathbf{X},\mathbf{X}^1)}{\sigma}\right) + Y^2 \cdot W\left(\frac{d_x(\mathbf{X},\mathbf{X}^2)}{\sigma}\right)}{W\left(\frac{d_x(\mathbf{X},\mathbf{X}^1)}{\sigma}\right) + W\left(\frac{d_x(\mathbf{X},\mathbf{X}^2)}{\sigma}\right)} \tag{4.27}$$

Der Gradient der Regressionsformel (4.27) für zwei Punkte an der Stelle $\mathbf{X}$ lautet:

$$\vec{\nabla}\hat{y}(\mathbf{X}) = (Y^1 - Y^2)\,\frac{\vec{W}_1'\,W_2 - \vec{W}_2'\,W_1}{(W_1 + W_2)^2} \tag{4.28}$$

$$\text{mit} \qquad W_1 := W\left(\frac{d_x(\mathbf{X},\mathbf{X}^1)}{\sigma}\right); \quad \vec{W}_1' := \vec{\nabla}\,W\left(\frac{d_x(\mathbf{X},\mathbf{X}^1)}{\sigma}\right)\Bigg|_{\mathbf{X}}$$
$$W_2 := W\left(\frac{d_x(\mathbf{X},\mathbf{X}^2)}{\sigma}\right); \quad \vec{W}_2' := \vec{\nabla}\,W\left(\frac{d_x(\mathbf{X},\mathbf{X}^2)}{\sigma}\right)\Bigg|_{\mathbf{X}}$$

Es gelte für die „Mitte" $\mathbf{X}^0$ zwischen $\mathbf{X}^1$ und $\mathbf{X}^2$:

$$d(\mathbf{X}^0,\mathbf{X}^1) = d(\mathbf{X}^0,\mathbf{X}^2) = \frac{1}{2}d(\mathbf{X}^1,\mathbf{X}^2) \tag{4.29}$$

Mit dem Spezialfall der GAUSS'schen Fensterfunktion lautet der Gradient an der Stelle $\mathbf{X}^0$:

$$\vec{\nabla}\hat{y}(\mathbf{X}^0) = (Y^1 - Y^2)\,\frac{d_x(\mathbf{X}^1,\mathbf{X}^2)}{8\sigma^2}\left(\vec{\nabla}\,d_x(\mathbf{X},\mathbf{X}^2)\Big|_{\mathbf{X}^0} - \vec{\nabla}\,d_x(\mathbf{X},\mathbf{X}^1)\Big|_{\mathbf{X}^0}\right) \tag{4.30}$$

Durch Gleichsetzen von $\left\|\vec{\nabla}\hat{y}(\mathbf{X}^0)\right\|$ mit dem Betrag $\frac{\left\|Y^1-Y^2\right\|}{d_x(\mathbf{X}^1,\mathbf{X}^2)}$ erhalten wir den Wert für σ, für den die Steigung der Regressionfunktion (4.27) in der Mitte $\mathbf{X}^0$ zwischen den beiden Stützstellen genau der Steigung der Geraden zwischen den Punkten $(\mathbf{X}^1, Y^1)$ und $(\mathbf{X}^2, Y^2)$ entspricht:

$$\sigma = \frac{1}{2} d_x(\mathbf{X}^1, \mathbf{X}^2) \sqrt{\frac{\left\|\vec{\nabla}\, d_x(\mathbf{X}, \mathbf{X}^2)|_{\mathbf{X}^0} - \vec{\nabla}\, d_x(\mathbf{X}, \mathbf{X}^1)|_{\mathbf{X}^0}\right\|}{2}} \tag{4.31}$$

Im Gegensatz zu Gleichung (4.25) von Heuristik I gehen hier nur die Metrik zwischen den zwei Punkten $\mathbf{X}^1, \mathbf{X}^2$ und ihr Gradient an der Stelle $\mathbf{X}^0$ und nicht die Funktionswerte Y^1, Y^2 ein.

Setzt man den Gradient der EUCLID'schen Metrik

$$\vec{\nabla} d_x(\mathbf{X}, \mathbf{X}') = \frac{\mathbf{X} - \mathbf{X}'}{d_x(\mathbf{X}, \mathbf{X}')} \tag{4.32}$$

zusammen mit den Bedingungen (4.29) für $\mathbf{X}^0$ in Gleichung (4.31) ein, so verschwindet der Term unter der Quadratwurzel.

Der Ausdruck für den Gradienten der Regressionsformel lautet außerdem:

$$\vec{\nabla}\hat{y}(\mathbf{X}^0) = \frac{(Y^1 - Y^2)}{4\sigma^2} (\mathbf{X}^1 - \mathbf{X}^2) \tag{4.33}$$

$$\text{bzw.} \quad \left|\vec{\nabla}\hat{y}(\mathbf{X}^0)\right| = \frac{\left|Y^1 - Y^2\right|}{4\sigma^2} d_x(\mathbf{X}^1, \mathbf{X}^2) \tag{4.34}$$

Für die GAUSS'sche Fensterfunktion und die EUCLID'sche Metrik lautet demnach die Kernelbreite σ:

$$\sigma = \frac{1}{2} d_x(\mathbf{X}^1, \mathbf{X}^2) \tag{4.35}$$

Man geht nun durch die Liste aller Trainingspunkte und bestimmt die minimalen und maximalen Werte σ^i_{min} und σ^i_{max} der Kernelbreite für den jeweiligen Punkt und seinen nächsten Nachbarn nach Gleichung (4.35). Falls einer der Punkte der nächste Nachbar von mehreren Punkten ist, so wird der Mittelwert der σ-Werte genommen.

In der Praxis erwies sich Heuristik II als besser geeignet, da die damit ermittelten minimalen Kernelbreiten den optimalen Kernelbreiten nach Kapitel 4.9.4 meist näher waren als mit Heuristik I. Außerdem benötigt Heuristik II keine zusätzlichen Parameter, und die Y^i-Werte müssen nicht skaliert werden.

Die Herleitung benutzte an keiner Stelle den konkreten Wert von $\mathbf{X}^0$. Falls eine andere Metrik verwendet wird, muß deren Ableitung in Gleichung (4.31) eingesetzt werden, wodurch die Abschätzung für σ nach Gleichung (4.35) eine andere Form annehmen wird.

4.9.3 Einfaches, schnelles Training

Der Rechenaufwand zur Bestimmung des Trainingsfehlers E_V ist nach der Abschätzung auf Seite 72 bei n Trainingsmustern von der Ordnung $O(n^2)$. Bereits einfache Optimierverfahren, die jedes σ^i einzeln ändern, brauchen — beispielsweise zur Bestimmung des Gradienten — mindestens n Schritte. Der Rechenaufwand steigt damit zur Ordnung $O(n^3)$! Dieser Aufwand ist nur dann gerechtfertigt, wenn die σ^i bereits sehr gut voroptimiert wurden. Dies ist die Aufgabe der Verfahren dieses Unterkapitels.

Schnelle Trainingsverfahren dürfen nicht jedes σ^i einzeln ändern, sondern variieren pro Schritt alle σ^i auf einmal. Es bieten sich dabei zwei Methoden an:

4.9.3.1 Schnelles Training I

Die erste Methode des schnellen Trainings ist das Multiplizieren aller σ^i mit einem Faktor m. Für m gilt:

$$0 \leq m \leq \min_i \left(\frac{\sigma^i_{max}}{\sigma^i_{min}} \right) \tag{4.36}$$

Ausgehend von $m = 1$ findet sich ein (sub-)optimaler Wert von m durch lokale, parabolische Suche [84][2]. Der hier implementierte Algorithmus benötigt selten mehr als 10 Schritte bis zur Konvergenz, je nach erwünschter Präzision des Trainingsfehlers E_V.

Ein GRNN, dessen Kernelbreiten σ^i mit den minimalen Kernelbreiten σ^i_{min} initialisiert wurden, generalisiert meist überraschend gut, weil Heuristik II häufig eine brauchbare Größenverteilung der σ^i_{min} liefert. Eine deutliche Verbesserung der Leistung eines GRNN kann durch Multiplizieren aller σ^i mit einem optimalen Faktor m erreicht werden.

4.9.3.2 Schnelles Training II

Die zweite schnelle Methode, ein GRNN mit bereits befriedigenden Approximationsleistungen zu trainieren, ist, die gleiche Kernelbreite $\sigma \equiv \sigma^1 = \ldots = \sigma^n$ für jeden

[2]Einfaches, lokales Optimierverfahren in einer Dimension: Gestartet wird mit drei unterschiedlichen Punkten, in die eine Parabel eingepaßt wird. Deren Minimum/Maximum dient als neuer Punkt. Dieser und die zwei ihm am nächsten liegenden Punkte dienen als Tripel für die nächste Iteration.

Punkt zu verwenden. Es ist damit nur noch ein einzelner freier Parameter festzulegen! Je gleichförmiger die zu lernenden Punkte verteilt sind, desto besser wird das Fitverhalten in diesem Fall sein. Ein weiterer Vorteil ist, daß die Berechnung des Trainingsfehlers E_V nach Gleichung (4.22) doppelt so schnell ($\frac{n(n-1)}{2}$ Funktionsaufrufe der Kernelfunktion) durchgeführt werden kann, weil die Kernelfunktion $W\left(\frac{d_x(\mathbf{X},\mathbf{X}^i)}{\sigma}\right)$ in Gleichung (4.19) aufgrund der identischen σ für die Punkte $\mathbf{X}$ und $\mathbf{X}^i$ nur einmal berechnet werden muß.

Es erwies sich sehr bald, daß die erlaubten Bereiche für σ nach Kapitel 4.9.2 meist viele Größenordnungen überstreichen. Andererseits ist der interessante Bereich von σ für gute Approximationsleistungen vergleichsweise klein und liegt bei niedrigen Werten. Alle hier implementierten Trainingsverfahren arbeiten deshalb mit dem natürlichen Logarithmus von σ.

Der Trainingserfolg ist bei einem einzigen Parameter σ leicht zu visualisieren:

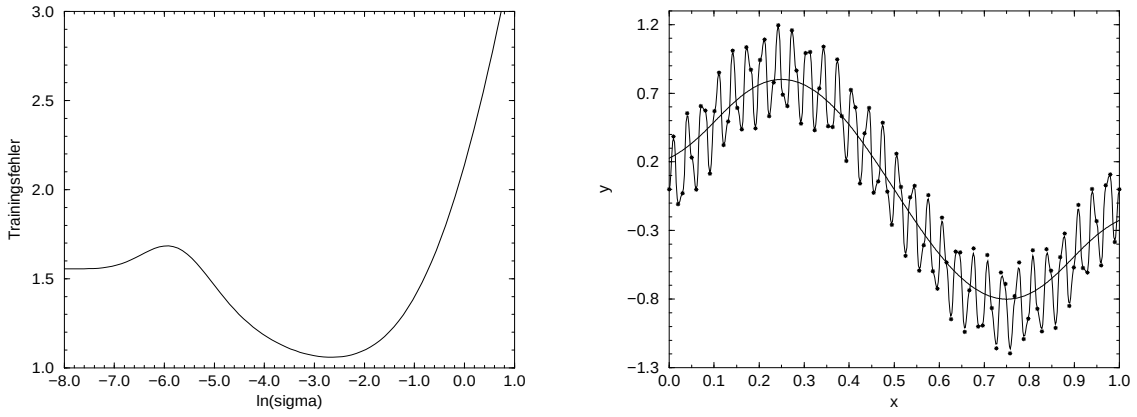

Abbildung 4.6: *Lernkurve und Trainingsergebnis des GRNN bei einer einzigen, für alle Punkte identischen Kernelbreite σ.*

In Abbildung 4.6 wurde das Training von σ an einem eindimensionalen Beispiel untersucht. Die 100 Trainingspunkte wurden in äquidistanten Abständen der Funktion $0.85 \cdot \sin(2\pi x) + 0.35 \cdot \sin(30 \cdot 2\pi x)$ entnommen und im rechten Diagramm als Sterne eingezeichnet. Im linken Diagramm ist der Trainingsfehler E_V über dem natürlichen Logarithmus von σ aufgetragen. Die Kurve demonstriert, daß — im Gegensatz zu Aussagen in [114] — die Trainingskurve mehrere Minima durchlaufen kann, deren korrespondierende Approximationskurven völlig unterschiedlich sein können. Die an den Trainingspunkten eng anliegende Kurve in rechten Bild entspricht dem kleinstmöglichen σ-Wert von $\sigma = 0.00028 = \exp(-8.181)$. Die Approximationskurve hat einen Trainingsfehler von $E_V = 1.56$. Dem zweiten lokalen und gleichzeitig

globalen Minimum der Trainingskurve ($\sigma = 0.0705 = \exp(-2.652)$) entspricht die „glatt“ durch die Trainingspunkte verlaufende Kurve mit einem Trainingsfehler von $E_V = 1.06$.

Als Trainingsstrategie bietet sich an, die eindimensionale Fehlerkurve auf ihr globales Minimum abzusuchen, in der Annahme, daß diesem die optimale Fitkurve entspricht. Diese Strategie wurde dahingehend implementiert, daß der Trainingsfehler E_V an einigen wenigen (äquidistanten) Stellen berechnet und von der Stelle mit dem geringsten Fehler ausgehend mit einem lokalen Suchalgorithmus (parabolische Suche) das globale Minimum gesucht wird. Die Anzahl Stützstellen ist ein benutzerdefinierter Wert.

4.9.3.3 Schnelles Training — der Algorithmus

Der vollständige Algorithmus zum schnellen Training eines GRNN setzt sich aus den bis jetzt beschriebenen Methoden zusammen:

1. **Initialisieren** der σ^i mit den σ^i_{min} aus Kapitel 4.9.2.
2. **Schnelles Training I**: Multiplizieren der σ^i mit dem optimalen Faktor m nach Kapitel 4.9.3.1. Diese σ^i werden als neue minimale σ^i_{min} gesetzt.
3. **Schnelles Training II**: Unabhängiges Trainieren des GRNN mit einer einzigen Kernelbreite σ für alle Punkte nach Kapitel 4.9.3.2. Nur falls σ zwischen σ^i_{min} und σ^i_{max} liegt, wird $\sigma^i = \sigma$ gesetzt. Diese Einschränkung rührt von Problemfällen, bei denen das „optimale“ σ bei $\sigma = 0$ oder $\sigma = +\infty$ liegt (siehe Kapitel 4.9.6).
4. **Schnelles Training I**: (ein zweites Mal)

Die Beispiele in Kapitel 4.10 zeigen, daß sich viele praktische Probleme bereits durch ein mit diesem Algorithmus trainiertes GRNN gut beschreiben lassen!

4.9.4 Präzises Training

Zum präziseren Training ist es erforderlich, jede Kernelbreite σ^i individuell zu optimieren. Verschiedene Methoden wurden hier getestet. Das Problem ist jedoch, daß der Rechenaufwand je nach Verfahren mindestens von der Ordnung $O(n^3)$ ist. Aus diesem Grund schieden die Optimierung durch einen genetischen Algorithmus sowie

lokales Optimieren nach dem Gradientenverfahren als unpraktikabel aus, obwohl sie sehr gute Resultate lieferten.

4.9.4.1 Grundidee

Das Prinzip ist, alle Kernelbreiten solange der Reihe nach um kleine Beträge zu *erhöhen*, bis der Trainingsfehler E_V nicht mehr kleiner wird. Es läßt sich zwar häufig auch durch Verkleinern der Kernelbreiten eine Verringerung von E_V erreichen, die daraus resultierenden Netze neigen jedoch insbesondere bei dünn verteilten Datensätzen stark zum überfitten. Die Kernelbreiten sollten so groß wie möglich und nötig gewählt werden.

Vorbereitend wird für jede Kernelbreite σ^i der Bereich zwischen $\ln(\sigma^i_{min})$ und $\ln(\sigma^i_{max})$ in eine vorgegebene Anzahl a äquidistanter Abschnitte unterteilt. a liegt je nach gewünschter Genauigkeit des Trainings und verfügbarer Rechenzeit in der Praxis zwischen 5 und 20. Die Punkte werden nach den Distanzen zu ihren jeweils nächstgelegenen Nachbarpunkten sortiert. Es werden damit zuerst die Kernelbreiten der am dichtesten beieinander liegenden Punkte erhöht.

Der Algorithmus sieht stark vereinfacht in Pseudo-Programmcode wie folgt aus:

```
forall a Abschnitte
    forall n (sortierten) Punkte
        erhöhe Kernelbreite um einen Schritt;
        if Trainingsfehler E_V größer geworden ist
            then nimm Veränderung der Kernelbreite wieder zurück;
        endif
    endforall
    if keine Kernelbreite mehr modifiziert wurde
        then fertig!
    endif
endforall
```

Es sei $R(n)$ der Rechenaufwand zum Berechnen des Trainingsfehlers E_V von n Punkten. Bei einer mittleren Anzahl Schritten $\bar{a}$ bis der Algorithmus konvergiert ist, ergibt sich ein gesamter Rechenaufwand für das Training von $\bar{a} \cdot n \cdot R(n)$. Mit der Abschätzung $R(n) = c \cdot n^2$ von Seite 72 liegt der Gesamtrechenaufwand bei $c \cdot \bar{a} \cdot n^3$ (c: Proportionalitätskonstante).

Ein Weg, diesen Rechenaufwand zu reduzieren, ist, mit einer kleinen Anzahl Punkten als aktuelles Trainingsset zu beginnen, diese nach obigem Algorithmus

zu trainieren, und so lange eine bestimmte Anzahl Punkte dazuzunehmen, bis alle trainiert sind. So kann man als zusätzliche Punkte diejenigen wählen, deren Fehler am größten ist. Unterteilt man das komplette Trainingsset in s Blöcke, werden pro Runde $\frac{n}{s}$ Punkte dem aktuellen Trainingsset zugefügt. Der Rechenaufwand dieses modifizierten Verfahrens ist $\sum_{i=1}^{s}\left[\bar{a}\cdot\frac{n}{s}\cdot R\left(i\frac{n}{s}\right)+R(n)\right]$.

Zum Vergleich des Rechenaufwands wird folgende Funktion als Verhältnis des Aufwands zwischen dem modifizierten und dem ursprünglichen Verfahren definiert:

$$V(s,\bar{a},n):=\frac{\sum_{i=1}^{s}\left[\bar{a}\cdot\frac{n}{s}\cdot R\left(i\frac{n}{s}\right)+R(n)\right]}{\bar{a}\cdot n\cdot R(n)}=\frac{s}{\bar{a}n}+\frac{1}{s}\sum_{i=1}^{s}\frac{R(i\frac{n}{s})}{R(n)} \tag{4.37}$$

Mit $R(n)=c\cdot n^2$ nimmt das Verhältnis $V(s,\bar{a},n)$ die Form

$$V(s,\bar{a},n)=\frac{s}{\bar{a}n}+\frac{1}{3}+\frac{1}{2s}+\frac{1}{6s^2} \tag{4.38}$$

an. Die Aufgabe ist nun, den Wert s für die Anzahl Blöcke zu ermitteln, für die das Verhältnis $V(s,\bar{a},n)$ minimal wird. Durch Gleichsetzen der Ableitung von $V(s,\bar{a},n)$ nach s mit Null findet sich unter Vernachlässigung höherer Potenzen von $\frac{1}{\sqrt{\bar{a}n}}$ der genäherte Ausdruck

$$s=\sqrt{\frac{\bar{a}n}{2}}+\frac{1}{3}-\frac{1}{9\sqrt{2\bar{a}n}} \tag{4.39}$$

für den das modifizierte Verfahren den geringsten Rechenaufwand im Vergleich zur ursprünglichen Variante des Algorithmus hat.

Eine implizit getroffene Annahme ist die Unabhängigkeit der mittleren Anzahl Schritte $\bar{a}$ bis der Algorithmus konvergiert ist von der aktuellen Größe des Trainingssets $i\frac{n}{s}$. In den Extremfällen $s=1$ und $s=n$ ist die Annahme falsch. Für mittlere Werte von s, wie sie Gleichung (4.39) liefert, führt die Annahme zu korrekten Trainingsergebnissen.

Beispiele: Es seien $n=500$ und $\bar{a}=5$. Nach Gleichung (4.39) wird das Trainingsset in $s=35$ Blöcke à 14 Punkte unterteilt. Bei $n=100$ Trainingspunkten wird das Trainingsset in $s=16$ Blöcke à 6 Punkte unterteilt.

Ein Nebeneffekt dieses Verfahrens ist, daß es redundante Punkte, die dem Netz keine Zusatzinformation liefern, identifizieren kann. Gerade bei großen Datensätzen kann auf diese Weise das Trainingsset teilweise drastisch verkleinert werden, ohne daß sich die Generalisierungsleistung des GRNN verschlechtert. Neben dem dadurch erleichterten Verständnis des Datensatzes beschleunigt sich auch die Berechnung der Ausgabe des GRNN, da diese nach Gleichung (4.19) von der Ordnung $O(n)$ ist.

4.9.4.2 Präzises Training — der Algorithmus

Der Algorithmus zum präzisen, vollständigen Training eines GRNN ist eine Erweiterung des Algorithmus aus Kapitel 4.9.3.3:

1. **Schnelles Training**: Trainieren der σ^i mit dem schnellen Algorithmus aus Kapitel 4.9.3.3.
2. **Präzises Training durch Maximierung der einzelnen Kernelbreiten**: Trainieren jeder einzelnen Kernelbreite σ^i mit dem Verfahren nach Kapitel 4.9.4.1.
3. **Schnelles Training I**: Multiplizieren der σ^i mit dem optimalen Faktor m nach Kapitel 4.9.3.1. Diese σ^i werden als neue minimale σ^i_{min} gesetzt.

4.9.5 Beschleunigung des Trainings

Die „Tricks“ zur Beschleunigung der hier entwickelten Trainingsverfahren sollen an dieser Stelle zusammengefaßt werden:

1. Programmierung des GRNN als Formel und *nicht* als künstliches neuronales Netz. Dadurch werden die folgenden Vereinfachungen erst ermöglicht.
2. Im Falle gleicher Kernelbreiten σ für alle Punkte muß zur Bestimmung des Trainingsfehlers E_V nur die halbe Anzahl an Kernelfunktionsaufrufen berechnet werden.
3. Eine erhebliche Beschleunigung des Trainings läßt sich durch eine vorberechnete Distanzmatrix sowie eine tabellierte Exponentialfunktion erreichen.
4. Wenn ein großer Hauptspeicher zur Verfügung steht, können viele Zwischenergebnisse gepuffert werden. Dadurch konnten bis zu 2 Größenordnungen an Geschwindigkeitszuwachs erreicht werden. Der Speicherbedarf steigt allerdings *quadratisch* mit der Größe des Trainingssets an.

4.9.6 Problemfälle

Leider existieren Datensätze mit Fehlerkurven, deren globales Minimum bei $\sigma = 0$ (siehe Abbildung 4.7) oder bei $\sigma = +\infty$ (siehe Abbildung 4.8) liegt. In diesen Fällen

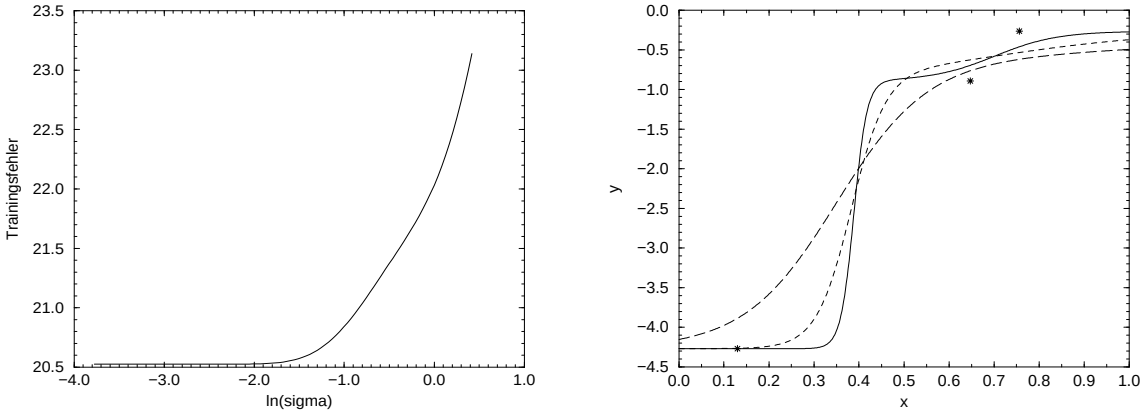

Abbildung 4.7: *Lernkurve und Trainingsergebnis — Problemfall I: Es gilt* $Y^2 > \frac{Y^1+Y^3}{2}$. *Das globale Minimum der Fehlerkurve (linkes Diagramm) liegt bei* $\sigma = 0$, *d.h. das GRNN lernt lediglich auswendig. Im rechten Diagramm sieht man die Approximationskurven für* $\sigma = 0.1353 = \exp(-2)$ *(durchgezogen),* $\sigma = 0.3679 = \exp(-1)$ *(gestrichelt) und* $\sigma = 1.0000 = \exp(0)$ *(lang gestrichelt).*

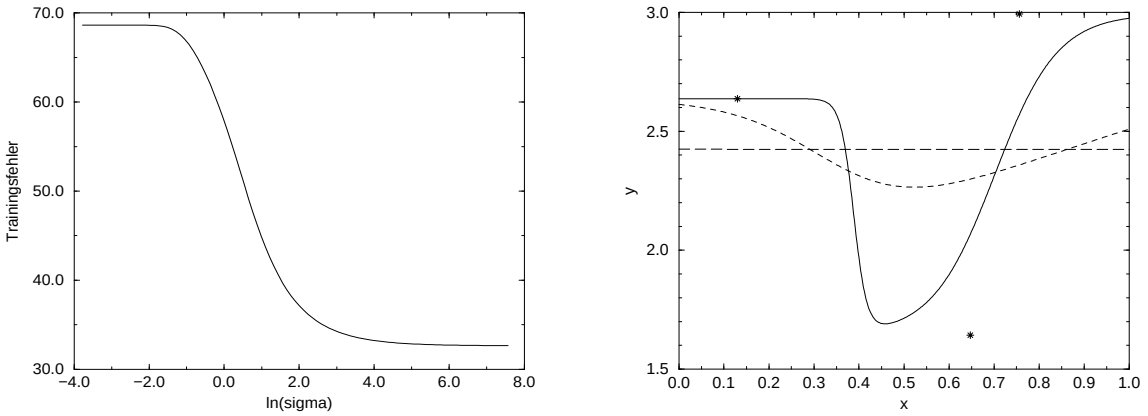

Abbildung 4.8: *Lernkurve und Trainingsergebnis — Problemfall II: Es gilt* $Y^2 < 2Y^1 - Y^3$. *Das globale Minimum der Fehlerkurve (linkes Diagramm) liegt bei* $\sigma = +\infty$, *d.h. das GRNN liefert für alle* x *den Mittelwert aller Punkte. Im rechten Diagramm sieht man die Approximationskurven für* $\sigma = 0.1353 = \exp(-2)$ *(durchgezogen),* $\sigma = 1.0000 = \exp(0)$ *(gestrichelt) und* $\sigma = 403.43 = \exp(6)$ *(lang gestrichelt).*

entstehen unbrauchbare Regressionsfunktionen, die entweder überfitten oder mit dem Mittelwert aller Punkte approximieren. Jedes Trainingsverfahren muß solche Fälle berücksichtigen.

Exemplarisch sollen die Existenz beider Fälle an einem aus den drei Punk-

ten (X^1, Y^1), (X^2, Y^2) und (X^3, Y^3) bestehenden eindimensionalen Trainingsset studiert werden. Es gilt o.B.d.A. $X^1 < X^2 < X^3$ und $X^2 > \frac{X^1+X^3}{2}$ und $Y^1 < Y^3$. Im folgenden wird untersucht, wie sich die Kurve des Trainingsfehlers für unterschiedliche Y-Werte des mittleren Punkts (X^2, Y^2) verhält. Das GRNN wird mit einer für alle drei Punkte identischen Kernelbreite $\sigma \equiv \sigma_1 = \sigma_2 = \sigma_3$ trainiert.

Nach der holdout–Methode wird die Fehlerfunktion $E_{ikl}(\sigma)$ definiert:

$$\begin{aligned} E_{ikl}(\sigma) &:= \left[\frac{Y^l \cdot W\left(X^i - X^l, \sigma\right) + Y^k \cdot W\left(X^i - X^k, \sigma\right)}{W\left(X^i - X^l, \sigma\right) + W\left(X^i - X^k, \sigma\right)} - Y^i\right]^2 \\ &= \left[\frac{Y_{li} + Y_{ki} W_{ikl}}{1 + W_{ikl}}\right]^2 \end{aligned} \tag{4.40}$$

$$\begin{aligned} \text{mit} \qquad Y_{ij} &:= Y^i - Y^j \\ X_{ij} &:= X^i - X^j \\ W_{ikl} &:= \frac{W(X_{ik}, \sigma)}{W(X_{il}, \sigma)} > 0 \end{aligned}$$

$W(X, \sigma)$ bezeichnet die Kernelfunktion; für den GAUSS–förmigen Kernel nach Gleichung (4.7) gilt die Abkürzung $W_{ikl} \equiv W\left(\sqrt{(X_{ik})^2 - (X_{il})^2}, \sigma\right)$.

Der Fehler $E_{ikl}(\sigma)$ ist die quadratische Abweichung zwischen Y^i und der Regression des GRNN an der Stelle X^i. Man beachte: Das GRNN besteht hier aufgrund der Aufteilung in Trainings- und Validierungsset nach der holdout–Methode nur aus den beiden jeweils verbleibenden Punkten (X^k, Y^k) und (X^l, Y^l). Da die Trainingspunkte in der Fundamentalgleichung des GRNN beliebig vertauschbar sind, gilt $E_{ikl}(\sigma) = E_{ilk}(\sigma)$.

Der vollständige Trainingsfehler E_V des GRNN für drei Trainingspunkte ist nach Gleichung (4.22):

$$E_V(\sigma) := \frac{1}{3}\left[E_{132}(\sigma) + E_{213}(\sigma) + E_{312}(\sigma)\right] \tag{4.41}$$

Die Ableitung von $E_{ikl}(\sigma)$ nach σ lautet:

$$\frac{dE_{ikl}(\sigma)}{d\sigma} = \underbrace{\frac{2}{\sigma^3}\frac{W_{ikl}}{(1 + W_{ikl})^3}}_{>0} \cdot \underbrace{\left((X_{ik})^2 - (X_{il})^2\right)}_{>0} \cdot Y_{kl}(Y_{li} + Y_{ki} W_{ikl}) \tag{4.42}$$

Der erste Term ist größer als Null für alle $\sigma \geq 0$ wegen der allgemeinen Eigenschaften der Kernelfunktionen. Da die Indizes k, l vertauschbar sind, ist der zweite Term für die gewählten Indexpaare $(i, k, l) = \{(1, 3, 2); (2, 1, 3); (3, 1, 2)\}$ größer Null. Der dritte Term ist verantwortlich für das Vorzeichen der Ableitung von $E_{ikl}(\sigma)$.

Eine Auswertung des Terms $Y_{kl}(Y_{li} + Y_{ki} W_{ikl})$ führt zu folgenden Bedingungen:

- $\frac{dE_{ikl}(\sigma)}{d\sigma} = 0$:
  $Y_{kl} = 0$ oder
  $\frac{Y_{ik}}{Y_{li}} > 1$, das Minimum von $E_{ikl}(\sigma)$ liegt dann an der Stelle $\sigma_0 := \sqrt{\frac{(X_{ik})^2-(X_{il})^2}{2\ln\left(\frac{Y_{ik}}{Y_{li}}\right)}}$.
  Es gilt $E_{ikl}(\sigma_0) = 0$, d.h. die Regressionkurve der Punkte mit den Indizes k und l schneidet den Punkt (X^i, Y^i).

- $\frac{dE_{ikl}(\sigma)}{d\sigma} > 0$:
  $Y_{kl} > 0 \ \wedge \ Y_{li} > 0 \ \wedge \ Y_{ki} > 0$ oder
  $Y_{kl} < 0 \ \wedge \ Y_{li} < 0 \ \wedge \ Y_{ki} < 0$

- $\frac{dE_{ikl}(\sigma)}{d\sigma} < 0$:
  $Y_{kl} > 0 \ \wedge \ Y_{li} < 0 \ \wedge \ (Y_{ki} < 0 \ \vee \ (Y_{ki} \geq 0 \ \wedge \ 2Y^i - Y^l \geq Y^k))$ oder
  $Y_{kl} < 0 \ \wedge \ Y_{li} > 0 \ \wedge \ (Y_{ki} > 0 \ \vee \ (Y_{ki} \leq 0 \ \wedge \ 2Y^i - Y^l \leq Y^k))$

Die Anwendung der Bedingungen ergibt die in Tabelle 4.1 zusammengefaßten Eigenschaften der Fehlerfunktionen $E_{132}(\sigma)$, $E_{213}(\sigma)$ und $E_{312}(\sigma)$ des Trainingssets in Abhängigkeit vom Wert Y^2. Zusätzlich sind noch die Grenzwerte der Fehlerfunktionen aufgeführt (siehe Punkt 7 auf Seite 62).

	$\sigma \to 0$	$\sigma \to +\infty$	$\frac{dE_{ikl}(\sigma)}{d\sigma} = 0$	$\frac{dE_{ikl}(\sigma)}{d\sigma} > 0$	$\frac{dE_{ikl}(\sigma)}{d\sigma} < 0$
$E_{132}(\sigma)$	$(Y^1 - Y^2)^2$	$\left[Y^1 - \frac{Y^2+Y^3}{2}\right]^2$	$Y^2 = Y^3$ $2Y^1-Y^3 < Y^2 < Y^1$	$Y^1 < Y^2 < Y^3$	$Y^2 > Y^3$ $Y^2 < 2Y^1-Y^3$
$E_{213}(\sigma)$	$(Y^2 - Y^3)^2$	$\left[Y^2 - \frac{Y^1+Y^3}{2}\right]^2$	$\frac{Y^1+Y^3}{2} < Y^2 < Y^3$	$Y^2 > Y^3$	$Y^2 \leq \frac{Y^1+Y^3}{2}$
$E_{312}(\sigma)$	$(Y^3 - Y^2)^2$	$\left[Y^3 - \frac{Y^1+Y^2}{2}\right]^2$	$Y^2 = Y^1$ $Y^3 < Y^2 < 2Y^3-Y^1$	$Y^1 < Y^2 < Y^3$	$Y^2 < Y^1$ $Y^2 > 2Y^3-Y^1$

Tabelle 4.1: *Eigenschaften der Fehlerfunktionen des GRNN–Trainings mit drei Punkten*

Liegt Y^2 im Intervall $\left[Y^1 + \frac{1}{6}(5-\sqrt{13})(Y^3-Y^1); Y^1 + \frac{1}{6}(5+\sqrt{13})(Y^3-Y^1)\right]$, läßt sich zeigen, daß $E_V(\sigma = 0) \leq E_V(\sigma \to \infty)$ gilt. In diesem Fall sind Verläufe des Trainingsfehlers E_V wie in Abbildung 4.7 möglich.

Liegt Y^2 außerhalb des Intervalls, kann der Trainingsfehler wie in Abbildung 4.8 verlaufen. Für $Y^2 < 2Y^1 - Y^3$ ist der Trainingsfehler E_V sogar streng monoton abnehmend.

Für alle Y^2, die nicht im Intervall $[Y^3; 2Y^3 - Y^1]$ liegen und größer als $2Y^1 - Y^3$ sind, setzt sich der Trainingsfehler E_V aus *zwei* Termen mit gleichem Monotonieverhalten und *einem* Term mit davon abweichendem Monotonieverhalten zusammen.

Der Trainingsfehler $E_V(\sigma)$ kann deshalb in diesen Fällen maximal ein einziges Minimum besitzen. Falls Y^2 im Intervall $[Y^3; 2Y^3 - Y^1]$ liegt, sind maximal zwei lokale Minima möglich.

Es wurde hier an einem einfachen Beispiel gezeigt, daß für den zu minimierenden Trainingsfehler E_V pathologische Fälle existieren, in denen die Suche zum globalen Minimum des Fehlers zu unbrauchbaren Regressionsfunktionen des GRNN führt.

Die Problemfälle I und II treten auch dann auf, wenn, wie beim präzisen Training (Kapitel 4.9.4), die Kernelbreiten σ^i eines jeden Punkts des Trainingssets individuell optimiert werden.

Um dieses unerwünschte Trainingsverhalten zu kompensieren, wird jedesmal, wenn ein σ^i kleiner als σ^i_{min} oder größer als σ^i_{max} wird, $\sigma^i = \sigma^i_{min}$ gesetzt!

Auch aus diesem Grund ist es wichtig, daß σ^i_{min} nach Kapitel 4.9.2 dem optimalen Wert bereits nahe kommt.

4.10 Beispiele

Eines der traditionellen, häufigsten Beispiele in der Literatur zum Testen von neuronalen Netzen ist das (verallgemeinerte) XOR–Problem. Da dort die Generalisierungsfähigkeit nicht von Bedeutung ist, sondern die Muster nur auswendig gelernt werden müssen, ist dieses Beispiel für das GRNN uninteressant, da zu einfach — es brauchen lediglich die Kernelbreiten für jeden der Trainingspunkte auf $\sigma = 0$ gesetzt werden. Mittlerweile wird dieses Problem vor allem in Arbeiten eingesetzt,

Parameter	**Wert**	**sinnvoller Wertebereich**
Anzahl Nachkommastellen des Trainingsfehlers E_V	4	$2 \rightarrow 10$
Anzahl Stützstellen für „schnelles Training II“ (Kapitel 4.9.3.2)	10	$5 \rightarrow 100$
Anzahl a äquidistanter Abschnitte für präzises Training (Kapitel 4.9.4.1)	5	$5 \rightarrow 20$
mittlere Anzahl Schritte $\bar{a}$ (Gleichung (4.39))	$\bar{a} := \frac{a}{2}$	$\frac{a}{4} \rightarrow \frac{a}{2}$

Tabelle 4.2: *Benutzerdefinierte Parameter des GRNN–Trainings*

in denen die *Architektur* von Netzen optimiert werden soll. Dies stellt beim GRNN kein Problem dar, deshalb wird auf Tests mit XOR- und verwandten Problemen verzichtet.

Alle Beispiele in diesem Kapitel wurden mit denselben, in Tabelle 4.2 festgelegten Parametern durchgeführt.

4.10.1 Separation zweier Spiralen

Dies erste Beispiel zählt in der Literatur (z.B. [59, 67]) als „hartes“ Problem für neuronale Netze, da es nicht — auch nicht teilweise — linear separabel ist. Die Aufgabe ist, zwei Datensets, die als zweidimensionale Spiralen ineinandergeschachtelt sind, zu klassifizieren. Das Testbeispiel erfreut sich einer gewissen Beliebtheit, da das Trainingsergebnis leicht zu visualisieren ist.

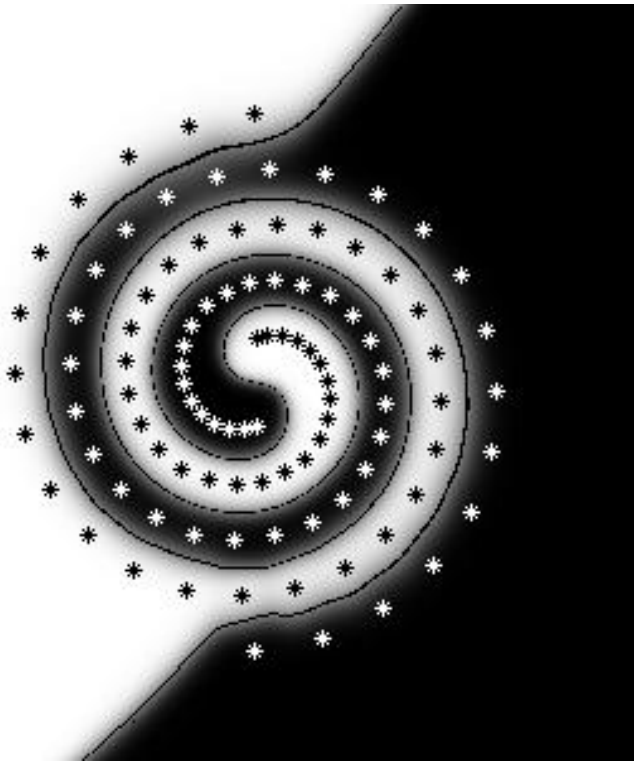

Abbildung 4.9: *Separation zweier Spiralen. Die 100 Sterne markieren das Trainingsset: Weiße Sterne haben den Wert 0, schwarze 1. Die Linie besteht aus den Punkten, an denen das GRNN den Wert 0.5 liefert, also der Grenze zwischen den Spiralen.*

In Abbildung 4.9 sind zunächst einmal beide Spiralen aus jeweils 50 Punkten zu sehen. Wegen der besseren Sichtbarkeit sind die Farben der Sterne in der Darstellung

invertiert. Die weißen (in Wirklichkeit schwarzen) Sterne entsprechen dem Wert $y = 0$, die schwarzen dem Wert $y = 1$.

Das GRNN muß die Abbildung zwischen den zweidimensionalen Koordinaten $\mathbf{X}$ und der Klasse y lernen. Zum Training wird das Problem als Approximationsaufgabe uminterpretiert. Nach abgeschlossenem Training bekommt das GRNN zum Test die Koordinaten aller Punkte der zweidimensionalen Ebene als Eingabe. Die Ausgabe des GRNN wird jeweils in einen Grauwert umgerechnet und als Bild dargestellt. Gibt das Netz den Schwellwert 0.5 aus, wird ein schwarzer Punkt zur Markierung der Grenzlinie zwischen den Klassen gezeichnet.

Es ist deutlich in Abbildung 4.9 zu erkennen, daß das Netz die Datensets optimal klassifiziert. Die Grenzlinie verläuft exakt in der Mitte zwischen beiden Datensets. Nach Kenntnis des Autors ist keines der gebräuchlichen neuronalen Netze in der Lage, automatisch (innerhalb von Sekunden) das Zwei-Spiralen–Problem mit dieser Qualität zu lösen. Die besten Ergebnisse wurden bis dato mit speziellen „Projection Pursuit“–Netzen gefunden [59].

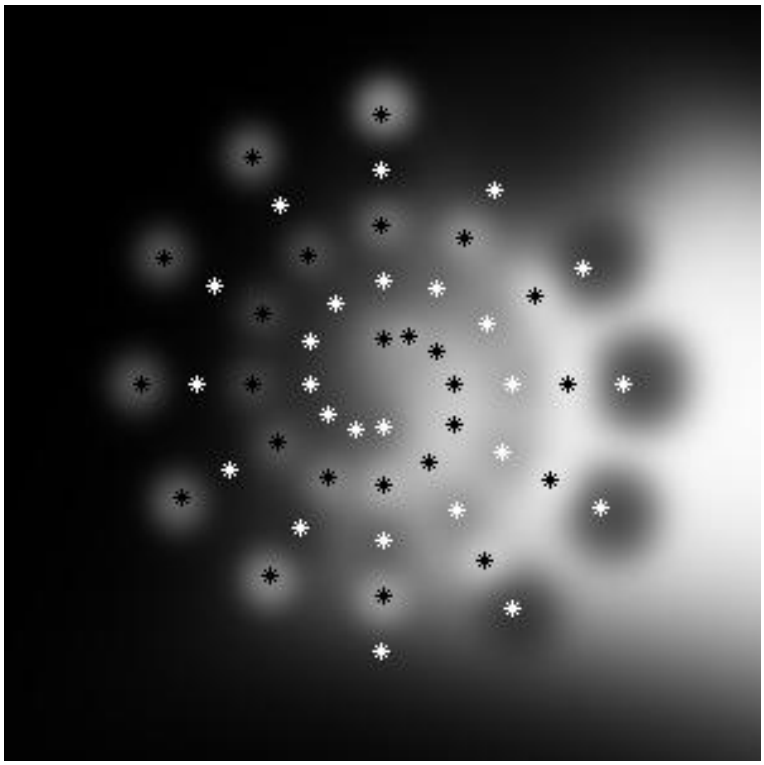

Abbildung 4.10: *Separation zweier Spiralen: Die 50 Sterne markieren das Trainingsset. Das GRNN hat Probleme, die richtige Klassifizierung zu finden.*

Ein interessanter Effekt tritt auf, wenn die Größe des Trainingssets reduziert wird: Ab einer scharfen Grenze von $n \leq 77$ kann das GRNN nicht mehr die optimale

Separation finden. Ein typisches Ergebnis ist in Abbildung 4.10 zu sehen. Während des Trainings kann beobachtet werden, daß bei einzelnen Punkten der Fall von Abbildung 4.8 auf Seite 82 eintritt. Bei diesen Punkten liegt die optimale Kernelbreite bei $\sigma^i = +\infty$. Entsprechend der Vorgehensweise in Kapitel 4.9.6 bekommen die Punkte die minimale Kernelbreite σ^i_{min} zugewiesen, wodurch allerdings optimale Generalisierung nicht garantiert ist. Es fällt auf, daß zuerst die äußersten Punkte der Spiralen Probleme bereiten. Je kleiner n wird, desto mehr Punkte können nicht mehr trainiert werden.

Ein Grund dafür ist, daß es ab der Grenze $n \leq 77$ Punkte gibt, für die der Abstand zum nächsten Punkt der gleichen Klasse größer wird als der Abstand zu Punkten der anderen Klasse. Dies betrifft mit abnehmenden n zuerst die äußersten Punkte. Lösen läßt sich dieses Problem nur mit anderen Kernelfunktionen. Eine der Vereinfachungen (Gleichung (4.13) auf Seite 60) bei der Herleitung der Fundamentalgleichung des GRNN war die Annahme identischer Kernelbreiten für jede der p Dimensionen der unabhängigen Variable $\mathbf{X}$. Hier haben wir ein Beispiel, bei dem unterschiedliche Kernelbreiten Vorteile brächten. Noch besser wären asymmetrische, drehbare Kernelfunktionen; dies zöge allerdings größere Änderungen in der Konstruktion des GRNN nach sich.

4.10.2 Benchmarkset „PROBEN1“

Die Sammlung an Benchmark–Tests wurde von LUTZ PRECHELT [83] zusammengestellt und ist seit 1994 im Internet frei verfügbar. Die einzelnen Datensätze stammen aus verschiedenen Quellen (z.B. dem UCI machine learning database archive [75]) und sind bis ins Detail standardisiert. Ebenso sind präzise Regeln zum Einsatz der Tests angegeben. Allen Problemen sind folgende Eigenschaften gemein:

- Sie sind für überwachtes Lernen konzipiert, da Eingabe ($\mathbf{X}$)- und Ausgabewerte ($\mathbf{Y}$) getrennt angegeben sind.
- Keiner der Datensätze ist künstlich generiert; alle stammen aus der Praxis und decken eine große Klasse an Problemen ab.
- Alle Probleme sind statisch, ändern sich also nicht während des Trainings.
- Die meisten Probleme haben sowohl kontinuierliche als auch binäre Eingabewerte.

- Die Datensets sind bereits in Trainings-, Validierungs- und Testset aufgeteilt. Jeder Datensatz existiert in drei verschiedenen Permutationen, damit die Effekte der Aufteilung berücksichtigt werden können.
- Die Codierung der Probleme ist für Neuronale Netze optimiert. Insbesondere wurden fehlende Daten auf geeignete Weise ersetzt.
- Die 15 Datensets bestehen aus 11 Klassifizierungs- und 4 Approximationsproblemen.

Lutz Prechelt nutzte den Proben1–Benchmark zum Testen von drei verschiedenen, optimierten neuronalen feedforward–Architekturen. Die damit erzielten Ergebnisse sind in [83] angegeben und dienen im Weiteren als Referenz.

Beim Einsatz der Proben1–Benchmarks in dieser Studie wurden exakt die Spezifikationen aus [83] eingehalten. In [83] sind die einzelnen Datensätze genau beschrieben, deshalb werden hier zur Charakterisierung nur die Namen der Datensets angegeben. Die einzigen Unterschiede sind, daß zum einen sowohl Ein- als auch Ausgabewerte auf Mittelwert 0 und Standardabweichung 1 normiert werden und zum anderen durch die holdout–Methode Trainings- und Validierungsset zusammengelegt werden.

Aus Gründen der Übersichtlichkeit sind die vollständigen Ergebnisse aller 72 mit dem GRNN durchgeführten Tests in Anhang A tabellarisch aufgeführt. In den folgenden Diagrammen werden die prinzipiellen Resultate der Tests dargestellt.

Als erstes interessiert die Frage, wie gut das GRNN zusammen mit den hier vorgestellten Trainingsverfahren Probleme lösen kann. Da die absolute Größe der Trainingsfehler stark vom Datensatz selbst abhängt, werden der Klassifizierungs- bzw. Regressionsfehler zur Normierung durch das jeweils beste Ergebnis in [83] dividiert. Ein Wert kleiner als 1 bedeutet, daß das GRNN ein besseres Ergebnis als das beste Netz aus [83] erzielen konnte. Analog zeigt ein Wert größer 1 ein schlechteres Ergebnis des GRNN als die Referenz.

In Abbildung 4.11 sehen wir die Häufigkeitsverteilung der Lösungen jeweils für schnelles (Kapitel 4.9.3) und präzises Training (Kapitel 4.9.4). Der Einfachkeit halber wurde in Abbildung 4.11 bei den Approximationstests der Regressionsfehler auch als „Klassifikationsfehler“ bezeichnet.

Bei beiden Kurven liegt das Maximum nahe bei 1. Dies bedeutet, daß beide Trainingsverfahren am häufigsten Netze liefern, die das jeweils gestellte Problem

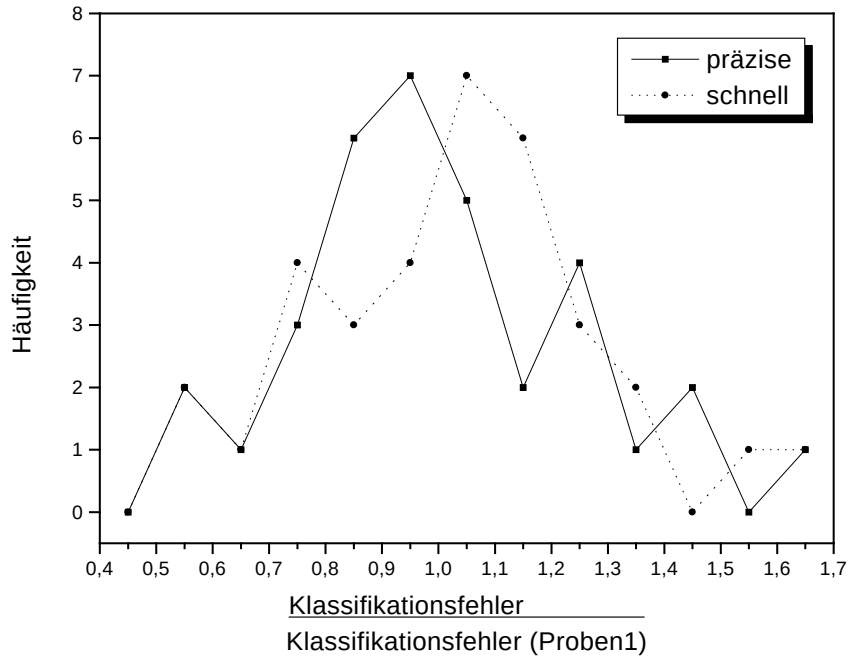

Abbildung 4.11: *Häufigkeitsverteilung der Ergebnisse von präzisem und schnellem Training eines GRNN.*

vergleichbar gut wie die Referenz–Netze in [83] lösen. Auffällig ist, daß die Ergebnisse einen breiten Schwankungsbereich aufweisen. Beide Trainingsverfahren liefern, wenn auch mit geringer Häufigkeit, Netze, deren Fehler nur noch etwa die Hälfte wie auch das 1.5-fache des Referenzfehlers betragen kann.

Erstaunlich ist vor allem, daß der statistische Unterschied zwischen schnellem und präzisem Trainingsverfahren vergleichsweise gering ist, obwohl der Unterschied im Trainingsaufwand gerade bei größeren Datensätzen einige Größenordnungen betragen kann. Wie zu erwarten werden mit präzisem Training tendenziell bessere Lösungen gefunden als mit schnellem Training. Die Häufigkeitsverteilung der präzisen Trainingsergebnisse in Abbildung 4.11 liegt etwas weiter „links“ bei niedrigeren Fehlern. Das Maximum der Lösungshäufigkeit liegt für präzises Training bei 0.95; für das schnelle Training bei 1.05.

Lutz Prechelt teilte jeden Datensatz jeweils dreimal in Trainings-, Validierungs- und Testset auf. Es existieren damit die Sets {Trainingsset1, Validierungsset1, Testset1}, {Trainingsset2, Validierungsset2, Testset2} und {Trainingsset3, Validierungsset3, Testset3}. In Abbildung 4.12 wird der Einfluß der unterschiedlichen Aufteilungen untersucht. Zur besseren Vergleichbarkeit werden die Fehler von Testset2 und Testset3 durch den Fehler von Testset1 dividiert.

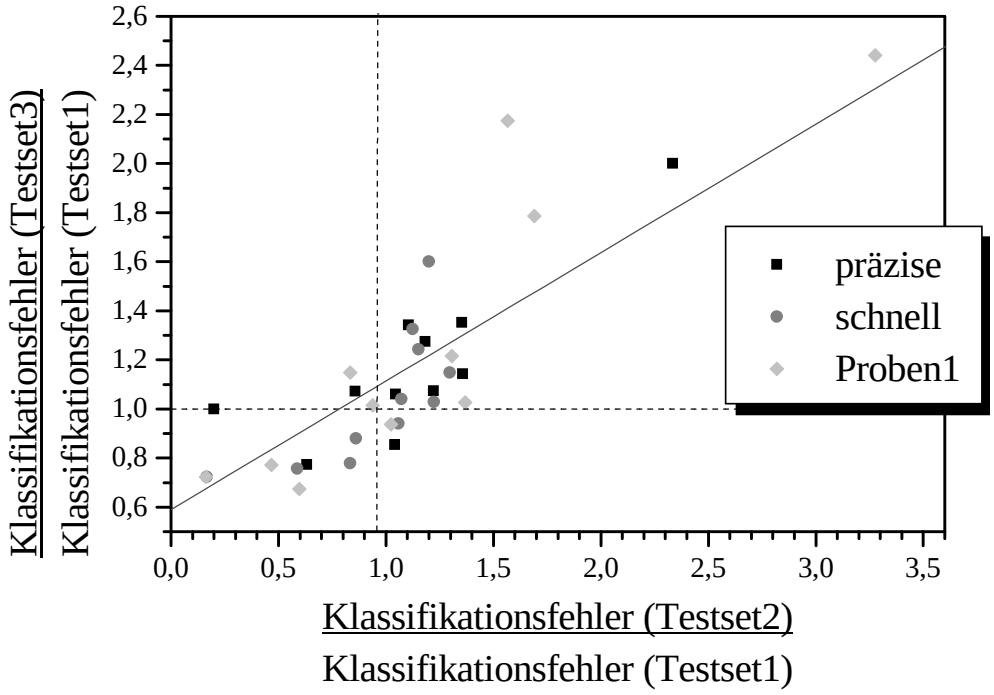

Abbildung 4.12: *Einfluß der Aufteilung in Trainings-, Validierungs- und Testset. Auffällig ist die hohe Korrelation des Fehlers zwischen Testset2 und Testset3.*

Im Idealfall repräsentiert jedes Set die statistischen Eigenschaften des vollständigen Datenraums. Nur unter dieser Voraussetzung führt die Minimierung des Trainingsfehlers E_V zur optimalen Generalisierung des Testsets. Insbesondere sollte der Fehler des jeweiligen Testsets unabhängig von der Aufteilung sein. Dies entspräche in Abbildung 4.12 dem Punkt (1,1). Statistische Fluktuationen in der Zusammensetzung der Sets sollten zu zufälligen Abweichungen vom Punkt (1,1) führen. Zu erwarten ist also eine kreisförmige „Punktwolke“ mit dem Zentrum (1,1). Tatsächlich zeigt der lineare Fit in Abbildung 4.12, daß eine positive Korrelation zwischen dem Fehler von Testset2 und Testset3 besteht, mit zum Teil großen Abweichungen vom Punkt (1,1). Das bedeutet, daß die Fehler von Testset2 und Testset3 häufig ähnlich groß sind, sich aber deutlich vom Fehler von Testset1 unterscheiden. Nach LUTZ PRECHELT ist bei einigen Datensets (z.B. `building`) das Set1 in der originalen Datenabfolge angeordnet, die beiden anderen Sets sind zufällig permutiert worden. Offenbar betrifft diese Art der Anordnung auch noch weitere Datensätze. Abbildung 4.12 beweist auf jeden Fall, daß bei den PROBEN1–Datensätzen Testset1 häufig einen anderen Bereich des Datenraums als Testset2 und Testset3 abdeckt, da der Zusammenhang zwischen Testset2 und Testset3 nicht vom Netztyp oder Trainingsverfahren abhängt.

Diese Untersuchung zeigt, wie stark der Einfluß der Datenaufteilung in Trainings- und Testdaten sein kann. Insbesondere bei kleinen Datensätzen werden die dadurch

induzierten Fehler immer größer. Angenommen, es existiere ein statistisches Verfahren, daß aus den Trainingsdaten die optimale Regressionsfunktion ermitteln kann. Es ist dann möglich, daß dieses (optimale) statistische Verfahren, auf Testdaten angewandt, *schlechtere* Ergebnisse als ein anderes, schlechteres statistisches Verfahren liefert! Die Leistungsfähigkeit verschiedener statistischer Verfahren läßt sich also nur dann objektiv vergleichen, wenn Trainings- und Testdaten sorgfältig und reproduzierbar ausgewählt sind. Dieser Effekt wird in den meisten Publikationen über Trainingsverfahren neuronaler Netze ignoriert. Andererseits kann er bedeuten, daß durch gezieltes *Verschlechtern* der Generalisierleistung des statistischen Verfahrens das Testset *besser* vorhersagbar wird.

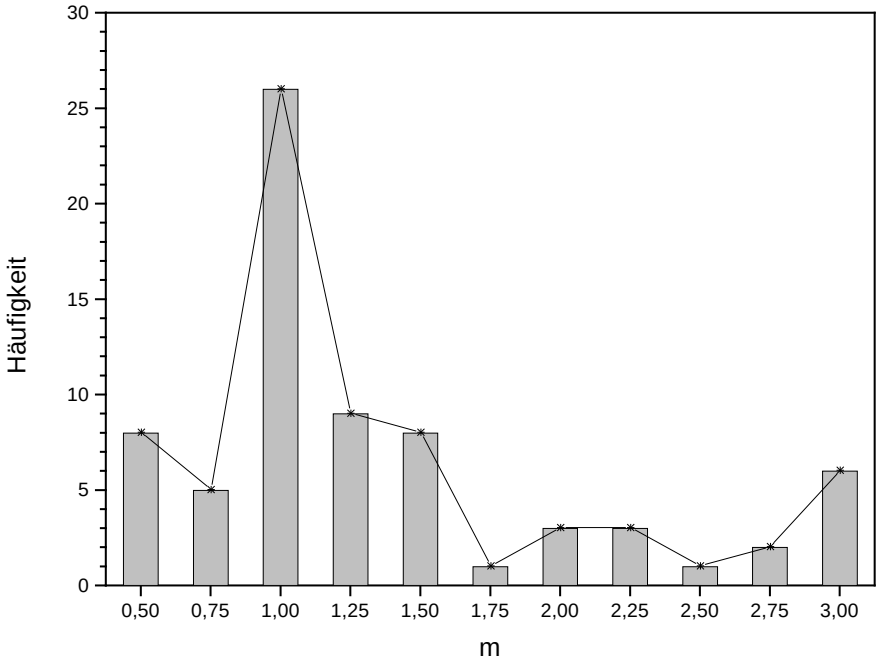

Abbildung 4.13: *Häufigkeitsverteilung der minimalen Fehler der Testsets in Abhängigkeit vom optimalen Sigmafaktor* m_{opt}.

Eine einfache Möglichkeit, die Generalisierleistung eines bereits trainierten GRNN zu verschlechtern[3], ist die Multiplikation aller Kernelbreiten mit einem Sigmafaktor $m \neq 1$.

[3]Zur Erinnerung:
Der letzte Schritt in beiden Trainingsverfahren ist das Multiplizieren der Kernelbreiten mit unterschiedlichen Faktoren m (siehe Kapitel 4.9.3.1), solange, bis der Trainingsfehler E_V minimal ist. Demzufolge erhöht jedes weitere Multiplizieren der Kernelbreiten mit einem Sigmafaktor $m \neq 1$ zwangsläufig wieder den Trainingsfehler E_V.

In Abbildung 4.13 wurde untersucht, wie sich der Fehler der Testsets bei unterschiedlichen Sigmafaktoren verändert. Zu diesem Zweck wurden *nach* dem Training die Kernelbreiten mit verschiedenen Sigmafaktoren zwischen 0.5 und 3 multipliziert und der Fehler des Testsets bestimmt. Der optimale Sigmafaktor m_{opt} ist derjenige Faktor, für den der Fehler des Testsets kleiner oder gleich dem Fehler des Testsets des ursprünglichen GRNN ist. Im Anhang A ist m_{opt} in Tabelle A.4 tabellarisch aufgeführt.

In Abbildung 4.13 sehen wir die Häufigkeitsverteilung der minimalen Fehler der Testsets in Abhängigkeit vom optimalen Sigmafaktor m_{opt}. Das Maximum bei $m_{opt} = 1$ bedeutet, daß mit dem unveränderten, trainierten GRNN und den PROBEN1–Datensätzen in $\frac{26}{72} \approx 36.1\%$ aller Fälle die Fehler der Testsets minimal waren. Andererseits heißt das auch, daß in 63.9% aller Beispiele die Minimierung von E_V

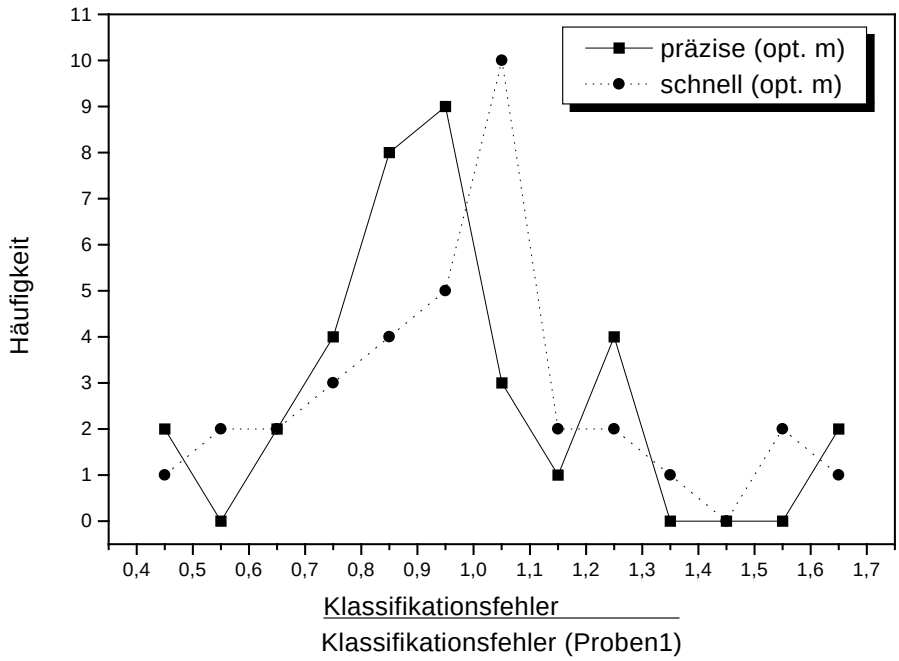

Abbildung 4.14: *Vergleich von präzisem und schnellen Training mit optimalem Sigmafaktor m_{opt}.*

nicht zur Minimierung des Fehlers des Testsets führte. In einigen Fällen war ein erhebliches Verstellen der Kernelbreiten ($2 \leq m_{opt} \leq 3$) mit gleichzeitiger Verschlechterung des Trainingsergebnisses nötig. Entscheidend ist, daß dort eine weitere Verbesserung des Trainingsalgorithmus keine Vorteile mehr brächte, da der Trainingsfehler E_V weiter minimiert würde, der Fehler der Testsets aufgrund von Asymmetrien zwischen Trainings- und Testset aber bei größeren Werten von E_V minimal ist. Bei solchen Datensätzen eignet sich demnach das Kriterium des *minimalen*

Testfehlers nicht zum Vergleich der Güte verschiedener statistischer Verfahren!

In Abbildung 4.14 sehen wir die Häufigkeitsverteilung der Lösungen für den optimalen Sigmafaktor m_{opt} jeweils nach schnellem (Kapitel 4.9.3) und präzisem Training (Kapitel 4.9.4). Im Vergleich mit Abbildung 4.11 ist zu erkennen, daß beide Kurven noch etwas weiter zu besseren Lösungen hin verschoben sind. Es existieren aber immer noch Beispiele, für die die besten Netze aus [83] bessere Lösungen bieten. Aufgrund der obigen Beobachtungen wäre es allerdings möglich, daß diese Netze nur aufgrund eines zufälligen Trends besser als die GRNN sind, da die Datensätze selbst einen Trend enthalten.

Während des Trainings wird kein Unterschied zwischen einem Regressions- und einem Klassifikationsproblem gemacht. Zum Klassifizieren wird zunächst die optimale Regression ermittelt, und anschließend die Klassenaufteilung vollzogen. Es stellt sich die Frage, ob bessere Regression tatsächlich immer zu besserer Klassifikation führt.

Zur Beantwortung wurden die Daten, die auf der Suche nach dem optimalen Sigmafaktor m_{opt} anfielen, auf eine andere Weise aufgetragen. Zur Normierung wurden für jedes m der Regressionfehler des Testsets durch den Regressionsfehler des Testsets des ursprünglich, trainierten GRNN (d.h. $m = 1$) dividiert. Desgleichen für die Klassifikationsfehler.

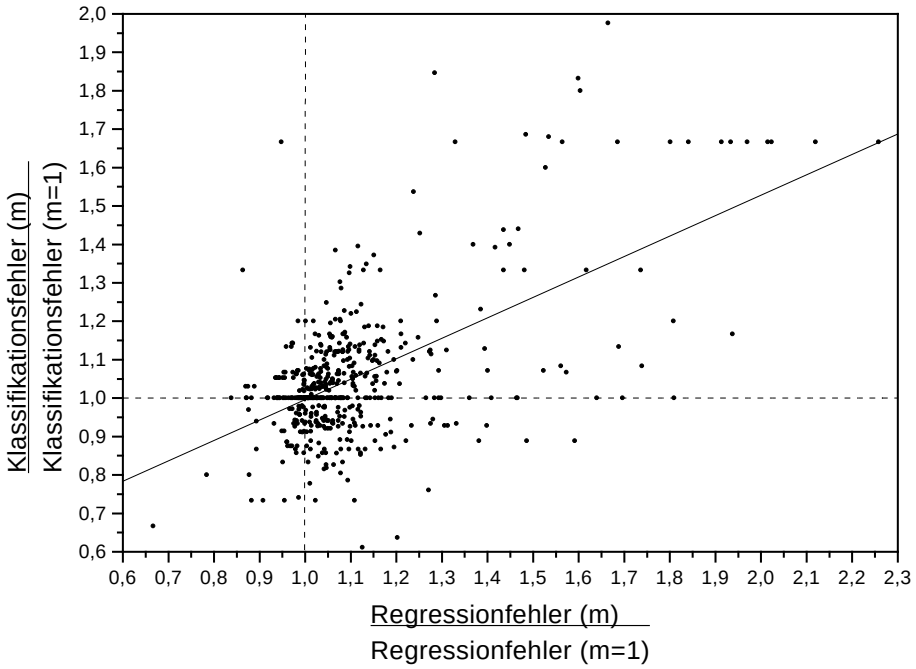

Abbildung 4.15: *Vergleich des Klassifikationsfehlers mit dem Regressionsfehler*

In Abbildung 4.15 wurden für alle durchgetesteten Sigmafaktoren m der normierte Klassifikationsfehler über dem (normierten) Regressionsfehler aufgetragen. Die zwei

gestrichelten Linien unterteilen das Diagramm in vier Quadranten mit dem Mittelpunkt (1,1), da jedes GRNN direkt nach dem Training den Punkt (1,1) belegt. Jeder Punkt im Diagramm, der *nicht* auf dem Mittelpunkt liegt, hat demzufolge einen größeren Trainingsfehler E_V als das zugehörige GRNN von Punkt (1,1). Alle Punkte mit einem normierten Regressionsfehler größer als 1 — auf der „rechten" Seite der vertikalen Linie — entsprechen Sigmafaktoren, für die die entsprechenden GRNN einen größeren Regressionsfehler des Testsets liefern. Analoges gilt für Punkte mit einem normierten Klassifikationsfehler größer als 1 — „oberhalb" der horizontalen Linie. Die durchgezogene Linie ist der lineare Fit durch alle Punkte und zeigt, wie zu erwarten, durch seine positive Steigung eine positive Korrelation zwischen Regressionsfehler und Klassifikationsfehler.

Interessant sind die Fälle in den Quadranten links/oben und rechts/unten, da sie eine negative Korrelation zeigen. Es existieren also Datensätze, bei denen ein *geringerer* Regressionsfehler des Testsets zu einem *höheren* Klassifikationsfehler führt und umgedreht. Ein Grund mag sein, daß es bei Funktionen, deren Funktionswerte nur aus 0 und 1 bestehen, viele gleichberechtigte Approximationsfunktionen geben kann, die bei stark diversen Datensets zu Schwankungen in der Klassifizierleistung führen. Ein weiterer Grund ist, daß der Regressionsfehler analog zum Trainingsfehler E_V (Gleichung (4.22)) als die mittlere *quadratische* Abweichung zwischen Ist- und Sollwert definiert ist. Der Klassifizierungsfehler ist aber der *absolute* Prozentsatz der fehlerhaft klassifizierten Daten.

Der Autor ist der Meinung, daß selbst mit einem „perfekten" statistischen Verfahren aus den PROBEN1–Datensätzen nicht wesentlich mehr Informationen extrahiert werden können, weil die Daten aus der Praxis stammen, teilweise unvollständig sind und den Datenraum ungleichmäßig abdecken. Damit erzeugt die Aufteilung in Trainings-, Validierungs- und Testsets weitere Fehler. Falls ein Verfahren deutlich bessere Ergebnisse für ein Testset liefert, ist davon auszugehen, daß es die zufälligen Abweichung des Testsets vom Trainingsset zufällig ebenfalls enthält, daher eigentlich sogar schlechter ist.

4.11 Zusammenfassung

In diesem Kapitel wurde das General Regression Neural Network als leistungsfähiges Approximations- und Klassifizierungswerkzeug vorgestellt. Mit Hilfe der beiden Trainingsverfahren war das GRNN in der Lage, für nahezu alle gestellten Aufgaben

brauchbare Lösungen zu finden. In vielen Fällen konnten sogar bessere Ergebnisse als mit hoch optimierten Backpropagation–Netzen erzielt werden.

Spielt Rechenzeit keine Rolle und ist die Güte des Netzes die entscheidende Anforderung, so ist das präzise Training nach Kapitel 4.9.4 die Methode der Wahl. Die Testergebnisse in Kapitel 4.10.2 und Anhang A zeigen, daß bereits mit dem schnellen Trainingsverfahren nach Kapitel 4.9.3 Netze generiert werden können, die einen Vergleich mit optimierten Backpropagation–Netzen nicht scheuen müssen.

Die Vorteile des General Regression Neural Network im Einzelnen:

- Die Architektur des Netzes und alle Gewichtsfaktoren sind eindeutig determiniert. Lediglich die n Kernelbreiten σ^i müssen festgelegt werden.

- Beide Trainingsverfahren benötigen einige benutzerdefinierte Parameter: Anzahl Nachkommastellen des Trainingsfehlers E_V, Anzahl Stützstellen für „schnelles Training II“ (Kapitel 4.9.3.2), Anzahl a äquidistanter Abschnitte für präzises Training (Kapitel 4.9.4.1), mittlere Anzahl Schritte $\bar{a}$ (Gleichung (4.39))

  Das Trainingsergebnis erwies sich als unempfindlich gegenüber Variationen der Werte dieser Parameter. Die konkrete Wahl der Parameter hat vor allem Einfluß auf den Rechenaufwand beim Training und nicht auf das Ergebnis selbst.

- Da zu keinem Zeitpunkt des Trainings (Pseudo-)Zufallszahlen eingesetzt werden, ist das Trainingsergebnis eindeutig und reproduzierbar.

- Die Willkür der fehlerinduzierenden Aufteilung in Trainings- und Validierungsset ist überflüssig, da das Netz die holdout–Methode erlaubt. Im Gegensatz zu feedforward–Netzen erlaubt das GRNN beliebig viele Trainings-/Überprüfungssetpaare, wodurch sich die statistische Genauigkeit des Netzes erhöht.

- Das GRNN setzt *nicht* zwingend voraus, daß die Sequenzen mit denen trainiert wird, konstante Länge haben müssen, da die Bewertung durch die Metrik zwischen zwei Sequenzen erfolgt. Mit einer geeignet definierten Metrik kann das Netz mit Sequenzen unterschiedlichen Typs und Länge trainiert werden, ohne daß gleicher Typ erzwungen werden muß. Diesen Vorteil bietet kein anderes bekanntes Netz. Wichtig: Bei einer anderen als der EUCLID'schen Metrik muß

in Kapitel 4.9.2 Gleichung (4.31) modifiziert werden. Damit verändern sich die minimalen und maximalen Werte σ^i_{min} und σ^i_{max} der Kernelbreiten.

Selbst der Begriff „Sequenz“ kann im Zusammenhang mit dem GRNN freier verstanden werden: Das GRNN kann problemlos mit Daten trainiert werden, die sich nur schwer als lineare Sequenz codieren lassen. Ein Beispiel wären Daten, die als Bäume organisiert sind, wie Moleküle.

- Für moderate Größen des Trainingssets (einige hundert Punkte) ist das Training des GRNN deutlich schneller als andere Netztypen.

Aufgrund dieser Vorteile ist der Autor der Meinung, daß das GRNN in Kombination mit den hier entwickelten Trainingsmethoden herkömmlichen neuronalen (feedforward–) Netzen deutlich überlegen ist.

4.12 Ausblick

Trotz der Leistungsfähigkeit des GRNN sind noch einige Erweiterungen denkbar:

- Die Trainingsverfahren haben gewisse Schwierigkeiten, falls die Trainingspunkte sehr ungleichmäßig verteilt sind. Durch Kombination beispielsweise mit Clusteralgorithmen müßte dieses Problem zu überwinden sein.
- Durch den Einsatz des modifizierten präzisen Trainingsverfahrens nach Kapitel 4.9.4.1 sollte das Trainings schneller werden und gleichzeitig ein weitgehend redundanzfreies Trainingsset zu erhalten sein. Außerdem sind diese Daten wahrscheinlich gleichförmiger im Suchraum verteilt.
- Das Beispiel der Separation zweier Spiralen (Kapitel 4.10.1) zeigt die Notwendigkeit der Erweiterung um asymmetrische, drehbare Kernelfunktionen. Das Problem wäre dabei allerdings, daß pro Datenpunkt noch einmal p zu trainierende Freiheitsgrade hinzukämen.
- Ein trainiertes GRNN sollte wesentlich besser zu analysieren sein, als ein feedforward–Netz. Es müßte ein Algorithmus zu entwickeln sein, der in der Lage ist, die Frage zu beantworten, „was das Netz eigentlich genau gelernt habe“.

Die skizzierten Erweiterungen zeigen, daß das GRNN ein Werkzeug zur Datenanalyse und nicht nur ein gutes Approximationsverfahren sein kann.

Kapitel 5

Der TST–Algorithmus

Der sog. **TST**–Algorithmus (nach den Erfindern Dirk **T**omandl, Andreas **S**chober, Marcel **T**hürk) basiert auf einer Idee von Marcel Thürk und Andreas Schober von 1993 [95].

Herkömmliche Optimierverfahren setzen meist eine explizite, analytische Fitnessfunktion voraus. Im Gegensatz dazu versucht der hier vorgestellte TST–Algorithmus auf Fitnesslandschaften, deren Fitnessfunktion unbekannt ist bzw. deren Fitnesswerte nur mit großem Aufwand zu ermitteln sind, zu optimieren. Das Ziel des Algorithmus ist, mit einer möglichst geringen Anzahl an Stichproben sowie Iterationen möglichst „gute“ (evtl. lokale) Optima zu finden.

5.1 Einleitung

Die Grundidee des in dieser Arbeit (weiter-)entwickelten iterativen Verfahrens ist, mit Hilfe der *statistischen Eigenschaften der bereits abgetasteten Sequenzen* neue, möglichst erfolgversprechende Sequenzen für die nächste Iterationsrunde vorzuschlagen, deren Fitness dann z.B. experimentell bestimmt wird. Der Einsatz von statistischen Verfahren soll für neu generierte Sequenzen die Wahrscheinlichkeit hoher Fitness erhöhen. Je genauer das statistische Verfahren die lokalen und globalen statistischen Eigenschaften der Fitnesslandschaft aus den bisherigen Sequenzen extrahieren kann, desto besser werden die Sequenzen der nächsten Iterationsrunde sein.

Der TST–Algorithmus verläuft nach dem Schema in Kapitel 2.3.9 auf Seite 21.

Das Schema sagt zunächst nichts über die Eigenschaften des statistischen Verfahrens aus. Insbesondere gibt es per se keine Einschränkungen in der Nutzung von problemspezifischem Zusatzwissen. Da der Algorithmus universell einsetzbar sein soll,

wurde hier zunächst auf den Einsatz von Zusatzwissen verzichtet. Außerdem können gerade theoretische Zusatzinformationen den Optimiererfolg beeinträchtigen, wenn sie auf ungenauen Modellvorstellungen beruhen. Das statistische Verfahren soll deshalb hier ausschließlich die Information *Sequenz* → *Fitness* zur Verfügung haben — näheres dazu in Kapitel 5.3.1. Prinzipiell kann jedoch jede Zusatzinformation verwendet werden.

Die neuen Sequenzen des Auswahl–Moduls werden zur bestehenden Population hinzugefügt und in der nächsten Iterationsrunde bewertet, d.h. ihre Fitness bestimmt. Die Populationsgröße bleibt damit nicht konstant, sondern wächst in gleichmäßigen Schritten von Generation zu Generation.

5.2 Der ursprüngliche Algorithmus

In der ursprünglichen, von MARCEL THÜRK programmierten Variante [120] übernahm ein *künstliches neuronales feedforward–Netz* („Backpropagation“) die Rolle des statistischen Verfahrens.

Der Autor dieser Arbeit setzte für die ersten Testversionen ebenfalls ein feedforward–Netz ein, das mit einer etwas modifizierten Backpropagation–Lernregel trainiert wurde. Das Backpropagation–Netz wurde hier solange mit den bereits bekannten Sequenz-Fitness–Paaren trainiert, bis ein bestimmter, vorgegebener Trainingsfehler erreicht war. Obwohl in vielen praktischen Anwendungen neuronaler Netze auf diese Weise verfahren wird, ist dies sehr problematisch, da damit nie garantiert werden kann, daß die statistischen Eigenschaften der zugrundeliegenden Fitnesslandschaft „optimal“ extrahiert werden. Zuverlässigere Kriterien liefern „Crossvalidation“–Methoden (siehe Kapitel 4.9.1).

Die Auswahl der neuen Sequenzen erfolgte durch einen *genetischer Algorithmus*, der in der durch das Netz approximierten Fitnesslandschaft nach Sequenzen suchte, für die das Netz die höchsten Werte liefert.

Trotz des Prototypen-Charakters des Verfahrens zeigte sich schon recht bald dessen potentielle Leistungsfähigkeit an Beispielen stark frustrierter Landschaften (Kapitel 2.4), wie Spinglas- und RNS-Sekundärstrukturlandschaften. In der Regel konnten (sogar globale) Optima in weniger als 10-20 Iterationen und mit insgesamt wenigen hundert Testsequenzen gefunden werden.

Unterschiede zum üblichen dynamischen Verhalten von genetischen Algorithmen bestanden darin, daß bei letzteren vor allem die mittlere Fitness einer Population

optimiert wird, während bei dem beschriebenen Verfahren die lokalen Optima besonders selektiert und im Vergleich zu genetischen Algorithmen „schneller“ gefunden wurden.

Diese ursprüngliche Variante des Algorithmus hatte vor allem folgende Nachteile:

1. Das neuronale Netz:
   - Ein Standard-Backpropagation–Netz setzt eine Reihe von Parametern voraus, die vom Benutzer vorzugeben sind. Der Trainingserfolg hängt meist sehr empfindlich von der „richtigen“ Wahl der Parameter ab. Bis dato existieren trotz intensiver Bemühungen der Neuronale-Netze–Gemeinde keine effizienten Trainingsalgorithmen, die völlig selbständig Architektur und Lernparameter bestimmen.
   - Das Training der Gewichte der Verbindungen kann leicht in lokalen Optima steckenbleiben, da der Backpropagation–Algorithmus wie alle seine Klone Variationen des Gradientenverfahrens sind. Dies gilt auch für die meisten anderen bekannten Trainingsmethoden für feedforward–Netze.
   - Es ist schwierig zu analysieren, was ein Backpropagation–Netz beim sog. „verallgemeinerten Lernen“ tatsächlich lernt.
   - Alle bekannten Trainingsverfahren benötigen viel Rechenzeit.
2. Das System kann nicht mit Sequenzen unterschiedlicher Längen arbeiten.
3. Sequenzen mit verschobenem Leseraster interpretiert das Netz als vollkommen neue Sequenzen.
4. Da der Algorithmus nur die Sequenzen mit den *höchsten* durch das Netz approximierten Fitnesswerten übernimmt, konvergiert er lediglich zum nächstgelegenen lokalen Optimum. Nur wenn eine Sequenz der Startpopulation bereits relativ nahe beim globalen Optimum liegt, besteht die Chance, daß es durch den Algorithmus gefunden werden kann.

Nach dieser Vorstudie wurde der eigentliche TST–Algorithmus entwickelt, der in den folgenden Kapiteln beschrieben werden soll.

5.3 Der TST–Algorithmus

Im Rahmen dieser Arbeit sollte nicht nur der Prinzipiennachweis des Algorithmus geführt, sondern eine im praktischen Alltag einsetzbare Version entwickelt werden.

Deshalb standen bei der Konzeption des entgültigen TST–Algorithmus zwei Forderungen im Vordergrund:

1. Weitestgehende Vermeidung der Nachteile der Prototypen aus Kapitel 5.2.
2. Minimale Anzahl an benutzerdefinierten Parametern. Falls bestimmte Parameter nicht zu vermeiden sind, sollte zumindest ein sinnvoller Wertebereich anzugeben sein. Auf jeden Fall soll der Algorithmus robust und vorhersagbar auf kleine Parameteränderungen reagieren.

5.3.1 Statistisches Modul

Das statistische Modul innerhalb des TST–Algorithmus bekommt als Eingabe die komplette Population an Sequenz-Fitness–Paaren. Seine Aufgabe ist die Analyse der Abbildung $Sequenz \rightarrow Fitness$. Insbesondere interessiert die Frage, welche Eigenschaften der Sequenzen für eine hohe Fitness verantwortlich sind und in welchen Bereichen des Suchraums die Optima angesiedelt sind. Des weiteren muß das statistische Verfahren in der Lage sein, auf der Basis seiner Analyse für unbekannte Sequenzen einen Fitnesswert abschätzen zu können.

Welche statistischen Verfahren eignen sich für den Einsatz im TST–Algorithmus? Grundsätzlich alle, die die eingangs genannten Eigenschaften erfüllen. Prinzipiell lassen sich zwei Vorgehensweisen unterscheiden:

1. Analyse der *Struktur* der Sequenzen:
   Hierbei wird versucht, gemeinsame statistische Eigenschaften der Sequenzen mit den höchsten Fitnesswerten zu analysieren, um mit dem Auswahl–Modul (Kapitel 5.3.2) neue Sequenzen mit denselben Charakteristika zu generieren.

   - Sequenz als Zeitreihe auffassen: Ermitteln der Bildungsregeln der Zeitreihe durch numerische und/oder analytische Methoden. Hilfsmittel können hierbei Classifier–Systeme, Fourier-Zerlegung, Autokorrelationsfunktion oder die Wavelet-Transformation sein. Die Wavelet-Transformation liefert im Gegensatz zur Fourier-Zerlegung ein zeitabhängiges Frequenzspektrum einer Zeitreihe. Damit können fraktale Eigenschaften von Sequenzen detektiert werden. Auch neuronale Netze (z.B. feedforward–Netze mit/ohne Rückkopplungen, ELMAN–Netze, TDNNs) können zur Zeitreihenanalyse eingesetzt werden. Verlustbehaftete Kompressionsverfahren

bieten zusätzlich Vorteile beim Eliminieren von „Rauschen“ innerhalb einer Zeitreihe.

- Sequenzanalyse, z.B. durch Homologievergleiche zwischen besten Sequenzen oder durch Methoden der Mustererkennung.
- Modellbehaftete Verfahren, die möglichst viel zusätzliche Information über die Daten besitzen und einsetzen. Beispielsweise fällt die Vielzahl an Modelling-Programmen zum rationalen Design von Molekülen in diese Kategorie.

2. Analyse der *Verteilung* der Sequenzen im Suchraum:
   Das statistische Modul soll eine Vorstellung davon bekommen, in welchen Bereichen des Suchraums sich Sequenzen mit hoher Fitness befinden.

   - Clusteranalyse der Sequenzen (z.B. durch KOHONEN–Netze nach Anhang D.2.5).
   - Hauptkomponentenanalyse der Sequenzen.
   - Multidimensionale Regression: Falls kein Modell zur Verteilung der Sequenzen zur Verfügung steht, muß mit modellfreier Regression gearbeitet werden. In diesem Zusammenhang haben sich künstliche neuronale feedforward–Netze bewährt.

Wünschenswert wäre es, daß das statistische Verfahren zusätzlich Daten analysieren kann, die nicht in Sequenzform vorliegen, wie beispielsweise organische Moleküle. Vorteilhaft wäre auch die Möglichkeit, die Präzision des Verfahrens durch einen Steuerparameter einstellen zu können.

In dieser Arbeit wurden aus der Fülle an möglichen statistischen Verfahren nur Regressionsverfahren getestet. Der Prototyp in Kapitel 5.2 setzte ein Backpropagation–Netz als statistisches Verfahren mit den geschilderten Nachteilen ein. Intensive Literaturrecherche zeigte, daß die grundsätzlichen Probleme neuronaler feedforward–Netze noch nicht überzeugend gelöst sind. Gerade die automatische Wahl der Netzwerk–Architektur benötigt meist einen enormen Rechenaufwand, der auch auf schnellen Workstations Tage betragen kann und trotzdem häufig unbefriedigende Ergebnisse liefert.

In Kapitel 4 wurde das „General Regression Neural Network“ (GRNN) von DONALD F. SPECHT als modellfreies Regressionsverfahren vorgestellt. Es wurde bis

jetzt in relativ wenigen Arbeiten eingesetzt, da es an schnellen, effizienten Trainingsalgorithmen mangelte. Das GRNN zusammen mit den hier entwickelten Trainingsverfahren weist keinen der genannten Nachteile eines Backpropagation–Netzes auf — siehe Kapitel 4.11. Im Kern berechnet das GRNN einen gewichteten Mittelwert aller Sequenzen und benötigt dazu eine Metrik. Da die Metrik frei wählbar ist, können bei geeigneter Definition auch Sequenzen unterschiedlicher Längen oder mit verschobenem Leseraster verglichen werden. Insbesondere bietet dies die Möglichkeit, Daten zu analysieren, die nicht als Sequenzen darstellbar sind. Die Genauigkeit der Approximation der Daten läßt sich durch die Multiplikation der Kernelbreiten σ^i mit einem Faktor $m \neq 1$ steuern (siehe Kapitel 4.9.3.1). Falls $m > 1$ ist, wird die Approximationfläche „glatter“ („underfitting“) und erfaßt stärker die globalen Strukturen der Fitnesslandschaft. Eine stärkere Betonung lokaler Eigenschaften wird durch $m < 1$ erreicht („overfitting“).

Das GRNN erfüllt also alle Anforderungen, die hier an das statistische Modul gestellt werden.

5.3.2 Auswahl–Modul

Der Benutzer übergibt dem Modul neben einem Zeiger auf das statistische Modul die Anzahl neuer Sequenzen als Parameter s. Das Auswahl–Modul hat nun die Aufgabe, s neue Sequenzen zu generieren, die mit einer gewissen Wahrscheinlichkeit eine hohe Fitness besitzen. Es nutzt dabei das statistische Modul aus Kapitel 5.3.1: Das statistische Modul muß lediglich in der Lage sein, auf der Basis seiner Analyse für beliebige Sequenzen des Suchraums einen Schätzwert für die Fitness angeben zu können. Eine Einstellung der Präzision der Schätzung ist nützlich, aber nicht unbedingt notwendig.

Kombinatorische Suche: Die einfachste Version des Auswahl–Moduls wäre es, jede kombinatorisch mögliche Sequenz des Suchraums zu generieren, vom statistischen Modul bewerten zu lassen und die s Sequenzen mit den höchsten geschätzten Fitnesswerten zu übernehmen. Für die meisten Probleme wird dieser Ansatz aufgrund der „kombinatorischen Explosion“ (siehe z.B. Anhang C.2) zu einem enormen Rechenaufwand führen. Diese Vorgehensweise ist dann sinnvoll, wenn die neuen Sequenzen aus einem vorgegebenen Pool (z.B. einer Bibliothek an organischen Molekülen) entnommen werden sollen und keine völlig neuen Sequenzen gesucht sind.

Genetischer Algorithmus: Die Suche nach hochbewerteten Sequenzen läßt sich als Optimierproblem auffassen und effizient mit Optimieralgorithmen, beispiels-

weise nach Kapitel 2.3 lösen. Hier wurde ein genetischer Algorithmus (Kapitel 2.3.2.1) gewählt. Um gleichzeitig lokale und globale Eigenschaften der durch das statistische Modul approximierten Landschaft erfassen zu können, wird der genetische Algorithmus mehrmals für unterschiedliche Präzisionen des statistischen Moduls gestartet.

Lokale Optimieralgorithmen: Einer der Nachteile des Prototypen des Autors (Kapitel 5.2) war, daß nur das globale Optimum in der geschätzten Fitnesslandschaft gesucht wurde. Um die dadurch bedingte vorzeitige Konvergenz in lokale Optima der eigentlichen Fitnesslandschaft zu erschweren, werden neben genetischen Algorithmen zusätzlich lokale Optimierverfahren eingesetzt, die von zufällig ausgewählten Startsequenzen aus starten. Auf diese Weise sollen nicht nur das globale, sondern auch lokale Optima der geschätzten Fitnesslandschaft lokalisiert werden.

Caching–Algorithmus: Alle hier eingesetzten Suchverfahren puffern die Sequenzen, auf die sie während ihrer Suche treffen in einem Caching–Speicher zwischen. Normalerweise wird ein solcher Mechanismus aus Performance–Gründen eingesetzt, wenn z.B. die Bewertung der Sequenzen durch das statistische Modul sehr rechenintensiv ist. Auch hier bewirkt der Caching–Algorithmus eine höhere Rechengeschwindigkeit; der eigentliche Zweck ist aber, daß nicht nur die (lokalen) Optima selbst registriert werden, sondern auch Stichproben aus der lokalen Umgebung der Optima mitprotokolliert werden. Erreicht wird dies dadurch, daß die Sequenzen im Cache nicht, wie üblich, nach Häufigkeit ihres Zugriffs, sondern nach Fitness sortiert werden. Falls der Cache während der Optimierung keinen Platz mehr für zusätzliche Sequenzen bietet, werden zuerst die schlechtesten Sequenzen gelöscht und durch die neuen Sequenzen ersetzt.

Rekombination und Mutation: Der Hauptunterschied zwischen genetischen Algorithmen/Evolutionsstrategien und anderen Optimierverfahren ist die Verwendung von genetischen Operatoren, die aus mehreren Vorfahren durch Rekombination mehrere Nachkommen kreieren. Die Hoffnung hierbei ist, daß sich durch die Rekombination Schemata mit einer Fitness größer als die mittlere Fitness der Population herausbilden („Schema–Theorem“ — Gleichung (2.1) auf Seite 12). Damit der TST–Algorithmus auch die Informationen eventueller vorhandener Schemata nutzen kann, werden die besten Sequenzen miteinander proportional zu ihrer Fitness rekombiniert und mutiert. Zu diesem Zweck wird ein Pool aus den bereits bewerteten Sequenz-Fitness–Paaren und den besten Sequenzen des statistischen Moduls gebildet. Letztere bekommen als Fitness die maximale Fitness der bereits bewerteten Sequenz-Fitness–Paare zugewiesen. Neben der Ausnutzung eventueller Schemata fließt auf diese Weise in die Auswahl neuer Sequenzen etwas Zufall mit ein. Der TST–

Algorithmus bekommt dadurch Informationen über Bereiche des Suchraum, die er durch reines Übernehmen der besten Sequenzen des statistischen Moduls nicht bekommen hätte.
Außerdem verhält sich der TST–Algorithmus im ungünstigsten Fall (keine erfolgreiche statistische Analyse möglich) wie ein genetischer Algorithmus.

Schema des Auswahl–Moduls

Fassen wir das bisher Gesagte zusammen, so ergibt sich folgender Algorithmus für das Auswahl–Modul:

Lege Sammel-Cache mit der Größe s^2 an;
Lege Such-Cache mit der Größe $2s$ an;
Setze Präzisions-Parameter des statistischen Moduls auf minimalen Wert;
for $(i = 1; i \leq s - n_{recombination}; i = i + 1)$
    if $i = n_{precision} + 1$
        then Setze Präzisions-Parameter des statistischen Moduls auf Standard-Wert;
    endif
    if $i \leq n_{precision}$
        then Genetischer Algorithmus;
            erhöhe Präzisions-Parameter um einen Schritt;
        else Lokales Optimierverfahren;
    endif
    Sortiere Such-Cache invers nach Fitness (d.h. erste Sequenzen sind die besten);
    Weise jeder Sequenz im Such-Cache als Fitness ihre negative Index-Nummer zu (0,-1,-2,...);
    Eliminiere alle Sequenzen im Such-Cache, die bereits im Sammel-Cache vorkommen;
    Addiere die verbleibenden (max. s) Sequenzen des Such-Cache zum Sammel-Cache;
    Leere Such-Cache;
endfor
Falls Sammel-Cache weniger als $s - n_{recombination}$ Sequenzen enthält,
    durch zufällige Sequenzen ergänzen;
Sortiere Sammel-Cache invers nach Fitness;
Lösche alle Sequenzen bis auf die ersten $s - n_{recombination}$ Sequenzen im Sammel-Cache;
Erzeuge Population aus Sequenzen durch Rekombinieren/Mutieren (siehe Seite 104);
Wähle aus dieser Population zufällig $n_{recombination}$ Sequenzen aus;
Füge die $n_{recombination}$ Sequenzen in den Sammel-Cache ein;

Folgende Variablen werden eingesetzt:

s	Anzahl auszuwählende neue Sequenzen
i	Index
$n_{recombination}$	Anzahl Sequenzen, die durch Rekombination/Mutation generiert werden
$n_{precision}$	Anzahl Präzisionsstufen des statistischen Moduls

Der Sammel-Cache enthält nun exakt s ausgewählte, neue Sequenzen. Diese müssen noch zur Population der bereits bewerteten Sequenzen hinzugefügt werden, damit auch sie in der nächsten Generation bewertet werden können.

Implementation des TST–Algorithmus

Der TST–Algorithmus wurde in den Rahmen der GOS (Kapitel 2) integriert. Die speziellen Eigenschaften des TST–Algorithmus finden sich ausschließlich in Nachfahren der Klasse `class_choose_new_individuals` — alle anderen Routinen der GOS können unverändert bleiben. Die Klasse führt neben diversen Verwaltungsroutinen das statistische Modul und das Auswahl–Modul hintereinander aus.

Im Auswahl–Modul finden ein genetischer Algorithmus und lokale Optimierverfahren Anwendung. Der genetische Algorithmus ist ebenfalls ein Nachfahre der GOS. Die genetischen Operatoren „Crossover“ und „Mutation“, der Caching-Algorithmus und die lokalen Optimierverfahren hängen vom konkreten Problem und dessen Datenstruktur ab und sind deswegen in (den Nachfahren) der Klasse `class_fitness` implementiert.

Notwendig sind außerdem Transformations-Routinen, die ein Problem in verschiedene Datenformate umwandeln können, da in der Regel die statistischen Verfahren eine andere Darstellung der Daten als die Optimierverfahren benötigen.

5.4 Benutzerdefinierte Parameter des TST–Algorithmus

Der TST–Algorithmus benötigt einige vom Benutzer vorzugebende Parameter, die in Tabelle 5.1 zusammengefaßt sind.

Die benutzerdefinierten Parameter des GRNN sind bereits in Tabelle 4.2 auf Seite 85 aufgeführt.

Insgesamt sind damit für die vorliegende Version des TST–Algorithmus genau *acht* Parameter durch den Benutzer festzulegen. Alle Parameter können in weiten Bereichen variiert werden, ohne daß sich am Verhalten des GRNN und des TST–Algorithmus Wesentliches ändert.

Parameter	Wert	sinnvoller Wertebereich
Anzahl Generationen g	10	≥ 1
Anzahl auszuwählender neuer Sequenzen s	64	$\geq n_{precision} + n_{recombination}$
Anzahl Sequenzen, die durch Rekombination/Mutation generiert werden $n_{recombination}$	$\frac{s}{4}$	$\frac{s}{8} \rightarrow \frac{s}{2}$
Anzahl Präzisionsstufen des statistischen Moduls $n_{precision}$	6	$1 \rightarrow s - n_{recombination}$

Tabelle 5.1: *Benutzerdefinierte Parameter des TST–Algorithmus*

5.5 Numerische Ergebnisse des TST–Algorithmus

In diesem Kapitel sollen die Arbeitsweise sowie die Leistungsfähigkeit des TST–Algorithmus an einigen typischen Beispielen demonstriert werden. Die dargestellten Programmläufe entsprechen dem *repräsentativen* Verhalten des Algorithmus und sind nicht etwa nur der eine, beste Programmlauf unter vielen.

5.5.1 Eindimensionales, reelles Beispiel

Am einfachsten ist die Arbeitsweise des TST–Algorithmus an einem eindimensionalen Problem zu verdeutlichen:

Gesucht ist das globale Maximum der Funktion

$$f(x) = \left| \frac{\sin x}{x} \right|^{\frac{1}{16}} \tag{5.1}$$

$$\text{Maxima bei } x_{max} = 0, \pm \left[\left(i + \frac{1}{2} \right) \pi - \frac{1}{\left(i + \frac{1}{2} \right) \pi} \right], \quad \text{mit} \quad i = 1, 2, \ldots \tag{5.2}$$

Das globale Maximum liegt exakt an der Stelle $x = 0$ und hat den Wert 1. Die übrigen Maxima liegen genähert an den in Gleichung (5.2) angegebenen Stellen. Im Bereich $x \in [-10; 40]$ hat die Funktion 16 Maxima. Der Definitionsbereich wurde in 10000 Abschnitte unterteilt, damit existieren genau 10000 mögliche Lösungen.

In den Abbildungen 5.1–5.3 ist die Funktion $f(x)$ gepunktet eingezeichnet. Mit Sternen werden die bereits bewerteten Punkte markiert, die dem GRNN als Trainingsset dienen. Die Approximation der Trainingspunkte durch das GRNN wird

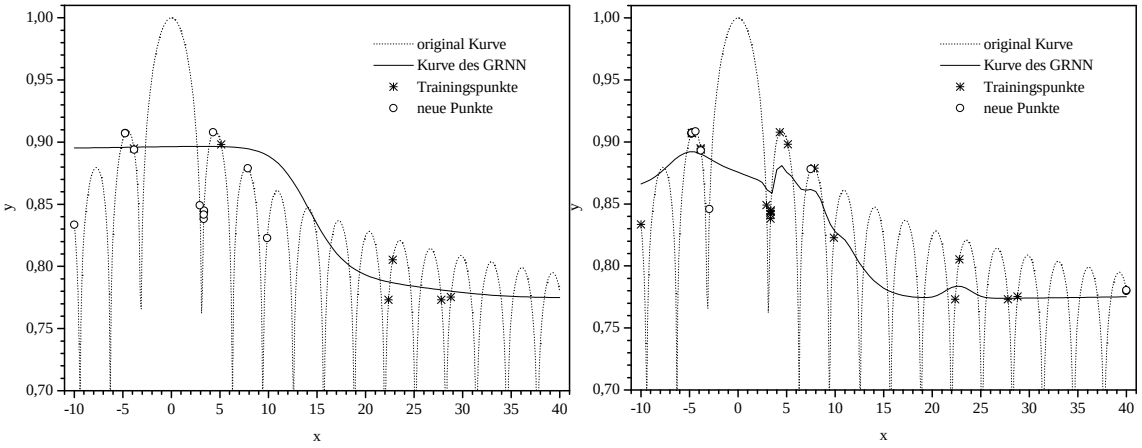

Abbildung 5.1: *Maximum von* $\left|\frac{\sin x}{x}\right|^{\frac{1}{16}}$*: Generationen 1 und 2*

als durchgezogene Linie eingezeichnet. Als Approximationskurve wird immer das GRNN mit Standard-Präzision (d.h. $m = 1$) verwendet. Der TST–Algorithmus nutzt natürlich auch die übrigen Präzisionseinstellungen zur Ermittlung der neuen Punkte. Die neu ausgewählten Punkte sind mit Kreisen gekennzeichnet.

Gestartet wird mit sechs zufällig ausgewählten Punkten, den Sternen im linken Diagramm in Abbildung 5.1. Der TST–Algorithmus wählt in jeder Generation 12 neue Punkte aus. Zwei der Startpunkte ($x = -3.85$, $x = 5.12$) liegen nahe

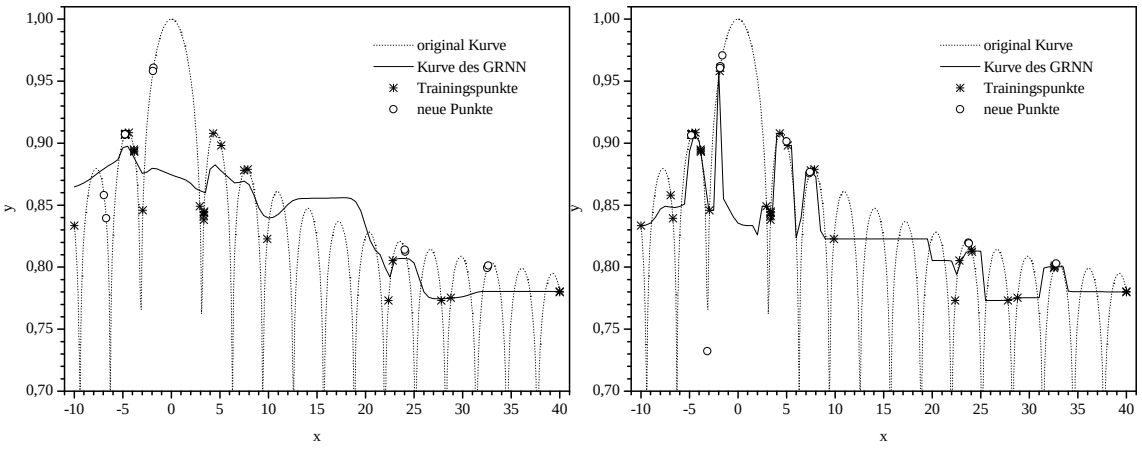

Abbildung 5.2: *Maximum von* $\left|\frac{\sin x}{x}\right|^{\frac{1}{16}}$*: Generationen 3 und 4*

bei den beiden höchsten lokalen Optima, die wiederum enge Nachbarn des globalen Optimums sind. Das GRNN approximiert mit einer ungefähr sigmoidalen Kurve ohne lokale Optima. Die Regressionskurve hat ihr globales Maximum an der Stelle $x \approx 3.5$. Der TST–Algorithmus wählt deshalb die Mehrzahl der neuen Punkte an dieser Stelle aus. Für eine höhere Präzision des GRNN ($m = \frac{1}{2}$) hat die Regressionskurve zusätzlich zwei lokale Maxima in der Nähe der beiden erwähnten Startpunkte.

Aus diesen Grund wählt der TST–Algorithmus auch noch einige der neuen Punkte in der Nähe der beiden höchsten lokalen Optima ($x = \pm 4.5$) aus. Diese Tendenz setzt sich in der 2.Generation fort.

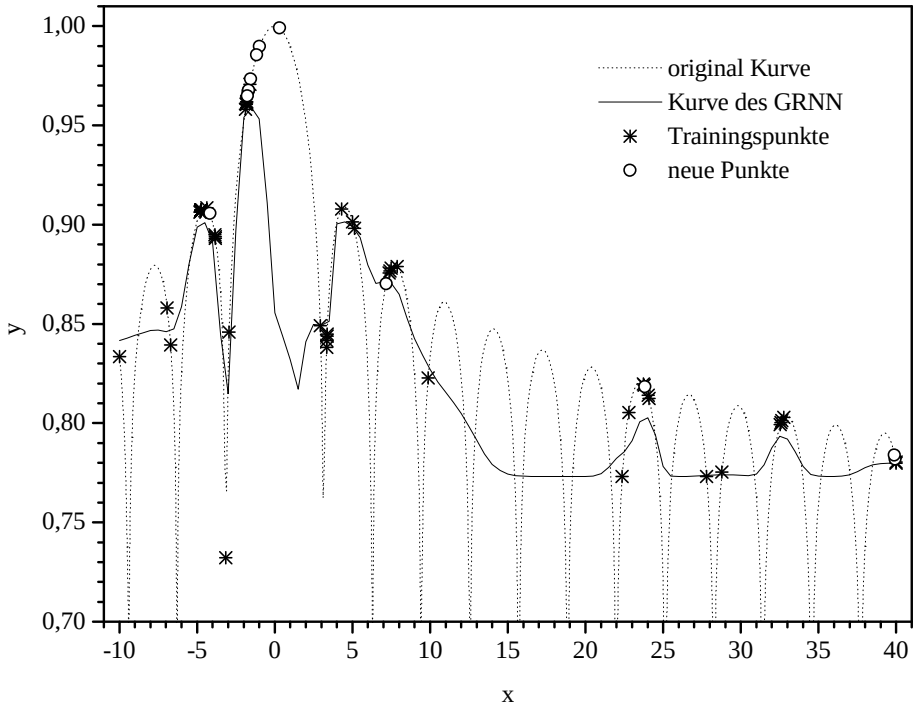

Abbildung 5.3: *Maximum von* $\left|\frac{\sin x}{x}\right|^{\frac{1}{16}}$*: 5.Generation*

In der 3.Generation (linkes Diagramm in Abbildung 5.2) bildet die Regressionskurve eine größere Anzahl lokaler Optima aus. An diesen Stellen ($x \approx -4.7,\ 24.1,\ 32.6$) werden eine Reihe neuer Punkte gewählt.

Interessant sind die neuen Punkte an der Stelle $x \approx -1.8$, da hier aufgrund der Verteilung der Trainingspunkte eigentlich kein lokales Optimum zu erwarten ist. Das GRNN erzeugt dort ein lokales Optimum durch den in Abbildung 4.5 auf Seite 67 beschriebenen Mechanismus. Dieses im Grunde unerwünschte Verhalten des GRNN bewirkt in Zusammenarbeit mit dem TST–Algorithmus eine gleichmäßigere Verteilung der neuen Punkte im Suchraum, da eine große Chance besteht, daß das GRNN zwischen zwei weiter entfernten Clustern an Trainingspunkten ein lokales Maximum ausbildet. Hier bewirkt dieses Artefakt des GRNN eine Beschleunigung der Optimierung, da die zwei neuen Punkte nahe an das globale Optimum gesetzt wurden, was sonst erst Generationen später stattgefunden hätte.

In der 4.Generation werden lediglich neue Punkte an bereits bekannte lokale Optima der Regressionskurve gesetzt, also kein wirklicher Optimierfortschritt.

In der 5. und letzten Generation sind genügend Punkte der Fitnesslandschaft abgetastet, so daß das GRNN die Fitnesslandschaft in der Umgebung des globalen Optimums ausreichend gut approximieren kann. Für sehr niedrige Präzision des GRNN ($m = 10$) sind die globalen Maxima der Approximationskurve und der Fitnesslandschaft nahezu deckungsgleich.

Der TST–Algorithmus hat in 5 Generationen mit insgesamt 66 Stichproben aus einer Menge von 10000 Datenpunkten das globale Optimum lokalisiert:
Der beste gefundene Punkt liegt an der Stelle $x = 0.3065$ mit dem Wert 0.99901883.

5.5.2 Zweidimensionales, reelles Beispiel

Das zweite reelle Beispiel ist die Suche nach dem globalen Maximum der sog. RASTRIGIN–Funktion (Kapitel 2.4.1, Seite 21). Zur besseren Darstellbarkeit wird der TST–Algorithmus auf die zweidimensionale Version der RASTRIGIN–Funktion angesetzt.

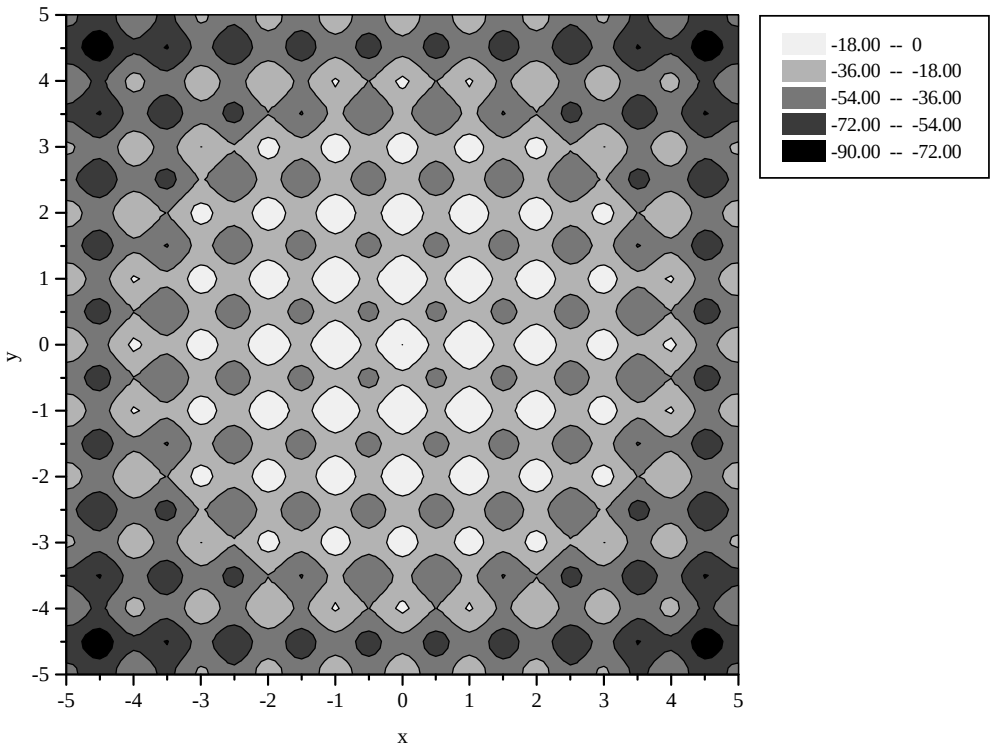

Abbildung 5.4: *Contourplot der zweidimensionalen* RASTRIGIN*–Funktion*

Eine räumliche Darstellung der zweidimensionalen RASTRIGIN–Funktion ist in Abbildung 2.3 auf Seite 22 angegeben. Die gleiche Funktion ist in Abbildung 5.4 als Contourplot von oben dargestellt. In dieser Art der Darstellung sind die Extrema

gut zu erkennen. Das globale Maximum liegt exakt an der Stelle $(0,0)$ und hat den Wert 0. Die Maxima liegen genähert an den Koordinaten $\left(i - \frac{3i}{3+4\pi^2 A}, j - \frac{3j}{3+4\pi^2 A}\right)$ mit $i, j \in \mathbb{Z}$. An den Koordinaten $\left(\frac{1}{2}\left[i - \frac{3i}{3-4\pi^2 A}\right], \frac{1}{2}\left[j - \frac{3j}{3-4\pi^2 A}\right]\right)$ liegen genähert die Minima mit $i, j \in \mathbb{Z} \setminus \{0\}$.

Der TST–Algorithmus startet mit 21 zufällig ausgewählten Punkten, die in Abbildung 5.5 als schwarze Kreuze markiert sind. In jeder Generation werden 20 neue Punkte hinzugenommen und als weiße Kreise eingezeichnet. Jede Dimension hat eine Auflösung von 10000 Punkten, damit stehen insgesamt 10^8 Lösungen zur Verfügung.

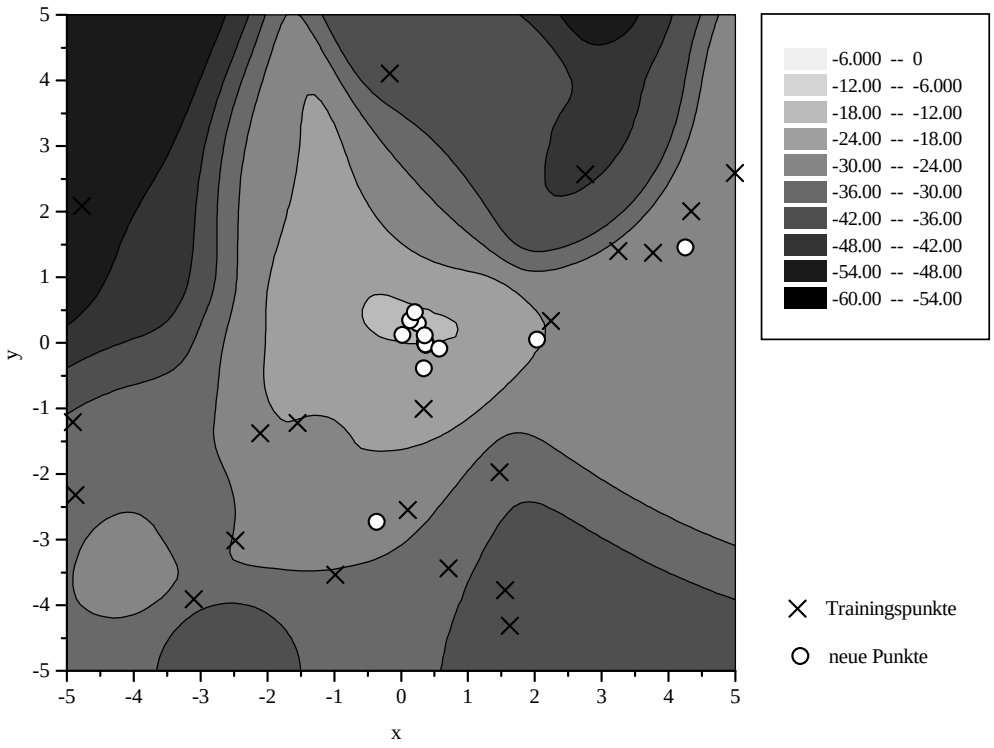

Abbildung 5.5: RASTRIGIN–*Funktion: 1.Generation*

Der beste Punkt der Startpopulation liegt an der Stelle $(0.337, -1.0089)$ und hat den Wert -16.34525. Bereits in der 1.Generation liegt die Mehrzahl der vom TST–Algorithmus ausgewählten Punkte in enger Nachbarschaft des globalen Optimums. Der beste Punkt hat die Koordinaten $(0.025, 0.117)$ und den Funktionswert -2.72001.

Offensichtlich hat das GRNN trotz der vielen lokalen Extrema kaum Schwierigkeiten bei der Erstellung einer Regressionsfläche, die bereits die generellen Eigenschaften der Fitnesslandschaft aufweist. Der Contourplot der approximierten Fläche zeigt ein deutliches Maximum an der Stelle $(0,0)$ des echten globalen Maximums.

Da das grobe Optimierziel bereits in der 1.Generation erreicht wurde, bringen weitere Generationen nur noch lokale Verfeinerungen des besten Punktes. In der

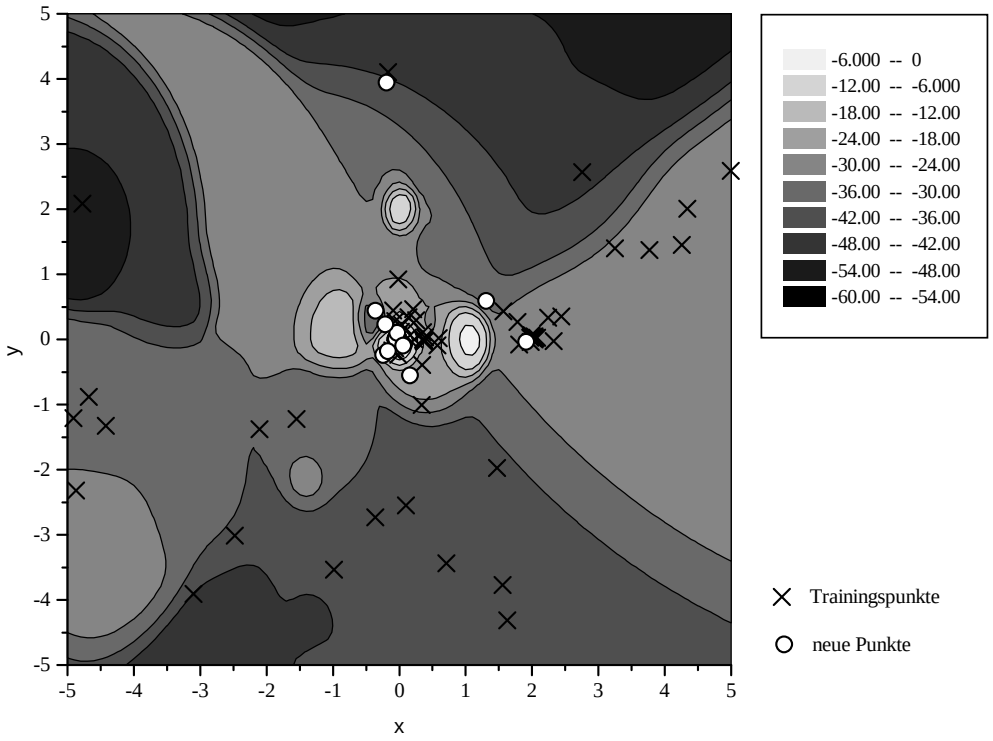

Abbildung 5.6: RASTRIGIN–*Funktion: 4.Generation*

4.Generation produzierte das GRNN die bereits aus Kapitel 5.5.1 bekannten Artefakte: Künstliche Maxima in den Bereichen $(1, 0)$, $(0, 2)$ und $(-1, 0)$.

Der beste Punkt liegt nach der 4.Generation an der Stelle $(-0.047, 0.001)$ mit dem Wert -0.43529. Es wurden insgesamt 101 Stichproben aus einem Suchraum der Größe 10^8 entnommen.

5.5.3 HAMMING–Abstand

Die bisherigen Testbeispiele des TST–Algorithmus nutzten reell kodierte Parameter in einer vorgegebenen Auflösung. Durch die geringe Anzahl Dimensionen ließ sich der Optimiervorgang leicht graphisch darstellen. Eine andere Klasse an Anwendungsbeispielen nutzt eine geringe Anzahl unterschiedlicher Buchstaben mit einer hohen Anzahl Dimensionen. Solche Fälle treten beispielsweise in der Biotechnologie bei der Optimierung von Polynukleotid- oder Peptid–Sequenzen auf.

Der einfachste Vertreter dieser Problemklasse ist das sog. HAMMING–Abstands–Problem, das in Kapitel 2.4.3 definiert wurde. Die Referenzsequenzen haben in den hier getesteten Fällen an jeder Position den Wert 0. Der Fitnesswert einer Sequenz ist identisch mit der Anzahl Positionen, die eine 1 enthalten. Die Zielsequenz ist deshalb an jeder Position mit 1 besetzt.

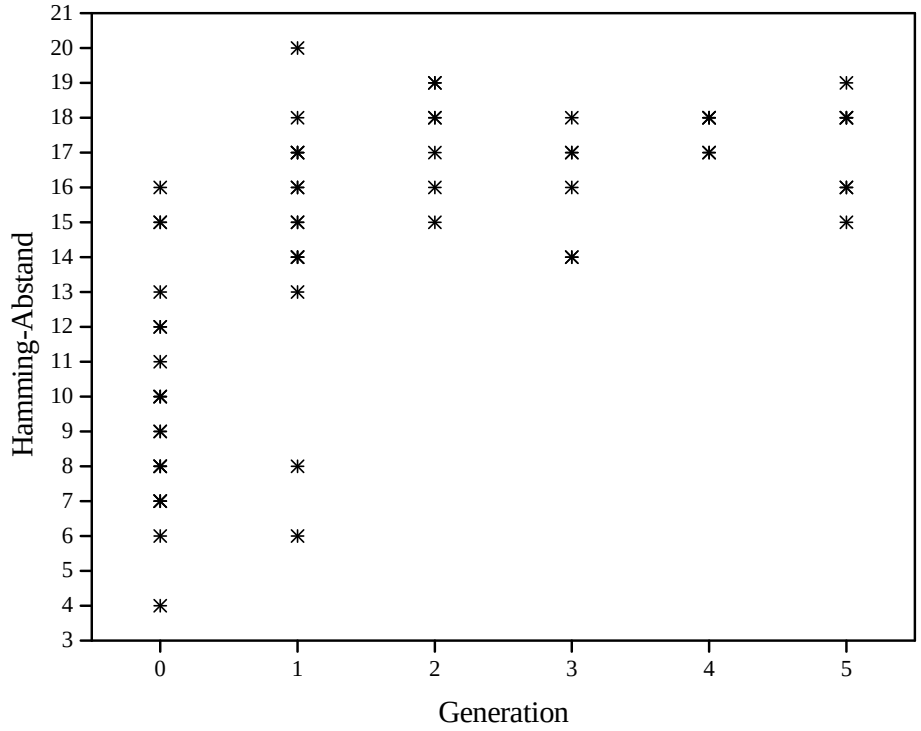

Abbildung 5.7: HAMMING–*Abstand Länge 20: Optimierverlauf durch TST–Algorithmus*

In Abbildung 5.7 ist der typische Verlauf einer Optimierung von binären Sequenzen der Länge 20 durch den TST–Algorithmus dargestellt. Der Algorithmus startet mit 21 zufällig ausgewählten Sequenzen und wählt in jeder Generation 20 neue Sequenzen aus. Bereits in der ersten Generation findet der TST–Algorithmus das globale Optimum 11111111111111111111 und benötigt dafür 41 Stichproben aus 1048576 möglichen Lösungen. Da das globale Optimum bereits gefunden wurde, kann der Algorithmus in den darauffolgenden Generationen nur noch Sequenzen mit einer maximalen Fitness von 19 finden. Der größte Teil der neuen Sequenzen in den Generationen $2 \rightarrow 5$ ist lediglich 1–2 Punktmutationen von der besten Sequenz entfernt. Die übrigen Sequenzen mit geringerer Fitness werden durch Crossover und Mutationen der besten Sequenzen generiert.

Im den Tabellen 5.2–5.4 werden die Startpopulation und die neuen Sequenzen der 1.Generation aufgelistet.

Da die Ziffern 0 und 1 mit der gleichen Wahrscheinlichkeit $\frac{1}{2}$ ausgewürfelt werden, ist die Häufigkeitsverteilung der Sequenzen der Startpopulation identisch mit der Binomialverteilung $\binom{20}{k}2^{-20}$ ($k = 0, 1, \ldots, 20$ ist Anzahl Einsen). Am häufigsten sind demnach Sequenzen mit der Fitness 10.

Die Startpopulation in Tabelle 5.2 hat hier mit der 15.Sequenz den höchsten Fitnesswert von 16. Die Approximation der Fitnesswerte durch das GRNN ist nahezu

Index	Sequenz	Fitness	Fitness durch GRNN
0	11111001101000001100	10	10.0020
1	00001101000011101000	7	6.9944
2	00101100001000111001	8	8.0104
3	01111101001110001101	12	12.0002
4	11000000101000110001	7	7.0019
5	00010000000110100111	7	7.0019
6	00000111100101101101	10	10.0003
7	00010101111100111010	11	11.0115
8	10000100111000111000	8	7.9923
9	11101111111010110101	15	14.9917
10	11000101000001011110	9	8.9980
11	10101010011010000000	7	6.8740
12	00100001010000000001	4	4.0117
13	11010011100111110011	13	12.9970
14	00100111111101100110	12	11.9992
15	00110110111111111111	16	15.9826
16	00110110001111010010	10	10.0040
17	10001000011011000000	6	6.0248
18	01010011101100100000	8	8.0036
19	00111111101110111011	15	14.9933
20	00001000011111101100	9	8.8919

Tabelle 5.2: HAMMING–*Abstand Länge 20: Startpopulation*

Index	Sequenz	Fitness	Fitness durch GRNN
21	11110110111111111110	17	15.7448
22	01110110111111111111	17	15.9704
23	00110110111011111111	15	15.9703
24	10110110111111111111	17	15.9680
25	00110110111111111101	15	15.9635
26	00110010111111111111	15	15.9626
27	00110110110111111111	15	15.9592
28	11101111111010110111	16	14.9877
29	11101111110010110101	14	14.9865
30	11101111111110110101	16	14.9841
31	11111111111010110101	16	14.9837
32	11100111111010110101	14	14.9835
33	11101111111000110101	14	14.9831
34	01110111111111111111	18	15.7897
35	11111111111111111111	20	15.0534

Tabelle 5.3: HAMMING–*Abstand Länge 20: Neue Sequenzen durch Suche in der durch das GRNN approximierten Landschaft*

perfekt.

Die neuen Sequenzen, die der TST–Algorithmus durch lokale und globale Suche in der durch das GRNN mit unterschiedlichen Genauigkeiten approximierten Fitnesslandschaft findet, verteilen sich um einige lokale Optima. In Tabelle 5.3 sind die zu den verschiedenen Optima gehörigen Sequenzen durch horizontale Linien voneinander getrennt. Es ist offensichtlich, daß die Sequenzen 21–27 Varianten der 15.Sequenz und die Sequenzen 28–33 um die 9.Sequenz verteilt sind. Die 34.Sequenz wurde bei sehr geringer Präzision gefunden. Die 35.Sequenz, die dem globalen Op-

timum entspricht, wurde bei geringster Präzision ($m = 10$) des GRNN ermittelt.

Dieses Beispiel zeigt, wie wichtig der Einsatz unterschiedlicher Präzisionen des statistischen Verfahrens ist. Die geringe Anzahl Startsequenzen bewirkte, daß das GRNN nicht die optimale Generalisierung der Fitnesslandschaft erreichen konnte, sondern mit zu großer Genauigkeit approximierte, also überfittete. Wäre ausschließlich nach maximalen Werten in der gelernten Landschaft gesucht worden, so wären dadurch nur in der engen Nachbarschaft der besten Sequenzen der Startpopulation neue Sequenzen ausgewählt worden. Dennoch ermittelte das GRNN den globalen Trend, der bei geringer Präzision auf das globale Optimum hinwies. Wichtig: Die beste Sequenz hat *nicht* den höchsten durch das GRNN approximierten Fitnesswert.

Index	Sequenz	Fitness	Fitness durch GRNN
36	01110101111111111111	17	15.4422
37	00110110110111111011	14	15.7863
38	11010111100111110010	13	12.9754
39	11000001001001010110	8	8.6470
40	00000100110000111000	6	8.0087

Tabelle 5.4: HAMMING–*Abstand Länge 20: Neue Sequenzen durch Crossover und Mutation*

Die letzten fünf Sequenzen wurden aus den bereits bekannten Sequenzen durch Mutation und Rekombination proportional zur Fitness konstruiert. Da bei diesem Beispiel das globale Optimum schon in der 1.Generation gefunden wurde, konnten diese Sequenzen nicht zur Suche beitragen.

Der TST–Algorithmus benötigte für die fünf Generationen insgesamt knapp 17 Sekunden Rechenzeit auf einem 300MHz-Pentium II–Rechner.

Ein etwas schwieriger zu lösendes Beispiel ist das HAMMING–Abstands–Problem der Länge 50.

Generation	Index	Sequenz	Fitness
0	5	00100111111101100110001101101111111111110011011000	32
	9	10100111111111110010011000010111101100111111000111	32
	49	10110111011101111101111001011101101000110010101101	32
1	95	10111111111111110111111101111101111110110011101001	40
2	115	10111111111111110110111110111111111111111011101001	42
3	182	11111111111111111110111110111111111111111011101001	44
4	248	11111111111111111110111110111111111111111011101101	45
5	310	11111111111111111110111110111111111111111111101111	47
6	385	11111111111111111111111111111111111111111111111111	50

Tabelle 5.5: HAMMING–*Abstand Länge 50: Optimierung während sechs Generationen durch TST–Algorithmus*

Der TST–Algorithmus beginnt hier mit 51 zufällig ausgewählten Sequenzen und fügt in jeder Generation 64 neue Sequenzen zur Population hinzu.

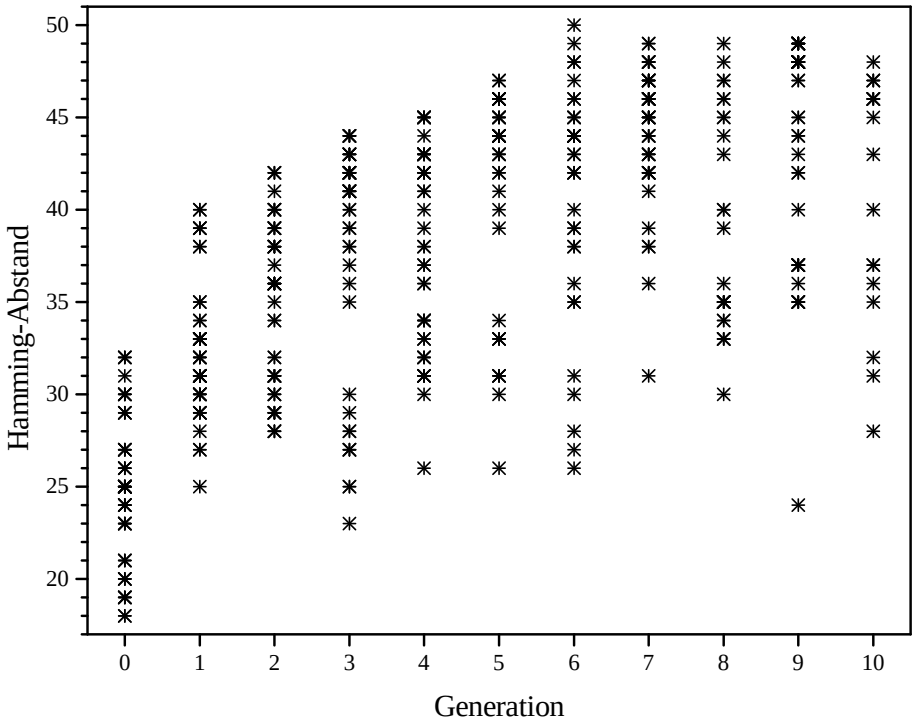

Abbildung 5.8: HAMMING–*Abstand Länge 50: Optimierverlauf durch TST–Algorithmus*

In Abbildung 5.8 und Tabelle 5.5 ist ein kontinuierlicher Optimierprozeß zu erkennen, bis in der 6.Generation das globale Optimum mit der Fitness 50 gefunden ist. Der TST–Algorithmus benötigt 435 Stichproben aus einem Lösungsraum der Größe $2^{50} \approx 10^{15}$ Sequenzen. Die zehn Generationen des TST–Algorithmus benötigten für das HAMMING–Abstands–Problem der Länge 50 insgesamt etwa 31 Minuten CPU-Zeit.

5.5.4 Spinglas

In Kapitel 2.4.2 wurde die sog. Spinglas–Landschaft als Beispiel für frustierte Fitnesslandschaften eingeführt. Bei allen hier gerechneten Beispielen sind die Einträge der Wechselwirkungsmatrix gleichverteilte reelle Zufallszahlen aus dem Intervall $[-5; +5]$. In Darstellungen symbolisieren „$-$“ und „$+$“ Spins der Ausrichtungen -1 und $+1$. Als Testbeispiele des TST–Algorithmus wurden zwei Fitnesslandschaften mit 20 und 50 miteinander wechselwirkenden Spins generiert. Gesucht ist jeweils das Tupel an Spins mit der maximalen Wechselwirkungsenergie nach Gleichung (2.3).

5.5.4.1 Spinglas Länge 20

Bei 20 Spins kann das Zieltupel noch durch Ausprobieren aller 2^{19} Spineinstellungen ermittelt werden. Für die hier eingesetzte Wechselwirkungsmatrix ergibt sich das Tupel `---++-+--+--+-+++-+-` bzw. sein inverses `+++--+-++-++-+---+-+` mit der Energie 192.083.

Der TST–Algorithmus startet mit 21 Sequenzen und addiert in jeder Generation 64 Sequenzen dazu. Dem Algorithmus steht an keiner Stelle Information über die Symmetrie der Fitnesslandschaft zur Verfügung. Der TST–Algorithmus findet bei 20 Spins mit hoher Wahrscheinlichkeit das globale Optimum, unabhängig von der verwendeten Wechselwirkungsmatrix oder der Startpopulation.

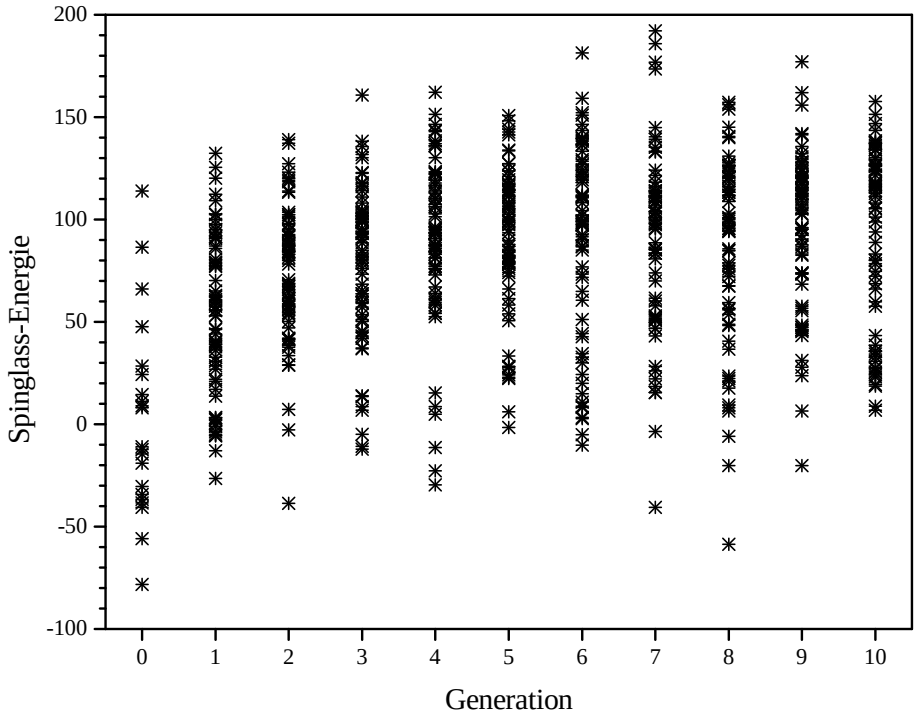

Abbildung 5.9: *Spinglas Länge 20 durch TST–Algorithmus*

In Abbildung 5.9 ist ein typischer Programmlauf dargestellt: Es wechseln sich langsame Anstiege und Sprünge der besten Fitness mit Stagnationsphasen ab, bis in der 7.Generation eines der beiden globalen Optima gefunden wird.

Die besten Spin-Sequenzen jeder Generation sind in Tabelle 5.6 aufgeführt. In der letzten Zeile sind die Änderungen von der besten Sequenz der Startpopulation bis hin zur besten gefundenen Sequenz zusammengefaßt — der TST–Algorithmus änderte insgesamt 8 der 20 Spins.

Generation	Index	Sequenz	Fitness	Anzahl Unterschiede
0	12	++++---+-+++++++---+	113.959	
		| |		2
1	21	++++---+--+++++---+	132.264	
		|		1
2	88	++++---+--+++++--+-+	138.997	
		||| || | |		7
3	153	+++-++--+-++-+---+-+	160.789	
		|		1
4	216	+++--+--+-++-+---+-+	162.095	
		|		1
5	277	+++--+--+-++-----+-+	150.872	
		| | |		3
6	341	+++-++-++-++-+---+-+	181.441	
		|		1
7	405	+++--+-++-++-+---+-+	192.083	
0	12	++++---+-+++++++---+	113.959	
		| | || | || |		8
7	405	+++--+-++-++-+---+-+	192.083	

Tabelle 5.6: *Spinglas Länge 20: Optimierung durch TST–Algorithmus während sieben Generationen. Durch vertikale Striche werden die Positionen der Unterschiede zwischen den jeweils besten Sequenzen angezeigt.*

Auffällig ist der Fitness-Sprung in der 3.Generation, der mit sieben Punktmutationen einhergeht. Dem GRNN gelang es an dieser Stelle, aus den verfügbaren Sequenzen-Fitness–Paaren die Lage eines der globalen Optima „grob“ zu lokalisieren. Die Spin-Konfiguration der 3.Generation ist nur noch zwei Punktmutationen von der besten Lösung entfernt. In der 5.Generation konnte keine Verbesserung erzielt werden, trotzdem halfen die dabei ausgewählten Sequenzen dem Algorithmus bei der genaueren Approximation der Fitnesslandschaft, der in einem Fitness-Sprung in der 6.Generation resultiert.

Der TST–Algorithmus benötigte 469 Stichproben von $2^{20} = 1048576$ möglichen Spin-Konfigurationen zur Ermittlung eines der beiden globalen Optima.

5.5.4.2 Spinglas Länge 50

Aufgrund der vielversprechenden Ergebnisse mit 20 Spins wurde der Schwierigkeitsgrad auf 50 Spins erhöht. Die Feststellung der globalen Optima durch konsequente Berechnung aller 2^{49} Spinglas-Energien erwies sich als zu rechenaufwendig. Deshalb wurde mittels einer Kombination aus genetischem Algorithmus und lokalem Optimierverfahren nach den Lösungen maximaler Energie gesucht. Das Tupel `-+--+-----+-+--+-++-+-+--++++++++-+----++-+-++----` mit der Wechselwirkungsenergie 736.436 wurde durch einige hundert Suchläufe ermittelt. Dieses

Tupel und sein Inverses werden im folgenden zur sprachlichen Vereinfachung als „globales Optimum“ bezeichnet.

Der TST–Algorithmus fängt mit 51 Sequenzen an und wählt in jeder Generation 64 neue Sequenzen aus.

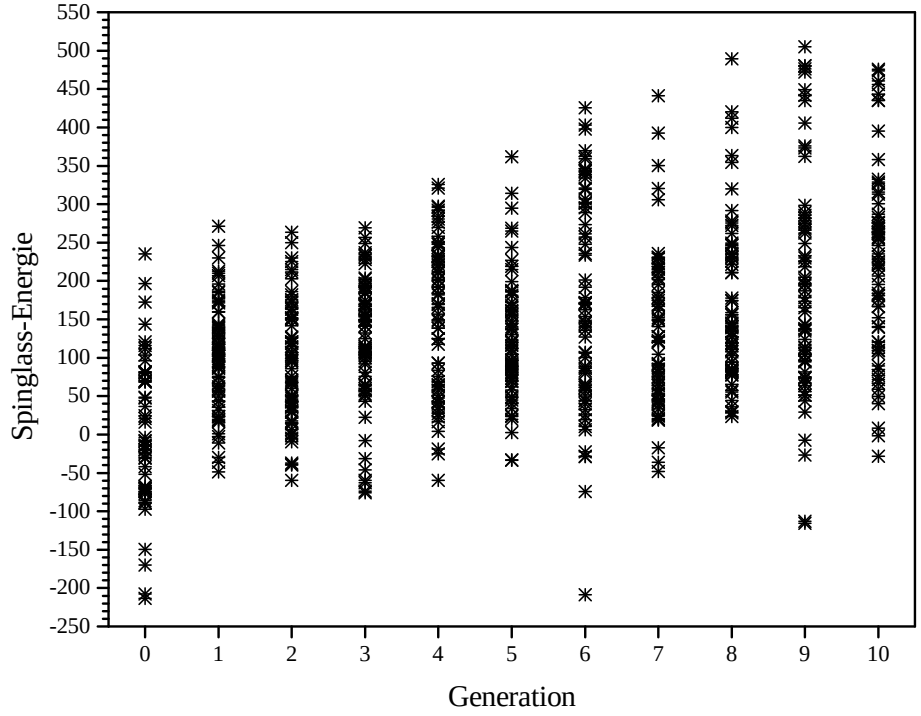

Abbildung 5.10: *Spinglas Länge 50 durch TST–Algorithmus*

Das wichtigste Ergebnis: Unabhängig von der gewählten Wechselwirkungsmatrix und der jeweiligen Startpopulation gelang es dem TST–Algorithmus in keinem einzigen Programmlauf, das globale Optimum zu ermitteln! Der Simulationsverlauf in Abbildung 5.10 und der maximale Energiewert von 504 sind typisch. Die besten gefundenen Sequenzen erreichen mit Energien von etwa 600 ungefähr 80% der Energie des globalen Optimums.

In Tabelle 5.7 werden die beste Sequenz der Startpopulation sowie die beste durch den TST–Algorithmus gefundene Sequenz jeweils mit den beiden globalen Optima verglichen. Auffällig ist, daß die beste Sequenz der Startpopulation „näher“ am Optimum `opt2` (19 Unterschiede) als am Optimum `opt1` (31 Unterschiede) liegt, während für die beste, durch den TST–Algorithmus gefundene Sequenz das Umgedrehte gilt. Der Grund dafür liegt im einem großen Sprung durch den Suchraum in der 4.Generation. Die jeweils besten Spin-Konfigurationen der 3. und 4.Generation unterscheiden sich in 28 von 50 Spins, d.h. fast 60% der Spins wurden invertiert.

Generation	Index	Sequenz	Fitness	
0	19	+-++-+-+----+--++--++-++-+------+--+-+---+--+++++	235.368	
		||||| | | ||| | ||||| || | ||| | |||		31
	opt1	-+--+-----+-+--+-++-+-+--++++++++-+----++-+-++----	736.436	
0	19	+-++-+-+----+--++--++-++-+------+--+-+---+--+++++	235.368	
		| || |||| ||| || || | | | |		19
	opt2	+-++-+++++-+-++-+--+-+-++--------+-++++--+-+--++++	736.436	
9	569	++-+++++--++++--+-+-+++++++--+-+++--+-++-++-+--+---+	504.904	
		| | || | || || | |||| | | || | |		20
	opt1	-+--+-----+-+--+-++-+-+--++++++++-+----++-+-++----	736.436	
9	569	++-+++++--++++--+-+-+++++++--+-+++--+-++-++-+--+---+	504.904	
		|| | || | |||| | | | ||| ||| ||||| ||||		30
	opt2	+-++-+++++-+-++-+--+-+-++--------+-++++--+-+--++++	736.436	

Tabelle 5.7: *Spinglas Länge 50: Vergleich der besten Sequenzen der 0. und 9.Generation mit den globalen Optima* `opt1` *und* `opt2`*. In der letzten Spalte steht die Anzahl Unterschiede zwischen zwei Sequenzen.*

Generation	Index	Sequenz	Fitness	
0	19	+-++-+-+----+--++--++-++-+------+--+-+---+--+++++	235.368	
		|		1
1	56	+-++-+-+----+--++-+++-++-+------+--+-+---+--+++++	271.082	
		|		1
2	117	+-++-+-+----+--++-++++++-+------+--+-+---+--+++++	263.241	
		| | | | |		4
3	182	+-++-+-+----+--++--+++++++------+--+-+------+++++	269.323	
		|| || | ||| ||| | | | | || ||| ||||| |||		28
4	253	++-+-++--+-+-+---+-+++++--+-+-+--+--++-++-++-+---+	325.701	
		|		1
5	307	++-++++--+-+-+---+-+++++--+-+-+--+--++-++-++-+---+	361.748	
		| |		2
6	419	++-++++--+++-+-+-+-+++++--+-+-+--+--++-++-++-+---+	425.306	
		|		1
7	435	++-++++--++++--+-+-+++++--+-+-+--+--++-++-++-+---+	440.995	
		|		1
8	500	++-++++--++++--+-+-+++++--+-+-+--+--++-++-+--+---+	489.381	
		|		1
9	569	++-++++--++++--+-+-+++++--+-+++--+-++-++-+--+---+	504.904	
0	19	+-++-+-+----+--++--++-++-+------+--+-+---+--+++++	235.368	
		|| | || ||| || | || | ||| ||| ||| |||		29
9	569	++-++++--++++--+-+-+++++--+-+++--+-++-++-+--+---+	504.904	

Tabelle 5.8: *Spinglas Länge 50: Optimierung durch TST–Algorithmus während neun Generationen. In der letzten Spalte steht die Anzahl Unterschiede zwischen zwei Sequenzen.*

In Tabelle 5.8 sind die besten Spin-Sequenzen jeder Generation aufgeführt. In der letzten Zeile sind die Änderungen von der besten Sequenz der Startpopulation bis hin zur besten gefundenen Sequenz zusammengefaßt — der TST–Algorithmus änderte insgesamt 29 der 50 Spins, den größten Teil davon in der 4.Generation. In der 2. sowie der 10.Generation gelang dem TST–Algorithmus keine Verbesserung des bisherigen Ergebnisses, in den übrigen Generationen konnte der Algorithmus optimieren. Bis auf die große Konfigurations-Änderung in der 4.Generation wurde der Optimierfortschritt durch vergleichsweise geringfügige Änderungen (1–4 Spin-Inversionen)

erzielt.

Der TST–Algorithmus benötigte hier 691 Stichproben von $2^{50} \approx 10^{15}$ möglichen Spin-Konfigurationen um ein lokales Optimum mit einer Wechselwirkungsenergie von etwa 68% der besten bekannten Sequenz zu ermitteln.

5.5.5 RNS–Sekundärstruktur

Die bisherigen Beispiele demonstrierten die Fähigkeit des TST–Algorithmus, mit einer vergleichsweise geringen Anzahl an Stichproben gute bis sehr gute Optima lokalisieren zu können. Der TST–Algorithmus wurde von Anfang an mit der Absicht entwickelt, auch bei Problemen der Molekülоptimierung behilflich zu sein. Während der Entwicklung des TST–Algorithmus bestand nicht die Möglichkeit, den Algorithmus iterativ mit *experimentell* bewerteten Molekülen anzuwenden. Das folgende Beispiel kommt einer Aufgabenstellung aus der Biotechnologie etwas näher, obwohl es ebenfalls künstlich im Computer generiert ist.

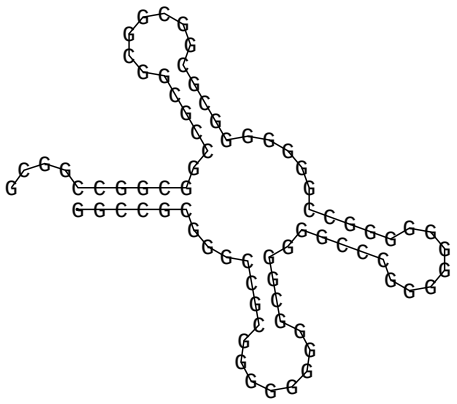

Abbildung 5.11: *tRNS–Sekundärstruktur als Optimierziel*

Ziel ist eine 76 Nukleotide lange RNS-Sequenz bestehend aus den Nukleotiden G und C mit der Sekundärstruktur einer tRNS (siehe Abbildung 5.11). Die Sekundärstruktur der minimalen freien Energie einer RNS-Sequenz wird mit dem `Vienna RNA Package` [52] berechnet. Näheres dazu in Kapitel 2.4.4. Als Fitness einer RNS-Sequenz dient die negative *tree-edit–distance* — ein Strukturabstandsmaß — zwischen der Struktur der RNS-Sequenz und der Struktur der tRNS dividiert durch die doppelte Länge der RNS-Sequenz.

Der TST–Algorithmus fängt mit 51 zufällig ausgewählten RNS-Sequenzen an und wählt in jeder Generation 64 neue Sequenzen aus.

	Index	Sequenz	MFE	Fitness	
0	47	GG.G...G..G.G..G..GGGG.G...GGGGGGGGGG..GGGGG.GGGGGG.G.GGGGGG...G....G.G.G.GG ....((((((((((.....))))))((((........)))).)))))..(((((((.....))))))).... |	-42.3	-0.289474	1
1	56	GG.G...GG.G.G..G..GGGG.G...GGGGGGGGGG..GGGGG.GGGGGG.G.GGGGGG...G....G.G.G.GG .((.(((((((...)))))))))..(((((........)))))..(((.((((((.((....)))))))).))).... |	-43.2	-0.302632	1
2	116	GG.G...GG.G.G..G..GGGG.G...GGGGGGGGGG..GGGGGGGGGGGG.G.GGGGGG...G....G.G.G.GG .((.(((((((...)))))))))..(((((........)))))........(((((((.....))))))).... |	-42.3	-0.263158	1
3	179	GG.G...GG.G.G..G..GGGG.G...GGGGGGGGGG..GGGGGGGGGGGG.G.GGGGGG..G....G.G.G.GG .((.(((((((...)))))))))..(((((........)))))........(((((((.....))))))).... || |||| | ||||| || ||| || | | | | | | | || |||| || ||| ||	-42.3	-0.263158	40
4	254	...GGGG.GGG..GG.G...G.G....G..GGGG.GGG..GG.GG.G.GGGGGGGGG..G.GG..GG..G...GGG ((((((((((....))))))(((((...))))).(((((((..(((((.......)))))))))))).)))))) || |||| | ||||| || ||| || | | | | | | | || |||| || ||| ||	-51.1	-0.236842	39
5	321	GG.G...GG.G.G..G..GGGG.G...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.G.G.GG .((.(((((((...)))))))))..(((((........)))))..... (((((((.......)))))))........ |	-39.5	-0.236842	1
6	378	GG.G...GG.G.G..G..GGGG.G...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.GGG.GG .((.(((((((...)))))))))..(((((........)))))..... (((((((.......)))))))........ |	-39.5	-0.236842	1
7	435	GG.G...GG.G.GG.G..GGGG.G...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.GGG.GG .((.(((((((...)))))))))..(((((........)))))..... (((((((.......)))))))........ | | | |	-39.5	-0.236842	4
8	499	GG.G...GG.G.G..G..GGGGGG...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.G.G. ....(((((((...)))))))..(((((.......)))))........(((((((.......)))))))........ | | |	-38.8	-0.223684	3
9	567	GG.G...GG.G.GG.G..G.GGGG...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.GGG.G. .(((((((.((((....)))).(((((.......)))))).........(((((((.......)))))))..))))))) |	-39.1	-0.131579	1
10	629	G..G...GG.G.GG.G..G.GGGG...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.GGG.G. ..((((((.((((....)))).(((((.......)))))).........(((((((.......)))))))..)))))).. |	-36.4	-0.118421	

Tabelle 5.9: *tRNS Länge 76: Optimierung durch TST–Algorithmus während zehn Generationen. In der ersten Spalte steht die Generation; in der letzten Spalte steht die Anzahl Unterschiede zwischen zwei Sequenzen. Das Nukleotid C ist der besseren Lesbarkeit halber durch einen Punkt ersetzt. MFE ist die Abkürzung von „Minimaler freier Energie“.*

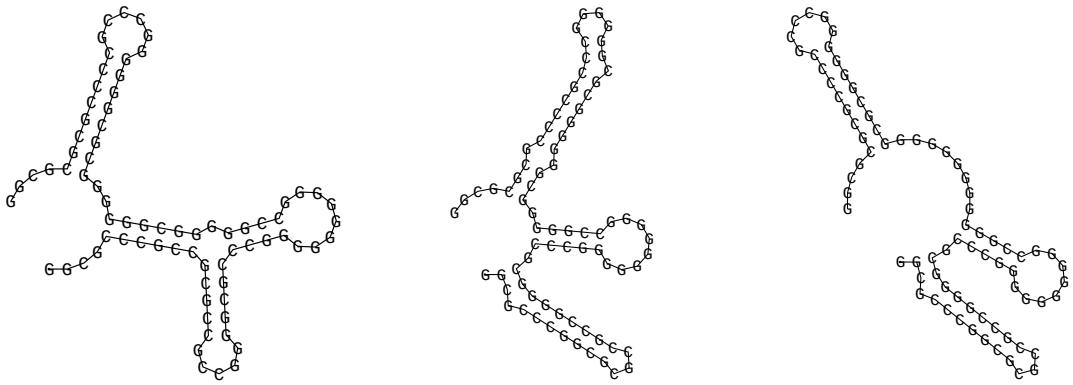

Abbildung 5.12: *Beste (t)RNS–Sekundärstrukturen der Generationen 0, 1 und 2*

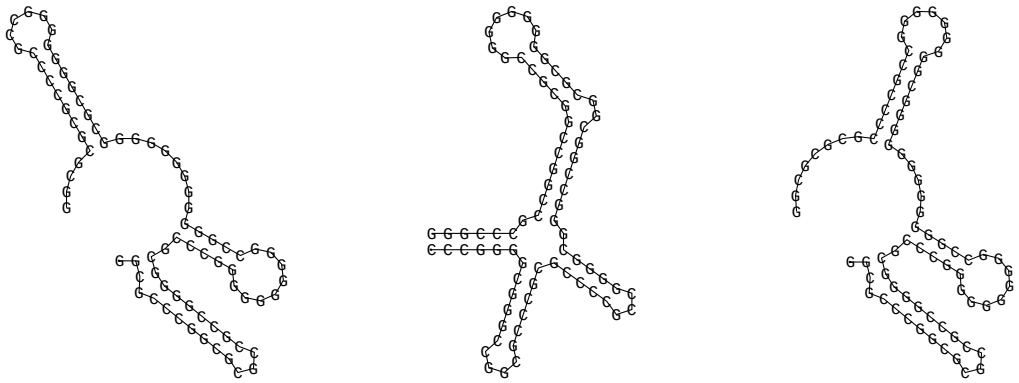

Abbildung 5.13: *Beste (t)RNS–Sekundärstrukturen der Generationen 3, 4 und 5*

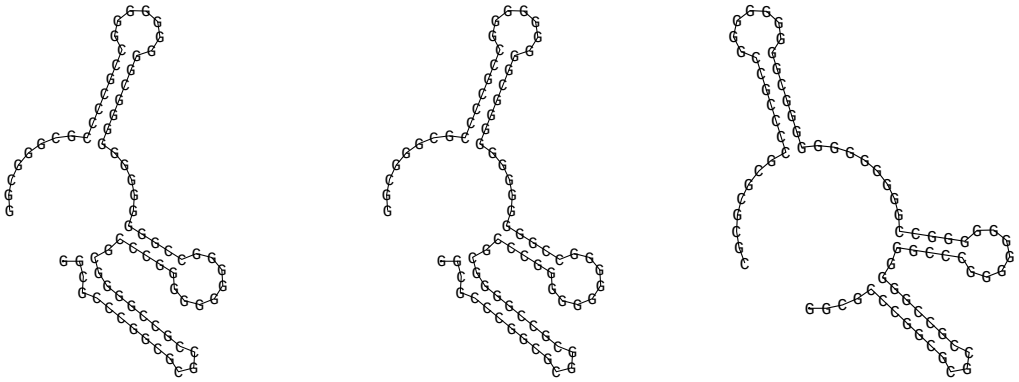

Abbildung 5.14: *Beste (t)RNS–Sekundärstrukturen der Generationen 6, 7 und 8*

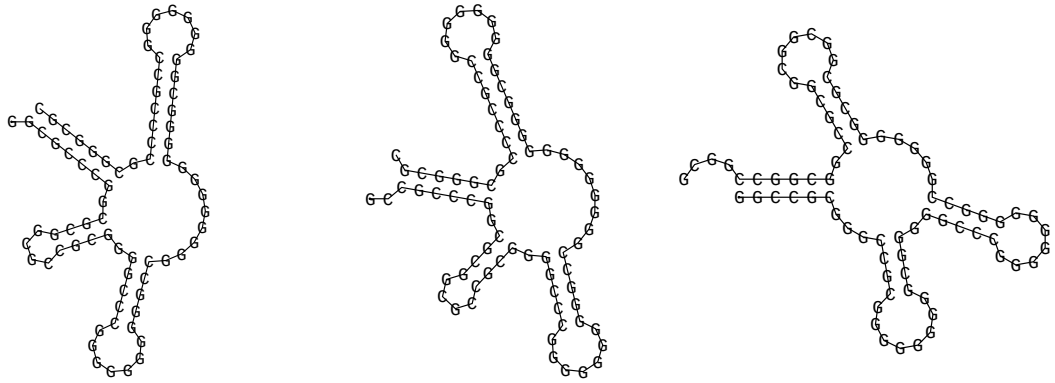

Abbildung 5.15: *Beste (t)RNS–Sekundärstrukturen der Generationen 9 und 10 sowie tRNS–Zielstruktur*

In Tabelle 5.9 ist der Optimierverlauf mit den jeweils besten Sequenzen jeder Generation aufgelistet. Unter der Sequenz ist die zugehörige Sekundärstruktur in der Bracket-Notation angegeben. Zur besseren Veranschaulichung werden die Sekundärstrukturen in den Abbildungen 5.12–5.15 graphisch dargestellt.

Wie beim Spinglas Länge 50 war der TST–Algorithmus in keinem einzigen Lauf

in der Lage, eines der globalen Optima zu identifizieren. Immerhin weist nach Abbildung 5.15 die beste Sequenz des hier vorgestellten Laufes eine große Ähnlichkeit mit der Zielstruktur auf: Sie besitzen beide die gleiche Anzahl an Strukturelementen wie hairpins (3), loops (1) und stems (1), wenn auch leicht gegeneinander verschoben.

Abgesehen von den großen Sprüngen durch den Sequenzraum in der 4. und 5.Generation fanden die Optimierungen weitgehend durch geringfügige Modifikationen der Sequenzen statt.

Generation	Index	Sequenz	Fitness	
0	47	GG.G...G..G.G..G..GGGG.G...GGGGGGGGGG..GGGGG.GGGGGG.G.GGGGGG...G....G.G.G.GG	-0.289474	
		| | | | | | | | | |		10
10	629	G..G...GG.G.GG.G..G.GGGG...GGGGGGGGGG..GGGGGGGGGGGG.GGGGGGGGG..G....G.GGG.G.	-0.118421	

Tabelle 5.10: *tRNS Länge 76: Unterschiede zwischen der besten Sequenz der Startpopulation und der besten gefundenen RNS-Sequenz. In der letzten Spalte steht die Anzahl Unterschiede zwischen beiden Sequenzen.*

Insgesamt modifizierte der TST–Algorithmus 10 Nukleotide von 76, um von der besten RNS-Sequenz der Startpopulation zur besten gefundenen RNS-Sequenz zu gelangen. Der TST–Algorithmus benötigte dazu 691 Stichproben von $2^{76} \approx 7.6 \cdot 10^{22}$ möglichen RNS-Sequenzen.

Hauptkomponentenanalyse des Laufs

Bei der bisherigen Beschreibung des Simulationslaufs wurden nur die jeweils beste Sequenz jeder Generation berücksichtigt. Interessant ist auch die Frage, wie der TST–Algorithmus die Population selbst durch den Sequenzraum bewegt. Es tritt dabei das Problem auf, den 76-dimensionalen Suchraum möglichst topologieerhaltend zweidimensional abzubilden. Wie schon in Kapitel 3.7.5 wird hier das Werkzeug der Hauptkomponentenanalyse zu Hilfe genommen.

In Abbildung 5.16 ist die vollständige Population aus 691 Sequenzen im Koordinatensystem ihrer ersten beiden Hauptkomponenten aufgetragen. Der Bereich der Fitnesswerte wurde in vier Abschnitte unterteilt und jedem Abschnitt jeweils eine Farbe zugewiesen. Die Population teilt sich offensichtlich in eine Reihe von Clustern auf.

In Abbildung 5.17 sind zum einen die Startpopulation und zum anderen die Sequenzen der ersten Generation in den ersten beiden Hauptkomponenten aufgetragen. Die Startpopulation ist, wie zu erwarten, weitgehend zufällig im Sequenzraum

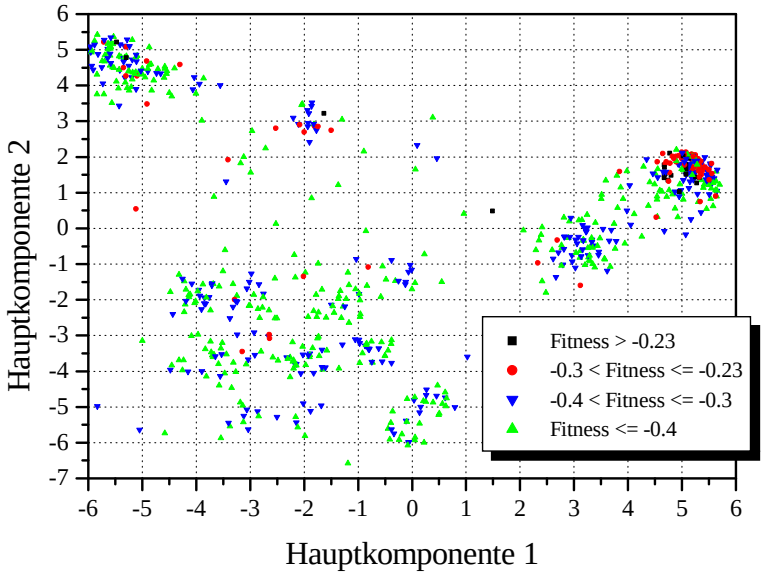

Abbildung 5.16: *Hauptkomponentenanalyse: Alle 691 Sequenzen in den ersten beiden Hauptkomponenten. Die Farben repräsentieren die Fitnesswerte.*

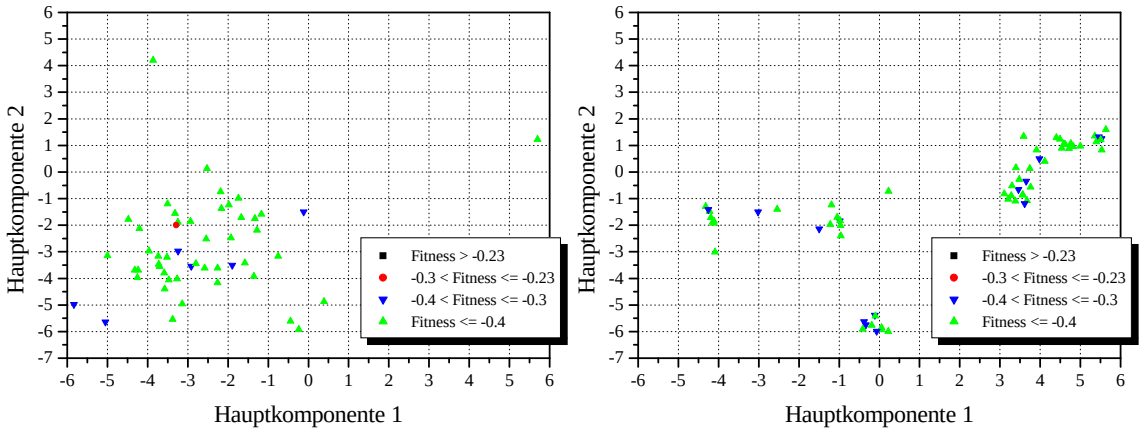

Abbildung 5.17: *Hauptkomponentenanalyse: Startpopulation und erste Generation*

verteilt; lediglich die Punkte $P_1 := (5.7; 1.2)$ und $P_2 := (-3.9; 4.2)$ liegen etwas exponierter. Dies ist ein Resultat der Hauptkomponentenanalyse, da in den weiteren Generationen ein großer Teil der Punkte verteilt um diese beiden Punkte ausgewählt werden wird. Die Sequenzen der ersten Generation sind im Wesentlichen auf drei Cluster aufgeteilt, wobei der Großteil in die Region um Punkt P_1 fällt. An diesem Verhalten ändert sich in den folgenden zwei Generationen nichts.

In der vierten Generation werden nun die meisten Sequenzen in der Umgebung von Punkt P_2 ausgewählt, unter anderem auch die 254.Sequenz, die die bis jetzt größte Fitness besitzt. Dies ist die Erklärung für den scheinbaren Sprung durch den Sequenzraum von der dritten zur vierten Generation: Der TST–Algorithmus

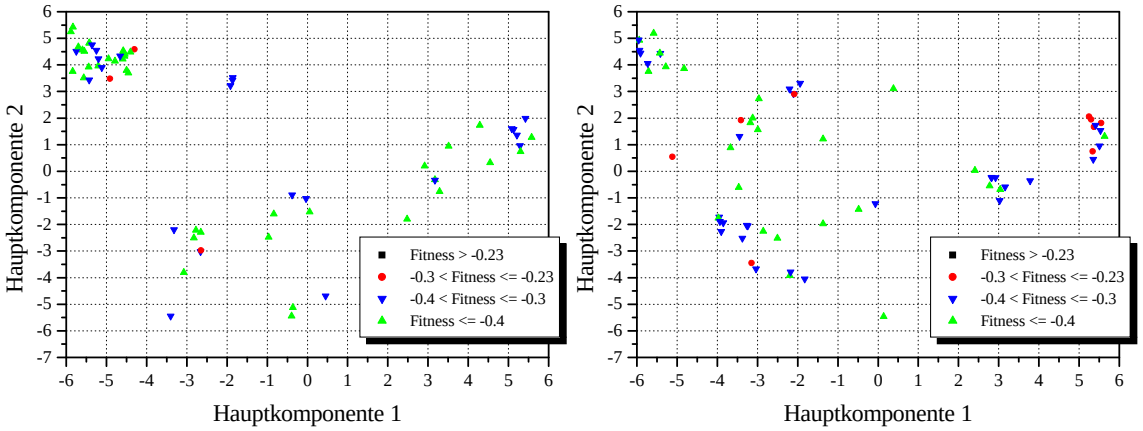

Abbildung 5.18: *Hauptkomponentenanalyse: Generationen 4 und 5*

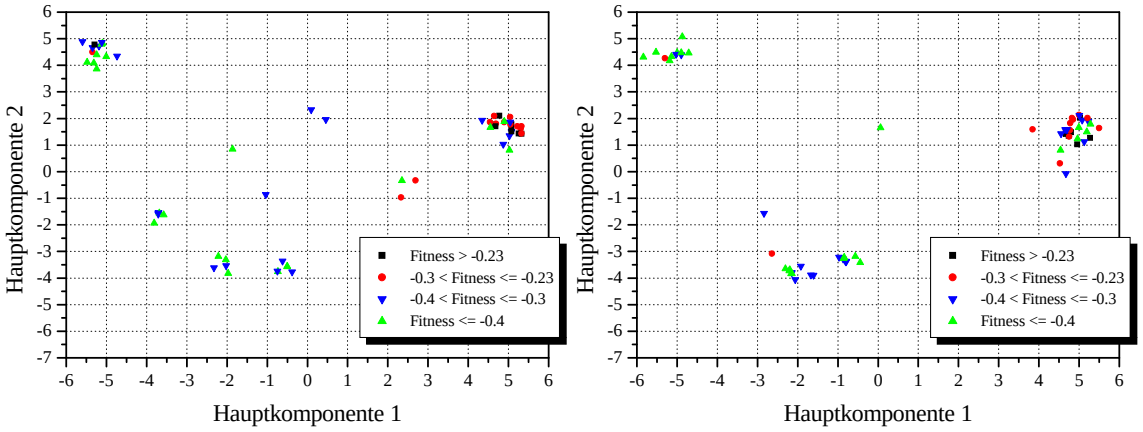

Abbildung 5.19: *Hauptkomponentenanalyse: Generationen 9 und 10*

sucht parallel unterschiedliche lokale Optima der approximierten Fitnesslandschaft ab. Die besten Sequenzen bis zur dritten Generation stammen aus der Umgebung von Punkt P_1; in der vierten Generation aber von P_2.

Ab der fünften Generation stammt die jeweils beste Sequenz immer aus der Umgebung von Punkt P_1, obwohl weiterhin bis zu zehn verschiedene lokale Optima abgescannt werden. Aus Platzgründen sind hier lediglich die letzten beiden Optimierzyklen in Abbildung 5.19 dargestellt.

5.6 Statistische Eigenschaften des TST–Algorithmus

Bisher wurden nur einzelne, ausgewählte Simulationsläufe vorgestellt. In diesem Kapitel soll das Optimierverhalten des TST–Algorithmus mit einer statistischen Anzahl

Versuchsläufe für jeden zu untersuchenden Parameter gezeigt werden. Untersucht wurden der Einfluß von Rauschen bei der Fitness-Bestimmung (Kapitel 5.6.2) und durch Hinzufügen zufälliger Sequenzen (Kapitel 5.6.3).

5.6.1 Eigenschaften der Test-Fitnesslandschaft

Als Test-Fitnesslandschaft dient im folgenden (wie schon in Kapitel 5.5.4.2) Spinglas Länge 50 mit einer festgelegten Wechselwirkungsmatrix, deren Elemente gleichverteilt aus dem Intervall $[-5; +5]$ stammen.

Der einfachste denkbare „Optimieralgorithmus“ besteht darin, in jeder Generation eine Anzahl n an Spin–Sequenzen zufällig auszuwählen und die Beste davon zu übernehmen. Jede Generation ist hierbei gleichberechtigt, da in keiner Generation Informationen der vorherigen Generationen genutzt werden. Dies bedeutet, daß die Wahrscheinlichkeitsverteilung der Lösungen in jeder Generation gleich ist und damit *kein* Optimierfortschritt zwischen je zwei Generationen zu beobachten sein kann.

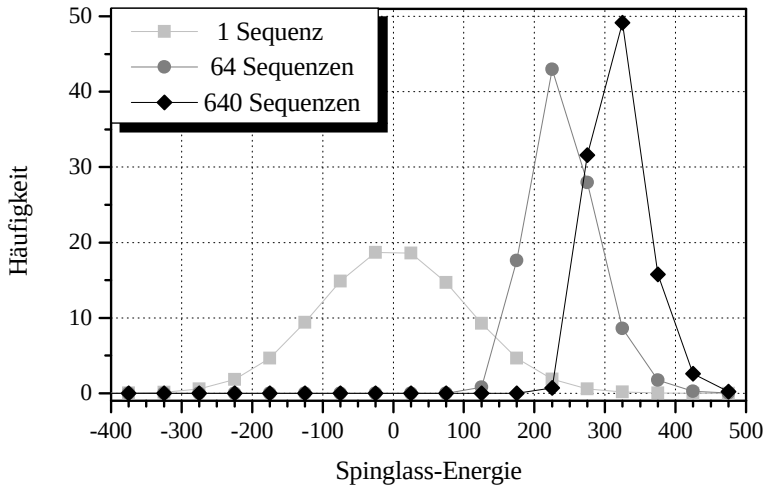

Abbildung 5.20: *Häufigkeitsverteilung der höchsten Spinglas–Energie bei* $(n = 1, 64, 640)$ *zufällig gewählten Spins der Länge 50.*

Für die hier ausgewählte Wechselwirkungsmatrix wurde die Wahrscheinlichkeitsverteilung der Lösungen durch Simulation bestimmt. Für drei verschiedene n wurden jeweils insgesamt 10^7 zufällige Spin–Sequenzen der Länge 50 ausgewählt. Damit ergeben sich für jedes n genau $\frac{10^7}{n}$ beste Lösungen, deren Verteilung ermittelt wurde. Der Bereich der Spinglas–Energie ist dabei in Abschnitte der Breite 50 unterteilt.

In Abbildung 5.20 ist die Häufigkeitsverteilung der besten Lösungen für $n = 1, 64, 640$ dargestellt.

Der Fall $n = 1$ entspricht der Verteilung der Spinglas–Energien der vollständigen Fitnesslandschaft. Die Spinglas–Energien sind normalverteilt mit dem Mittelwert 0 und der Standardabweichung $\sigma = 103.975$. Wenn eine Kette an 50 Spins ausgewürfelt wird, so liegt hier ihre Wechselwirkungs–Energie mit 68.27% Wahrscheinlichkeit im Intervall $[-\sigma; +\sigma]$ bzw. mit 95.45% Wahrscheinlichkeit im Intervall $[-2\sigma; +2\sigma]$. Aus Abbildung 5.20 läßt sich entnehmen, daß die Wahrscheinlichkeit einer Sequenz mit einer Energie größer als 300 $p_{E\geq 300} = 0.17556\%$ beträgt. Für die Energien 350 und 400 gilt $p_{E\geq 350} = 0.032\%$ und $p_{E\geq 400} = 0.00439\%$.

Prinzipiell läßt sich aus der Verteilung mit $n = 1$ jede Verteilung mit $n > 1$ berechnen. Hier jedoch wurden alle Verteilungen der Einfachheit halber numerisch bestimmt.

Pro Generation wählt hier der TST–Algorithmus 64 neue Sequenzen aus. Die Häufigkeitsverteilung der besten Sequenz einer einzelnen Generation, wenn die Sequenzen zufällig ausgewählt worden wären, ist in Abbildung 5.20 durch die Kurve für $n = 64$ dargestellt. Mit einer Wahrscheinlichkeit von 97.16% liegt die Spinglas–Energie im Intervall $[+150; +350]$. Immerhin noch 36.58% der Energien liegen im Bereich $[+250; +350]$. Die Wahrscheinlichkeit, daß sich unter den 64 zufälligen Sequenzen mindestens eine Sequenz mit einer Energie größer als 300 befindet, beträgt $1 - (1 - p_{E\geq 300})^{64} \approx 10.63\%$.

Während der 10 Generationen eines Laufs wählt der TST–Algorithmus 640 Sequenzen aus. Abbildung 5.20 zeigt die Häufigkeitsverteilung der jeweils besten Sequenz durch die Kurve für $n = 640$, wenn diese zufällig ausgewürfelt worden wären. Die Spinglas–Energie liegt mit 96.51% Wahrscheinlichkeit im Intervall $[+250; +400]$. 15.76% aller besten Sequenzen fallen in den Bereich $[+350; +400]$. Das Maximum der Häufigkeitsverteilung liegt bei einer Energie von 325, bzw. 49.16% aller besten Sequenzen befinden sich im Energieintervall $[+300; +350]$. Die Wahrscheinlichkeit, daß sich unter 640 zufälligen Sequenzen mindestens eine Sequenz mit einer Energie größer als 400 befindet, beträgt $1 - (1 - p_{E\geq 400})^{640} \approx 2.77\%$.

Abbildung 5.20 zeigt also die Untergrenze der Performance eines jeden Optimierverfahrens auf, die nicht unterschritten werden kann, da es sonst schlechter als das eingangs erwähnte Zufallsverfahren sein müßte. Jedes Optimierverfahren kann pro Generation (64 Sequenzen) mit besten Sequenzen zwischen 150 und 300 Spinglas–Energie rechnen. Nach 10 Generationen (d.h. 640 Sequenzen) liegen die besten Sequenzen mit großer Wahrscheinlichkeit zwischen 250 und 400. Das bedeutet, daß ein Optimierverfahren, das für sich beansprucht, besser als der Zufall zu sein, deutlich über diesen Werten liegen muß.

5.6.2 Einfluß von Fehlern bei der Fitnessbestimmung

In allen bisherigen Test-Beispielen war die Berechnung der Fitness fehlerfrei. Der TST–Algorithmus soll in Zukunft auch iterativ mit der *experimentellen* Bestimmung der Fitness der vorgeschlagenen Sequenzen eingesetzt werden. Da es keinen Meßwert ohne Fehler gibt, soll hier die Empfindlichkeit des TST–Algorithmus gegenüber statistischen Fehler bei der Fitnessbestimmung studiert werden.

Nach dem zentralen Grenzwertsatz liegen unabhängige Messungen GAUSS-verteilt um den „wahren“ Erwartungswert. Der Fehler der Fitnessberechnung wird deshalb als normalverteilt mit Mittelwert 0 und Standardabweichung $2 \cdot 736.436 \cdot \frac{\sigma}{100}$ angenommen. Es wird hierbei davon ausgegangen, daß die minimale Spin–Energie gleich der negativen maximalen Energie ist. σ wurde in unterschiedlichen Schrittweiten zwischen 0% und 200% durchvariiert. Zur Erhöhung der statistischen Signifikanz wurden für jedes σ fünf Simulationen mit gleicher Startpopulation aber unterschiedlichem Startwert des Zufallsgenerators durchgeführt.

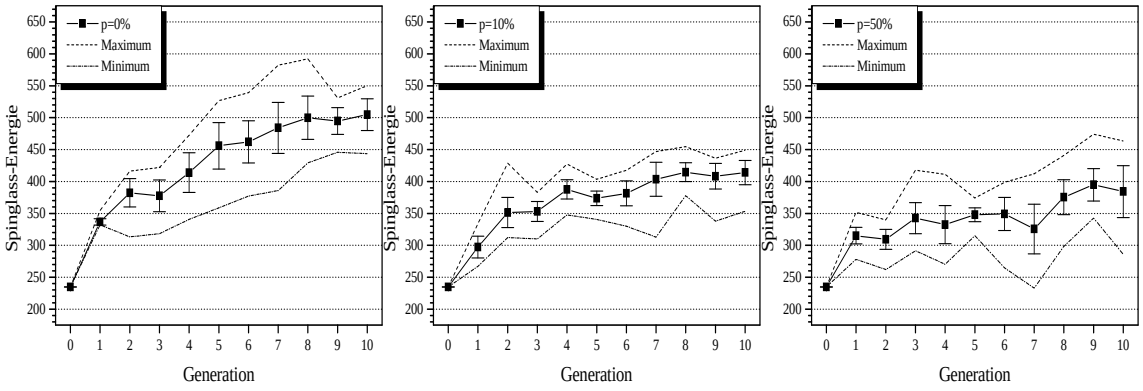

Abbildung 5.21: *Performance des TST–Algorithmus für unterschiedliche Standardabweichungen* ($\sigma = 0\%, 10\%, 50\%$) *des Fehlers bei der Fitnessberechnung*

Für $\sigma = 1\%$ liegen 95.45% aller Werte des Rauschterm im Bereich ± 29.46. Bei einem Fehler von $\sigma = 10\%$ und $\sigma = 50\%$ erhöht sich der Bereich des Rauschterms auf ± 294.57 bzw. ± 1472.87. Die Wechselwirkungs–Energien jedes Mitglieds der zufällig gewählten Startpopulation werden nach Abbildung 5.20 mit 95.45% Wahrscheinlichkeit im Intervall $[-208; +208]$ liegen. Dies bedeutet, daß die eigentlichen Fehler bei der Fitnessberechnung größer sind, als der Wert von σ suggeriert, da (vor allem in den ersten Generationen) nicht der vollständige Energiebereich $[-736.436; +736.436]$ ausgenutzt wird.

In Abbildung 5.21 ist die Performance des TST–Algorithmus für drei verschiedene Standardabweichungen (0%, 10%, 50%) des Fehlers bei der Fitnessberechnung

dargestellt. Ausgewertet wurden hier in jeder Generation nur die Sequenz mit der jeweils höchsten Fitness. Jedes Diagramm zeigt zum einen den Mittelwert der Fitness der jeweils besten Sequenz jeder Generation der fünf Programmläufe zusammen mit der Standardabweichung. Zum anderen sind noch die beste und die schlechteste Sequenz der jeweils besten Sequenz jeder Generation der fünf Programmläufe eingezeichnet.

Abbildung 5.21 zeigt, daß der TST–Algorithmus bei $\sigma = 10\%$ bereits deutlich langsamer zu Optima mit niedrigerer Fitness (≈ 420) findet, als ohne Rauschterm (≈ 500). Bei einem Rauschen von 50% ist nur noch langsames Optimieren festzustellen, das im Mittel bei Optima der Fitness ≈ 390 endet. Außerdem erhöht sich hier die Streuung der Ergebnisse der einzelnen Simulationsläufe.

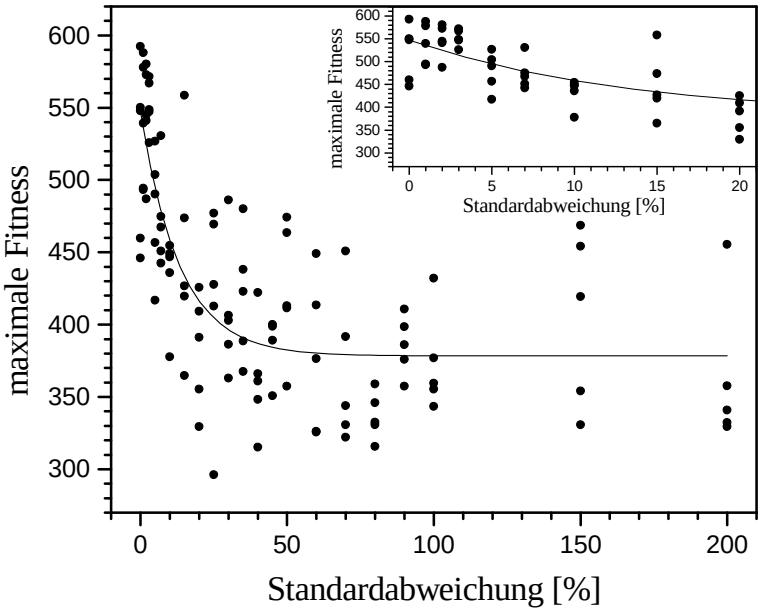

Abbildung 5.22: *Die maximale Fitness jedes Laufs in Abhängigkeit vom Fehler*

Die Leistungsfähigkeit des TST–Algorithmus in Abhängigkeit von der Stärke des Rauschens ist in Abbildung 5.22 detaillierter dargestellt. Die höchste erreichte Fitness eines jeden Simulationslaufs ist über die Standardabweichung σ des Rauschterms aufgetragen. Jedem Lauf entspricht damit ein eingezeichneter Punkt. Die eingepaßte Kurve hat die Form $F_0 + F_1 \exp\left(-\frac{\sigma}{c}\right)$, mit $F_0 := 378.366$, $F_1 := 168.799$, $c := 13.450$. Für steigende Werte von σ (Einschub) nimmt die Performance immer mehr ab, bis sie sich (für $\sigma > 50$) dem Verhalten eines Zufallsverfahrens (siehe Kapitel 5.6.1) annähert. Dies zeigt die gefittete Kurve, die für $\sigma \to +\infty$ gegen F_0 konvergiert. F_0 liegt etwas über der Lage des Maximums (325) der Häufigkeitsverteilung des Zufallsverfahrens für $n = 640$ in Abbildung 5.20.

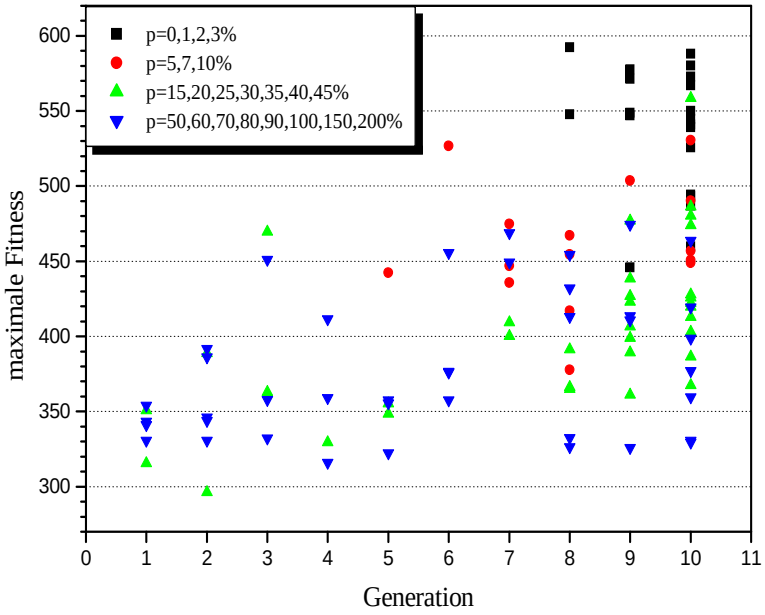

Abbildung 5.23: *Die maximale Fitness jedes Laufs in Abhängigkeit von der Generation, in der sie erreicht wurde*

Mit steigendem Fehler verändert sich das Optimierverhalten des TST–Algorithmus. Da es wenig sinnvoll ist, jeden einzelnen Lauf als eigenes Diagramm darzustellen, andererseits durch das Mitteln mehrerer Läufe die Charakteristika verwischt werden (Abbildung 5.21), wurde folgende Auftragung gewählt: In Abbildung 5.23 ist die höchste erreichte Fitness eines jeden Simulationslaufs über die Generation, in der sie erreicht wurde, aufgetragen. Die Punkte wurden entsprechend ihrer Standardabweichung σ eingefärbt. Wie in Abbildung 5.22 ist auch hier zu erkennen, daß mit steigendem Fehler die maximal erreichte Fitness abnimmt. Die maximale Fitness wird zu einem früherem Zeitpunkt erreicht, als bei niedrigem Fehler und umgedreht. Allerdings existiert hier bei steigendem Fehler eine immer größere Schwankungsbreite, beispielsweise kann bei hohem Fehler ($\sigma > 10\%$) die maximale Fitness bereits in der ersten, aber auch erst in der letzten Generation erreicht werden. Dies ist ein weiteres Indiz dafür, daß für hohe Fehler ($\sigma > 50$) der TST–Algorithmus an Bedeutung verliert und durch Zufall ausgewählte beste Sequenzen dominieren.

In den Abbildungen 5.24 und 5.25 werden jeweils ein einzelner Simulationslauf des TST–Algorithmus mit einer Standardabweichung von $\sigma = 10\%$ bzw. $\sigma = 50\%$ dargestellt. In den linken Diagrammen sind jeweils die Fitnesswerte der in jeder Generation neu ausgewählten Sequenzen abgebildet. Unterschieden werden dabei die, durch Suche in der durch das GRNN approximierten Landschaft gefundenen Sequenzen und die durch Rekombination und Mutation gebildeten Sequenzen. Bei

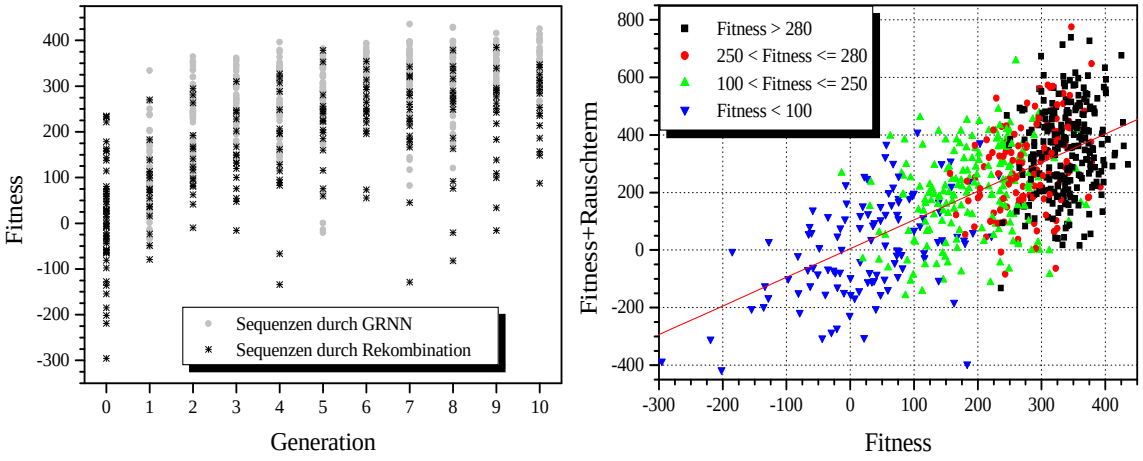

Abbildung 5.24: *Einzelner Lauf bei einem Rauschanteil von* $\sigma = 10\%$

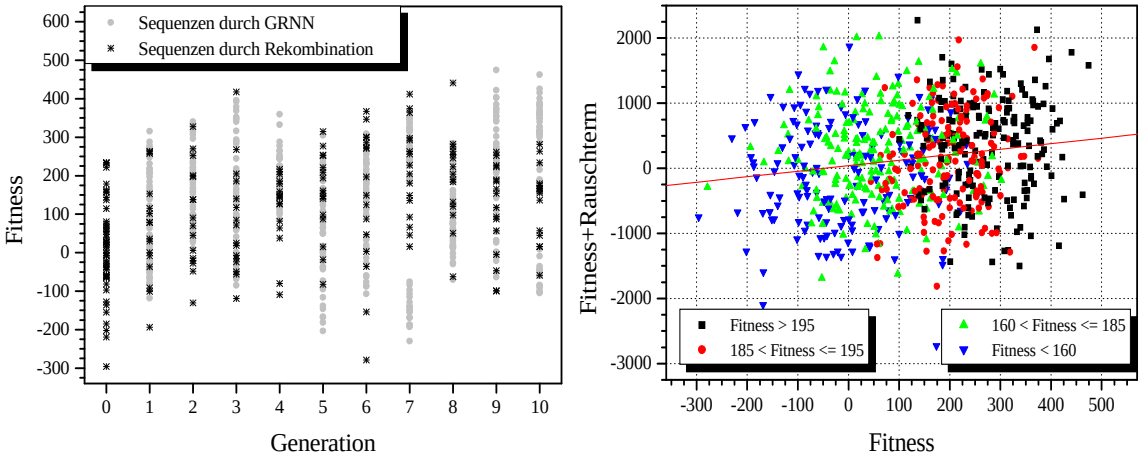

Abbildung 5.25: *Einzelner Lauf bei einem Rauschanteil von* $\sigma = 50\%$

einem Fehler von 10% ist die Fitness der durch Rekombination generierten Sequenzen durchweg niedriger, als die der durch das GRNN gefilterten. Dies gilt für die meisten Generationen auch bei einem Fehler von 50%. Allerdings ist da die Schwankung größer; das GRNN hat bei einem Fehler dieser Größe bei einigen Generationen (5–8) Schwierigkeit mit der Datenapproximation.

In den rechten Diagrammen der Abbildungen 5.24 und 5.25 ist die verrauschte Fitness, wie sie das GRNN präsentiert bekommt, über der „echten“ Fitness aufgetragen. Es kommen dabei jeweils alle 691 Sequenzen des vollständigen Laufs zum Einsatz. Bei einem Fehler von 10% besteht eine deutlich positive Korrelation zwischen verrauschter Fitness und „echter“ Fitness, wie die Fitgerade belegt. Im Gegensatz dazu sind, bei einem Fehler von 50%, beide Werte nur noch schwach unkorreliert.

In beiden Diagrammen sind die Fitnesswerte, wie sie das in der 10.Generation trainierte GRNN zurückliefert, in jeweils vier Wertebereiche aufgeteilt. Jedem

Wertebereich entspricht eine Farbe. Für beide Fehler (10%, 50%) ist das GRNN offensichtlich bei der Mehrzahl der Sequenzen nicht mehr in der Lage, den korrekten Fitnesswert anzugeben. Dennoch kann es — selbst für einen großen Fehler von $\sigma = 50\%$ — Sequenzen mit hohen Fitnesswerten von Sequenzen niedriger Fitness unterscheiden, wie die zusammenhängenden Punkte gleicher Farbe zeigen. Insbesondere sind die meisten Sequenzen mit der höchsten „echten“ Fitness auch durch das GRNN hoch bewertet, obwohl die Rauschterme zum Teil ein Vielfaches der eigentlichen Fitness betragen. Bedingt durch diese enorme Stabilität des GRNN kann der TST–Algorithmus auch bei sehr hohen statistischen Fehlern noch optimieren.

Der TST–Algorithmus erweist sich als bemerkenswert robust gegenüber *statistischen* Fehlern. Dies ist eine wichtige Voraussetzung für den praktischen Einsatz. Das Auftreten systematischer Fehler wurde hier nicht untersucht. Es ist allerdings anzunehmen, daß der TST–Algorithmus auf solche empfindlicher als auf statistische reagieren wird.

5.6.3 Zufällige Sequenzen

Hier wurde der Einfluß eines Anteils a zufälliger Sequenzen an den (hier: 64) neu ausgewählten Sequenzen jeder Generation durch den TST–Algorithmus untersucht. Die Absicht dabei ist, einem eventuellen Bias in der Auswahl der Startpopulation entgegenzuwirken sowie das vorzeitige Konvergieren in lokale Optima zu erschweren.

Es gibt zwei Möglichkeiten der Umsetzung: Entweder werden die zufälligen Sequenzen *zusätzlich* zu den ausgewählten addiert oder sie *ersetzen* einen entsprechenden Anteil der ausgewählten Sequenzen. Wenn man davon ausgeht, daß beim experimentellen Einsatz eine genau vorgegebene maximale Anzahl an Stichproben in jeder Generation bewertet werden kann, entspricht dies der zweiten Variante.

Wegen der einfacheren Implementation werden bei den hier durchgeführten Simulationen die durch Rekombination generierten Sequenzen durch zufällige ersetzt. Zur Erhöhung der statistischen Genauigkeit wurden für jeden Anteil a fünf Simulationen mit gleicher Startpopulation aber unterschiedlichem Startwert des Zufallsgenerators durchgeführt.

In Abbildung 5.26 ist die Performance des TST–Algorithmus für drei verschiedene Anteile ($a = 0\%, 20\%, 90\%$) an zufälligen Sequenzen dargestellt. Die Art der Darstellung ist identisch mit Abbildung 5.21 auf Seite 129. Auffällig ist die Ähnlichkeit der Kurven zwischen den drei Diagrammen. Der Anteil an zufälligen Sequenzen scheint die Performance des TST–Algorithmus kaum zu beeinflussen.

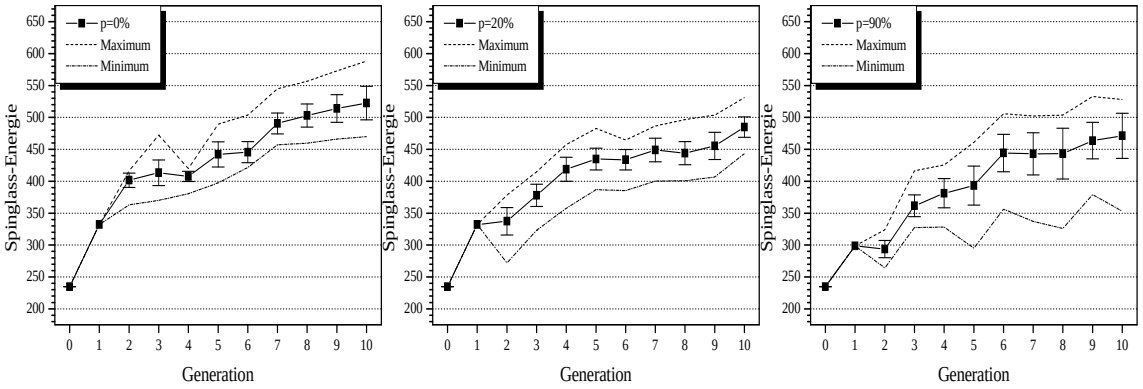

Abbildung 5.26: *Performance des TST–Algorithmus für unterschiedliche Anteile an zufälligen Sequenzen* ($a = 0\%, 20\%, 90\%$)

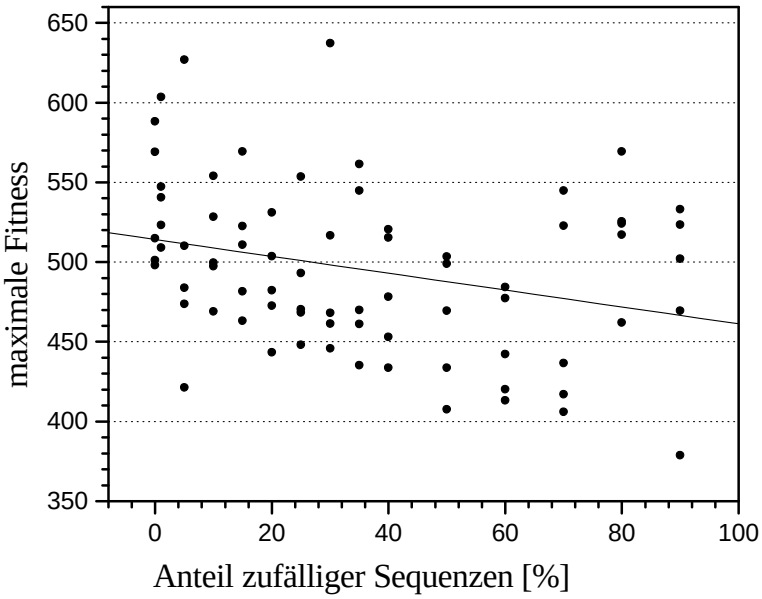

Abbildung 5.27: *Die maximale Fitness jedes Laufs in Abhängigkeit vom Anteil an zufälligen Sequenzen*

Mit steigendem Anteil a sinkt die maximale erreichte Fitness und es erhöht sich die Streuung der Ergebnisse der einzelnen Simulationsläufe. Diese Aussage wird in Abbildung 5.27 weiter bestätigt. Abbildung 5.27 ist das Pendent zu Abbildung 5.22 auf Seite 130. Die höchste erreichte Fitness eines jeden Simulationslaufs ist über dem Anteil a an zufälligen Sequenzen aufgetragen. Die einzelnen Meßwerte sowie die Trendgerade zeigen einen Abfall der mittleren, maximalen Fitness in Abhängigkeit von a. Selbst für sehr hohe Anteile an zufälligen Sequenzen zeigt der TST–Algorithmus ein deutliches Optimierverhalten und ist immer besser als ein reines Zufallsverfahrens (siehe Kapitel 5.6.1).

In Abbildung 5.28 ist zu beobachten, daß selbst im Extremfall $a = 90\%$ in allen

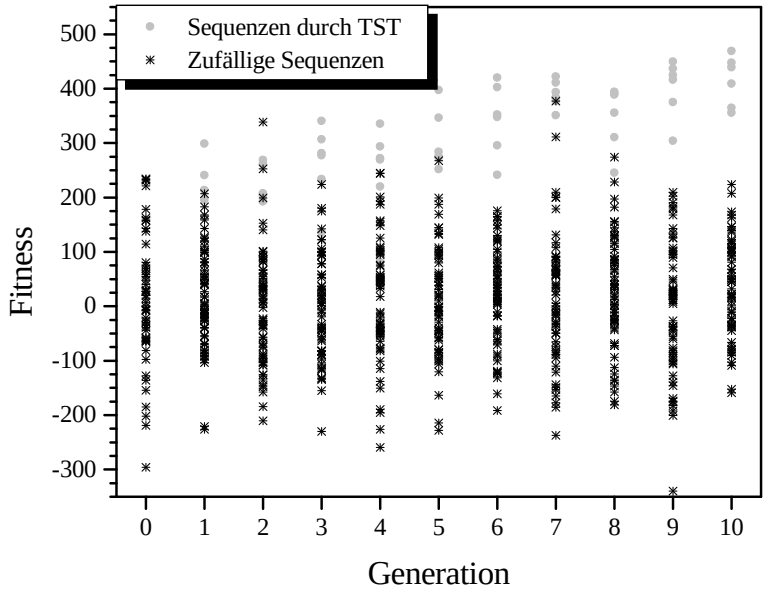

Abbildung 5.28: *Einzelner Lauf bei einem Anteil von 90% zufälligen Sequenzen*

Generationen die Fitness der durch das GRNN bewerteten und vom TST–Algorithmus ausgewählten Sequenzen deutlich über der der zufälligen Sequenzen liegt — offensichtlich optimiert der TST–Algorithmus. Dies ist umso erstaunlicher, als dem TST–Algorithmus bei diesem hohen Anteil nur noch sechs Sequenzen von 64 zur Suche in der durch das GRNN approximierten Landschaft zur Verfügung stehen. Dies entspricht gerade der Anzahl an unterschiedlichen Präzisionen des GRNN. Der TST–Algorithmus kann deshalb in diesem Fall nur die globalen Optima (für unterschiedliche Präzisionen des GRNN) durch Suche mit einem genetischen Algorithmus ermitteln; die lokalen Optima müssen unabgetastet bleiben.

Der TST–Algorithmus zeigt eine verhältnismäßig geringe Beeinflussung durch das Hinzufügen zufälliger Sequenzen. Je höher der Anteil a ist, desto weniger lokale Optima können ausgetestet werden. Deshalb sinkt die Performance mit steigendem a. Andererseits liefern die zufälligen Sequenzen dem TST–Algorithmus zusätzliche Informationen über noch unbekannte Bereiche der Fitnesslandschaft. Beide Effekte arbeiten gegenläufig — dies ist der Grund, warum die Performance mit steigendem a so langsam sinkt.

Die eingangs aufgestellte Vermutung, daß ein (geringer) Anteil an zufälligen Sequenzen dem TST–Algorithmus bei der Überwindung lokaler Optima von Nutzen sein könnte, bestätigte sich (leider) nicht. Andernfalls hätte die Performance in Abbildung 5.27 ein Maximum für ein $a > 0$ aufweisen müssen. Möglicherweise ist dieser Effekt aber für andere Testlandschaften zu beobachten.

5.7 Arbeiten anderer Autoren

Zunächst soll hier betont werden, daß sich die erste schriftliche Erwähnung des allgemeinen TST–Zyklus (Iterativ: statistische Analyse mit neuronalem Netz, Suche in der approximierten Landschaft mit genetischem Algorithmus, experimentelle Bewertung der vorgeschlagenen Substanzen) von 1993 in einer Patentanmeldung von Andreas Schober *et al.* befindet [95]. Alle dem Autor bekannten Arbeiten, die eine Ähnlichkeit mit dem hier vorgestellten TST–Algorithmus aufweisen, wurden später veröffentlicht.

Gisbert Schneider und Paul Wrede von der Freien Universität Berlin veröffentlichten 1994 die erste [93] einer ganzen Reihe an Arbeiten [94], in denen mit einer Variante des TST–Zyklus kurze Peptid–Sequenzen auf eine bestimmte Aufgabe hin optimiert werden. Der „simulierte molekulare Evolution“ (SME) genannte Zyklus unterscheidet sich vom TST–Algorithmus durch einige wesentliche Details:

- Als statistisches Verfahren werden Backpropagation–Netze eingesetzt, mit allen bereits dargestellten Nachteilen (Kapitel 5.2). Der Algorithmus kann damit nur Sequenzen konstanter Länge verarbeiten. Ein methodisches Problem ist außerdem, daß den Ausgabewerten des Netzes direkt eine biologische Funktion zugeordnet wird. Beispielsweise wird in [93] einer Netzausgabe ≥ 0.5 die Funktion „Sequenz hat cleavage site“ zugeordnet. Ein Wert kleiner 0.5 wird als negatives Beispiel interpretiert. Beim TST–Algorithmus besitzen die konkreten Ausgabewerte des GRNN keinerlei Bedeutung — es wird lediglich dazu benutzt, die Lage der Optima in der Fitnesslandschaft zu identifizieren.

- Die Peptide werden nicht direkt als Buchstabensequenzen kodiert, sondern durch die entsprechenden Werte für Hydrophobizität und Volumen der Seitenkette jeder Aminosäure. Dies muß nicht unbedingt ein Nachteil sein, widerspricht aber dem Ansatz des TST–Algorithmus, mit einem Minimum an problemspezifischer Information auszukommen.

- Die Suche innerhalb der durch das Netz approximierten Fitnesslandschaft erfolgt nicht, wie beim TST–Algorithmus durch einen genetischen Algorithmus und lokale Suchverfahren, sondern durch Evolutionsstrategien (siehe Kapitel 2.3.2.2). In der hier eingesetzten Variante ähnelt es allerdings eher klassischem „hill–climbing“.

- Der TST–Algorithmus führt die Schritte „Statistische Analyse“, „Vorschlag neuer Sequenzen“ und „Experimentelle Bewertung“ iterativ über mehrere Generationen aus. Der SME wurde nach Kenntnis des Autors immer nur eine einzige Runde angewandt und nicht zur schrittweisen Optimierung von Peptiden eingesetzt.

Im Gegensatz von TST–Algorithmus ist der SME auf die Optimierung von Peptiden spezialisiert. Das Verfahren von SCHNEIDER und WREDE hat allerdings den wichtigen Vorteil, daß es seine Leistungsfähigkeit bereits an praktischen Beispielen unter Beweis stellen konnte. FRANK DARIUS *et al.* stellten in [14] eine Reihe an Kritikpunkten am SME zusammen.

Die Gruppe um VENKAT VENKATASUBRAMANIAN von der Purdue University beschäftigt sich mit dem Design von Polymeren. VENKATASUBRAMANIAN *et al.* schlugen 1996 in [123] einen TST–ähnlichen Zyklus vor. Sie zeigten, daß mit Backpropagation–Netzen eine bessere numerische Vorhersage der Struktur-Aktivitäts–Beziehung von Polymeren als mit klassischen statistischen Methoden zu erreichen ist. Die Polymere wurden durch eine feste Anzahl chemischer Deskriptoren codiert. Das inverse Problem lösten sie durch einen genetischen Algorithmus [122]. Bis dato erschien allerdings keine Veröffentlichung mehr, in der der vorgeschlagene Zyklus aus [123] tatsächlich umgesetzt worden wäre.

Von LUTZ WEBER *et al.* [126, 127] und JASBIR SINGH *et al.* [108] stammen fast zeitgleich zwei vergleichbare Arbeiten, in denen Moleküle durch einen in die Praxis umgesetzten genetischen Algorithmus optimiert werden. JASBIR SINGH *et al.* optimierten Hexapeptide als Stromelysin-selektive Substrate, während LUTZ WEBER *et al.* [126, 127] eine Vier-Komponenten-UGI–Reaktion als Thrombin-Inhibitor optimierten. Aufgrund der Ähnlichkeit des Ansatzes soll hier nur die Arbeit von LUTZ WEBER *et al.* skizziert werden:
Zwei der vier Komponenten der UGI–Reaktion enthielten je 10 Reagentien, die beiden verbleibenden je 40 — dadurch sind insgesamt 160000 unterschiedliche UGI–Reaktionen möglich. Jede der Reagentien jeder Komponente wurde durch ein eineindeutiges Bitmuster codiert. Durch Zusammensetzen der Bitmuster jeder Komponente ergibt sich ein Bitmuster konstanter Länge, das eine eindeutige UGI–Reaktion repräsentiert. Diese Bitmuster dienen einem einfachen genetischen Algorithmus als Spielmaterial, bei dem die Fitness durch die experimentelle Umsetzung der jeweils codierten Reaktion und der entsprechenden Inhibitor-Effizienz bestimmt wird. Das Verfahren beginnt mit 20 zufälligen Bitmustern, deren Fitness experimentell ermit-

telt wird. Durch den genetische Algorithmus werden daraus durch Mutation und Rekombination 20 neue Bitmuster errechnet, deren Fitness ebenfalls bestimmt wird. Aus den 40 Sequenzen werden die 20 besten ausgewählt, die dann als Eltern der nächsten Generation dienen. Nach 20 Generationen wurde das Verfahren beendet; mit nur 400 Stichproben aus 160000 möglichen UGI–Reaktionen verbesserte sich die Inhibitoraktivität um etwa drei Größenordnungen.

5.8 Zusammenfassung

Die vorgestellten Beispiele (Kapitel 5.5) zeigen, daß der TST–Algorithmus prinzipiell in der Lage ist, mit einer niedrigen Anzahl an Stichproben erfolgreich auf unbekannten Fitnesslandschaften zu optimieren. Er benutzt dabei keinerlei zusätzliches Expertenwissen über die speziellen Eigenschaften der Fitnesslandschaften.

Durch den Einsatz des GRNN ergibt sich für den TST–Algorithmus ein großes Spektrum an Optimieraufgaben, da keine speziellen Anforderungen an die Datenstruktur gegeben sind, solange eine Metrik existiert. Das Einstellen der Präzision des GRNN erlaubt in Zusammenarbeit mit den eingesetzten Suchverfahren ein feines und gezieltes Abtasten der lokalen Optima und des globalen Optimums der approximierten Fitnesslandschaft.

Der TST–Algorithmus weist folgende Charakteristika auf:

- Der TST–Algorithmus braucht die Information „schlechter“ Sequenzen, da sonst das GRNN keine Approximation der Fitnesslandschaft mit lokalen Optima bilden kann. Die ausschließliche Nutzung bester Sequenzen würde günstigstenfalls zu einer lokalen Suche in einer engen Umgebung der besten Sequenz der Startpopulation führen.

- Selten stimmen die Lagen der approximierten lokalen Optima des GRNN exakt mit den Lagen der lokalen Optima der Fitnesslandschaft überein. Deshalb ist es wichtig, nicht nur die jeweils lokal beste Sequenz zu übernehmen, sondern eine gewisse Anzahl an schlechteren zur präziseren Lokalisierung der lokalen Optima.

- Große Sprünge durch den Suchraum während einer Generation sind möglich und entsprechen dem Übergang zwischen lokalen Optima.

Es gibt in der Literatur vergleichbare Ansätze [93, 123], von denen allerdings nur die Arbeiten von SCHNEIDER und WREDE ausgereift sind. Keiner der Ansätze ist so allgemein formuliert wie der TST–Algorithmus, sondern jeweils an eine spezielle Aufgabenstellung angepaßt.

In den Arbeitsgruppen von LUTZ WEBER *et al.* [126, 127] und JASBIR SINGH *et al.* [108] wurde unabhängig voneinander gezeigt, daß bereits mit sehr einfach aufgebauten genetischen Algorithmen mit geringer Stichprobengröße verblüffend gute Moleküle in jeder Generation vorgeschlagen werden können. Dies läßt darauf schließen, daß — zumindest bei den ausgewählten Beispielen — die zugrundeliegenden Fitnesslandschaften eher „einfach“, d.h. nicht „rugged“ strukturiert sind. Konstruktionsbedingt verhält sich der TST–Algorithmus im ungünstigsten Fall wie ein genetischer Algorithmus, da ein Teil der neuen Sequenzen durch Rekombination und Mutation generiert wird (siehe Seite 104). Der ungünstigste Fall tritt ein, wenn das GRNN keinerlei statistische Eigenschaften aus den bisherigen Daten extrahieren kann und die Suche in der approximierten Landschaft lediglich zufällige Sequenzen liefert. Es ist deshalb zu erwarten, daß der Einsatz des TST–Algorithmus in Vergleich mit den experimentellen genetischen Algorithmen aus [108, 127] einen Performancesprung zur Folge haben müßte.

5.9 Ausblick

Ein Hauptproblem der vorliegenden Implementation des TST–Algorithmus ist, daß das GRNN eine deutliche Tendenz zum Überfitten aufweist, wie die Diagramme 5.1–5.3 des eindimensionalen Beispiels (Kapitel 5.5.1) zeigen. Dieser Effekt tritt vor allem bei ungleichmäßig verteilten Trainingspunkten auf, wie sie der TST–Algorithmus typischerweise generiert. Hier besteht noch Bedarf an Weiterentwicklung und Verbesserung.

Wie in Kapitel 5.3.1 bereits ausgeführt, kommt für den Einsatz im TST–Algorithmus eine Fülle an statistischen Verfahren in Frage. Es ist möglich, daß bei unterschiedlichen Aufgabenstellungen unterschiedliche statistische Verfahren von Vorteil sind. Dies könnte Gegenstand einer ausführlichen Studie sein, in der die charakteristischen Eigenschaften verschiedener statistischer Verfahren im Zusammenspiel mit dem TST–Algorithmus untersucht werden.

Der TST–Algorithmus wird in Zukunft auf variable Sequenzlängen erweitert werden. Damit muß an Stelle der EUCLID'schen eine an das jeweilige Problem angepaßte

Metrik eingeführt werden.

Das Hauptziel ist indes die Adaption des TST–Algorithmus an organische Moleküle. Dies setzt einige wesentliche Entwicklungen voraus:

- Codierung organischer Moleküle
- Definition geeigneter Operatoren wie Mutation und Rekombination in Abhängigkeit von der gewählten Codierung. Konstruktion eines lokalen Optimierverfahrens aus diesen Operatoren.
- Implementation geeigneter Metriken zwischen organischen Molekülen, die ein Minimum an modellbehafteter, chemischer Information einsetzen.

Kapitel 6

Ausblick

Zusätzlich zu den Zusammenfassungen und Erweiterungsvorschlägen der einzelnen Themen dieser Arbeit wie dem Doping–Algorithmus (Kapitel 3.10, Kapitel 3.11), dem General Regression Neural Network (Kapitel 4.11, Kapitel 4.12) und dem TST–Algorithmus (Kapitel 5.8, Kapitel 5.9) sollen hier noch einige eher visionäre Anwendungen der beschriebenen Verfahren andiskutiert werden.

Doping–Algorithmus

Falls sich der Experimentator für die *präzise* Umsetzung der berechneten Nukleotid-Syntheseschemata interessiert, ist dies, je nach Anzahl benötigter Synthesetöpfe mit zum Teil enormem experimentellen Aufwand verbunden. Dies legt die Entwicklung einer *automatisierten Misch- und Synthesestation* nahe. An der hypothetischen Maschine gibt der Experimentator die Doping–Schemata in das Benutzerprogramm ein, das für jede Aminosäureposition den Doping–Algorithmus nach Kapitel 3 aufruft. Die berechneten Doping–Schemata werden vom Syntheseroboter nach der „split-and-mix"–Methode umgesetzt. Als Resultat liefert die Maschine eine Bibliothek an DNS–Strängen mit den gewünschten Doping–Schemata.

TST–Algorithmus

Für den TST–Algorithmus bieten sich in Zukunft mehrere Einsatzmöglichkeiten an:

Unterstützung bei der **Parameteroptimierung** (z.B. chemische Konzentrationen und Reaktionsbedingungen), falls die Parameterbewertung mit hohem Aufwand verbunden ist.

Denkbar ist bei einer **Kombination des TST–Algorithmus mit dem Doping–Algorithmus** folgender Zyklus: Beginnend mit einem Set an bekannten Peptid-Fitness–Paaren schlägt der TST–Algorithmus neue Peptide vor. Diese werden nicht

direkt synthetisiert, sondern zur Berechnung von Doping–Schemata benutzt. Mit Hilfe der Doping–Schemata wird eine Bibliothek erzeugt, die beispielsweise durch Phage Display experimentell bewertet wird. Deren beste Peptide werden sequenziert und dienen zusammen mit den Fitnesswerten der nächsten Generation des TST–Algorithmus als Eingabe. Dieses Verfahren ist eine Verbesserung von „Recursive Ensemble Mutagenesis“ (REM) nach DOUGLAS YOUVAN [22].

Ein wichtiger Bereich ist der iterative Einsatz des TST–Algorithmus in Verbindung mit der Synthese- und Screening–Apparatur unserer Arbeitsgruppe. Damit soll zum einen die Optimierung sequenziell codierbarer Moleküle wie Peptide oder RNS–Moleküle, zum anderen die **Optimierung organischer Moleküle** unterstützt werden. Hier bieten sich folgende Strategien an:

1. Analog zum bisherigen Verfahren schlägt der TST–Algorithmus völlig neue Moleküle vor. Die vorgeschlagenen Moleküle müssen allen chemischen Bildungsregeln genügen. Möglicherweise bieten die vorgeschlagenen Moleküle, selbst wenn sie schwierig zu synthetisieren sein sollten, dem Chemiker zusätzliche Strukturvorschläge. Es mag sinnvoll sein, mittels (heuristischer) Regeln schwer oder nicht synthetisierbare Moleküle auszuschließen.

2. Eine stärkere strukturelle Einschränkung kann durch die Festlegung auf kombinatorische Synthesestrategien erreicht werden. Der TST–Algorithmus schlägt nur Mitglieder aus einer vorgegebenen Menge an Reagentien für jede Komponente vor. Jedes Molekül ist damit automatisch synthetisierbar und bewertbar. Der TST–Algorithmus würde iterativ mit einer kleinen Auswahl aus der kombinatorischen Bibliothek optimieren, ohne daß diese vollständig dargestellt werden muß.

3. Nach dem Training des GRNN mit einem repräsentativen Trainingssatz an Molekül-Fitness–Paaren werden vorhandene Molekül-Datenbanken virtuell mit dem GRNN auf höherbewertete Moleküle abgesucht. Diese Substanzen werden experimentell bewertet und dem Trainingssatz hinzugefügt. Auch hier ist eine iterative Optimierung innerhalb des „Suchraums“ einer Molekül-Datenbank möglich.

Die skizzierten Visionen sind teilweise relativ leicht umsetzbar, teilweise erfordern sie einen größeren Entwicklungsaufwand. Insbesondere der Einsatz des TST–Algorithmus zur Moleküloptimierung wird eine der großen Herausforderungen der nächsten Jahre sein.

Anhang A

Ergebnisse der PROBEN1–Benchmarks

In diesem Kapitel sind die vollständigen Trainingsergebnisse mit den PROBEN1–Datensätzen tabellarisch aufgelistet. Bei allen Datensets wurden die Standard–PROBEN1–Benchmark–Regeln eingehalten, damit die Ergebnisse mit anderen Arbeiten vergleichbar sind. Bei der Vorbereitung der Daten für das GRNN wird nicht zwischen kontinuierlichen und diskreten Variablen unterschieden. Alle Daten werden als reelle Werte betrachtet und auf Mittelwert 0 und Standardabweichung 1 normiert.

Aus Gründen der Rechenzeit wurde bei einigen Datensätzen nur die 1.Permutation verwendet. Meist konnte dieser Datensatz auch nur mit der „schnellen“ und nicht der „präzisen“ Methode trainiert werden.

In allen Tabellen steht in der Spalte „Datensatz“ der PROBEN1–Name des Datensatzes; die Nummer gibt die Permutation der Daten an. Detaillierte Beschreibungen der einzelnen Datensätze finden sich in [83] samt Literaturangaben. Damit der Leser eine Vorstellung von dem Bereich der abgedeckten Probleme gewinnen kann, ist in Tabelle A.1 eine Kurzbeschreibung der Datensätze zusammengestellt.

„Netz–Typ“ bezeichnet die verwendete Netz–Architektur: Bei den PROBEN1–Netzen wird der Name des Netztyps aus [83] mit dem *minimalen Fehler des Testsets* aufgeführt; bei den GRNN der verwendete Trainingsalgorithmus. Der Trainingsfehler E_V wird in zwei Versionen der Gleichung (4.22) von Seite 70 berechnet.

LUTZ PRECHTELT schlug in [83] folgende Variante des Trainingsfehlers vor:

$$E_{V,Proben1}(p_1,\ldots,p_k) := 100 \cdot \frac{y_{max}-y_{min}}{S \cdot V} \cdot \sum_{s=1}^{S}\sum_{j=1}^{V}\left[R_s\left(\mathbf{X}_{v_{j,s}},p_1,\ldots,p_k\right) - Y_{v_{j,s}}\right]^2 \quad \text{(A.1)}$$

mit $y_{min},\ y_{max}$: minimaler und maximaler Wert von $\mathbf{Y}$

Datensatz	Daten-Typ	Beschreibung
`building`	`Regression`	Vorhersage des Energieverbrauchs eines Gebäudes
`cancer`	`Klassifikation`	Klassifikation von Brustkrebs als gut- oder bösartig, basierend auf mikroskopischen Zelleigenschaften
`card`	`Klassifikation`	Vorhersage, ob eine Bank einem Kunden eine Kreditkarte gewährt oder nicht
`diabetes`	`Klassifikation`	Diagnose, ob ein Pima Indianer Diabetes hat oder nicht, basierend auf seinen persönlichen medizinischen Daten
`flare`	`Regression`	Vorhersage der Anzahl von Sonnenflecken in drei Größen
`gene`	`Klassifikation`	Detektion von Intron/Exon–Grenzen von DNS–Sequenzen
`glass`	`Klassifikation`	Klassifikation von Glastypen z.B. Gebäude- oder Autofenster, Lampen, Flaschen etc. basierend auf der chemischen Analyse von Glassplittern
`heart`	`Klassifikation`	Vorhersage von Herzerkrankungen: Entscheidung, ob eine der vier Hauptgefäße um mehr als 50% im Durchmesser reduziert ist, basierend auf persönlichen (Alter, Rauchgewohnheiten) und medizinischen Daten
`heartc`	`Klassifikation`	Ein zuverlässigerer Ausschnitt der `heart`–Daten
`hearta`	`Regression`	Die analoge Version des `heart`–Datensatzes
`heartac`	`Regression`	Die analoge Version des `heartc`–Datensatzes
`horse`	`Klassifikation`	Vorhersage des Schicksal eines Pferdes nach einer Kolik: Überleben, Sterben oder Gnadentod, basierend auf einer tierärztlichen Untersuchung
`mushroom`	`Klassifikation`	Unterscheide eßbare von giftigen Pilzen, basierend auf äußerlichen Beschreibung der Pilze aus einem Pilzbuch
`soybean`	`Klassifikation`	Erkenne 19 verschiedene Krankheiten von Soyabohnen, basierend auf der äußerlichen Beschreibung der Bohne
`thyroid`	`Klassifikation`	Diagnostiziere Schilddrüsenfunktion: Normal, Unter- oder Überfunktion, basierend auf medizinischen Daten des Patienten

Tabelle A.1: *Tabellierte Kurzbeschreibung der* PROBEN1*–Benchmarks*

Seine Idee war vermutlich, den Trainingsfehler in Prozent des Wertebereichs anzugeben. Allein — das *Multiplizieren* mit dem maximalen Wertebereich kann nicht zum gewünschten Erfolg führen. Deshalb wurde folgende Definition ebenfalls implementiert:

$$E_{V,Dirk}(p_1,\ldots,p_k) := 100 \cdot \frac{1}{S \cdot V} \cdot \sum_{s=1}^{S} \sum_{j=1}^{V} \left[\frac{R_s\left(\mathbf{X}_{v_{j,s}}, p_1, \ldots, p_k\right) - Y_{v_{j,s}}}{y_{max} - y_{min}} \right]^2 \qquad \text{(A.2)}$$

Falls $y_{max} - y_{min} = 1$ — dann und nur dann — sind die Definitionen (A.1) und (A.2) äquivalent. Da nach den PROBEN1–Konventionen unterschiedliche Klassen mit 0 und 1 codiert sind, ergibt sich für Klassifizieraufgaben kein Unterschied bei der Berechnung des Trainingsfehlers. Anders bei Regressionsaufgaben: Bei Angabe der Regressionsfehler werden deshalb beide Werte im Format $E_{V,Dirk}/E_{V,Proben1}$ angegeben. Intern fand nur Gleichung (A.2) Einsatz. Dies muß in Spalte „E_V“ berücksichtigt werden. Übrigens bezeichnen manche andere Publikationen den Trainingsfehler E_V mit MSE für „mean squared error“.

Datensatz	Netz-Typ	E_V	Regressionsfehler Trainingset [%]	Klassifikationsfehler Trainingset [%]	Regressionsfehler Testset [%]	Klassifikationsfehler Testset [%]
building1	GRNN: schnell	0.456	0.069 / 0.010		2.382 / 0.400	
building1	GRNN: präzise	0.326	0.060 / 0.009		2.549 / 0.448	
building1	PROBEN1 (linear)		----- / 0.565		----- / 0.780	
cancer1	GRNN: schnell	2.975	1.586	2.286 (12/525)	2.383	2.874 (5/174)
cancer1	GRNN: präzise	2.609	1.736	2.667 (14/525)	1.194	1.724 (3/174)
cancer1	PROBEN1 (no shortcut)		2.360		1.320	1.380 (2.40/174)
cancer2	GRNN: schnell	2.615	0.817	0.952 (5/525)	2.823	3.448 (6/174)
cancer2	GRNN: präzise	1.957	0.592	0.952 (5/525)	3.171	4.023 (7/174)
cancer2	PROBEN1 (pivot)		1.925		3.400	4.520 (7.86/174)
cancer3	GRNN: schnell	2.468	0.721	0.571 (3/525)	3.540	4.598 (8/174)
cancer3	GRNN: präzise	2.024	0.402	0.190 (1/525)	2.394	3.448 (6/174)
cancer3	PROBEN1 (pivot)		2.295		2.570	3.370 (5.86/174)
card1	GRNN: schnell	16.976	5.759	3.282 (17/518)	15.939	19.186 (33/172)
card1	GRNN: präzise	13.398	3.192	2.703 (14/518)	15.150	19.767 (34/172)
card1	PROBEN1 (no shortcut)		8.775		10.350	14.050 (24.17/172)
card2	GRNN: schnell	16.660	5.623	1.737 (9/518)	18.698	22.093 (38/172)
card2	GRNN: präzise	12.694	3.778	2.510 (13/518)	17.690	26.744 (46/172)
card2	PROBEN1 (linear)		9.520		14.910	19.240 (33.09/172)
card3	GRNN: schnell	17.037	9.968	3.668 (19/518)	18.601	23.837 (41/172)
card3	GRNN: präzise	14.133	6.825	4.247 (22/518)	17.947	26.744 (46/172)
card3	PROBEN1 (linear)		8.930		12.670	14.420 (24.80/172)
diabetes1	GRNN: schnell	16.419	11.698	14.410 (83/576)	17.078	26.042 (50/192)
diabetes1	GRNN: präzise	15.098	11.357	15.625 (90/576)	16.513	25.000 (48/192)
diabetes1	PROBEN1 (no shortcut)		15.145		16.990	24.100 (46.27/192)
diabetes2	GRNN: schnell	16.503	12.555	14.410 (83/576)	16.570	22.396 (43/192)
diabetes2	GRNN: präzise	14.860	10.906	14.757 (85/576)	16.803	26.042 (50/192)
diabetes2	PROBEN1 (linear)		16.200		17.690	24.690 (47.40/192)
diabetes3	GRNN: schnell	16.151	11.342	13.368 (77/576)	16.645	22.917 (44/192)
diabetes3	GRNN: präzise	14.799	11.146	15.451 (89/576)	16.057	21.354 (41/192)
diabetes3	PROBEN1 (no shortcut)		15.705		16.480	22.590 (43.37/192)
flare1	GRNN: schnell	0.677	0.577 / 0.230		1.000 / 0.432	
flare1	GRNN: präzise	0.667	0.595 / 0.249		0.980 / 0.423	
flare1	PROBEN1 (linear)		----- / 0.355		----- / 0.520	
flare2	GRNN: schnell	0.573	0.517 / 0.409		0.276 / 0.254	
flare2	GRNN: präzise	0.561	0.511 / 0.402		0.290 / 0.267	
flare2	PROBEN1 (linear)		----- / 0.440		----- / 0.310	
flare3	GRNN: schnell	0.529	0.477 / 0.390		0.411 / 0.327	
flare3	GRNN: präzise	0.524	0.476 / 0.387		0.412 / 0.327	
flare3	PROBEN1 (linear)		----- / 0.425		----- / 0.350	
gene1	GRNN: schnell	9.520	0.014	0.042 (1/2382)	9.211	15.259 (121/793)
gene1	GRNN: präzise	8.537	0.014	0.042 (1/2382)	8.709	15.763 (125/793)
gene1	PROBEN1 (no shortcut)		5.445		8.660	16.670 (132.19/793)

Tabelle A.2: *Tabellierte Ergebnisse der* PROBEN1*–Benchmarks: Fehler von Trainings- und Testset — Teil I*

Als Klassifikationsfehler dient das Verhältnis $100 \cdot \frac{\text{Anzahl falsch klassifizierte Punkte}}{\text{Größe Testset}}$. Diese Zahlen sind zusätzlich in Klammern angegeben. Bei den PROBEN1–Netzen wird für die „Anzahl falsch klassifizierte Punkte“ der Mittelwert aus mehreren Trainingsläufen genommen.

Die Trainingszeiten hängen von der verwendeten Rechnerarchitektur ab. Für diese Studie wurden Pentium II–Rechner mit 300 MHz Taktfrequenz und 196 MB Hauptspeicher eingesetzt. Wichtiger sind allerdings weniger die absoluten Laufzeiten, sondern ihre Größen relativ zueinander.

Datensatz	Netz-Typ	E_V	Regressionsfehler Trainingset [%]	Klassifikationsfehler Trainingset [%]	Regressionsfehler Testset [%]	Klassifikationsfehler Testset [%]
glass1	GRNN: schnell	8.257	4.270	10.559 (17/161)	7.995	32.075 (17/53)
glass1	GRNN: präzise	6.058	2.541	6.211 (10/161)	6.621	26.415 (14/53)
glass1	PROBEN1 (no shortcut)		8.155		9.240	32.700 (17.33/53)
glass2	GRNN: schnell	6.559	1.996	5.590 (9/161)	7.729	33.962 (18/53)
glass2	GRNN: präzise	4.624	1.271	3.106 (5/161)	7.737	35.849 (19/53)
glass2	PROBEN1 (linear)		9.495		10.340	55.280 (29.30/53)
glass3	GRNN: schnell	7.184	2.989	9.317 (15/161)	7.601	30.189 (16/53)
glass3	GRNN: präzise	5.326	2.117	7.453 (12/161)	8.043	30.189 (16/53)
glass3	PROBEN1 (no shortcut)		8.340		10.740	58.400 (30.95/53)
heart1	GRNN: schnell	12.940	7.370	8.261 (57/690)	13.542	17.391 (40/230)
heart1	GRNN: präzise	11.306	6.138	7.536 (52/690)	13.267	16.522 (38/230)
heart1	PROBEN1 (no shortcut)		11.170		14.190	19.720 (45.36/230)
heart2	GRNN: schnell	12.300	6.444	7.971 (55/690)	14.098	19.565 (45/230)
heart2	GRNN: präzise	11.039	5.438	6.667 (46/690)	13.653	18.261 (42/230)
heart2	PROBEN1 (linear)		11.940		13.520	16.430 (37.79/230)
heart3	GRNN: schnell	11.582	5.566	6.667 (46/690)	16.127	23.043 (53/230)
heart3	GRNN: präzise	10.122	4.314	5.652 (39/690)	15.594	22.174 (51/230)
heart3	PROBEN1 (linear)		10.940		16.390	22.650 (52.10/230)
hearta1	GRNN: schnell	7.050	3.467 / 1.775		6.799 / 3.481	
hearta1	GRNN: präzise	6.189	3.210 / 1.643		6.281 / 3.216	
hearta1	PROBEN1 (linear)		----- / 4.120		----- / 4.470	
hearta2	GRNN: schnell	8.748	6.506 / 3.331		8.319 / 4.260	
hearta2	GRNN: präzise	6.960	5.008 / 2.564		6.567 / 3.363	
hearta2	PROBEN1 (linear)		----- / 4.225		----- / 4.190	
hearta3	GRNN: schnell	6.773	3.086 / 1.580		6.991 / 3.579	
hearta3	GRNN: präzise	5.946	2.897 / 1.483		6.663 / 3.412	
hearta3	PROBEN1 (linear)		----- / 4.100		----- / 4.540	
heartac1	GRNN: schnell	9.783	6.056 / 3.100		7.718 / 3.951	
heartac1	GRNN: präzise	7.652	3.097 / 1.586		6.021 / 3.083	
heartac1	PROBEN1 (no shortcut)		----- / 4.180		----- / 2.470	
heartac2	GRNN: schnell	7.882	3.173 / 1.625		8.277 / 4.238	
heartac2	GRNN: präzise	6.537	2.279 / 1.167		7.119 / 3.645	
heartac2	PROBEN1 (linear)		----- / 4.290		----- / 3.870	
heartac3	GRNN: schnell	7.479	3.326 / 1.703		8.033 / 4.113	
heartac3	GRNN: präzise	6.350	3.034 / 1.553		7.674 / 3.929	
heartac3	PROBEN1 (pivot)		----- / 3.805		----- / 5.370	
heartc1	GRNN: schnell	13.387	5.682	5.702 (13/228)	17.594	24.000 (18/75)
heartc1	GRNN: präzise	11.366	4.078	5.263 (12/228)	17.503	20.000 (15/75)
heartc1	PROBEN1 (linear)		9.910		16.120	19.730 (14.80/75)
heartc2	GRNN: schnell	17.115	9.618	9.649 (22/228)	7.848	4.000 (3/75)
heartc2	GRNN: präzise	14.348	6.867	7.456 (17/228)	5.586	4.000 (3/75)
heartc2	PROBEN1 (linear)		13.870		6.340	3.200 (2.4/75)
heartc3	GRNN: schnell	15.910	9.005	5.702 (13/228)	16.709	17.333 (13/75)
heartc3	GRNN: präzise	12.306	5.362	6.140 (14/228)	13.383	20.000 (15/75)
heartc3	PROBEN1 (linear)		12.180		12.530	14.270 (10.75/75)
horse1	GRNN: schnell	15.030	2.243	1.465 (4/273)	12.583	29.670 (27/91)
horse1	GRNN: präzise	12.902	1.224	1.099 (3/273)	12.236	29.670 (27/91)
horse1	PROBEN1 (pivot)		13.215		13.950	26.650 (24.25/91)
horse2	GRNN: schnell	14.003	1.569	1.465 (4/273)	16.554	38.462 (35/91)
horse2	GRNN: präzise	12.028	0.730	1.099 (3/273)	17.642	36.264 (33/91)
horse2	PROBEN1 (linear)		12.305		17.430	34.840 (31.7/91)
horse3	GRNN: schnell	14.438	2.652	2.564 (7/273)	14.119	34.066 (31/91)
horse3	GRNN: präzise	12.736	1.389	0.733 (2/273)	13.618	31.868 (29/91)
horse3	PROBEN1 (linear)		13.010		15.500	32.420 (29.50/91)
mushroom1	GRNN: schnell	0.960	0.678	0.000 (0/6093)	0.856	0.000 (0/2031)
mushroom1	PROBEN1 (linear)		0.014		0.011	0.000 (0/2031)
soybean1	GRNN: schnell	0.623	0.180	2.339 (12/513)	0.755	10.588 (18/170)
soybean1	GRNN: präzise	0.392	0.093	0.975 (5/513)	0.692	8.235 (14/170)
soybean1	PROBEN1 (pivot)		0.585		1.030	9.060 (15.40/170)
soybean2	GRNN: schnell	0.624	0.195	1.949 (10/513)	0.603	8.824 (15/170)
soybean2	GRNN: präzise	0.452	0.110	0.585 (3/513)	0.508	7.059 (12/170)
soybean2	PROBEN1 (linear)		0.805		1.050	4.240 (7.21/170)
soybean3	GRNN: schnell	0.540	0.138	1.170 (6/513)	0.673	8.235 (14/170)
soybean3	GRNN: präzise	0.376	0.071	0.195 (1/513)	0.729	8.824 (15/170)
soybean3	PROBEN1 (linear)		0.870		1.030	7.000 (11.90/170)
thyroid1	GRNN: schnell	3.328	2.531	5.593 (302/5400)	3.285	6.111 (110/1800)
thyroid1	PROBEN1 (pivot)		0.820		1.310	2.320 (41.76/1800)

Tabelle A.3: *Tabellierte Ergebnisse der* PROBEN1*–Benchmarks: Fehler von Trainings- und Testset — Teil II*

Datensatz	Netz-Typ	Sigma-Faktor m	Regressionsfehler Testset [%] (opt. m)	Klassifikationsfehler Testset [%] (opt. m)	Trainingszeit			
					h	min	sec	millisec
building1	GRNN: schnell	3.00	1.840 / 0.329		0	20	34	495
building1	GRNN: präzise	3.00	1.960 / 0.373		63	10	18	176
cancer1	GRNN: schnell	0.50	1.869	2.299 (4/174)	0	0	19	307
cancer1	GRNN: präzise	0.50	0.796	1.149 (2/174)	0	2	40	570
cancer2	GRNN: schnell	3.00	2.684	2.874 (5/174)	0	0	24	105
cancer2	GRNN: präzise	1.00	3.171	4.023 (7/174)	0	2	31	978
cancer3	GRNN: schnell	1.25	3.437	4.023 (7/174)	0	0	17	516
cancer3	GRNN: präzise	0.50	2.591	2.874 (5/174)	0	2	19	801
card1	GRNN: schnell	1.00	15.939	19.186 (33/172)	0	0	23	664
card1	GRNN: präzise	3.00	16.996	16.860 (29/172)	0	2	56	43
card2	GRNN: schnell	2.50	19.526	20.349 (35/172)	0	0	22	731
card2	GRNN: präzise	3.00	19.071	21.512 (37/172)	0	2	45	938
card3	GRNN: schnell	1.75	19.119	22.093 (38/172)	0	0	24	596
card3	GRNN: präzise	2.75	19.328	23.837 (41/172)	0	2	42	694
diabetes1	GRNN: schnell	1.00	17.078	26.042 (50/192)	0	0	23	344
diabetes1	GRNN: präzise	0.75	16.910	23.438 (45/192)	0	4	52	430
diabetes2	GRNN: schnell	1.00	16.570	22.396 (43/192)	0	0	32	115
diabetes2	GRNN: präzise	1.50	16.813	25.000 (48/192)	0	5	21	170
diabetes3	GRNN: schnell	1.00	16.645	22.917 (44/192)	0	0	26	528
diabetes3	GRNN: präzise	1.00	16.057	21.354 (41/192)	0	5	0	862
flare1	GRNN: schnell	0.50	0.970 / 0.419		0	2	24	288
flare1	GRNN: präzise	0.50	0.946 / 0.409		0	29	2	556
flare2	GRNN: schnell	1.50	0.274 / 0.249		0	2	25	650
flare2	GRNN: präzise	2.00	0.276 / 0.250		0	27	43	60
flare3	GRNN: schnell	1.25	0.411 / 0.325		0	2	23	896
flare3	GRNN: präzise	1.25	0.412 / 0.325		0	28	23	759
gene1	GRNN: schnell	1.50	10.368	9.332 (74/793)	0	19	39	675
gene1	GRNN: präzise	1.25	9.313	13.493 (107/793)	9	24	29	561
glass1	GRNN: schnell	2.25	8.236	28.302 (15/53)	0	0	11	447
glass1	GRNN: präzise	0.50	7.430	22.642 (12/53)	0	0	26	458
glass2	GRNN: schnell	1.00	7.729	33.962 (18/53)	0	0	9	814
glass2	GRNN: präzise	1.00	7.737	35.849 (19/53)	0	0	26	989
glass3	GRNN: schnell	1.00	7.601	30.189 (16/53)	0	0	6	459
glass3	GRNN: präzise	2.75	8.049	28.302 (15/53)	0	0	29	282
heart1	GRNN: schnell	1.00	13.542	17.391 (40/230)	0	0	34	279
heart1	GRNN: präzise	1.00	13.267	16.522 (38/230)	0	6	29	701
heart2	GRNN: schnell	1.50	14.270	16.957 (39/230)	0	0	35	762
heart2	GRNN: präzise	3.00	16.070	16.522 (38/230)	0	6	31	203
heart3	GRNN: schnell	1.00	16.127	23.043 (53/230)	0	0	37	393
heart3	GRNN: präzise	1.50	15.303	20.870 (48/230)	0	6	29	400
hearta1	GRNN: schnell	1.00	6.799 / 3.481		0	0	23	484
hearta1	GRNN: präzise	1.00	6.281 / 3.216		0	5	42	593
hearta2	GRNN: schnell	1.00	8.319 / 4.260		0	0	26	559
hearta2	GRNN: präzise	1.00	6.567 / 3.363		0	5	55	971
hearta3	GRNN: schnell	1.00	6.991 / 3.579		0	0	28	160
hearta3	GRNN: präzise	1.25	6.616 / 3.387		0	5	52	507
heartac1	GRNN: schnell	0.75	7.519 / 3.850		0	0	2	644
heartac1	GRNN: präzise	0.75	6.020 / 3.082		0	0	13	218
heartac2	GRNN: schnell	1.25	8.125 / 4.160		0	0	2	403
heartac2	GRNN: präzise	1.25	7.005 / 3.587		0	0	13	599
heartac3	GRNN: schnell	0.75	7.936 / 4.063		0	0	2	744
heartac3	GRNN: präzise	0.75	7.542 / 3.862		0	0	13	349
heartc1	GRNN: schnell	2.25	18.393	22.667 (17/75)	0	0	3	656
heartc1	GRNN: präzise	1.00	17.503	20.000 (15/75)	0	0	16	603
heartc2	GRNN: schnell	1.00	7.848	4.000 (3/75)	0	0	3	225
heartc2	GRNN: präzise	1.00	5.586	4.000 (3/75)	0	0	18	847
heartc3	GRNN: schnell	1.00	16.709	17.333 (13/75)	0	0	4	246
heartc3	GRNN: präzise	1.50	13.105	17.333 (13/75)	0	0	17	285
horse1	GRNN: schnell	1.25	12.740	26.374 (24/91)	0	0	10	265
horse1	GRNN: präzise	1.50	12.063	21.978 (20/91)	0	0	48	370
horse2	GRNN: schnell	1.25	15.899	34.066 (31/91)	0	0	8	201
horse2	GRNN: präzise	1.50	15.760	34.066 (31/91)	0	0	54	489
horse3	GRNN: schnell	0.50	15.358	32.967 (30/91)	0	0	8	371
horse3	GRNN: präzise	1.00	13.618	31.868 (29/91)	0	0	53	506
mushroom1	GRNN: schnell	1.00	0.856	0.000 (0/2031)	5	36	34	127
soybean1	GRNN: schnell	2.25	1.043	9.412 (16/170)	0	4	14	506
soybean1	GRNN: präzise	1.00	0.692	8.235 (14/170)	0	29	6	61
soybean2	GRNN: schnell	0.50	0.593	8.235 (14/170)	0	3	5	456
soybean2	GRNN: präzise	1.00	0.508	7.059 (12/170)	0	27	22	0
soybean3	GRNN: schnell	2.00	0.736	6.471 (11/170)	0	3	36	852
soybean3	GRNN: präzise	2.00	0.643	6.471 (11/170)	0	27	26	898
thyroid1	GRNN: schnell	1.00	3.285	6.111 (110/1800)	1	53	16	432

Tabelle A.4: *Tabellierte Ergebnisse der* PROBEN1*–Benchmarks: Fehler des Testsets für optimales m sowie Trainingszeit*

Anhang B

Verteilung von Distanzen in Sequenzräumen

Bei der Untersuchung des Trainingsverhaltens des General Regression Neural Networks in Kapitel 4.7 stellte sich heraus, daß das Netz umso erfolgreicher trainiert werden kann, je gleichförmiger die Trainingspunkte im Suchraum verteilt sind.

In diesem Kapitel wird der Einfluß der Anzahl Buchstaben und Dimensionen eines Sequenzraumes auf die Verteilung der Distanzen zufällig ausgewählter Punkte untersucht.

Es seien b die Anzahl Buchstaben und d die Anzahl Dimensionen des zugrundeliegenden Sequenzraums. Ein Punkt des Sequenzraums ist damit $p \equiv (p_1, \ldots, p_d)$ mit $p_i \in \{0, 1, \ldots, b-1\}$ $(i = 1, \ldots, d)$. Als Distanzmaß zwischen zwei Punkten $p^{(1)}$ und $p^{(2)}$ wird hier der normierte EUCLID'sche Abstand eingesetzt:

$$D(p^{(1)}, p^{(2)}) := \sqrt{\frac{\sum_{i=1}^{d} \left(p_i^{(1)} - p_i^{(2)}\right)^2}{(b-1)^2 \cdot d}} \tag{B.1}$$

Die Metrik nach Gleichung (B.1) liefert Werte im Intervall $[0; 1]$.

Zur numerischen Bestimmung der Verteilung der Distanzen wurde 10^6-mal je ein Punktepaar zufällig ausgewürfelt. Die relativen Häufigkeitsverteilungen der Abstände jedes Punktepaars sind für verschiedene Parameter in den Abbildungen B.1 bis B.4 aufgetragen. Alle Kombinationen der Parameter $b = 2, 4, 10, 100$ und $d = 1, 2, 10, 100$ wurden getestet.

Die Häufigkeitsverteilungen haben für $d \geq 2$ ein deutliches Maximum, dessen Lage von der Anzahl Dimensionen unabhängig zu sein scheint. Je größer die Anzahl Buchstaben b, desto „glatter“ verlaufen die Kurven.

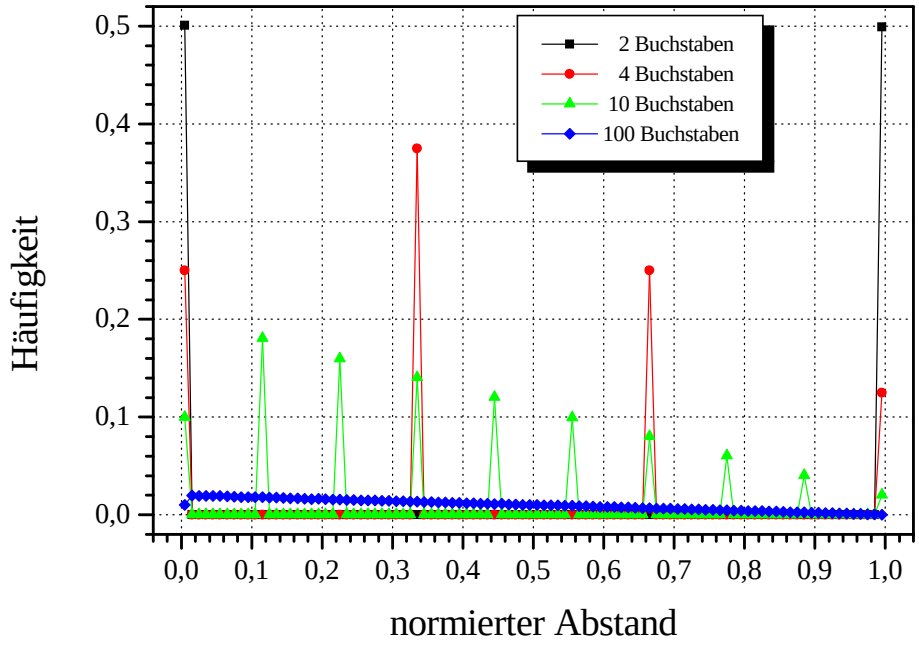

Abbildung B.1: *Häufigkeitsverteilung von Distanzen in einer Dimension*

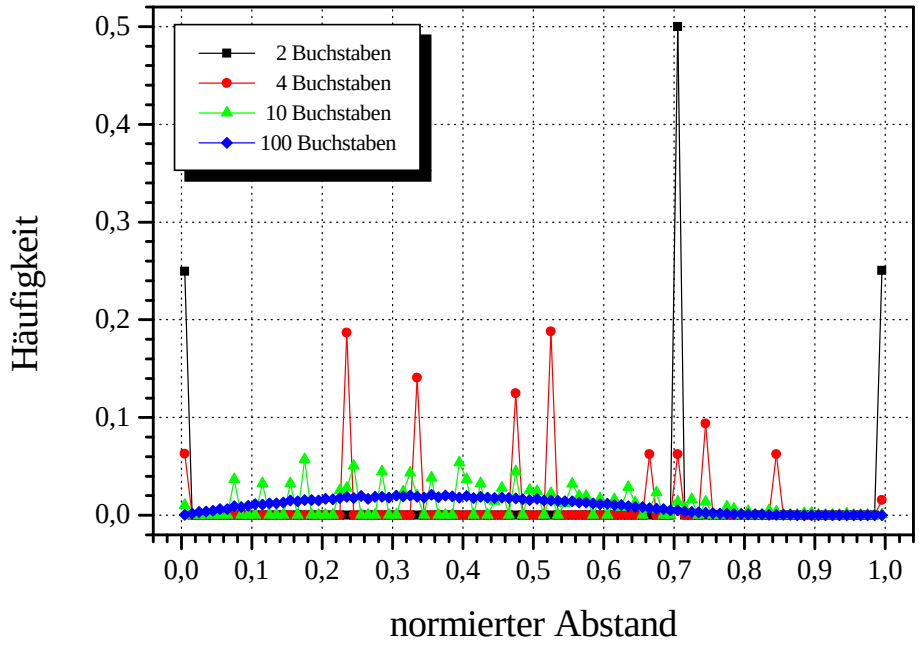

Abbildung B.2: *Häufigkeitsverteilung von Distanzen in zwei Dimensionen*

Das wichtigste Ergebnis für die vorliegende Arbeit:
Für eine steigende Anzahl Dimensionen sinkt die Breite der Kurven, d.h. die Verteilung der Abstände innerhalb einer zufällig ausgewählten Menge an Punkten wird immer gleichförmiger, wie vor allem die Abbildungen B.3 und B.4 zeigen.

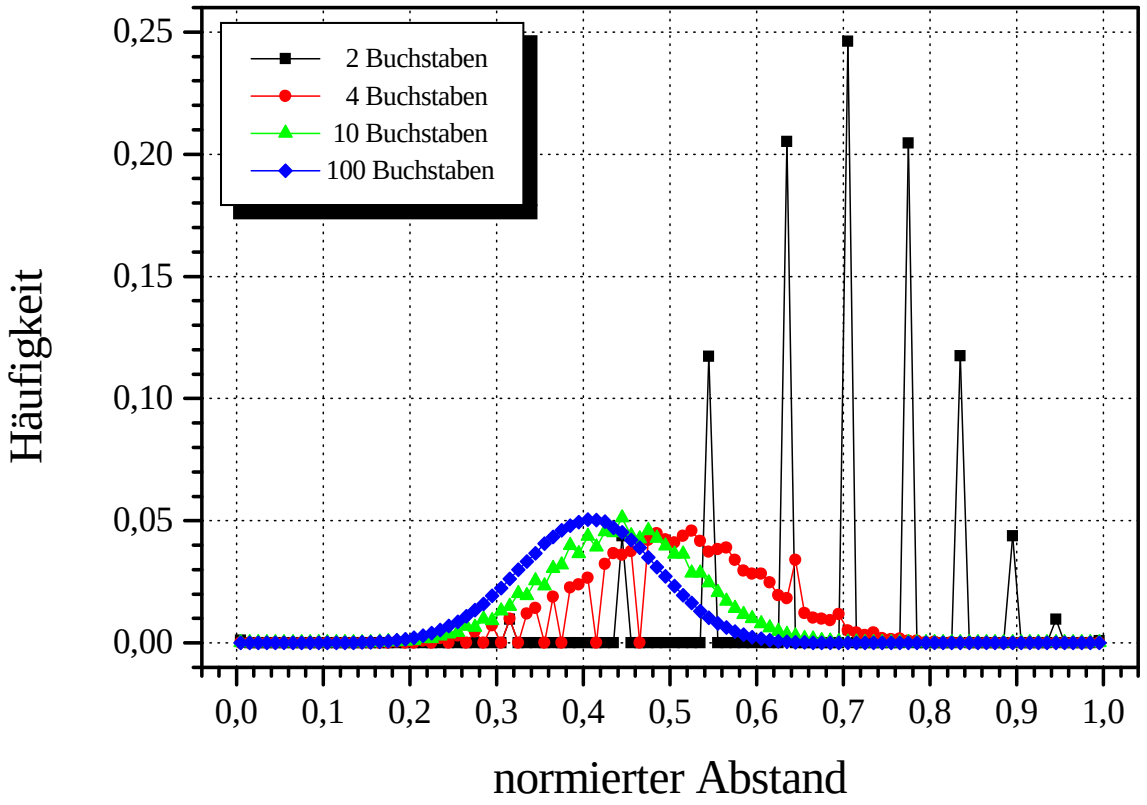

Abbildung B.3: *Häufigkeitsverteilung von Distanzen in zehn Dimensionen*

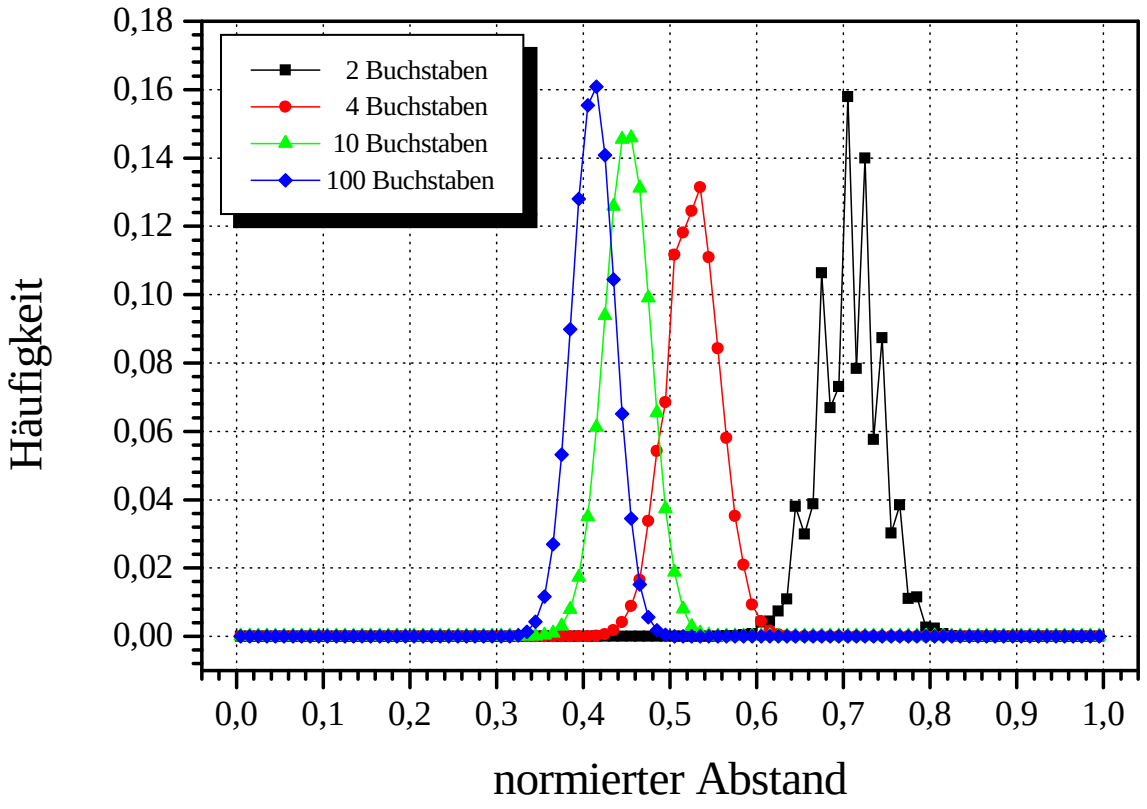

Abbildung B.4: *Häufigkeitsverteilung von Distanzen in 100 Dimensionen*

Anhang C

Proteine — Grundlagen

In diesem Kapitel werden einige elementare Grundbegriffe im Zusammenhang mit Proteinen und Proteindesign vorgestellt.

In Zellen gehören Proteine zu den wichtigsten biologischen Funktionsträgern. Deshalb hat die chemische Industrie großes Interesse an ihrem Einsatz, wie ausgewählte Beispiele demonstrieren:

- regio- und stereoselektive Katalysatoren, die eine Energieeinsparung um viele Größenordnungen ermöglichen, Nebenprodukte minimieren (Umweltschutz) und geringere Ansprüche an die Reaktionstechnik stellen. Als Ausgangspunkt können natürliche Enzyme dienen, die unter physiologischen Bedingungen arbeiten und dann modifiziert werden, um in organischen Lösungsmitteln ebenfalls aktiv zu sein.
- Entwicklung neuer Enzymaktivitäten, die den Abbau von Umweltgiften bewirken.
- Proteaseinhibitoren für die Medizin (beispielsweise Thrombin-Inhibitoren, die das Herzinfarktrisiko mindern).

C.1 Proteinaufbau

Proteine sind Biopolymere, die aus Monomeren — den sogenannten Aminosäuren — aufgebaut sind. Diese Monomere sind über die Säureamidbindung ähnlich einer Perlenkette miteinander verknüpft. Die Reihenfolge der Aminosäuren ist im Erbgut der Zelle, der DNS, festgelegt. Die DNS ist ein Makromolekül, das in Chromosomem

unterteilt ist. Jedes Chromosom besteht wiederum aus nicht-codierenden und codierenden Abschnitten, den Genen. Ein Gen ist aus einer Abfolge der vier möglichen Nukleotide Thymin (T), Cytosin (C), Adenin (A) und Guanin (G) aufgebaut.

1.Position	2.Position				3.Position
	T	C	A	G	
T	TTT Phe (F)	TCT Ser (S)	TAT Tyr (Y)	TGT Cys (C)	T
	TTC - '' -	TCC - '' -	TAC - '' -	TGC - '' -	C
	TTA Leu (L)	TCA - '' -	TAA END	TGA END	A
	TTG - '' -	TCG - '' -	TAG END	TGG Trp (W)	G
C	CTT Leu (L)	CCT Pro (P)	CAT His (H)	CGT Arg (R)	T
	CTC - '' -	CCC - '' -	CAC - '' -	CGC - '' -	C
	CTA - '' -	CCA - '' -	CAA Gln (Q)	CGA - '' -	A
	CTG - '' -	CCG - '' -	CAG - '' -	CGG - '' -	G
A	ATT Ile (I)	ACT Thr (T)	AAT Asn (N)	AGT Ser (S)	T
	ATC - '' -	ACC - '' -	AAC - '' -	AGC - '' -	C
	ATA - '' -	ACA - '' -	AAA Lys (K)	AGA Arg (R)	A
	ATG Met (M)	ACG - '' -	AAG - '' -	AGG - '' -	G
G	GTT Val (V)	GCT Ala (A)	GAT Asp (D)	GGT Gly (G)	T
	GTC - '' -	GCC - '' -	GAC - '' -	GGC - '' -	C
	GTA - '' -	GCA - '' -	GAA Glu (E)	GGA - '' -	A
	GTG - '' -	GCG - '' -	GAG - '' -	GGG - '' -	G

Tabelle C.1: *Tabelle des genetischen Codes. Jedes Protein startet mit ATG (Met) und endet mit einem der drei STOP-Codone TAA, TAG oder TGA.*

Jeweils drei hintereinander angeordnete Nukleotide bezeichnet man als *Codon*. Die $4^3 = 64$ möglichen Codone werden von der Zelle in 20 Aminosäuren übersetzt; die Übersetzungtabelle (siehe Tabelle C.1) wird auch als „genetischer Code“ bezeichnet. Auf diese Art und Weise wird auf jeweils einem[1] Gen ein Protein codiert. Im Erbgut einer Zelle liegen je nach Spezies tausende bis hunderttausende verschiedener Gene und damit Proteine vor.

Da den 64 möglichen Codone lediglich 20 Aminosäuren gegenüberstehen, ist der genetische Code redundant ausgelegt. Einzelne Aminosäuren können durch bis zu sechs verschiedene Codone codiert werden. Es können auch dann noch alle Aminosäuren dargestellt werden, wenn an der dritten Codonposition nur G und C zugelassen werden. Dies hat den Vorteil, daß dann anstelle der drei STOP–Codone nur noch eines vorkommt.

Die meisten Spezies setzen nicht den vollständigen Satz, sondern nur eine Untermenge an Codonen ein. Diesen Effekt nennt man *Codonusage*. Die nicht benutzten

[1]In Wirklichkeit ist es noch komplizierter, da ein Gen häufig in vielen Fragmenten auf einem Chromosom verteilt ist und zuerst korrekt zusammengefügt werden muß.

Codone wirken beim Exprimieren des Proteins in den Ribosomen wie STOP-Codone, sollten also während der Optimierung vermieden werden.

Noch während ein Protein in einer Zelle synthetisiert wird, wechselwirken die bereits angefügten Aminosäuren miteinander, und aus der „Perlenkette“ wird ein kompliziert gefaltetes Objekt — das Protein. Diese räumliche Struktur ist von Protein zu Protein verschieden und muß an die entsprechende Funktion angepaßt sein. Neben diesen strukturellen Voraussetzungen sind die chemischen Eigenschaften der Monomere eines Proteins für die Funktion von entscheidender Bedeutung.

C.2 Proteinoptimierung

Wo liegt das Problem, maßgeschneiderte Proteine zu finden? Oder naiv gefragt: Warum probiert man nicht einfach alle Möglichkeiten aus, tauscht also alle Aminosäuren der Reihe nach aus? Die Antwort liegt in der Kombinatorik bzw. der sog. „kombinatorischen Explosion“:

Es existieren genau 20 verschiedene Aminosäuren. Für ein Protein der Länge n gibt es demzufolge 20^n verschiedene Aminosäurekombinationen. Typische Proteine können aus vielen hundert Aminosäuren bestehen. Für ein Protein der Länge 100 beispielsweise gibt es $20^{100} \approx 10^{130}$ verschiedene Aufbaumöglichkeiten! Diese Zahl übertrifft die Anzahl Atome im Universum bei weitem.

Es ist also — sowohl für die Natur als auch den Ingenieur — ausgeschlossen, alle Varianten einzeln auszutesten. Die handhabbare Größe von Peptidbibliotheken ist experimentell momentan auf etwa 10^9 Moleküle beschränkt. Es ist also unmöglich, alle Sequenzmöglichkeiten ab Längen $n > 7$ im Labor zu realisieren.

Der Wissenschaftler steht vor dem Problem, daß es einerseits unmöglich ist, alle Aminosäurekombinationen eines Proteins durchzuprobieren und auf der anderen Seite nur ein winziger Bruchteil aller möglichen Sequenzen zu funktionsfähigen Proteinen führt. Der einzige Ausweg aus diesem Dilemma ist, den Suchraum auf sinnvolle Weise drastisch einzuschränken. Es sollen möglichst wenige, dafür besonders vielversprechende Sequenzen übrigbleiben, die dann experimentell auf ihre Eignung überprüft werden. In der Grundlagenforschung kristallisieren sich dazu mittlerweile folgende Lösungsansätze heraus:

„Rationales Design“ und „Protein Engineering“

Dies sind die (noch) am häufigsten eingesetzten Verfahren. Es kann dabei von einem natürlichen Protein, dem Wildtyp, gestartet werden, dessen DNS- und damit auch Aminosäure-Sequenz bekannt ist. Durch theoretisches wie auch experimentelles Auswechseln von einzelnen Aminosäuren (meist im reaktiven Zentrum) wird versucht, das Protein zu optimieren. Ob die Modifikationen den gewünschten Erfolg bringen, muß dabei das Experiment zeigen. Da meist nur einige Aminosäuren modifiziert werden, wird der Suchraum automatisch reduziert.

Damit diese Algorithmen optimieren, ist entscheidend, daß der Wissenschaftler die „richtigen“ Veränderungen vornimmt. Aufgrund der Komplexität des Problems ist der wissenschaftliche Aufwand enorm, der Erfolg relativ gering. Denn es ist notwendig, nach dem (theoretischen) Austausch einiger Aminosäuren das dazugehörige Protein dreidimensional im Computer zu falten. Mittels dessen vorhergesagter räumlicher Struktur versucht man, seine Funktionalität nach dem Schlüssel-Schloß–Prinzip abzuschätzen. Leider ist das Problem des räumlichen Faltens eines Proteins bislang noch nicht befriedigend gelöst und außerdem extrem rechenzeitaufwendig.

Aufgrund der prinzipiellen und praktischen Probleme ist es nahezu unmöglich, mit diesen Methoden völlig neue Proteine zu designen. Immerhin können Näherungslösungen gefunden werden, die in der Praxis allerdings häufig ungenügend arbeitsfähig sind.

„Irrationales oder evolutives Design“

Mit diesen noch sehr jungen Verfahren versucht man, die Nachteile des rationalen Ansatzes zu vermeiden. Im wesentlichen geht es hier darum, die in der Natur so beeindruckend erfolgreichen Gesetze der Evolution (Mutation, Rekombination, Selektion) auf Gen-Ebene zum Optimieren von Proteinen zu imitieren:

Man beginnt mit einer geeignet gewählten Startpopulation an Genen. Eines von vielen möglichen Verfahren ist, die Gene in Bakterien oder Phagen einzupflanzen, welche dann die neuen Proteine produzieren. Die Proteine werden entsprechend ihrer Fähigkeit, die Zielaufgabe zu erfüllen, selektiert. Die Gene der besten Proteine werden für die nächste Runde miteinander rekombiniert und mutiert. Die Rekombinationen entsprechen auf Protein-Ebene dem Austausch ganzer Proteinsegmente. Es werden also Unterstrukturen, die sowieso schon funktionieren, in neue Proteinbereiche gebracht. Erfolgreiche „Erfindungen“ einzelner Proteine werden damit schnell

an andere weitergegeben. Damit erhöht sich drastisch die Erfolgswahrscheinlichkeit des neuen Proteins, und es wird trotzdem nur ein vergleichsweise kleiner Bereich des Sequenzraums ausprobiert.

Im Unterschied zum rationalen Design optimiert hier der Algorithmus „von selbst“, der Wissenschaftler muß „lediglich“ das Verfahren als solches beherrschen. Nachteil ist aber, daß der experimentelle Aufwand wesentlich größer ist, da immer mit einer größeren Anzahl von Proteinen (häufig $10^6 - 10^8$) gearbeitet werden muß und unter Umständen viele Iterationszyklen notwendig sind. Der Vorteil ist, daß diese Methoden prinzipiell in der Lage sind, unkonventionelle, neue Lösungen zu finden. Eine Übersicht über evolutive Methoden im Enzymdesign wird von Andreas Schwienhorst in [104] gegeben.

Anhang D

Künstliche neuronale Netze — eine Einführung

In diesem Anhang soll ohne Herleitungen und Beweise ein kurzer Abriß über die wichtigsten Architekturen und Trainingsverfahren künstlicher neuronaler Netze gegeben werden. Das Ziel ist lediglich ein Grundverständnis der wichtigsten Konzepte und Begriffe bezüglich neuronaler Netze. Diese Zusammenfassung kann aufgrund des umfangreichen Fachgebiets nur die Oberfläche berühren, weshalb auf weitere einführende Literatur wie beispielsweise die Habilitationsschrift von Prof. Andreas Zell [132] verwiesen wird, an der sich die Darstellung in diesem Kapitel in Teilen orientiert.

Neuronale Netze, genauer *künstliche neuronale Netze* oder *artificial neural networks* (ANN), sind informationsverarbeitende Systeme, die sich aus einfachen Einheiten, Neuronen genannt, zusammensetzen. Die Neuronen senden sich Informationen über gerichtete Verbindungen zu. Deshalb kann ein ANN als gerichteter, gewichteter Graph verstanden werden, wobei die Kanten die gewichteten Verbindungen zwischen den Neuronen darstellen.

Die ursprüngliche Motivation zur Untersuchung von ANNs entstammte dem Wunsch, die Arbeitsweise biologischer Gehirne auf abstrakte Weise verstehen zu wollen. Die erste Arbeit dazu stammt aus dem Jahr 1943 von Warren McCulloch und Walter Pitts [73].

Das Fachgebiet verselbstständigte sich allerdings rasch bis hin zu Netzwerktypen, die nicht mehr biologisch motiviert sind. Künstliche neuronale Netze sind mittlerweile zu eigenständigen statistischen Verfahren weiterentwickelt worden, die vielfach leistungsfähiger sind als Methoden der traditionellen Statistik. Haupteinsatzgebiete,

aus denen sie nicht mehr wegzudenken sind, sind *nichtlineare Regression*, *Klassifizierung* und *Zeitreihenanalyse*.

In diesem Kapitel soll aus Platzgründen weder auf Probleme der Netze bei der optimalen Generalisierung noch auf die Fülle an Algorithmen zur Bestimmung der freien Trainingsparameter eingegangen werden. Ein großes, ebenfalls übergangenes Thema sind konstruktive und pruning[1]–Algorithmen, die zur Optimierung der Netzwerkarchitektur entwickelt wurden.

D.1 Grundsätzliches

Obwohl sich die unterschiedlichen Netzwerktypen teilweise enorm unterscheiden, gibt es einige grundsätzliche Eigenschaften, die allen Netzvarianten gemeinsam sind.

Vorneweg wichtige Termini:

Repräsentierbarkeit steht für die Fähigkeit eines ANN, eine gegebene Funktion realisieren zu können. Die Topologie ist dabei vorgegeben, die Gewichte dürfen optimiert werden.

Lernfähigkeit ist die Fähigkeit eines Trainingsverfahrens, einem ANN eine repräsentierbare Funktion durch Manipulation der Gewichte beibringen zu können.

Zur Nomenklatur: Ein Netz wird als s–stufig bezeichnet, wenn es s Schichten trainierbarer Verbindungen besitzt. Insbesondere wird damit die Eingabeschicht nicht mitgezählt.

Eine *Zeitreihe* ist eine Abfolge von Werten, bei denen die zeitliche Reihenfolge entscheidend ist. Damit ist nicht nur das Muster selbst wichtig, sondern seine Position in der vollständigen Sequenz. Ein prominentes Beispiel für eine Zeitreihe ist der Börsenkurs einer Aktie.

Es sei im folgenden n die Anzahl der p-dimensionalen Trainingsmuster.

Neuronale Netze setzen sich im allgemeinen aus folgenden Bestandteilen zusammen:

Topologie bzw. Architektur

Die Struktur eines Netzes wird durch die Matrix $W = [w_{ij}]$ der Verbindungen aller Neuronen beschrieben. Der Eintrag w_{ij} steht für die gerichtete Verbindung von

[1] Gezieltes Eliminieren von Verbindungen und Neuronen aus dem Netz.

Neuron i zum Neuron j mit dem Gewicht w_{ij}. Biologisch interpretiert bedeutet ein negatives Gewicht Hemmung, ein positives Gewicht Anregung des nachfolgenden Neurons j. Ist das Gewicht gleich Null, so besteht keine Verbindung zwischen den Neuronen i und j.

Propagierungsfunktion

Sie gibt an, wie sich die Eingabe eines Neurons aus den Ausgaben der übrigen Neuronen und den Verbindungsgewichten zusammensetzt. Die Netzeingabe $net_j(t)$ von Zelle j zum Zeitpunkt t berechnet sich aus dem Skalarprodukt zwischen dem Gewichtsvektor $\vec{w}_j$ und dem Ausgabevektor $\vec{o}$ der mit Zelle j verbundenen Neuronen:

$$net_j(t) := \sum_i w_{ij}\ o_i(t) \tag{D.1}$$

Neuronen als elementare Einheiten

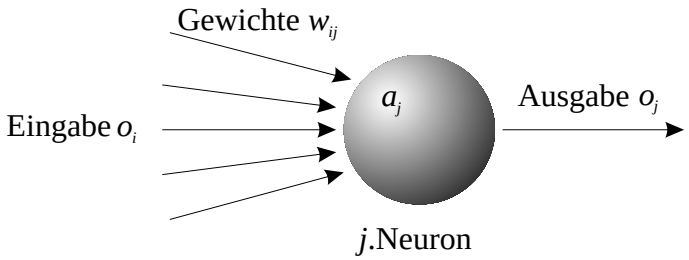

Abbildung D.1: *Abstrahiertes Neuron als elementarer Baustein neuronaler Netze*

- Aktivierungszustand $a_i(t)$ gibt den Grad der Aktivierung von Zelle i zum Zeitpunkt t an.

- Die Aktivierungsfunktion $f_{act}(a_j(t), net_j(t), \theta_j)$ gibt an, wie sich ein neuer Aktivierungszustand $a_j(t+1)$ des Neurons j aus der alten Aktivierung $a_j(t)$, dem Schwellwert θ_j und der Netzeingabe $net_j(t)$ berechnet:

$$a_j(t+1) := f_{act}\left(a_j(t), net_j(t), \theta_j\right) \tag{D.2}$$

  Die häufigsten Aktivierungsfunktionen sind (der Übersichtlichkeit halber ohne Indizes)

  die Identitätsfunktion $f_{act}(x, \theta) = x - \theta$,

die binäre Schwellwertfunktion $f_{act}(x,\theta) = \begin{cases} +1, & \text{für } x \geq \theta \\ -1 \text{ (oder 0)}, & \text{für } x < \theta \end{cases}$,

die logistische Funktion $f_{act}(x,\theta) = \frac{1}{1+e^{-(x-\theta)}}$

und der Tangens hyperbolicus $f_{act}(x,\theta) = \tanh(x-\theta)$.

Es kann o.B.d.A. für alle i Neuronen $\theta_i = 0$ gesetzt und dafür ein zusätzliches einzelnes Bias- oder Schwellwertneuron eingeführt werden. Das Schwellwertneuron hat zu jedem Zeitpunkt die konstante Ausgabe 1. Die Gewichte vom Schwellwertneuron zu den übrigen Neuronen sind äquivalent mit den Schwellwerten θ_i. Damit lassen sich die Trainingsalgorithmen vereinfachen, da die Schwellwerte nicht gesondert trainiert werden müssen.

- Die Ausgabe der Zelle j wird durch die Ausgabefunktion $f_{out}(a_j(t+1))$ aus der Aktivierung $a_j(t+1)$ bestimmt:

$$o_j(t+1) := f_{out}(a_j(t+1)) \tag{D.3}$$

In der Regel werden in der Literatur die Ausgabefunktion f_{out} und die Aktivierungsfunktion f_{act} nicht getrennt betrachtet, sondern in der Aktivierungsfunktion f_{act} zusammengefaßt. Die Ausgabefunktion f_{out} ist deshalb meist die Identitätsfunktion.

Lernregel bzw. Trainingsalgorithmus

Der Trainingsalgorithmus ist das Verfahren, mit dem das neuronale Netz lernt, für eine vorgegebene Eingabe eine gewünschte Ausgabe zu liefern. Erreicht wird dies durch

- Modifikation der Gewichte w_{ij}. Ein Setzen der Verbindung zwischen Neuron i und j entspricht dem Verändern von $w_{ij} = 0$ zu $w_{ij} \neq 0$. Analog geschieht das Löschen von Verbindungen.

- Einfügen neuer Neuronen bzw. Eliminieren bestehender Neuronen respektive ihrer Gewichte w_{ij}.

- Modifizieren der Aktivierungs-, Propagierungs- und Ausgabefunktion bestehender Neuronen.

Die beiden letzten Punkte verändern zusätzlich die Netzwerkarchitektur. Trainingsverfahren, die sowohl die Gewichte als auch die Topologie optimieren, gewinnen in der Praxis immer mehr an Bedeutung.

Die Trainingsalgorithmen lassen sich in drei Gruppen unterteilen:

1. **Überwachtes Lernen („supervised learning“):**
   Ein externer „Lehrer“ gibt zu jedem Eingabemuster des Trainingssets das korrekte bzw. beste Ausgabemuster an. Aufgabe des Lernverfahrens ist es, das Netz (v.a. seine Gewichte) so zu ändern, daß es die Abbildung zwischen Eingabe- und Ausgabemustern lernt. Insbesondere ist Ziel des Trainings nicht das einfache Auswendiglernen des Trainingssets, sondern das Extrahieren der verallgemeinerten Regeln zwischen Ein- und Ausgabemustern, so daß das Netz auch für unbekannte Eingabemuster die korrekte Ausgabe liefern kann. Da hier die erwünschten Aktivierungen aller Ausgabeneuronen vorgegeben werden müssen, ist dieser Ansatz biologisch schwierig zu begründen.

   Das prominenteste Beispiel für überwachtes Lernen sind feedforward–Netze, die nach dem Backpropagation–Algorithmus (siehe Kapitel D.2.2.1) trainiert werden. Der größte Teil aller Publikationen über ANNs beschäftigt sich mit Anwendungen und Variationen von Backpropagation–Netzen.

2. **Verstärkendes Lernen („reinforcement learning“):**
   Hier gibt der Lehrer bei jedem Trainingsmuster lediglich an, ob richtig oder falsch klassifiziert wurde, aber nicht die korrekte Ausgabe. Das Netz muß die korrekte Ausgabe von selbst finden. Diese Art zu Lernen ist langsamer als überwachtes Lernen, aber biologisch plausibler.

   Zum Beispiel können feedforward–Netze mit verstärkendem Lernen trainiert werden.

3. **Unüberwachtes Lernen („unsupervised learning“):**
   Im Gegensatz zu den beiden bereits vorgestellten Klassen von Trainingsverfahren existiert hier überhaupt kein Lehrer. Lernen geschieht durch Selbstorganisation; dem Netz werden nur Eingabemuster präsentiert. Das Netz klassifiziert ähnliche Eingabemuster in ähnliche Klassen durch Aktivierung von gleichen oder räumlich benachbarten Neuronen. Diese Art des Lernen ist biologisch am plausibelsten, allerdings nur für Klassifikationsaufgaben und topologieerhaltende Abbildungen geeignet.

Die bekanntesten Beispiele sind die selbstorganisierenden Karten (siehe Kapitel D.2.5) von Prof. TEUVO KOHONEN der Helsinki Universität in Finnland.

Von DONALD O. HEBB stammt eines der ersten und bekanntesten Trainingsverfahren, die **Hebb'sche Lernregel**:

$$\Delta w_{ij} := \eta \, o_i \, a_j \tag{D.4}$$

Die Regel ist biologisch motiviert und besagt, wenn Neuron j eine Eingabe von Neuron i erhält und beide stark aktiviert sind, soll die Stärke w_{ij} der Verbindung von i nach j erhöht werden.

D.2 Populäre Netztypen

Nach der Zusammenfassung der Grundlagen künstlicher neuronaler Netze sollen nun die wichtigsten Netztypen und Trainingsverfahren vorgestellt werden. Häufig wird in der Literatur bei der Namensgebung die Trennung zwischen Netzarchitektur und Trainingsverfahren nicht strikt vollzogen. Ein neuer Trainingsalgorithmus definiert oft schon einen „neuen" Netztyp. Bekannteste Beispiele dafür sind Backpropagation-, Cascade Correlation- oder Radiale Basisfunktionen–Netze, die allesamt zum Architekturtyp der feedforward–Netze gehören.

D.2.1 Perzeptron

Unter der Bezeichnung „Perzeptron" firmiert eine ganze Familie neuronaler Netze, die von FRANK ROSENBLATT 1958 in [89] entwickelt wurde. MINSKY und PAPERT analysierten diese Klasse mathematisch rigoros in ihren Buch [74] von 1969.

Die typische Struktur eines Perzeptrons ist in Abbildung D.2 dargestellt. Mit Perzeptrons wurden am häufigsten Probleme der visuellen Mustererkennung studiert. Als Eingabezellen dienen Zellen der künstlichen Retina. Diese werden mit Neuronen der ersten Verarbeitungsschicht fest verbunden. M. MINSKY und S. PAPERT bezeichnen diese Neuronen als „Detektoren für einfache Muster". Alle Detektoren werden über gewichtete Verbindungen mit einem einzelnen Ausgabeneuron verbunden. Das Ausgabeneuron dient als Klassifikator, ob das anliegende Muster erkannt wird oder nicht.

Da nur eine Ebene trainierbarer, reeller Gewichte vorliegt, wird diese Perzeptron–Variante als *einstufiges Perzeptron* bezeichnet. Die Aktivierungen der Neuronen

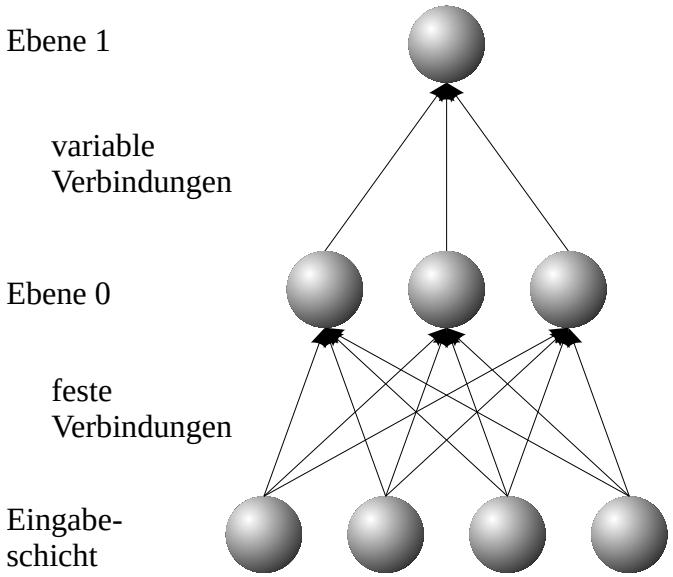

Abbildung D.2: *Schema des Perzeptrons*

dürfen in den meisten Anwendungen nur binäre Werte annehmen. Das Training erfolgt mit der HEBB'schen Lernregel nach Gleichung (D.4).

FRANK ROSENBLATT bewies 1962 in [90] das berühmte, häufig falsch verstandene Perzeptron-Lern–Theorem:

> Der Lernalgorithmus des Perzeptrons konvergiert in endlicher Zeit,
> d.h. das Perzeptron kann in endlicher Zeit alles lernen,
> *was es repräsentieren kann.*

Das Problem ist, daß ein einstufiges Perzeptron nur sehr wenige Funktionen repräsentieren kann:
Einstufige Perzeptrons können nur linear separable Funktionen repräsentieren.

Zweistufige Perzeptrons können konvexe, zusammenhängende Polygone durch Überlagerung von Hyperebenen klassifizieren.

Dreistufige Perzeptrons können durch Überlagerung und Schnitt konvexer Polygone Mengen beliebiger Form, die nicht zusammenhängend sein müssen, identifizieren.

Höhere Stufen besitzen keine zusätzlichen Fähigkeiten mehr.

MINSKY und PAPERT zeigten in [74], daß die Gewichte mit einer Genauigkeit, die *exponentiell* von der Problemgröße abhängt, trainiert werden müssen — eine praktisch nicht erfüllbare Forderung. Dieser Satz führte zu einem jahrelangen Erlahmen von Forschung und Anwendung künstlicher neuronaler Netze. Erst die

(Wieder-)Entdeckung von feedforward–Netzen mit dem berühmten Backpropagation–Trainingsverfahren Mitte der 80er Jahre führte zur heutigen Popularität künstlicher neuronaler Netze.

D.2.2 Feedforward–Netze

Die populärsten, in der Praxis eingesetzten Netztypen sind neuronale Netze mit Verbindungen, deren Richtung ausschließlich von der Eingabe- zur Ausgabeschicht verläuft. In der Gewichtsmatrix W sind alle Gewichte w_{ij} mit $i \leq j$ gleich Null.

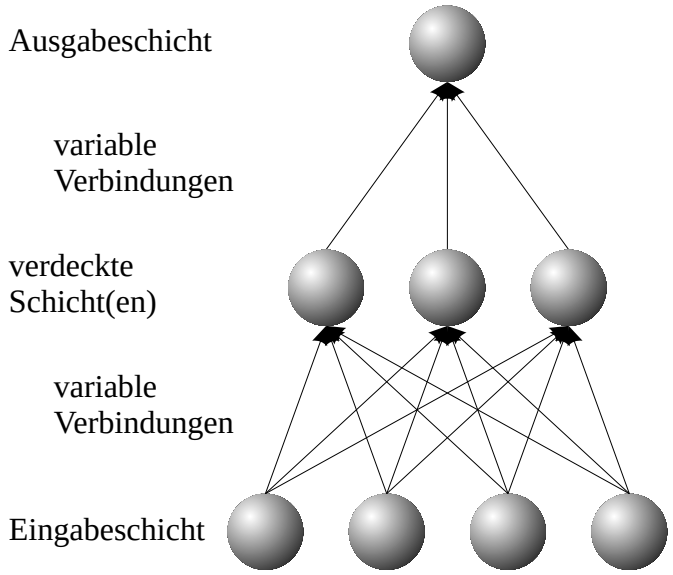

Abbildung D.3: *Struktur eines zweischichtigen feedforward–Netzes*

In Abbildung D.3 ist die typische Architektur eines zweischichtigen feedforward–Netzes dargestellt. Das Netz hat in dieser Darstellung eine Eingabeschicht, eine verdeckte Schicht („hidden layer“) und eine Ausgabeschicht.

Einige Probleme („2–Spiralen“) lassen sich nur durch feedforward–Varianten mit mehreren verdeckten Schichten und Verbindungen, die Ebenen überspringen können („shortcut connections“), lösen.

Es existieren etliche theoretische Arbeiten (z.B. G. Cybenko [12] , Kurt Hornik [57]), die nachweisen, daß bereits ein zweischichtiges feedforward–Netz nach Abbildung D.3 mit einer endlichen Anzahl verdeckter Neuronen beliebige (quadratintegrable) multidimensionale Funktionen auf beliebige Genauigkeit approximieren kann. Dies ist mit einer großen Klasse von Aktivierungsfunktionen möglich, deren wesentliche Eigenschaft die Beschränktheit ist. Die Aussage gilt nur, wenn der

Trainingsfehler des Netzes nahe dem globalem Optimum liegt. Leider gibt es keine analytische Möglichkeit, die optimale Anzahl verdeckter Neuronen im Vorfeld des Trainings zu ermitteln. Dies ist die Aufgabe einer Fülle von heuristischen Verfahren.

D.2.2.1 Backpropagation–Trainingsverfahren

Das Backpropagation–Trainingsverfahren ist mit seinen Varianten das am häufigsten eingesetzte Trainingsverfahren für neuronale feedforward–Netze. Obwohl es bereits 1974 in der Dissertation von PAUL WERBOS [128] entwickelt wurde, fand es erst 1986 durch die Veröffentlichung [91] von DAVID E. RUMELHART *et al.* große Beachtung.

Backpropagation ist ein Gradientenabstiegsverfahren, d.h. Gewichtsänderungen finden in der negativen Richtung des Gradienten der Fehlerfunktion statt:

$$\Delta W = -\eta \nabla E(W) \qquad \text{bzw.} \qquad \Delta w_{ij} = -\eta \frac{\partial}{\partial w_{ij}} E(W) \tag{D.5}$$

η ist ein „kleiner“ Wert $(0 < \eta \ll 1)$ und wird als *Lernrate* bezeichnet.

Aus Platzgründen wird an dieser Stelle nicht — wie üblich — die Backpropagation–Regel hergeleitet. Die Formel ist in jedem Einführungsbuch und vielen Artikeln zu finden. Man muß eine Fehlerfunktion wie z.B. die quadratische Abweichung zwischen Ist- und Sollwert definieren. Zur Berechnung des Istwertes muß der Ausgabewert des Netzes bestimmt werden, dazu muß die Aktivierungsfunktion der Neuronen, wie die logistische Funktion bekannt sein. Unter Anwendung der Kettenregel kann aus Gleichung (D.5) die Korrekturfunktion Δw_{ij} abgeleitet werden.

Ein Problem dieses Ansatzes ist offenkundig:
Jedes Gradientenabstiegsverfahren kann nur dann garantiert das Optimum finden, wenn keine lokalen Optima existieren. In der Praxis ist diese Forderung nur selten zu erfüllen. Um ein frühzeitiges Konvergieren in lokalen Optima zu erschweren, existieren eine Fülle an Varianten des Backpropagation–Verfahrens, durch die es erst praxistauglich wird. Ein Beispiel ist der Quickprop–Algorithmus von SCOTT E. FAHLMAN in [34]. Obwohl das Verfahren bereits 1988 entwickelt wurde, ist es immer noch konkurrenzfähig bezüglich Geschwindigkeit. Die Studie [34] ist gleichzeitig eine sehr gelungene und empfehlenswerte Untersuchung verschiedener Aspekte des Backpropagation–Verfahrens.

D.2.2.2 Cascade-Correlation Learning Architecture (CCLA)

Die *Cascade-Correlation Learning Architecture* [36] von SCOTT E. FAHLMAN und CHRISTIAN LEBIERE ist ein konstruktives Verfahren, das sowohl Gewichte als auch

Architektur von feedforward–Netzen optimiert.

Es startet mit einem Netz, das nur aus Eingabe- und Ausgabeschicht besteht, also ohne verdeckte Schicht. Dieses Netz wird mit einem frei wählbaren feedforward–Trainingsverfahren solange trainiert, bis der Trainingsfehler „optimal“ ist. Iterativ werden anschließend verdeckte Neuronen eingefügt, die nicht nur vollständig mit der Eingabe- und Ausgabeschicht, sondern mit *allen* bereits eingefügten verdeckten Neuronen verknüpft werden. Entscheidend ist, daß, sobald ein neues verdecktes Neuron eingefügt wird, die übrigen Gewichte „eingefroren“, also nicht mittrainiert werden. Als Fehlerfunktion dient nicht die quadratische Abweichung, sondern die *Korrelation* zwischen Ist- und Sollwert, daher der Name des Verfahrens.

Viele Probleme lassen sich durch dieses Verfahren deutlich schneller als mit Varianten von Backpropagation lösen.

FAHLMAN entwickelte noch eine leistungsfähige, rekurrente Version der Cascade-Correlation–Architektur [35].

D.2.2.3 Radiale-Basisfunktionen–Netze (RBF)

Radiale-Basisfunktionen–Netze sind feedforward–Netze mit einer Schicht verdeckter Neuronen und wurden 1989 von T. POGGIO und F. GIROSI in [82] vorgestellt.

Die Eingabeschicht ist vollständig mit der verdeckten Schicht verbunden. Jedes Neuron i der verdeckten Schicht besitzt eine radialsymmetrische Aktivierungsfunktion $h_i(|\mathbf{X} - \mathbf{X}_i|)$ mit der Stützstelle $\mathbf{X}_i$. Die Gewichte der Verbindungen zwischen dem Neuron i und der Eingabeschicht sind identisch mit den Komponenten der Stützstelle $\mathbf{X}_i$. Die einzigen zu trainierenden Gewichte sind damit die Verbindungen w_i von der verdeckten zur Ausgabeschicht.

Mit N verdeckten Neuronen ergibt sich als Netzausgabe $\sum_{i=1}^{N} w_i h_i(|\mathbf{X} - \mathbf{X}_i|)$. Da die Netzausgabe für jedes angelegte Trainingsmuster $\mathbf{X}_j$ gleich y_j sein soll, können die Gewichtsfaktoren w_i durch das folgende lineare Gleichungssystem bestimmt werden:

$$\sum_{i=1}^{N} w_i h_i(|\mathbf{X}_j - \mathbf{X}_i|) = y_j \quad \text{für alle } j = 1, \ldots, n \tag{D.6}$$

Wenn für jedes Trainingsmuster genau ein verdecktes Neuron existiert, d.h. $N = n$, dann läßt sich das Gleichungssystem (D.6) exakt lösen. Die Überlagerung der h_i ergibt eine *Approximation* der Trainingsmuster. Falls $N \neq n$ gilt, so läßt sich das

Gleichungssystem (D.6) nur genähert lösen[2]. Die resultierende Überlagerung der Aktivierungsfunktionen *approximiert* in diesem Fall die Trainingsmuster.

Für die Aktivierungsfunktionen h_i sind verschiedene Klassen an Funktionen zulässig; insbesondere wird häufig die GAUSS'sche Funktion verwendet.

Die Wahl und die Anzahl der Stützstellen $\mathbf{X}_i$ sind neben der Wahl der optimalen Aktivierungsfunktionen h_i Inhalt diverser Erweiterungen der ursprünglichen RBF–Netze. Die Wahl der Stützstellen kann z.B. durch Vorbehandlung des Trainingssets mit KOHONEN–Karten (Kapitel D.2.5) erfolgen; die Feinbestimmung durch Backpropagation–ähnliche Trainingsverfahren.

D.2.2.4 Time-Delay–Netze (TDNN)

Time-Delay–Netze sind eine Variante von feedforward–Netzen, die von A. WAIBEL [125] ursprünglich zur Spracherkennung entwickelt wurden. Das eigentliche Ziel war, Netze zu konstruieren, die Merkmale in zeitlich veränderlichen Mustern an unterschiedlicher Position und mit unterschiedlichen Längen erkennen können.

Will man mit neuronalen Netzen Zeitreihen analysieren und vorhersagen, so kann dies im einfachsten Fall mit feedforward–Netzen geschehen. Dazu zerlegt man die Zeitreihe in Teilfolgen der Länge l. Mit den Teilfolgen als Eingabemuster und dem jeweils folgenden Wert als Ausgabewert wird das Netz trainiert. Entscheidend für die Güte des Verfahrens ist die Größe des Fensters l. Sie kann beispielsweise mittels Hauptkomponentenanalyse der Korrelationsmatrix durch die kleinste Dimension der Trajektorie abgeschätzt werden.

TDNNs lösen das Problem der Zeitreihenanalyse auf eine ähnliche Weise. Charakteristisch ist dabei die spezielle Behandlung der Eingabevektoren. Anstelle eines Eingabefensters der Länge l werden die einzelnen Werte der Zeitreihe dem TDNN der Reihe nach präsentiert. In der Eingabeschicht teilt sich jedes Eingabeneuron in l Verzögerungsneuronen auf. Jedes Verzögerungsneuron steht für einen bestimmten diskreten Zeitschritt. In Abbildung D.4 sind die Verzögerungsneuronen in unterschiedlichen Grautönen gehalten. Gleiche Grautöne entsprechen gleichen Zeitverzögerungen. Dasselbe Prinzip gilt für die verdeckten Schichten und die Ausgabeschicht. Die Anzahl Verzögerungsneuronen pro Neuron in jeder Schicht wird mit $l^{(s)}$ bezeichnet. Der Index s steht für die Nummer der Schicht. Die Eingabeschicht hat den Index $s = 0$. Deshalb gilt $l^{(0)} \equiv l$.

[2]Falls die Gleichung $H \cdot W = Y$ überbestimmt ist, existiert häufig die genäherte Lösung $W = (H^T H)^{-1} H^T \cdot Y$

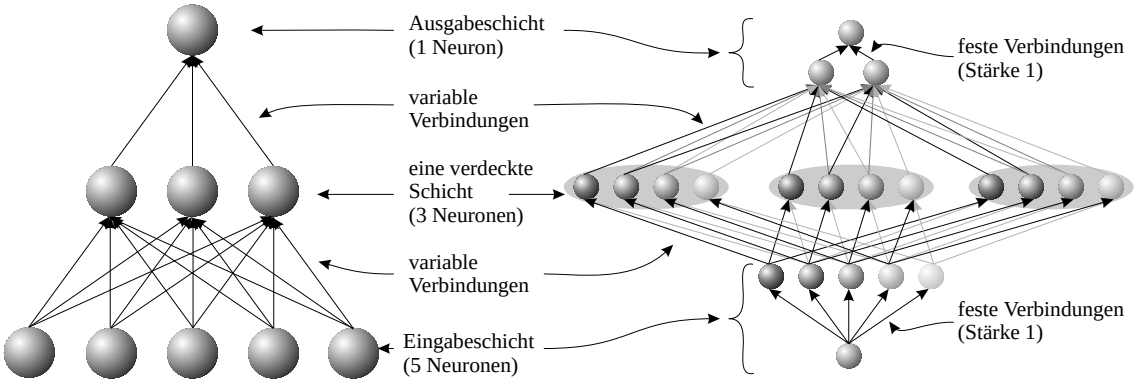

Abbildung D.4: *Struktur eines einfachen Time-Delay–Netzes. Das Diagramm zeigt die Ähnlichkeit zwischen einem feedforward–Netz auf der linken Seite und dem korrespondierenden TDNN auf der rechten Seite.*

Im Unterschied zu feedforward–Netzen werden in der Ausgabeschicht die *Quadrate* der Ausgaben der zu jedem Ausgabeneuron gehörenden Verzögerungsneuronen summiert.

Würden nun alle Neuronen zwischen aufeinanderfolgenden Schichten vollständig miteinander verknüpft, so entstünden sehr schnell zu viele Verknüpfungen, als daß das Netz in der Praxis einsetzbar wäre. Deshalb führte A. WAIBEL das Konzept der *rezeptiven Felder* ein. Ein Neuron „sieht“ damit nur einen kurzen zeitlichen Ausschnitt der Länge $r^{(s)}$ aus der vorhergehenden $(s-1)$.Schicht. Jedes Neuron einer Schicht ist mit $r^{(s)}$ Neuronen der vorhergehenden Schicht verbunden. Für $l^{(s)}$ und $r^{(s)}$ gilt folgende Beziehung:

$$l^{(s+1)} = l^{(s)} - r^{(s+1)} + 1 \qquad \text{(D.7)}$$
$$\text{mit} \quad l^{(0)} := l \quad \text{und} \quad r^{(0)} := 0$$

Damit ergibt sich sofort $l^{(s+1)} \leq l^{(s)}$, d.h. die Anzahl Verzögerungsneuronen kann zwischen zwei aufeinanderfolgenden Schichten nicht zunehmen.

Das Beispielnetz in Abbildung D.4 demonstriert das Prinzip der Verknüpfung zwischen Neuronen unterschiedlicher Schichten. Beispielsweise haben die Verzögerungsneuronen der verdeckten Schicht des TDNN jeweils ein rezeptives Feld der Länge $r^{(1)} = 2$. Für die Ausgabeschicht gilt $r^{(2)} = 3$. Ein verzögertes Neuron der verdeckten Schicht ist damit mit zwei Verzögerungs–Eingabeneuronen verknüpft. Für ein Eingabefenster der Länge $l = 5$ ergibt sich als Anzahl Verzögerungsneuronen in der verdeckten Schicht $l^{(1)} = 5 - 2 + 1 = 4$. Desgleichen ist die Anzahl Verzögerungsneuronen in der Ausgabeschicht $l^{(2)} = 4 - 3 + 1 = 2$.

Durch die Einführung der rezeptiven Felder stellen die Verzögerungsneuronen eines Neurons den zeitlichen Verlauf der Aktivierung dieses Neurons dar. Aus diesem Grund werden alle „parallelen“ Gewichte zwischen den Verzögerungsneuronen eines Neurons und den Verzögerungsneuronen eines Neurons der Vorgängerschicht auf dem gleichen Wert gehalten. Zur Verdeutlichung sind in Abbildung D.4 gleiche Gewichte mit gleicher Graufärbung markiert.

TDNNs werden mit Varianten des Backpropagation–Algorithmus trainiert.

D.2.3 JORDAN- und ELMAN–Netze

Eine andere Methode zur Zeitreihenanalyse ist die direkte Repräsentation von Zeit innerhalb des neuronalen Netzes durch *rückgekoppelte* Neuronen.

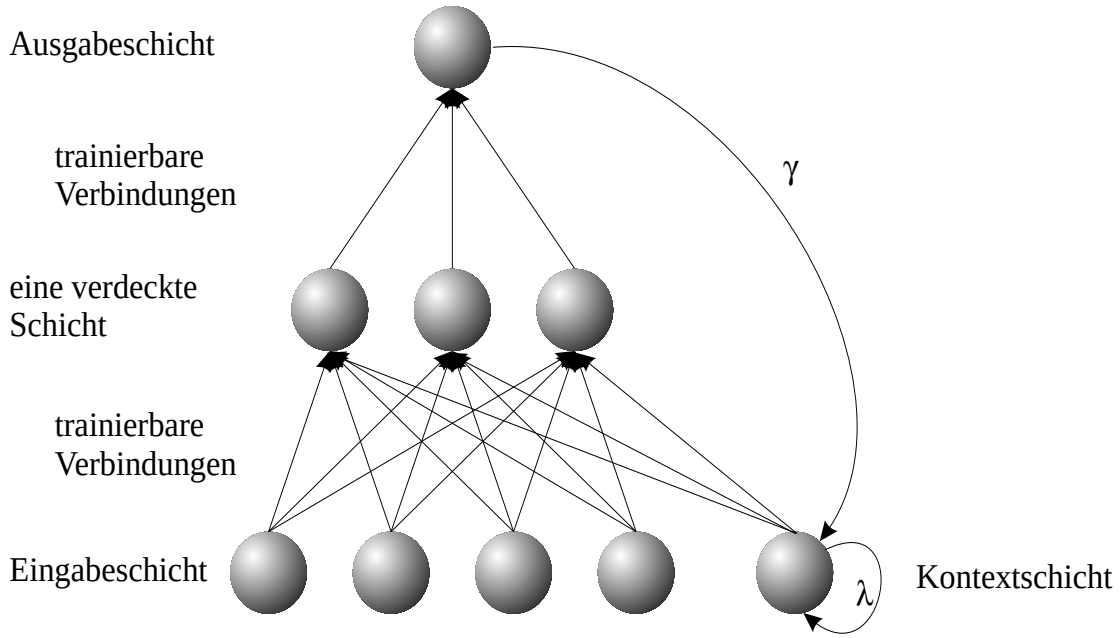

Abbildung D.5: *Struktur eines* JORDAN*–Netzes*

JORDAN–Netze [61] gehören zur Klasse der *partiell rekurrenten Netze*. M. JORDAN erweiterte die Architektur einfacher feedforward–Netze durch Kontextzellen, die den Ausgabezustand speichern, wie Abbildung D.5 zeigt. Für jede Ausgabezelle existiert eine Kontextzelle, die direkt mit der Stärke γ mit der Ausgabezelle verbunden ist. Meist gilt $\gamma = 1$. Außerdem besitzen die Kontextzellen direkte Rückkopplungen der Stärke λ mit $\lambda \in [0; 1]$, die ebenfalls nicht trainierbar sind. Die Aktvierungsfunktion der Kontextzellen ist die identische Abbildung. Es läßt sich zeigen, daß für die

Ausgabewerte $S(t)$ der Kontextzellen zum Zeitpunkt t folgende Beziehung gilt:

$$S(t) = \sum_{i=1}^{t-1} \lambda^{i-1} O(t-i) \tag{D.8}$$

$O(t)$ ist die Ausgabe des Ausgabeneurons zur Zeit t. Die Übergangsfunktion $S(t)$ ist eine exponentiell gewichtete Summe aller bisherigen Ausgaben. Der Wert λ steuert das „Erinnerungsvermögen“ des Netzes. Je größer λ ist, desto stärker werden ältere Ausgaben berücksichtigt.

ELMAN–Netze [32] sind eine Modifikation von JORDAN–Netzen, bei der die Rückkopplungen nicht mehr von der Ausgabeschicht, sondern von der verdeckten Schicht aus zur Kontextschicht verlaufen. Außerdem entfallen die direkten Rückkopplungen der Kontextneuronen. ELMAN–Netze haben gegenüber JORDAN–Netzen den Vorteil, daß die Eignung des Netzes für eine bestimmte Anwendung nicht direkt von der zu erzeugenden Ausgabesequenz abhängig ist, da die Zeit in den internen Zuständen der verdeckten Neuronen codiert ist. Für kompliziertere Probleme bieten sich ELMAN–Netze mit mehreren verdeckten Schichten, sog. *hierarchische* ELMAN–Netze an. Jede der verdeckten Schichten hat dabei ihre eigene Schicht an Kontextneuronen, die auch direkt rekurrente Verbindungen besitzen können. Mit hierarchischen ELMAN–Netzen konnten bessere Ergebnisse für die Prognose von Protein-Sekundärstrukturen als mit reinen feedforward–Netzen und Fenstertechnik [85] erzielt werden.

Partiell rekurrente Netze können mit einer leicht modifizierten Form des online-Backpropagation–Trainingsverfahrens trainiert werden. Entscheidend ist, daß die Gewichte der Kontextschichten durch das Training nicht verändert werden, da sie bereits im Vorfeld festgelegt wurden. In die Fehlerfunktion $E(W)$ gehen die Ausgaben der Kontextneuronen mit ein, damit verändert sich Gleichung (D.5) ein klein wenig. Vor Präsentation des nächsten Trainingsmusters muß der Folgezustand der Kontextneuronen berechnet werden. Dies ist der einzige Schritt, an dem sich die Trainingsmethode vom Standardtraining neuronaler feedforward–Netze unterscheidet.

D.2.4 Zeitabhängige Backpropagation (BPTT)

Sollen Netze mit Zyklen, also nicht nur partiell rekurrente Netze trainiert werden, so ist eine andere Vorgehensweise vonnöten. M. MINSKY und S. PAPERT zeigten in [74], daß jedem rekurrenten Netz mit einer vorgegebenen Anzahl an diskreten Zeitschritten ein äquivalentes feedforward–Netz zugeordnet werden kann. Durch die

Technik des *zeitlichen Entfaltens* werden für das äquivalente feedforward–Netz alle Neuronen und Gewichte des rekurrenten Systems für jeden diskreten Zeitschritt durch eigene Neuronen ersetzt. Das resultierende Netz kann dann mit den üblichen Trainingsmethoden für feedforward–Netze trainiert werden.

D.2.5 KOHONEN–Karten

TEUVO KOHONEN entwickelte zwei Gruppen an Lernverfahren [64], die eng miteinander verwandt sind. In beiden Fällen handelt es sich um Netze mit einer einzigen Schicht aktiver Neuronen, den sogenannten KOHONEN–Neuronen.

D.2.5.1 Lernende Vektorquantisierung (LVQ)

Die lernende Vektorquantisierung ist ein überwachtes Lernverfahren, bei dem zu jedem Eingabevektor bekannt sein muß, zu welcher Klasse er gehört.

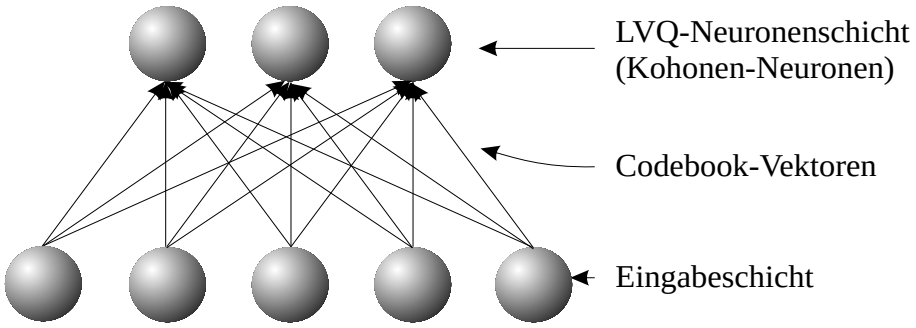

Abbildung D.6: *Struktur eines LVQ–Netzes*

Zur Initialisierung sollen die Gewichtsvektoren $W_j = (w_{1j}, \ldots, w_{nj})$ der n Eingabeneuronen möglichst repräsentativ über dem Raum der Eingabevektoren verteilt werden. KOHONEN nennt die Gewichtsvektoren auch *Codebook–Vektoren*. Das Grundprinzip der verschiedenen LVQ–Varianten besteht darin, daß ein Eingabevektor mit allen Codebook–Vektoren verglichen wird. Der Gewichtsvektor W_c des Gewinnerneurons c, das dem Eingabevektor X am ähnlichsten ist, wird noch ähnlicher dem Eingabevektor gemacht, wenn c der gleichen Klasse wie der Eingabevektor angehört, sonst unähnlicher.

$$W_c(t+1) = \begin{cases} W_c(t) + \alpha(t)[X(t) - W_c(t)], & \text{falls } \text{Klasse}(W_c) = \text{Klasse}(X) \\ W_c(t) - \alpha(t)[X(t) - W_c(t)], & \text{falls } \text{Klasse}(W_c) \neq \text{Klasse}(X) \end{cases} \tag{D.9}$$

$$W_j(t+1) = W_j(t), \quad \text{für alle } j \neq c \tag{D.10}$$

$$0 < \alpha(t) < 1 \tag{D.11}$$

$\alpha(t)$ ist der zeitabhängige Bruchteil des Differenzvektors.

D.2.5.2 Selbstorganisierende Karten (SOM)

Im Gegensatz zur LVQ sind die selbstorganisierenden Karten unüberwachte Lernverfahren, die eine selbsttätige Klassifizierung vornehmen. Zusätzlich existiert eine lokale Nachbarschaftsbeziehung der m KOHONEN–Neuronen in einem niedrigdimensionalen Gitter. Aus Darstellungsgründen wird die KOHONEN–Schicht meist in ein, zwei oder drei Dimensionen angeordnet. In der Literatur werden selbstorganisierende Karten häufig als KOHONEN–Netze bezeichnet, da sie der am häufigsten eingesetzte Netztyp von TEUVO KOHONEN sind.

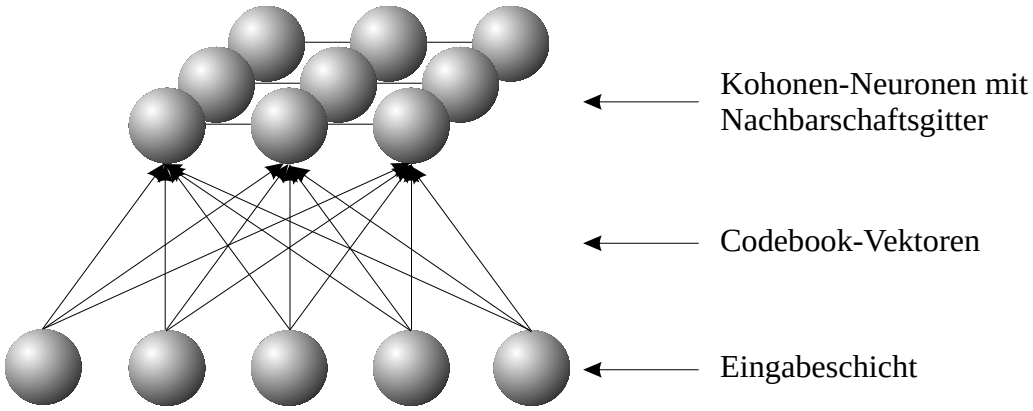

Abbildung D.7: *Struktur einer selbstorganisierenden Karte bzw. eines* KOHONEN*–Netzes*

Wie bei der LVQ sind die p Eingabeneuronen vollständig mit den m KOHONEN–Neuronen verbunden, obwohl dies Abbildung D.7 nur unvollkommen zeigt. Die Lernregel ist der der LVQ sehr ähnlich, nur daß keine Klasseninformationen vorgegeben werden müssen. Während des Trainings wird jetzt nicht nur der Gewichtsvektor des KOHONEN–Neurons verändert, dessen Gewichtsvektor dem Eingabevektor am ähnlichsten ist, sondern die Gewichtsvektoren aller Neuronen in der topologischen Nachbarschaft des Gewinnerneurons c.

$$W_j(t+1) = W_j(t) + \alpha(t) h_{cj}(t)[X(t) - W_j(t)] \tag{D.12}$$

Die topologische Nähe der Neuronen untereinander steckt in der Distanzfunktion $h_{cj}(t)$. Typischerweise werden während des Trainings sowohl der Bruchteil $\alpha(t)$ als auch die Distanzfunktion $h_{cj}(t)$ mit der Zeit t reduziert. Als Distanzfunktion wird häufig die normalisierte GAUSS–Funktion $h_{cj}(t) = \frac{1}{\sqrt{2\pi}\sigma(t)} \exp\left(-\frac{|r_c - r_j|^2}{2\sigma^2(t)}\right)$ eingesetzt.

Interessant sind SOMs in der Praxis, da sie neben dem selbstständigen Klassifizieren einen n–dimensionalen Raum auf einen m–dimensionalen Raum unter Beibehaltung der topologischen Eigenschaften abbilden können.

D.2.6 Counterpropagation

ROBERT HECHT-NIELSEN entwickelte 1987 das *Counterpropagation*–Netzwerk [50]. Counterpropagation ist ein überwachtes Lernverfahren, das aus zwei unterschiedlichen Netztypen, einer KOHONEN–Ebene und einer GROSSBERG–Ebene zusammengesetzt ist. Anwendungen des Verfahrens liegen auf dem Gebiet der Mustererkennung, der Musterklassifikation und der Mustervervollständigung.

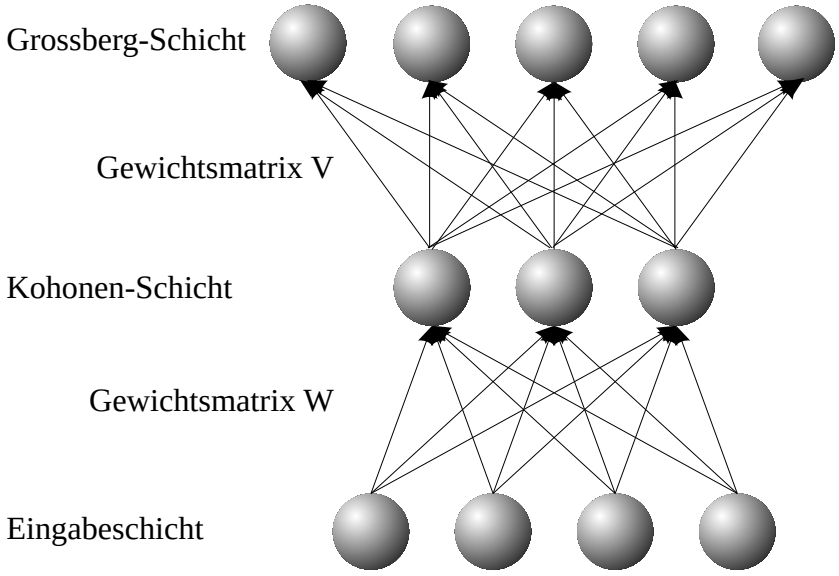

Abbildung D.8: *Vereinfachte Struktur eines Counterpropagation–Netzes*

Die KOHONEN–Schicht ist sowohl mit der Eingabeschicht als auch der GROSSBERG–Schicht vollständig verbunden. Die in Abbildung D.8 eingezeichneten Gewichtsmatrizen W und V sind trainierbar.

Beim Anlegen eines Musters reagiert nur das einzelne Neuron mit 1 in der KOHONEN–Schicht, dessen Gewichtsvektor dem Eingabemuster am ähnlichsten ist, die übrigen Neuronen antworten mit 0. Die Neuronen der GROSSBERG–Schicht besitzen als Ausgabefunktion die Identität, aber ohne *winner-takes-all*–Charakteristik. Da nur ein KOHONEN–Neuron eine Ausgabe $o_i = 1$ besitzt, liefert jedes GROSSBERG–Neuron j als Ausgabe das Gewicht v_{ij} vom aktiven KOHONEN–Neuron zu ihm selbst.

Das Training der KOHONEN–Schicht erfolgt durch unüberwachtes Training analog zu Kapitel D.2.5.2. Hierbei findet im Wesentlichen eine Clusterung der Ein-

gabemuster statt. Die GROSSBERG–Neuronen werden durch überwachtes Training optimiert. Die Gewichtsänderung Δv_{ij} ist proportional zur Ausgabe des KOHONEN–Neurons i und zur Differenz zwischen erwünschter Ausgabe und dem Verbindungsgewicht v_{ij}. Zu beachten ist dabei, daß dadurch nur Gewichte verändert werden, deren KOHONEN–Neuron eine Ausgabe ungleich Null hat.

ROBERT HECHT-NIELSEN schlug eine Erweiterung, die sog. vollständige Counterpropagation–Architektur vor, die die bemerkenswerte Fähigkeit besitzt, nicht nur die vektorwertige Abbildung $f(\mathbf{X}) = \mathbf{Y}$ zu modellieren, sondern auch die Inverse $f(\mathbf{Y}) = \mathbf{X}$ zu approximieren.

D.2.7 HOPFIELD–Netze

Der Physiker JOHN HOPFIELD entwickelte 1982 das nach ihm benannte Netz [55]. Durch die Herleitung nach Spin-Glass–Modellen konnten HOPFIELD–Netze mathematisch intensiv analysiert werden. Sie fanden außerdem Anwendung als deterministische, lokale Optimierverfahren.

D.2.7.1 Grundlagen von HOPFIELD–Netzen

HOPFIELD–Netze bestehen aus einer einzigen Schicht an Neuronen, die gleichzeitig als Eingabe- und als Ausgabeneuronen dienen. Jedes Neuron ist mit allen übrigen Neuronen rekursiv verbunden, nur nicht mit sich selbst, also gilt $w_{ii} = 0$. Alle Verbindungen zwischen zwei Neuronen sind außerdem symmetrisch, d.h. $w_{ij} = w_{ji}$.

HOPFIELD–Netze werden meist als binäre Varianten eingesetzt, existieren aber auch als kontinuierliche Versionen. Das prinzipielle Verhalten ist in beiden Fällen gleich. Binäre HOPFIELD–Netze werden wie folgt beschrieben:

$$net_j(t+1) := \sum_{i \neq j} w_{ij} o_i(t) + I_j \tag{D.13}$$

$$o_j(t+1) = f_{act}(net_j(t+1)) := \begin{cases} 1, & \text{falls } net_j(t+1) > \theta_j \\ 0, & \text{falls } net_j(t+1) < \theta_j \\ o_j(t), & \text{sonst} \end{cases} \tag{D.14}$$

I_j : Netzeingabe des j.Neurons

Jedes Neuron berechnet die gewichtete Summe aller Eingangsverbindungen. Die Ausgabe eines Neurons ist entweder 1, falls die Netzeingabe net_j größer ist als der Schwellwert θ_j und 0, falls die Eingabe kleiner als der Schwellwert ist.

Dieses System ist einem Spinglas sehr ähnlich!

In der Literatur wurden zwei Progagierungsregeln genauer studiert:

1. Asynchrone Aktivierung mit zufälliger Auswahl:
   Jedes Neuron berechnet seinen neuen Zustand nach einer zufälligen diskreten Anzahl von Zeitschritten. Zu jedem Zeitpunkt ändert nur ein Neuron seine Aktivierung. Dies erleichtert die theoretische Analyse.

2. Synchrone Aktivierung:
   Alle Neuronen ändern ihren Zustand gleichzeitig.

Für HOPFIELD–Netze gilt das COHEN-GROSSBERG–Theorem:

Rekurrente Netze sind stabil,
d.h. die Änderungen des Systems nehmen kontinuierlich ab,
wenn die Gewichtsmatrix W symmetrisch und die Hauptdiagonale leer ist.

Der Beweis erfolgt mit Hilfe einer positiven LIAPUNOV–Energiefunktion

$$E(T) := -\frac{1}{2}\sum_i\sum_j w_{ij}o_i(t)o_j(t) - \sum_j I_j o_j(t) + \sum_j \theta_j o_j(t) \qquad \text{(D.15)}$$

die sich bei jedem Zeitschritt des rekurrenten Netzes verringert.

D.2.7.2 BOLTZMANN–Maschinen

Sollen HOPFIELD–Netze zu Optimierzwecken eingesetzt werden, so muß das Problem in eine Energiefunktion analog zu Gleichung (D.15) umgeformt werden. Die Gewichte w_{ij} des Netzes werden durch Koeffizientenvergleich mit Gleichung (D.15) festgelegt.

Das *Startmuster* ist äquivalent zum Eingabemuster des Netzes, das sich während der gesamten Laufzeit nicht ändert. Das *lokale Optimum* ist das Ausgabemuster des Netzes, nachdem es konvergiert ist.

JOHN HOPFIELD und DAVID TANK fanden beispielsweise einen Weg [56], mit speziellen HOPFIELD–Netzen Näherungslösungen für *Travelling-Salesman*–Probleme zu finden.

Ein Problem bei HOPFIELD–Netzen ist die starke Neigung, sich in lokalen Optima zu stabilisieren. Aus diesem Grund wurde die BOLTZMANN–Maschine eingeführt.

Sie unterscheidet sich von HOPFIELD–Netzen durch eine stochastische Aktivierungsfunktion und eine unterschiedliche Lernregel. Außerdem können noch verdeckte Neuronen eingefügt werden, die von außen unsichtbar sind.

Die stochastische Aktivierungsfunktion eines Neurons der BOLTZMANN–Maschine lautet wie folgt:

$$o_j = f_{act}(net_j) := \begin{cases} 1, & \text{falls } \text{rand}() \leq \dfrac{1}{1+\exp\left(-\frac{\Delta E_j}{T}\right)} \\ 0, & \text{sonst} \end{cases} \tag{D.16}$$

ΔE_j ist die Energiedifferenz des Netzes zwischen den beiden Zuständen, bei denen Neuron j den Wert 0 oder 1 angenommen hat, wobei die anderen Neuronen ihre Ausgabe unverändert beibehalten haben. T ist ein künstlicher Temperaturparameter, der von hohen Werten beginnend langsam erniedrigt wird. Die Funktion `rand()` liefert Pseudo-Zufallszahlen.

Die relative Wahrscheinlichkeit zweier Zustände α und β des Netzwerks genügt der BOLTZMANN–Verteilung:

$$\frac{P_\alpha}{P_\beta} = e^{-\frac{E_\alpha - E_\beta}{T}} \tag{D.17}$$

Die BOLTZMANN–Maschine zeigt auf diese Weise dasselbe globale Verhalten wie ein HOPFIELD–Netz, konvergiert aber bei geeignetem Verlauf der Temperatur $T(t)$ schneller zu besseren Optima.

Das Lernverfahren der BOLTZMANN–Maschine ist eine Variante des *Simulated Annealing*. Ziel des Lernverfahrens ist es, die Gewichte so anzupassen, daß das Netz genau die gleiche Wahrscheinlichkeitsverteilung der Zustände auf den Eingabe/Ausgabe–Neuronen erreicht, wie im Temperaturgleichgewicht ohne externe Eingaben. Die Lernregel lautet (ohne Herleitung):

$$\Delta w_{ij} = \eta(p_{ij} - \tilde{p}_{ij}) \tag{D.18}$$

p_{ij} ist die mittlere Wahrscheinlichkeit, daß beide Neuronen i und j bei konstanter externer Eingabe gleichzeitig aktiv sind. $\tilde{p}_{ij}$ ist die entsprechende Wahrscheinlichkeit, wenn keine externe Eingabe anliegt.

D.2.8 Adaptive Resonance Theory (ART)

Adaptive Resonance Theory ist eine Familie von Modellen neuronaler Netze, die von GAIL CARPENTER und STEPHEN GROSSBERG ab Mitte der 70er Jahre an der Boston University entwickelt wurden. Ihre Aufgabe ist die Musterklassifikation.

Die Lösung des „*Stabilitäts-Plastizitäts*–Dilemmas“ neuronaler Netze war die ursprüngliche Intention zur Entwicklung von ART. Allen ART–Architekturen ist gemeinsam, daß sie alte Assoziationen nicht mehr vergessen (Stabilität), aber trotzdem flexibel genug sind, neue Muster lernen zu können (Plastizität).

Die bekanntesten Vertreter von ART–Architekturen sind:

- ART-1: ursprüngliche Version für binäre Eingabevektoren [46]
- ART-2: Erweiterung von ART-1 auf kontinuierliche Eingabevektoren [47]
- ART-3: Erweiterung von ART-2 zur Modellierung der Vorgänge von Synapsen
- ARTMAP: Kombination zweier ART–Netze mit überwachtem Lernverfahren
- FUZZY ART: Kombination von ART und Fuzzy Logic

Die prinzipielle Arbeitsweise eines ART–Netzes verläuft nach folgendem Schema:

1. Nach Anlegen des Eingabemusters versucht das Netz, ihn einer der vorhandenen Klassen zuzuordnen. Die Zuordnung erfolgt durch die Ähnlichkeit mit gespeicherten Mustern.
2. Falls der Eingabevektor keiner gespeicherten Klasse ähnelt, wird eine neue Klasse durch Speichern eines dem Eingabevektor ähnlichen Musters erzeugt.
3. Falls ein Muster gefunden wird, das dem Eingabemuster innerhalb einer gegebenen Toleranz ähnelt, wird dieses leicht modifiziert, um es dem Eingabevektor ähnlicher zu machen.
4. Muster, die dem Eingabevektor nicht ähnlich sind, bleiben unverändert.

Die Architekturen der verschiedenen ART–Netze sind kompliziert und werden selten in der Praxis eingesetzt. Deshalb wäre eine weitergehende Darstellung aus Platzgründen hier nicht angemessen.

Literaturverzeichnis

[1] A.W. Andrus. Evaluation and isolating synthetic oligonucleotides. *Applied Biosystems*, User Bulletin 13, 1992.

[2] A.P. Arkin and Douglas C. Youvan. Optimizing nucleotide mixtures to encode specific subsets of amino acids for semi-random mutagenesis. *Biotechnology-N-Y*, 10(3):297–300, 1992.

[3] A. Babajide, I.L. Hofacker, M.H. Sippl, and P.F. Stadler. Neutral networks in protein space: a computational study based on knowledge-based potentials of mean force. *Fold-Des.*, 2(5):261–269, 1997.

[4] R.F. Balint and J.W. Larrick. Antibody engineering by parsimonious mutagenesis. *Gene*, 137(1):109–118, 1993.

[5] F. Bonekamp and K.F. Jensen. The agg codon is translated slowly in e. coli even at very low expression levels. *Nucleic Acids Res.*, 16(7):3013–3024, 1988.

[6] U. Brinkmann, R.E. Mattes, and P. Buckel. High-level expression of recombinant genes in escherichia coli is dependent on the availability of the dnay gene product. *Gene*, 85(1):109–114, 1989.

[7] T. Cacoullos. Estimation of a multivariate density. *Ann. Inst. Statist. Math.*, 18(2):179–189, 1966.

[8] Miguel Catasus, Wayne Branagh, and Eric D. Salin. Improved calibration for inductively coupled plasma-atomic emission spectrometry using generalized regression neural networks. *Applied Spectroscopy*, 49(6):798–807, 1995.

[9] Maureen Caudill. *Neural Network Primer*. CA: Miller Freeman Inc., 1990.

[10] Maureen Caudill. Grnn and bear it. *AI Expert*, 8(5):28–33, May 1993.

[11] J.B. Chattopadhyaya and C.B. Reese. Chemical synthesis of tridecanucleoside dodecaphosphate sequence of sv40 dna. *Nucl. Acids Res.*, 8(9):2039–2053, 1980.

[12] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[13] G.B. Dantzig, D.R. Fulkerson, and S.M. Johnson. Solution of a large-scale travelling-salesman problem. *Oper. Res.*, 2:393–410, 1954.

[14] Frank Darius and Raúl Rojas. „Simulated Molecular Evolution“ or Computer-Generated Artifacts? *Biophysical J.*, 67:2120–2122, November 1994.

[15] Charles R. Darwin. *On the Origin of Species by Means of Natural Selection.* John Murray, London, 1859.

[16] Charles R. Darwin. *Über die Entstehung der Arten.* Reclam, Stuttgart, 1963. in german.

[17] Charles R. Darwin. *Die Abstammung des Menschen.* Franz Spiegel Buch GmbH, Ulm, 1986. ISBN 3-925037-03-9, in german.

[18] A.R. Davidson, K.J. Lumb, and R.T. Sauer. Cooperatively folded proteins in random sequence libraries. *Nat. Struct. Biol.*, 2:856–864, 1995.

[19] A.R. Davidson and R.T. Sauer. Folded proteins occur frequently in libraries of random amino acid sequences. *Proc Natl Acad Sci USA*, 91(6):2146–2150, March 1994.

[20] Richard Dawkins. *The Selfish Gene.* Oxford University Press, 1976.

[21] Richard Dawkins. *Das egoistische Gen.* Spektrum Akademischer Verlag GmbH, Heidelberg, second edition, 1994. ISBN 3-499-19609-3, in german.

[22] S. Delagrave, E.R. Goldman, and Douglas C. Youvan. Recursive ensemble mutagenesis. *Protein-Eng.*, 6(3):327–331, 1993.

[23] K.M. Derbyshire, J.J. Salvo, and N.D.F. Grindley. A simple and efficient procedure for saturation mutagenesis using mixed oligodeoxynucleotides. *Gene*, 46:145–152, 1986.

[24] Rüdiger Dietrich, Frank Wirsching, Thomas Opitz, and Andreas Schwienhorst. Gene assembly based on blunt-ended double-stranded DNA-modules. *Biotechnology Techniques*, 12(1):49–54, 1998.

[25] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics — Part B*, 26(1):1–13, 1996.

[26] Gunter Dueck and Tobias Scheuer. A general purpose optimization algorithm appearing superior to simulated annealing. Technical Report TR 88.10.011, IBM Scientific Center, Heidelberg, Germany, October 1988.

[27] Manfred Eigen, John McCaskill, and Peter Schuster. Molecular quasispecies. *J.Phys.Chem.*, 92:6881–6891, 1988.

[28] Manfred Eigen, John McCaskill, and Peter Schuster. The molecular quasispecies. *Adv.Chem.Phys.*, 75:149–263, 1989.

[29] Manfred Eigen and Peter Schuster. The hypercycle. A principle of natural self-organization. Part A: Emergence of the hypercycle. *Naturwissenschaften*, 64:541–565, 1977.

[30] Manfred Eigen and Peter Schuster. The hypercycle. A principle of natural self-organization. Part B: The abstract hypercycle. *Naturwissenschaften*, 65:7–41, 1978.

[31] Manfred Eigen and Peter Schuster. The hypercycle. A principle of natural self-organization. Part C: The realistic hypercycle. *Naturwissenschaften*, 65:341–369, 1978.

[32] J.L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[33] J.F. Ernst and E. Kawashima. Variations in codon usage are not correlated with heterologous gene expression in saccharomyces cerevisiae and escherichia coli. *J. Biotechnol.*, 8:1–10, 1988.

[34] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. In D.S. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proc. of the 1988 Connectionist Models Summer School (June 17-26 1988)*. Carnegie Mellon University, Morgan Kaufmann Publishers, Inc., 1989.

[35] Scott E. Fahlman. The recurrent cascade-correlation architecture. In R.P. Lippman, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 190–198. Morgan Kaufmann Publishers, Inc., 1991.

[36] Scott E. Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan Kaufmann Publishers, Inc., 1990.

[37] R.A. Fisher. *The Genetical Theory of Natural Selection.* Clarendon Press, Oxford, 1930.

[38] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, 1975.

[39] Paul Geladi and Bruce R. Kowalski. Partial least-squares regression: A tutorial. *Analytica Chimica Acta*, 185:1–17, 1986.

[40] S. Glaser, D. Yelton, and W.D. Huse. Antibody engineering by codon-based mutagenesis in a filamentous phage vector system. *J. Immunol.*, 149(12):3903–3913, 1992.

[41] F. Glover. Tabu Search — Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[42] F. Glover. Tabu Search — Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[43] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

[44] D.E. Goldberg. Real-coded genetic algorithms, virtual alphabets and blocking. *Complex Systems*, 5:139–160, 1991.

[45] E.R. Goldman and Douglas C. Youvan. An algorithmically optimized combinatorial library screened by digital imaging spectroscopy. *Biotechnology-N-Y*, 10(12):1557–1561, 1992.

[46] Stephen Grossberg. Adaptive pattern classification and universal recoding: I. parallel developement and coding of neural feature detectors. *Biological Cybernetics*, 23:121–134, 1976.

[47] Stephen Grossberg and Gail A. Carpenter. Art-2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930, 1987.

[48] W.D. Hamilton. The genetical evolution of social behaviour I. *Journal of Theoretical Biology*, 7:1–16, 1964.

[49] W.D. Hamilton. The genetical evolution of social behaviour II. *Journal of Theoretical Biology*, 7:17–52, 1964.

[50] Robert Hecht-Nielsen. Counterpropagation networks. In M. Caudill and C. Butler, editors, *Proc. of the IEEE First Int. Conference on Neural Networks*, volume 2, pages 19–32. San Diego, CA, 1987.

[51] J.D. Hermes, S.M. Parekh, S.C. Blacklow, H. Köster, and J.R. Knowles. A reliable method for random mutagenesis: the generation of mutant libraries using spiked oligodeoxyribonucleotide primers. *Gene*, 84(1):143–151, 1989.

[52] Ivo L. Hofacker, Walter Fontana, Peter Stadler, L.S. Bonhoeffer, M. Tacker, and Peter Schuster. Vienna RNA Package. Anonymous FTP: `/pub/RNA/ViennaRNA-1.21/` on `ftp.itc.univie.ac.at` (Public Domain Software).

[53] F. Hoffmeister and T. Bäck. Genetic algorithms and evolution strategies: Similarities and differences. Technical Report Sys–1/92, University Dortmund, Germany, 1992.

[54] J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Harbor, 1975.

[55] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proc. of the National Academy of Sciences*, volume 79, pages 2554–2558, 1982.

[56] John J. Hopfield and David W. Tank. „Neural“ computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[57] K.M. Hornik, M. Stinchcombe, and H. White. Multi-layer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[58] B.H. Horwitz and D. DiMaio. Saturation mutagenesis using mixed oligonucleotides and m13 templates containing uracil. *Methods in Enzymology*, 185:599–611, 1990.

[59] Jenq-Neng Hwang, Shih-Shien You, Shyh-Rong Lay, and I-Chang Jou. What's wrong with a cascaded correlation learning network: a projection pursuit learning perspective. Technical report, Dept of Electrical Engineering, University of Washington, 1994.

[60] C.L. Jellis, T.J. Cradick, P. Rennert, P. Salinas, J. Boyd, T. Amirault, and G.S. Gray. Defining critical residues in the epitope for a hiv-neutralizing monoclonal antibody using phage display and peptide array technologies. *Gene*, 137(1):63–68, 1993.

[61] M.I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proc. of the Eighth Annual Conference of the Cognitive Science Society*, pages 531–546. Erlbaum, Hillsdale NJ, 1986.

[62] Motoo Kimura. *The Neutral Theory of Molecular Evolution.* Cambridge University Press, UK, 1983.

[63] Scott Kirkpatrick, Charles D. Gelatt jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[64] Teuvo Kohonen. *Self-Organization and Associative Memory.* Springer Series in Information Sciences. Springer-Verlag, third edition, 1989.

[65] K.S. Lam, V.J. Hruby, M. Lebl, R.J. Knapp, W.M. Kazmierski, E.M. Hersh, and S.E. Salmon. The chemical synthesis of large random peptide libraries and their use for the discovery of ligands for macromolecular acceptors. *Bioorg. Med. Chem. Lett.*, 3:419–424, 1993.

[66] K.S. Lam, S.E. Salmon, E.M. Hersh, V.J. Hruby, W.M. Kazmierski, and R.J. Knapp. A new type of synthetic peptide library for identifying ligand-binding activity. *Nature*, 354(6348):82–84, 1991. published errata appear in Nature 1992 Jul 30;358(6385):434 and 1992 Dec 24-31;360(6406):768.

[67] K. Lang and M. Whitbrock. Learning to tell two spirals apart. In D.S. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proc. of the 1988 Connectionist Models Summer School (June 17-26 1988)*, pages 52–59. Carnegie Mellon University, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1989.

[68] J.W. Little. Saturation mutagenesis of specific codons: elimination of molecules with stop codons from mixed pools of dna. *Gene*, 88(1):113–115, 1990.

[69] M.H. Lyttle, E.W. Napolitano, B.L. Calio, and L.M. Kauvar. Mutagenesis using trinucleotide beta-cyanoethyl phosphoramidites. *BioTechniques*, 19(2):274–281, 1995.

[70] L. Makowski. Phage display: structure, assembly and engineering of filamentous bacteriophage m13. *Current Op. Struct. Biol.*, 4:225–230, 1994.

[71] Timothy Masters. *Advanced Algorithms for Neural Networks.* John Wiley: New York, 1995.

[72] M.D. Matteucci and H.L. Heyneker. Targeted random mutagenesis: the use of ambiguously synthesized oligonucleotides to mutagenize sequences immediately 5' of an atg initiation codon. *Nucl. Acids Res.*, 11(10):3113–3121, 1983.

[73] Warren McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[74] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[75] Patrick M. Murphy and David W. Aha. UCI machine learning database archive. Anonymous FTP: `/pub/machine-learning-databases/` on `ics.uci.edu`.

[76] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[77] S.S. Ner, T.C. Atkinson, and M. Smith. A method for introducing random single point deletions in specific dna target sequences using oligonucleotides. *Nucl. Acids Res.*, 17(11):4015–4023, 1989.

[78] A. Ono, A. Matsuda, J. Zhao, and D.V. Santi. The synthesis of blocked triplet-phosphoramidites and their use in mutagenesis. *Nucl. Acids Res.*, 23(22):4677–4682, 1995.

[79] Károly F. Pál. Genetic algorithm with local optimization. *Biol. Cybern.*, 73:335–341, 1995.

[80] R. Palmer. Optimization on rugged landscapes. In A.S. Perelson and S.A. Kauffman, editors, *Molecular Evolution on Rugged Landscapes: Proteins, RNA*

and the Immune System, pages 3–25, Redwood City, CA, 1991. Addison Wesley.

[81] E. Parzen. On estimation of a probability density function and mode. *Ann. Math. Statist.*, 33:1065–1076, 1962.

[82] T. Poggio and F. Girosi. A theory of networks for approximation and learning. MIT AI Memo No. 1140, 1989.

[83] Lutz Prechelt. PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany, September 1994. Anonymous FTP: `/pub/papers/techreports/1994/1994-21.ps.Z` on `ftp.ira.uka.de`.

[84] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian Flannery. *Numerical Recipes in C — the Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[85] N. Quian and T.J. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884, 1988.

[86] Ingo Rechenberg. *Evolutionsstrategie.* Friedrich Frommann Verlag, Stuttgart, 1973. in german.

[87] M. Robinson, R. Lilley, S. Little, J.S. Emtage, G. Yarranton, P. Stephens, A. Millican, M. Eaton, and G. Humphreys. Codon usage can affect efficiency of translation of genes in escherichia coli. *Nucleic Acids Res.*, 12(17):6663–6671, 1984.

[88] D.S. Rokhsar, P.W. Anderson, and D.L. Stein. Selforganization in prebiological systems: Simulations of a model for the origin of genetic information. *J. Mol. Evol.*, 23:119–126, 1986.

[89] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65:386–408, 1958.

[90] Frank Rosenblatt. *Prinziples of Neurodynamics.* Spartan Books, New York, 1962.

[91] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[92] P.M. Schenk, S. Baumann, R. Mattes, and H.H. Steinbiss. Improved high-level expression system for eukaryotic genes in escherichia coli using t7 rna polymerase and rare argtrnas. *BioTechniques*, 19(2):196–200, 1995.

[93] Gisbert Schneider and Paul Wrede. The rational design of amino acid sequences by artificial neural networks and simulated molecular evolution: *de novo* design of an idealized leader peptidase cleavage site. *Biophysical J.*, 66:335–344, February 1994.

[94] Gisbert Schneider, Paul Wrede, O. Landt, S. Klages, A. Fatemi, and U. Hahn. Peptide design aided by neural networks: biological activity of artificial signal peptidase I cleavage site. *Biochemistry*, 37(11):3588–3593, March 1998.

[95] A. Schober, J.M Köhler, R. Günther, M. Thürk, A. Schwienhorst, M. Eigen, and M. Döhring. Parallele Führung von Reaktionen. Patent Application PCT/EP 94/02173 02.07.1993.

[96] A. Schober, G. Schlingloff, A. Thamm, D. Tomandl, M. Gebinoga, H.J. Kiel, Ch. Scheffler, M. Döring, J.M. Köhler, and G. Mayer. Systemintegration of microsystems/chip elements in miniaturized automata for high-throughput synthesis and screening in biology, biochemistry and chemistry. *Microsystems*, 1997.

[97] A. Schober, G. Schlingloff, A. Thamm, D. Vetter, D. Tomandl, M. Gebinoga, H.J. Kiel, Ch. Scheffler, M. Döring, J.M. Köhler, and G. Mayer. Systemintegration of microsystems/chip elements in miniaturized automata for high-throughput synthesis and screening in biology, biochemistry and chemistry. *Microsystems*, 1996.

[98] A. Schober, G. Schlingloff, D. Tomandl, J.M. Köhler, G. Mayer, A. Groß, St. Wuchty, H. Deppe, B. Diefenbach, and H. Wurziger. Chemical and biochemical synthesis and screening in silico. *Microsystems*, 1998.

[99] Andreas Schober, Marcel Thürck, and Manfred Eigen. Optimization by hierarchical mutant production. *Biological Cybernetics*, 69:493–501, 1993.

[100] Eberhard Schöneburg, Frank Heinzmann, and Sven Feddersen. *Genetische Algorithmen und Evolutionsstrategien.* Addison Wesley, 1994. ISBN 3-89319-493-2, in german.

[101] Peter Schuster. Genotypes with phenotypes: Adventures in an RNA toy world. *Biophys. Chem.*, 66:75–110, 1997. Also published as: Preprint No. 97-04-036, Santa Fe Institute, Santa Fe, NM 1997.

[102] Hans-Paul Schwefel. *Evolutionsstrategie und numerische Optimierung.* PhD thesis, Technische Universität Berlin, Cambridge, MA, May 1975. in german.

[103] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels Evolutionsstrategien.* Birkhäuser Verlag, Basel, 1977. in german.

[104] Andreas Schwienhorst. Evolutive Methoden im Enzymdesign. *Biospektrum*, 2:44–48, 1998. in german.

[105] F. Sebestén, G. Dibó, A. Kovács, and A. Furka. Chemical synthesis of peptide libraries. *Bioorg. Med. Chem. Lett.*, 3:413–418, 1993.

[106] D.P. Siderovski and T.W. Mak. Ramha: a pc-based monte-carlo simulation of random saturation mutagenesis. *Comput. Biol. Med.*, 23(6):463–474, 1993.

[107] Karl Sigmund. *Spielpläne.* Droemersche Verlagsanstalt Th. Knaur Nachf., München, 1997. ISBN 3-426-77270-1, in german.

[108] Jasbir Singh, Mark A. Ator, Edward P. Jaeger, Martin P. Allen, David A. Whipple, James E. Soloweij, Swapan Chowdhary, and Adi M. Treasurywala. Application of genetic algorithms to combinatorial synthesis: a computational approach to lead identification and lead optimization. *J. Am. Chem. Soc.*, 118:1669–1676, 1996.

[109] M.J. Sippl, S. Weitckus, and H. Flöckner. In search of protein folds. In K. Merz Jr. and S. LeGrand, editors, *The protein folding problem and tertiary structure prediction.* Birkhäuser Publ., Boston, 1994.

[110] G.P. Smith. Filamentous fusion phage: novel expression vectors that display cloned antigens on the virion surface. *Science*, 228(4705):1315–1317, 1985.

[111] John Maynard Smith. Game theory and the evolution of fighting. In John Maynard Smith, editor, *On Evolution*, pages 8–28. Edinburgh University Press, 1972.

[112] J. Sondek and D. Shortle. A general strategy for random insertion and substitution mutagenesis: substoichiometric coupling of trinucleotide phosphoramidites. *Proc. Natl. Acad. Sci. USA*, 89(8):3581–3585, 1992.

[113] R.A. Spanjaard and J. van Duin. Translation of the sequence agg-agg yields 50 *Proc. Natl. Acad. Sci. U.S.A.*, 85(21):7967–7971, 1988.

[114] Donald F. Specht. A general regression neural network. *IEEE Transactions on Neural Networks*, 2(6):568–576, November 1991.

[115] Donald F. Specht. The general regression neural network — rediscovered. *Neural Networks*, 6:1033–1034, 1993.

[116] Donald F. Specht and Harlan Romsdahl. Experience with adaptive probabilistic neural networks and adaptive general regression neural networks. *Proceedings of the IEEE World Congress on Computational Intelligence*, 2:1203–1208, 1994.

[117] Donald F. Specht and Philip D. Shapiro. Generalizational accuracy of probabilistic neural networks compared with back-propagation networks. *Proc. of the International Joint Conference on Neural Networks*, pages 887–892, 1991.

[118] Daniel Stein. Spingläser. *Spektrum der Wissenschaft*, pages 102–108, 1989. in german.

[119] William P.C. Stemmer. Searching sequence space. *Bio/Technology*, 13:549–553, 1995.

[120] Marcel Thürk. private communication, 1994.

[121] Dirk Tomandl, Andreas Schober, and Andreas Schwienhorst. Optimizing doped libraries by using genetic algorithms. *Journal of Computer-aided Molecular Design*, 11:29–38, 1997.

[122] Venkat Venkatasubramanian, King Chan, and James M. Caruthers. Evolutionary design of molecules with desired properties using the genetic algorithm. *J. Chem. Inf. Comput. Sci.*, 35:188–195, 1995.

[123] Venkat Venkatasubramanian, Anantha Sundaram, King Chan, and James M. Caruthers. Computer-aided molecular design using neural networks and genetic algorithms. In James Devillers, editor, *Genetic Algorithms in Molecular Modelling*, ISBN: 0122138104, 1996. Carfax Publishing Company, UK.

[124] B. Virnekäs, L. Ge, A. Plückthun, K.C. Schneider, G. Wellnhofer, and S.E. Moroney. Trinucleotide phosphoramidites: ideal reagents for the synthesis of mixed oligonucleotides for random mutagenesis. *Nucl. Acids Res.*, 22(25):5600–5607, 1994.

[125] A. Waibel and J. Hampshire. Building blocks for speech. *BYTE*, pages 235–242, August 1989.

[126] Lutz Weber, Sabine Wallbaum, Clemens Broger, and Klaus Gubernator. *Angew. Chem. Int. Ed. Engl.*, 34:2280–2282, 1995.

[127] Lutz Weber, Sabine Wallbaum, Clemens Broger, and Klaus Gubernator. Optimierung der biologischen Aktivität von kombinatorischen Verbindungsbibliotheken durch einen genetischen Algorithmus. *Angew. Chem.*, 107(20):2452–2454, 1995. in german.

[128] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA, 1974.

[129] G.C. Williams. *Adaptation and Natural Selection*. Princeton University Press, Princeton, 1966.

[130] Frank Wirsching, Cornelia Luge, Thomas Opitz, Dirk Tomandl, and Andreas Schwienhorst. Selection of pepsin resistant hirudin variants from phage libraries. 1999. to be published.

[131] Frank Wirsching, Thomas Opitz, Rüdiger Dietrich, and Andreas Schwienhorst. Display of functional thrombin inhibitor hirudin on the surface of phage m13. *Gene*, 204:177–184, 1997.

[132] Andreas Zell. *Simulation neuronaler Netze*. Addison-Wesley, ISBN 3-89319-554-8, Germany, 1994. in german.

[133] G. Zon, K.A. Gallo, C.J. Samson, K.L. Shao, M.F. Summers, and R.A. Byrd. Analytical studies of 'mixed sequence' oligodeoxyribonucleotides synthesized by competitive coupling of either methyl- or beta-cyanoethyl-n,n-diisopropylamino phosphoramidite reagents, including 2'-deoxyinosine. *Nucl. Acids Res.*, 13(22):8181–8196, 1985.

Lebenslauf

Persönliche Daten:

Name:	Dirk Tomandl
Adresse:	Ammerbach Str.87, 64372 Ober-Ramstadt
Geburtsdatum:	07.11.1965
Geburtsort:	Würzburg (Deutschland)
Schulbildung:	5 Jahre Grundschule, 10 Jahre Gymnasium, Abitur 1986

Nach dem 20-monatigen Zivildienst begann die

Akademische Ausbildung:

SS 1988 – WS 1988	Beginn des Studiums der Politischen Wissenschaften mit Nebenfächern Philosophie und Psychologie
WS 1988 – SS 1994	Studium Diplom-Physik mit Nebenfach Biophysik. Zusatzscheine in Evolution und Numerischer Mathematik. Theoretische Diplomarbeit in der Atomphysik („Untersuchungen der Feldionisation wasserstoffartiger Ionen im Zusammenhang mit dem Erlanger RYDFISS-Experiment“: Note 1,00).
November 1994	Diplomprüfung — Gesamtnote 1,95
01.12.1994 – 31.12.1997	Anstellung als wissenschaftlicher Mitarbeiter am IMB Jena mit dem Ziel der Promotion bei Prof. Dr. Peter Schuster.
01.01.1998 – heute	Anstellung als wissenschaftlicher Mitarbeiter am IPHT Jena mit dem Ziel der Promotion bei Prof. Dr. Peter Schuster.

Publikationen:

Dirk Tomandl, Andreas Schober, and Andreas Schwienhorst. Optimizing doped libraries by using genetic algorithms. *Journal of Computer-aided Molecular Design*, 11:29–38, 1997.

A. Schober, G. Schlingloff, A. Thamm, D. Vetter, D. Tomandl, M. Gebinoga, H.J. Kiel, Ch. Scheffler, M. Döring, J.M. Köhler, and G. Mayer. Systemintegration of microsystems/chip elements in miniaturized automata for high-throughput synthesis and screening in biology, biochemistry and chemistry. *Microsystems*, 1996.

A. Schober, G. Schlingloff, A. Thamm, D. Tomandl, M. Gebinoga, H.J. Kiel, Ch. Scheffler, M. Döring, J.M. Köhler, and G. Mayer. Systemintegration of microsystems/chip elements in miniaturized automata for high-throughput synthesis and screening in biology, biochemistry and chemistry. *Microsystems*, 1997.

A. Schober, G. Schlingloff, D. Tomandl, J.M. Köhler, G. Mayer, A. Groß, St. Wuchty, H. Deppe, B. Diefenbach, and H. Wurziger. Chemical and biochemical synthesis and screening in silico. *Microsystems*, 1998.