Automated Optimization of Dynamic Neural Network Structure using Genetic Algorithms

Christiane Sandner, Johannes Günther, Klaus Diepold

Technical Report

Automated Optimization of Dynamic Neural Network Structure using Genetic Algorithms

Christiane Sandner, Johannes Günther, Klaus Diepold

14. September 2017

Institute for Data Processing
Technische Universität München

Christiane Sandner, Johannes Günther, Klaus Diepold. *Automated Optimization of Dynamic Neural Network Structure using Genetic Algorithms.* Technical Report, Technische Universität München, Munich, Germany, 2017.

Supervised by Prof. Dr.-Ing. K. Diepold ; submitted on 14. September 2017 to the Department of Electrical and Computer Engineering of the Technische Universität München.

Institute for Data Processing, Technische Universität München, 80290 München, Germany, `http://www.ldv.ei.tum.de`.

Zusammenfassung

Diese Arbeit beschreibt, wie die Architektur dynamischer neuronaler Netze mit Hilfe eines genetischen Algorithmus optimiert werden kann. Die Netze werden darauf optimiert, die Parameter eines zeitabhängigen PID-Reglers zu berechnen und sind Teil des Regelkreises.

Die Codierung der Netze im genetischen Algorithmus erfolgt durch eine Adjazenzmatrix, welche sowohl verzögerungsfreie, als auch verzögerte Verbindungen der Netze speichern kann. Der Algorithmus verändert diese Matrizen mit Hilfe genetischer Operatoren und bildet so neue Netze. Die Qualität der Netze wird mit einer Fitnessfunktion bestimmt, welche durch die gewichtete Summe aus dem Regelfehler, sowie der Anzahl der Verbindungen im Netz definiert ist. Ziel des Algorithmus ist es, die Fitnessfunktion zu minimieren, um einen geringen Regelfehler, sowie eine möglichst kurze Simulationszeit für das Netz im PID-Regler zu erhalten.

Die Implementierung des Algorithmus erfolgte in Python. Die hohe Laufzeit des Verfahrens konnte durch die Verwendung von Multiprossesing und eines just-in-time Compilers erheblich reduziert werden.

Das Verfahren wurde unter Verwendung zweier verschiedener Gewichtungsfaktoren der Fitnessfunktion getestet, zusätzlich wurden zwei unterschiedliche Selektionsmethoden verwendet. Bei der Auswertung zeigte sich bei keinem der Versuche eine signifikante Verbesserung der Fitness des jeweils besten Individuums einer Generation. Zudem konvergierte die durchschnittliche Fitness aller Individuen innerhalb weniger Generationen zu einem Optimum. Der zeitliche Verlauf der Ist- und Sollwertkurven des optimierten Systems zeigte, dass auch bei geringem gemessenen Fehler das System stark zu Schwingungen neigte.

Abstract

This work describes the optimisation of the architecture of dynamic neural networks using a genetic algorithm. The networks are included in a control loop and learn to determine the parameters of a time variant PID controller.

In the genetic algorithm, the networks are represented by adjacency matrices, enabling the representation of connections with and without delays. By means of genetic operators, the genetic algorithm changes the elements of the matrices to produce new individuums. The quality of a network is determined by a fitness function, which is realized as a weighted sum of the control error and the number of connections in the network. The algorithm aims to minimize the fitness function in order to achieve a small control error as well as a low simulation time of the networks.

The method is tested with two different sets of weight factors for the fitness function. Additionally, two selection methods are used. Evaluation of the fitness of the best individuum of all generations showed that in none of the experiments, an appreciable improvement of the fitness could be achieved. Furthermore, the mean value of the fitness of all individuums showed a fast convergence to the optimum. By evaluating the output of the controlled system over time using the best individuums produced by the genetic algorithm, it was determined that the control error is not a suitable metric to grade the quality of a network. Networks rated with a better fitness caused an oscillation of the system's state in most cases. The entire implementation was done in python. The long runtime of the algorithm was improved significantly by using multiprocessing and a just-in-time compiler.

Contents

Symbolverzeichnis

A_i	Querschnittsfläche eines Tanks *i* des Zweitank-Systems
A_{oi}	Querschnittsfläche der Öffnung eines Tanks *i* des Zweitank-Systems
e	Momentaner Regelfehler eines geschlossenen Regelkreises
E	Fehlerfunktion des adaptiven PID-Reglers
$\hat{E}$	Fehlerfunktion eines neuronalen Netzes
$F(\hat{x})$	Fitnessfunktion eines Individuums $\hat{x}$ im genetischen Algorithmus
g	Gravitationskonstante
$\boldsymbol{h}$	Hilfsvektor zur Bestimmung von ε bei der numerischen Berechnung der Jacobimatrix
$\boldsymbol{J}$	Jacobimatrix
$\hat{\boldsymbol{J}}$	Numerische Approximation der Jacobimatrix $\boldsymbol{J}$
k_P	Pumpkonstante des Zweitank-Systems
K_p, K_i, K_d	Parameter des proportionalen, integralen und differentialen Teil eines PID-Reglers
n_i	Neuron *i* eines neuronalen Netzes
$p(\hat{x})$	Selektionswahrscheinlichkeit eines Individuums $\hat{x}$
$\boldsymbol{P}_i$	Population einer Generation *i* im genetischen Algorithmus
u	Stellgröße eines geschlossenen Regelkreises
v	Systemzustand im geschlossenen Regelkreis
$\boldsymbol{V}(\hat{x})$	Verbindungsmatrix eines Individuums $\hat{x}$
w	Sollgröße eines geschlossenen Regelkreises
$\boldsymbol{x}$	Eingangsvektor eines neuronalen Netzes
$\hat{x}$	Individuum einer Population $\boldsymbol{P}$ im genetischen Algorithmus
$\boldsymbol{y}$	Ausgangsvektor eines neuronalen Netzes
$\hat{\boldsymbol{y}}$	Sollwert eines Ausgangsvektors eines neuronalen Netzes
$\boldsymbol{z}$	Systemzustand des Zweitank-Systems
α	Schrittweite des Gradientenabstiegsverfahrens
γ	Gewichtungskonstante der additive Fitnessfunktion
ε	Schrittweite der Rückwärtsdifferenz
ε_{min}	Maschinengenauigkeit (doppelte Fließkommazahl)
λ	Schrittweite des Levenberg-Marquardt-Algorithmus
Φ	Aktivierungsfunktion eines Neurons
$\boldsymbol{\omega}$	Gewichtsvektor eines neuronalen Netzes

1 Einleitung

Maschinelles Lernen ist aus unserem heutigen Leben nicht mehr wegzudenken. Immer mehr Aufgaben werden von selbst lernenden Maschinen übernommen.

Beispielsweise kommt maschinelles Lernen in Sprachassistenten wie Siri von Apple zum Einsatz. Diese kommunizieren verbal mit dem Benutzer, um Probleme des Alltags zu lösen. Weiterhin nutzen soziale Netzwerke wie Instagram, Facebook oder Twitter Lernalgorithmen, um die Interessen eines Nutzers zu speichern und personalisierte Werbung anzuzeigen. Ebenso basiert maschinelle Handschrifterkennung auf maschinellem Lernen und wird beispielsweise in Mobiltelefonen eingesetzt [1, 2].

Darüber hinaus ist Maschinelles Lernen ein wichtiges Forschungsgebiet und wird laufend auf neue Problemstellungen angewendet. So setzt die Automobilindustrie maschinelles Lernen in Autos ein, die mit Testfahrten trainiert werden, um später eigenständig im Straßenverkehr zu fahren [3, 4, 5, 6]. Maschinelles Lernen wird auch erfolgreich auf formale mathematische Problemstellungen angewendet: Das Programm AlphaGo wurde von Google implementiert und besiegte einige der weltbesten Spieler in dem Brettspiel “Go” [7].

Allgemein formuliert werden im maschinellen Lernen Algorithmen entwickelt, welche Computern die Fähigkeit geben, anhand von Beispieldaten zu lernen, um dieses Wissen anschließend auf neue, unbekannte Daten anzuwenden. Heutzutage existieren sehr viele verschiedene Verfahren, welche, welche sich je nach Anwendungsbereich unterscheiden.

Eine der Hauptmethoden des maschinellen Lernens sind künstliche neuronale Netze, welche nach dem Vorbild des menschlichen Gehirns entwickelt wurden. Neuronale Netze sind gut erforscht und kommen in vielen Bereichen zum Einsatz. Google Street View verwendet beispielsweise neuronale Netze, um aus mehreren Millionen Straßenbildern Informationen wie Straßennamen oder Namen von Geschäften zu extrahieren und diese in Google Maps darzustellen [8].

Einfach ausgedrückt, bestehen neuronale Netze aus Neuronen und deren Verknüpfungen. Ein künstliches Neuron bildet dabei ein Eingangssignal auf ein Ausgangssignal ab und kann durch eine Funktion $F(x) \rightarrow y$ beschrieben werden. Durch die Anordnung mehrerer Neuronen erhält man eine Verkettung vieler Funktionen, wodurch sich komplexe Zusammenhänge beschreiben lassen. Die Parameter der Funktionen lernt das Netz anhand von Beispieldaten und passt sich so der Problemstellung an.

1.1 Problemstellung

Um neuronale Netze erfolgreich einzusetzen, bedarf es mehrerer Entwicklungsphasen: Hierzu gehört unter anderem die Problemdefinition, die Wahl einer angemessenen Netzstruktur, das Sammeln von Trainingsdaten und das Trainieren des Netzwerks. Häufig wird als Faktor für die Effektivität neuronaler Netze allein der Trainingsalgorithmus zur Ermittlung der optimalen Parameter im Netz betrachtet. Eine entscheidende Voraussetzung für die Leistungsfähigkeit eines Netzes ist jedoch an erster Stelle die Netzarchitektur [9, S. 1]. Allgemein lässt sich die Netzarchitektur durch die Anzahl der Neuronen, sowie die Verbindungen zwischen den Neuronen definieren. Diese können vom Entwickler beliebig gesetzt werden.

Die Entwicklung der endgültigen Topologie eines Netzes ist meist noch eine Aufgabe die von Hand durch Ausprobieren gelöst wird und in der Regel auf der Erfahrung des Entwicklers aus ähnlichen Anwendungen basiert. Für Probleme der realen Welt und ohne ausreichende Kenntnis ist das Design der Netzstruktur oftmals ein langwieriger Prozess und führt nicht zu einer optimalen Lösung [10]. Zudem ist die Gestaltung komplexer Netze ein aufwändiger Vorgang. Hat ein Netz beispielsweise über Tausend Neuronen, bedarf es sehr viel Aufwand, diese individuell zu gestalten. Es ist daher wünschenswert, automatisiert die optimale Topologie eines neuronalen Netzes zu finden.

Die Optimierung der Netzarchitektur enthält viele Parameter und ist schwierig mit gewöhnlichen Optimierungsverfahren durchzuführen. Die Entwicklung von neuronalen Netzen mit Hilfe von genetischen Algorithmen ist seit vielen Jahren Bestandteil der Forschung [11, 12, 10, 13, 14].

Genetische Algorithmen sind Optimierungsverfahren, welche nach den Prinzipien der biologischen Evolution arbeiten. Die Evolution funktioniert nach dem bekannten Prinzip *Survival of the Fittest*: Die Lebewesen, welche am besten an ihre Umweltbedingungen angepasst sind, überleben. Anders ausgedrückt treibt die Evolution die Lebewesen zu einer starken Anpassung an ihre Umweltbedingungen. Analog dazu werden genetische Algorithmen eingesetzt, um neuronale Netze an äußere Gegebenheiten beziehungsweise an eine bestimmte Problemstellung anzupassen.

1.2 Stand der Technik

Ein grundlegender Aspekt für die Optimierung der Netzarchitektur mit Hilfe genetischer Algorithmen ist die Codierung der Netze, auch als Genotyp bezeichnet. Diese ist notwendig, um durch Kreuzung zweier Individuen neue Netze bilden zu können. Die Codierung sollte dabei alle notwendigen Information der Netzarchitektur enthalten, die optimiert werden sollen. Es existieren bereits einige Ansätze, neuronale Netze in genetischen Algorithmen darzustellen. In der Literatur wird zwischen direkten und indirekten Codierungsmethoden unterschieden. Bei direkten Methoden lässt sich die Netzstruktur direkt ablesen, wohingegen eine indirekte Darstellung bestimmte Regeln definiert, welche nacheinander angewendet

werden, um die Netzstruktur zu codieren [15]. Eine gute Übersicht der verschiedenen Darstellungsmöglichkeiten liefert [16].

Die einfachste Möglichkeit ein Netz direkt zu codieren ist die Verbindungen mit Hilfe einer binären Verbindungsmatrix darzustellen [17]. Diese Methode konnte bereits in [18] erfolgreich für die Optimierung von Feedforward Netzen angewendet werden, um einfache Probleme wie die XOR-Funktion zu realisieren. Der Nachteil an dieser Darstellung ist die Entstehung von ungültigen Netzen [19].

In [20] wird eine Methode vorgestellt, mit welcher Feedforward Netze durch eine Liste der Neuronen im Netz repräsentiert werden. Jedes Neuron speichert dabei seine Vorgänger. Neue Individuen werden gebildet, indem zwei Listen an einer Stelle getrennt werden und die Endteile ausgetauscht werden. Diese Methode bietet eine Möglichkeit, Netze mit unterschiedlicher Neuronenanzahl zu kreuzen, jedoch ist die Entstehung von Zyklen nicht ausgeschlossen.

Die Autoren von [21] verwenden eine kontextfreie Grammatik, um einzelne Pfade zu beschreiben. Ein Pfad wird dabei als eine Liste an Neuronen dargestellt und führt immer von einem Eingangs-zu einem Ausgangsneuron. Eine Mutation verändert einzelne Pfade, indem Neuronen hinzugefügt oder entfernt werden, wobei das Crossover ganze Pfade austauscht. Verzögerte Verbindungen sind in dieser Darstellungsmethode nicht enthalten. Angewendet auf eine einfache Regelungsaufgabe konnte dieses Verfahren gute Ergebnisse erzielen.

Lindenmayersysteme sind ein Beispiel für indirekte Codierung und eignen sich ebenfalls für die Darstellung von Feedforward Netzen [16]. Ein Netz wird dabei durch Symbole dargestellt. Feste Regeln definieren, wie diese Symbole umgeschrieben werden, sodass am Ende ein Netz konstruiert werden kann. Der genetische Algorithmus produziert neue Individuen, indem diese Regeln verändert werden. Durch die festen Regeln erhält das Netz bestimmte Strukturen, was den Suchraum des genetischen Algorithmus einschränkt, da nicht jede beliebige Verbindung umgesetzt werden kann [15].

Die Optimierung rekurrenter Netze mit Hilfe von genetischen Algorithmen findet bis jetzt wenig Anwendung.

Rekurrente neuronale Netze haben gegenüber Feedforward Netzen den Vorteil, zeitliche Zusammenhänge in Daten zu erfassen. Sie eignen sich daher unter anderem für Probleme der Regelungstechnik [22]. Die Anpassung der Parameter eines PID-Reglers mit Hilfe von rekurrenten Netzen kommt immer mehr zum Einsatz und konnte bereits gute Ergebnisse erzielen [23, 24, 25].

1.3 Ziel der Arbeit

In dieser Arbeit soll ein genetischer Algorithmus implementiert werden, welcher die Netzarchitektur rekurrenter neuronaler Netze optimiert. Die Netze sollen dabei jede beliebige, sowie verzögerte Verbindung enthalten können und keiner bestimmten Struktur folgen. Die

Optimierung der Netzarchitektur erfolgt für eine bestimmte Problemstellung, nämlich zum automatischen Einstellen der Parameter eines PID-Reglers.

Ziel ist es, eine geeignete Darstellung der Netze zu finden, die jede beliebige Verbindung, mit und ohne Verzögerung, zulässt. Der genetische Algorithmus soll auf dieser Darstellung die Netzstruktur optimieren, sodass der Fehler des zu regelnden Systems minimiert wird.

1.4 Aufbau der Arbeit

Der erste Teil der Arbeit (Kapitel 2) gibt eine Einführung in neuronale Netze, die Funktionsweise genetischer Algorithmen, sowie PID-Regler. Anschließend wird die Implementierung eines adaptiven PID-Reglers, welcher durch ein neuronales Netz kontrolliert wird, beschrieben. Den Hauptteil der Arbeit beinhaltet Kapitel 4, welches die Optimierung der Netzarchitektur dynamischer neuronaler Netze mit Hilfe eines genetischen Algorithmus behandelt. Dabei wird insbesondere auf die Darstellung der Netze im Algorithmus eingegangen, sowie die Implementierung des genetischen Algorithmus unter Verwendung dieser Darstellung. Einen wichtigen Punkt der Implementierung stellt die Laufzeitoptimierung dar, welche durch zwei Methoden umgesetzt wurde. Die Implementierung wird durch ein Experiment getestet, welches in Kapitel 5 beschrieben wird. Zusätzlich wird der genaue Ablauf des genetischen Algorithmus erklärt. In Kapitel 6 wird das Verfahren unter Verwendung verschiedener Einstellungen getestet und ausgewertet. Abschließend wird ein Fazit und ein kurzer Ausblick gegeben.

2 Theoretische Grundlagen

2.1 Neuronale Netze

Die Fähigkeit des menschlichen Gehirns, zu lernen und mühelos komplexe Probleme zu lösen, ist eine erstrebenswerte Eigenschaft im Bereich der künstlichen Intelligenz. Das menschliche Gehirn besitzt schätzungsweise eine Billion Neuronen [26]. Jedes Neuron ist mit etwa 10^3 bis 10^4 anderen Neuronen verknüpft [27], wodurch das zentrale Nervensystem gespannt wird und Informationen in Form von elektrischen Signalen übertragen werden.

Künstliche neuronale Netze (KNN) emulieren den Mechanismus der Signalverarbeitung von biologischen Neuronen. Um den Hintergrund von KNNs besser zu verstehen, werden im folgenden Abschnitt kurz die Analogien zwischen künstlichen und biologischen Neuronen erläutert. Die Funktionsweise von Nervenzellen wird sehr vereinfacht dargestellt, da die biologischen Prozesse für KNNs nicht von Relevanz sind. Eine detaillierte Beschreibung kann zum Beispiel [28] entnommen werden.

2.1.1 Analogie zwischen biologischen und künstlichen neuronalen Netzen

Der Aufbau einer biologischen Nervenzelle ist in Abbildung 2.1 dargestellt. Im Inneren einer Nervenzelle befindet sich der Zellkern (Nucleus), welcher vom Zellkörper (Soma) umgeben ist. Die Dendriten sind kleine Verästelungen um den Zellkörper herum und ermöglichen die Verbindung zu anderen Nervenzellen. Über die Synapsen können die Dendriten Signale anderer Nervenzellen empfangen, welche anschließend zum Zellkörper weitergeleitet und dort verarbeitet werden. Dabei unterscheidet man zwischen erregenden (exzitatorischen) und hemmenden (inhibitorischen) Synapsen. Die einen verstärken die Anregung, während die anderen diese schwächen. Alle eingehenden Signale, sowohl exzitatorische als auch inhibitorische werden im Zellkörper “aufsummiert”. Wie groß ein ankommendes Signal ist, hängt letztlich davon ab, wie effizient der Transport an der Synapse ist. Dies wird oft auch als Stärke oder Intensität der Synapse bezeichnet [29, S. 9]. Erreichen die Eingangssignale einen bestimmten Schwellenwert, feuert das Neuron ein Aktionspotential ab, welches über das Axon zu anderen Nervenzellen weitergeleitet wird.

In Abblidung 2.2 ist der Aufbau eines künstlichen Neurons zu sehen. Analog zur Verarbeitung der Signale im Zellkörper wird das künstliche Neuron als Recheneinheit modelliert. Ein Neuron kann eine beliebige Anzahl an Eingangssignalen x_i erhalten, welche mit Skalaren ω_i gewichtet und aufsummiert werden. Die Gewichtung entspricht der Modellierung der Signale beim Transport durch die Synapsen. Zusätzlich wird ein Bias eingeführt,

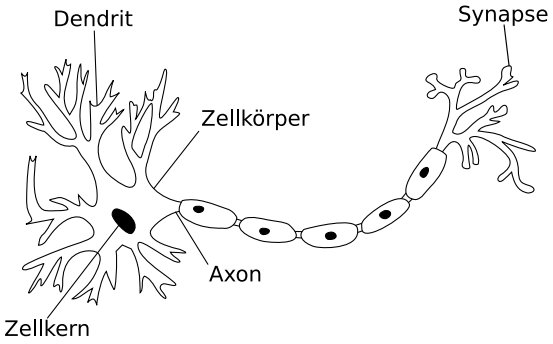


Figure 2.1: Schematischer Aufbau einer Nervenzelle [1]

welcher als Eingangssignal $x_b = 1$ repräsentiert wird und dessen Gewicht mit $\omega_{b,i}$ gekennzeichnet ist. Der Ausgang des Neurons wird wie folgt berechnet

$$y = \phi\left(\sum_{i=1}^{N} \omega_i x_i + 1 \cdot \omega_b\right), \tag{2.1}$$

wobei N die Anzahl der Eingangsignale des Neurons ist. Die Aktivierungsfunktion $\phi(\cdot)$ kann mit der Eigenschaft des biologischen Neurons, ein Aktionspotential zu feuern, verglichen werden [30]. In KNNs dient sie zum einen dazu, den Wertebereich des Ausgangswerts eines Neurons zu beschränken, zum anderen lässt sich eine Nichtliniarität einbringen. Verschiedene Aktivierungsfunktionen können das Lernverhalten eines KNNs unterschiedlich beeinflussen, sie muss daher problemspezifisch gewählt werden. In [31] werden drei verschiedene Aktivierungsfunktionen für unterschiedliche Probleme getestet und verglichen. Tabelle 2.1 zeigt einige Beispiele von möglichen Aktivierungsfunktionen. Der Bias ermöglicht eine Verschiebung der Aktivierungsfunktion.

Der Lernvorgang im biologischen Netzwerk erfolgt durch die Modifikation und Anpassung der Stärke in der synaptischen Übertragung [30]. Gleichermaßen ist es bei KNNs das Ziel, die Gewichtungen ω_i kontinuierlich anzupassen.

Es gibt verschiedene Möglichkeiten, die Neuronen in einem KNN anzuordnen und zu verbinden. Prinzipiell unterscheidet man zwischen statischen und dynamischen Netzen. In den nächsten drei Abschnitten werden unterschiedliche Arten von KNNs vorgestellt.

2.1.2 Statische neuronale Netze

In statischen neuronalen Netzen, auch bekannt als Feedforward-Netze, verlaufen die Verbindungen zwischen den Neuronen nur in eine Richtung, das heißt von der Eingangsschicht aus “fließt” das Signal in Richtung Ausgang. Diese Konstruktion stellt sicher, dass im

[1] `https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg`

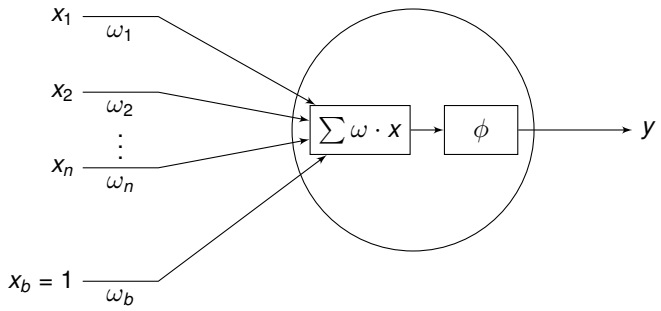


Figure 2.2: Künstliches Neuron: Die Eingangssignale x_i werden mit den Gewichten ω_i multipliziert und im Neuron aufsummiert. Anschließend bildet eine Aktivierungsfunktion $\Phi(\cdot)$ den Ausgangswert y.

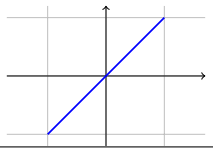
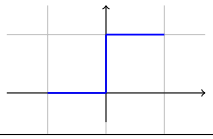
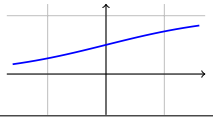
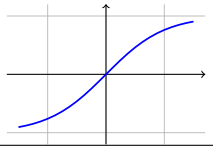
Name	Plot	Funktion	Differenzierbar	Wertebereich
Identität		x	Ja	$]-\infty,\infty[$
Einheitssprung		$\begin{cases} 0 \text{ für } x \leq 0 \\ 1 \text{ für } x > 0 \end{cases}$	Nein	$\{0,1\}$
Sigmoidfunktion		$\frac{1}{1+\exp(-x)}$	Ja	$[0,1]$
Tangens Hyperbolicus		$\tanh(x)$	Ja	$[-1,1]$

Table 2.1: Beispiele für Aktivierungsfunktionen

Netz keine Zyklen entstehen, was der grundlegende Unterschied zu dynamischen Netzen ist.

Die einfachste Form von Feedforward-Netzen ist das Perzeptron [32]. Es besteht aus einem einzelnen Neuron und wurde bereits im letzten Abschnitt in Abbildung 2.2 gezeigt. Mit einem Perzeptron lässt sich lediglich eine einfache Funktion implementieren, welche einen mehrdimensionalen Vektor auf ein Skalar abbildet. Mit Hilfe eines Perzeptrons lassen sich somit ausschließlich lineare Klassifizierungsprobleme lösen [30]. Durch die Anordnung mehrerer Perzeptronen können größere Netze gebildet werden, was zu einer weiteren Klasse von Feedforward-Netzen führt.

Mehrlagiges Perzeptron

Mit einem mehrlagigen Perzeptron (MLP) können im Gegensatz zum einzelnen Perzeptron beliebig komplexe Probleme gelöst werden [27]. Aufgrund seiner einfachen Netzstruktur gilt es als eines der beliebtesten Netze [33]. Es findet daher Anwendung in vielen Bereichen, so zum Beispiel in der Mustererkennung, Steuerungstechnik, Datenkomprimierung oder Bildverarbeitung [30].

MLPs besitzen eine bestimmte Struktur, bei der die Neuronen in einzelne Schichten unterteilt sind. So gibt es eine Eingangsschicht, eine Ausgangsschicht und dazwischen beliebig viele, aber mindestens eine, versteckte Schicht. Die Neuronen der Eingangsschicht erhalten den Wert des Eingangssignals, wobei das Eingangssignal nicht gewichtet wird. Die Ausgangsschicht liefert den Ergebnisvektor. Die Aktivierungsfunktion der Ausgangsneuronen muss je nach Art des Problems gewählt werden. Für binäre Klassifizierung eignet sich zum Beispiel der Einheitssprung oder die Sigmoidfunktion. Die versteckten Schichten interagieren nicht mit der äußeren Umgebung und werden daher als “versteckt” bezeichnet [30]. Durch Hinzufügen der versteckten Schichten ist es möglich, auch nichtlineare Probleme zu lösen. Die Anzahl der versteckten Schichten muss abhängig von der Komplexität der Daten angepasst werden.

Die Neuronen eines MLPs sind vollständig miteinander verbunden, so dass jedes Neuron aus Schicht k eine Verbindung zu jedem Neuron in Schicht $k+1$ besitzt. In Abbildung 2.3 ist ein Beispiel eines MLPs mit zwei versteckten Schichten zu sehen. Der Bias wird hier als zusätzlicher Eingang dargestellt. In allen weiteren Abbildungen wird er der Einfachheit halber nicht mit eingezeichnet.

MLPs sind besonders gut für die Klassifizierung von Mustern geeignet und werden häufig im Bereich der Bildanalyse verwendet. In [34] wird mit Hilfe von MLPs die Erkennung von handschriftlichen Ziffern, sowie die Erkennung und Positionsbestimmung von 3D Objekten implementiert. Auch im medizinischen Bereich finden KNNs ihre Anwendung. Die Autoren von [33] entwickelten ein System mit MLPs zur Unterstützung der Diagnose von Herzerkrankungen.

Es gibt verschiedene Verfahren zum Trainieren von neuronalen Netzen. Einer der am häufigsten verwendeten Trainingsalgorithmen für MLPs ist der Backpropagation Algorithmus

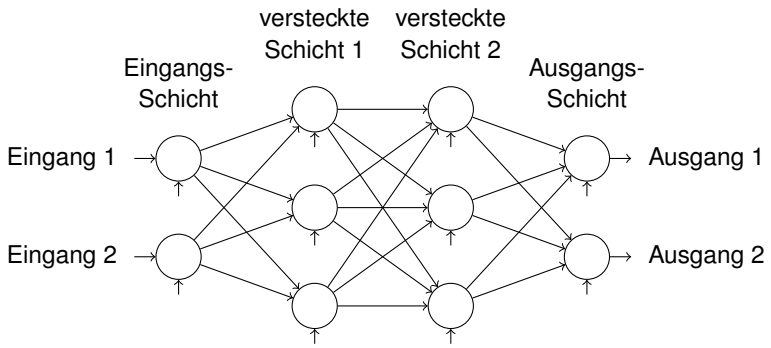


Figure 2.3: Mehrlagiges Perzeptron mit zwei versteckten Schichten

(BP-Algorithmus), da dieser eine einfache Optimierung der Gewichte ermöglicht. Im nächsten Abschnitt wird das Trainieren von MLPs mit Hilfe des BP-Algorithmus erklärt.

Backpropagation

Im folgenden wird der BP-Algorithmus für das Prinzip des *Überwachten Lernens* erklärt. Dies bedeutet, dass die Gewichte im Netz so angepasst werden sollen, dass ein beliebiger Eingangsvektor $\boldsymbol{x}$ auf einen gewünschten Ausgangsvektor $\hat{\boldsymbol{y}}$ abgebildet werden kann. Das Netz approximiert also eine Funktion anhand von Beispieldaten, welche als Trainingsdaten bezeichnet werden. Die Trainingsdaten müssen vorher bekannt sein und Ein-und Ausgangswertepaare enthalten.

Aus dem Sollwert $\hat{y}_i$ und dem tatsächlichem Wert y_i eines Ausgangsneurons lässt sich die Fehlerfunktion

$$\hat{E} = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{2.2}$$

formulieren, welche in Abhängigkeit der Gewichte im Netz minimiert werden soll. Dabei kennzeichnet N die Anzahl der Ausgangsneuronen. Der BP-Algorithmus verwendet das Gradientenabstiegsverfahren zur Optimierung der Gewichte und wiederholt für alle Wertepaare $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ folgende zwei Schritte:

1. Der aktuelle Eingangsvektor $\boldsymbol{x}_t$ wird durch das Netz propagiert und der Wert aller Ausgangsneuronen $\boldsymbol{y}_t$ wird berechnet. Daraus wird der Gesamtfehler E nach 2.2 berechnet.

2. Alle Gewichte werden gemäß

$$\omega_t = \omega_{t-1} - \alpha\frac{\partial\hat{E}}{\partial\omega} \tag{2.3}$$

aktualisiert. Die Variable α entspricht der Schrittweite des Gradientenabstiegsverfahren.

Um die Gewichte nach 2.3 anpassen zu können, müssen die partiellen Ableitungen des Fehlers E bezüglich aller Gewichte ω berechnet werden. In den verdeckten Schichten gestaltet sich dies sehr mühsam, da die Ableitungen hier in Abhängigkeit aller Nachfolgeneuronen bestimmt werden müssen.

Der BP-Algorithmus ermöglicht ein effizientes Berechnen aller Ableitungen. Die Idee dabei ist, mit der Ausgangsschicht zu beginnen und bereits berechnete Teile für die Ableitungen in vorherigen Schichten wiederzuverwenden.

Der Netzeingang eines Neurons i in Schicht $k+1$ berechnet sich nach [35] mit der Formel

$$n_i^{k+1} = \sum_{j=1}^{m} \omega_{ij}^{k+1} o_j^k + b_i^{k+1}, \tag{2.4}$$

wobei m die Gesamtanzahl an Eingängen des Neurons i kennzeichnet und o_j^k das Ausgangssignal des Vorgängerneurons. Der Ausgangswert des Neurons i entspricht dann

$$o_i^{k+1} = \Phi(n_i^{k+1}). \tag{2.5}$$

Die Ableitung des Fehlers nach einem Gewicht ω_{ij}^k kann mit Hilfe der Kettenregel

$$\frac{\partial \hat{E}}{\partial \omega_{ij}^k} = \frac{\partial \hat{E}}{\partial o_i^k} \frac{\partial o_i^k}{\partial n_i^k} \frac{\partial n_i^k}{\omega_{ij}} \tag{2.6}$$

berechnet werden. Führt man die einzelnen Ableitungen für ein Gewicht der Ausgangsschicht aus, lässt sich die Fehlerfunktion 2.2 direkt ableiten und man erhält

$$\frac{\partial \hat{E}}{\partial \omega_{ij}^k} = \underbrace{-(\hat{y}_i - y_i) \frac{\partial \Phi(n_i^k)}{\partial n_i^k}}_{\delta_i^k} o_j^{k-1}. \tag{2.7}$$

Dabei substituiert man einen Teil mit δ_i^k, welches für die Berechnung der Ableitungen der Vorgänger-Schicht verwendet wird.

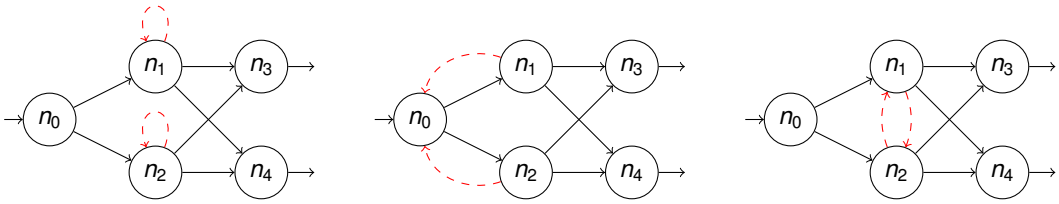
Für ein Gewicht der versteckten Schicht k berechnet sich δ_i^k nach [30]

$$\delta_i^k = \frac{\partial \Phi(n_i^k)}{\partial n_i^k} \sum_{l=1}^{L} \delta_l^{k+1} \omega_{li}^{k+1}. \tag{2.8}$$

Dabei ist L die Gesamtanzahl der Nachfolgeneuronen. Daraus folgt die allgemeine Formel für die Berechnung der partiellen Ableitung

$$\frac{\partial E}{\partial \omega_{ij}^k} = \delta_i^k o_j^{k-1}. \tag{2.9}$$

Eine detailliertere Herleitung der Formeln für den BP-Algorithmus geben die Autoren von [36].



(a) Direkte rückwärtige Verbindung

(b) Indirekte rückwärtige Verbindung

(c) Laterale rückwärtige Verbindung

Figure 2.4: Beispiele rekurrenter Netze. Der Übersichtlichkeit halber werden verzögerte Verbindungen als gestrichelte, rote Linien dargestellt.

2.1.3 Dynamische neuronale Netze

Im Gegensatz zu statischen Netzen besitzen dynamische neuronale Netze ein Gedächtnis. Dies bedeutet, dass frühere Werte der Neuronen gespeichert werden und das aktuelle Ausgangssignal nicht allein vom aktuellen Eingangssignal abhängt, sondern auch von vorherigen Zuständen des Netzes [37]. Umgesetzt wird das Gedächtnis mit Hilfe von Feedback-Verbindungen, welche eine zeitliche Verzögerung besitzen. Durch die rückwärtigen Verbindungen entstehen im Netz Zyklen, die allerdings auf Grund der Verzögerungen kein Problem darstellen. Dies ist wichtig, da der Wert eines Neurons nicht in Abhängigkeit seines eigenen aktuellen Wertes berechnet werden kann.

Ein Netz mit Feedback-Verbindungen ist demnach ein dynamisches System, welches einen internen Zustand besitzt. Dies ermöglicht die Verarbeitung von zeitabhängigen Daten [38].

In Abbildung 2.4 sind drei Arten von rückwärtigen Verbindungen dargestellt, welche hier als rote, gestrichelte Linien gekennzeichnet sind. Eine direkte rückwärtige Verbindung führt den Ausgangswert eines Neurons direkt zu sich selber als Eingang zurück. Der Wert $a_1(t)$ von Neuron n_1 zum Zeitpunkt t aus Abbildung 2.4a ließe sich dann für eine Zeitverzögerung von 1 mit $a_1(t) = \phi(\omega_0 a_0(t) + \omega_1 a_1(t-1))$ berechnen. Dabei ist ω_0 das Gewicht der nicht verzögerten und ω_1 das Gewicht der verzögerten Verbindung. Eine indirekte rückwärtige Verbindung verbindet Neuronen einer Schicht mit Neuronen einer vorherigen Schicht, wie in Abbildung 2.4b dargestellt. Verbindungen zwischen Neuronen derselben Schicht werden als lateral bezeichnet und sind in Abbildung 2.4c zu sehen [39].

Ausgehend von einem FNN können durch Hinzufügen von Feedback-Verbindungen verschiedene Arten von rekurrenten neuronalen Netzen (RNN) generiert werden. Eine der ersten Formen von RNNs war das sogenannte Elman-Netzwerk [40]. In dieser Struktur wird der Ausgang von Neuronen der versteckten Schicht zu einem Kontextneuron geführt, welches nur als Hilfsneuron dient. Das Gewicht dieser Verbindungen wird dabei fest auf 1 gesetzt und nicht optimiert. Der Wert der Kontextneuronen wird dann über eine verzögerte Verbindung zu den selben Neuronen zurückgeführt. Dies entspricht im Grunde einer direkten, rückwärtigen Verbindung [41]. Ein weiteres Beispiel eines RNNs ist das

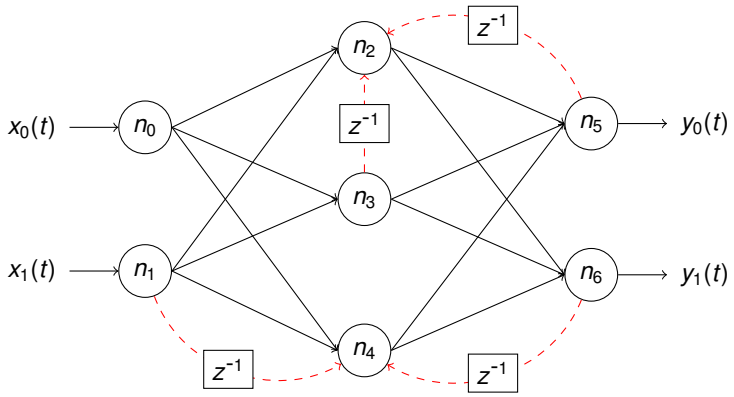


Figure 2.5: Neuronales Netz mit verzögerten Vorwärts-und Rückwärtsverbindungen

Jordan-Netzwerk [42], welches im Gegensatz zum Elman-Netzwerk die Ausgangsneuronen in die Feedback-Struktur involviert.

Die eben genannten Beispiele enthalten jeweils nur eine Art von rekurrenten Verbindungen. Durch Kombination der verschiedenen rückwärtigen Verbindungen mit beliebiger Verzögerung, sowie Integration von Neuronen unterschiedlicher Schichten können beliebige neuronale Netze gebildet werden.

Dynamische neuronale Netze mit beliebigen Verbindungen

Im Gegensatz zu den oben genannten Arten von neuronalen Netzen, welche einer vordefinierten Struktur folgen, werden in dieser Arbeit dynamische neuronale Netze mit beliebigen Verbindungen betrachtet. In Abschnitt 2.1.2 wurde das MLP eingeführt, welches nur Verbindungen zwischen direkt aufeinander folgenden Schichten zulässt. In Netzen mit beliebiger Struktur soll jedes Neuron mit jedem anderen beliebigen Neuron verbunden werden können. Zusätzlich kann jede Verbindung (sowohl Vorwärts-, als auch Rückwärtsverbindung) eine beliebig große Verzögerung besitzen (in dieser Arbeit werden nur Verzögerungen von einer Zeiteinheit verwendet). Hierbei gilt ebenfalls die Bedingung aus 2.1.3, dass keine Zyklen entstehen dürfen, die nicht simulierbar sind.

Dynamische Netze lassen sich nicht mit dem klassischen BP-Algorithmus aus Kapitel 2.1.2 trainieren. Zwei grundlegende Verfahren zum Trainieren dynamischer Netze sind der *Backpropagation-Through-Time* Algorithmus (BPTT), sowie der *Real-Time Recurrent Learning* Algorithmus (RTRL) [37]. In Kapitel 3.1 wird die in dieser Arbeit verwendete Trainingsmethode im Detail beschrieben.

2.2 Genetische Algorithmen

Genetische Algorithmen wurden erstmals von dem amerikanischen Wissenschaftler John Henry Holland im Jahre 1960 entwickelt. 1975 veröffentlichte er das Buch *Adaptation in Natural and Artificial Systems* [43]. Mit seiner Arbeit wollte er den Prozess der Anpassung - wie er durch die biologische Evolution zustande kommt - erkunden und in technischen Systemen umsetzen [44].

Heutzutage werden genetische Algorithmen hauptsächlich zum Lösen komplexer Optimierungsaufgaben verwendet. Von anderen Optimierungsalgorithmen unterscheiden sie sich durch die Methode, neue Lösungskandidaten zu finden. So müssen zum Beispiel keine Ableitungen berechnet werden. Dies ist vor allem für komplexe Probleme, welche nicht differenzierbar sind oder die Berechnung des Gradienten zu aufwendig wäre, vorheilhaft [45]. Ziel ist es, von einer Startpopulation aus durch die Evolution von Lösungskandidaten über mehrere Generationen hinweg die beste Lösung zu finden.

In diesem Kapitel werden die wichtigsten Grundlagen zu genetischen Algorithmen, wie der allgemeine Ablauf, die Repräsentation von Individuen, die Anwendung genetischer Operatoren, sowie die Bedeutung der Fitnessfunktion zusammengefasst.

2.2.1 Algorithmus

Wie bereits in 1 erwähnt, arbeiten genetische Algorithmen nach dem Prinzip der biologischen Evolution: Ein Pool aus Individuen definiert eine Population, wobei ein Individuum ein Teil des Suchraums ist. Die einzelnen Individuen werden mithilfe einer Fitnessfunktion bewertet, um eine Aussage darüber treffen zu können, wie gut ein Individuum als Lösung des Problems geeignet ist. Durch Anwendung von bestimmten Selektionsverfahren werden probabilistisch gute Individuen ausgewählt und anschließend miteinander gekreuzt. Dies entspricht dem Austausch von Erbmaterial in der Genetik. Zusätzlich werden einzelne Individuen mutiert. Daraus entstehen neue Individuen, die in die Population integriert werden und schließlich die nächste Generation an Individuen gebildet wird. Diese Schritte werden für eine bestimmte Anzahl an Generationen oder bis zur Erfüllung eines bestimmten Kriteriums wiederholt [44]. Die Idee dabei ist, fitte Individuen für die Fortpflanzung auszuwählen in der Annahme, dass dadurch fittere Nachkommen entstehen und die schlechteren Individuen in der Population ersetzt werden. Der Algorithmus soll sich so dem globalen Optimum nähern. Algorithmus 1 zeigt die grundlegenden Schritte eines genetischen Algorithmus.

In der Genetiklehre unterscheidet man zwischen dem Genotyp und dem Phänotyp eines Individuums. Analog dazu werden bei der Anwendung genetischer Algorithmen ebenfalls zwei Repräsentationsweisen verwendet: Der Genotyp eines Individuums wird dazu benötigt, die Individuen im genetischen Algorithmus miteinander zu kreuzen und so neue Individuen zu bilden. Dabei enthält der Genotyp nicht die gesamte Information eines Individuums. Nur bestimmte Parameter des Individuums, welche optimiert werden sollen, werden so codiert. Eine häufig verwendete Darstellung ist ein einfacher binärer Vektor, welcher aus einer Abfolge von Bits besteht: $\hat{x} = (1,0,0,1,1,1)$.

Daten : Anzahl an Generationen N
Ergebnis : Bestes Individuum $\hat{x}_{\text{opt}}$
Generiere zufällige Startpopulation $\boldsymbol{P}_0 = \{\hat{x}_0, \hat{x}_1, ..., \hat{x}_n\}$
Berechne Fitnessfunktion $F(\hat{x}_i)$ aller Individuen $\hat{x}_i$
für $k = 1$ **bis** N **tue**
  Wähle Eltern mit Hilfe von Selektionsmethoden
  Bilde Nachkommen durch Anwendung von Crossover und Mutation
  Bewerte Fitness der Nachkommen
  Bilde neue Generation $\boldsymbol{P}_k$
Ende
zurück $\hat{x}_{\text{opt}}$

Algorithmus 1 : Wesentilcher Ablauf eines genetischen Algorithmus

Der Phänotyp entspricht dem äußeren Erscheinungsbild eines Individuums und ist durch seinen Genotyp festgelegt [9, S. 17f]. Im Beispiel von neuronalen Netzen ist das Netz selber der Phänotyp und bestimmte Eigenschaften des Netzes, wie Anzahl der Neuronen und Verbindungen im Netz der Genotyp. In jeder Iteration des genetischen Algorithmus müssen die neuen Individuen in ihre Darstellung als Netze transformiert werden, um sie trainieren und evaluieren zu können.

2.2.2 Fitnessfunktion

Je nachdem welche Parameter der Individuen optimiert werden sollen, sollte die Fitnessfunktion die Leistung eines Individuums hinsichtlich dieser Eigenschaften reflektieren. Die Fitnessfunktion muss folglich abhängig vom Problem bestimmt werden. Wie bereits im letzten Abschnitt erwähnt, selektiert der Algorithmus die Individuen anhand ihrer Fitness und soll sich so von Generation zu Generation dem globalen Optimum nähern. Die Wahl der Fitnessfunktion spielt daher eine wichtige Rolle für die Optimierung. Der Autor von [46, S. 15f] liefert eine ausführliche Beschreibung, wie die Fitnessfunktion bestimmt werden kann. Diese wird hier kurz zusammengefasst.

Die Fitness kann zum Beispiel in Hinblick auf die Gültigkeit eines Individuums [46, S. 39-43] oder auf die Anzahl der zu optimierenden Parameter angepasst werden. Entstehen im genetischen Algorithmus Individuen, welche einen ungültigen Phänotypen besitzen, sollte diese Information in die Fitnessfunktion integriert werden. Ein ungültiges Individuum definiert sich durch die Bedingungen des Optimierungsproblems, zum Beispiel durch Verletzung von physikalischen Bedingungen oder anderen problemspezifischen Anforderungen.

Bei der Optimierung der Topologie neuronaler Netze könnte zum Beispiel ein Netz, welches Neuronen enthält, die keine Wirkung auf den Ausgangswert haben, als ungültiges Netz definiert werden.

Eine einfache Methode, Individuen mit bestimmten Eigenschaften zu vermeiden, ist die “Todesstrafe”. Dabei wird jedes neu entstandene Individuum zunächst auf seine Gültigkeit

überprüft. Verletzt es die Bedingungen des Optimierungsproblems, so wird es gelöscht und ein neues Individuum wird generiert. Dies wird so oft wiederholt, bis ein gültiges Individuum entsteht.

Eine andere Möglichkeit besteht darin, die Fitness der Individuen mit Hilfe einer Bestrafungsfunktion abzuwerten. Je nachdem wie stark ein Individuum die Bedingungen verletzt, wird dessen Fitness mehr oder weniger "bestraft". Die Beeinflussung der Fitness durch eine Bestrafungsfunktion gilt als effizienter, da sie die Selektionswahrscheinlichkeit ungültiger Individuen nur verringert, aber nicht den Suchraum einschränkt. Weiterhin kann die Todesstrafe die Laufzeit verlängern, wenn die Generierung eines gültigen Individuums viele Versuche erfordert.

Ein weiterer Spezialfall ergibt sich, wenn mit Hilfe des genetischen Algorithmus mehrere Parameter gleichzeitig optimiert werden sollen. Meist wird hierzu für jeden Parameter eine eigene Fitness berechnet, welche dann als gewichtete Summe

$$F(\hat{x}) = \beta_1 F_1(\hat{x}) + \beta_2 F_2(\hat{x}) \cdots + \beta_k F_k(\hat{x}) \tag{2.10}$$

zusammengefasst und gleichzeitig optimiert werden. Je nachdem welche Eigenschaft von größerer Bedeutung ist, können die Gewichte angepasst werden.

Das richtige Einstellen der Gewichte einer additiven Fitnessfunktion ist meist keine triviale Aufgabe, insbesondere wenn die Verbesserung der einen Fitness zu einer Verschlechterung der anderen führt [47].

Im nächsten Abschnitt wird die Ausführung der genetischen Operatoren erklärt.

2.2.3 Genetische Operatoren

Die genetischen Operatoren dienen dazu, neue Individuen zu bilden und somit den Suchraum zu erkunden. Das Crossover "tauscht" dabei Eigenschaften der Individuen und erlaubt dem Algorithmus, bestimmte Charakteristiken zu erhalten. Die Mutation dient dazu, Eigenschaften der Individuen nur leicht zu ändern. Wie die genetischen Operatoren angewendet werden, hängt an erster Stelle von der Darstellung der Individuen ab. Im Folgenden werden Crossover und Mutation anhand von Chromosomen, die durch binäre Vektoren repräsentiert werden, gezeigt.

Crossover

Beim Crossover werden Teile der Eltern-Chromosomen zu einem neuen Chromosom zusammengesetzt. Dabei gibt es verschiedene Möglichkeiten, die Chromosomen zu kreuzen [9, S. 27ff]. Im folgenden werden drei Arten erklärt.

Beim **One-Point-Crossover** werden die Chromosomen der beiden Eltern an einer zufällig gewählten Stelle getrennt. Der abgeschnittene Teil wird dann unter den Eltern ausgetauscht. Abbildung 2.6a veranschaulicht den Austausch der Chromosomenteile und die dabei entstehenden Nachkommen.

Durch die Wahl von zwei zufälligen Crossover Punkten, lässt sich auch Genmaterial aus der Mitte der Chromosomen austauschen. Abbildung 2.6b zeigt ein Bespiel von **Two-Point-Crossover**.

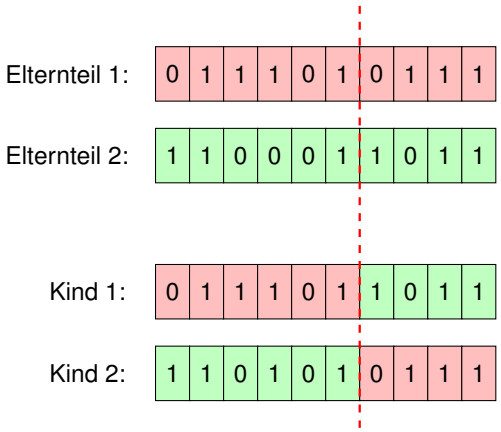
Das **Uniform-Crossover** tauscht im Gegensatz zu den anderen beiden Methoden nicht nur eine zusammenhängende Zeichenkette der Eltern aus. Bei dieser Methode wird jedes Bit der Nachkommen mit einer bestimmten Wahrscheinlichkeit vom Vater oder von der Mutter übernommen. Dadurch können mehrere Crossover Punkte entstehen, wie Abbildung 2.6c veranschaulicht.

Mutation

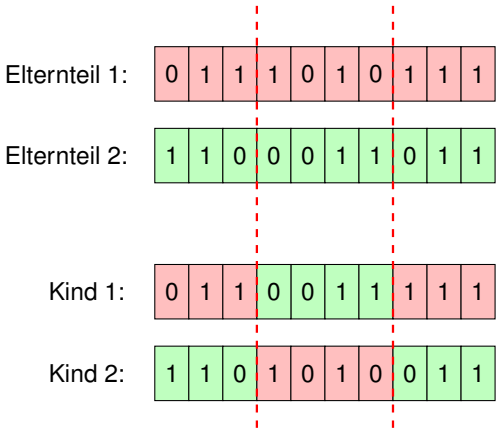
Mit Hilfe der Mutation können ebenfalls neue Individuen gebildet werden, indem Genmaterial zufällig verändert wird. Die Mutation ist wichtig für den Erhalt der Vielfalt innerhalb der Population, sowie für die Exploration des Suchraums. Entstehen beim Crossover zu ähnliche Individuen, können mit der Mutation neue Merkmale (der Individuen) geschaffen werden. Durchgeführt wird die Mutation meist, indem jedes einzelne Gen der neuen Nachkommen mit einer bestimmten Wahrscheinlichkeit mutiert wird [9, S. 29]. Im Beispiel der binären Zeichenketten würde man ein Bit einfach negieren. Wie hoch die Rate für die Mutation gesetzt wird, beeinflusst das Verhalten des genetischen Algorithmus. Eine zu hohe Mutationswahrscheinlichkeit würde willkürliche Nachkommen produzieren und die Fortpflanzung der “fitten” Individuen wäre nutzlos.

2.2.4 Selektionsmethoden

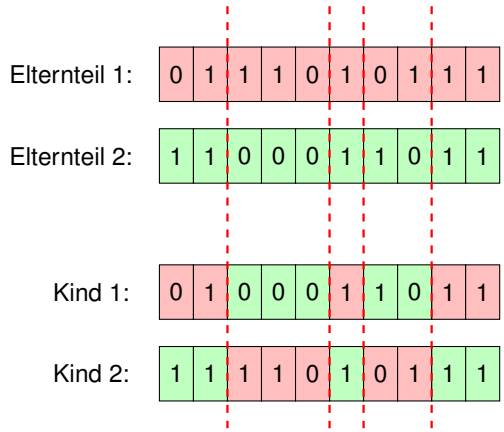
Bei der Selektion werden Individuen aus der Population für die “Fortpflanzung” ausgewählt. Dabei gibt es verschiedene Kriterien, nach welchen die Eltern ausgewählt werden können. Die Wahl der Selektionsmethode hat einen großen Einfluss auf die Effizienz des Algorithmus, da sie entscheidend für seine Konvergenz, sowie die Exploration des Lösungsraums ist. Die grundlegende Idee dabei ist, Individuen mit einer besseren Fitness mit höherer Wahrscheinlichkeit für das Crossover zu selektieren als Individuen mit schlechter Fitness. Dennoch sollten alle Individuen eine Chance haben, sich fortzupflanzen, um die Vielfalt innerhalb der Population zu gewährleisten. Fällt die Wahl der Eltern ausschließlich auf die besten Individuen (Roulettekesselselektion), entsteht dadurch ein hoher Selektionsdruck. Die Gefahr hierbei ist die Entstehung von ähnlichen Nachkommen. Viele unterschiedliche Individuen in der Population hingegen erweitern den Suchraum, was wiederum die Wahrscheinlichkeit reduziert, frühzeitig gegen ein lokales Optimum zu konvergieren. Ein rein zufälliges Auswahlverfahren wiederum würde die Idee der natürlichen Selektion zunichte machen und der Algorithmus wäre nichts anderes als eine Zufallssuche. Die Autoren von [44] geben einen guten Überblick über die Auswahl an Selektionsmethoden. Prinzipiell lässt sich zwischen proportionaler Selektion, Rangselektion und Turnierselektion unterscheiden.



(a) One-Point-Crossover

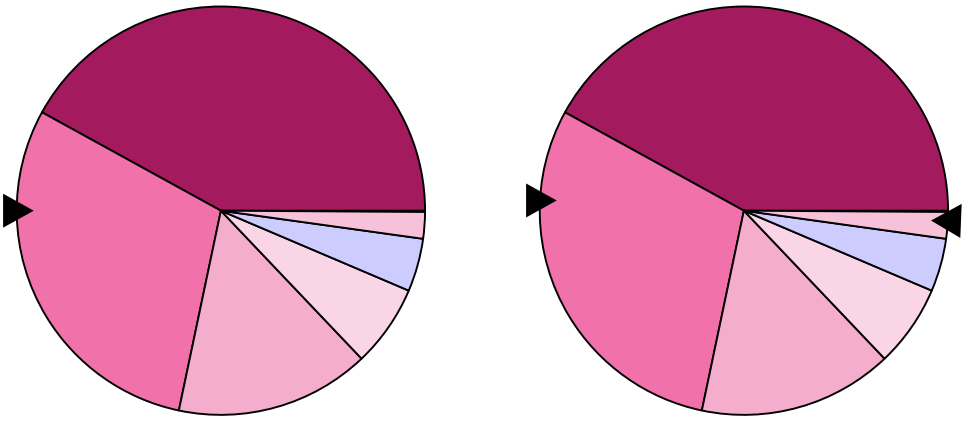


(b) Two-Point-Crossover



(c) Uniform-Crossover

Figure 2.6: Verschiedene Arten von Crossover



(a) Roulettekessel Methode **(b)** Stochastisches universelles Sampling

Figure 2.7: Proportionale Selektionsmethoden

Proportionale Selektion

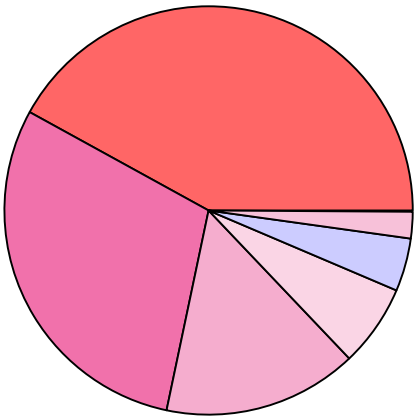
Die einfache Variante der proportionalen Selektion ist die sogenannte Roulettekesselselektion. Abbildung 2.7a veranschaulicht das Prinzip dieser Selektion. Jedem Individuum wird ein unterschiedlich großes Stück auf dem Kessel zugeteilt, abhängig von dessen Fitness. Das kleine schwarze Dreieck dient als Zeiger. Der Kessel wird gedreht und der Zeiger bestimmt, welches Individuum selektiert wird. Die Wahrscheinlichkeit eines Individuums x ausgewählt zu werden, ist somit proportional zur Fitness $F(\hat{x})$ und definiert sich durch

$$p(\hat{x}) = \frac{F(\hat{x})}{\sum F}. \tag{2.11}$$

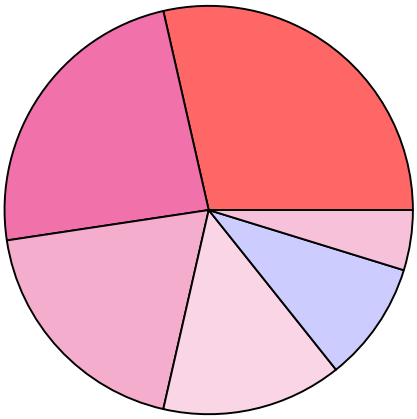
Der Nachteil an dieser Methode ist, dass fitte Individuen sehr häufig selektiert werden. Eine andere Variante des Roulettekessels ist das stochastische universelle Sampling (SUS). Auch hier wird ein Roulettekessel in Abhängigkeit der Fitnesswerte aufgeteilt, jedoch werden hier beide Eltern auf einmal gewählt. In Abbildung 2.7b gibt es hierfür zwei Zeiger, welche die Individuen bestimmen sollen. Dadurch erhalten auch weniger fitte Individuen eine Chance, selektiert zu werden.

Rangselektion

Im Gegensatz zur proportionalen Selektion, bestimmt die Rangselektion die Selektionswahrscheinlichkeit eines Individuums anhand seines Rangs innerhalb der Population, nicht anhand seiner Fitness. Die Individuen werden hierfür abhängig von ihrem Fitnesswert sortiert und einem Rang zugeordnet. Dabei bekommt das schlechteste Individuum Rang 1 und das beste Individuum den höchsten Rang n. Die Selektionswahrscheinlichkeiten



(a) Wahrscheinlichkeiten proportional zur Fitness



(b) Wahrscheinlichkeiten definiert nach dem Rang

Figure 2.8: Verteilung der Selektionswahrscheinlichkeiten proportional zur Fitness und nach dem Rang berechnet

können entweder mit einer linearen Funktion (Lineare Rangselektion)

$$p(\hat{x}) = \frac{2 \cdot Rang(\hat{x})}{n(n+1)}, \tag{2.12}$$

oder einer nicht linearen (z. B. Exponentielle Rangselektion) bestimmt werden. Abbildung 2.8 zeigt den Unterschied zwischen proportionaler Wahrscheinlichkeitsverteilung und der Verteilung nach einem Ranking. Individuen, die vorher eine hohe Selektionswahrscheinlichkeit besaßen, werden nun weniger wahrscheinlich gewählt. Bei der Roulettekessel-Selektion dominieren besonders gute Individuen die Population, wodurch der Algorithmus schnell gegen ein lokales Minimum konvergieren kann. Die Rangselektion bietet sich daher besonders bei großen Unterschieden zwischen den Fitnesswerten an. Jedoch besteht hier wiederum die Gefahr einer zu langsamen Konvergenz, da sich die fitten Individuen weniger von den restlichen Individuen hervorheben [48]. Anstatt die Individuen nur mit ihrem Rang zu gewichten, kann der Rang zusätzlich mit einer Funktion skaliert werden und der Selektionsdruck variiert werden. Ein Beispiel wird in [48] gezeigt.

Turnierselektion

Bei der Turnierselektion werden zunächst k Individuen zufällig aus der Population ausgewählt. Anschließend wird aus diesen Individuen das Individuum mit der besten Fitness für die Fortpflanzung selektiert. Der Parameter k wird dabei als Turniergröße bezeichnet und beeinflusst das Verhalten des Algorithmus. Je mehr zufällige Individuen anfangs gewählt werden, desto höher wird der Selektionsdruck, daher wird meist ein kleines k verwendet. Somit ist die Turnierselektion eine gute Methode, die Vielfalt innerhalb der Population zu

erhalten, da jedes Individuum mit gleicher Wahrscheinlichkeit anfangs gewählt werden kann. Das Individuum mit dem kleinsten Fitnesswert kann jedoch niemals ausgewählt werden, da es immer schlechter ist als seine Konkurrenten. Abbildung 2.9 veranschaulicht das Prinzip der Turnierselektion.

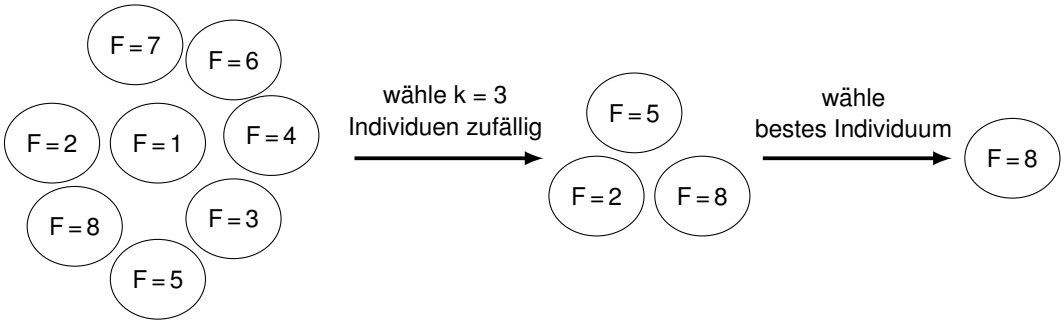


Figure 2.9: Turnierselektion [48]

2.3 PID-Regler

Die Regelungstechnik beschäftigt sich mit der Entwicklung von Reglern, mit denen dynamische Systeme in einen gewünschten Zustand überführt werden. Ein dynamisches System ist meist ein physikalisches System, welches eine zeitliche Änderung erfährt und dessen Verhalten durch Differentialgleichungen beschrieben wird. Das zu regelnde System wird auch als Regelstrecke bezeichnet und der Systemausgang als Regelgröße oder Istwert. Durch die Rückführung des Systemausgangs entsteht ein Regelkreis und es lässt sich eine Regeldifferenz aus gewünschtem Zustandswert (Sollwert) und Istwert berechnen. Anhand des Regelfehlers bestimmt der Regler die Steuergröße, welche die Regelstrecke gezielt beeinflusst, um den Regelfehler zu minimieren beziehungsweise im Optimalfall zu Null zu regeln.

Ein häufig eingesetzter Regler ist der PID-Regler, welcher aus einem proportionalen, integralen und differentialen Anteil besteht. Abbildung 2.10 zeigt das Blockschaltbild eines solchen Reglers. Die Stellgröße $u(t)$ des PID-Reglers ist definiert durch Summe der drei Glieder (entspricht einer Parallelschaltung) und lässt sich mit

$$u(t) = \underbrace{K_p e(t)}_{P} + \underbrace{K_i \int_0^t e(\tau)\,d\tau}_{I} + \underbrace{K_d \frac{d}{dt} e(t)}_{D} \tag{2.13}$$

berechnen. Mithilfe der drei Konstanten K_p, K_i und K_d lässt sich der Regler einstellen. Diese müssen abhängig von der Regelstrecke angepasst werden, sodass der Regelfehler $e(t) = w(t) - y(t)$ zwischen Soll-und Istwert minimiert wird. Oft gestaltet es sich jedoch als schwierig, die Parameter des Reglers optimal einzustellen. Aufgrund von Änderungen der Systemeigenschaften, der Sollgröße oder auch Störungen bieten konstante Regelparameter meistens keine optimale Regelleistung. Um das Reglerverhalten zu optimieren, können die

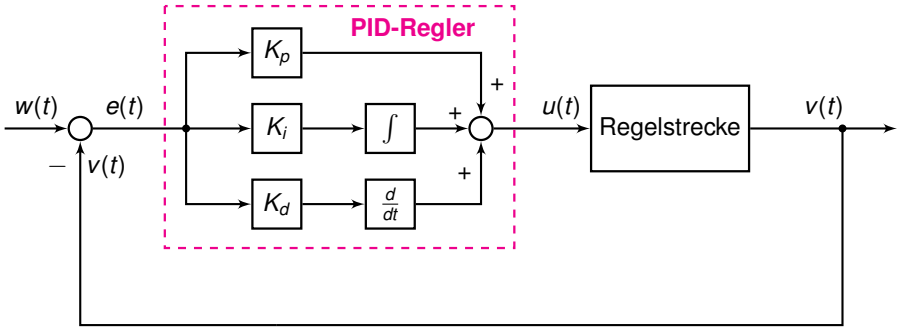


Figure 2.10: PID-Regler

Parameter abhängig vom Systemverhalten in Echtzeit angepasst werden. Diese Regelung wird als adaptive PID-Regelung bezeichnet [49]. Die Anpassung der Parameter kann durch ein neuronales Netz kontrolliert werden [50]. Im folgenden Kapitel wird die Implementierung eines adaptiven PID-Reglers beschrieben.

3 Implementierung eines adaptiven PID-Reglers

Das folgende Kapitel beschreibt, wie ein adaptiver PID-Regler mit Hilfe eines dynamischen neuronalen Netzes implementiert werden kann. Hierfür werden zunächst der allgemeine Aufbau des adaptiven PID-Reglers, sowie der Trainingsablauf des neuronalen Netzes erläutert. Anschließend werden die Simulation eines dynamischen Netzwerks und die Optimierung der Gewichte erklärt.

In Abbildung 3.1 ist zu sehen, wie die einzelnen Komponenten miteinander verschaltet sind. Die Struktur entspricht dem Regelkreis aus Abschnitt 2.3, nur werden hier die Parameter des Reglers mit Hilfe eines neuronalen Netzes in jedem Zeitschritt neu berechnet. Das Netz bekommt den aktuellen Regelfehler $e(t)$, sowie die Regelgröße $y(t)$ als Input und kann daraus die Parameter an die (zuvor gelernte) Charakteristik der Regelstrecke anpassen.

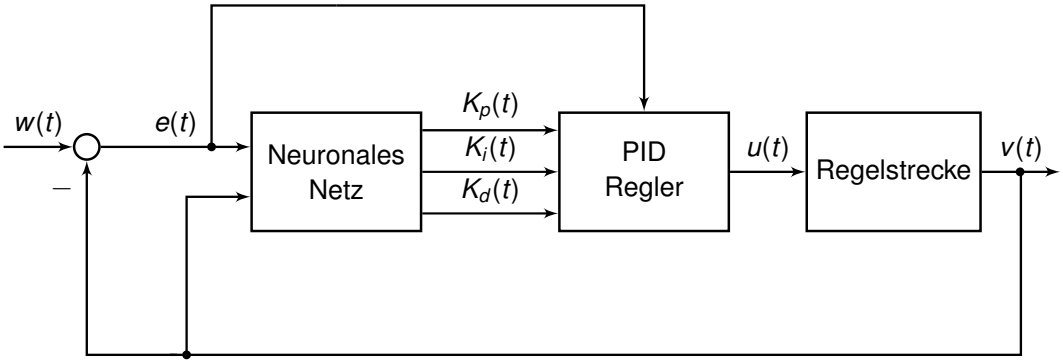


Figure 3.1: Regelkreis mit neuronalem Netz zum automatischen Einstellen der PID-Parameter

3.1 Trainieren dynamischer neuronaler Netze

Damit das Netz die Parameter des PID-Reglers automatisch einstellen kann, muss es zunächst trainiert werden. Das heißt, die Gewichte des Netzes müssen so angepasst werden, dass die Ausgangsneuronen gute Werte für die PID-Parameter liefern und der Regelfehler für beliebige Sollwerte $w(t)$ minimiert werden kann. Der zu minimierende Fehler ergibt sich dabei nicht am Ausgang des neuronalen Netzes, sondern am Systemausgang

der Regelstrecke. Die Kostenfunktion

$$E(\boldsymbol{x}, \boldsymbol{\omega}) = \frac{1}{2}\big(\boldsymbol{w} - \boldsymbol{v}(\boldsymbol{x}, \boldsymbol{\omega})\big)\big(\boldsymbol{w} - \boldsymbol{v}(\boldsymbol{x}, \boldsymbol{\omega})\big)^T \tag{3.1}$$

berechnet die Summe des quadratischen Fehlers zwischen Soll- und Istwert in Abhängigkeit des Gewichtsvektors $\boldsymbol{\omega}$ und für einen bestimmten Eingangsvektor $\boldsymbol{x}$. Das Optimierungsproblem lautet demnach

$$\arg\min_{\boldsymbol{\omega}} E(\boldsymbol{x}, \boldsymbol{\omega}). \tag{3.2}$$

Der Eingangsvektor $\boldsymbol{x}(t)$ des Netzes ist definiert durch den Regelfehler und den Systemzustand: $\big[e(t), \quad v(t)\big]^T$. Eine Anpassung der Gewichte erfolgt immer nach einer Sequenz von Trainingsdaten. Das ganze System aus Netz, Regler und Regelstrecke wird also zunächst für eine bestimmte Abfolge an Setpoints $\boldsymbol{w} = [w(0), ..., w(T)]$ simuliert und der resultierende Systemausgang $\boldsymbol{v} = [v(0), ..., v(T)]$ gespeichert. Anschließend können daraus die Fehlerfunktion berechnet und die Gewichte aktualisiert werden. Die Optimierung der Gewichte wird mit Hilfe des Levenberg-Marquardt Verfahrens durchgeführt, welches in Abschnitt 3.1.3 beschrieben wird. Zusätzlich wird der gesamte Trainingsablauf mit Hilfe von Pseudocode gezeigt.

Wie bereits erwähnt, erfordert das Training des Netzwerks ein Anregungssignal. Dafür wird die Führungsgröße $\boldsymbol{w}$ verwendet. Die Generierung eines solchen Signals wird im Folgenden erklärt.

3.1.1 Generierung von Trainingssignalen

Um das neuronale Netz auf das Einstellen eines PID-Reglers zu trainieren, werden angemessene Trainingsdaten benötigt. Mit Hilfe der Daten lernt das Netz Eigenschaften des Systemverhaltens, daher ist es wichtig, mit möglichst vielen Zuständen des Systems zu trainieren. Abbildung 3.2 zeigt ein Beispiel eines Anregungssignals. Generiert wird das Signal aus einer Abfolge von gleichverteilten Zufallszahlen, welche als Amplitudenwerte dienen. Die Zeitdauer eines Amplitudenwerts wird ebenfalls mit Hilfe einer Zufallszahl bestimmt. Um das Signal reproduzieren zu können, wird der Zufallsgenerator mit einem bestimmten Wert initialisiert. Zum Trainieren und Testen der Gewichte wird jeweils ein unterschiedlicher Wert verwendet, um sicherzustellen, dass die optimierten Gewichte auch für andere Signalverläufe ein gutes Ergebnis liefern. Man spricht hierbei von der *Generalisierungsfähigkeit* eines Netzes.

3.1.2 Simulation dynamischer neuronaler Netze

Für die Simulation dynamischer Netze ist es wichtig, die richtige Reihenfolge zu ermitteln, in welcher die Werte der einzelnen Neuronen berechnet werden müssen. Da diese Art von Netzen neben Vorwärts- auch Rückwärtsverbindungen, sowie verzögerte Verbindungen enthalten können, können die Werte nicht wie bei Feedforward-Netzen schichtenweise berechnet werden.

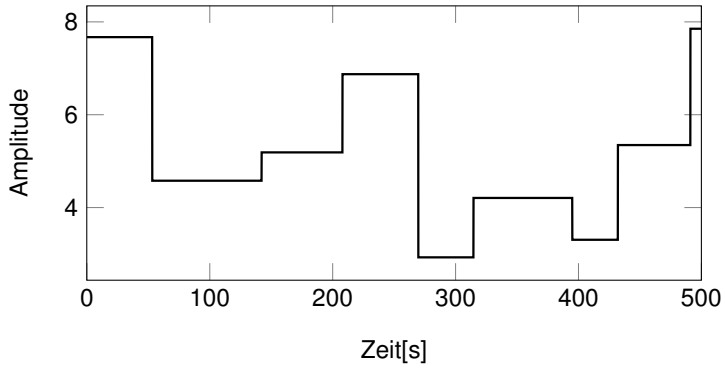


Figure 3.2: Beispiel eines zufälligen Anregungssignals

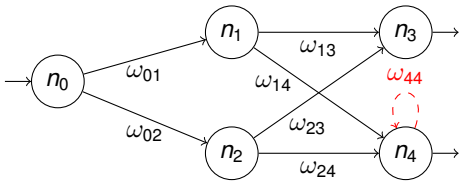


Figure 3.3: Beispielnetz mit Gewichten. Im Code speichert jedes Neuron seine Vorgängerneuronen, die zugehörigen Gewichtsindizes, sowie die Verzögerung der Verbindung. Zum Beispiel wird Neuron n_4 wie folgt dargestellt: $n_4 = [(n_1, n_2, n_4), (\omega_{14}, \omega_{24}, \omega_{44}), (0, 0, 1)]$.

Die Berechnung der Aktivierungen wurde mit Hilfe einer rekursiven Funktion implementiert. Für jedes Neuron im Netz wird eine Liste angelegt, welche seine jeweiligen Vorgängerneuronen speichert, die Gewichtsindizes der Verbindungen zu allen Vorgängern, sowie die Information, ob die jeweilige Verbindung verzögert ist. Abbildung 3.3 zeigt dies anhand eines Beispielnetzes. Zum Beispiel wird Neuron n_4 wie folgt dargestellt: $n_4 = [(n_1, n_2, n_4), (\omega_{14}, \omega_{24}, \omega_{44}), (0, 0, 1)]$ Wird die Funktion für ein Neuron aufgerufen, so werden rekursiv alle Aktivierungen der Vorgängerneuronen berechnet, welche für den Wert des Neurons relevant sind. Dieses Verfahren entspricht einer Tiefensuche, wodurch eine topologische Sortierung nicht mehr notwendig ist. Algorithmus 2 zeigt die eben erklärte Funktion.

Der Nachteil dieser Implementierung ist der hohe Zeitaufwand. Durch die Rekursivität wird zwar die richtige Reihenfolge zur Berechnung der Aktivierungen ermittelt, jedoch wird dies für jede Simulation durchgeführt. Da das Netz bei nur einer Trainingsiteration mehrere Male simuliert werden muss, ist diese Implementierung sehr ineffizient. Optimiert werden kann dies zum Beispiel, indem die rekursive Funktion dafür genutzt wird, die einzelnen Berechnungsschritte für jedes Neuron zu speichern und nur ein Mal vor der Optimierung aufgerufen wird. Hierfür werden zwei Schleifen benötigt: die erste entspricht

Funktion `berechne_aktivierung(`*neuron, t,* $\boldsymbol{\omega}$`) /* Rekursive Funktion */`
Daten : Zeit t, Gewichtsvektor $\boldsymbol{\omega}$,
Neuron n = [[Vorgänger], [Gewichtsindizes], [Verzögerungen]]
Ergebnis : Aktivierung $a_{t,neuron}$
wenn *a schon berechnet* **dann**
  zurück a
Ende
$a = 0$
für *p,* ω_{index}*, d in (n.Vorgänger, n.Gewichtsindizes, n.Verzögerungen)* **tue**
  a += `berechne_aktivierung(`*p, t - d,* $\boldsymbol{\omega}$`)` $\cdot\ \boldsymbol{\omega}[\omega_{\text{index}}]$
Ende
speichere $\tanh(a)$
zurück $\tanh(a)$

Algorithmus 2 : Berechnung der Aktivierungen

der Schleife aus Algorithmus 2 und ist notwendig, um alle Vorgänger für das aktuelle Neuron durchzugehen. Sobald alle Vorgänger eines Neurons besucht wurden, werden in der nächsten Schleife die notwendigen Informationen gespeichert. Hierfür werden vier Arrays für jeweils

- den Startindex jeder Kante ***s***
- den Zielindex jeder Kante ***e***
- den Gewichtsindex jeder Kante $\boldsymbol{\omega}_{index}$
- die Information, ob die Kante verzögert ist ***d***

angelegt. Algorithmus 3 zeigt den Code für das Speichern der Berechnungsschritte in Arrays.

Das Netz kann nun sehr einfach simuliert werden, indem die Elemente der Arrays zu einer Formel kombiniert werden. Die bereits berechneten Aktivierungen müssen währenddessen gespeichert werden, um die Werte für die Folgeneuronen wiederverwenden zu können. Hierfür wird ein weiteres Array $\boldsymbol{A} \in n \times 2$ angelegt. Es enthält zwei Spalten für jeweils

- alle aktuellen Aktivierungen $\boldsymbol{a}_t$
- alle vorherigen Aktivierungen $\boldsymbol{a}_{t-1}$.

In Algorithmus 4 werden die Aktivierungen mit Hilfe der Arrays berechnet. Je nachdem ob eine Verbindung verzögert ist oder nicht, wird Spalte 1 oder Spalte 0 der Matrix $\boldsymbol{A}$ verwendet.

Funktion `speicher_rechenschritte(`*neuron, t,* $\boldsymbol{\omega}$`) /* Rekursive Funktion */`
  Daten : Zeit t, Gewichtsvektor $\boldsymbol{\omega}$,
  Neuron n = [[Vorgänger], [Gewichtsindizes], [Verzögerungen]]
  Ergebnis : $\boldsymbol{s}, \boldsymbol{e}, \boldsymbol{\omega}, \boldsymbol{d} \in$ Anzahl Verbindungen $\times 1$
  wenn *neuron schon besucht* **dann**
    zurück
  Ende
  für p, ω_{index}, d *in (n.Vorgänger, n.Gewichteindizes, n.Verzögerungen)* **tue**
    `speicher_rechenschritte(`p, t - d, $\boldsymbol{\omega}$`)` $\cdot\ \boldsymbol{\omega}[w_{\text{index}}]$
  Ende
  für p, ω_{index}, d *in (n.Vorgänger, n.Gewichteindizes, n.Verzögerungen)* **tue**
    $\boldsymbol{s}[k], \boldsymbol{e}[k], \boldsymbol{\omega}[k], \boldsymbol{d}[k] = p,\ n,\ \omega_{\text{index}},\ d$
  Ende

Algorithmus 3 : Speichern der Schritte für die Berechnung der Aktivierungswerte

Daten : $\boldsymbol{s}$, $\boldsymbol{e}$, $\boldsymbol{d}$, $\boldsymbol{\omega}$, $\boldsymbol{\omega}_{index}$
Ergebnis : Aktivierungsmatrix $\boldsymbol{A}$
a = 0
für $i \leftarrow 0$ **bis** n **tue**
  a += $\boldsymbol{A}[\boldsymbol{s}[i], \boldsymbol{d}[i]] \cdot \boldsymbol{\omega}[\boldsymbol{\omega}_{index}[i]]$
  wenn ***e[i] != e[i+1]*** **dann** `// Neuron fertig`
    $\boldsymbol{A}[\boldsymbol{e}[i], 0] = \tanh(\text{a})$
    a = 0
  Ende
Ende

Algorithmus 4 : Optimierte Berechnung der Aktivierungen

3.1.3 Optimierung der Gewichte mithilfe des Levenberg-Marquardt-Algorithmus

In Kapitel 2.1.2 wurde der BP-Algorithmus erklärt, welcher die Gewichte mit Hilfe des Gradientenabstiegsverfahrens (GAV) optimiert. Wie effizient der Algorithmus ist, hängt in erster Line von der Schrittweite des GAVs ab. Ist der Gradient sehr steil, sollte die Schrittweite klein sein, um das Minimum nicht zu verfehlen. In Bereichen, in denen der Gradient flach ist, führt eine kleine Schrittweite jedoch zu langsamer Konvergenz. Verschiedene Varianten des BP Algorithmus konnten nur wenig Verbesserung hinsichtlich der Effizienz erzielen.

Verglichen mit dem Gradientenabstiegsverfahren besitzt das Gauss-Newton-Verfahren die Eigenschaft, sehr schnell zu konvergieren. Jedoch ist hier nicht sicher gestellt, dass es tatsächlich konvergiert, was wiederum ein Nachteil gegenüber dem GAV ist. Beim Gauss-Newton-Verfahren werden die Gewichte mit der Formel

$$\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_k - (\boldsymbol{J}_k^T \boldsymbol{J}_k)^{-1} \boldsymbol{J}_k \boldsymbol{E}_k \tag{3.3}$$

aktualisiert. Die Matrix $\boldsymbol{J}$ kennzeichnet dabei die Jacobimatrix welche durch die partiellen Ableitungen der Fehlerfunktion $E(\boldsymbol{\omega})$ nach allen Gewichten definiert ist. Die Berechnung der Jacobimatrix wird im folgenden Abschnitt 3.1.4 gezeigt. Der Levenberg-Marquardt-Algorithmus kombiniert das Gradientenabstiegsverfahren mit dem Gauss-Newton-Algorithmus und aktualisiert die Gewichte mit der Formel

$$\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_k - (\boldsymbol{J}_k^T \boldsymbol{J}_k + \lambda \boldsymbol{I})^{-1} \boldsymbol{J}_k \boldsymbol{E}_k. \tag{3.4}$$

Durch Anpassung des Parameters λ kann zwischen den beiden Verfahren gewechselt werden. Für $\lambda \ll \boldsymbol{J}_k^T \boldsymbol{J}_k$, entspricht das Verfahren dem Gauss-Newton-Algorithmus. Wird λ erhöht, sodass $\lambda \gg \boldsymbol{J}_k^T \boldsymbol{J}_k$, nähert sich der Algorithmus dem Gradientenabstiegsverfahren und es gilt

$$\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_k - (\lambda)^{-1} \boldsymbol{J}_k \boldsymbol{E}_k. \tag{3.5}$$

Der Koeffizient $\frac{1}{\lambda}$ entspricht dabei der Schrittweite α.

Während des Trainings wird λ in jeder Iteration angepasst. Die Idee dabei ist, bei einem großen Fehler den Parameter λ zu erhöhen, um mit Hilfe des GAVs in Richtung des steilsten Abstiegs zu gehen. Je größer λ ist, desto kleiner wird dabei die Schrittweite. Wird der Fehler wiederum kleiner, so kann λ verringert werden. Dadurch konvertiert das Verfahren zum Gauss-Newton Algorithmus und erzielt eine schnellere Konvergenz, sobald die Lösung nahe eines lokalen Optimums ist. Eine ausführliche Herleitung des Levenberg-Marquardt Verfahrens kann aus [51, 12-1ff] entnommen werden.

Algorithmus 5 zeigt, wie das Levenberg-Marquardt-Verfahren auf das Trainieren von dynamischen Netzen angewendet wurde.

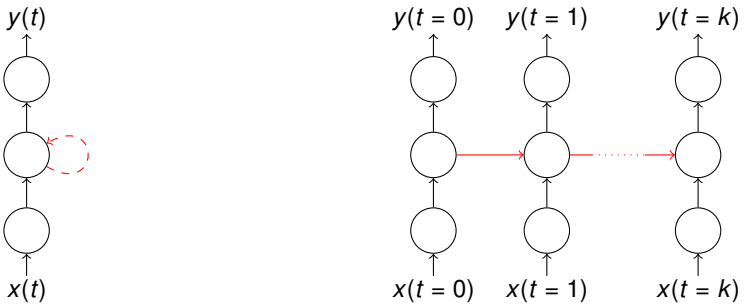
Daten : Sollwert $\boldsymbol{w} \in \mathbb{R}^n$, Anzahl der Trainings n_T, Anzahl der Epochen n_E, Testsignal $\boldsymbol{w}_\mathrm{test}$, Anzahl der Traningsdaten t_0, Eingangssignal Netz $\boldsymbol{x}$
Ergebnis : Optimaler Gewichtsvektor $\boldsymbol{\omega}_\mathrm{opt}$
$E_\mathrm{opt} = \inf$
$t = t_0$
für $k = 0$ **bis** n_T **tue**
  Initialisiere zufälligen Gewichtsvektor $\boldsymbol{\omega}$ */
  solange $t < n$ **tue**
    $\boldsymbol{w}_\mathrm{tmp} = \boldsymbol{w}[0:t]$ // Setze Anzahl der Trainingsdaten
    für $k = 0$ **bis** n_E **tue**
      $(\boldsymbol{J}^T\boldsymbol{J} + \lambda \boldsymbol{I})\boldsymbol{\sigma} = \boldsymbol{J}^T(\boldsymbol{w}_\mathrm{tmp} - \boldsymbol{y}(\boldsymbol{x}, \boldsymbol{\omega}))$ // Löse Gleichungssystem für $\boldsymbol{\sigma}$
      wenn $\sum(\boldsymbol{w}_{tmp} - \boldsymbol{v}(\boldsymbol{x}, \boldsymbol{\omega} + \boldsymbol{\sigma}))^2 < \sum(\boldsymbol{w}_{tmp} - \boldsymbol{v}(\boldsymbol{x}, \boldsymbol{\omega}))^2$ **dann**
        $\boldsymbol{\omega} = \boldsymbol{\omega} + \boldsymbol{\sigma}$, $\lambda = \frac{\lambda}{2}$
      sonst
        $\lambda = 2\lambda$
      Ende
    Ende
    Erhöhe aktuelle Anzahl t der Trainingsdaten
  Ende
  Berechne Fehler E_test für Testsignal $\boldsymbol{w}_\mathrm{test}$
  wenn $E_{test} < E_{opt}$ **dann**
    $\boldsymbol{\omega}_\mathrm{opt} = \boldsymbol{\omega}$
    $E_\mathrm{opt} = E_\mathrm{test}$
  Ende
Ende
zurück $\boldsymbol{\omega}_\mathrm{opt}$

Algorithmus 5 : Trainieren eines dynamischen neuronalen Netzes mit Hilfe des Levenberg-Marquardt-Algorithmus

3.1.4 Berechnung der Jacobimatrix

Der Levenberg-Marquardt Algorithmus erfordert die Berechnung der Jacobimatrix. Da dynamische Netze einen internen Zustand besitzen und deren Ausgangswerte nicht allein vom Eingangsvektor abhängen, kann die Jacobimatrix nicht standardmäßig in einem gegebenen Punkt berechnet werden.

Wie bereits in Kapitel 2.1.3 erläutert, bilden Netze mit verzögerten Verbindungen eine Eingangssequenz auf eine Ausgangssequenz ab, ausgehend von einem bestimmten internen Startzustand. Ein dynamisches Netz lässt sich als FNN darstellen, indem die verzögerten Verbindungen zeitlich entfaltet werden [41]. Abbildung 3.4 zeigt diesen Prozess anhand eines Beispiels. Die zeitlich verzögerte Verbindung aus 3.4a wird entfaltet, indem das Netz für jeden Zeitschritt dargestellt wird. Mit Hilfe der zeitlichen Entfaltung kann die Ja-



(a) Neuronales Netz mit verzögerter Verbindung **(b)** Zeitliche Entfaltung

Figure 3.4: Zeitliche Entfaltung eines neuronalen Netzes mit verzögerter Verbindung

cobimatrix eines dynamischen Netzes für eine bestimmte Sequenz an Eingangsvektoren $\boldsymbol{x}_t \in \mathbb{R}^n$ berechnet werden, wobei t den Zeitschritt kennzeichnet und n die Gesamtanzahl der Zeitschritte. Laut [37] ist es ausreichend, die partiellen Ableitungen der Ausgangsvektoren zu berechnen, anstelle die der Fehlerfunktion. Angewendet auf den adaptiven PID-Regler, werden nicht die Ausgangsvektoren des Netzes abgeleitet, sondern die Systemausgänge $\boldsymbol{v}(\boldsymbol{x}, \boldsymbol{\omega}) \in \mathbb{R}^m$ der Regelstrecke. Daraus folgt die Formel für die Jacobimatrix

$$\boldsymbol{J}(\boldsymbol{x},\boldsymbol{\omega_i}) = \begin{pmatrix} \frac{\partial}{\partial\omega_0} v_0(\boldsymbol{x}_0,\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_0(\boldsymbol{x}_0,\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_0(\boldsymbol{x}_0,\boldsymbol{\omega_i}) \\ \frac{\partial}{\partial\omega_0} v_0(\boldsymbol{x}_1,\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_0(\boldsymbol{x}_1,\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_0(\boldsymbol{x}_1,\boldsymbol{\omega_i}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial\omega_0} v_0(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_0(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_0(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) \\ \frac{\partial}{\partial\omega_0} v_1(\boldsymbol{x}_0,\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_1(\boldsymbol{x}_0,\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_1(\boldsymbol{x}_0,\boldsymbol{\omega_i}) \\ \frac{\partial}{\partial\omega_0} v_1(\boldsymbol{x}_1,\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_1(\boldsymbol{x}_1,\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_1(\boldsymbol{x}_1,\boldsymbol{\omega_i}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial\omega_0} v_1(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_1(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_1(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial\omega_0} v_{m-1}(\boldsymbol{x}_0,\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_{m-1}(\boldsymbol{x}_0,\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_{m-1}(\boldsymbol{x}_0,\boldsymbol{\omega_i}) \\ \frac{\partial}{\partial\omega_0} v_{m-1}(\boldsymbol{x}_1,\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_{m-1}(\boldsymbol{x}_1,\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_{m-1}(\boldsymbol{x}_1,\boldsymbol{\omega_i}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial\omega_0} v_{m-1}(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) & \frac{\partial}{\partial\omega_1} v_{m-1}(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) & \cdots & \frac{\partial}{\partial\omega_{k-1}} v_{m-1}(\boldsymbol{x}_{n-1},\boldsymbol{\omega_i}) \end{pmatrix} \in \mathbb{R}^{nm\times k}. \tag{3.6}$$

Eine analytische Berechnung der Jacobimatrix würde hierbei einen hohen Rechenaufwand erfordern. Mit Hilfe der finiten Rückwärtsdifferenz kann die Jacobimatrix numerisch approximiert werden [52], sodass $\hat{\boldsymbol{J}} \approx \boldsymbol{J}$:

$$\hat{\boldsymbol{J}} = \begin{pmatrix} \boldsymbol{j}_0 \\ \boldsymbol{j}_1 \\ \vdots \\ \boldsymbol{j}_k \end{pmatrix}^T,\ \boldsymbol{j}_i = \frac{\boldsymbol{v}(\boldsymbol{x},\boldsymbol{\omega}) - \boldsymbol{v}(\boldsymbol{x},\boldsymbol{\omega} - \boldsymbol{h}\varepsilon(\omega_i))}{\varepsilon(\omega_i)},\ h_p = \begin{cases} 1, & p = i \\ 0, & p \neq i \end{cases}. \tag{3.7}$$

Eine Spalte der Jacobimatrix ist durch $\boldsymbol{j}_i$ definiert. Ausgehend vom aktuellen Systemzustand $\boldsymbol{y}(\boldsymbol{x},\boldsymbol{\omega})$ wird jeweils ein Gewicht ω_i um $\varepsilon(\omega_i)$ variiert und der Ausgangswert berechnet. Der Vektor $\boldsymbol{h}$ legt fest, welches Gewicht durch die Schrittweite ε abgeändert wird. Die Schrittweite wird hierbei abhängig vom aktuellen Gewicht mit Hilfe der Formel

$$\varepsilon(\omega_i) = \max(1, |\omega_i|)\sqrt{\varepsilon_{min}} \tag{3.8}$$

berechnet. Die Konstante ε_{min} kennzeichnet dabei die Maschinengenauigkeit. Algorithmus 6 zeigt die numerische Berechnung der Jacobimatrix.

Daten : System: $\boldsymbol{v}(\boldsymbol{x}, \boldsymbol{\omega}) \in \mathbb{R}^m$, Netzeingang $\boldsymbol{x} \in \mathbb{R}^n$, Gewichtsvektor $\boldsymbol{\omega} \in \mathbb{R}^k$
Ergebnis : Approximation der Jacobimatrix $\hat{\boldsymbol{J}} \in \mathbb{R}^{nm \times k}$
für $j = 0$ **bis** k **tue**
  $\varepsilon = \max(1, |\omega_j|)\sqrt{\varepsilon_{min}}$
  für $t = 0$ **bis** $n-1$ **tue**
    $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{x}_t, \boldsymbol{\omega})$, $\hat{\boldsymbol{v}} = \boldsymbol{v}(\boldsymbol{x}_t, \boldsymbol{\omega} - \boldsymbol{h}\varepsilon)$
    für $i = 0$ **bis** $m-1$ **tue**
      $\hat{\boldsymbol{J}}(t \cdot m + i, k) = \frac{\boldsymbol{v} - \hat{\boldsymbol{v}}}{\varepsilon}$
    Ende
  Ende
Ende
zurück $\hat{\boldsymbol{J}}$

Algorithmus 6 : Numerische Berechnung der Jacobimatrix

4 Optimierung der Netzarchitektur mithilfe eines genetischen Algorithmus

In diesem Kapitel wird die Implementierung eines genetischen Algorithmus erklärt, welcher die Netztopologie dynamischer neuronaler Netze mit beliebigen Verbindungen optimieren soll.

Der erste Abschnitt beschreibt, wie die Netze im genetischen Algorithmus repräsentiert werden. Anschließend wird der gesamte Ablauf des Algorithmus geschildert und einzelne Schritte wie die Generierung der Startpopulation, die Transformation zwischen den Darstellungsformen, die Evaluierung der Fitness, sowie der Crossover Operator werden im Detail erklärt. Zuletzt wird auf die Methoden zur Verbesserung der Laufzeit eingegangen.

4.1 Codierung dynamischer Netze

Der genetische Algorithmus optimiert die Individuen auf Basis der Genotypen, die Repräsentation der Individuen im genetischen Algorithmus ist daher ein zentraler Aspekt. Die Darstellungsform der Individuen muss so gewählt werden, dass die zu optimierenden Parameter repräsentiert werden und der Algorithmus diese mit Hilfe der genetischen Operatoren optimieren kann.

Ziel des genetischen Algorithmus ist es, die Verbindungen unter den Neuronen so zu wählen, dass der Trainingsfehler minimal wird. Hierbei soll der Algorithmus auch die Möglichkeit haben, die Anzahl der Neuronen im Netz zu variieren. Die Anzahl der Eingangs- sowie Ausgangsneuronen wird anfangs festgelegt und wird dabei nicht verändert, da diese durch das Problem definiert ist. Neben den Verbindungen selber soll auch die Anzahl der Verbindungen im Netz optimiert werden, da jede zusätzliche Verbindung die Laufzeit eines Trainings erheblich erhöht.

Da hier beliebige Netze ohne eine Schicht-Struktur generiert werden, bietet sich die Darstellung mit Hilfe einer Adjazenzmatrix $\boldsymbol{V} \in n \times n$ an, wobei n der Anzahl der Neuronen im Netz entspricht. Eine Adjazenzmatrix speichert die Verbindungen zwischen den einzelnen Neuronen, wobei die Zeilen den Startknoten und die Spalten den Zielknoten entsprechen. Besteht eine Verbindung von Neuron i zu Neuron j, so enthält die Matrix eine 1 in der $i-$ten Zeile und $j-$ten Spalte. Eine 0 kennzeichnet dementsprechend, dass keine Verbindung existiert.

Mit dieser Darstellung können Netze entstehen, welche verzögerungsfreie Zyklen enthalten. Wie bereits in Kapitel 2.1.3 erklärt, ist ein Netz mit Zyklen nicht simulierbar. Die

Bildung von verzögerungsfreien Zyklen wird verhindert, indem Elemente auf der Diagonalen der Matrix nicht berücksichtigt werden, ebenso wie Elemente der unteren Dreiecksmatrix. Dadurch werden keine Netze ausgeschlossen, da ein Netz ohne Zyklen topologisch sortiert werden kann, um eine Darstellung als Dreiecksmatrix zu erhalten.

Für die Repräsentation verzögerter Verbindungen wird eine zweite binäre Matrix hinzugefügt, sodass man eine Matrix $\boldsymbol{V} \in n \times 2n$ erhält. Eine 1 steht dabei für eine Verbindung, die um eine Zeiteinheit verzögert ist. Größere Verzögerungen werden hier nicht repräsentiert. Für die verzögerten Verbindungen kann die volle Matrix verwendet werden, da diese Verbindungen nicht zu verzögerungsfreien Zyklen führen. Welche Neuronen die Eingangsvektoren erhalten, wird in einem 3. Teil gespeichert. Hierbei entspricht die Spalte dem jeweiligen Eingangsneuron und die Zeile dem Zielneuron. Die Ausgangsneuronen werden vorher festgelegt. Die gesamte Matrix besitzt schließlich die Dimension $\boldsymbol{V} \in n \times 2n+z$, wobei z der Anzahl der Eingangsvektoren entspricht. Abbildung 4.1 zeigt die gesamte Darstellung eines Netzwerks anhand eines Beispiels. Die Bias-Verbindungen sind in dieser Darstellung nicht enthalten, da diese nicht optimiert werden und das Bias-Neuron automatisch eine Verbindung zu jedem Neuron bekommt.

	n_0	n_1	n_2	n_3	n_4	n_0	n_1	n_2	n_3	n_4	x_0	x_1
n_0	*	1	1	0	0	0	0	0	0	0	1	0
n_1	*	*	0	1	0	0	0	0	0	0	0	0
n_2	*	*	*	1	1	0	1	0	0	0	0	1
n_3	*	*	*	*	0	1	0	0	0	0	0	0
n_4	*	*	*	*	*	0	0	0	0	0	0	0

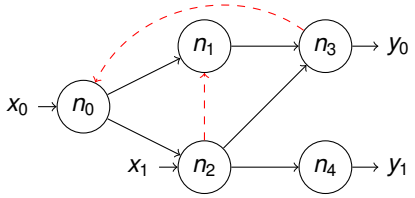


Figure 4.1: Darstellung eines Individuums im genetischen Algorithmus mit Hilfe einer Verbindungsmatrix: Der erste Teil der Matrix repräsentiert Verbindungen ohne Verzögerung, der zweite Teil Verbindungen mit einer Verzögerung von 1. Der dritte Teil speichert die Verbindungen ausgehend von den Eingangsneuronen. Die Eingangssignale u_0 und u_1 werden hier an n_0 und n_2 übergeben. Die Neuronen n_3 und n_4 dienen als Ausgänge, welche mit y_0 und y_1 gekennzeichnet sind.

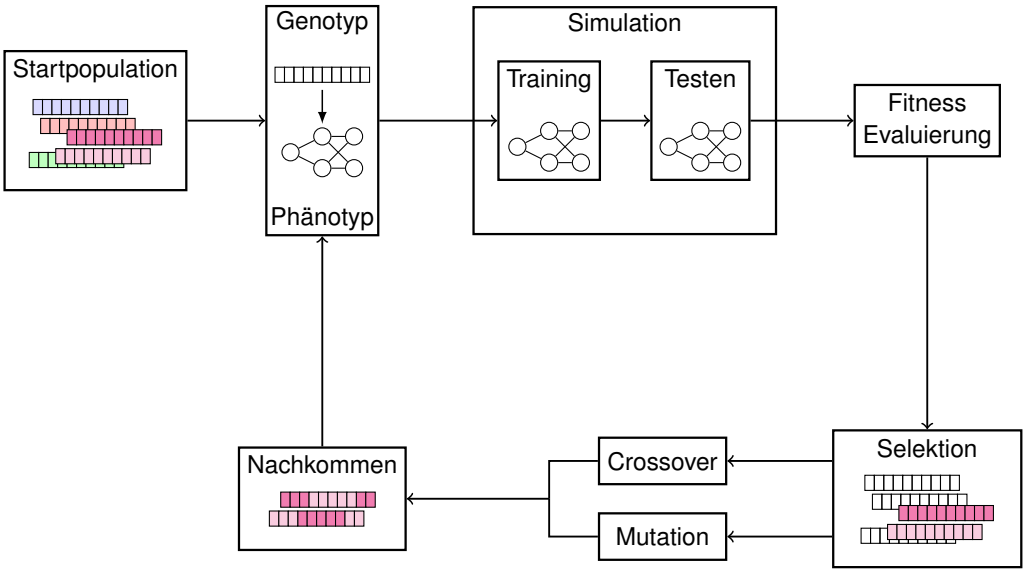


Figure 4.2: Ablauf des verwendeten genetischen Algorithmus zur Optimierung der Netztopologie [19]

4.2 Implementierung des genetischen Algorithmus

Der grundlegende Ablauf eines genetischen Algorithmus wurde bereits in Kapitel 2.2.1 behandelt. Die einzelnen Schritte des Algorithmus sollen nun in Bezug auf die Optimierung der Netztopologie erklärt werden. Abbildung 4.2 stellt die einzelnen Schritte schematisch dar. Im ersten Schritt wird eine Startpopulation erstellt, wobei die Individuen in der Matrixdarstellung aus 4.1 repräsentiert werden. Anschließend werden alle Individuen in die Form aus Kapitel 3.1.2 gebracht, welche vier Arrays für jeweils alle Startindizes, Zielindizes und Gewichtsindizes, sowie den Wert der Verzögerung einer Verbindung speichert. Damit kann erneut Algorithmus 5 zum Trainieren der Netze verwendet werden. Beide Repräsentationsformen der Individuen werden dabei gespeichert, sodass die Umwandlung pro Individuum nur ein Mal durchgeführt werden muss. Das Training liefert für jedes Individuum einen Fehler, mit dessen Hilfe die Netze bewertet werden. Die genauere Berechnung der Fitnessfunktion wird in Kapitel 5.3 erläutert. Anhand der Fitness erfolgt die Selektion der Eltern, welche nach Anwendung der genetischen Operatoren neue Nachkommen produzieren. Die Nachkommen werden ebenfalls in die “Netzdarstellung” umgewandelt und trainiert und können dann in die Population eingegliedert werden. Das Einfügen der neuen Nachkommen in die Population ist in Abbildung 4.2 nicht dargestellt.

4.2.1 Generierung einer Zufallspopulation

Die Erstellung einer Startpopulation erfolgt durch die Generierung zufälliger binärer Matrizen in Form der Matrizen aus Abschnitt 4.1. Die Anzahl der Neuronen kann beliebig gesetzt werden und bestimmt die Größe der Matrizen, wobei die Anzahl der Eingangs- und Ausgangsneuronen durch das Problem definiert ist. Um die Anzahl der Verbindungen der Matrizen in der Initialpopulation zu kontrollieren, wird jeder Eintrag einer Matrix mit einer bestimmten Wahrscheinlichkeit unabhängig voneinander auf 1 gesetzt.

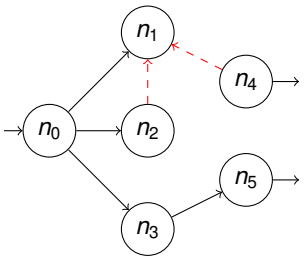
Bei der Generierung zufälliger Matrizen (und später auch bei der Anwendung von Crossover und Mutation) kann es passieren, dass Netze entstehen, welche überflüssige Verbindungen enthalten. Mit überflüssig sind Neuronen gemeint, welche Eingangssignale von anderen Neuronen erhalten, aber keine Nachfolger besitzen und somit keinen Einfluss auf die Ausgangsneuronen haben. Umgekehrt können Neuronen entstehen, deren Ausgangswert zwar zu anderen Neuronen führt, welche aber keine Vorgänger besitzen und somit das ursprüngliche Eingangssignal nicht verarbeiten. Überflüssige Kanten kosten Rechenzeit, ohne einen Gewinn zu bringen. Bevor ein Netz zur Population hinzugefügt wird, wird daher zunächst überprüft, ob es die eben erklärte Bedingung erfüllt.

Eine Möglichkeit hier entgegenzuwirken, wäre der Ausschluss solcher Netze und die in Kapitel 2.2.2 erklärte Todesstrafe anzuwenden. Eine effizientere Methode besteht darin, die überflüssigen Verbindungen zu ermitteln und zu löschen, indem die entsprechenden Elemente der Verbindungsmatrix auf 0 gesetzt werden. Abbildung 4.3 zeigt den Vorgang anhand eines Beispielnetzes. Neuron n_1 besitzt offensichtlich keine Nachfolger, daher können alle Eingangskanten gelöscht werden. Anschließend muss überprüft werden, ob dadurch neue nachfolgerlose Neuronen entstanden sind, wie es für n_2 der Fall ist. Nach Entfernung der Verbindungen besteht das Netz effektiv aus den Neuronen n_0, n_3 und n_5, wobei das 2. Ausgangsneuron n_4 auch erhalten bleiben muss. Eine Ausnahme stellen die Eingangs- und Ausgangsneuronen dar, welche keine Vorgänger beziehungsweise keine Nachfolger benötigen. Implementiert werden kann die Eliminierung der Verbindungen zum Beispiel mit Hilfe einer Tiefensuche. Diese wird in beide Richtungen ausgeführt, einmal von den Eingangsneuronen und einmal von den Ausgangsneuronen aus startend. Die Neuronen, die in beide Richtungen besucht werden, bleiben erhalten.

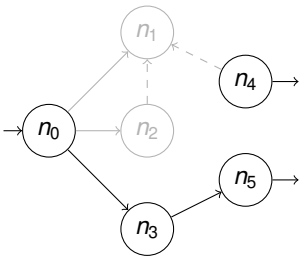
Diese Methode minimiert die Simulationszeit eines Netzes (sofern es überflüssige Kanten enthält) und ermöglicht dem Algorithmus zudem, die Anzahl der Neuronen zu minimieren. Sobald die Startpopulation generiert wurde, werden die Individuen in ihre Phänotypen transformiert.

4.2.2 Transformation Genotyp zu Phänotyp

Um die Qualität der neuronalen Netze bewerten zu können, müssen sie trainiert werden. Hierfür werden die Netze in die Darstellung aus Kapitel 3.1.2 transformiert, um erneut Algorithmus 4 zum Berechnen der Ausgangsneuronen verwenden zu können. Da die Verbindungsmatrix der Individuen eine Dreiecksmatrix ist, sind die Netze bereits topologisch



(a) Das Neuron n_1 besitzt keine Nachfolger und ist somit überflüssig.



(b) Alle Verbindungen zu n_1 werden entfernt. Dadurch besitzt auch n_2 keine Nachfolger mehr und die Kante $n_0 \rightarrow n_2$ wird ebenfalls entfernt.

Figure 4.3: Beispiel eines RNNs mit überflüssigen Verbindungen: $n_0 \rightarrow n_1$, $n_2 \rightarrow n_1$, $n_4 \rightarrow n_1$, $n_0 \rightarrow n_2$.

sortiert, was die Berechnung der Ausgänge wesentlich erleichtert. Eine vorherige Sortierung mit Algorithmus 3 ist daher nicht notwendig. Algortihmus 7 zeigt, wie die Arrays für alle Start- und Zielknoten, sowie für die Verzögerungen aus der Matrix gebildet werden. Jede Spalte der Matrix $\boldsymbol{V}$ signifiziert einen Zielknoten, dessen Vorgänger an den Zeilen abgelesen werden können. Die Verzögerung wird entsprechend gesetzt, je nachdem ob sich die Verbindung in der linken oder rechten Matrixhälfte befindet.

Daten : Verbindungsmatrix $\boldsymbol{V} \in \mathbb{R}^n$
Ergebnis : Startknoten $\boldsymbol{s}$, Zielknoten $\boldsymbol{e}$, Verzögerungen $\boldsymbol{d}$
k = 0
für $c \leftarrow 0$ **bis** n **tue**
  für $r \leftarrow 0$ **bis** n **tue**
    wenn *V[r, c] == 1* **dann** `// Verbindung ohne Verzögerung`
      $\boldsymbol{s}[k], \boldsymbol{e}[k], \boldsymbol{d}[k] = r, c, 0$
      $k+ = 1$
    Ende
    wenn *V[r, c + n] == 1* **dann** `// verzögerte Verbindung`
      $\boldsymbol{s}[k], \boldsymbol{e}[k], \boldsymbol{d}[k] = r, c, 1$
      $k+ = 1$
    Ende
  Ende
Ende
zurück $\boldsymbol{s}$, $\boldsymbol{e}$, $\boldsymbol{d}$

Algorithmus 7 : Transformation von Genotyp zu Phänotyp

Anschließend wird der Regler zusammen mit der Regelstrecke simuliert und der Fehler am Systemausgang berechnet. Der Fehler jedes Individuums wird gespeichert, sodass in allen weiteren Generationen nur die neuen Nachkommen trainiert werden müssen.

4.2.3 Anwendung von Crossover und Mutation

Der Austausch von Genmaterial zweier Individuen erfolgt nach dem selben Schema, das bereits in Kapitel 2.2.3 erklärt wurde. Anhand von zwei Beispielnetzen (Abbildung 4.4) soll das Crossover für die Darstellungsform aus 4.1 gezeigt werden. In Abbildung 4.4a sind zwei Netze und deren Darstellung als Matrix zu sehen. Um zwei Individuen miteinander zu kreuzen, werden die Matrizen zeilenweise in Vektoren umgeschrieben. Die untere Dreiecksmatrix sowie die Hauptdiagonale werden dabei ausgeschlossen. Abbildung 4.4b zeigt die beiden Matrizen in Vektordarstellung, wobei die Zeichenketten, die ausgetauscht werden sollen, grün markiert sind. Angewendet wird hierbei das Two-Point-Crossover mit den Kreuzpunkten $p_1 = 11$ und $p_2 = 21$.

Die neu entstandenen Vektoren werden nach dem selben Prinzip wieder in Matrizen umgeschrieben. Abbildung 4.4c veranschaulicht die Netzstruktur der neuen Nachkommen und die zugehörigen Matrizen. Vergleicht man die Eltern mit deren Nachkommen, so lässt sich erkennen, dass die Kanten $n_2 \rightarrow n_1$, sowie $n_1 \rightarrow n_3$ in Graph I. mit den Kanten $n_1 \rightarrow n_0$, $n_2 \rightarrow n_0$, $n_1 \rightarrow n_1$ und $n_1 \rightarrow n_4$ aus Graph II. vertauscht wurden.

4.2.4 Laufzeitoptimierung

Die gesamte Implementierung erfolgte in der Programmiersprache Python. Python ist eine sehr mächtige Sprache, die viele nützliche Module besitzt und in zahlreichen Bereichen eingesetzt wird.

Die Laufzeit des genetischen Algorithmus wurde mit Hilfe von zwei Python-Modulen optimiert. Zum einen bietet es sich im genetischen Algorithmus an, die Auswertung der Individuen parallel durchzuführen. Das heißt, dass sowohl die Netze der Startpopulation, als auch neue Nachkommen (innerhalb einer Generation) gleichzeitig trainiert werden können, da die Trainings unabhängig voneinander sind. Für die Parallelisierung der Trainings wurde das Python-Modul *Multiprocessing* [1] verwendet, mit dessen Hilfe Aufgaben in mehrere Prozesse aufgeteilt werden können und welches die auf einer Maschine maximal zur Verfügung stehenden Prozessoren parallel nutzt.

Im Algorithmus werden in jeder Generation drei neue Netze evaluiert, welche jeweils zehn Mal mit unterschiedlichen Startgewichten trainiert werden. Die Startgewichte sind dabei voneinander unabhängige Zufallszahlen. Um Multiprocessing optimal zu nutzen, werden nicht die kompletten Trainings der drei Netze parallelisiert, sondern die einzelnen Trainingsrunden aller Netze, sodass sich $3 \cdot 10$ Aufgaben ergeben und aufteilen lassen: [$Netz_0(\boldsymbol{\omega}_0)$, $Netz_0(\boldsymbol{\omega}_1)$, ..., $Netz_0(\boldsymbol{\omega}_9)$, $Netz_1(\boldsymbol{\omega}_0)$, ..., $Netz_1(\boldsymbol{\omega}_9)$, ..., $Netz_2(\boldsymbol{\omega}_9)$].

Eine weitere Optimierung wurde durch Verwendung der just-in-time (JIT) Funktionalität des Moduls *Numba* [53] durchgeführt. Diese wandelt unter anderem Quellcode in Maschinencode um und verkürzt damit die Laufzeit. Laut der Dokumentation[2] kann Python-Code damit ähnlich schnell wie in C geschriebener Code kompiliert werden.

[1] `https://docs.python.org/2/library/multiprocessing.html`
[2] `http://numba.pydata.org`

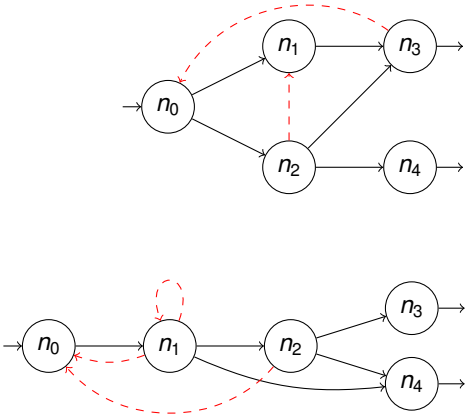
	Zeit [s]	relative Zeitdauer
Ohne Optimierung	18109.79	100.0 %
JIT	885.38	4.9 %
Multiprocessing	5403.66	29.8 %
JIT & Multiprocessing	532.46	2.9 %

Table 4.1: Laufzeitoptimierung des genetischen Algorithmus mit Hilfe verschiedener Module

Der größte Anteil der Laufzeit liegt in der Simulation der Netzwerke. Um die Effektivität der Module zu evaluieren, wurde ein kleines Testprogramm geschrieben, welches drei Netze jeweils 10 Mal für 5 Generationen trainiert. Tabelle 4.1 zeigt die Laufzeiten für das Programm ohne Optimierung, sowie jeweils mit JIT oder Multiprocessing und unter Verwendung beider Module. Insgesamt konnte die Implementierung um das 34-fache beschleunigt werden.

I.
$$\left(\begin{array}{ccccc|ccccc|c} * & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ * & * & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

II.
$$\left(\begin{array}{ccccc|ccccc|c} * & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ * & * & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ * & * & * & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$



(a) Neuronale Netze mit zugehöriger Matrixdarstellung. Die beiden Netze dienen als Eltern und sollen miteinander gekreuzt werden.

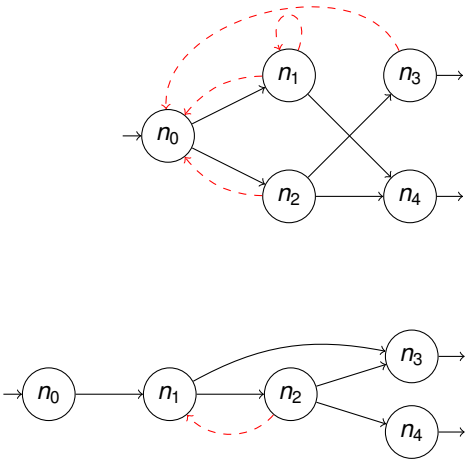
1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Vektordarstellung der Verbindungsmatrizen der beiden Elternnetze aus 4.4a. Die grün markierten Substrings sollen ausgetauscht werden.

I.
$$\left(\begin{array}{ccccc|ccccc|c} * & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ * & * & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ * & * & * & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 \\ * & * & * & * & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

II.
$$\left(\begin{array}{ccccc|ccccc|c} * & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ * & * & 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ * & * & * & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$



(c) Neuronale Netze der Nachkommen mit Matrixdarstellung. Nach Austausch der Substrings (siehe 4.4b) der beiden Elternnetze aus 4.4a entstehen neue Nachkommen. Die fett geschriebenen Zahlen der Matrizen kennzeichnen die ausgetauschten Zahlenketten und entsprechen den grün markierten Zahlen in 4.4b.

Figure 4.4: Two-Point-Crossover anhand eines Beispiels

5 Versuchsaufbau

Das folgende Kapitel beschreibt den Versuchsaufbau des adaptiven PID-Reglers, sowie des genetischen Algorithmus. Zusätzlich wird eine Netzstruktur vorgestellt, welche zur Verifikation des adaptiven PID-Reglers verwendet wurde.

5.1 Adaptiver PID-Regler

Im Folgenden werden die verwendeten Einstellungen des adaptiven PID-Reglers aufgeführt. In Kapitel 3 wurde der Aufbau des adaptiven PID-Reglers erklärt. Dieser setzt sich zusammen aus Netz, Regler und System.

Die implementierten neuronalen Netze besitzen zwei Eingangsneuronen, jeweils für den Systemzustand der Regelstrecke und den Regelfehler, sowie drei Ausgangsneuronen, welche die PID-Parameter K_p, K_i und K_d liefern. Die Anzahl der verstecken Neuronen kann beliebig gesetzt werden. Als Aktivierungsfunktion für die Eingangs- sowie Ausgangsneuronen wird die Identität, für die versteckten Neuronen hingegen wird der Tangens Hyperbolikus $f(x) = \tanh(x)$ verwendet.

Das Training der Netze erfolgt für mehrere, zufällig gesetzte Startgewichte, um eine gute Lösung zu erhalten. Die genaue Trainingsanzahl wird in Kapitel 6.1 erläutert. Aus jedem Training resultiert ein Gewichtsvektor, welcher unter Verwendung eines vom Trainingssignal verschiedenen Sollwertsignals validiert wird. Die endgültige Auswertung eines Gewichtsvektors erfolgt anhand des RMSE (auf englisch root mean square error), definiert durch

$$\text{RMSE} = \sqrt{\frac{(\boldsymbol{y} - \boldsymbol{w})^T (\boldsymbol{y} - \hat{\boldsymbol{y}})}{n}}. \tag{5.1}$$

Dabei entspricht n der Anzahl der Daten und $\boldsymbol{w}$ der Sollwertkurve des Validierungssignals. Die Gewichte, welche am Ende den geringsten RMSE liefern, werden letztendlich im Regelkreis für die automatische Berechnung der PID-Parameter verwendet.

Das Training erfolgt für ein Signal von 3500 Zeitschritten. Die anschließende Validierung wird für ein Signal der Länge 10000 durchgeführt. Um die Implementierung einfach zu halten, wird sowohl für den PID-Regler, als auch für das Netz dieselbe Schrittweite von 0.1 s verwendet.

Die Implementierung des adaptiven PID-Reglers wurde zunächst mit Hilfe eines RNNs mit fester Netzstruktur getestet. Die verwendete Netzstruktur ist in Abbildung 5.1 zu sehen. Die Eingangsschicht besitzt zwei Neuronen, wobei n_0 und n_1 den reinen Eingangsvektor erhalten. Das Netz enthält 2 verdeckte Schichten, wobei die Ausgangswerte der Neuronen

der zweiten versteckten Schicht als verzögertes Input an die erste versteckte Schicht zurückgeführt werden. Die Neuronen n_7, n_8 und n_9 dienen als Ausgangsneuronen.

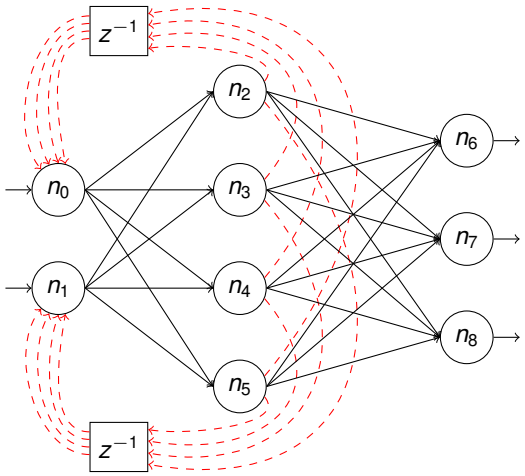


Figure 5.1: Feste Netzstruktur des neuronalen Netzes zur auotmatischen Berechnung der PID-Parameter

5.2 Nichtlineares Zweitank-System

Der adaptive PID-Regler wurde für die Regelung eines nichtlinearen Zweitank-Systems angewendet. Abbildung 5.2 zeigt den Aufbau des Systems. Es besteht aus zwei Tanks, die miteinander gekoppelt sind. Der erste Tank wird mit Hilfe einer Pumpe mit Wasser gefüllt. Das Wasser kann über ein Rohr weiter in den zweiten Tank fließen. Das Regelungsproblem besteht darin, den Wasserstand des zweiten Tanks $v(t) = z_2(t)$ auf einen gegebenen Sollwert zu regeln. Der Wasserzufluss im ersten Tank wird über ein Ventil reguliert, wobei die Stellung des Ventils durch die Größe $u(t)$ kontrolliert wird. Daduch kann nur direkter Einfluss auf den Wasserstand des ersten Tanks $z_1(t)$ ausgeübt werden. Mit Hilfe der Differentialgleichungen [54]

$$\begin{aligned} \dot{z}_1(t) &= -\frac{A_{o1}}{A_1}\sqrt{2gz_1(t)} + \frac{k_p}{A_1}u(t), \\ \dot{z}_2(t) &= \frac{A_{o1}}{A_2}\sqrt{2gz_1(t)} - \frac{A_{o2}}{A_2}\sqrt{2gz_2(t)} = v(t), \end{aligned} \tag{5.2}$$

lassen sich die Zustände des Systems beschreiben. Die Konstanten A_1, A_{o1}, A_2, sowie A_{o2} entsprechen jeweils den Querschnittsflächen der beiden Tanks und der Rohre. Die Gravitationskonstante ist durch g und die Pumpkonstante durch k_p gekennzeichnet. Die Werte aller Konstanten wurden ebenfalls aus [54] entnommen.

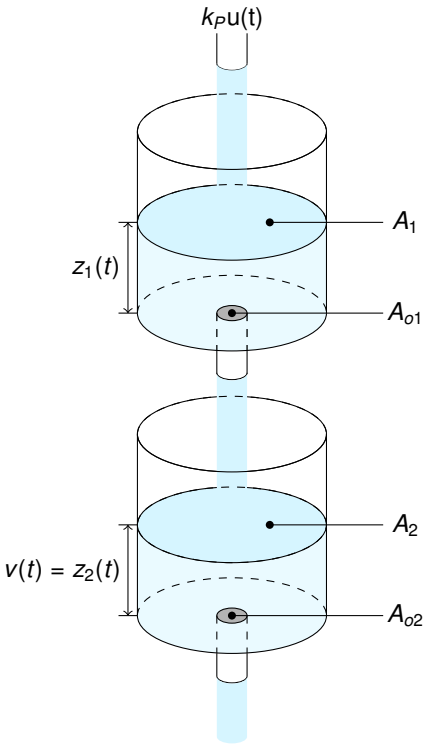


Figure 5.2: Nichtlineares Zweitank-System.

5.3 Genetischer Algorithmus

Algorithmus 8 zeigt den genauen Ablauf des implementierten genetischen Algorithmus: Nach der Generierung einer Startpopulation (Kapitel 4.2.1) durchläuft der Algorithmus eine bestimmte Anzahl an Generationen. In jeder Generation werden zwei Individuen mit Hilfe einer bestimmten Selektionsmethode ausgewählt und gekreuzt. Dadurch entstehen zwei neue Nachkommen, welche anschließend beide mit einer geringen Wahrscheinlichkeit von 15% mutiert werden. Bei der Mutation wird jedes Bit der Verbindungsmatrix mit einer Wahrscheinlichkeit von 5% invertiert. So ist sicher gestellt, dass die Individuen durch die Mutation nicht zu sehr verändert werden. Bevor die neuen Individuen zur Population hinzugefügt werden, wird zusätzlich in jeder Generation das schlechteste Individuum der Population mutiert und ersetzt automatisch das ursprüngliche Individuum, unabhängig davon ob es nun eine noch schlechtere Fitness besitzt.

Die beiden neuen Nachkommen werden nach dem Prinzip *Steady-State Genetic Algorithm* [55] in die Population integriert. Das heißt, dass die Populationsgröße in jeder Generation gleich bleibt. Die zwei schwächsten Individuen werden durch die neuen ersetzt, unter der Bedingung, dass die neuen Nachkommen eine stärkere Fitness besitzen. Zusätzlich wird sichergestellt, dass keine identischen Genotypen in der Population existieren.

Falls Nachkommen entstehen, die bereits vorhanden sind, werden diese nicht in die Population integriert und neue produziert. Die Entfernung der schwächsten Individuen führt auf der einen Seite zu einem hohen Selektionsdruck, allerdings kann dadurch ein schnelleres Konvergieren erzielt werden. Dies kann aufgrund der langen Laufzeit des genetischen Algorithmus vorteilhaft sein.

Daten : Anzahl an Generationen N
Ergebnis : Bestes Individuum $\hat{x}_{\text{opt}}$
Generiere zufällige Startpopulation $\boldsymbol{P}_0 = \{\hat{x}_0, \hat{x}_1, ..., \hat{x}_n\}$ und entferne überflüssige Verbindungen der Netze wie in Abschnitt 4.2.1 erklärt
Transformiere alle Individuen zu Phänotyp mit Algorithmus 7
Trainiere alle Netze mit Algorithmus 5
Berechne Fitnessfunktion $F(\hat{x}_i)$ aller Individuen $\hat{x}_i$
für $k = 1$ **bis** N **tue**
  Wähle zwei Eltern mit Hilfe von Selektionsmethoden
  solange $\{\hat{x}_1, \hat{x}_2, \hat{x}_i\} \cap \boldsymbol{P}_k \neq \emptyset$ **tue**
    Bilde zwei Nachkommen $\hat{x}_1$, $\hat{x}_2$ durch Anwendung von Two-Point-Crossover (Abschnitt 4.2.3)
    Mutiere Nachkommen mit 15% Wahrscheinlichkeit
    Mutiere schlechtestes Individuum $\hat{x}_i$ aus $\boldsymbol{P}_k$
  Ende
  Entferne überflüssige Verbindungen der neuen Individuen
  Trainiere neue Individuen mit Algorithmus 5
  Berechne Fitness der neuen Individuen
  Füge neue Individuen zu $\boldsymbol{P}_k$ hinzu, sortiere alle Individuen nach ihrer Fitness und entferne die schlechtesten zwei Individuen aus der Population
Ende
zurück $\hat{x}_{\text{opt}}$

Algorithmus 8 : Ablauf des implementierten genetischen Algorithmus

Wie bereits erklärt, werden die Netze im genetischen Algorithmus darauf optimiert, die Parameter eines PID-Reglers automatisch anzupassen. Die Evaluierung der Netze erfolgt ebenfalls anhand des Zweitank-Systems. Die Qualität eines Netzwerks wird einerseits anhand des Regelfehlers aus Gleichung 5.1 bewertet und wird im Weiteren mit $R(\hat{x})$ bezeichnet. Andererseits fließt die Anzahl der Verbindungen $C(\hat{x})$ eines Netzes in dessen Fitness mit ein, da dies ein wichtiger Faktor für die Trainingsdauer ist. Die Gesamtfitness eines Individuums wird als gewichtete Summe

$$F(\hat{x}) = \gamma \cdot R(\hat{x}) + (1 - \gamma) \cdot C(\hat{x}) \tag{5.3}$$

definiert. Durch die Konstante γ kann die Gewichtung der beiden Größen eingestellt werden. Die Anzahl der Verbindungen eines Netzes $\hat{x}$ wird bestimmt durch die Summe der 1er in der Verbindungsmatrix eines Individuums.

Sowohl der Fehler, als auch die Anzahl der Verbindungen sollen minimiert werden, es ergibt sich daher ein Minimierungsproblem und das Individuum mit der kleinsten Fitness wird gesucht.

6 Ergebnisse

In den letzten Kapiteln wurden die Parameter eines genetischen Algorithmus erklärt, welche den Verlauf der Optimierung unterschiedlich beeinflussen können. Diese sind die Selektionsmethode (Kapitel 2.2.4), die Art des Crossovers (Kapitel 4.2.3), die Wahrscheinlichkeit einer Mutation (Kapitel 2.2.3), die Fitnessfunktion (Kapitel 2.2.2), die Anzahl der Individuen in der Population, sowie die Anzahl der Generationen. Die Wahrscheinlichkeit für eine Verbindung der zufällig generierten Netze der Startpopulation (Kapitel 4.2.1) wurde zusätzlich für diese Anwendung eingeführt und gilt nicht allgemein für genetische Algorithmen. Ebenso ist die Anzahl der Trainings pro Netz ein problemspezifischer Parameter des genetischen Algorithmus. Die korrekte Wahl der Parameter eines genetischen Algorithmus ist abhängig von der Problemstellung und gilt allgemeien als schwierig [56].

Für die Durchführung der Experimente wurden einige Parameter für alle Experimente gleich gewählt, andere wurden variiert.

Dieses Kapitel beschreibt zunächst die Wahl aller Parameter. Anschließend werden die Ergebnisse gezeigt, wobei auf das Optimierungsverhalten des genetischen Algorithmus, sowie auf den Selektionsdruck und auf die Effizienz der Fitnessfunktion eingegangen wird.

6.1 Feste Parameter

Zu den festen Parameter zählen die Mutationswahrscheinlichkeit, die Anzahl der Generationen, die Anzahl der Individuen, die maximale Anzahl der Neuronen pro Netz, das Crossover, sowie die Anzahl der Trainings pro Netz. Diese sollen nun im Einzelnen erklärt werden. Die Durchführung der Mutation wurde bereits in Kapitel 5.3 beschrieben.

Die Anzahl der Generationen muss ausreichend groß sein, damit der Algorithmus zu einer optimalen Lösung konvergieren kann. Demgegenüber spiegelt sich die Anzahl der Generationen linear in der Laufzeit wieder und es muss ein Kompromiss gemacht werden. Die Experimente wurden für 200 Generationen ausgeführt, was zu einer durchschnittlichen Laufzeit von 10 Tagen pro Experiment führte.

Die Individuen innerhalb der Population geben dem Algorithmus eine Auswahl an Paarungskandidaten für die Produktion neuer Individuen. Je größer die Anzahl der Individuen ist, desto mehr unterschiedliche Kreuzungsergebnisse können in jeder Generation produziert werden. Bei einer zu kleinen Populationsgröße würde der Algorithmus ähnliche Nachkommen produzieren und schnell zu einem lokalen Optimum konvergieren. Um die Laufzeit für das Training der Startpopulation gering zu halten und dennoch eine gute Auswahl an Individuen zu bekommen, wurde die Anzahl der Individuen auf 30 gesetzt. Die

Trainingslaufzeit der Startpopulation entsprach somit der selben Laufzeit, die das Berechnen von 10 Generationen benötigt.

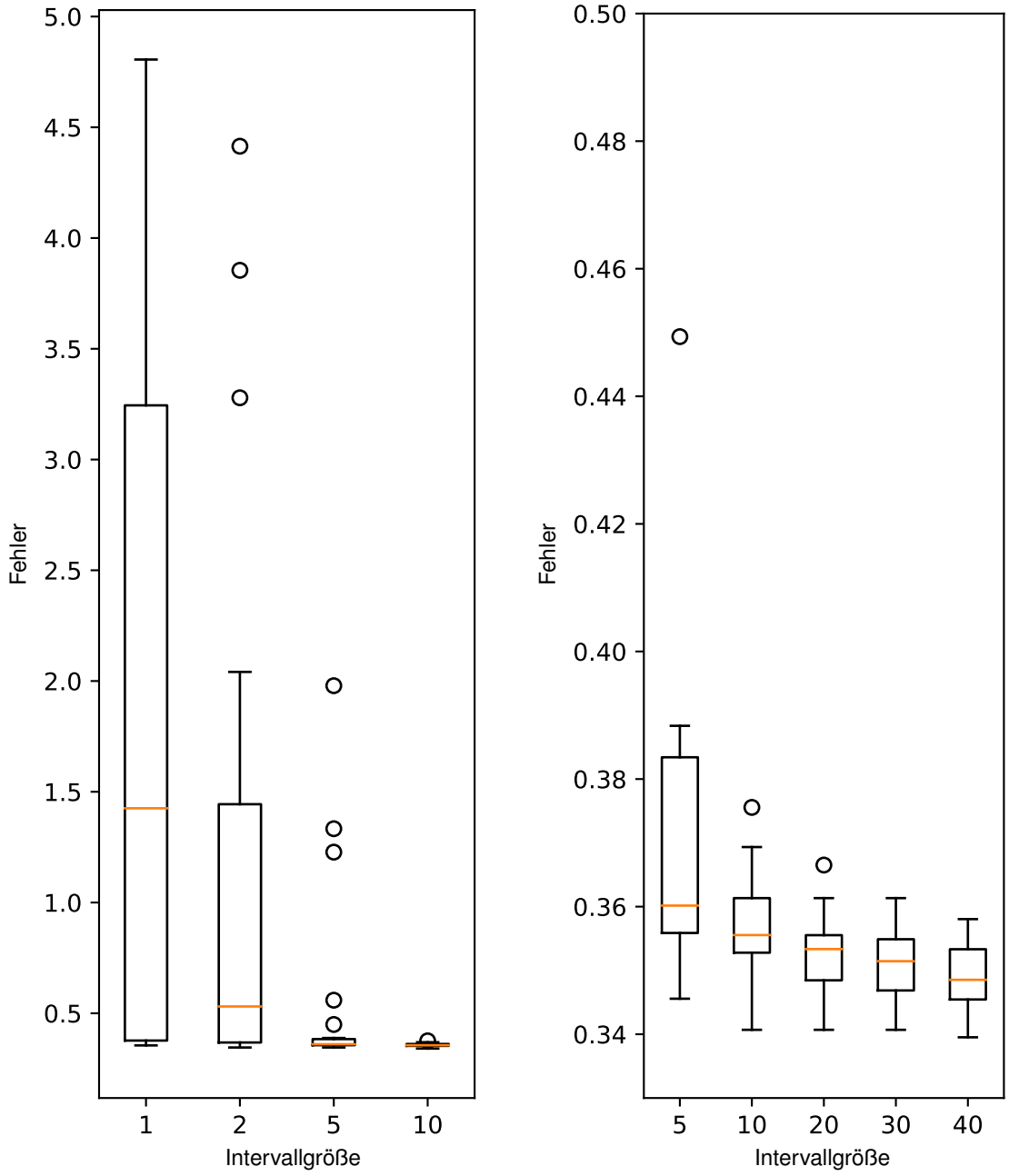
Die Wahl der Anzahl der Neuronen basiert auf den Ergebnissen des festen Netzes aus Abbildung 5.1, welches 9 Neuronen besitzt. Da der genetische Algorithmus auch Netze produzieren kann, die weniger Neuronen enthalten als durch die Größe der Matrix vorgegeben, wurde hierbei eine größere Anzahl von 15 Neuronen verwendet.

Aus den drei Crossover-Möglichkeiten aus Kapitel 4.2.3 wurde das Two-Point-Crossover verwendet. Während das One-Point-Crossover ein ganzes Endstück eines Chromosoms austauscht, verändert das Uniform-Crossover die Struktur der Netze sehr stark. Beim Two-Point-Crossover spielt die Struktur der Elternnetze noch eine Rolle und bietet dennoch mehr Variation als das One-Point-Crossover.

Wie bereits in Kapitel 4.2.4 erläutert, ist das Trainingsergebnis eines Netzes abhängig von den Startgewichten und das Netz muss mehrmals für unterschiedliche Startgewichte trainiert werden. Aus allen Trainingsdurchläufen wird das beste Ergebnis für die Berechnung der Fitness verwendet. Allerdings ist nie sicher gestellt, dass dieses Ergebnis auch das Optimale ist und die Fitness die tatsächliche Leistung des Netzes widerspiegelt. Dadurch kann das Netz eine schlechtere Stellung im genetischen Algorithmus erhalten, als es tatsächlich besitzt und wird vom Algorithmus aussortiert. Weiterhin sind die Trainings die Hauptursache für die lange Laufzeit des implementierten genetischen Algorithmus. Es muss daher abgewägt werden, wie viele Trainings notwendig sind, um das Rauschen in der Fitness gering zu halten und die Laufzeit dabei dennoch vertretbar zu machen.

Um eine angemessene Trainingsanzahl zu ermitteln, wurde ein Experiment durchgeführt. Es sollte herausgefunden werden, wie konsistent der minimale Fehler ist, wenn ein Netz n Mal trainiert wird. Um hierfür eine Statistik zu erstellen, sollte das Netz mehrmals n Mal trainiert werden. Das Experiment wurde für verschiedene n durchgeführt. Ziel war es, das kleinste n zu ermitteln, welches ein akzeptables Ergebnis lieferte und die Fitness eines Netzes mit wenig Rauschen behaftet war.

Hierfür wurde das Netz mit fester Struktur aus Abbildung 5.1 1000 Mal für jeweils unterschiedliche, zufällige Startgewichte trainiert und der jeweilige Fehler gespeichert. Um die Statistik für eine bestimmte Anzahl an Trainings n zu erstellen, wurden die Fehler in Intervalle der Größe n aufgeteilt und aus jedem Intervall der minimale Fehler extrahiert. Es wurden die Trainingsanzahlen 1, 2, 5, 10, 20, 30 und 40 ausgewertet, wobei immer dieselbe Anzahl an Intervallen verwendet wurde. Abbildung 6.1a zeigt die Verteilung des minimalen Fehlers für die Intervallgrößen 1, 2, 5 und 10 anhand von Boxplotdiagrammen. Die Boxplots lassen erkennen, dass der Bereich des minimalen Fehlers ab 5 Trainingsdurchläufen deutlich reduziert ist. Für $n = 10$ enthalten die Daten weniger Ausreißer. In Abbildung 6.1b sind die Boxplots für $n = 5$, 10, 20, 30 und 40 dargestellt. Von 5 auf 10 Trainingsdurchläufe kann der Bereich des minimalen Fehlers nochmal eingeschränkt werden, für größere n ist die Verbesserung nur noch gering. Eine Trainingsanzahl von 10 ist daher ein guter Kompromiss zwischen Laufzeit und Betrag des Rauschens in der Fitnessfunktion.



(a) Verteilung des minimalen Fehlers für die Trainingsanzahlen $n = 1, 2, 5$ und 10

(b) Verteilung des minimalen Fehlers für die Trainingsanzahlen $n = 5, 10, 20, 30$ und 40

Figure 6.1: Verteilung des minimalen Fehlers für bestimmte Anzahlen an Trainings des neuronalen Netzes aus 5.1. Die Boxplots zeigen, wie konsistent der minimale Fehler ist, wenn das Netz für eine Anzahl von $n = 1, 2, 5, 10, 20, 30$ und 40 trainiert wird.

	Versuch 1	Versuch 2	Versuch 3
Anzahl der Generationen	200	200	200
Anzahl der Individuen	30	30	30
Anzahl der Neuronen pro Netz	15	15	15
Trainingsepochen	10	10	10
WSK für Verbindung	30%	30%	30%
WSK für Mutation	15%	15%	15%
Selektionsmethode	Turnier	SUS + Rang	SUS + Rang
γ	.9998	.9998	.999

Table 6.1: Verwendete Parameter im genetischen Algorithmus. WSK steht hierbei für die Wahrscheinlichkeit.

6.2 Variierte Parameter

Bei den Versuchen wurden zwei verschiedene Selektionsmethoden angewendet. Die Autoren von [57] verwenden einen genetischen Algorithmus mit Turnierselektion, um die Architektur von Feedforward-Netzen zu optimieren. Da diese Methode erfolgreich eingesetzt wurde, wurde hier ebenfalls die Turnierselektion herangezogen. Zusätzlich wurde eine weitere Selektionsmethode, die Rangselektion (Kapitel 2.2.4) zusammen mit stochastischem universellen Sampling getestet.

Ein weiterer Parameter der variiert wurde, ist der Gewichtungsfaktor γ der Fitnessfunktion (Gleichung 5.3). Die Aufstellung der Fitness eines Individuums wurde bereits in Kapitel 5.3 erläutert. Diese ist definiert durch den Regelfehler der Strecke, sowie durch die Anzahl der Verbindungen im Netz.

Der Fehler eines Netzes sollte ausschlaggebend für den Wert seiner Fitness sein, während die Anzahl der Verbindungen eine geringere Rolle spielen sollte. Der Fehler wird daher stärker gewichtet als die Anzahl der Verbindungen. Zusätzlich muss bei der Gewichtung die unterschiedliche Größenordnung der beiden Parameter berücksichtigt werden. So ist die Anzahl der Verbindungen um den Faktor 1000 größer als der Fehler. Es wurden zwei verschiedene Werte für γ getestet. In Tabelle 6.1 sind alle verwendeten Werte der Parameter zusammengefasst.

6.3 Auswertung

Im letzten Abschnitt wurden die verwendeten Parameter des genetischen Algorithmus aufgeführt und erläutert. Unter Verwendung dieser Parameter soll das Verfahren nun evaluiert werden. Hierfür wird zunächst der Verlauf der Fitnessfunktion aus Gleichung 5.3 analysiert, sowie deren Summanden im Einzelnen. Anschließend wird der Selektionsdruck

anhand der Matrixstrukturen der Netze untersucht, sowie die Qualität der entstandenen Netze durch Anwendung auf den PID-Regler.

6.3.1 Verlauf des genetischen Algorithmus

Ziel der Anwendung des implementierten genetischen Algorithmus ist es, die Fitness der Individuen über die Generationen hinweg zu minimieren. Die Abbildungen 6.2 (Versuch 1), 6.3 (Versuch 2) und 6.4 (Versuch 3) zeigen den Verlauf der Fitness, des Fehlers, sowie der Anzahl der Verbindungen in Abhängigkeit der Generationen. Dabei werden sowohl die durchschnittlichen Werte einer ganzen Population, als auch die Werte des besten Individuums der jeweiligen Generation dargestellt.

Versuch 1: Der erste Versuch konnte keine Verbesserung des besten Individuums erzielen. Sowohl der Fehler, als auch die Anzahl der Verbindungen bleiben für alle Generationen konstant und das beste Individuum entsteht bereits in der zufälligen Startpopulation. Zudem ist deutlich zu sehen, dass die Fitnessfunktion allein vom Fehler abhängt, was der durchschnittliche Verlauf der beiden Größen zeigt. Der Mittelwert der Fitness konvergiert bereits nach 10 Generationen. Die durchschnittliche Anzahl der Verbindungen fällt bis unter 80, während der Fehler konstant bleibt.

Versuch 2: Versuch 2 unterscheidet sich allein durch die Selektionsmethode von Versuch 1. Es kann ein minimal späteres Konvergieren beobachtet werden (etwa ab Generation 14), als bei Versuch 1. Das beste Individuum der letzten Generation zeigt eine minimal kleinere Fitness, als das der Startpopulation. Die Anzahl der Verbindungen hingegen konnte sowohl für den Mittelwert als auch für das beste Individuum nicht bedeutend minimiert werden und ist überwiegend konstant geblieben. Auch hier wird die Fitness stark durch den Fehler dominiert.

Versuch 3: Für Versuch 3 wurde der Gewichtungsfaktor der Fitnessfunktion verändert, um die Anzahl der Verbindungen in der Optimierung zu verstärken. Als Selektionsverfahren wurde erneut die Rangselektion zusammen mit stochastischem universellen Sampling verwendet, da diese Methode in Versuch 2 ein minimal langsameres Konvergieren erzielte. Der Verlauf der Fitness, sowie des Fehlers zeigt erneut lediglich eine geringe Minimierung. Die Anzahl der Verbindungen hingegen konnte in diesem Versuch um 20% reduziert werden.

Tabelle 6.2 zeigt die Fitness, den Fehler und die Anzahl der Verbindungen des besten Individuums jeweils für die Start- und die Endgeneration.

6.3.2 Selektionsdruck

Um den Selektionsdruck des Verfahrens besser zu untersuchen, wurden die Matrizen der Netze auf ihre Ähnlichkeit zueinander überprüft. Ein hoher Selektionsdruck kennzeichnet sich dadurch, dass sich das stärkste Individuum der Population schnell durchsetzt und der Algorithmus zu diesem Optimum konvergiert.

Die Struktur der Matrizen wurde mit Hilfe einer Farbtabelle analysiert, welche die Häufigkeit jedes Matrixelements innerhalb einer Population darstellt. Abbildung 6.5 zeigt dies für

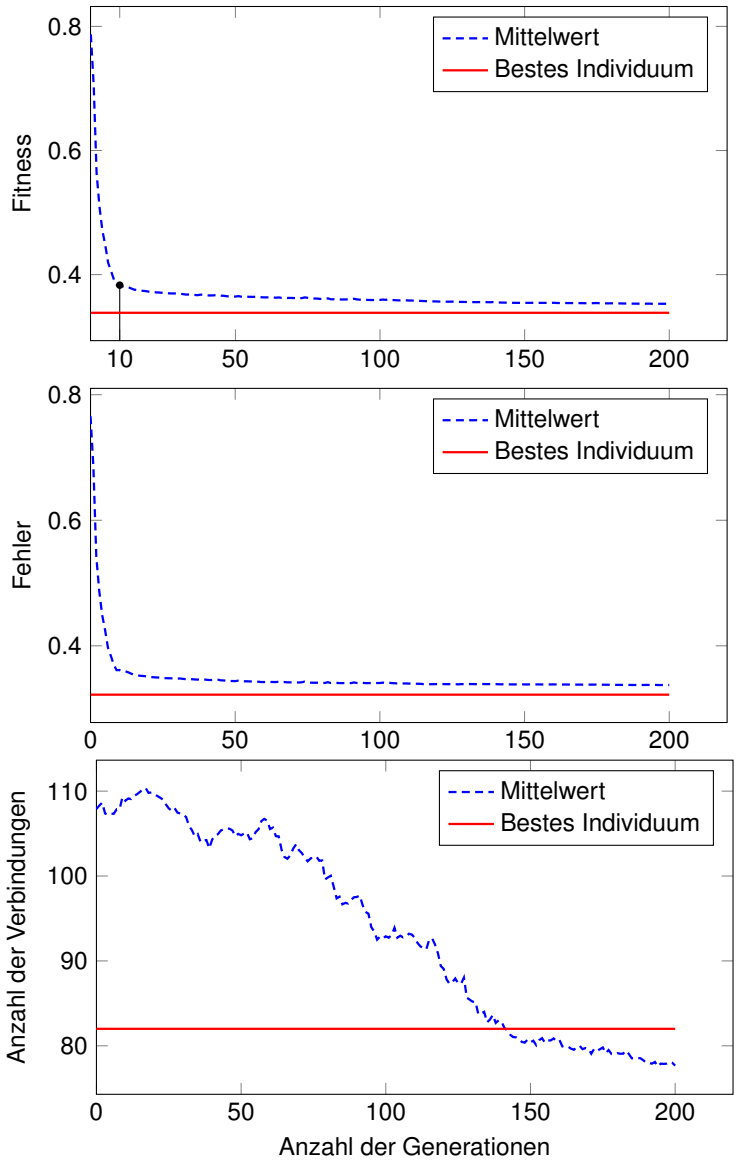


Figure 6.2: Evolution eines dynamischen neuronalen Netzes für 200 Generationen für Versuch 1

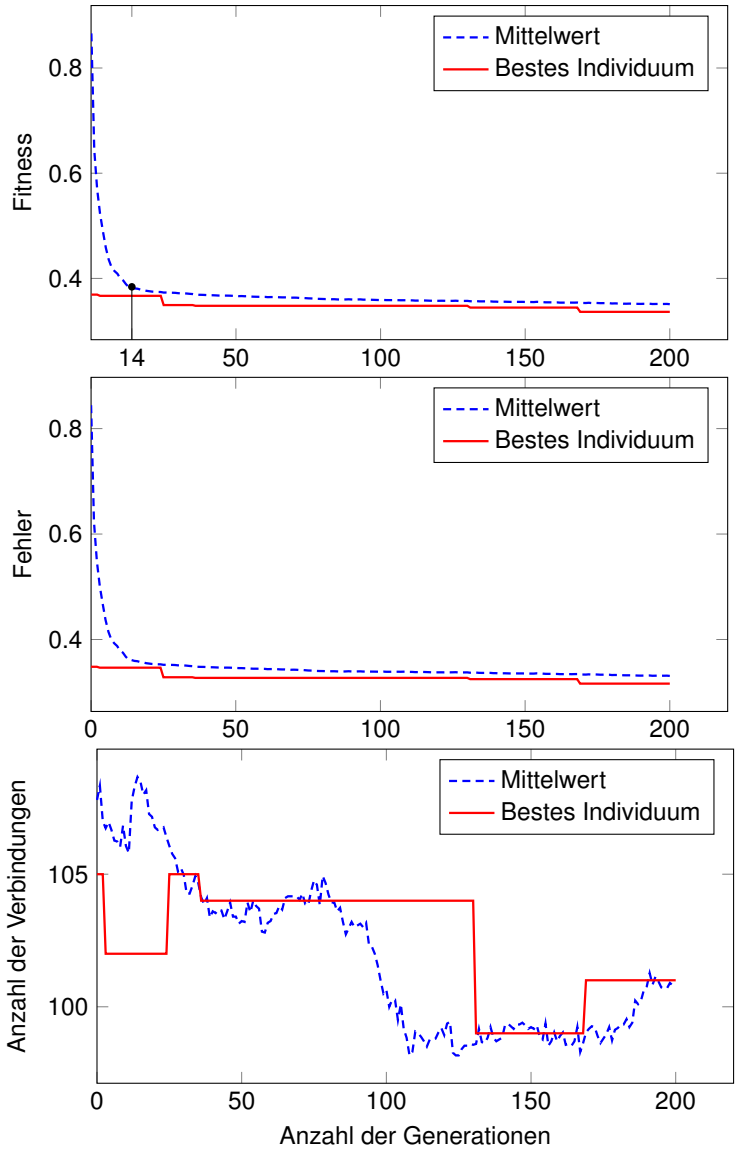


Figure 6.3: Evolution eines dynamischen neuronalen Netzes für 200 Generationen für Versuch 2

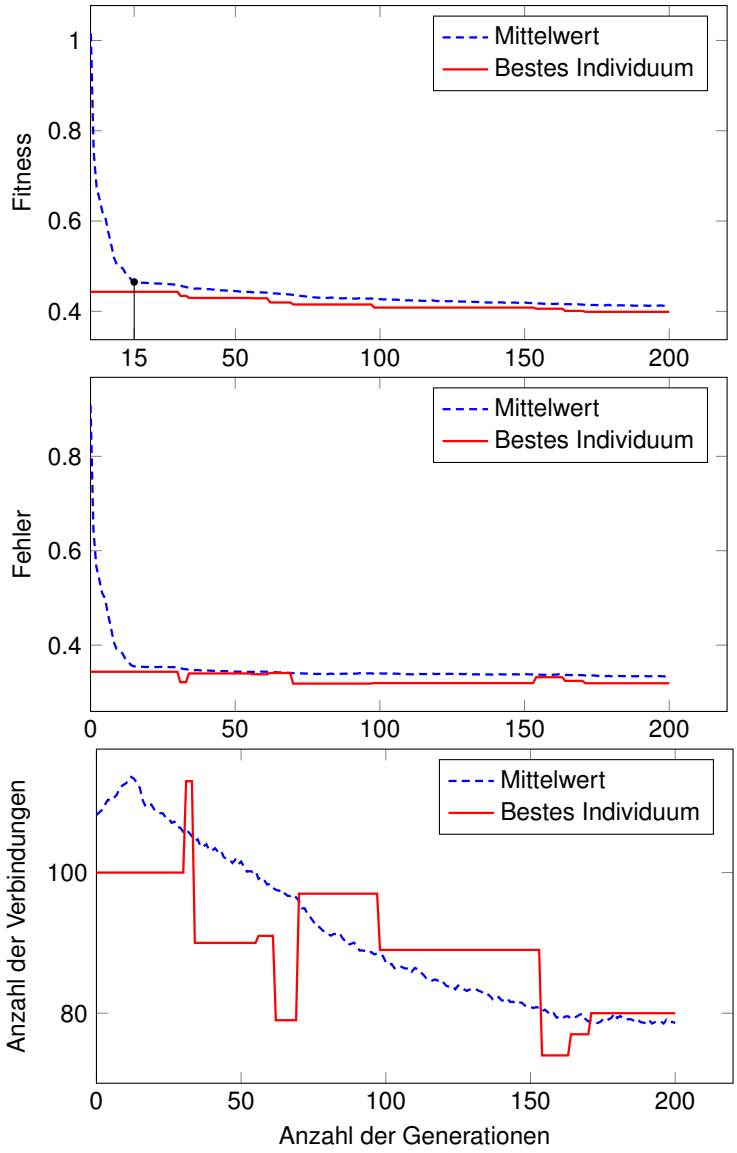


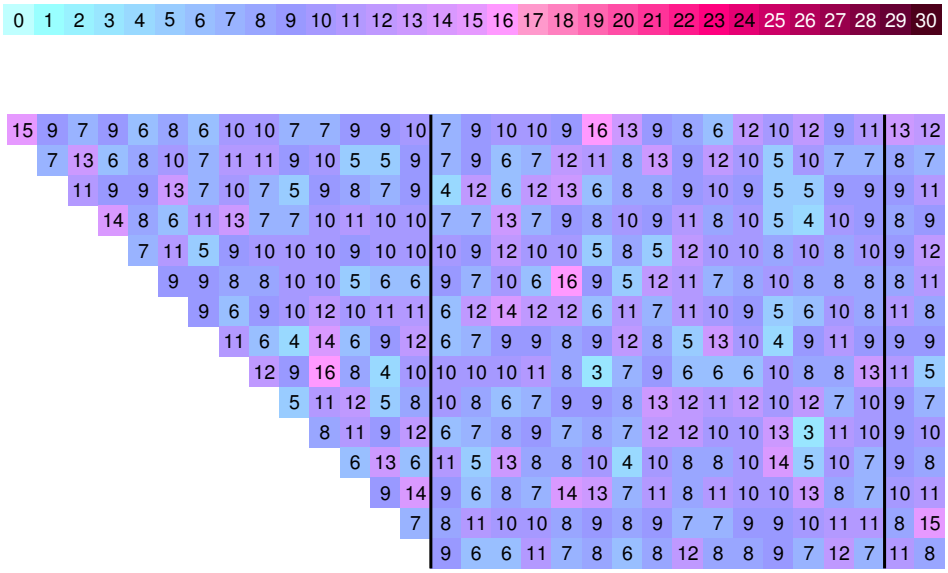
Figure 6.4: Evolution eines dynamischen neuronalen Netzes für 200 Generationen für Versuch 3

Dataset	Versuch 1			Versuch 2			Versuch 3		
	Start	Ende	Δ	Start	Ende	Δ	Start	Ende	Δ
Fitness	0.338	0.338	0%	0.369	0.336	8.9%	0.443	0.399	9.9%
Fehler	0.322	0.322	0%	0.348	0.316	9.2%	0.344	0.319	7.3%
Verbindungen	82	82	0%	105	101	3.8%	100	80	20%

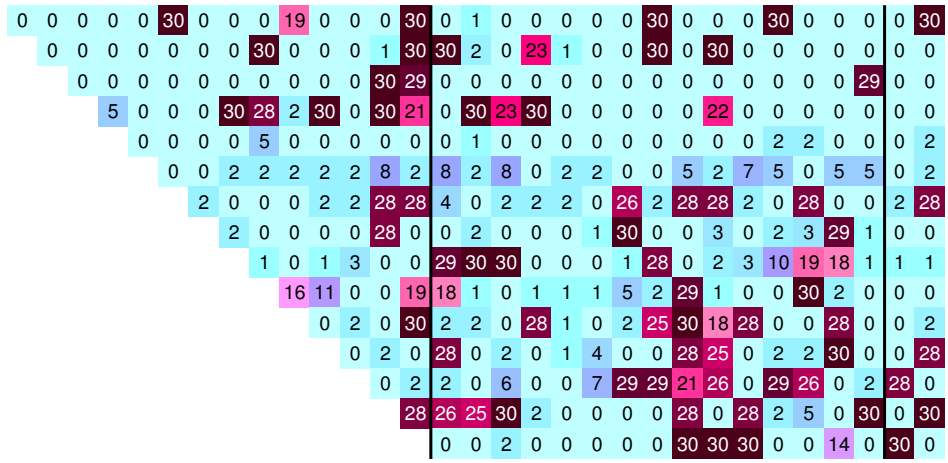
Table 6.2: Fitness, Fehler und Anzahl der Verbindungen der besten Individuen (Individuum mit der kleinsten Fitness) der Startpopulation, sowie der letzten Generation (200). Δ steht hierbei für die relative Verbesserung.

Versuch 1. Dabei wird die Farbtabelle für die Matrizen der Startpopulation, sowie für die der letzten Generation dargestellt. Während die Startpopulation eine gleichmäßige Verteilung der Elemente aufweist, sind die Matrizen der Endgeneration deutlich zu einer Struktur konvergiert. Abbildung 6.6 visualisiert schließlich den Verlauf aller Versuche beginnend mit der Startpopulation über Generation 50, 100 und 150 bis zur letzten Generation. Wie die letzte Auswertung der Versuche bereits gezeigt hat, dominiert beim ersten Versuch die beste Struktur sehr schnell die Population, was an der Farbverteilung der Matrizen in 6.6a erkennbar ist. Die letzte Generation enthält schließlich größtenteils die Farben des größten und kleinsten Werts in der Farbskala, was auf einen hohen Selektionsdruck schließen lässt. Im Vergleich dazu ist bei Versuch 2 ein etwas langsameres Konvergieren der Struktur erkennbar. Die Matrizen der letzten Generation enthalten hier etwas mehr Vielfalt. Im letzten Versuch zeigen die Farbtabellen wiederum ein früheres Konvergieren als bei Versuch 1. Bereits in Generation 50 sind Verbindungen zu erkennen, die bis zur letzten Generation erhalten bleiben.

Abbildung 6.7 stellt die Struktur der Matrizen der jeweils besten Individuen der Startpopulation dar. Vergleicht man diese jeweils mit den Farbtabellen der letzten Generation aus Abbildung 6.6, so lässt sich für Versuch 1 eine hohe Ähnlichkeit zwischen den Strukturen erkennen. Somit dominiert bereits das beste Individuum der zufällig generierten Startpopulation die Population, was der Verlauf der Fitnessfunktion des besten Individuums (siehe Abbildung 6.2) bereits gezeigt hat. Für Versuch 2 kann dies nicht beobachtet werden. Für Versuch 3 ist eine Ähnlichkeit der Strukturen erkennbar, jedoch ist diese weniger stark ausgeprägt als bei Versuch 1.

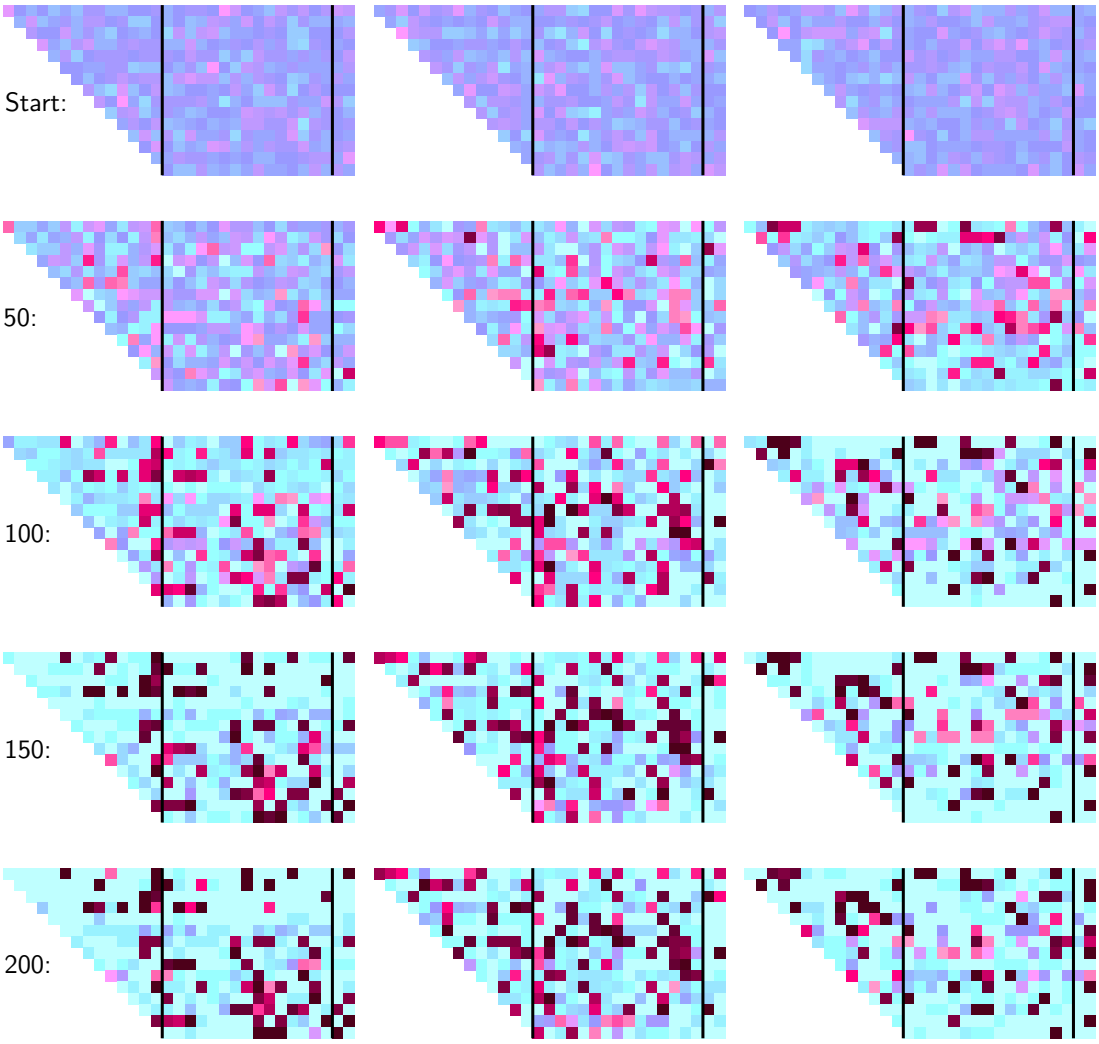


(a) Matrizenstruktur der Individuen der zufälligen Startpopulation



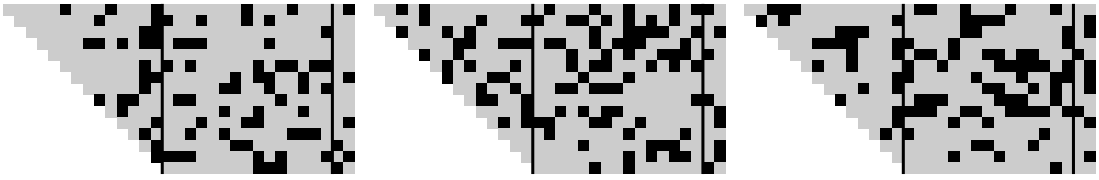
(b) Matrizenstruktur der Individuen der letzten Generation (200)

Figure 6.5: Farbtabelle für die Darstellung der Häufigkeit aller Matrixelemente innerhalb einer Population (Versuch 1)



(a) Versuch 1 **(b)** Versuch 2 **(c)** Versuch 3

Figure 6.6: Verlauf der Matrizenstruktur über mehrere Generationen für die Versuche aus 6.3.1. Die zugehörige Farbskala findet sich in Abbildung 6.5.



(a) Versuch 1 **(b)** Versuch 2 **(c)** Versuch 3

Figure 6.7: Matrizen der besten Individuen der Startpopulationen. Die schwarzen Kästchen kennzeichnen eine 1 in der Verbindungsmatrix.

6.3.3 Evaluierung des besten Individuums

Nachdem gezeigt werden konnte, dass der Algorithmus die Fitness der Individuen minimal verbessert, sollen nun die entstandenen Netze anhand des Regelungstechnikproblems ausgewertet werden. Hierfür wird jeweils das beste Individuum der Populationen zum Einstellen des PID-Reglers verwendet. Es wird nur Versuch 2 behandelt. In Abbildung 6.8 sind die Zeitverläufe des Systemzustands für das beste Netz der Startpopulation, sowie für die besten Netze der Generationen 14, 30, 150 und 200 zu sehen. Zwischen diesen Generationen konnte jeweils eine Verbesserung der Fitness beobachtet werden. Sie sollen daher verglichen werden.

In Generation 14 kann eine minimale Verbesserung des Regelverhaltens beobachtet werden. Das Überschwingen des Zustandssignals beim Ändern der Sollwertkurve konnte im Vergleich zur Startpopulation reduziert werden. In Generation 30 beginnt der Systemzustand zu schwingen, was allgemein nicht erwünscht ist. Für die nächste Generation wird dieses Schwingen zwar reduziert, in der letzten Generation wiederum deutlich verstärkt. Daraus kann die Schlussfolgerung gezogen werden, dass der RMSE keine optimale Metrik zur Bewertung der Netze im genetischen Algorithmus darstellt, da die Schwingungen von der Fehlerfunktion nicht erfasst werden können. Für Versuch 3 konnte dieselbe Beobachtung gemacht werden, dieser wird daher hier nicht gezeigt.

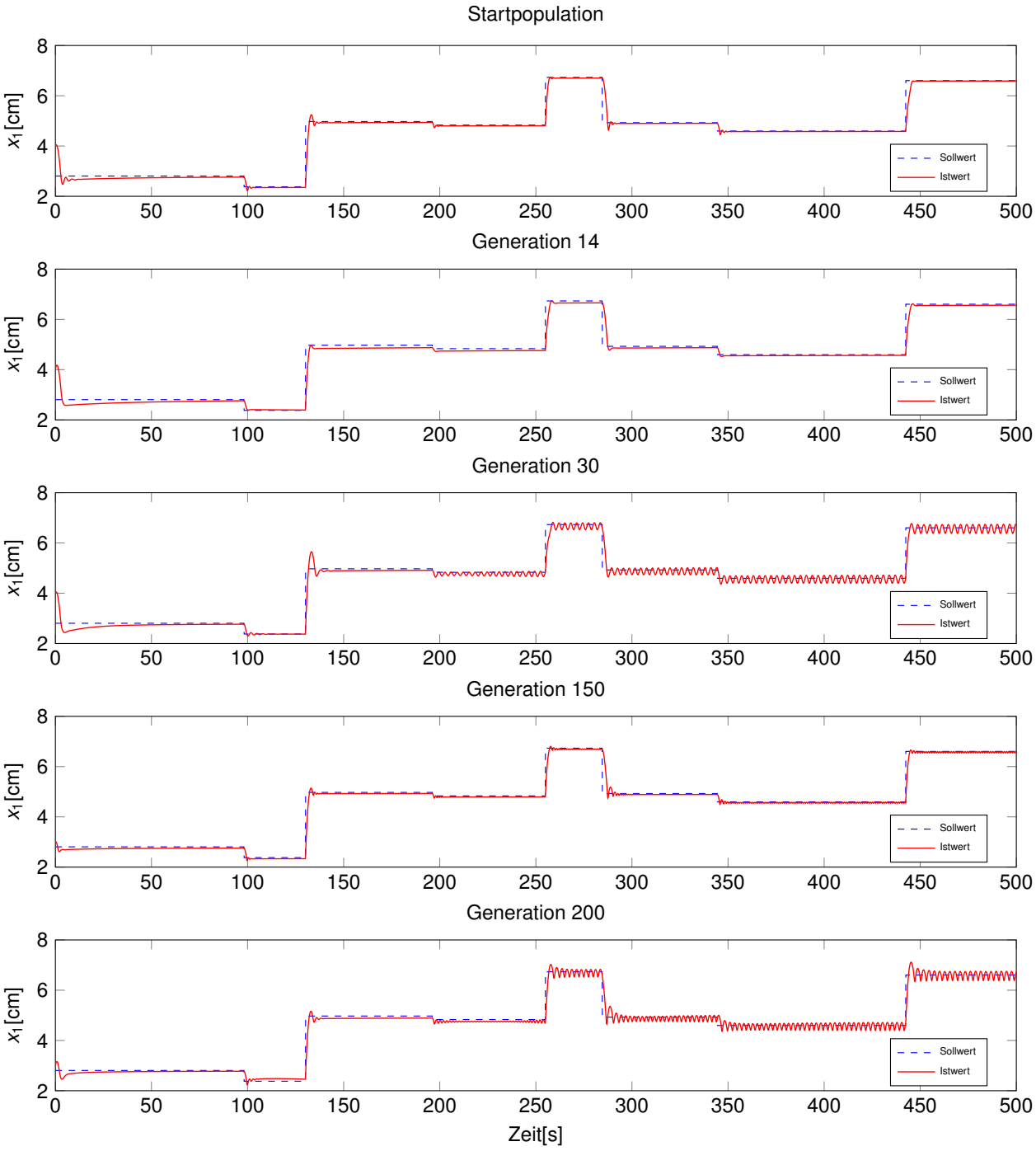


Figure 6.8: Zeitverlauf des Systemzustands des Zweitank-Systems unter Verwendung der besten Netze verschiedener Generationen zum Einstellen des PID-Reglers.

7 Zusammenfassung und Ausblick

In den letzten Kapiteln wurde die Implementierung eines Systems beschrieben, welches aus einem Regelkreis, einem in diesen eingebetteten dynamischen neuronalen Netz und einem genetischen Algorithmus, besteht. Das neuronale Netz diente dabei der Einstellung der Parameter eines zeitabhängigen PID-Reglers. Ziel war es, mit Hilfe des genetischen Algorithmus die Architektur des neuronalen Netzes für diese Problemstellung zu optimieren. Der erzielte Nutzen des genetischen Algorithmus war dabei, ohne vorherige Kenntnis über die Netzstruktur ein neuronales Netz zu erhalten, welches für ein spezifisches Problem ein optimales Ergebnis liefert.

Die Bewertung der Netze im genetischen Algorithmus erfolgte durch eine Fitnessfunktion, welche als gewichtete Summe aus dem Regelfehler und der Anzahl der Verbindungen im Netz realisiert wurde. Die Implementierung wurde für unterschiedliche Gewichtungen der Fitnessfunktion getestet. Zusätzlich wurden zwei verschiedene Selektionsverfahren verwendet: eine Turnierselektion proportional zur Fitness und eine Rangselektion.

Die Auswertung erfolgte anhand des Verlaufs der Fitnessfunktion über eine Zeit von 200 Generationen. Bei allen Versuchen konnte keine nennenswerte Verbesserung der Fitness erzielt werden. Der Verlauf der durchschnittlichen Fitness der Populationen zeigte ein sehr rasches Konvergieren innerhalb der ersten 10 bis 15 Generationen, was auf einen hohen Selektionsdruck hindeutet.

Um den Selektionsdruck im genetischen Algorithmus zu untersuchen, wurde die Struktur der Matrizen durch Farbkarten dargestellt. Diese veranschaulichen die Häufigkeit eines Matrixelements innerhalb einer Population. Es konnte gezeigt werden, dass die Netze deutlich zu einer bestimmten Struktur konvergierten, was den hohen Selektionsdruck bestätigte. Bei der Turnierselektion zeigte sich dies deutlich stärker als bei der Rangselektion.

Der hohe Selektionsdruck beschleunigte einerseits das Konvergieren des Algorithmus, was die Wahrscheinlichkeit, ein globales Optimum zu erhalten allgemein reduziert. Andererseits war die Simulation des Systems aus Netz, Regler und Strecke sehr rechenaufwendig und führte zu langen Laufzeiten des gesamten Verfahrens. Ein niedriger Selektionsdruck hätte die Laufzeit nochmals vergrößert.

Die hohe Laufzeit des Algorithmus stellte ein Problem bei der Auswertung dar. Daher wurden bei der Implementierung mehrere geschwindigkeitssteigernde Ansätze berücksichtigt. Durch die Parallelisierung von mehreren Prozessen und Verwendung eines just-in-time-Compilers konnte das in Python geschriebene Programm gegenüber der ursprünglichen Implementierung etwa um das 34-fache beschleunigt werden. Ein Test von 200 Generationen mit einer Populationsgröße von 30 Netzen benötigte dennoch rund 10 Tage an Laufzeit.

Für einen produktiven Einsatz dieses Verfahrens wären noch zusätzliche Optimierungen nötig.

Einen weiteren verbesserungswürdigen Punkt stellt die Fehlerfunktion des neuronalen Netzes dar, welche in die Fitnessfunktion mit einfließt. Die Auswertung der Netze anhand des Regelungstechnikproblems zeigte, dass Netze mit einem geringeren berechneten Fehler teilweise zu Schwingungen im System führten. In der Praxis wäre daher eine andere Metrik wünschenswert.

List of Figures

List of Tables

Liste der Algorithmen

Literaturverzeichnis

[1] S. Bothe, T. Gärtner, and S. Wrobel. *On-line handwriting recognition with parallelized machine learning algorithms*, pages 82–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[2] C. Bahlmann, B. Haasdonk, and H. Burkhardt. Online handwriting recognition with support vector machines - a kernel approach. In *Proceedings Eighth International Workshop on Frontiers in Handwriting Recognition*, pages 49–54. IEEE, 2002.

[3] A. L. C. Bazzan and F. Klugl. *Introduction to Intelligent Systems in Traffic and Transportation*, volume 25 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, San Rafael, California, USA, 2014.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. DeepDriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 2722–2730. IEEE Computer Society, 2015.

[5] D. Stavens and S. Thrun. A self-supervised terrain roughness estimator for off-road autonomous driving. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*, UAI'06, pages 469–476. AUAI Press, 2006.

[6] A. El Sallab, M. Abdou, E. Perot, and S. Yogaman. Deep reinforcement learning framework for autonomous driving. *Computing Research Repository*, abs/1704.02532(7):70–76, 2017.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 2016.

[8] Z. Wojna, A. N. Gorban, D. Lee, K. Murphy, Q. Yu, Y. Li, and J. Ibarz. Attention-based extraction of structured information from Street View imagery. *Computing Research Repository*, abs/1704.03549, 2017.

[9] E. Vonk, L. C. Jain, and R. P. Johnson. *Automatic generation of neural network architecture using evolutionary computation*, volume 14 of *Advances in fuzzy systems*. World Scientific, Singapore, 1997.

[10] X. Yao and Y. Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90, 1998.

[11] Y. Liu and X. Yao. Evolutionary design of artificial neural networks with different nodes. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 670–675. IEEE, 1996.

[12] L. Marti. Genetically generated neural networks. II. searching for an optimal representation. In *Proceedings of the 1992 International Joint Conference on Neural Networks*, volume 2, pages 221–226. IEEE, 1992.

[13] D. White and P. Ligomenides. *GANNet: A genetic algorithm for optimizing topology and weights in neural network design*, pages 322–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[14] F. H. F. Leung, H. K. Lam, S. H. Ling, and P. K. S. Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *Neural Networks, IEEE Transactions on*, 14(1):79–88, 2003.

[15] J. Jung and J. A. Reggia. *The automated design of artificial neural networks using evolutionary Computation*, pages 19–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] J. Fekiač, I. Zelinka, and J. C. Burguillo. A review of methods for encoding neural network topologies in evolutionary computation. In *Proceedings of the 25th European Conference on Modeling and Simulation ECMS 2011*, pages 410–416, 2011.

[17] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann Publishers Inc., 1989.

[18] D. Dasgupta and D. R. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96. IEEE, 1992.

[19] M. Mandischer. *Representation and evolution of neural networks*, pages 643–649. Springer Vienna, Vienna, 1993.

[20] W. Schiffmann. Encoding feedforward networks for topology optimization by simulated evolution. In *Proceedings of the fourth International Conference on Knowledge-Based Intelligent Engineering Systems and Allied Technologies, 2000*, volume 1, pages 361–364. IEEE, 2000.

[21] C. Jacob and J. Rehder. *Evolution of neural net architectures by a hierarchical grammar-based genetic system*, pages 72–79. Springer Vienna, Vienna, 1993.

[22] J. Qiao, X. Huang, and H. Han. *Recurrent neural network-based control for wastewater treatment process*, pages 496–506. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[23] T. Varshney and S. Sheel. A new online tuning approach for PID control of multivariable systems using diagonal recurrent neural network. In *2011 IEEE International Conference on Control System, Computing and Engineering*, pages 317–320. IEEE, 2011.

[24] A. Zribi, M. Chtourou, and M. Djemel. A new PID neural network controller design for nonlinear processes. *Computing Research Repository*, abs/1512.07529, 2015.

[25] R. Sharma, V. Kumar, P. Gaur, and A. P. Mittal. An adaptive PID like controller using mix locally recurrent neural network for robotic manipulator with variable payload. *ISA Transactions*, 62:258–267, 2016.

[26] F. A. C. Azevedo, L. R. B Carvalho, L. T. Grinberg, J. M. Farfel, R. E. L. Ferretti, R. E. P. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541, 2009.

[27] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31–44, 1996.

[28] T. P. Trappenberg. *Fundamentals of computational neuroscience*. Oxford University Press, 2nd edition, 2010.

[29] K. Mehrotra, C. K. Mohan, and S. Ranka. *Elements of artificial neural networks*. A Bradford Book, Cambridge, 4th edition, 1997.

[30] I. A. Basheer and M. Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods*, 43(1):3–31, 2000.

[31] J. Moody and N. Yarvin. Networks with learned unit response functions. In *Advances in Neural Information Processing Systems 4*, pages 1048–1055. Morgan Kaufmann Publishers Inc., 1992.

[32] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[33] H. Yan, Y. Jiang, J. Zheng, C. Peng, and Q. Li. A multilayer perceptron-based medical decision support system for heart disease diagnosis. *Expert Systems with Applications*, 30(2):272–281, 2006.

[34] A. Khotanzad and C. Chung. Application of multi-layer perceptron neural networks to vision problems. *Neural Computing and Applications*, 7(3):249–259, 1998.

[35] M. T. Hagan and M. B. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[36] Y. M. Najjar, I. A. Basheer, and M. N. Hajmeer. Computational neural networks for predictive microbiology: I. methodology. *International Journal of Food Microbiology*, 34(1):27–49, 1997.

[37] O. De Jesus and M. T. Hagan. Backpropagation algorithms for a broad class of dynamic networks. *Neural Networks, IEEE Transactions on*, 18(1):14–27, 2007.

[38] Y.-M. Chiang, M.-J. Tsai, F.-J. Chang, L.-C. Chang, and Y.-F. Wang. Dynamic neural networks for real-time water level predictions of sewerage systems-covering gauged and ungauged sites. *Hydrology and Earth System Sciences*, 14(7):1309–1319, 2010.

[39] D. Kriesel. *A brief introduction to neural networks*. `http://www.dkriesel.com`, 2007. Online; Stand Juli 2017.

[40] J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[41] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *Computing Research Repository*, abs/1506.00019, 2015.

[42] M. Jordan. Serial order: A parallel distributed processing approach. Technical Report ICS Report 8604, Institute for Cognitive Science, University of California, San Diego, 1986.

[43] J. H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[44] K. Jebari and M. Madiafi. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3(4):333–344, 2013.

[45] M. J. Er and F. Liu. Genetic algorithms for MLP neural network parameters optimization. In *2009 Chinese Control and Decision Conference*, pages 3653–3658. IEEE, 2009.

[46] O. Kramer. *Genetic algorithm essentials*, volume 679 of *Studies in Computational Intelligence*. Springer International Publishing, Cham, Germany, 2017.

[47] A. Konak, D. W. Coit, and A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91(9):992–1007, 2006.

[48] N. M. Razali and J. Geraghty. Genetic algorithm performance with different selection strategies in solving tsp. *Lecture Notes in Engineering and Computer Science*, 2191(1):1134–1139, 2011.

[49] J. Jung, V. Q. Leu, T. D. Do, E. Kim, and H. H. Choi. Adaptive PID speed control design for permanent magnet synchronous motor drives. *Power Electronics, IEEE Transactions on*, 30(2):900–908, 2015.

[50] S. Omatu and M. Yoshioka. Self-tuning neuro-PID control and applications. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 3, pages 1985–1989. IEEE, 1997.

[51] B. M. Wilamowski and J. D. Irwin. *Intelligent Systems*. CRC Press, Inc., Boca Raton, Florida, USA, 2nd edition, 2011.

[52] K. Zhou, J. Hou, H. Fu, B. Wei, and Y. Liu. Estimation of relative permeability curves using an improved levenberg-marquardt method with simultaneous perturbation jacobian approximation. *Journal of Hydrology*, 544:604–612, 2017.

[53] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6. ACM, 2015.

[54] H. Pan, H. Wong, V. Kapila, and M. S. de Queiroz. Experimental validation of a nonlinear backstepping liquid level controller for a state coupled two tank system. *Control Engineering Practice*, 13(1):27–40, 2005.

[55] M. Lozano, F. Herrera, and J. R. Cano. Replacement strategies to preserve useful diversity in steady-state genetic algorithms. *Information Sciences*, 178(23):4421–4433, 2008.

[56] F. G. Lobo, C. F. Lima, and Z. Michalewicz. *Parameter Setting in Evolutionary Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1st edition, 2007.

[57] A. Fiszelew, P. Britos, A. Ochoa, H. Merlino, E. Fernández, and R. García-Martínez. Finding optimal neural network architecture using genetic algorithms. 27:15–24, 2007.