Hochschule Flensburg
University of applied Sciences

MASTER-THESIS

Thema: Neuroevolution mit MPI - Analyse und Optimierung von NEAT für ein verteiltes System

von: Simon Hauck

Matrikel-Nr.:	660158
Studiengang:	Angewandte Informatik
Betreuer/in und Erstbewerter/in:	Prof. Dr. rer. nat. Tim Aschmoneit
Zweitbewerter/in:	Prof. Dr. rer. nat. Torben Wallbaum
Ausgabedatum:	04.05.2020
Abgabedatum:	04.10.2020

Zusammenfassung

Neuroevolutionäre Algorithmen werden im Vergleich zu klassischen Verfahren wie dem Backpropagation Algorithmus weniger häufig zum Optimieren von künstlichen neuronalen Netzen eingesetzt. Allerdings sind Algorithmen dieser Art gut parallelisierbar, was im Bereich *High Performance Computing* ein entscheidender Vorteil sein kann. In dieser Arbeit wird der NEAT Algorithmus zuerst sequenziell implementiert und die benötigte Ausführungszeit in verschiedenen Optimierungsproblemen analysiert. Auf Basis der dabei erhaltenen Ergebnisse wird eine parallelisierte Implementierung mit MPI erstellt. Das Verfahren wird auf einem Beowulf Cluster evaluiert und die Ausführungszeit im Vergleich zum sequenziellen Verfahren bewertet. Die Ergebnisse zeigen eine stark reduzierte Ausführungszeit und insgesamt sehr gute Effizienzwerte.

Abstract

Neuroevolutionary algorithms are used less frequently to optimize artificial neural networks compared to more traditional methods like the backpropagation algorithm. However, algorithms of this type can easily be parallelized, which can be an important advantage in the field of*High Performance Computing*. In this thesis a sequential version of the NEAT Algorithm is implemented and the required execution time is analyzed in various optimization problems. Based on the obtained results, a parallelized implementation with MPI is created. The new implementation will be evaluated on a Beowulf cluster and the execution time is compared to the sequential implementation. The results show a greatly reduced execution time and overall very good efficiency values.

Inhaltsverzeichnis

Abbildungsverzeichnis

Akronymverzeichnis

API	*Application Programming Interface*
CPU	*central processing unit*
EA	Evolutionäre Algorithmen
GIL	*Global Interpreter Lock*
GPU	*graphics processing unit*
HPC	*High Performance Computing*
KNN	Künstliche neuronale Netze
MDP	*Markov Decision Process*
MPI	*Message Passing Interface*
NEAT	*NeuroEvolution of Augmenting Topologies*
OpenMP	*Open Multi-Processing*
PNS	Periphere Nervensystem
RAM	*random-access memory*
SC	Super Computer
SSH	*Secure Shell*
tanh	Tangens Hyperbolicus
TWEANN	*Topology and Weight Evolving Artificial Neural Network*
ZNS	Zentrale Nervensystem

1 Motivation

In den letzten Jahren sind mit Künstlichen neuronalen Netzen (KNN) wissenschaftliche Durchbrüche in verschiedenen Themengebieten erreicht worden. Dementsprechend besteht ein hohes Interesse an diesem Forschungsgebiet. Dies trifft auch auf die generelle Öffentlichkeit zu. Beispielsweise hat der Nutzer *SethBling* im Jahr 2015 auf der Plattform *YouTube* ein Video veröffentlicht, indem er den *NeuroEvolution of Augmenting Topologies* (NEAT) Algorithmus vorstellt, welcher zum Optimieren von KNN genutzt wird. Zum Zeitpunkt dieser Arbeit hat das Video fast zehn Millionen Aufrufe und ist zu 98% positiv bewertet [1].

Für den großen Erfolg von KNN sind mehrere Gründe verantwortlich. Einer hiervon ist, dass es sich prinzipiell um eine mathematische Abbildungsvorschrift handelt, die Eingabewerte entsprechend einfacher Rechenoperationen zu Ausgabewerten konvertiert. Dieses Konzept kann mit unterschiedlichen Konfigurationen leicht für diverse Aufgabengebiete angepasst werden. Der dabei entstehende Implementierungsaufwand ist meistens gering. Für den erfolgreichen Einsatz von KNN wird zusätzlich ein Optimierungsverfahren benötigt. Dieses wird in der sogenannten Lern- oder Trainigsphase auf das KNN angewendet. Dabei sollen die internen Parameter so verändert werden, dass die Ausgabewerte den gewünschten Ergebnissen entsprechen.

Prinzipiell gibt es verschiedene Optimierungsverfahren. Diese können unterschiedliche Ansätze und Schwerpunkte umsetzen. Ein sehr bekanntes und häufig genutztes Verfahren ist der Backpropagation Algorithmus, welcher zu den gradientenbasierte Verfahren gehört. Eine Alternative hierzu sind die neuroevolutionären Algorithmen, welche von der natürlichen Evolution inspiriert sind. Bei diesen Verfahren wird ein künstliche Evolution simuliert, mit dem Ziel die KNN zu optimieren. Dabei werden diverse Begriffe wie die Selektion, Rekombination und Mutation vom biologischen Vorbild übernommen. Einer der bekanntesten Vertreter neuroevolutionäre Algorithmen ist der bereits genannte NEAT Algorithmus. Dieser hat im Vergleich zu anderen Algorithmen beeindruckende Ergebnisse erzielt und dient

als Basis für viele Erweiterungen.

1.1 Ziel der Arbeit

Das Optimieren eines KNN ist zeitlich gesehen sehr aufwändig. Trainingszeiten von mehreren Stunden oder Tagen können notwendig sein, bis das KNN eine ausreichende Genauigkeit erzielt. Neuroevolutionäre Algorithmen sind hierbei keine Ausnahme. Das Simulieren einer künstlichen Evolution benötigt häufig längere Laufzeiten als alternative Verfahren, wie zum Beispiel der Backpropagation Algorithmus [2]. Dies ist mitunter ein Grund, warum neuroevolutionäre Algorithmen bedeutend seltener eingesetzt werden als gradientenbasierte Verfahren. Ein Vorteil für die neuroevolutionäre Algorithmen ist, dass sie gut parallelisierbar sind [3]. Im Bereich von *High Performance Computing* (HPC), wo viele unabhängige Prozessoren zur Verfügung stehen, kann es ein entscheidender Vorteil sein, wenn durch das Hinzufügen von weiteren Rechenressourcen die Ausführungszeit verringert werden kann. Häufig ist dies eine der kosteneffizientesten Möglichkeiten die Ausführungszeit von Algorithmen zu reduzieren. Wenn diese Eigenschaft auf neuroevolutionäre Algorithmen zutrifft, können sie einen Vorteil gegenüber anderen Verfahren bieten.

Das Ziel dieser Arbeit besteht aus drei Teilen. Zuerst soll das Optimierungsverfahren NEAT stellvertretend für neuroevolutionäre Algorithmen in der Sprache Python als sequenzielles Verfahren implementieren werden. Hierfür ist eine geeignete Bibliothek zu erstellen, die eine Schnittstelle für verschiedene Optimierungsprobleme bietet. Im zweiten Schritt ist eine Analyse des Verfahrens anhand mehrere Beispiele durchzuführen. Die hierfür benötigten Ausführungszeiten müssen erfasst und ausgewertet werden. Auf Basis der Ergebnisse ist im dritten Teil eine geeignete Strategie für die Parallelisierung zu wählen und zu implementieren. Danach sollen die zuvor verwendeten Beispiele wiederholt werden. Am Ende ist eine Beurteilung über die Effizienz abzugeben. Eine Anforderung an das parallelisierte Verfahren ist, dass es für den Bereich HPC geeignet sein soll. Daher wird für die Kommunikation innerhalb der parallelisierten Implementierung der weit verbreitete Standard MPI verwendet.

1.2 Struktur der Arbeit

Kapitel 2 sind die benötigten Grundlagen vorgestellt. Zuerst wird auf den Aufbau und die Funktionsweise von KNN eingegangen. Im Anschluss dazu sind die

Evolutionäre Algorithmen (EA) beschrieben. Für diese werden zuerst die allgemeinen Konzepte vorgestellt und danach wird veranschaulicht, wie diese auf neuroevolutionäre Algorithmen übertragen werden. Der dritte Teil der Grundlagen stellt den NEAT Algorithmus im Detail vor. Dieser wird stellvertretend für die neuroevolutionären Algorithmen verwendet. Im letzten Teil der Grundlagen wird auf HPC eingegangen und das Protokoll MPI vorgestellt. Dieses soll für die Parallelisierung verwendet werden. Im Kapitel 3 wird die Softwarearchitektur vorgestellt. Dies umfasst eine Beschreibung der verwendeten Klassen und Schnittstellen. Zusätzlich wird im Rahmen dieses Kapitels eine sequenzielle Implementierung von NEAT erstellt. In Kapitel 4 wird diese analysiert. Dabei wird zuerst die korrekte Funktionalität verifiziert und danach die Laufzeit des Verfahrens in verschiedenen Beispielen erfasst und ausgewertet. Auf Basis der dabei erhaltenen Ergebnisse wird in Kapitel 5 das Verfahren mit MPI parallelisiert und die Laufzeit erneut gemessen. Zuletzt wird die Effizienz der Parallelisierung im Vergleich zum sequenziellen Verfahren bewertet. Der erstellte Programmcode, die verwendeten Beispielprojekte und Diagramme sind auf *GitHub* unter folgendem Link verfügbar:

`https://github.com/simonhauck/MPI_NEAT`

2 Grundlagen

In diesem Kapitel werden die benötigten Grundlagen für die Arbeit vorgestellt. Diese bestehen aus vier Teilbereichen. Zuerst wird in Kapitel 2.1 auf die grundlegende Funktion von neuronalen Netzen eingegangen. Im darauf folgenden Kapitel werden die Konzepte der evolutionären Algorithmen vorgestellt. Diese dienen als Basis für die neuroevolutionären Algorithmen zu denen auch das Verfahren aus Kapitel 2.3 gehört. Dieses wird im Rahmen dieser Arbeit implementiert. Zuletzt wird in Kapitel 2.4 auf die theoretischen Grundlagen der Parallelisierung mit MPI eingegangen.

2.1 Neuronale Netze

Klassische Algorithmen in der Informatik beschreiben, mit welchen Schritten ein spezielles Problem gelöst werden kann. In vielen Anwendungsfällen, wie zum Beispiel beim Sortieren einer Liste, verwenden Computersysteme diese und lösen das gegebene Problem schneller und effizienter als es Menschen möglich ist. Andere Aufgaben hingegen können von Menschen ohne Aufwand gelöst werden, stellen aber Computersysteme vor große Herausforderungen. Hierzu zählt unter anderem die Klassifizierung von Bildern. Ein Mensch kann beispielsweise Bilder von Hunden und Katzen unabhängig von Blickwinkel und Bildqualität unterscheiden beziehungsweise richtig zuordnen. Trotzdem ist es aufwendig, für solche Probleme klassische Algorithmen zu entwickeln, da die Lösung von vielen subtilen Faktoren abhängt [4]. Häufig werden in diesen Aufgabenfeldern Künstliche neuronale Netze (KNN) eingesetzt, welche von biologischen neuronalen Netzen inspiriert sind und zum Forschungsgebiet des maschinellen Lernens gehören. Auch wenn die KNN heute aktuell sind und viel Aufmerksamkeit erhalten, bildet die bereits 1943 veröffentlichte Arbeit von McCulloch und Pitts die Grundlage für das Forschungsgebiet. In der Arbeit wird ein einfaches neuronales Netz mit Schwellwerten entwickelt, das die Berechnung von logischen und arithmetischen Funktionen ermöglicht [5]. In den folgenden Jahrzehnten wurde die Funktionsweise der neuronalen Netze weiterentwickelt und der Einsatz in verschiedenen Aufgabenfeldern ermöglicht.

Hierzu zählen neben der Klassifizierung von Bildern [6] unter anderem das Erkennen und die Interpretation von Sprache [7], [8] sowie das selbständige Lösen von Computer- und Gesellschaftsspielen [9], [10]. Bevor im weiteren Verlauf dieses Kapitels auf den genauen Aufbau und die Funktionsweise von KNN eingegangen wird, sind im folgenden die biologischen neuronalen Netze vorgestellt.

2.1.1 Biologische neuronale Netze

Das Forschungsgebiet der KNN ist von den erfolgreichen biologischen neuronalen Netzen inspiriert, wie zum Beispiel dem menschlichen Gehirn [4]. In diesem Abschnitt werden die Eigenschaften betrachtet, die das Vorbild erfolgreich machen und für die KNN übernommen werden sollen. Im Zuge dessen wird ein grober Überblick über die Struktur und Funktionsweise des menschlichen Gehirns gegeben.

Jede Sekunde erfassen die Rezeptoren des menschlichen Körpers unzählige Reize, wie zum Beispiel Licht, Druck, Temperatur und Töne. Die Reize werden anschließend elektrisch oder chemisch kodiert und über Nervenbahnen an das Gehirn geleitet, welches die Aufgabe hat, diese zu filtern, zu verarbeiten und entsprechend zu reagieren. Als Reaktion können zum Beispiel Signale an entsprechende Muskeln oder Drüsen gesendet werden [11]. Hierbei zeichnet sich das biologische neuronale Netz durch drei Eigenschaften aus, die klassische Algorithmen entweder nicht besitzen oder nur schwer umsetzen können. Ziel ist es, diese auf die KNN zu übertragen [4].

1. **Fähigkeit zu Lernen**
   Das menschliche Gehirn ist nicht wie ein klassischer Algorithmus für seine Aufgaben programmiert. Stattdessen besitzt es die Fähigkeit, durch Nachahmen oder Ausprobieren zu lernen [4]. Dafür wird das angestrebte Ergebnis mit dem tatsächlich erzielten verglichen und das Verhalten entsprechend angepasst. Dies ermöglicht es Menschen, verschiedene Aufgabengebiete erfolgreich zu lösen und sich ändernden Anforderungen anzupassen.

2. **Fähigkeit zur Generalisierung**
   Auch für unbekannte Situationen findet das Gehirn meist plausible Lösungen, da es die Fähigkeit zur Generalisierung besitzt [4]. Das bedeutet, dass viele Situationen bereits bekannten Problemen zugeordnet werden können, mithilfe derer eine passende Verhaltensstrategie ausgewählt wird.

3. **Toleranz gegenüber Fehlern**
   Zudem zeichnen sich biologische neuronale Netze durch eine hohe Fehlertole-

ranz gegenüber verrauschten Daten aus. Beim zuvor genanntem Beispiel der Klassifizierung kann ein Teil des Bildes fehlen oder unscharf sein, trotzdem kann das abgebildete Motiv richtig zugeordnet werden.

Struktur des menschlichen Gehirns

Das Forschungsgebiet der Neurowissenschaften befasst sich unter anderem mit dem menschlichen Gehirn, dessen Funktionsweise auch heute noch nicht vollständig erforscht ist. Dennoch ist schon seit 1861 durch die Arbeit von Paul Broca bekannt, dass es im menschlichen Gehirn verschiedene Regionen mit unterschiedlichen Aufgaben gibt [12]. Zum Beispiel wird das sogenannte Kleinhirn (Cerebellum) für einen Großteil der motorischen Koordination verwendet, während das Großhirn (Telencephalon) unter anderem visuelle Reize empfängt [4]. Trotz der unterschiedlichen Aufgaben haben alle Bereiche des Gehirns einen gemeinsamen Grundbaustein, die sogenannten Neuronen [12]. Im Folgenden wird der Aufbau und die Funktionsweise von diesen oberflächlich in Bezug auf die später vorgestellten künstlichen Neuronen betrachtet. Für einen vollständigen Überblick und eine genaue Beschreibung der Vorgänge ist auf entsprechende Fachliteratur zu verweisen.

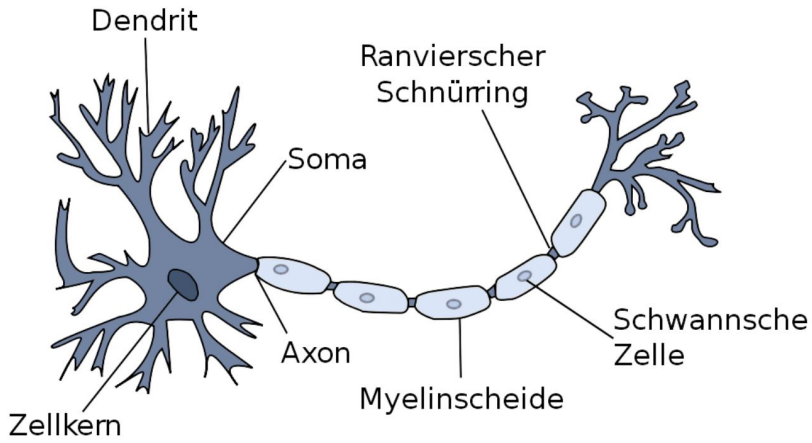

Abbildung 2.1: Schematische Abbildung einer Nervenzelle [4].

Das menschliche Gehirn besitzt ungefähr 10^{11} einzelne Neuronen, deren schematischer Aufbau in Abbildung 2.1 dargestellt ist. Jedes Neuron besitzt einen Zellkern, der sich im Zellkörper (Soma) befindet. Von dem Zellkörper gehen mehrere Fasern aus, die Dendriten genannt werden [12]. An diesen befinden sich Synapsen, welche als Übertragungsstelle fungieren und elektrische oder chemische Signale von Rezeptoren oder anderen Neuronen empfangen [4]. Typischerweise empfängt ein Neuron

Signale von 2000 bis 10.000 anderen Nervenzellen [13]. Synapsen, die elektrische Signale erhalten, haben eine starke, direkte, nicht regulierbare Verbindung vom Sender zum Empfänger. Diese sind für hart kodierte Verhaltensmechanismen nützlich, wie zum Beispiel den Fluchtreflex. Die chemische Synapse hingegen ist nicht direkt mit dem Sender verbunden, sondern durch den synaptischen Spalt getrennt. Zur Übertragung eines elektrischen Signals wird dieses auf der präsynaptischen Seite in ein chemisches Signal kodiert, indem Neurotransmitter freigesetzt werden. Diese können über den synaptischen Spalt übertragen und anschließend auf der postsynaptischen Seite wieder in ein elektrisches Signal kodiert werden. Ein großer Vorteil dieser Übertragungsart ist die Regulierbarkeit [4]. Verschiedene Neurotransmitter können unterschiedliche Effekte auf das Neuron haben, beispielsweise anregend (exzitatorisch) oder hemmend (inhibitorisch) wirken [14]. Zusätzlich kann die Menge der freigesetzten Neurotransmitter die Stärke des Signals beeinflussen [4]. Langfristig gesehen können auch neue Verbindungen bzw. Synapsen entstehen oder alte aufgelöst werden. Es wird angenommen, dass dies die Grundlage des Lernens im menschlichen Gehirn ist [12].

Sowohl die anregenden als auch hemmenden Signale werden über die Dendriten an den Axonhügel weitergeleitet, welcher sich zwischen dem Soma und dem Axon befindet. Dort werden die Signale akkumuliert. Beim Überschreiten eines gewissen Schwellwerts wird ein elektrischer Impuls erzeugt, den das Axon weiterleitet [14]. Das Axon ist typischerweise einen Zentimeter, in Ausnahmen sogar bis zu einem Meter lang und wird von der Myelinscheide umgeben, die unter anderem Schutz vor mechanischer Überbeanspruchung bietet [12]. Zusammen mit den Ranvierschen Schnürringen ermöglicht sie zudem eine schnellere Weiterleitung des Aktionspotenzials [14]. Das Axon endet mit dem sogenannten Endknopf, auch Axonterminal genannt. Dieses ist mit den Synapsen von anderen Neuronen verbunden und setzt beim Eintreffen eines Signals die Neurotransmitter frei, wodurch das Signal übertragen wird [14]. Typischerweise gibt ein einzelnes Neuron sein Signal an 1000 bis 10.000 andere Neuronen weiter, in Extremfällen sogar an bis zu 150.000 andere Neuronen [13], die alle parallel arbeiten. So entsteht ein großes und leistungsfähiges neuronales Netz.

2.1.2 Künstliche neuronale Netze

KNN sind ein mathematisches Modell, das im Vergleich zum biologischen Vorbild stark vereinfacht und idealisiert ist. Trotzdem können unterschiedliche mathematische Funktionen abgebildet werden. In diesem Kapitel werden die grundsätzliche

Funktionsweise sowie die einzelnen Komponenten der KNN vorgestellt.

Abbildung 2.2: KNN als Blackbox mit einem Eingabe- und Ausgabevektor

Betrachtet man ein KNN als Blackbox (Abbildung 2.2), gibt es eine gewisse Anzahl an Eingabewerten, die in einem Eingabevektor kodiert sind und eine Anzahl an Ausgaben, die in einem Ausgabevektor kodiert sind [15]. Die Eingaben werden im Fall der KNN nicht durch Rezeptoren erfasst, sondern sind durch ein Optimierungsproblem gegeben. Der Ausgabevektor soll das gewünschte Ergebnis enthalten. Je nach Optimierungsproblem kann die Interpretation von diesem variieren. Betrachtet man die Struktur der KNN, sind einige Ähnlichkeiten zum biologischen Vorbild erkennbar. Diese werden im Folgenden genauer betrachtet [13]:

1. **Neuronen**
   Ähnlich zu den biologischen neuronalen Netzen, besteht auch das KNN aus vielen Neuronen [13]. Dies sind einfache Recheneinheiten, die primitive Funktionen bestimmen können [15] und deren genaue Funktionsweise in Kapitel 2.1.3 erläutert wird. Vorweggenommen sei, dass ein Neuron mehrere Eingabewerte besitzt, welche gewichtet sind und akkumuliert werden. Hierbei entsteht ein skalarer Ausgabewert, der den Aktivierungsgrad des Neurons repräsentiert und von anderen Neuronen als Eingabe verwendet werden kann [4].

2. **Gerichtete gewichtete Verbindungen**
   Wie in der Betrachtung der Neuronen angedeutet, sind diese über gerichtete Verbindungen miteinander vernetzt. Der Aktivierungszustand eines Neurons wird entsprechend der Verbindungen an die Zielneuronen weitergegeben, welche diesen Wert als Eingabe verarbeiten. Wie bei den biologischen neuronalen Netzen auch, können Eingaben unterschiedlich stark anregend oder hemmend wirken. Dies wird bei den KNN über Gewichte in den Verbindungen realisiert [13].

3. **Struktur und Gewichte**
   Der Ausgabevektor eines KNN ist abhängig von der Struktur des Netzes und der Gewichte in den einzelnen Verbindungen. Für das erfolgreiche Lösen eines Optimierungsproblems muss ein KNN die richtige Kombination aus Neuronen, Netzstruktur und gewichteten Verbindungen besitzen. Diese müssen durch Lernverfahren bestimmt werden, auf die in Kapitel 2.1.5 näher eingegangen wird.

Trotz der vorgestellten Ähnlichkeiten sind einige Unterschiede zwischen den biologischen neuronalen Netzen und den KNN zu verdeutlichen, für die als Beispiel der Größenunterschied zu nennen ist. Das menschliche Gehirn mit seinen 10^{11} Neuronen besitzt pro Neuron ungefähr 10^4 Verbindungen, wogegen die meisten KNN nur 10^2 bis 10^4 Neuronen mit insgesamt 10^5 Verbindungen besitzen. Auch werden keine chemischen Effekte, die auf benachbarte Neuronen wirken, sowie zeitliche und räumliche Lokalitätsprinzipien beachtet [13]. Aus diesen Gründen sind die KNN keine Nachbildung der biologischen neuronalen Netze, sondern verwenden diese ausschließlich als Inspiration.

2.1.3 Das Neuron

In diesem Kapitel wird die Funktionsweise der einzelnen Neuronen betrachtet. Hierfür werden drei Phasen, die Propagierungsfunktion, die Aktivierungsfunktion sowie die Ausgabefunktion vorgestellt, in denen der Ausgabewert eines einzelnen Neurons berechnet wird. Betrachtet man ein KNN, führen typischerweise mehrere Verbindungen zu einem Neuron j, welche von den Neuronen $i_1, i_2, ..., i_n$ ausgehen [4]. Dies ist schematisch in Abbildung 2.3 dargestellt.

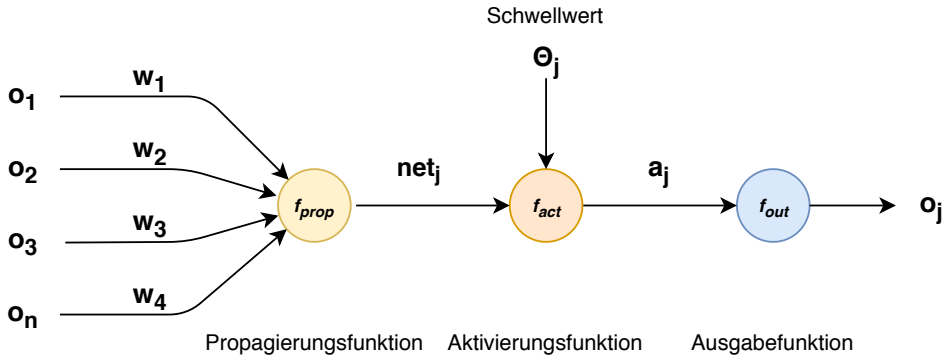

Abbildung 2.3: Schematische Darstellung eines einzelnen künstlichen Neurons

Propagierungsfunktion

Die Ausgabewerte $o_{i_1}, o_{i_2}, ..., o_{i_n}$ der Neuronen $i_1, i_2, ..., i_n$ werden als Eingabewerte für das Neuron j verwendet. Für jeden Eingabewert existiert ein entsprechendes Gewicht $w_1, w_2, ..., w_n$ [4]. Somit repräsentiert w_{ij} das Gewicht für die Verbindung von Neuron i zu Neuron j [13]. Die Propagierungsfunktion f_{prop} berechnet die Netzeingabe net_j, welche in der darauffolgenden Phase weiterverwendet wird [4].

$$net_j = f_{prop}(o_1, o_2, ..., o_n, w_1, w_2, ..., w_n)$$

Die meist verwendete Propagierungsfunktion, welche auch in den späteren Beispielen genutzt wird, ist die gewichtete Summe. Hierbei werden entsprechend der Formel die Werte o_i mit dem entsprechenden Gewicht w_i multipliziert und aufsummiert [4]:

$$net_j = \sum_i (o_i \cdot w_{i,j})$$

Aktivierungsfunktion

Der Aktivierungszustand $a_j(t)$ gibt den Grad der Aktivierung von Neuron j zum Zeitpunkt t an. Ein neuer Aktivierungszustand zum Zeitpunkt $t + 1$ wird mit der Aktivierungsfunktion f_{act} berechnet. Diese berücksichtigt nicht nur die Netzeingabe $net_j(t)$, sondern auch den vorherigen Aktivierungszustand $a_j(t)$ und den Schwellwert Θ der Aktivierungsfunktion [13]. Ein Schwellwert Θ_j, auch *Bias* genannt, ist dem Neuron j zugeordnet und gibt die Stelle an, an welcher die Aktivierungsfunktion die größte Steigung hat [4]. Somit kann die Berechnung der Aktivierung $a_j(t + 1)$ durch folgende Formel ausgedrückt werden [13]:

$$a_j(t + 1) = f_{act}(a_j(t), net_j, \Theta_j)$$

Bei der Berechnung kommt dem Schwellwert Θ eine besondere Bedeutung zu. Oftmals verwenden einige oder alle Neuronen eines KNN dieselbe Aktivierungsfunktion, die Schwellwerte hingegen unterscheiden sich je nach Neuron. Des Weiteren sei angemerkt, dass die vorherige Aktivierung $a_j(t)$ je nach Netzstruktur oft keine Berücksichtigung bei der Berechnung findet [4]. Zudem wird in der Praxis bei Verwendung der gewichteten Summe als Propagierungsfunktion der Schwellwert eines Neurons oft schon in der ersten Phase miteinbezogen. Hierdurch ändert sich die Berechnung der Netzeingabe zu $net_j = \sum_i (o_i \cdot w_{i,j}) - \Theta_j$. Bei der Berechnung der Aktivierungsfunktion gilt dann $\Theta_j = 0$.

Je nach Anwendungsgebiet können verschiedene Aktivierungsfunktionen mit unterschiedlichen Eigenschaften eingesetzt werden. Vier Beispiele sind im Folgenden vorgestellt, für die angenommen wird, dass $\Theta_j = 0$ ist. Das einfachste Beispiel für eine Aktivierungsfunktion ist die sogenannte binäre Schwellwertfunktion, welche nur die Werte 0 und 1 zurückgeben kann [4]. Die Formel hierfür ist:

$$f_{act}(net_j) = \begin{cases} 1 & \text{wenn } net_j \geq 0 \\ 0 & \text{wenn } net_j < 0 \end{cases}$$

Allerdings ist für diese Funktion der Wert der Ableitung immer 0, ausgenommen an dem Schwellwert, an welchem sie nicht differenzierbar ist. Diese Eigenschaften machen sie ungeeignet für bestimmte Lernverfahren, wie zum Beispiel den Backpropagation Algorithmus, auf den in Kapitel **??** kurz eingegangen wird [4].

Dieses Problem kann durch die Verwendung einer Sigmoidfunktion gelöst werden. Zwei bekannte Beispiele für Sigmoidfunktionen sind die logistische Funktion und der Tangens Hyperbolicus (tanh) [16]. Die logistische Funktion kann Werte von 0 bis 1 annehmen und wird berechnet durch [4]:

$$f_{act}(net_j) = \frac{1}{1 + e^{-net_j}}$$

Allerdings können neuronale Netze je nach Verfahren schneller optimiert werden, wenn das durchschnittliche Gewicht aller Verbindungen nahe 0 ist. In diesem Fall kann die tanh Funktion besser geeignet sein, da sie Werte zwischen -1 und 1 annehmen kann [16]. Das abschließend vorgestellte Beispiel ist die sogenannte *Rectifier*-Funktion. Diese wird oft in Zusammenhang mit dem Backpropagation Algorithmus erfolgreich eingesetzt [17]. Berechnet wird sie durch:

$$f_{act}(net_j) = max(0, net_j)$$

Ausgabefunktion

Die Ausgabefunktion f_{out} berechnet die Ausgabe o_j von Neuron j. Als Eingabewert wird die Aktivierung a_j verwendet [13]. Definiert ist die Funktion durch:

$$o_j = f_{out}(a_j)$$

Ähnlich der Aktivierungsfunktion ist die Ausgabefunktion in der Praxis meist global für alle Neuronen definiert. Als eine solche Ausgabefunktion wird häufig

die Identitätsfunktion verwendet, für welche $o_j = a_j$ gilt [4]. Diese kommt auch in den später vorgestellten Beispielen zur Anwendung. Ist die Ausgabe o_j berechnet, kann sie als Eingabewert für andere verbundene Neuronen dienen.

2.1.4 Netzstrukturen

Aus dem vorherigen Kapitel wird deutlich, dass die Gewichte einen großen Einfluss auf den Ausgabewert eines einzelnen Neurons haben. Der Ausgabevektor eines KNN wird zusätzlich von der Anzahl an Neuronen sowie deren Verbindungsstruktur beeinflusst. Dieses Kapitel führt verschiedene Varianten ein, die je nach Optimierungsproblem eingesetzt werden können.

Typischerweise besitzt jedes KNN Eingabe- und Ausgabeneuronen. Optional kann ein KNN beliebig viele verdeckte Neuronen enthalten. Diese werden auch als *Input*-, *Output*- und *Hidden*-Neuronen bezeichnet [13]. Die Anzahl der Eingabe- und Ausgabeneuronen ist abhängig von der Größe des Eingabe- bzw. Ausgabevektors. Für jedes Element in den Vektoren gibt es ein entsprechendes Neuron. Bei vielen Netzstrukturen werden die Neuronen des KNN verschiedenen Schichten zugeordnet. In der ersten Schicht befinden sich die Eingabeneuronen, in der letzten die Ausgabeneuronen. Dazwischen befinden sich n Schichten mit verdecken Neuronen [13].

Bei der Berechnung eines KNN werden zuerst die Werte des Eingabevektors in die entsprechenden *Input*-Neuronen eingesetzt. Anschließend werden alle Neuronen in einer bestimmten Reihenfolge aktiviert bzw. berechnet. Zuletzt bilden die Werte der *Output*-Neuronen den Ausgabevektor. Die *Hidden*-Neuronen, deren Bezeichnung darin begründet ist, dass ihr Ausgabewert nur ein Zwischenergebnis darstellt und vor dem Anwender verborgen bleibt, befinden sich zwischen den *Input*- und *Output*-Neuronen. Trotzdem sind sie ein elementarer Bestandteil der KNN und bestimmen maßgeblich deren Leistungsfähigkeit. Beispielsweise kann ein aus *Input*- und *Output*-Neuronen bestehendes KNN nur eine lineare Funktion repräsentieren. Ein KNN mit einer ausreichend großen verdeckten Schicht kann jede beliebige kontinuierliche Funktion darstellen. Mit zwei Schichten kann ein KNN sogar jede unstetige mathematische Funktion mit beliebiger Genauigkeit abbilden [12]. Je nach Art des Verbindungsmusters zwischen den Neuronen werden KNN einer von zwei Gruppen zugeordnet. Die erste Gruppe enthält Netze ohne Rückkopplung, welche auch *feedforward*-Netze genannt werden. Die zweite Gruppe sind die sogenannten *recurrent*-Netze, zu welchen KNN mit Rückkopplungen

gehören [13].

Netze ohne Rückkopplung

Die Definition der *feedforward*-Netze sieht vor, dass es keine Verbindung geben darf, die von einem Neuron j ausgeht und wieder zu diesem selbst führt. Dabei ist es irrelevant, ob eine direkte oder indirekte Verbindung über Zwischenneuronen besteht. Es entsteht ein azyklischer Graph [13] und das KNN kann infolgedessen keinen internen Zustand besitzen. Für die gleiche Eingabe wird immer dasselbe Ergebnis berechnet. Innerhalb dieser Kategorie gibt es zwei Untergruppen, die ebenenweise verbundenen KNN und die KNN, welche über sogenannte *shortcut* Verbindungen verfügen. Bei den rein ebenenweise verbundenen KNN stammen die

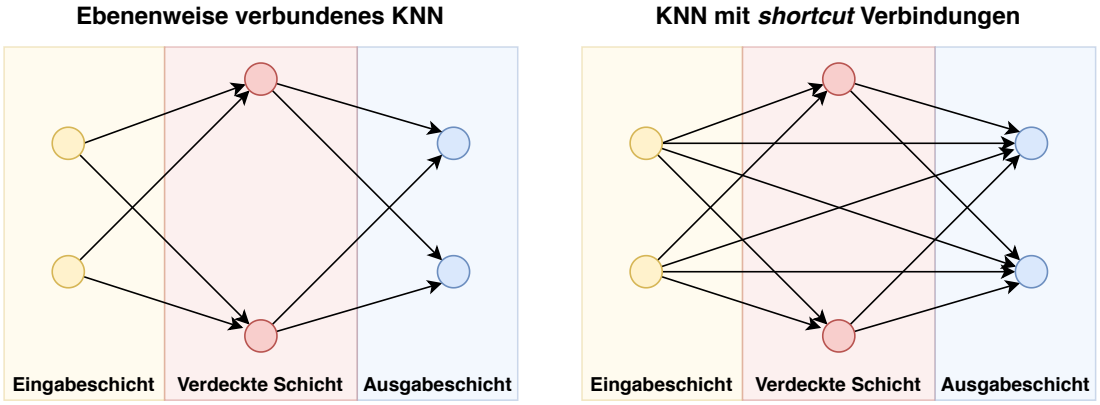

Abbildung 2.4: Links ein ebenenweise verbundenes *feedforward* KNN, rechts ein KNN mit *shortcut* Verbindungen

Eingabewerte eines Neurons immer aus der vorherigen Schicht. Der berechnete Ausgabewert eines Neurons wird nur an die Neuronen der nächsten Schicht weitergeleitet [13]. Ein Beispiel hierfür ist in Abbildung 2.4 links dargestellt. Im Gegensatz dazu stehen die KNN mit *shortcut* Verbindungen. Wie in Abbildung 2.4 rechts dargestellt, können *shortcut* Verbindungen eine oder mehrere Schichten überspringen. Für gewisse Optimierungsprobleme, unter anderem für das in Kapitel 4.2 dargestellte XOR-Problem, können so kleinere KNN erzeugt werden [13].

Netze mit Rückkopplung

Durch rückgekoppelte Verbindungen entstehen Zyklen in der Berechnung, wodurch sich das KNN selbst beeinflussen kann. Damit ist unter anderem das Zwischenspeichern von Ergebnissen möglich [12]. Somit kann die Berechnung des Ausgabevektors sowohl durch die Eingabewerte als auch durch die vorherigen Ergebnisse beeinflusst werden [18]. Wie die *feedforward*-Netze können auch die

Netze mit Rückkopplungen je nach Verbindungsart verschiedenen Untergruppen zugeordnet werden [13]. Diese sind in Abbildung 2.5 dargestellt und im Folgenden beschrieben.

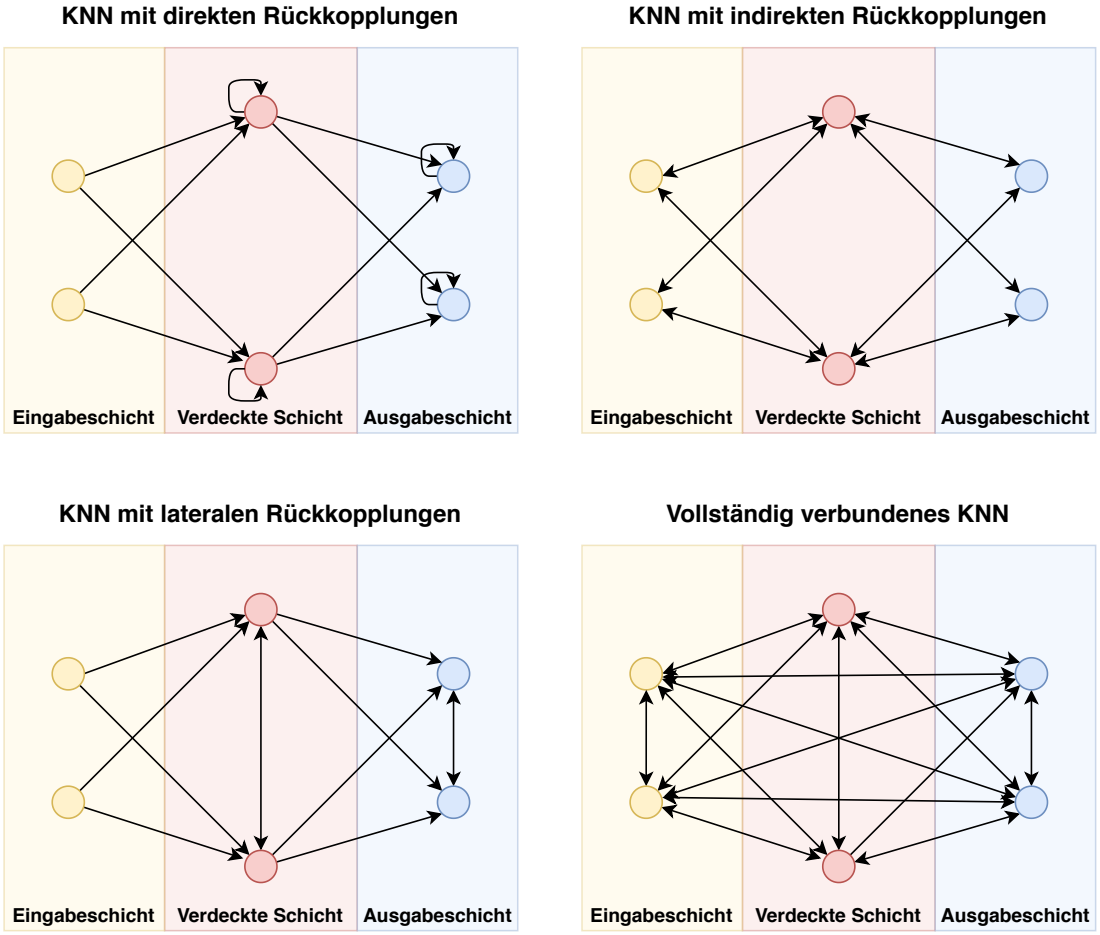

Abbildung 2.5: Schematische Darstellung des Verbindungsmusters von verschiedenen KNN mit Rückkopplungen

1. Bei KNN mit direkten Rückkopplungen können Neuronen Verbindungen zu sich selbst haben und hierdurch ihre eigene Aktivierung verstärken oder abschwächen [13].

2. Netze mit einer indirekten Rückkopplung erlauben im Gegensatz zu den *feedforward*-Netzen auch Verbindungen in die vorherige Schicht [13]. Wie bei der direkten Rückkopplung kann sich ein Neuron j selbst beeinflussen, wenn es seinen Ausgabewert an ein Neuron i der nächsten Schicht weiterleitet, welches eine Rückkopplung zu j hat [4].

3. KNN mit lateralen Rückkopplungen erlauben Verbindungen von Neuronen innerhalb einer Schicht, welche hemmend oder aktivierend wirken können.

Oft entsteht dabei ein *Winner-Takes-All*-Schema, da das beste Neuron alle anderen hemmt und sich selbst aktiviert [4].

4. Bei den vollständig verbundenen Netzen darf ein Neuron zu jedem anderen eine Verbindung besitzen. Ein Sonderfall sind hier die sogenannten Hopfield-Netze. Bei diesen müssen die Neuronen zu jedem anderen eine Verbindung besitzen mit Ausnahme zu sich selbst [4].

2.1.5 Optimierungsmöglichkeiten

In den vorherigen Kapiteln ist aufgezeigt, dass das erfolgreiche Lösen eines Optimierungsproblems mit einem KNN von vielen Faktoren abhängt. In der Praxis ist eine manuelle Bestimmung dieser bei komplexen Aufgaben nicht möglich. Aus diesem Grund muss ein Optimierungsverfahren, welches auch als Lernverfahren bezeichnet wird, angewendet werden. Ziel hierbei ist, einen Teil oder alle Parameter des KNN durch einen Algorithmus automatisch zu bestimmen. Typischerweise ist das Lernverfahren unabhängig vom eigentlichen Optimierungsproblem und kann daher in verschiedenen Bereichen ohne großen zusätzlichen Aufwand eingesetzt werden.

Ein Lernverfahren kann theoretisch auf vier verschiedene Arten die Eigenschaften eines KNN optimieren [13]. Diese sind im Folgenden kurz zusammengefasst.

1. **Modifizieren der Verbindungsgewichte**:
   Die Gewichte der einzelnen Verbindungen werden in der Praxis von allen Lernverfahren optimiert [13]. Gründe hierfür sind, dass ein KNN mehrere Millionen Verbindungen besitzen kann, welche unmöglich manuell optimiert werden können, und dass die Gewichte entscheidend für die erfolgreiche Optimierung sind.

2. **Modifizieren der Schwellwerte**:
   Die Schwellwerte der Neuronen werden wie die Gewichte von den meisten Lernverfahren optimiert. In der Praxis ist der hierbei verwendete Vorgang oft identisch mit der Gewichtsoptimierung. Dies ist möglich, wenn, wie in einigen Implementierungen umgesetzt, die Schwellwerte durch Gewichte repräsentiert werden. Hierzu wird einem KNN ein sogenanntes *Bias*-Neuron hinzugefügt, welches immer den Wert 1 hat. Von diesem gehen Verbindungen zu allen Neuronen aus. Der Schwellwert Θ_j von einem Neuron j wird durch das Gewicht $w_{\Theta j}$ repräsentiert. Dieses ist der eingehenden Verbindung vom

Bias-Neuron zugeordnet, sodass gilt $1 \cdot w_{\Theta j} = \Theta_j$. Dadurch muss bei der Berechnung eines Neurons der Schwellwert nicht mehr explizit miteinbezogen werden, sondern wird im Rahmen der Propagierungsfunktion indirekt mit den anderen gewichteten Eingaben verarbeitet. Bezüglich der Optimierung wird die Verbindung zum *Bias*-Neuron wie andere gewichtete Verbindungen behandelt [13].

3. **Hinzufügen und Entfernen von Verbindungen oder Neuronen**:
   Das Hinzufügen beziehungsweise Entfernen von Verbindungen und Neuronen ist im Vergleich zu den bereits vorgestellten Möglichkeiten aufwendig und in der Umsetzung schwieriger. Daher ist es in vielen bekannten Algorithmen nicht implementiert. Bei diesen muss die Struktur mithilfe von Expertenwissen oder Erfahrung festgelegt werden [19], andernfalls ist eine geeignete Struktur experimentell zu ermitteln. Da dieses Vorgehen häufig nicht effizient ist, gibt es dennoch einige Algorithmen, die das Hinzufügen und Entfernen von Strukturen bei der Optimierung nutzen. Diese gehören oft zu der Klasse der neuroevolutionären Algorithmen, auf welche in Kapitel 2.2.3 eingegangen wird [4].

4. **Ändern der Propagierungs-, Aktivierungs- und Ausgabefunktion**:
   Die Optimierung der verwendeten Propagierungs-, Aktivierungs- und Ausgabefunktion ist theoretisch möglich, die Umsetzung ist in der Praxis allerdings nicht sehr verbreitet [13]. Auch in dieser Arbeit werden diese Funktionen nicht durch einen Algorithmus angepasst und daher nicht weiter betrachtet.

2.1.6 Arten von Optimierungsverfahren

In Kapitel 2.1.5 sind Optimierungsmöglichkeiten aufgelistet, welche von einem Lernverfahren in der sogenannten Trainingsphase des KNN angepasst werden können. Ziel ist, dass am Ende dieser Phase der Ausgabevektor des KNN dem gewünschten Ergebnis entspricht. Voraussetzung hierfür ist, dass das gewünschte Ergebnis erkennbar ist [13]. Bei den Lernverfahren wird grundsätzlich zwischen dem überwachten, unüberwachten und bestärkenden Lernen differenziert, welche unterschiedliche Arten des Lernens für verschiedene Aufgabenstellungen repräsentieren. Im Folgenden wird ein Überblick über diese gegeben. Für eine genaue Beschreibung und die dazugehörigen Algorithmen wird auf entsprechende Fachliteratur verwiesen.

Überwachtes Lernen

Das überwachte Lernen, auch *supervised learning* genannt, wird häufig mit dem Backpropagation Algorithmus und seinen Derivaten umgesetzt und beruht auf bekannten Beispieldaten. Diese müssen in großer Anzahl schon vor dem Lernvorgang vorhanden sein und den Eingabevektor sowie den gewünschten Ausgabevektor des KNN enthalten [13]. In der sogenannten Trainingsphase analysiert das Lernverfahren die vorhanden Beispieldaten mit dem Ziel, Muster zu extrahieren. Am Ende der Trainingsphase soll das KNN nicht nur die korrekte Lösung für die bekannten Beispiele angeben können, sondern auch für ähnliche, unbekannte Eingabedaten. Damit ist die Eigenschaft der Generalisierung erfüllt. [13]. Um den Erfolg des Verfahrens zu überprüfen, werden die Beispieldaten in Trainings- und Testdaten unterteilt. Die Trainingsphase wird nur mit den Trainingsdaten durchgeführt, sodass die Testdaten dem KNN unbekannt sind. Ist diese Phase abgeschlossen, weil beispielsweise das KNN eine gute Genauigkeit erreicht hat, werden die Testdaten zur Validierung eingesetzt. Hierbei wird überprüft, ob das KNN auch für unbekannte Eingabevektoren die richtigen Ergebnisse berechnet [4]. Diese Art des Lernens ist im Vergleich zu den anderen Varianten sehr schnell [13], aber das Anwenden ist nicht in jeder Situation möglich. Liegen keine Beispiele vor, kann das KNN nicht trainiert werden. Sind die Beispieldaten fehlerhaft oder verrauscht, kann das Training langsam, nicht zufriedenstellend oder unmöglich sein.

Unüberwachtes Lernen

Beim unüberwachten Lernen, auch *unsupervised learning* genannt, stehen ebenfalls Beispieldaten zur Verfügung, allerdings enthalten diese nur den Eingabevektor und keine dazugehörigen Ausgabevektoren. Ziel solcher Lernverfahren ist, Muster in den Eingabedaten automatisch zu erkennen und diese verschiedenen Gruppen zuzuordnen, sodass ähnliche Eingabewerte derselben Gruppe zugewiesen werden [13]. Solche Verfahren können einige Vorteile gegenüber dem überwachten Lernen bieten [20]. Zum Beispiel müssen vor dem Training keine Beispieldaten mit Ausgabevektoren vorliegen, welche teilweise sehr teuer und aufwendig zu erstellen sind. Des Weiteren kann je nach Algorithmus die Anzahl an vorhandenen Gruppen automatisch zugewiesen werden. So können auch unterschwellige Muster, die nicht von einem Menschen erkannt werden würden, die Zuweisung beeinflussen [20].

Bestärkendes Lernen

Die letzte Klasse ist das bestärkende Lernen, auch *reinforcment learning* genannt. Typischerweise wird diese Art des Lernens in dynamischen Umgebungen eingesetzt, in welcher ein sogenannter Agent mit einer Umgebung interagiert. Ein häufig genannter Begriff in diesem Kontext ist der *Markov Decision Process* (MDP), welcher in Abbildung 2.6 dargestellt ist und anhand dessen im Folgenden das bestärkende Lernen beschrieben ist [21].

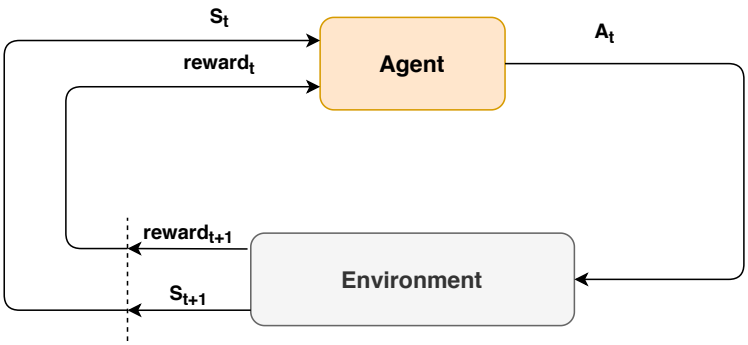

Abbildung 2.6: Interaktion des Agenten mit der Umgebung im MDP

Zwei wichtige Grundkomponenten des MDPs sind der Agent und die Umgebung. Der aktuelle Zustand der Umgebung zu einem Zeitpunkt t wird durch die Variable S_t repräsentiert. Ist die Umgebung zum Beispiel ein Computerspiel, kann S_t unter anderem die aktuelle Position sowie Zielkoordinaten enthalten. Der Zustand S_t steht dem Agenten zur Verfügung, der daraufhin eine Aktion A_t ausführt. Als Basis für die Entscheidung kann ein KNN dienen, welches als Eingabevektor den aktuellen Zustand verwendet und einen Ausgabevektor mit der gewählten Aktion erzeugt. Die verfügbaren Aktionen sind je nach System unterschiedlich. So können zum Beispiel bei der Steuerung von Robotern sowohl direkte Steuersignale für die Motoren ausgegeben werden als auch *high-level* Entscheidungen wie die Bewegungsrichtung. Nach Ausführung der Aktion wird der Zustand der Umgebung entsprechend angepasst und ein neuer Zustand S_{t+1} entsteht [21], für den der Agent eine neue Aktion auswählen kann. Zusätzlich wird ein Belohnung, auch als *reward* bezeichnet, vergeben. Dies ist ein numerischer Wert, der angibt, wie richtig oder falsch die gewählte Aktion war [13]. Eine richtige Aktion zeichnet sich dadurch aus, dass sie den Agenten näher an sein gewünschtes Ziel bringt. Im zuvor genannten Beispiel des Computerspiels ist der *reward* größer, wenn der Agent die Distanz zum Ziel verringert und kleiner bzw. negativ, wenn der Agent sich wieder entfernt. Ziel eines Optimierungsalgorithmus ist, die Summe der erhaltenen Belohnungen zu maximieren. Hierdurch ergeben sich komplexe Anforderungen an

das Lernverfahren. Bei der Entscheidung, welche Aktion A_t bei einem Zustand S_t den meisten Erfolg verspricht, muss sowohl der direkte als auch zukünftige *reward* berücksichtigt werden [21]. Dies ist notwendig, da ein Agent viele Aktionen in derselben sich ändernden Umgebung ausführt und eine Entscheidung Auswirkungen auf die Zukunft hat. Somit kann es bei vielen Optimierungsproblemen lohnenswert sein, zur Schaffung einer besseren Ausgangslage zuerst eine schlechte Belohnung in Kauf zu nehmen, wenn dadurch im weiteren Verlauf größere Belohnungen erreicht werden können. Eine weitere Herausforderung für solche Algorithmen ist, dass ein Gleichgewicht zwischen dem Nutzen von Erfahrung und Ausprobieren gefunden werden muss. Möglichst hohe Belohnungen kann ein Agent nur erhalten, wenn er bekannte Entscheidungen trifft, die in der Vergangenheit erfolgreich waren. Allerdings müssen auch neue unbekannte Aktionen ausgewählt werden, da diese unter Umständen besser sein können. Für ein gutes Lernverfahren ist es notwendig, eine Kombination aus beidem zu ermöglichen [21]. Bei vielen Optimierungsproblemen wird das bestärkende Lernen erfolgreich eingesetzt. Ein Nachteil jedoch ist die lange Laufzeit im Vergleich zu anderen Verfahren, wie zum Beispiel dem überwachten Lernen. Grund hierfür ist, dass eine niedrige Belohnung keine Aussage darüber trifft, wie das KNN optimiert werden muss, um bessere Ergebnisse zu erzielen. Anpassungen können sich sowohl positiv als auch negativ auf den Agenten auswirken [13].

2.1.7 Backpropagation Algorithmus

Das Backpropagation Verfahren ist ein bekanntes und weit verbreitetes Lernverfahren, welches häufig im Zusammenhang mit dem überwachten Lernen verwendet wird. Dieses Kapitel stellt das Verfahren oberflächlich vor. Es ist anzumerken, dass nur die generellen Konzepte genannt werden und auf eine genaue Beschreibung verzichtet wird. Der Grund hierfür ist, dass das Verfahren im weiteren Verlauf der Arbeit nicht verwendet wird. Für einen genauen Überblick ist auf entsprechende Fachliteratur verwiesen [13].

Ziel des Backpropagation Verfahrens ist die Optimierung der Verbindungsgewichte und Schwellwerte. Die Struktur des KNN wird nicht angepasst. Eine weiterer wichtiger Unterschied zu den später vorgestellten neuroevolutionären Algorithmen ist, dass bei der Trainingsphase nur ein KNN benötigt wird. Im Zusammenhang mit dem überwachtem Lernen wird über die Beispieldaten iteriert. Für die Eingabedaten berechnet das KNN die vermuteten Ausgabewerte. Danach wird der Fehler zwischen dem erhaltenen und erwarteten Ergebnis gebildet. In

Abhängigkeit der Größe des Fehlers, werden die Gewichte und Schwellwerte direkt angepasst, sodass der Fehler insgesamt geringer wird. Dieser Vorgang wird solange wiederholt, bis das KNN die gewünschte Genauigkeit bietet. Häufig ist dieses Verfahren beim finden einer Lösung sehr schnell. Jede Änderung ist gerechtfertigt und sollte die Leistung des KNN verbessern bzw. den Fehler verringern.

2.2 Evolutionäre Algorithmen

Für die Optimierung von KNN können verschiedene Algorithmen eingesetzt werden. Der in Kapitel (TODO KApitel) vorgestellte Backpropagation Algorithmus ist hierbei nur ein Beispiel. In dieser Arbeit wird ein Verfahren eingesetzt, welches in Kapitel 2.3 vorgestellt wird und zur Gruppe der neuroevolutionären Algorithmen gehört, welche auf den sogenannten Evolutionären Algorithmen (EA) basieren. Zu diesen gehören eine Vielzahl von unterschiedlichen Verfahren, welche dennoch einige gemeinsame Grundprinzipien haben. Ziel von diesen ist, eine möglichst gute Näherungslösung für ein Optimierungsproblem zu finden. Umgesetzt wird dies mit einer simulierten Evolution, welche durch das biolgische Pendant inspiriert ist [22].

2.2.1 Biologische Evolutionäre Konzepte

Einer der bedeutendsten Wissenschaftler in Bezug auf die Evolutionstheorie ist Charles Darwin, welcher 1859 mit seiner Arbeit *On the Origin of Species by Means of Natural Selection* einen wichtigen Grundbaustein gelegt hat [12]. Theoretisch wird bei Betrachtung der Evolution zwischen unbelebten Systemen und lebenden Organismen unterschieden [22]. Da die EA von Letzteren inspiriert sind, wird im weiteren Verlauf dieser Arbeit nur auf diese Bezug genommen.

Die später vorgestellten EA übernehmen aus der Biologie verschiedene Begriffe, wie zum Beispiel Population, Individuum, Genotyp, Phänotyp, Selektion, Rekombination und Mutation. Diese werden im Folgenden aus Sicht des biologischen Vorbilds betrachtet. Die Erklärungen in dieser Arbeit sind stark vereinfacht und es werden zudem nur die konzeptionellen Prinzipien betrachtet. Der genaue biologische Ablauf ist für diese Arbeit nicht relevant.
Eine Population setzt sich aus vielen unterschiedlichen und unabhängigen Individuen zusammen, welche alle zur selben Art gehören. Eine Art ist hierbei so definiert, dass die einzelnen Individuen einen gemeinsamen Genpool teilen und sich miteinander paaren können. Jedes Individuum besitzt ein Genom, welches

das genetische Erbgut enthält. Dieses besteht aus mehreren Chromosomen, die wiederum mehrere Gene besitzen [22]. Hierbei kann ein Gen, welches zum Beispiel für die Fell- bzw. Haarfarbe des Individuums verantwortlich ist, verschiedene Werte annehmen. Jede dieser Ausprägungen, in diesem Fall schwarze und braune Haare, werden als Allel bezeichnet [22]. Somit ist das Genom der Bauplan für ein Individuum und bestimmt maßgeblich dessen Erscheinungsbild [14]. Der Phänotyp wird durch das Genom beeinflusst und beschreibt die tatsächlichen, äußerlich feststellbaren Ausprägungen der einzelnen Gene [22]. Allerdings kann der Phänotyp auch durch die Umwelt beeinflusst werden [14]. Die Kombination aus Genom und Phänotyp bildet das bereits vorgestellte Individuum.

Nachdem im vorherigen Absatz die grundlegenden Begriffe bezüglich einzelner Individuen erläutert wurden, soll jetzt mit Bezug auf die Evolution die Population als Ganzes betrachtet werden. Die heute existierende Vielfalt an Tier- und Pflanzenarten hat sich über viele Millionen Jahren entwickelt. Der genaue Ursprung, wie die ersten Lebewesen mit Stoffwechselprozessen entstanden sind, ist dennoch unbekannt. Bezüglich der Evolution stellt sich die Frage, wie sich das genetische Material im Laufe der Zeit ändern kann. Hierfür sind fünf Faktoren zu nennen [22].

1. Der erste und wichtigste Faktor sind zufällige Mutationen. Hierbei entstehen beim Vervielfältigen des genetischen Erbguts, zum Beispiel bei der Fortpflanzung, Fehler gemacht, die zu zufälligen Änderungen führen. Hierdurch kann beispielsweise ein neues Allel entstehen, welches zu einer neuen, bisher nicht vorhandenen Haar- bzw. Fellfarbe führt [22].

2. Der zweite Faktor betrifft die Selektion. Damit verschiedene Allele langfristig ähnlich häufig in der Population vorkommen, müssen mehrere Faktoren zutreffen. Ein Faktor ist die Überlebenschance der verschiedenen Individuen in der Umwelt, die sogenannte Umweltselektion [22]. Eine auffällige Fellfarbe kann zum Beispiel ein Nachteil sein, da diese von den natürlichen Feinden leichter entdeckt wird. Da diese Individuen häufiger gefressen werden, haben sie eine geringere Chance zur Fortpflanzung und es somit möglich, dass das genetische Material verloren geht. Doch nicht nur die Umweltselektion hat einen Einfluss auf die Anzahl der Nachkommen. Hierfür sind ebenfalls die erfolgreiche Partnersuche sowie Fortpflanzungsrate verantwortlich [22].

3. Besonders in kleinen Populationen kann der Tod einzelner Individuen große Auswirkungen auf das Verhältnis der unterschiedlichen Allele haben. Hierbei

können durch Zufall einzelne Allele komplett verloren gehen und die nachfolgenden Generationen stark beeinflusst werden. In diesem Fall spricht man von Gendrift. Der Effekt hiervon ist bei größeren Populationen vernachlässigbar [22].

4. Wie bereits beschrieben, sollen sich Individuen einer Art fortpflanzen können. Doch es kommt auch vor, dass Individuen einer Art abwandern und sich an zwei räumlich getrennten Orten weiterentwickeln. Kommt es zu einem späteren Zeitpunkt wieder zu einer Zuwanderung können die neu entwickelten Gene die Population maßgeblich verändern. Dieser Effekt wird auch Genfluss genannt [22].

5. Der letzte Faktor ist die Rekombination. Bezüglich der biologischen Evolution beschreibt dies den Vorgang der sexuellen Paarung zweier Individuen, sodass ein oder mehrere Nachkommen erzeugt werden. Dabei wird das Ergbut für diese aus einer Kombination der Elterngenome erstellt. Somit handelt es sich aus Sicht der klassischen Evolutionslehre nicht um einen Evaluationsfaktor, da nur Bekanntes neu kombiniert wird und keine neuen Gene oder Allele entstehen. Dennoch wird die Rekombination heute meistens als Evaluationsfaktor genannt. Grund hierfür ist, dass die einzelnen Gene nicht wie lange in der Populationsgenetik angenommen komplett unabhängig voneinander sind, sondern stattdessen stark vernetzt sind und viel Einfluss aufeinander haben. So können auch bei der Kombination von bekannten Genotypen neue phänotypische Eigenschaften entstehen [22].

Durch die vorgestellten Arten der Evolution kann sich eine Population verschiedensten Umweltsituationen anpassen und gegenüber konkurrierenden Arten behaupten. Beispiel hierfür sind Bakterien, welche Resistenzen gegen bestimmte Antibiotika entwickeln. Während anfänglich nur wenige Individuen die Resistenz besitzen, wird diese aufgrund der hohe Verbreitung von Bakterien schnell an Nachkommen weitergegeben und ist nach kurzer Zeit in der ganzen Population vorhanden.

2.2.2 Evolutionäre Algorithmen

Im vorherigen Kapitel ist die biologische Evolution vorgestellt, durch die eine Vielzahl von unterschiedlichen Lebensformen entstanden ist, die sich sehr gut an ihre jeweilige Umwelt angepasst haben. Da dieses Vorgehen in der Biologie sehr erfolgreich ist, wurden schon im Jahr 1950 erste Versuche durchgeführt, dies auf Computersysteme zu übertragen. Hierbei wird eine bedeutend vereinfachte

künstliche Evolution simuliert mit dem Ziel, ein Optimierungsproblem zu lösen [22]. Heute gibt es eine Vielzahl von verschiedenen Algorithmen, die unterschiedliche Aspekte der Evolution imitieren. Im Folgenden werden die Grundkomponenten vorgestellt und verschiedene beispielhafte Umsetzungsmöglichkeiten aufgezeigt.

Genotyp und Phänotyp

Wie bei der biologischen Evolution gibt es bei den EA Individuen, welche durch ein Genom und einen Phänotyp definiert sind [22]. Das Genom enthält alle Informationen die nötig sind, um den Phänotypen des Individuums zu erstellen. Die eigentliche Form des Phänotyps ist abhängig vom gegebenen Optimierungsproblem und kann je nach Einsatzszenario unterschiedlich umgesetzt sein [23]. Die Repräsentation des Genoms ist in vielen klassischen Algorithmen binär. In diesen Fällen wird das Genom durch einen Vektor x von der Länge l repräsentiert, welcher nur aus den Werten 0 und 1 besteht, somit gilt $x = (x_1, x_2, ..., x_l) \in \{0, 1\}^l$ [23]. Allerdings kann diese Art der Kodierung nicht ausreichend sein. In diesen Fällen kann der Vektor auch natürliche, ganze oder rationale Zahlen enthalten [23]. Grundsätzlich sind diese Arten der Repräsentationen nur als Beispiele zu verstehen. Jeder Algorithmus kann die Repräsentation der Genome anpassen, sodass es für das Verfahren zuträglich ist. In Kapitel 2.3.1 wird die in dieser Arbeit verwendete Art der Kodierung vorgestellt.

Optimierungsproblem

Wie bereits beschrieben, ist es das Ziel von EA, Optimierungsprobleme zu lösen. Diese können aus vielen verschiednen Bereichen wie Forschung, Wirtschaft sowie Industrie kommen [22] und unterschiedliche Anforderungen haben. Grundsätzlich muss jedes Optimierungsproblem aus einem dreier Tupel $(\Omega, f, \succ)$ bestehen [22]. Die Variable Ω repräsentiert dabei den Suchraum, also jeden möglichen Lösungsansatz. Dieser wird typischerweise mit einem Individuum und dessen Genom bzw. Phänotyp getestet. Die Funktion f ist definiert als $f : \Omega \to \mathbb{R}$ und bewertet jeden Lösungsansatz aus dem Suchraum und weist diesem einen reellen Wert zu [22]. Dieser wird auch als Güte- bzw. Fitnesswert bezeichnet. Der letzte Teil des Optimierungsproblems ist eine Vergleichsrelation $\succ \in \{<, >\}$, welche angibt, ob es das Ziel, ist ein Minimum oder Maximum in der Fitnessfunktion zu finden [22]. Im Kontext von EA wird meistens das Maximum gesucht, so auch in dieser Arbeit. Daher wird im Weiteren stets angenommen, dass die Maximierung des erreichten Fitnesswertes das Ziel ist.
Bei allen Optimierungsproblemen ist die Fitnessfunktion ein elementarer Be-

standteil. Nur diese Funktion gibt dem Algorithmus ein Feedback, wie gut oder schlecht eine Lösung ist. Mithilfe dieser Funktion muss jeder EA ableiten, in welche Richtung eine Optimierung sich entwickeln soll, um möglichst effizient eine Lösung zu finden [22]. Aus diesem Grund ist die erste Anforderung an eine solche Funktion, dass sie keine absolute sondern eine graduelle Bewertung der verschiedenen Lösungsansätze bietet [22]. Beispiel für eine absolute Bewertung ist, wenn die Fitnessfunktion für eine Lösung den Wert 1 liefert, wenn das Optimierungsproblem gelöst ist und andernfalls den Wert 0. In diesem Fall kann nicht festgestellt werden, welche Änderungen der Suchparameter erfolgversprechend sind, somit ist es auch nicht möglich, diese gezielt zu ändern. Infolgedessen müssen mehr Lösungsansätze aus dem Suchraum getestet werden, was den Rechenaufwand und die benötigte Zeit erhöht. Des weiteren muss die Fitnessfunktion möglichst vollständig die Ziele des Optimierungsproblems abbilden. Andernfalls kann zwar durch den Algorithmus das Ergebnis der Fitnessfunktion maximiert werden, aber die hierdurch gefundene Lösung enthält nicht die vom Anwender gewünschten Eigenschaften [22].

Ablauf evolutionärer Algorithmen

Aus den vorherigen Kapiteln ist ersichtlich, dass Individuen aus einem Genotyp sowie Phänotyp bestehen und dass diese versuchen, ein Optimierungsproblem zu lösen. Die Aufgabe eines evolutionären Algorithmus ist es, die Individuen langfristig so anzupassen, dass sie bessere Fitnesswerte in dem Optimierungsproblem erzielen und dementsprechend eine gute Lösung finden. Hierzu werden die aus der Biologie bekannten Verfahren Selektion, Rekombination und Mutation eingesetzt. Bevor in den weiteren Kapiteln verschiedene Umsetzungen beispielhaft vorgestellt werden, wird in diesem Abschnitt der grundlegende Ablauf von EA erläutert.

Abbildung (TODO ABBILDUNG) zeigt den beispielhaften Ablauf von EA, wobei die Phasen Evaluation, Selektion, Mutation und Rekombination die größte Bedeutung haben. Bevor der eigentliche Programmablauf starten kann, muss eine erste initiale Population erzeugt werden. Wie in der biologischen Evolution, besteht diese auch in diesem Fall aus mehreren unabhängigen Individuen [23]. Im Gegensatz zum biologischen Vorbild verwenden die meisten Algorithmen ein feste Populationsgröße, da ansonsten die später benötigte Evaluationszeit und der damit verbundene Rechenaufwand stark ansteigen könnte [23]. Die für die Individuen benötigten Genome werden zufällig erstellt [22], wobei je nach Algorithmus verschiedene Zufallsverteilungen genutzt werden können.

Danach beginnt die Evaluationsphase mit der initialen Population [23]. Hierfür wird der Phänotyp für jedes Individuum mit dem entsprechenden Genom gebildet. Jeder von diesen stellt eine mögliche Lösung für das gegebene Optimierungsproblem dar. Wie im vorherigen Kapitel beschrieben, muss dieses eine Fitnessfunktion enthalten, mit welcher jeder Phänotyp bewertet wird. An dieser Stelle soll nochmals hervorgehoben werden, dass die Gesamtheit aller Gene den Phänotyp bestimmen und daher keine Bewertung der einzelnen Gene möglich ist [23]. Die Evaluationsphase endet, wenn für alle Phänotypen ein Fitnesswert ermittelt ist. Der nächste Schritt ist die Überprüfung einer Abbruchbedingung. Trifft diese zu, wird die Ausführung des Algorithmus abgebrochen und das Genome des besten Individuums als Ergebnis zurückgegeben [22]. Je nach Umsetzung der Abbruchbedingung kann zum Beispiel überprüft werden, ob ein gewisser Fitnesswert überschritten und somit eine Lösung mit der gewünschten Genauigkeit bzw. Korrektheit gefunden wurde oder ob eine vorher definierte maximale Ausführungszeit überschritten ist.

Die Abbruchbedingung wird zu Beginn mit sehr hoher Wehrscheinlicht nicht erfüllt sein, da die Genome nur zufällig erstellt sind und bisher kein Lernprozess durchgeführt wurde. In diesem Fall werden die Phasen Selektion, Rekombination und Mutation durchgeführt [23]. Diese werden in den folgenden Kapiteln ausführlich erläutert, daher wird in diesem Abschnitt nur ein kurzer Überblick gegeben. In der ersten Phase, der Selektion, wird auf Basis des erhaltenen Fitnesswertes für jedes Individuum festgelegt, ob und wenn ja wie viele Nachkommen dieses erzeugen darf [22]. Bei der Rekombination werden die tatsächlichen Nachkommen erzeugt. Typischerweise werden zwei, in machen Fällen auch mehr Individuen als Elterngenome ausgewählt und gekreuzt. Bei diesem Vorgang wird das genetische Material, welches in den Genomen der Eltern-Individuen enthalten ist, gemischt und an das neu erstellte Kind-Individuum übertragen. Das Ziel dieser Operation ist, dass das Kind immer einen Teil der Gene von beiden Eltern erhält und somit auch Eigenschaft von beiden vereint. Langfristig sollen sich durch ein solches Verfahren nur die besten Gene durchsetzen [22]. Die letzte Phase ist die Mutation. In diesem Schritt besteht für jedes neu erstellte Individuum die Wahrscheinlichkeit, dass ein kleiner Teil des Genoms zufällig abgeändert wird [24]. Die Art der Mutation ist hierbei abhängig von der Umsetzung des Genotypen und dem Algorithmus. Bezüglich der drei Phasen muss verdeutlicht werden, dass die Selektion auf Basis des Phänotypen mit dem Fitnesswert erfolgt, die Rekombination

und Mutation hingegen auf Basis des Genotypen. Somit können keine Eigenschaften, die im Phänotyp gespeichert sind auf die Nachkommen übertragen werden [23].

Nach Abschluss dieser drei Phasen sind die neuen Individuen fertig erstellt. Da, wie bereits in diesem Kapitel beschrieben, die Populationsgröße meistens begrenzt ist, wird an dieser Stelle typischerweise die Elterngeneration komplett entfernt und durch dieselbe Anzahl an Nachkommen ersetzt [22]. Allerdings gibt es auch andere Ansätze, bei denen nicht so viele neue Individuen gleichzeitig erstellt werden und diese dann direkt in die bestehende Population integriert werden können [25]. Die neue Population mit den neuen Individuen durchläuft dieselben Schritte wie die vorherige Population. Ein kompletter Durchlauf des vorgestellten Zyklus wird als eine Generation bezeichnet [22]. Häufig wird die neu erstellte Population daher auch als neue Generation bezeichnet.

Selektion

Bei EA werden viele Individuen eingesetzt, um verschiedene Lösungsansätze gleichzeitig zu betrachten. In der Phase der Selektion wird bestimmt, welche Individuen als Elternteil für die nächste Generation ausgewählt werden und wie viele Nachkommen ihnen zustehen. Bei einem solchen Auswahlverfahren ist zwischen zwei grundlegenden Umsetzungen zu unterscheiden. Entweder kann allen Individuen einer Generation dieselbe Menge an Nachkommen zugewiesen werden oder die Anzahl ist abhängig von dem erreichten Fitnesswert. Typischerweise wird die zweite Variante in EA verwendet, welche auch als fitnessproportionale Selektion bezeichnet wird. Die erste Variante erzeugt keinen Selektionsdruck, da die Individuen unabhängig von ihrer Leistung Nachkommen zugewiesen bekommen. Bei der zweiten Variante werden Individuen mit höheren Fitnesswerten bevorzugt, mit dem Ziel, dass sich die positiven Eigenschaften der erfolgreichen Individuen durchsetzen und schlechte aussterben. Dennoch ist es nicht das Ziel, nur die allerbesten Genome als Elternteile auszuwählen. Wäre dies der Fall, würde die Population sehr schnell konvergieren, ihre Vielfalt verlieren und nur noch ähnliche Lösungsansätze bieten. Somit wird es unwahrscheinlich, dass neue unbekannte, aber eventuell bessere Lösungsstrategien gefunden werden [22].

Der genaue Selektionsvorgang, wie er mi Algorithmus dieser Arbeit umgesetzt ist, wird in Kapitel 2.3.4 erläutert. Im Folgenden werden zwei verschiedene Varianten der fitnessproportionalen Verfahren vorgestellt, die in anderen Algorithmen als Selektionsfunktion verwendet werden. Bei diesen wird die Anzahl an Nachkommen

durch den jeweils erreichten Fitnesswert beeinflusst [22].

1. **Probabilistische Selektion**:
   Einer der bekanntesten Umsetzungen ist die probabilistische Selektion, welche grundsätzlich einfach zu implementieren ist. Im ersten Schritt werden die erreichten Fitnesswerte f der einzelnen Individuen i in der Population P aufsummiert. Im zweiten Schritt wird für jedes Individuum j die Wahrscheinlichkeit $Pr[j]$ berechnet, welche angibt, wie groß die Chance ist, dass dieses als Elternteil ausgewählt wird. Hierzu muss der erhaltene Fitnesswert durch die bereits berechnete Summe geteilt werden. Somit ergibt sich die Formel $Pr[j] = \frac{f_j}{\sum_{i \in P} f_i}$. Diese Art der Selektion ist in vielen Anwendungsfällen sehr erfolgreich. Allerdings kann es vorkommen, dass zum Beispiel bei sehr hohen Fitnesswerten die prozentualen Unterschiede zwischen guten und schlechten Individuen sehr gering sind und infolgedessen der Selektionsvorteil für gute Lösungen niedrig ist. Ein möglicher Lösungsansatz besteht in der Skalierung der Fitnesswerte, sodass auch bei hohen Durchschnittswerten kleine Steigerungen einen evolutionären Vorteil bieten [22]. Ein weiterer Lösungsansatz ist die Verwendung der rangbasierten Selektion, welche im Folgenden vorgestellt wird.

2. **Rangbasierte Selektion**:
   Bei der rangbasierten Selektion ist der tatsächlich erhaltene Fitnesswert nicht von Bedeutung. Die Individuen werden bezüglich ihrer Fitness geordnet. Das beste Individuum erhält die größte und das schlechteste Individuum die geringste Wahrscheinlichkeit, als Elternteil ausgewählt zu werden [22].

Sind die Wahrscheinlichkeiten für alle Individuen berechnet, ist im letzten Schritt festzulegen, welche Genome tatsächlich als Elternteile ausgewählt werden und wie viele Nachkommen diese erzeugen. Bei der fitnessproportionalen Selektion wird hierfür ein Zufallsgenerator benötigt, welcher basierend auf den Wahrscheinlichkeiten die Elterngenome auswählt. Dies wird häufig mit einem Roulette-Rad veranschaulicht. Die Felder am Rand der Scheibe entsprechen den verschiedenen Individuen und die Größe ist proportional zur berechneten Wahrscheinlichkeit. Für jedes auszuwählende Elternindividuum wird der Zeiger zufällig gedreht und das Individuum entsprechend zu dem gewählten Feld wird verwendet (TODO ABBILDUNG). Dieses Verfahren kann einen Nachteil haben. Obwohl das beste Individuum die höchste Wahrscheinlichkeit hat, als Elternteil ausgewählt zu werden, kann es auf Basis des Zufalls dazu kommen, dass dieses nicht verwendet wird. Da typischerweise die Population am Ende des Evolutionszyklus ersetzt wird, würde

das genetische Material dieses Individuums verloren gehen [22].

Um einen solchen Verlust zu verhindern, kann bei der Selektion zusätzlich ein sogenannter Elitismus verwendet werden. Hierbei wird typischerweise der Genotyp des besten Individuums ausgewählt, kopiert und unverändert wieder in die nächste Generation eingesetzt [3]. Zu Beachten ist hierbei, dass im weiteren Verlauf ein Nachkomme weniger produziert wird um eine konstante Populationsgröße zu garantieren.

Rekombination

Die Phase der Rekombination wird auch als *Crossover* bezeichnet und findet nach der Selektion statt. Die Aufgabe von dieser ist, die ausgewählten Elternindividuen zu nutzen um neue Nachkommen zu erstellen. Typischerweise werden zwei, in einigen Fällen noch mehr Elternteile kombiniert um mindestens ein Kind-Individuum zu erzeugen [22]. Die Rekombination gilt als eine der wichtigsten Phasen, da die Nachkommen tendenziell bessere Ergebnisse erzielen sollen als die Elternteile [12]. Wie auch bei anderen Phasen der EA gibt es verschiedene Arten der Umsetzung, die als kombinierende, interpolierende und extrapolierende Selektion bezeichnet werden [22]. Die bekannteste dieser drei Varianten ist die kombinierende Selektion, welche sowohl in dieser Arbeit als auch in vielen anderen Algorithmen verwendet wird. Die beiden Alternativen werden bedeutend seltener gewählt und sind auch in der Literatur oft nicht erwähnt. Dennoch werden im Folgenden alle Varianten kurz vorgestellt.

Die kombinierende Rekombination ist stark von der Biologie inspiriert. Bei diesem Vorgang werden die Genome der Elternteile zuerst nebeneinander aufgereiht. Im zweiten Schritt wird zufällig entschieden, welcher Abschnitt von welchem Elternteil für das Genom des Nachkommen verwendet werden sollen [22]. Der Vorteil dieses Verfahrens ist, dass große Informationsblöcke, welche unabhängig voneinander optimiert wurden und sinnvolle Funktionen realisieren, von den Elternteilen geerbt werden können [12]. Der aus der Rekombination entstehenden Nachkomme kann hierdurch einen Vorteil bei der Evaluation besitzen und letztendlich eine bessere Lösung für das Optimierungsproblem bieten. Da diese Art der Rekombination keine neuen Gene erstellt bzw. bestehende modifiziert, ist der Erfolg abhängig davon, ob die Population eine gewisse Diversität bietet, sodass bei der Rekombination tatsächlich verschiedene Gene kombiniert werden können [22]. In der praktischen Umsetzung muss zuletzt noch entschieden werden, an welchen Stellen

eine Rekombination möglich ist [24]. Bei der uniformen Rekombination wird für jedes einzelne Gen unabhängig zufällig entschieden, von welchem Elternteil es übernommen wird [22]. Allerdings gibt es auch andere Umsetzungen, bei denen die Gene in Gruppen eingeteilt werden und dann zwischen diesen zufällig entschieden wird.

Bei der interpolierenden Rekombination, werden die einzelnen Gene nicht direkt von einem Elternteil übernommen, stattdessen werden sie gemischt, sodass ein neuer Wert entsteht, der sich zwischen den Werten der Elternteile befindet. Im Gegensatz zur kombinierenden Selektion, welche versucht die Diversität zu erhalten, wird diese hierbei deutlich verringert. Aus diesem Grund ist es notwendig, dass zu Beginn eine Population mit einer großen Diversität vorhanden ist, sodass der Suchraum des Optimierungsproblems ausgiebig überprüft wird. Ein Beispiel für eine solche technische Umsetzung ist die sogenannte Arithmetische-Rekombination, welche für reellwertig repräsentierte Genome verwendet werden kann. Angenommen die Werte A_i und B_i repräsentieren die Gene der Eltern A und B, dann wird für jedes Gen i eine Zufallszahl u zwischen 0 und 1 gewürfelt. Das Gen C_i des Nachkommen C wird dann berechnet mit $C_i = u \cdot A_i + (1 - u) \cdot B_i$ [22].

Die letzte Variante ist die extrapolierende Rekombination, welche mit mehreren Elternteilen versucht, eine Prognose darüber abzugeben, wo im Lösungsraum eine Steigerung des Fitnesswertes möglich ist und dementsprechend versucht die Genome der Nachkommen zu ändern. Somit kann diese Art der Rekombination auch neue Gene erstellen. Ein solches Verfahren hat allerdings zwei Nachteile. Um eine Prognose abzugeben, ist es erstens nötig ein Grundwissen über den Lösungsraum zu haben, zweitens besteht die Gefahr, dass die vorgegebene Richtung nicht korrekt und die Funktion danach nicht fähig ist, eine systematische Suche durchzuführen [22].

Mutation

Der letzte Schritt, bevor die bestehende Population durch die neu erstellten Individuen ersetzt wird, ist typischerweise die Mutation. Allerdings ist die Funktionsweise und Relevanz dieser Phase stark abhängig von der verwendeten Kodierung und dem eigentlichen Algorithmus [22]. Zum Beispiel wird in Quelle [3] gar keine Reproduktion verwendet und die Optimierung wird nur mithilfe der Mutation umgesetzt, während in anderen Quellen, wie zum Beispiel in [26], diese Phase als

untergeordnet beschrieben wird, welche nur selten eingesetzt werden sollte.

Wird die Mutation häufig verwendet, erfüllt sie zwei Aufgaben. Das erste Ziel ist die Feinabstimmung der einzelnen Individuen, sodass das tatsächliche Optimum so genau wie möglich erreicht wird. Das zweite Ziel ist die Erforschung des Suchraums, welche stichprobenartig durchgeführt wird, um ein besseres Optimum zu finden [22]. Wird die Mutation selten eingesetzt, müssen die Funktionen Feinabstimmung und Erforschung durch andere Komponenten, wie zum Beispiel der Rekombination, durchgeführt werden. In diesem Fall ist es das Ziel der Mutation neue Diversität, in die Population zu bringen, beziehungsweise diese zu erhalten [22]. Denn wie bereits im vorherigen Kapitel beschrieben, ist vor allem bei der viel verwendeten Rekombination die Diversität sehr wichtig, allerdings wird diese meistens durch die Rekombination selbst verringert.

Da die Umsetzung der Mutation sowohl von der Kodierung und dem Algorithmus abhängig ist, gibt es keine Empfehlungen, welche Implementierung besonders viele Vorteile bietet. Im Folgenden werden zwei mögliche Beispiele vorgestellt, wobei das erste für Individuen mit einer binären und das zweite für eine reellwertige Repräsentation angewendet werden kann [22]. Die einfachste Mutation ist die Binär-Mutation, welche für Genome mit einer binären Repräsentation verwendet werden kann. Bei der Mutation wird für jedes Bit eine Zufallszahl u zwischen 0 und 1 gewürfelt. Ist diese kleiner als die festgelegte Mutationswahrscheinlichkeit p_m, wird das Bit invertiert. Sind die Individuen durch reellwertige Zahlen repräsentiert, ist die sogenannte Gauss-Mutation eine mögliche Umsetzung. Bei dieser wird für jedes Gen eine Zufallszahl basierend auf einer Gauss-Verteilung gewählt, wobei eine zuvor festgelegte Standardabweichung σ die Verteilung beeinflusst. Die hierdurch erhaltene Zahl wird auf den bereits bestehenden Wert addiert [22].

2.2.3 Neuroevolution

Wie im vorherigen Kapitel erläutert, sind EA Verfahren, um möglichst gute Näherungslösungen für verschiedene Optimierungsprobleme zu finden. Für die in Kapitel 2.1 vorgestellten KNN wird ein Verfahren benötigt, welches die verschiedenen anpassbaren Parameter optimiert. Es ist dementsprechend möglich, die evolutionären Prinzipien zur Optimierung von KNN einzusetzen. Dies wird als Neuroevolution bezeichnet [27]. Algorithmen dieser Art sind somit eine Alternative zu den klassischen Verfahren, wie zum Beispiel dem Backpropagation Algorithmus [2].

Im Vergleich zu diesem haben neuroevolutionäre Verfahren sowohl Vor- als auch Nachteile, welche ausführlich in Kapitel (TODO Kapite) beschrieben werden. Dennoch soll ein großer Vorteil bereits hier genannt sein. Das Ziel der meisten neuroevolutionären Algorithmen ist das Optimieren von Verbindungsgewichten und Schwellwerten. Einige Verfahren versuchen, zusätzlich die Struktur bzw. Topologie des KNN zu optimieren, sodass diese nicht mehr manuell durch einen Entwickler festgelegt werden muss. Wie in Kapitel 2.1.4 beschrieben, ist die Topologie ein entscheidender Faktor und kann das erfolgreiche Lösen maßgeblich beeinflussen. Diese Algorithmen werden als *Topology and Weight Evolving Artificial Neural Network* (TWEANN) Verfahren bezeichnet [28].

Auch der in Kapitel 2.3 vorgestellte Algorithmus optimiert sowohl die Topologie als auch die Gewichte eines KNN. Im Folgenden wird der Ablauf von neuroevolutionären Algorithmen vorgestellt und die Besonderheiten erläutert. Danach wird auf die Vor- und Nachteile eingegangen.

Ablauf Neuroevolution

Der grundsätzliche Ablauf von neuroevolutionären Algorithmen ist fast identisch zu den klassischen EA. Auch bei diesen gibt es eine Population, welche aus verschiedenen Individuen besteht, die einen Genotyp und Phänotyp besitzen. Letzteres besteht bei der Neuroevolution aus einem KNN, welches durch den Genotyp kodiert ist. Auch die Definition des Optimierungsproblems ist identisch zu der Erklärung in Kapitel 2.2.2. Ein Phänotyp, in diesem Fall ein KNN, versucht, das Optimierungsproblem zu lösen und erhält hierdurch einen Fitnesswert, welcher angibt, wie gut oder schlecht die Lösung ist. Wenn dieser Wert für alle Mitglieder einer Population berechnet ist, beginnen die Phasen Selektion, Rekombination und Mutation. Hierdurch werden neue Individuen mit neuen Genotypen und Phänotypen erstellt, welche dann die vorherige Population ersetzen. Dieser Zyklus wird so oft wiederholt, bis eine Abbruchbedingung erreicht ist. Natürlich ist es möglich, dass die praktische Umsetzung je nach Algorithmus angepasst wird. Einige grundlegende Anpassungen werden in den folgenden Abschnitten erläutert. Wie die angesprochenen Punkte von dem in dieser Arbeit verwendeten Algorithmus umgesetzt werden, wird in Kapitel 2.3 erläutert.

Genotyp und Phänotyp

Eine Komponente, die auf jeden Fall angepasst werden muss, ist der Genotyp. Dieser muss alle optimierbaren Parameter für den Phänotyp kodieren. Typischer-

weise umfasst dies die Struktur des Netzes, die Gewichte der Verbindungen sowie die Schwellwerte [28]. Die Propagierungs-, Aktivierungs- und Ausgabefunktion müssen nur enthalten sein, wenn diese ebenfalls durch den EA angepasst werden. Hierdurch ergibt sich die Frage, wie diese Informationen im Genom zu Kodieren sind, sodass auch eine erfolgreiche Durchführung der Rekombinations- und Mutationsphase möglich ist.

Grundsätzlich gibt es zwei verschiedene Kodierungansätze, die ein Algorithmus verwenden kann. Diese werden als direkte und indirekte Kodierung bezeichnet. Bei einer direkten Kodierung wird im Genom jede einzelne Verbindung und jedes Neuron explizit spezifiziert, sodass diese einfach im Phänotyp übernommen werden können. Diese Art der Kodierung wird sehr häufig verwendet, da sie einfach zu implementieren ist und auch die Rekombination und Mutation damit gut umsetzbar sind. Die Alternative ist die indirekte Kodierung. Diese spezifiziert Regeln, die angeben, wie aus einem Genom ein KNN erstellt werden soll. Der Vorteil hiervon ist, dass nicht jede Verbindung und jedes Neuron einzeln kodiert werden muss, die Repräsentation dementsprechend kompakter ist und somit weniger Rechenkapazität zur Speicherung benötigt wird [28]. Nachteil von dieser Methode ist, dass die Rekombination und Mutation komplexer umzusetzen sind. Im Folgenden wird nur noch auf die direkte Kodierung Bezug genommen, da diese in dieser Arbeit verwendet wird.

Theoretisch kann für eine direkte Kodierung des Genoms, wie bei klassischen EA, eine binäre Repräsentation verwendet werden. Umsetzbar ist dies mit einer Matrix, welche für jede mögliche Verbindung angibt, ob diese besteht oder nicht. Allerdings hat eine solche Kodierung einige Nachteile, weswegen sie für diesen Anwendungsfall eher ungeeignet ist. Ein Grund ist der benötigte Speicherplatz für ein einzelnes Genom. Die Matrix enthält für ein KNN mit n Neuronen insgesamt n^2 Einträge für die möglichen Verbindungen [28]. Für große neuronale Netze skaliert dieser Ansatz schlecht.

Alternativ ist hierzu eine Graphen-Kodierung, welche auch von vielen TWEANN Algorithmen verwendet wird. Eine mögliche Umsetzung hiervon ist in der Arbeit [29] von Pujol und Poli vorgestellt. Bei diesen besteht die Kodierung aus zwei Teilen. Der erste Teil beschreibt die Struktur des Graphen beziehungsweise des KNN, während der zweite Teil ein linearer Vektor ist, welcher die Neuronen und Verbindungen enthält. In dieser Arbeit wird eine weitere Variante verwendet,

welche ausführlich in Kapitel 2.3.1 erläutert wird.

Rekombination

Chronologisch gesehen findet die Selektion vor der Rekombination statt. Da sich diese Phase nicht ändert, wird im Folgenden mit der Rekombination fortgefahren. Prinzipiell ist das Verfahren dasselbe wie in Kapitel 2.2.2 beschrieben. Die Selektion hat zwei Elterngenome ausgewählt und diese werden im Rahmen der Rekombination zu einem neuen Genom kombiniert. Typischerweise wird bei diesem Vorgang zufällig entschieden, welche Verbindung von welchem Elternteil übernommen werden soll. Das Ziel ist, ein KNN zu erzeugen, welches in den meisten Fällen die positiven Eigenschaften der Eltern erbt und somit insgesamt besser wird. Eine Schwierigkeit, die hierbei im Bezug zu Neuroevolution entstehen kann wird als das *Competing Conventions* Problem bezeichnet [28].

Der Begriff *Competing Conventions* beschreibt ein Phänomen, bei dem mehrere KNN dieselbe Lösung für ein Optimierungsproblem bieten, aber sich die Repräsentationen der Genome dennoch unterscheiden. Ein solches Beispiel ist in Abbildung (TODO Abbildung) dargestellt. Die beiden KNN besitzen je drei verdeckte Neuronen (4, 5, 6) mit den dazugehörigen Verbindungen in den Farben blau (B), rot (R) und grün (G). Die tatsächlichen Gewichte sind hierbei nicht von Interesse und werden deshalb für eine bessere Übersichtlichkeit durch die Farben repräsentiert. Das erste KNN kann beispielsweise durch den Vektor (B, R, G) und das zweite KNN durch den Vektor (G, R, B) kodiert werden. Es wird hierbei angenommen, dass in der Kodierung die einzelnen Gewichte für alle Verbindungen enthalten sind.

Wie aus der Abbildung zu erkennen ist, sind die beiden KNN symmetrisch und produzieren infolgedessen für dieselbe Eingabe auch dasselbe Ergebnis. Dennoch unterscheiden sich die Genotypen bezüglich ihrer Kodierung. Werden diese beiden KNN durch die Selektion ausgewählt um einen Nachkommen zu erzeugen, wird das *Competing Conventions* Phänomen wahrscheinlich zu einem Problem führen.

Wie bei traditionellen EA wird auch bei der Neuroevolution typischerweise zufällig entschieden, von welchem Elternteil welches Gen übernommen werden soll. Angenommen, es werden die ersten beiden Gene vom linken Elternteil und das letzte Gen vom rechten Elternteil übernommen, dann ist das daraus resultierende neue Genom kodiert mit (B, R, B). Es ist ersichtlich, dass bei diesem Vorgang ein Drittel der Informationen verloren gegangen sind beziehungsweise nicht vererbt

wurden. Dies ist in den meisten Fällen ein großes Problem, da der fehlende Teil mit hoher Wahrscheinlichkeit einen notwendigen Beitrag für einen Lösungsansatz kodiert hat, der das Elterngenom erfolgreich gemacht hat. Aus diesem Grund wird das neu erstellte Individuum wahrscheinlich weniger erfolgreich sein als seine Elternindividuen, erhält einen dementsprechend niedrigeren Fitnesswert und wird letztendlich aussterben [28]. Tritt dieser Fall nur bei einem einzelnen Genom in einer Population auf, sind die Auswirkungen meistens vernachlässigbar. Allerdings gibt es für ein KNN mit x verdeckten Neuronen in einer Schicht $x!$ viele Kombinationsmöglichkeit, dieselbe Lösung durch unterschiedliche Genome zu repräsentieren [28]. Somit ist die Wahrscheinlichkeit relativ hoch, dass das *Competing Conventions* Problem eintritt. Die Folge ist, dass entweder viel Rechenzeit für Genome verwendet wird, die mit hoher Wahrscheinlichkeit keine besseren Lösungen bereit stellen und dadurch das Optimierungsverfahren bedeutend länger dauert, oder dass das Verfahren scheitert, da keine ausreichend gute Lösung gefunden werden kann.

Bei den TWEANN Algorithmen ist das *Competing Conventions* Problem noch größer [28]. Wie bereits beschrieben, versuchen diese, auch die Struktur des KNN zu optimieren und können im Zuge dessen neue Neuronen und Verbindungen hinzufügen. Im nächsten Kapitel wird hierauf genauer eingegangen. An dieser Stelle sei vorweggenommen, dass im Laufe des Optimierungsverfahrens eine Population mit unterschiedlichen Strukturen und Topologien entstehen kann. Hierbei stellt sich die Frage, wie unterschiedliche KNN rekombiniert werden können, wenn die Genome unterschiedlich groß sind und nicht zuzuordnen ist, welche Gene der Eltern rekombiniert werden können. Dies ist die schwierigste Form des *Competing Conventions* Problems [28]. Wie der in dieser Arbeit verwendete Algorithmus dieses Problem löst, ist in Kapitel 2.3 erläutert.

Mutation

Neuroevolutionäre Algorithmen setzen in der Regel häufig die Mutation ein. Hierbei wird zwischen zwei Arten unterschieden. Bei der ersten Art werden nur die Verbindungsgewichte und Schwellwerte mutiert. Hierfür kann die bereits in Kapitel 2.2.2 vorgestellte Gauss-Mutation verwendet werden, wobei der erhaltene Zufallswert auf das Gewicht bzw. den Schwellwert addiert wird [30]. Die zweite Art verändert die Topologie und wird als strukturelle Mutation bezeichnet. Diese Art wird somit nur für die TWEANN Verfahren benötigt, welche neue Neuronen und Verbindungen dem KNN hinzuzufügen können [28].

Das eigentliche Hinzufügen von neuen Strukturen ist technisch einfach umzusetzen. Zum Beispiel kann eine neue Verbindung mit einem zufälligen Gewicht jederzeit zwei zuvor nicht verbundene Neuronen verknüpfen. Allerdings wird hierdurch der Fitnesswert in den meisten Fällen initial sinken, da es unwahrscheinlich ist, dass eine zufällige Struktur eine nützliche Funktionalität oder Teillösung bietet. Infolgedessen sinkt auch die Chance, dass das Genom bei der Selektion in der nächsten Generation als Elternteil ausgewählt und die neue Struktur an die Nachkommen vererbt wird. So kommt es häufig vor, dass die strukturellen Innovationen direkt in der nächsten Generation verloren gehen, auch wenn sie für eine erfolgreiche Lösung nötig wären. Die daraus resultierende Herausforderung für TWEANN Verfahren ist, neue Innovationen am Anfang zu schützen, sodass der Algorithmus diese optimieren kann. Nur so ist es möglich, neue Strukturen langfristig in die Population zu integrieren [28].

Eine mögliche Lösung hierfür ist in der Arbeit von Angeline, Saunders und Pollack vorgestellt. Hierbei werden neue Neuronen ohne jegliche Verbindungen in das KNN eingefügt, während neue Verbindungen initial das Gewicht 0 haben. Die so hinzugefügten Strukturen haben anfänglich keinen Effekt auf das KNN. Ziel hiervon ist, dass der Fitnesswert nicht absinkt und sich die Strukturen im Laufe von mehreren Generationen selbstständig entwickeln [31]. Diese Umsetzung ist nicht ideal, da die Strukturen unter Umständen nicht richtig in das KNN integriert werden, aber dennoch die benötigte Rechenkapazität erhöhen [28].

Der in Kapitel 2.3 vorgestellte Algorithmus verfolgt einen anderen Ansatz, bei dem ähnliche KNN einer Spezies zugeordnet werden. Ein Individuum soll nicht mehr mit der ganzen Population konkurrieren, sondern nur mit denen seiner Spezies. Dies wird auch als *niching* bezeichnet. Eine neue große strukturelle Mutation wird einer neuen Spezies zugeordnet und kann innerhalb dieser entwickelt und optimiert werden. Hierfür wird eine Kompatibilitätsfunktion benötigt, welche bestimmen kann, ob ein Genom einer Spezies zugehörig ist. Die Umsetzung von dieser Funktion wird durch das *Competing Conventions* Problem erschwert [28]. In Kapitel 2.3.4 wird hierauf genauer eingegangen.

Initiale Population

Die initiale Population der ersten Generation enthält zufällig erstellte KNN. Die hierfür benötigten Verbindungsgewichte und Schwellwerte werden häufig zufällig

mithilfe einer Gauss-Verteilung gewählt [30]. Die Wahl der initialen Topologie kann sich aber stark unterscheiden. Bei den neuroevolutionären Algorithmen, welche nur die Verbindungsgewichte und Schwellwerte optimieren, wird diese zu Beginn einmalig für alle KNN festgelegt und ändert sich auch nicht im Lauf des Verfahrens.

Bei den TWEANN Algorithmen hingegen können verschiedene initiale Topologien verwendet werden, welche unter anderem die Laufzeit des Algorithmus stark beeinflussen können. Viele dieser Algorithmen erstellen zu Beginn verschiedene zufällig gewählte Topologien, mit dem Ziel, Diversität in die Population zu bringen. Ein Nachteil dieser Strategie ist, dass hierbei ungeeignete KNN entstehen können, bei denen beispielsweise nicht alle Eingabeneuronen mit allen Ausgabeneuronen verbunden sind. Zudem ergibt sich in diesem Fall noch ein weiteres Problem [28]. Grundsätzlich ist es sinnvoll, die kleinst mögliche Struktur zu wählen, die dennoch einen mit größeren KNN vergleichbaren Fitnesswert für ein Optimierungsproblem erzielt [32]. Diese hat im Vergleich zu größeren Strukturen weniger anpassbare Parameter, was generell die benötigte Optimierungszeit reduziert. Wenn die Topologien zufällig erstellt sind, werden hierbei viele unnötige Strukturen enthalten sein, was zu einem größeren Suchraum führt und die Laufzeit des Optimierungsverfahrens erhöht [28].

In einem solchen Fall muss das Optimierungsverfahren versuchen, die Größe des KNN aktiv zu minimieren. Ein Ansatz diesbezüglich ist in der Arbeit von Zhang und Mühlenbein beschrieben, in welcher die Größe des KNN den Fitnesswert beeinflusst. In diesem Fall haben kleinere KNN bei gleicher Performance hinsichtlich des Optimierungsproblems einen größeren Fitnesswert und dementsprechend eine größere Chance, bei der Selektion ausgewählt zu werden. Das endgültige Ziel hierbei ist, dass langfristig nur die benötigten Strukturen erhalten bleiben [32]. Auch wenn die zugrunde liegende Idee gut ist, kann in der Praxis die Entscheidung schwierig sein, wie viel Einfluss die Größe des KNN tatsächlich auf den Fitnesswert haben soll [28].

Eine weiterer Lösungsansatz ist das Starten mit einer minimalen Struktur, sodass das KNN zu Beginn keine verdeckten Neuronen besitzt. Neue Strukturen werden nur integriert, wenn sie einen höheren Fitnesswert ermöglichen. Der Vorteil dieser Strategie ist, dass keine Rechenzeit verwendet werden muss, um unnötige Strukturen zu entfernen und dass zusätzlich die Anzahl an anzupassenden Parametern gering ist [28].

2.2.4 Neuroevolution im Vergleich

Grundsätzlich wächst das Interesse an neuroevolutionären Algorithmen und in vielen Anwendungsfällen sind bereits auch im Vergleich mit anderen Ansätzen gute Ergebnisse erzielt worden [27]. Dennoch werden Algorithmen dieser Art oft mit Skepsis betrachtet [27] und weniger eingesetzt als gradientenbasierte Verfahren beispielsweise der Backpropagation Algorithmus. Daher wird in vielen Arbeiten, wie zum Beispiel in [24, 27, 3, 2], ein Vergleich zwischen auf Gradienten basierenden Verfahren und neuroevolutionären Algorithmen gezogen. In diesem Kapitel werden einige Vor- und Nachteile vorgestellt, die häufig in diesem Zusammenhang genannt werden.

Für einen aussagekräftigen Vergleich müssen neuroevolutionäre Algorithmen zuvor noch einer Art des Lernens zugeordnet werden, da sich die Aufgaben und Anwendungsfälle in diesen stark unterscheiden können. Die drei Arten, das überwachte, unüberwachte und bestärkende Lernen sind in Kapitel 2.1.6 vorgestellt. Neuroevolutionäre Algorithmen werden hierbei der letzten Art zugeordnet [2]. Häufig werden hierfür zwei Gründe genannt. Erstens ist es das Ziel des bestärkenden Lernens, den insgesamt erhaltenen *reward* zu maximieren [3], was vergleichbar mit dem Fitnesswert bei neuroevolutionären Algorithmen ist. Zweitens haben Sutton und Barto bestärkendes Lernen in ihrer Arbeit als eine Methode beschrieben, welche MDP Probleme lösen kann. Das MDP Problem ist in Kapitel 2.1.6 vorgestellt und beschreibt die Interaktion zwischen einem Agenten und seiner Umgebung. Hierbei kann der Agent als ein Individuum von neuroevolutionären Algorithmen und die Umgebung als Optimierungsproblem betrachtet werden.

Ein grundsätzlicher Vorteil von neuroevolutionären Algorithmen ist, dass durch die vielen unterschiedlichen Individuen unterschiedliche Lösungsansätze überprüft werden können. Dies ermöglicht es den Algorithmen, lokale Optima zu überwinden und die Suche nach dem globalen Optimum weiter fortzuführen [24]. Dies kann für gradientenbasierte Verfahren schwierig sein. Ein weiterer Vorteil ist, dass bei neuroevolutionären Algorithmen keine Gradienten berechnet werden müssen [24]. Dies kann sehr aufwendig sein, wenn zum Beispiel das KNN viele Rückkopplungen enthält oder unmöglich, wenn beispielsweise die binäre Schwellwertfunktion als Aktivierungsfunktion verwendet wird [2]. Für die neuroevolutionären Algorithmen wird die Fitnessfunktion benötigt.

Allerdings kann diese einfache Art der Evaluation zum Nachteil werden, wenn bei-

spielsweise das Optimierungsproblem bzw. die Umwelt verschiedene Startzustände hat. So kann ein Individuum mit einem schlechten Lösungsansatz durch Zufall einen einfachen Startzustand erhalten und so unter Umständen einen höheren Fitnesswert erzielen als ein besserer Lösungsansatz eines anderen Individuums, welches einen ungünstigen Startzustand erhalten hat. Hierdurch werden einige gute Individuen verloren gehen, da diese aufgrund des geringen Fitnesswertes nicht bei der Selektion ausgewählt werden. Die Fitnessfunktion wird in einem solchen Fall als verrauscht oder *noisy* bezeichnet. Ein möglicher Lösungsansatz ist, dass alle Individuen immer denselben Startzustand erhalten. Dies verhindert das vorgestellte Problem, es entsteht aber ein neues. Die Individuen lernen nur eine Lösung für den ausgewählten Startzustand und sind mit hoher Wahrscheinlichkeit in anderen, ungünstigeren Startzuständen unbrauchbar. Eine Alternative ist, dass jedes Individuum mehrmals mit verschiedenen Startzuständen getestet wird und der Fitnesswert von allen akkumuliert wird [2]. Nachteil dieses Ansatzes ist, dass sich die Laufzeit linear erhöht je mehr verschiedene Startzustände getestet werden.

Ein weiteres Problem von neuroevolutionären Algorithmen ist die Laufzeit. Das Entwickeln von unterschiedlichen Lösungen durch verschiedene Individuen benötigt viel Rechenzeit [24]. Für eine Population mit beispielsweise 1000 Mitgliedern werden pro Generation 1000 KNN erstellt, von denen mit jedem versucht wird, das Optimierungsproblem zu lösen. Die hierfür benötigte Laufzeit ist oft höher als bei gradientenbasierten Verfahren, bei welchen nur ein KNN evaluiert werden muss [2]. Gleichzeitig kann dies aber auch ein großer Vorteil für neuroevoltionäre Algorithmen sein, wie unter anderem in Quelle [3] gezeigt ist. Da die einzelnen Individuen unabhängig voneinander sind, lässt sich die Evaluation von diesen gut parallelisieren [24]. Der Vorteil hiervon ist, dass eine Reduzierung der Ausführungszeit unabhängig von Limitierungen einzelner Prozessoren möglich ist [33]. Das bedeutet, dass im Gegensatz zu sequentiellen Programmen, bei denen die Ausführungszeit nur durch einen schnelleren Prozessor gesenkt werden kann, die Laufzeit von parallelisierbaren Algorithmen auch durch das Hinzufügen von weiteren Prozessoren verringert werden kann. Dies ist in vielen Fällen einfacher und kostengünstiger umzusetzen.

Wie erfolgreich eine Parallelisierung sein kann, haben Such u. a. in ihrer Arbeit gezeigt. Hierbei wurde ein KNN zum Lösen von Atari Spielen optimiert. Die hierfür benötigte Trainingszeit mit einem einfachen, parallelisierten neuroevolutionären Algorithmus hat ungefähr vier Stunden gedauert. Verglichen wurde das Ergebnis

mit zwei gradientenbasierten Verfahren, welche für ein vergleichbares Ergebnis mehrere Tage Trainingszeit benötigt haben [3]. Dies zeigt die Leistungsfähigkeit von parallelisierten neuroevolutionären Algorithmen.

2.3 NeuroEvolution of Augmenting Topologies

Der in dieser Arbeit verwendete Algorithmus heißt *NeuroEvolution of Augmenting Topologies* (NEAT), welcher im Jahr 2002 von Stanley und Miikkulainen vorgestellt wurde. Bei der Veröffentlichung hat NEAT für die meisten Optimierungsprobleme im Vergleich zu anderen Verfahren schneller Lösungen gefunden, obwohl es neben den Gewichten des KNN auch die Struktur optimiert [28]. Somit gehört der Algorithmus zur Gruppe der TWEANN Verfahren. Heute gilt NEAT immer noch als einer der bekanntesten Vertretern der neuroevolutionären Algorithmen und dient als Basis für viele Erweiterungen wie zum Beispiel HyperNEAT [55]. Für den großen Erfolg von NEAT sind drei Eigenschaften besonders relevant [28]:

1. Erfolgreiche Reproduktion trotz verschiedener Strukturen
2. Schützen von neuen Innovationen durch verschiedene Spezies
3. Wachsen einer minimalen Struktur

In diesem Kapitel wird die grundsätzliche Funktionsweise von NEAT erläutert, wie sie in der originalen Publikation vorgestellt ist. Wenn nicht anderweitig gekennzeichnet, beziehen sich alle Informationen aus diesem Kapitel auf Quelle [28]. Für eine bessere Lesbarkeit wird in diesem Kapitel auf weitere Zitierungen verzichtet.

2.3.1 Kodierung

NEAT verwendet ein direktes Kodierungsverfahren. Ein Genom enthält, wie in Abbildung (TODO ABBILDUNG) beispielhaft dargestellt, je eine Liste für Neuronen und Verbindungen. Ein Neuron wird durch eine ID identifiziert und enthält den Typ (*Input*, *Output*, *Hidden*). Eine Verbindung enthält das Start- und Zielneuron, das dazugehörige Gewicht, ein Aktivierungsbit sowie eine Innovationsnummer. Das Aktivierungsbit gibt an, ob die Verbindung im Phänotyp, in diesem Fall dem neuronalen Netz enthalten, ist. Auf die Funktionsweise und Bedeutung der Innnovationsnummer wird später genauer eingegangen.

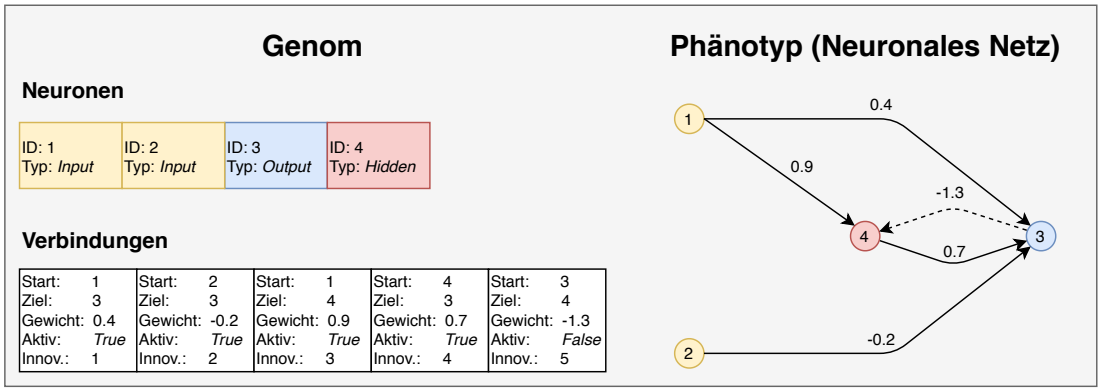

Abbildung 2.7: Schematische Darstellung von einem Genom mit dazugehörigem Phänotyp

2.3.2 Mutation

Ein Genom kann auf verschiedene Arten mutieren, welche entweder die Struktur des KNN oder die Gewichte der Verbindungen beeinflussen. Die Mutation der Gewichte ist ähnlich zu anderen neuroevolutionären Algorithmen. Für jedes Gewicht besteht eine Wahrscheinlichkeit zur Mutation. In diesem Fall wird das Gewicht entweder leicht abgeändert oder ein neuer zufälliger Wert gewählt.

Strukturelle Mutationen können in zwei verschiedenen Arten auftreten. Bei der

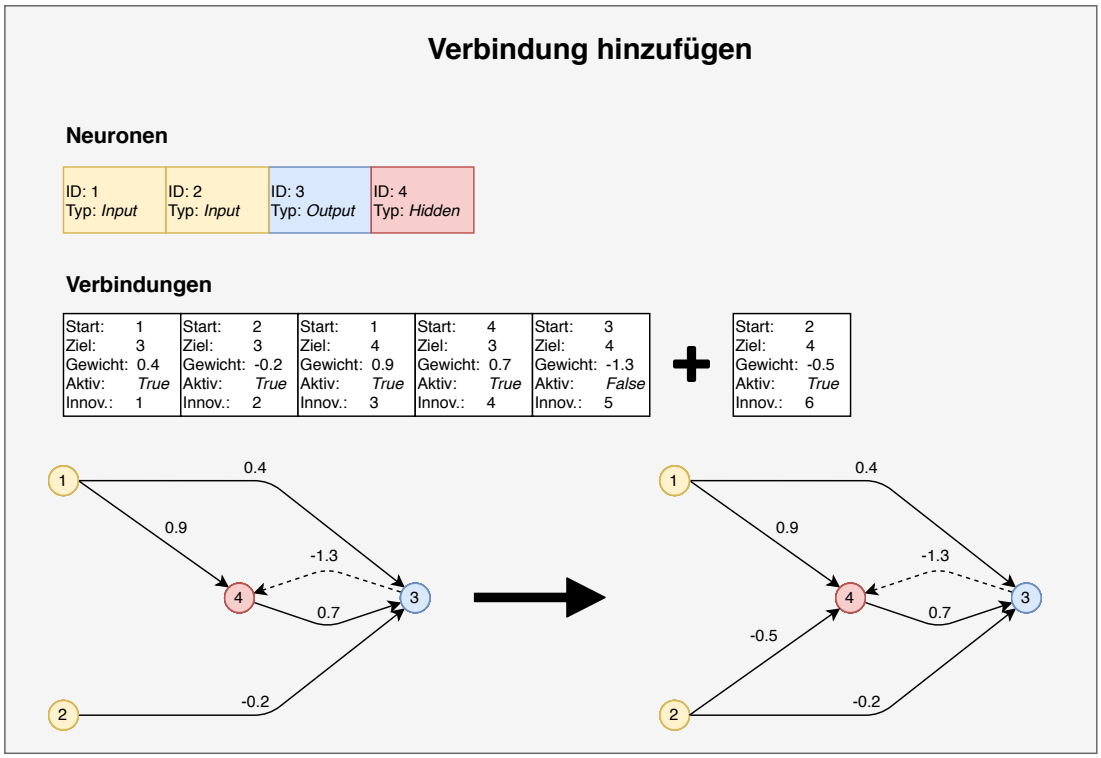

Abbildung 2.8: Schematische Darstellung von einem Genom mit dazugehörigem Phänotyp

ersten wird eine einzelne neue Verbindung dem Genom hinzugefügt. Bei der Auswahl des Start- und Zielneurons ist zu beachten, dass diese nicht bereits über eine solche Verbindung verfügen. Das Gewicht für die neue Verbindung wird zufällig gewählt und das Aktivierungsbit auf *True* gesetzt. Ein Beispiel für diese Mutation ist in Abbildung (TODO ABBILDUNG) dargestellt. Bei der zweiten Art der strukturellen Mutation wird ein neues Neuron in das KNN eingefügt. Hierzu wird zu Beginn eine aktive Verbindung con_{ij} zufällig ausgewählt, welche von Neuron i zu Neuron j führt. Anschließend wird ein neues Neuron x zwischen den Neuronen i und j platziert und zwei weitere Verbindungen werden hinzugefügt. Die erste Verbindung con_{ix} führt vom alten Startneuron i zum neu hinzugefügtem und erhält das Gewicht 1. Die zweite Verbindung con_{xj} beginnt bei dem neuen Neuron und endet im dem alten Zielneuron j und erhält dasselbe Gewicht wie die Verbindung con_{ij}. Zuletzt wird die ausgewählte Verbindung con_{ij} deaktiviert, indem das Aktivierungsbit auf $False$ gesetzt wird. Diese Art der Mutation reduziert den initialen Effekt des neuen Neurons. So kann es direkt vom KNN verwendet werden, ohne dass die Verbindungsgewichte stark optimiert werden müssen.

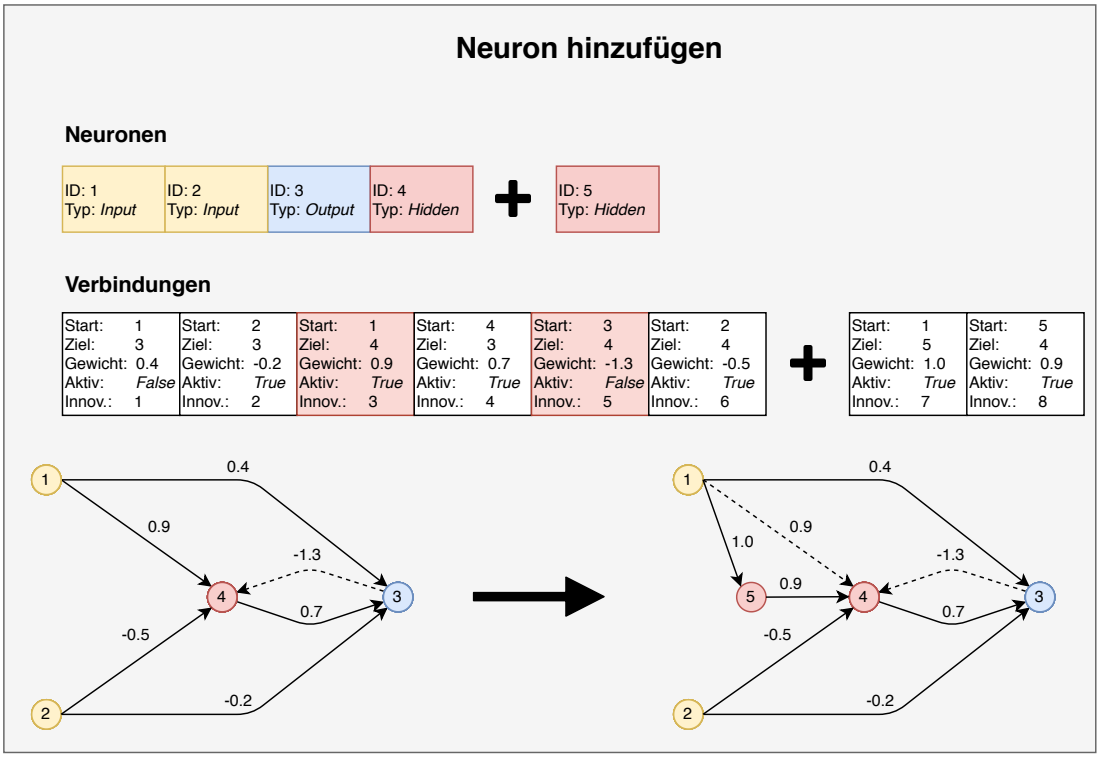

Abbildung 2.9: Schematische Darstellung von einem Genom mit dazugehörigem Phänotyp

2.3.3 Reproduktion

Das Ergebnis der in Kapitel **??** vorgestellten Mutationen ist eine Population mit verschieden Genomen, welche unterschiedliche Gewichte und Strukturen haben können. Dies ist die schwierigste Form des in Kapitel **??** vorgestellten *competing convention* Problems und macht das Erstellen von Nachkommen besonders schwierig.
NEAT löst dieses Problem, indem es den historischen Ursprung von jeder strukturellen Mutation überwacht. Haben zwei Verbindungen denselben Ursprung, haben sie in der Vergangenheit dieselbe Struktur repräsentiert, auch wenn sie inzwischen unterschiedliche Gewichte haben. Zu diesem Zweck besitzt jede Verbindungen die im Kapitel 2.3.1 erwähnte Innovationsnummer. Jedes Mal, wenn eine neue Verbindung entsteht, wird ein globaler Zähler inkrementiert und der Wert als Innovationsnummer der Verbindung verwendet. Abbildung (TODO ABBOLDUNG) zeigt die Zuweisung beispielhaft. Die erste Mutation, welche nur eine neue Verbindung herstellt, hat die Innovatiosnummer X zugewiesen bekommen. Wenn im Folgenden ein neues Neuron mit zwei weiteren Verbindungen hinzugefügt wird, erhalten diese die Nummern Y und Z. Werden Verbindungen von einem Genom in der Reproduktiosnphase für die Nachkommen ausgewählt, wird auch die Innovationsnummer übertragen. Somit ist auch bei den nachfolgenden Generationen ersichtlich, was der historische Ursprung einer Verbindung ist. Tritt durch Zufall dieselbe Mutation in einer Generation mehrfach auf, erhalten die neuen Verbindungen dieselben Innovationsnummern. Hierfür müssen alle aufgetretenen Mutation einer Generation zwischengespeichert werden.
Die Innovationsnummern können nicht nur ressourcensparend implementiert werden, sie machen auch das Erzeugen von Nachkommen in der Reproduktionsphase bedeutend einfacher, da beim Kreuzen von zwei Elternteilen keine aufwendige Strukturanalyse erforderlich ist. Abbildung (TODO ABBILDUNG) zeigt beispielhaft, wie ein Nachkommen aus zwei Elterngenomen X und Y entsteht. Die sogenannten *maching genes* sind Verbindungen, deren Innovationsnummern in beiden Elterngenomen vorkommen. Beim Erstellen der Nachkommen wird für jede Verbindung in den *matching genes* zufällig entschieden, aus welchem Elternteil diese übernommen wird. Die sogenannten *disjoint genes* und *excess genes* sind Verbindungen, die nur in einem Elternteil vorkommen. Zu den *disjoint genes* gehören die Verbindungen, deren Innovationsnummer kleiner als die größte Innovationsnummer des zweiten Elterngenoms ist. Die *excess genes* sind Verbindungen, deren Innovationsnummer größer als die höchste Innovationsnummer im anderen Elternteil ist. Beim Erzeugen von Nachkommen werden nur die *excess genes* und

disjoint genes von dem Elternteil übernommen, welches den höheren Fitnesswert erzielt hat. Haben beide Elternteile den selben Wert, werden die Verbindungen von beiden übernommen. Bei dieser Implementierung wird angenommen, dass der Schwellwert der Neuronen, wie in Kapitel (TODO KAPITE) erläutert, durch eine Verbindung zu einem *Bias*-Neuron ausgedrückt wird. Dadurch enthalten die Neuronen keine spezifischen Informationen, die sich zwischen den Elterngenomen unterscheiden. Die Nachkommen übernehmen deshalb immer die Neuronen des Elternteils mit dem größeren Fitnesswert.

2.3.4 Spezies

Die vorgestellten Arten der Mutation und die erfolgreiche Reproduktion ermöglichen es NEAT, eine Population mit vielen verschiedenen Strukturen zu entwickeln. Dennoch reichen diese Faktoren nicht aus, da in der Praxis neue strukturelle Innovationen nur eine geringe Chance haben, langfristig integriert zu werden und es wahrscheinlicher ist, dass sie nach wenigen Generation aussterben. Die Gründe hierfür sind, dass kleinere KNN schneller optimiert werden können als große und dass das Hinzufügen von neuen Neuronen und Verbindungen den Fitnesswert meistens initial senkt, auch wenn die neuen Strukturen für das erfolgreiche Lösen des Optimierungsproblems notwendig sind. Die Folge ist, dass die kleinen Genome anfänglich bessere Fitnesswerte erzielen, die größeren Genome nicht für die Reproduktion ausgewählt werden wodurch die strukturellen Innovationen wieder verloren gehen.

Das Problem wird von NEAT durch das Einführen von verschiedenen Spezies gelöst. Das Ziel ist, Genome, die sich strukturell ähnlich sind in einer Spezies zu gruppieren. Bei der Auswahl der Elterngenome für die Nachkommen muss ein Genom nicht mehr mit der ganzen Population konkurrieren, sondern nur noch mit den anderen Genomen der eigenen Spezies. Somit sind neue Innovationen erst einmal in ihrer Spezies vor dem Aussterben geschützt und können mit der Zeit optimiert werden. Für die Implementierung eines solchen Verfahrens wird eine Funktion benötigt, die messen kann, wie ähnlich oder unterschiedlich zwei Genome sind. Auch hier kann wie bei der Rekombination auf eine aufwendige Strukturanalyse verzichtet werden, da dies mit den bereits bekannten Innovationsnummern umsetzbar ist. Je mehr *excess genes* und *disjoint genes* zwei Genome besitzen, desto weniger evolutionäre Geschichte teilen sie und sind somit unterschiedlicher. Auch der Gewichtsunterschied ist ein wichtiger Faktor, wie in Kapitel **??** dargestellt. Die von NEAT verwendete Formel, um die Kompatibilität δ zwischen zwei

Genomen zu berechnen, ist im Folgenden abgebildet:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

Die Variablen E und D ergeben sich aus der Anzahl an *excess genes* und *disjoint genes*. $\overline{W}$ ist die durchschnittliche Gewichtsdifferenz der *matching genes*. Die Faktoren c_1, c_2 und c_3 ermöglichen es die Wichtigkeit der einzelnen Komponenten je nach Optimierungsproblem zu justieren. N steht für die Anzahl der Verbindungen im größeren Genom und normalisiert die Anzahl der *excess genes* und *disjoint genes*. Somit ist der Effekt auf den Kompatibilitätswert δ bei einer neuen Verbindung in großen Genomen gering und in kleinen sehr groß. Je nach Konfiguration kann für kleine Genome $N = 1$ gelten.

Die Zuordnung von neu erstellten Genomen zu einer Spezies erfolgt nach der Reproduktions- und Mutationsphase. Hierfür wird eine geordnete Liste mit allen verfügbaren Spezies benötigt. Jede Spezies wird durch ein Genom repräsentiert, welches in der vorherigen Generation ein Mitglied von dieser war. Bei der Zuordnung von einem Genom wird über die Liste der Spezies iteriert und zu jedem Repräsentanten der Kompatibilitätswert δ gebildet. Ist $\delta \leq \delta_t$, wobei δ_t ein konfigurierbarer Schwellwert ist, wird das Genom der Spezies zugeordnet und die Suche abgebrochen. Ist das Genom zu keiner Spezies kompatibel, wird eine neue erstellt und das Genom als Repräsentant gesetzt.

Zum Erhalten von verschiedenen Strukturen muss verhindert werden, dass eine Spezies zu groß wird und die restlichen verdrängt, auch wenn viele der Mitglieder gute Fitnesswerte erzielen. Zusätzlich müssen insbesondere neue Spezies geschützt werden. Diese haben initial wenige Mitglieder und somit eine geringere Chance, als Elterngenome ausgewählt zu werden. Zum Lösen dieses Problems verwendet NEAT sogenanntes *explicit fitness sharing*, welches 1987 von Goldberg, Richardson u. a. in ihrer Arbeit [34] vorgestellt wurde. Jede Spezies bekommt bei der Reproduktion eine Anzahl an Nachkommen zugewiesen, welche proportional zur Fitness f_s der Spezies ist. Diese ergibt sich aus der Summe aller angepassten Fitnesswerte f' der Mitglieder. Der angepasste Fitnesswert f' eines Genoms wird berechnet, indem die erreichte Fitness f durch die Anzahl der Mitgliedern der Spezies geteilt wird. Ziel dieser Maßnahme ist, dass große Spezies im Vergleich zu kleinen benachteiligt werden. Hierdurch werden kleineren erfolgreichen Spezies entsprechend viele Nachkommen zugeordnet. Ein Beispiel hierfür ist in Abbildung xy (TODO ABBILDUNG) dargestellt. Obwohl die zweite Spezies bedeutend

weniger, jedoch gute Genome besitzt, werden ihr mehr Nachkommen zugewiesen. Würde die Anzahl der Nachkommen einer Spezies proportional zu der Summe der erreichten Fitnesswerte vergeben, hätte die kleinere Spezies weniger Nachkommen zugewiesen bekommen.

Ist der Fitnesswert f_s von jeder Spezies berechnet und die Nachkommen proportional zugeteilt, beginnt die Reproduktion. Die Elterngenome werden hierfür zufällig aus der Mitgliederliste ausgewählt mit der Einschränkung, dass nur die besten 50% der Genome ausgewählt werden. Sind alle Nachkommen erstellt, wird die ganze Population gelöscht und durch die Nachkommen ersetzt. Diese werden mit dem bereits vorgestellten Verfahren wieder den Spezies zugeordnet.

2.3.5 Starten mit einer minimalen Struktur

Das Ziel von NEAT sowie vieler anderer Optimierungsalgorithmen ist, eine Lösung in kürzester Zeit zu finden. Ein wichtiger Faktor ist hierbei die Größe des KNN. Ein zu großes KNN hat viele modifizierbare Verbindungsgewichte und Schwellwerte, welche nicht zur erfolgreichen Lösung benötigt werden. Trotzdem wird die Laufzeit des Algorithmus erhöht, da auch diese optimiert werden müssen. Ein zu kleines KNN kann, wie in Kapitel (TODO REF XOR) veranschaulicht, unter Umständen nicht in der Lage sein, eine Lösung zu finden. Somit ist die richtige Größe des KNN entscheidend für die schnelle Optimierung. Für Algorithmen, welche nur die Gewichte eines KNN optimieren, muss diese Struktur von einem Menschen festgelegt werden. Meistens basiert dies auf Expertenwissen oder Erfahrung [19]. Im Gegensatz hierzu stehen die TWEANN Algorithmen, welche selbstständig eine gute Struktur bilden sollen. Diese starten oft mit einer initialen Population mit vielen verschiedenen zufällig erstellten Topologien mit dem Ziel, genetische Diversität zu bieten. Wie in Kapitel 2.2.3 erläutert, ist dies oft ineffizient, da viele Strukturen nicht gebraucht werden und Zeit benötigt wird, diese zu entfernen.

NEAT hingegen startet mit einer Population, bei der alle Genome dieselbe minimale Struktur besitzen. Die entstehenden KNN haben nur *Input*- und *Output*-Neuronen und keine *Hidden*-Neuronen. Jedes *Input*-Neuron besitzt eine Verbindungen zu jedem *Output*-Neuron mit einem zufällig gewählten Gewicht. Neue Strukturen werden durch die vorgestellten Arten der Mutation hinzugefügt, von denen nur diejenigen langfristig integriert werden, welche den Fitnesswert erhöhen. Somit ist die Existenz von jeder Struktur in einem Genom gerechtfertigt. Insgesamt gibt dies NEAT einen Vorteil bezüglich der Evaluationszeit gegenüber

anderen TWEANN Algorithmen, da die Anzahl der zu optimierenden Parameter und somit die Dimensionen des Suchraums minimiert sind.

2.4 Parallelisierung

In den vorherigen Kapiteln ist der Ablauf und die Funktionsweise von neuroevolutionären Algorithmen erläutert. Die benötigte Ausführungszeit von diesen ist sehr von der Größe des KNN und der Komplexität des Problems abhängig. Für große KNN werden teilweise Trainingszeiten von mehreren Stunden oder Tagen benötigt und dies trotz der Verwendung von aktueller Hardware [3]. Natürlich kann diese Zeit durch Weiterentwicklungen von einzelnen Prozessoren zunehmend verringert werden. Allerdings ist der Leistungsanstieg von neuen Prozessorgenerationen nicht ausreichend, um die benötigte Rechenzeit von solch anspruchsvollen Anwendungen massiv zu senken und skaliert somit in diesem Anwendungskontext schlecht. Zusätzlich ist es auch finanziell aufwendig, immer die neuesten Prozessoren zu kaufen und diese nach kurzer Zeit wieder zu ersetzen [35]. Die Parallelisierung ist ein weiterer Ansatz, um die benötigte Ausführungszeit zu verringern. Hierbei wird ein großes Problem in mehrere kleine und voneinander unabhängige Teilprobleme zerlegt. Diese können dann auf verschiedenen Prozessoren gleichzeitig berechnet werden [35]. Häufig wird in diesem Zusammenhang auch der Begriff *High Performance Computing* (HPC) verwendet, auf welchen im Folgenden näher eingegangen wird.

2.4.1 High Performance Computing

Der Bereich HPC beschäftigt sich mit verschiedenen Bereichen der parallelen Programmierung. Hierzu gehören unter anderem die benötigte Software, Programmiersprachen, Tools, aber auch die Hardware. Zusammenfassend ist festzustellen, dass sich der Bereich HPC mit der Forschung, Entwicklung und dem Betreiben von Super Computern (SC) beschäftigt. Dies sind Cluster, welche aus mehreren Millionen *central processing unit*s (CPUs) bestehen können und zum Lösen von verschiedenen parallelisierbaren Problemen aus Forschung und Industrie verwendet werden können [36]. Eine Liste mit den 500 leistungsfähigsten SC ist in Quelle [37] zu finden. Stand Juni 2020 ist die Nummer eins ein SC in Japan, welcher über sieben Millionen Prozessoren besitzt und eine Leistung von über 500.000 TFlops bietet. Natürlich sind SC aufgrund hoher Kosten, die unter anderem durch den Stromverbrauch entstehen, für viele Probleme nicht rentabel. Dennoch gibt es einige Anwendungsszenerain für solche Systeme.

Häufig werden SC für verschiedene Simulationen eingesetzt, beispielsweise wenn es zu teuer oder gefährlich ist, diese in realer Umgebung durchzuführen. Mögliche Anwendungsszenarien sind hierfür die Simulation von Flugzeugabstürzen oder nuklearen Waffen. Ein weiterer Grund für den Einsatz von SC liegt vor, wenn die Berechnung oder Simulation auf einem gewöhnlichen Gerät zu viel Zeit benötigt oder wenn das Ergebnis nur eine gewisse Zeit gültig bzw. verwendbar ist. Ein Problem dieser Kategorie ist beispielsweise die Wettervorhersage. Ist es nicht möglich, das Ergebnis rechtzeitig zu erhalten, ist die Vorhersage obsolet. Ein weiteres Gebiet ist die Analyse von großen Datenmengen, die nicht auf einem einzelnen Gerät effizient durchführbar ist [36].

Architektur

Typischerweise werden die Architekturen von HPC Systemen einer von zwei Kategorien zugeordnet, welche als *shared memory* und *distributed memory* bezeichnet werden. Bei der *shared-memory* Architektur verwenden typischerweise alle Prozessoren des Systems denselben Programmspeicher, auch als *random-access memory* (RAM) bezeichnet. Die Kommunikation zwischen den Prozessoren ist in vielen Fällen durch *Open Multi-Processing* (OpenMP) realisiert [36]. Diese Art der Parallelisierung kann auch auf Computersystemen angewendet werden, die für den Massenmarkt produziert wurden. Moderne CPUs besitzen auf demselben Chip mehrere physische Prozessoren, welche zur Parallelisierung von verschiedenen Anwendungen verwendet werden können und häufig die Ausführungszeit bedeutend reduzieren.

Die Alternative hierzu ist die *distributed memory* Architektur, bei der jeder Prozessor seinen eigenen RAM Speicher besitzt. Im Gegensatz zu *shared memory* Architekturen können die Prozessoren dieser Art nicht auf Speicherbereiche von anderen Prozessoren zugreifen. Um eine Kommunikation zwischen den einzelnen Prozessoren zu ermöglichen, müssen sich diese im selben Netzwerk befinden und explizite Nachrichten unter einander austauschen. Die Ausführungsgeschwindigkeit beziehungsweise die Effizienz der parallelisierten Anwendung ist nicht nur von den einzelnen Prozessoren abhängig, sondern auch von der Latenz, Bandbreite und Netztopologie. Die Latenz beschreibt hierbei die Zeit, welche benötigt wird, eine Kommunikation zu initiieren und die Bandbreite die Übertragungsgeschwindigkeit der Daten. Neben diesen beiden Architekturen gibt es noch weitere Formen, welche auch *graphics processing unit*s (GPUs) nutzen können [36]. Da diese Arbeit ein

System mit einer *distributed memory* Architektur verwendet, wird auf diese nicht weiter eingegangen.

Beowulf Cluster

Viele professionelle SC bestehen aus spezialisierter Hardware, welche in der Anschaffung sehr teuer ist [38]. Zusätzlich sind auch die Betriebskosten eines solchen Systems sehr hoch, zum Beispiel durch den entstehenden Stromverbrauch [36]. Daher ist die Anschaffung eines solchen Systems für die meisten Unternehmen, Universitäten oder Hochschulen finanziell nicht tragbar. Dennoch kann der Einsatz von kleineren Clustern sinnvoll sein, zum Beispiel in der Lehre oder für weniger rechenaufwendige Probleme, welche trotzdem von einer Parallelisierung profitieren.

In einem solchen Szenario sind sogenannte Beowulf Cluster eine mögliche Lösung. Diese haben zwar bedeutend weniger Leistung, sind dafür aber in der Anschaffung sowie im Betrieb um ein vielfaches günstiger und eignen sich somit auch für Privatpersonen [39]. Diese Cluster zeichnen sich durch verschiedene Eigenschaften aus. Ein großer Unterschied zu professionellen SC ist, dass die Hardware für den eigentlichen Beowulf Cluster, aus Geräten besteht, welche serienmäßig für den Massenmarkt produziert werden. Gleiches gilt auch für die Netzwerkinfrastruktur. Beispielsweise kann ein Cluster aus mehreren Desktopgeräten bestehen, welche über ein Ethernet-Netzwerk miteinander verbunden sind. Weitere Anforderungen sind, dass alle Geräte des Clusters, nur *open source* Software verwenden, das Netzwerk exklusiv für die Kommunikation des Beowulf-Clusters reserviert ist und das Aufgabengebiet im Bereich des HPC liegt [38]. Diese Anforderungen müssen für einen klassischen Beowulf Cluster erfüllt sein, dennoch gibt es verschiedene abgewandelte Varianten für andere Einsatzszenerien.

Es gibt noch weitere Eigenschaften, welche auf die meisten Beowulf Cluster zutreffen, aber keine Voraussetzung sind. Typischerweise basiert das Betriebssystem der einzelnen Geräte, welche als Nodes bezeichnet werden, auf Linux. Zudem wird häufig ein Node als *Master* ausgewählt, die restlichen Nodes werden als *Slaves* bezeichnet. Der *Master* dient häufig als Schnittstelle zwischen dem eigentlichen Cluster und der externen Umgebung. So kommt es häufig vor, dass dieser über eine angeschlossene Tastatur und einen Bildschirm verfügt, welche eine Interaktion mit dem Cluster ermöglicht. Zusätzlich hat dieser häufig als einziges Gerät neben der Verbindung zu den *Slaves* auch eine externe Netzwerkanbindung. Dies wird benötigt, da der *Master* in der Regel viele organisatorische Aufgaben übernimmt.

Beispiel hierfür kann das Bereitstellen und die Synchronisation von Dateien im Netzwerk sein [38].

Aufgrund der vergleichsweise niedrigen Kosten sowie der einfachen Anschaffung und Inbetriebnahme bieten sich solche Cluster für viele verschiedene Projekte an. Auch in dieser Arbeit wird ein Beowulf Cluster verwendet. Auf die Installation und Konfiguration von diesem wird in Kapitel (TODO Kapitel) eingegangen.

MPI und MapReduce (TODO Change Title)

Für eine erfolgreiche Parallelisierung in einem *distributed memory* System, wie zum Beispiel in einem Beowulf Cluster, müssen sich die einzelnen Prozessoren untereinander synchronisieren sowie Nachrichten austauschen können. Hierfür gibt es verschiedene Bibliotheken und Frameworks, welche einen Großteil dieser Funktionen bereitstellen. Beispiele hierfür sind *Message Passing Interface* (MPI) und *MapReduce*. Diese unterscheiden sich in ihrer Funktionalität und der Umgebung, in welcher sie eingesetzt werden [36].

In dieser Arbeit wird MPI verwendet, welches in Kapitel 2.4.2 genauer erläutert wird. Vorteil von diesem ist, dass eine Vielzahl verschiedener Kommunikations- und Synchronisationsoperatoren implementiert sind, welche flexibel an viele Anforderungen angepasst werden können. Allerdings besteht keine Fehlertoleranz gegenüber Hardware- und Netzwerkfehlern. Treten diese auf, bricht die Ausführung der gesamten Anwendung ab und nicht gespeicherte Zwischenergebnisse gehen verloren. In diesem Punkt bietet die Alternative *MapReduce* einen Vorteil, da diese Fehler automatisch verarbeiten kann. Nachteil gegenüber MPI ist, dass nicht so viele verschiedene, flexible Methoden der Parallelisierung geboten werden [36].

2.4.2 MPI

Wie im vorherigen Kapitel erläutert, müssen in einem parallelen System die Prozessoren miteinander kommunizieren können. Vor allem bei Systemen mit einer *distributed memory* Architektur wird häufig MPI verwendet, welches ein anerkannter Standard im Bereich HPC ist. Zusätzlich kann MPI auch auf *shared memory* Architekturen angewendet werden. Die erste Version des Standards wurde bereits 1991 entwickelt [36]. An der Entwicklung waren über 80 Personen aus ungefähr 40 verschiedenen Organisationen beteiligt [40]. Im Jahr 2008 wurde die derzeit aktuelle Version 3 veröffentlicht. Bevor die verschiedenen Funktionen von

MPI vorgestellt werden, wird zunächst auf einige Besonderheiten eingegangen. Der Standard MPI ist keine konkrete Implementierung, sondern ein *Application Programming Interface* (API), welches nur die grundsätzliche Funktionsweise sowie einige Basisoperationen definiert [36]. Hierdurch ergeben sich viele Vorteile. Die Implementierung des Standards ist nicht an eine einzelne Programmiersprache geknüpft [36]. Dies gibt Herstellern die Möglichkeit, ihre eigene Implementierung in jeder gewünschten Sprache umzusetzen. Hieraus sind viele kommerzielle Produkte entstanden, aber auch zwei bekannte *open source* Lösungen, welche MPICH und Open MPI heißen. Diese können als Bibliotheken in andere Projekte eingebunden werden und vereinfachen die Entwicklung einer parallelen Anwendung enorm, da verschiedene *low-level* Funktionen, wie beispielsweise die Netzwerkkommunikation, bereits implementiert sind. Des Weiteren ermöglicht der MPI Standard eine hohe Portabilität, da Implementierungen mit wenig Aufwand ausgetauscht werden können und auf einer Vielzahl von Systemen lauffähig sind [41].

Die Funktionen von MPI können heutzutage in verschiedenen Sprachen, wie zum Beispiel Java, C, C++, Python und Fortran verwendet werden [36]. Im Folgenden wird auf die wichtigsten Grundfunktionen von MPI eingegangen. Der Beispielcode ist in der Sprache Python und der Bibliothek *mpi4py* aus Quelle [41] implementiert. Diese werden auch im weiteren Verlauf dieser Arbeit verwendet.

Prozessorgruppen

Bevor die verschiedenen Kommunikations- und Synchronisationsmöglichkeiten vorgestellt werden, muss der grundlegende Ablauf eines auf MPI basierenden Programms betrachtet werden. Beim Starten eines solchen Programms wird auf jedem der beteiligten Prozesse, welche in einer Konfigurationsdatei spezifiziert sind, eine Kopie des Programms ausgeführt. Eine grundlegende Voraussetzung ist, dass das Programm auf allen beteiligten Nodes vorliegt.

Die sogenannten *Communicators*, auch häufig im Programmcode mit *COMM* abgekürzt, sind Gruppen von verschiedenen Prozessen, welche miteinander kommunizieren können. Beim initialen Starten des Programms wird nur eine Prozessgruppe erstellt, welche alle beteiligten Prozesse enthält und als *MPI_COMM_WORD* bezeichnet wird. Jeder Prozess in einem *Communicator* wird durch einen Rang zwischen 0 und $P - 1$ identifiziert, wobei P die Anzahl an Prozessoren ist [36]. Beide Werte können im Programmcode angefragt und basierend auf diesen Entscheidungen getroffen werden. Ein Beispiel hierfür ist in Abbildung 2.10 dargestellt.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print("Hello world, Rank:", rank, ", Size:", size)

>>> Hello world, Rank: 1, Size: 2
>>> Hello world, Rank: 0, Size: 2
```

Abbildung 2.10: *HelloWord* MPI Programm in Python

Der gegebene Programmabschnitt wurde mit zwei Prozessoren ausgeführt. An der Ausgabe ist erkennbar, dass die Anweisung *print()* von jedem Prozess einmal aufgerufen wurde und sich nur die Variable *rank* unterscheidet.

Point-to-Point Kommunikation

Typischerweise wird MPI unter anderem zum Austauschen von Nachrichten verwendet. In diesem Abschnitt soll die sogenannte *Point-to-Point* Kommunikation vorgestellt werden. Hierbei sendet ein Prozess entweder synchron oder asynchron Daten an einen anderen [36]. Abbildung 2.4.2 zeigt ein Beispiel für die synchrone Kommunikation. Die hierfür benötigten Methoden sind die *send()* und *recv()* Funktion. Die *send()* Funktion versendet Daten an einen anderen Prozess. Als Parameter müssen die zu sendenden Daten und das Ziel übergeben werden. Dieses wird durch den entsprechenden Rang identifiziert. Optional kann dieser Funktion noch ein *tag* übergeben werden, welches zusätzliche Informationen zu den Daten enthält. Beispielsweise kann dieses verwendet werden, um dem Zielprozess zu signalisieren, wie die Daten verarbeitet werden sollen. Die *recv()* Funktion wird zum Empfangen der Daten verwendet. Diese kann prinzipiell ohne Parameter aufgerufen werden, sodass von jedem Prozess jede Nachricht akzeptiert wird. Sollen spezielle Nachrichten mit einem gewissen *tag* oder von einem spezifischen Prozess empfangen werden, ist dies durch Übergeben von weiteren Parametern möglich. Dies kann wichtig sein, wenn zum Beispiel Nachrichten in einer gewissen Reihenfolge empfangen werden müssen.

Die synchrone Kommunikation kann zwei Probleme verursachen. Das erste Problem betrifft die Effizienz, da sowohl die *send()* als auch *recv()* Funktion die weitere Ausführung des Programmcodes blockieren bis die Übertragung erfolgreich abgeschlossen ist. Sollte einer der beiden Prozesse ausgelastet sein, wird

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
  data_to_send = 42
  comm.send(data_to_send, dest=1)
  print("Data sent: ", str(data_to_send))
elif rank == 1:
  data_recv = comm.recv()
  print("Data received: " + str(data_recv))

>>> Data sent:  42
>>> Data received: 42
```

Abbildung 2.11: *Point-to-Point* Kommunikation mit MPI in Python

der andere auf ihn warten und in dieser Zeit keine Berechnungen durchführen [36]. Kommen diese Wartezeiten häufig und lange vor, kann dies die Performanz des Systems stark beeinträchtigen. Das zweite Problem besteht darin, dass Fehler im Programmcode zu *Deadlocks* führen können. Eine solche Situation kann entstehen, wenn zwei Prozesse eine Nachricht vom jeweils anderen erwarten und daher nicht mit der Programmausführung fortfahren [36]. Da kein Prozess eine Nachricht schicken wird, muss das Programm in einem solchen Fall extern manuell beendet werden und alle nicht gespeicherten Zwischenergebnisse gehen verloren. Um diese Probleme zu umgehen, kann die asynchrone Variante der *Point-to-Point* Kommunikation verwendet werden, welche mit den Funktionen *isend()* und *irecv()* implementiert ist. Möchte ein Prozess Daten senden bzw. empfangen und ist der Kommunikationspartner noch nicht bereit, wird mit der Ausführung des Programmcodes fortgefahren. Wenn der Kommunikationspartner letztendlich für die Übertragung bereit ist, wird diese automatisch gestartet [36].

Gruppenkommunikation

Der MPI Standard definiert nicht nur die *Point-to-Point*-, sondern auch verschiedene Formen der Gruppenkommunikation. Diese können in einigen Fällen bedeutend effizienter sein, als Nachrichten mit allen Prozessen einzeln auszutauschen. Einige der wichtigsten Operationen sind die *Broadcast*, *Scatter*, *Gather* und *Reduce* Funktion, welche beispielhaft in Abbildung 2.12 dargestellt sind [40].

Mit der *Broadcast* Funktion kann ein Prozess eine Nachricht M an alle an-

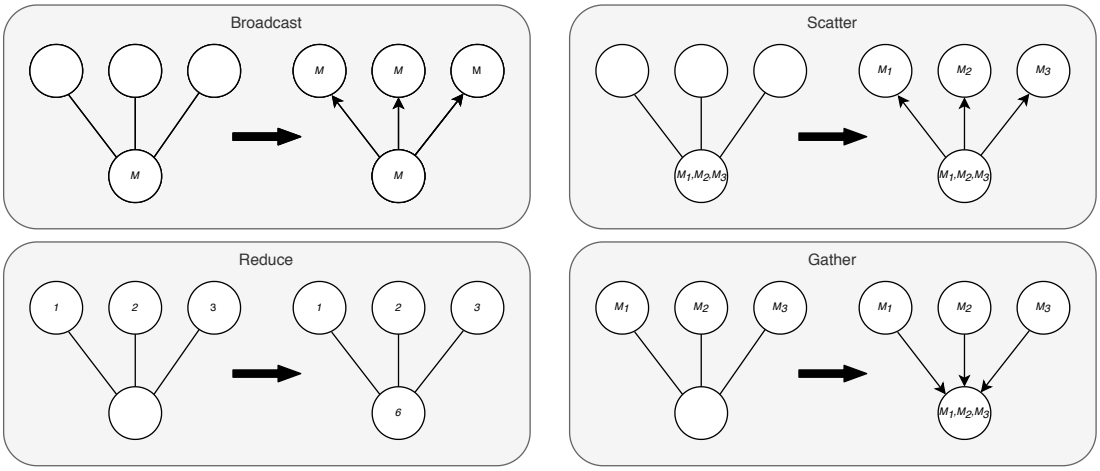

Abbildung 2.12: Schematische Darstellung der *Broadcast*, *Scatter*, *Gather* und *Reduce* Funktion in *MPI*

deren beteiligten Prozesse senden [40]. Diese einfache Operation wird in vielen Fällen benötigt, wenn zum Beispiel der *Master* Prozess in einem Cluster Daten an alle *Slaves* senden muss. Mit der *Scatter* Funktion kann ein Prozess die Elemente einer Liste bzw. Arrays gleichmäßig an die anderen Prozesse verteilen, sodass jeder Empfänger nur einen Teil der Daten erhält [36]. Diese Operation kann eingesetzt werden, wenn auf jedem Element der Liste eine vom Rest unabhängige Berechnung durchgeführt werden muss. Zwar wäre es in einem solchen Szenario auch möglich, die *Broadcast* Funktion zu verwenden und dann basierend auf dem Rang die Berechnung durchzuführen, aber bei der *Scatter* Funktion werden insgesamt weniger Daten übertragen und somit eine geringere Bandbreite benötigt. Das Pendant zur *Scatter* Funktion ist die *Gather* Funktion. Hierbei erhält ein Prozess von allen anderen eine Nachricht M_i und aggregiert diese in einer Liste oder einem Array [36]. Die *Scatter* und *Gather* Funktionen werden häufig zusammen verwendet. Mit der *Scatter* Funktion können die Daten an die verschiedenen Prozesse verteilt werden, diese berechnen ihre Ergebnisse, welche zuletzt mit der *Gather* Funktion wieder gesammelt werden. Die letzte hier vorgestellte Funktion wird *Reduce* genannt. Mit dieser können verschiedene globale Berechnungen mit einem binären kommutativen Operator durchgeführt werden [36]. Eine Rechenoperation ist kommutativ, wenn für alle a und b einer Menge M die Gleichung $a * b = b * a$ gilt [42]. In Abbildung 2.12 wird die Summe berechnet, aber MPI bietet auch viele andere Standardoperatoren, wie zum Beispiel die Berechnung des Produkts, Minimums oder Maximums [36].

MPI bietet noch weitere Arten der Gruppenkommunikation, auf welche an dieser

Stelle nicht ausführlich eingegangen wird. Die *AllGather* Funktion kombiniert die *Gather* Funktion mit einem anschließenden *Broadcast* des Ergebnisses. Bei der *AllToAll* Funktion besitzt jeder Prozess eine Liste und sendet jeweils ein Element an jeden anderen Prozess [40]. Eine spezielle Art der Gruppenkommunikation sind die sogenannten Synchronisationsbarrieren. An diesen wird die Ausführung der einzelnen Prozesse angehalten bis alle Teilnehmer diese erreicht haben [36]. Abschließend sind zwei Anmerkungen zu nennen, welche für die verschiedenen Arten der Gruppenkommunikation wichtig sein können. Erstens ist jede dieser Operationen nur als synchrone Funktion verfügbar. Die zweite Anmerkung betrifft die Synchronisationsbarrieren. Bei vielen Implementierungen haben Funktionen wie der *Broadcast* einen Synchronisationsseiteneffekt. Dies ist durch den *MPI* Standard möglich, jedoch keine Voraussetzung und kann somit die Portierung von Projekten beeinflussen, die von diesen Seiteneffekten abhängig sind [40].

2.4.3 Performance

Das Erstellen eines gut parallelisierten Algorithmus ist häufig bedeutend schwieriger als das Erstellen desselben in einer sequentiellen Variante. Der Grund hierfür ist, dass zwar jeder parallele Algorithmus sequentiell ausführbar ist, indem die unabhängigen Teilaufgaben nacheinander abgearbeitet werden, dies aber umgekehrt nicht möglich ist. Um gute Ergebnisse zu erzielen, muss der parallelisierte Algorithmus häufig neu erstellt oder umstrukturiert werden. [36]. Dieses Kapitel stellt Methoden vor, mit welchen die tatsächliche und maximal zu erwartende Performance eines parallelen Algorithmus berechnet werden kann.

Bevor auf die genauen Berechnungen eingegangen wird, sei angenommen, dass die serielle Ausführung eines Algorithmus t_{seq} lange dauert und dass t_q die Zeit angibt, welche derselbe parallele Algorithmus benötigt, welcher von P vielen Prozessen ausgeführt wird. Der sogenannte *SpeedUp* gibt an, um welchen Faktor die parallele Ausführung eines Programms schneller ist. Berechnet wird dies mit $SpeedUp(P) = \frac{t_{seq}}{t_P}$ [36]. Der *SpeedUp* gibt nicht an, wie gut oder schlecht die tatsächliche Parallelisierung ist. Hierfür wird die *Efficiency* e benötigt, welche mit $e = \frac{SpeedUp(P)}{P}$ berechnet wird. An dieser kann abgelesen werden, wie gut die parallele Ausführung tatsächlich ist. Der berechnete Wert liegt meistens zwischen 0 und 1, wobei ein niedriger Wert ein dafür Indiz ist, dass durch die Parallelisierung und eventuelle Kommunikation viel zusätzlicher Rechenaufwand entsteht. Dementsprechend zeigt ein hoher Wert, dass wenig zusätzlicher Aufwand entsteht und die Parallelisierung effizient erfolgt. In seltenen Fällen ist es sogar möglich, dass höhere

Werte als 1 erzielt werden, was einem *super-linear SpeedUp* entspricht. Dies kann vorkommen, wenn beispielsweise mehrere Prozesse denselben Cache verwenden. Hierbei ist es möglich, dass ein Prozess Rechenzeit einsparen kann, indem er ein Ergebnis aus dem Cache wiederverwendet, das ein anderer Prozess berechnet hat [36].

Mit den vorgestellten Formeln lassen sich der *SpeedUp* und die *Efficiency* berechnen, allerdings geben diese keine Auskunft darüber, was der höchst mögliche *SpeedUp* bzw. die kürzeste Ausführungszeit ist. Um diese zu berechnen, gibt es zwei Theoreme, welche unter den Begriffen *Amdahl's Law* und *Gustafson's Law* bekannt sind [36].

Amdahl's Law wurde 1967 von Gene Amdahl in seiner Arbeit in Quelle [43] beschrieben. Die im Folgenden vorgestellten Formeln wurden erst später hiervon abgeleitet [43]. Angenommen wird ein Algorithmus bestehend aus zwei Teilen, bei dem der erste Teil α_{seq} rein sequentiell und der zweite Teil α_{par} parallelisierbar ist. Weiterhin wird angenommen, dass $\alpha_{seq} + \alpha_{par} = 1$ ist und dass t_1 die Zeit angibt, welche ein Prozess zum Ausführen eines Programmabschnittes benötigt. Die Ausführungszeit t_P eines solchen parallelisierten Programms mit P Prozessen kann mit $t_P = \alpha_{seq} \cdot t_1 + \alpha_{par} \cdot \frac{t_1}{P}$ berechnet werden. Um den höchsten erreichbaren *SpeedUp* zu berechnen wird angenommen, dass $t_1 = t_{seq}$ gilt und dass der erhaltenen Wert t_P in die bereits vorgestellte Formel für die Berechnung des *SpeedUps* eingesetzt wird. Hierdurch ergibt sich die bekannte Formel von *Amdahl's Law* [36]:

$$SpeedUp(P) = \frac{1}{\alpha_{seq} + \frac{\alpha_{par}}{P}}$$

Angenommen es gäbe unendlich viele Prozessoren ($P \to \infty$), dann ist die Ausführungszeit des parallelisierbaren Programmteils vernachlässigbar gering, sodass nur der sequentielle Anteil das Ergebnis maßgeblich beeinflusst und somit gilt [36]:

$$\lim_{P \to \infty} SpeedUp(P) = \frac{1}{\alpha_{seq}}$$

Das bedeutet beispielsweise, dass der maximale *SpeedUp* für einen Algorithmus 20 ist, wenn dieser zu 95% parallelisierbar und dementsprechend zu 5% seriell ist.

Amdahl's Law gilt unter der Annahme, dass bei steigender Prozessanzahl der Rechenaufwand der zu lösenden Aufgabe konstant bleibt, was für viele Anwendungsszenarien zutrifft [36]. Im Gegensatz hierzu steht das *Gustafson's Law*,

welches von John Gustafson stammt. Dieser argumentiert, dass *Amdahl's Law* parallelen Systemen nicht gerecht wird, da der eigentliche Vorteil von diesen die Bearbeitung von größeren Datenmengen in derselben Zeit ist. In diesem Fall wird mit steigender Prozessanzahl ebenfalls der Rechenaufwand erhöht. Dies ermöglicht es, höhere *SpeedUp* Werte zu erzielen [44].

3 Softwarearchitektur und Implementierung

Für die Analyse von NEAT wird eine sequenzielle Implementierung benötigt, welche im Rahmen dieses Kapitels umgesetzt wird. Dafür werden zuerst in Kapitel 3.1 einige grundlegende Anforderungen definiert. Diese sind sowohl von der sequenziellen als auch parallelisierten Implementierung umzusetzen. In Kapitel 3.2 ist die Softwarearchitektur dieses Projektes beschreiben. Das umfasst die verwendeten Klassen und eine Beschreibung der Schnittstelle. Zuletzt wird in Kapitel 3.3 das sequenzielle Verfahren vorgestellt.

3.1 Anforderungen

Aus dem Untertitel dieser Arbeit *Analyse und Optimierung von NEAT für ein verteiltes System* können drei verschiedene Ziele abgeleitet werden, mit welchen sich die folgenden Kapitel beschäftigten. Das erste Ziel ist die Implementierung einer sequentiellen Version des NEAT Algorithmus. Hierauf basiert die zweite Phase, in welcher die korrekte Funktionalität der Implementierung verifiziert sowie eine Performance Analyse durchgeführt wird. Basierend auf der Analyse ist das dritte Ziel die Optimierung des Algorithmus mit MPI für einen Bewoulf Cluster.

Dieses Kapitel befasst sich mit dem ersten Ziel, der Implementierung des sequenziellen Algorithmus. Hierzu werden in einem ersten Schritt verschiedene Anforderungen an die Software definiert, welche sowohl von der sequenziellen als auch der später vorgestellten parallelisierten Implementierung erfüllt werden müssen. Danach wird auf die verschiedenen Datenstrukturen, Schnittstellen und Implementierungsdetails eingegangen.

Die erste Anforderung betrifft die Programmiersprache. Für dieses Projekt soll die Sprache Python verwendet werden. Dies ist eine moderne, einfach zu lernende Hochsprache, deren Vorteile eine einfache Syntax sowie die dynamische Typisie-

rung von Variablen und Parametern sind. Dies ermöglicht in vielen Fällen eine schnelle Entwicklung von Prototypen oder auch vollwertigen Softwaresystemen. Zusätzlich bietet die Sprache viele verschiedene *open source* Pakete, welche einfach in bestehende Projekte integriert werden können und fertige Lösungen für diverse Aufgabengebiete bieten. Aus diesem Grund ist Python eine der bekanntesten Sprachen im Bereich Forschung und Entwicklung [45]. Dies gilt insbesondere für die Bereiche *Machine Learning* und KNN, da viele hierfür verwendete Bibliotheken primär für Python entwickelt sind, wie beispielsweise Tensorflow [46], PyTorch [47] und das OpenAI Gym [48]. Letzteres ist eine Bibliothek, welche verschiedene Testumgebungen für das bestärkende Lernen bietet und welche auch in dieser Arbeit eingesetzt wird.

Die nächste Anforderung betrifft das Projekt selbst. Dieses soll wie eine Bibliothek eingesetzt werden können und muss somit eine einfache Programmierschnittstelle für den Anwender bieten. Das Ziel ist, dass mit wenig Aufwand verschiedene Optimierungsprobleme umgesetzt und gelöst werden können. Dies soll sowohl für Testumgebungen aus dem OpenAI Gym als auch für andere Quellen gelten.

Eine weitere wichtige Anforderung ist, dass der ganze Optimierungsprozess mithilfe eines *Seeds* beeinflussbar und somit wiederholbar sein muss. Der Grund hierfür ist, dass wie in Kapitel 2.2 und Kapitel 2.3 aufgezeigt, neuroevolutionäre Algorithmen viele Zufallswerte verwenden, die den Ablauf und das erhaltene Ergebnis stark beeinflussen können. Für einen einfachen Vergleich zwischen der sequentiellen und parallelisierten Implementierung ist es notwendig, dass beide Algorithmen exakt denselben Ablauf an Instruktionen und dieselben Zufallswerte haben und somit letztendlich auch dieselbe Lösung liefern. Dies ist für einen sequenziellen Algorithmus einfach umzusetzen, wenn ein globaler Zufallsgenerator mithilfe eines *Seeds* verwendet wird. Bei der Verwendung eines Beowulf Clusters ist dies nicht möglich, da hierbei physisch getrennte Geräte verwendet werden. Somit ist die Herausforderung, dass sowohl der sequenzielle als auch der parallelisierte Algorithmus unabhängig von der Anzahl an verwendeten Geräten unter Verwendung desselben *Seeds* dieselben Lösungen berechnen.

Zwei weitere wichtige Anforderungen betreffen die Performance und Visualisierung. Sowohl in der sequenziellen als auch der parallelisierten Implementierung muss es möglich sein, die benötigte Ausführungszeit zu messen. Die hierbei erhaltenen Ergebnisse sind die Grundlage für den späteren Performance Vergleich. Zusätzlich

sollen die erfassten Werte in entsprechenden Diagrammen visualisiert werden können. Neben den Zeiten sollen auch die durch NEAT erzielten Ergebnisse dargestellt werden. Dies umfasst unter anderem die Struktur der optimierten KNN, die erreichten Fitnesswerte, die Verteilung der Spezies (TODO Eventuell weg?) und in einigen Fällen auch die Lösungsstrategie. Letzteres bezieht sich vor allem auch auf das OpenAI Gym. Die Testumgebungen von diesem können in vielen Anwendungsfällen gerendert werden, was allerdings die Laufzeit enorm erhöht. Deswegen ist es bei dem eigentlichen Optimierungsprozess üblich, hierauf zu verzichten. Das letzte Ziel bezüglich der Visualisierung ist, dass ein gespeichertes Genom nach einem Trainingsverfahren geladen werden kann, hieraus ein KNN gebildet wird und die Interaktion von diesem in der Testumgebung gerendert wird. Hierdurch ist es möglich, die entwickelten Strategien und Fortschritte nachzuverfolgen, ohne die Trainingszeit zu erhöhen.

Hieraus ergibt sich auch die letzte Anforderung, die persistente Speicherung der Ergebnisse. Es soll möglich sein, die Genome, Fitnesswerte und gemessenen Laufzeiten persistent in einer Datei abzuspeichern. Die Speicherung der Genome ist Voraussetzung, dass diese visualisiert oder auch in einer Produktivumgebung eingesetzt werden können. Bezüglich der Fitnesswerte und Laufzeiten wäre es theoretisch möglich, die erstellten Graphen als Bilddatei oder Ähnliches zu speichern. Dies hat den Nachteil, dass die Werte nicht automatisiert ausgelesen werden können und ein genauer Vergleich mit anderen Datensätzen schwierig ist. Aus diesem Grund sollen in dieser Arbeit die Rohdaten gespeichert werden, sodass auch eine nachträgliche Verarbeitung beziehungsweise ein Vergleich mit anderen Durchläufen möglich ist. Neben der generellen Speicherung der finalen Ergebnisse soll auch eine regelmäßige Zwischenspeicherung möglich sein, zum Beispiel nach jeder Generation. Der Grund hierfür ist MPI. In Kapitel 2.4.1 ist erläutert, dass dieses beim Auftreten von Fehlern die Ausführung komplett abbricht und nicht gespeicherte Ergebnisse verloren gehen. Durch die Speicherung der Zwischenergebnisse entsteht kein Datenverlust und somit auch keine Verschwendung der bisher genutzten Rechenzeit.

3.2 Softwarearchitektur

Im Folgenden wird auf die Architektur der Software sowie der Implementierung des sequenziellen Verfahrens eingegangen. Das grundsätzliche Ziel ist, dass so viele Komponenten wie möglich von der sequentiellen und parallelisierten Imple-

mentierung genutzt werden können, sodass einerseits der programmiertechnische Aufwand möglichst gering ist und andererseits ein einfaches Austauschen der beiden Varianten möglich ist. Hierfür wird eine Kombination aus verschiedenen Interfaces sowie Serviceklassen verwendet, auf welche im Rahmen dieses Kapitels genauer eingegangen wird. Zuvor werden noch die verwendeten Datenstrukturen und das Interface für die Bibliothek vorgestellt.

Datenstrukturen

Die verwendeten Datenstrukturen sind für beide Implementierungen identisch und in Abbildung (TODO ABBILDUNG) in einem UML Diagramm dargestellt. Wie zu erkennen ist, besitzen diese keine Funktionen und können so für beide Implementierungen genutzt werden. Die eigentliche Funktionalität wird in den später vorgestellten Serviceklassen umgesetzt.

Die erste Struktur ist das *Genom*, welches aus Kapitel 2.3 bekannt ist und alle Informationen zur Konstruktion eines KNN besitzt. Hierfür werden zwei Listen benötigt, wobei die erste alle Neuronen und die zweite alle Verbindungen des KNN enthält. Diese Repräsentation entspricht der Kodierung aus Kapitel 2.3.1.

Ein Neuron wird durch die Klasse *Node* repräsentiert und enthält unter anderem den *Bias*-Wert und die zu verwendende Aktivierungsfunktion, welche für die Berechnungen im KNN benötigt werden. Zusätzlich sind einige weitere Werte vorhanden, welche die spätere Implementierung der Anforderungen vereinfachen. Hierzu gehört unter anderem der Neuronentyp, welcher angibt, ob es sich um ein *Input*-, *Hidden*- oder *Ouput*-Neuron handelt. Dies vereinfacht die spätere Visualisierung sowie das Einsetzen der Eingabewerte und das Auslesen des Ausgabevektors. Zusätzlich besitzt jedes Neuron eine X-Koordinate, welche die relative X-Position für eine spätere Visualisierung angibt. Die *Input*- und *Output*-Neuronen besitzen die Werte 0 und 1. Die *Hidden*-Neuronen werden, wie in Kapitel 2.3.5 beschrieben, nur durch strukturelle Mutationen hinzugefügt und dabei zwischen zwei anderen Neuronen platziert. Die X-Koordinate von diesen wird berechnet, indem der Mittelwert der zwei anderen Neuronen gebildet wird. Zuletzt enthält jedes Neuron zur Identifizierung eine ID, welche von den Verbindungen genutzt wird.

Eine Verbindung zwischen zwei Neuronen wird im Genom durch eine Instanz der Klasse *Connection* repräsentiert. Jede von diesen enthält zwei Werte, welche die ID des Start- und Zielneurons der Verbindung enthalten. Zusätzlich ist das Gewicht

sowie ein Aktivierungsbit gegeben. Letzteres gibt entsprechend der vorgestellten Kodierung an, ob die Verbindung im KNN enthalten sein soll. Die letzte Variable ist die Innovationsnummer, welche in Kapitel 2.3.3 beschrieben ist. Diese wird durch strukturelle Mutationen zugewiesen und in der Reproduktionsphase zum Lösen des *Competing Conventions* Problems verwendet.

In Kapitel 2.2 ist der Begriff des Individuums erläutert. In dieser Arbeit werden diese durch Instanzen der Klasse *Agent* repräsentiert und sind nach dem Agenten benannt, welcher beim bestärkenden Lernen mit der Umwelt interagiert. Der hierbei erhaltene Fitnesswert, welcher im späteren Verlauf die Selektion maßgeblich beeinflusst, wird in der entsprechenden Variable *fitness* gespeichert. Das Feld *additional_info* kann Zusatzinformationen enthalten, welche vom Optimierungsproblem stammen können.

Die zwei letzten Klassen in diesem Diagramm sind die *Generation* und *Spezies*. Letzteres ist aus dem NEAT Kapitel bekannt und ist eine Gruppierung von ähnlichen Individuen bzw. Agenten. Entsprechend der in Kapitel 2.3.4 enthaltenen Definition ist eine Spezies durch ein Genom repräsentiert und besitzt eine Mitgliederliste. Zusätzlich wird der höchste erreichte Fitnesswert des besten Mitglieds gespeichert sowie die Generation, in welcher dieser Wert erzielt wurde. Dies wird benötigt, um den Fortschritt zu überwachen, da eine Spezies nicht für die Reproduktion ausgewählt wird, deren Mitglieder keine Steigerung des Fitnesswertes in einer festgelegten Anzahl an Generationen erzielen.

Die letzte Klasse ist die *Generation*, welche aus Kapitel 2.2.2 bekannt ist. Diese enthält sowohl eine Liste mit allen Individuen als auch eine der verschiedenen Spezies. Zusätzlich wird diese durch eine Nummer identifiziert, welche angibt, wie viele Zyklen der Evaluation, Selektion, Rekombination und Mutation bereits durchgeführt wurden.

Künstliches neuronales Netz

Aus dem Genom wird ein KNN erstellt, welches dann im Optimierungsproblem eingesetzt wird. In Kapitel 2.1 sind der Aufbau und die Funktionsweise von diesen ausführlich erläutert. Prinzipiell können hierfür verschiedene bereits implementierte Bibliotheken verwendet werden, wie beispielsweise Tensorflow [46] oder PyTorch [47]. Mit diesen können vor allem rein ebenenweise verbundene KNN, wie in Kapitel 2.1.4 vorgestellt, schnell und einfach erstellt werden. Allerdings

können durch NEAT auch KNN mit *shortcut* Verbindungen entstehen sowie Netze mit Rückkopplungen. Zwar können diese auch mit Tensorflow und PyTorch abgebildet werden, aber dies erfordert einen bedeutend höheren Aufwand. Da dies im Rahmen dieser Arbeit nicht möglich ist, wird eine eigene vereinfachte Implementierung verwendet. Ein Nachteil von dieser ist, dass sie im Vergleich zu Implementierungen aus großen Bibliotheken insgesamt langsamer ist. Der Grund hierfür ist, dass die Bibliotheken gut optimiert sind und auch die Verwendung von GPUs unterstützen, welche die benötigte Ausführungszeit stark reduzieren können. Auch wenn diese nicht verwendet werden, wird im Folgenden ein Grundgerüst erstellt, sodass eine spätere Integration einfach möglich ist. Dies wird mit einem Interface realisiert, welches die grundlegenden Funktionen definiert. Sowohl die in dieser Arbeit verwendete Implementierung als auch zukünftige Erweiterungen mit Tensorflow oder ähnlichem können dieses nutzen und ermöglichen somit ein einfaches Austauschen der verschiedenen Implementierungen. Das Interface ist in Abbildung (TODO ABBILDUNG) dargestellt und definiert drei Funktionen mit den Namen *build()*, *reset()* und *activate()*. Bei der *build()* Funktion wird ein Genom übergeben und hieraus ein KNN konstruiert. Die *activate()* Funktion bekommt eine Liste mit Eingabewerte übergeben und produziert den Ausgabevektor des KNN. Die Anzahl an Eingabe- und Ausgabewerten entspricht der Anzahl an *Input*- und *Output*-Neuronen im KNN. Bei der *reset()* Funktion werden eventuell gespeicherte Ergebnisse, wie sie bei einem Netz mit Rückkopplungen enthalten sein können, entfernt und auf den Startwert zurückgesetzt.

Die in dieser Arbeit verwendete *BasicNeuralNetwork* Implementierung setzt dieses Interface mit dem Ziel um, dass sowohl Netze mit und ohne Rückkopplungen umgesetzt werden können. Da eine genaue Beschreibung der Implementierung nicht von Interesse ist, wird an dieser Stelle nur der oberflächliche Ablauf beschrieben. Bei der *build()* Funktion wird je eine Liste für die *Input*- und *Output*-Neuronen angelegt, in welche die Neuronen mit den entsprechenden Typen sortiert werden. Eine dritte Liste enthält alle im KNN enthaltenen Neuronen. Jedes Neuron speichert zudem eine Liste mit den Verbindungen, welche zu ihm führen und den aktuellen sowie letzten berechneten Wert. Die letzte Aufgabe dieser Funktion ist die Festlegung der Reihenfolge, in welcher die einzelnen Neuronen aktiviert bzw. deren Zwischenergebnisse berechnet werden. Bei der Reihenfolge ist zu beachten, dass bei einem Netz ohne Rückkopplung jedes vorherige Neuron bis zu den *Input-Neuronen* bereits aktiviert sein muss, da ansonsten das Ergebnis verfälscht wird. Bei Netzen mit Rückkopplungen gilt diese Anforderung prinzipiell

auch, außer für Verbindungen, welche in derselben Schicht oder von Neuronen der nachfolgenden Schichten ausgehen. Dies sind Rückkopplungen, bei welchen der zuletzt berechnete Wert zurückgeben wird, sodass keine Endlosschleife entsteht.

Bei der *activate()* Funktion wird ein Eingabevektor an das KNN übergeben. Jeder darin enthaltene Wert wird in ein *Input*-Neuron gesetzt. Dies ist mit der zuvor in der *build()* Funktion erstellten Liste mit allen *Input*-Neuronen effizient umzusetzen. Danach beginnt die eigentliche Berechnung des Ausgabevektors. Hierzu werden die Neuronen in der zuvor festgelegten Reihenfolge aktiviert bzw. berechnet. Zuletzt wird der Ausgabevektor erstellt. Auch dies ist effizient umsetzbar, da zuvor ein Liste mit allen *Output*-Neuronen erstellt wurde. Über diese wird iteriert und die entsprechenden Ergebnisse in eine neue Liste kopiert, welche schlussendlich den Ausgabevektor repräsentiert und als Ergebnis der Funktion zurückgegeben wird.

Die *reset()* Funktion iteriert über die Liste mit allen Neuronen und entfernt die gespeicherten Zwischenergebnisse.

Optimierungsproblem

In den vorherigen Kapiteln sind die grundlegenden Datenstrukturen und die Funktionalität des KNN vorgestellt. Bevor der eigentliche Optimierungsalgorithmus implementiert werden kann, wird noch eine grundlegende Komponente benötigt und zwar das Optimierungsproblem. Wie in Kapitel 2.2.2 vorgestellt, können diese aus verschiedenen Domänen stammen und sich sehr unterscheiden. Somit muss auch für dieses ein Interface mit dem Ziel erstellt werden, möglichst viele Szenarien abbilden zu können, welche einfach und mit wenig Aufwand durch das in dieser Arbeit implementierte Verfahren optimierbar sind.

Das Interface ist in Abbildung (TODO ABBILDUNG) dargestellt und zeigt fünf Funktionen, von welchen lediglich die *evaluate()* Funktion immer benötigt wird. Die restlichen vier Funktionen sind optional und können je nach Optimierungsproblem zusätzlich eingesetzt werden. Im Folgenden wird auf die Bedeutung von diesen genauer eingegangen.

Die *initialization()* Funktion kann zum Initialisieren der Umgebung verwendet werden und wird einmalig zu Beginn durch die später vorgestellte Bibliothek aufgerufen. Dies kann beispielsweise verwendet werden, um notwendige Daten

initial aus einer Datenbank zu laden oder um die Umgebung zu erstellen. Ist die Initialisierung abgeschlossen, können prinzipiell verschiedene Agenten in dieser evaluiert werden. Vor jeder Evaluation wird die *before_evaluation()* Funktion aufgerufen, mit welcher zum Beispiel der Zustand der Umgebung zurückgesetzt werden kann, sodass jeder Agent denselben Startzustand vorfindet. Danach wird die eigentliche Evaluation mit der Funktion *evaluate()* durchgeführt. Dies ist die wichtigste Funktion in diesem Interface und muss von jedem Optimierungsproblem implementiert werden. Als Parameter wird ein initialisiertes KNN übergeben, welches mit dem Genom eines Agenten erstellt wurde. Mit diesem soll das Optimierungsproblem gelöst werden. Die genaue Implementierung dieser Funktion kann je nach Problem sehr unterschiedlich sein. Verschiedene Beispiele werden in Kapitel 4 genauer vorgestellt. Bei der Implementierung ist zu beachten, dass die Funktion zwei Rückgabewerte erwartet. Der erste ist vom Typ *float* und repräsentiert den erreichten Fitnesswert. Wie in Kapitel 2.2.2 vorgestellt, bewertet dieser die Leistung des KNN und wird für die spätere Selektionsphase benötigt. Der zweite Rückgabewert ist vom Typ *Dict* und kann verschiedene *Key-Value* Paare mit Zusatzinformationen des Optimierungsproblems enthalten. Diese werden letztendlich im Datenfeld *additional_info* des Agenten gespeichert, welches in Kapitel 3.2 vorgestellt ist. Die darin gespeicherten Daten können für eine spätere Auswertung sowie in der Abbruchbedingung für den Algorithmus verwendet werden. Nach jeder durchgeführten Evaluation wird das Pendant zur *before_evaluation()* Funktion ausführt, dies ist die *after_evaluation()* Funktion. Mit dieser sowie der letzten in diesem Interface enthaltenen Funktion *clean_up()* können verschiedene Tätigkeiten umgesetzt werden, welche Ressourcen freigeben oder den originalen Zustand der Umgebung wiederherstellten. Der Unterschied zwischen diesen Funktionen ist, dass die *clean_up()* Funktion das Pendant zur *initialization()* Funktion ist und nur einmalig am Ende des Trainingsverfahren aufgerufen wird.

Schnittstelle der Bibliothek

Nachdem die für diese Arbeit benötigten Komponenten vorgestellt sind, kann auf die Schnittstelle der Bibliothek genauer eingegangen werden. Ziel ist, dass diese die grundlegenden Funktionen definiert, welche sowohl von der sequentiellen als auch der parallelisierten Implementierung umgesetzt werden. Somit bieten beide Implementierungen dieselben Funktionalitäten, können einfach ausgetauscht werden und ermöglichen einen einfachen Vergleich.

Die Schnittstelle der Bibliothek besteht aus drei Interfaces, die als *NeatOptimizer*, *NeatReporter* und *NeatOptimizerCallback* bezeichnet werden und deren Beziehung zueinander in Abbildung (TODO ABBILDUNG) dargestellt ist. Zu erkennen ist, dass der *NeatOptimizerCallback* von dem *NeatReporter* erbt und somit dessen Funktionalität erweitert. Der *NeatOptimizer* besitzt genau einen *NeatOptimizerCallback* sowie beliebig viele *NeatReporter*. Auf die Funktionalität von diesen wird später genauer eingegangen. Im Folgenden wird zuerst der *NeatOptimizer* betrachtet, welcher fünf Funktionen besitzt, von denen vier für das Hinzufügen und Entfernen von Instanzen der Klasse *NeatReporter* und *NeatOptimizerCallback* verwendet werden. Die letzte Funktion hat den Namen *evaluate()*, startet den Ablauf des Optimierungsproblems und erhält als Parameter die hierfür benötigten Werte. Die ersten beiden Parameter bestimmen die Anzahl der *Input*- und *Output*-Neuronen und somit die Größe des Eingabe- und Ausgabevektors. Da der Optimierungsvorgang in NEAT, wie in Kapitel 2.3.5 beschrieben, mit einer minimalen Struktur beginnt, müssen keine *Hidden*-Neuronen angegeben werden. Der nächste Parameter ist eine Referenz auf eine Aktivierungsfunktion, welche von allen Neuronen verwendet wird. Einige bekannte Vertreter, von denen ein Teil in Kapitel 2.1.3 vorgestellt ist, sind standardmäßig in diesem Projekt enthalten, wobei das Hinzufügen von weiteren Funktionen jederzeit möglich ist. Hierauf folgt der Parameter *challenge*, der eine Klasse repräsentiert, die das Interface des Optimierungsproblems implementiert. Der Algorithmus wird versuchen, das hierin enthaltene Problem zu optimieren. Die letzten beiden Parameter sind als *config* und *seed* bezeichnet. Letzteres soll die Generierung der Zufallswerte beeinflussen und somit den Optimierungsvorgang wiederholbar und vergleichbar machen. Die *config* repräsentiert eine Konfiguration, in welcher verschiedene Parameter des Verfahrens spezifiziert sind. In dieser wird beispielsweise angegeben, wie hoch die Chance auf eine strukturelle Mutation ist oder wie sehr sich die Gewichte der Verbindungen ändern können. Auf die tatsächlich verwendeten Konfigurationen wird im Rahmen der Analyse in Kapitel 4 weiter eingegangen.

Das Ausführen der *evaluate()* Funktion startet den gesamten Optimierungsprozess, welcher je nach Komplexität eine Laufzeit von mehreren Stunden oder Tagen haben kann. Häufig ist es in diesen Anwendungsfällen gewünscht, Zwischenergebnisse und Fortschritte anzuzeigen, sodass die verbleibende Laufzeit und der Erfolg des Verfahrens besser abschätzbar ist. Dies wird in dieser Arbeit durch Callbacks realisiert, welche durch die Interfaces *NeatReporter* und *NeatOptimizerCallback* implementiert werden. Das Interface *NeatReporter* definiert einige Methoden,

welche während der Laufzeit des Algorithmus regelmäßig an bestimmen Punkten aufgerufen werden. Klassen, die diese implementieren, können hierdurch Statusinformationen über den aktuellen Zustand sowie den Fortschritt des Verfahrens erhalten. Dieses Interface wird in dieser Arbeit unter anderem für die regelmäßige Speicherung des besten Agenten und zur Messung der Performance genutzt. Die erste hierbei implementierte Funktion ist die *on_initialization()*, welche einmalig zu Beginn aufgerufen wird. Das Pendant hierzu ist die *on_cleanup()* Funktion, welche einmalig am Ende aufgerufen wird. Auch für die Zwischenzeit gibt es einige Methoden, die den Beginn und das Ende verschiedener Phasen signalisieren. Die Funktionen *on_generation_evaluation_start()* und *on_generation_evaluation_end()* werden zum Beginn und Ende der Evaluationsphase aufgerufen. Als Parameter wird an beide Funktionen die aktuell evaluierte Generation übergeben. Dies kann beispielsweise zum Nachverfolgen des besten und durchschnittlichen Fitnesswertes verwendet werden. Mit den Funktionen *on_agent_evaluation_start()* und *on_agent_evaluation_endt()* wird angegeben, wann die Evaluierung eines einzelnen Agenten beginnt und abgeschlossen ist. An diese Funktion wird der eigentliche Agent sowie ein Index übergeben, welcher angibt, wie viele Evaluationen in dieser Generation bereits durchgeführt wurden. Diese Informationen können beispielsweise für einen Fortschrittsbalken verwendet werden. Die Funktionen *on_reproduction_start()* und *on_reproduction_end()* markieren den Beginn und das Ende der Reproduktionsphase, welche mit der Selektion startet und mit dem Ersetzen der vorherigen Generation durch die neu erstellten Agenten endet. Um die Zeit zu messen, welche für die Rekombination und Mutation eines einzelnen neuen Agenten benötigt wird, sind die Funktionen *on_compose_offsprings_start()* und *on_compose_offsprings_end()* enthalten. Die letzte in diesem Interface enthaltene Funktion heißt *on_finish()* und erhält als Parameter die aktuelle Generation. Sie wird einmalig am Ende der Optimierung aufgerufen nachdem die Abbruchbedingung erfüllt ist. Diese kann verwendet werden, um das beste KNN und die erhaltenen Ergebnisse zu visualisieren und zu speichern.

Die Klasse *NeatOptimizerCallback* erbt von dem Interface *NeatReporter* und kann daher alle bereits vorgestellten Funktionen nutzen. Diese sind für den Programmablauf optional und werden nicht zwingend benötigt. Dies trifft nicht auf die letzte Funktion zu, welche im *NeatOptimizerCallback* zusätzlich implementiert wird. Dies ist die Funktion *finish_evaluation()*, welche die aktuelle Generation als Parameter übergeben bekommt und einen Wert vom Typ *boolean* zurückgeben muss. Hiermit wird die Abbruchbedingung umgesetzt. Die Funktion wird nach

Beendigung der Evaluationsphase mit der aktuellen Generation sowie mit allen erzielten Fitnesswerten aufgerufen. Ist das Ergebnis der Funktion *True*, wird die Ausführung des Algorithmus beendet und die entsprechenden Callback Funktionen *on_finish()* und *on_cleanup()* aufgerufen. Liefert die Funktion *False*, wird mit der Selektion, Rekombination und Mutation fortgefahren und der Zyklus startet erneut. Diese Art der Abbruchbedingung ermöglicht vielfältige Umsetzungen. Zum Beispiel kann der Algorithmus beendet werden, wenn der Fitnesswert eines Agenten einen Schwellwert übersteigt, eine gewisse Anzahl an Zyklen bzw. Generationen durchgeführt ist oder eine festgelegte Trainingszeit überschritten ist.

Serviceklassen

Ein wichtiges Ziel bei der Entwicklung ist das Einsparen von unnötigem Implementierungsaufwand, dies gilt auch für diese Arbeit. Aus diesem Grund wurden die bereits vorgestellten Interfaces erstellt, welche sowohl von der sequenziellen als auch der parallelisierten Implementierung verwendet werden. Der Vorteil hierdurch ist, dass ein einfaches Austauschen der beiden Umsetzungen möglich ist und einen einfachen Vergleich ermöglicht. Die eigentliche Implementierung wird sich zwangsläufig an einigen Punkten unterscheiden. Dennoch wird ein großer Teil der NEAT Komponenten gleich bleiben, da sich die Funktionen nicht durch eine Parallelisierung ändern. Aus diesem Grund wird ein Großteil dieser Funktionen als Serviceklassen implementiert, die keinen internen Zustand besitzen und somit zwei Vorteile bieten. Erstens ist es einfach möglich, die Funktionen in verschiedenen Implementierungen zu nutzen. Zweitens ermöglicht eine solche Struktur ein einfaches automatisiertes Testen der Implementierung. Dies ist besonders wichtig, da Fehler im Trainingsverfahren zu einem späteren Zeitpunkt aufgrund der Größe der Population schwer zu lokalisieren sind.

Insgesamt werden drei Serviceklassen mit den Namen *GenerationService*, *ReproductionService* und *SpeciesService* erstellt, welche Funktionen entsprechend ihrer Benennung übernehmen. Somit befasst sich die erste Klasse mit allen Funktionen, welche die ganze Generation betreffen, die zweite mit Funktionen bezüglich dem Erstellen und Modifizieren von Genomen sowie Agenten und die letzte Klasse mit Funktionen bezüglich der verschiedenen Spezies. Im Folgenden wird auf die enthaltenen Funktionen jeder Serviceklasse genauer eingegangen. Allerdings sei hierzu angemerkt, dass keine genauen Implementierungsdetails vorgestellt werden. Die theoretische Funktionalität ist in den Kapiteln 2.2 und 2.3 erläutert. Für die genaue praktische Umsetzung wird auf den veröffentlichten Programmcode

verwiesen.

Der *ReproductionService* besitzt vor allem Funktionen, um neue Genome mittels Reproduktion zu erzeugen oder bestehende zu mutieren. Die erste Funktion heißt *cross_over()* und setzt die eigentliche Reproduktion um. In ihr werden zwei Elterngenome verwendet, um einen Nachkommen zu erzeugen. Grundsätzlich entsprechen sowohl die Umsetzungen dieser Funktion als auch der anderen in diesem Kapitel vorgestellten Funktionen den Erläuterungen der vorherigen Kapitel. Eine Besonderheit soll an dieser Stelle hervorgehoben werden. Ein wichtiges Ziel für ein wiederholbares Ergebnis ist, dass unabhängig von einem Prozessor mit demselben Seed immer dasselbe Ergebnis erzielt wird. Dies ist mit einem einfachen globalen Zufallsgenerator in einem verteilten System nicht möglich. Aus diesem Grund wird für jedes Genom ein neuer Zufallsgenerator erzeugt, dessen Seed sich aus den beiden Elterngenomen ergibt und welcher für alle Zufallsoperationen verwendet wird, die dieses Genom betreffen. Der hieraus resultierende Vorteil ist, dass unabhängig vom Prozessor und dessen internem Zustand dieselben Eltern immer denselben Nachkommen produzieren. Der hierfür erstellte Zufallsgenerator wird auch für die Funktionen *mutate_weights()*, *mutate_add_connection()* und *mutate_add_node()* verwendet. Die erste Funktion mutiert die Gewichte aller Verbindungen, die zweite fügt eine neue Verbindung und die dritte ein neues Neuron dem Genom hinzu. Bei letzterem werden entsprechend der Definition in Kapitel 2.3.2 zusätzlich zwei neue Verbindungen erstellt.

Die Klasse *GenerationService* besitzt einige Funktionen zum Erzeugen der initialen Population. Wie bei der Schnittstellendefinition beschrieben, wird anfänglich nur die Anzahl an *Input*- und *Output*-Neuronen übergeben. Die Funktion *create_genome_structure()* erzeugt mit diesen Informationen eine Struktur für das KNN, welche die entsprechenden Neuronen und Verbindungen enthält. In der Funktion *create_initial_generation()* kann mit der generierten Struktur letztendlich eine neue Generation erstellt werden. Hierfür wird das Genom für jeden zu erstellenden Agenten einmal kopiert und die Gewichte der Verbindungen zufällig gesetzt. Schlussendlich werden die Agenten noch den Spezies zugeordnet.

Der *SpeciesService* besitzt im Vergleich zum *GenerationService* bedeutend mehr Funktionen. Um die erstellten Agenten den verschiedenen Spezies zuzuordnen, kann die Funktion *sort_agents_into_species()* genutzt werden. Entsprechend der Erläuterung in Kapitel 2.3.4 iteriert diese über die Liste mit allen Agen-

ten und überprüft für jeden die Kompatibilität mit den existierenden Spezies. Der Kompatibilitätswert zwischen zwei Genomen kann mit der Funktion *calculate_genetic_distance()* berechnet werden. Ist dieser kleiner als ein konfigurierter Schwellwert, wird der Agent der Spezies zugeordnet. Existiert keine passende Spezies, wird eine neue erstellt. Dies sind aber nicht die einzigen Funktionen. In Kapitel 2.3.4 ist das *explicit fitness sharing* Verfahren eingeführt worden, welches mit der Funktion *calculate_adjusted_fitness()* umgesetzt ist. Der angepasste Fitnesswert wird schließlich bei der Selektion verwendet, welche mit den Funktionen *calculate_amount_offspring()* und *create_offspring_pairs()* umgesetzt ist. Die erste Funktion berechnet, wie viele Nachkommen jede Spezies erhalten soll. Die zweite Funktion erstellt die tatsächlichen Elternpaare, welche zum Erzeugen der Nachkommen verwendet werden. Zuletzt sind noch einige kleinere Funktionen enthalten. Bei NEAT können nur die besten 50% der Genome zur Reproduktion ausgewählt werden, die Funktion *remove_low_genomes()* entfernt die restlichen. Beim NEAT Algorithmus wird nach jeder Generation ein neues Genom zum Repräsentieren jeder Spezies ausgewählt. Dies ist in diesem Projekt mit der Funktion *select_new_representative()* möglich. Die letzten beiden Funktionen heißen *reset_species()* und *get_species_with_members()*. Die erste hiervon entfernt alle Mitglieder einer Spezies. Dies wird typischerweise durchgeführt bevor die neu erstellten Agenten den Spezies zugewiesen werden. Nach der Zuweisung wird die zweite Funktion aufgerufen. Diese filtert Spezies heraus, welche keine Mitglieder zugewiesen bekommen haben und entfernt diese aus der Liste.

Performance Messung

Es gibt verschiedene Kriterien um die Performance eines neuroevolutionären Algorithmus zu beurteilen. In der Literatur, so auch in Quelle [28], wird häufig die Anzahl an Generationen in Kombination mit der Populationsgröße angegeben bis eine Lösung für ein Optimierungsproblem gefunden wurde. Hiermit kann die Anzahl der evaluierten KNN abgeleitet werden und ein Vergleich zu anderen Verfahren ist möglich. In vielen Fällen ist dies sinnvoller, als ein direkter Vergleich der tatsächlichen Laufzeiten, da diese sehr von der verwendeten Hardware, Programmiersprache und der effizienten Implementierung abhängig sind beziehungsweise beeinflusst werden können. Auch in dieser Arbeit wird diese Art der Performance Messung mit der Klasse *FitnessReporter* durchgeführt, welcher zwei Aufgaben erfüllt. Diese erbt von der bereits vorgestellten Klasse *NeatReporter* und hat somit Zugriff auf die Funktion *on_generation_evaluation_end()*. Der *FitnessReporter* legt für jede Generation einen Datensatz an, in welcher der durchschnittliche

und beste Fitnesswert gespeichert werden. Hierdurch ist einerseits ersichtlich, wie viele Generationen der Algorithmus benötigt und zusätzlich kann der Fortschritt zwischen den einzelnen Generationen betrachtet werden. Dies kann wertvolle Erkenntnisse liefern, wenn der Algorithmus schlechter oder langsamer ist als erwartet.

In dieser Arbeit spielt auch die tatsächlich benötigte Ausführungszeit eines Algorithmus, welche auch als *wall clock time* bezeichnet wird, eine große Rolle. Im Rahmen dieser Arbeit werden die Zeiten erfasst, und dienen als Basis für den Vergleich zwischen der sequenziellen und parallelisierten Implementierung. Hierfür wird die Klasse *TimeReporter* erstellt, welche ebenfalls das Interface *NeatReporter* implementiert und Zugriff auf die verschiedenen bereits vorgestellten Funktionen besitzt. Mit diesen können die benötigten Ausführungszeiten für verschiedene Phasen des Algorithmus erfasst werden. Auf die genaue Unterteilung wird im Rahmen von Kapitel 4 genauer eingegangen.

Visualisierung

Die verschiedenen erfassten Messwerte sollen für eine bessere und einfachere Auswertung graphisch dargestellt werden können. Die beste Fitnesswerte pro Generation soll in einem Liniendiagramm und die erfassten Ausführungszeiten in einem Säulendiagramm dargestellt werden. Bei letzterem sollen die verschiedenen Phasen gestapelt sein. Dies hat den Vorteil, dass sowohl das Verhältnis der verschiedenen Phasen zueinander als auch die gesamte Ausführungszeit über mehrere Generationen hinweg ausgewertet werden kann. Für diese Diagramme soll das Paket Matplotlib für die Sprache Python verwendet werden [49]. Dieses ermöglicht ein einfaches und schnelles erstellen von diversen Diagrammtypen und ist zusätzlich sehr gut in die Entwicklungsumgebung Pycharm integriert, welche im Rahmen dieser Arbeit verwendet wird.

Zusätzlich zu den Diagrammen sollen auch die erstellten KNN visualisiert werden, was eine größere Herausforderung darstellt. Eine mögliche Umsetzung welche in diesem Rahmen evaluiert wird, nutzt die Software Grapviz [50]. Diese besitzt eine eigene Beschreibungssprache, welche als *DOT* bezeichnet wird und die Beschreibung von verschiedene Graphentypen ermöglicht. Würde ein KNN in dieser Arbeit mit Graphviz visualisiert werden, müssten die im Genom kodierten Informationen im *DOT* Format in eine Textdatei exportiert werden und erst danach kann ein PDF oder ähnliches mithilfe von Graphviz erstellt werden. Da dieser Vorgang relativ aufwändig ist, wird eine alternative Lösung mithilfe des Pythonpakets

NetworkX umgesetzt [51]. Dieses wird normalerweise primär für die Analyse von Graphen eingesetzt, aber ermöglicht auch die Visualisierung von diesen. Als Basis hierfür verwendet das Paket entweder die vorgestellte Software Graphviz oder alternativ Matplotlib. letzteres bietet sich mehr an, da es bereits in diesem Projekt verwendet wird und eine bessere Integration in die Entwicklungsumgebung bietet. Allerdings müssen auch bei diesem Verfahren die im Genom enthaltenen Informationen in einem gewissen Format an das Paket übergeben werden. Eine weitere Herausforderung bei der Implementierung ist, dass die Knoten des Graphen, in diesem Fall die Neuronen, normalerweise eine zufällige Position zugewiesen bekommen und daher keine Darstellung der Schichten möglich ist. Um diese zu erhalten müssen sämtliche Positionen manuell gesetzt werden. Hierbei ist das Bestimmen der X-Koordinate einfach, da diese für jedes Neuron bereits im Genom gespeichert ist. Die Y-Koordinate wird dann in Abhängigkeit von der Anzahl an weiteren Neuronen in derselben Schicht bestimmt.

Persistenz

Die persistente Speicherung der Optimierungsergebnisse ist die letzte Anforderung für diese Arbeit. Dies umfasst sowohl die Speicherung der Genome sowie die Ergebnisse der Performance Messung. Für die Speicherung der Daten wurden prinzipiell zwei Ansätze evaluiert. Beim ersten Ansatz würde eine klassische SQL- oder NoSQL-Datenbank zur Speicherung eingesetzt werden. Der Vorteil von einem solchen System ist, dass viele Nutzer gleichzeitig die Daten lesen können. Allerdings ist dies für den gegebenen Anwendungskontext nicht notwendig und der Aufwand, welcher durch die Implementierung der verschiedenen Anfragen entsteht, ist im Vergleich zur zweiten Variante bedeutend höher. Zusätzlich muss ein dauerhaft verfügbarer Datenbankservice zur Verfügung gestellt werden, sodass die Ergebnisse mit anderen Nutzern teilbar sind.

Der zweite Ansatz ist bedeutend einfacher. Das Ziel ist, dass die Ergebnisse nur in einer lokalen Datei gespeichert beziehungsweise aus einer solchen Datei geladen werden können. Ein solches Vorgehen bietet einige Vorteile. Der erste ist, dass ein solches Verfahren mit wenig Aufwand implementierbar ist. Das standardmäßig in Python enthaltene Pickle Modul erfüllt genau diese Voraussetzungen. Dieses bietet sowohl vorgefertigte Funktionen zum serialisieren von unterschiedliche Python Objekte welche dann in einer Datei gespeichert werden können sowie Funktionen zum Laden von solchen Datensätzen. Zwei Vorteile von diesem Verfahren sind, dass der programmiertechnische Aufwand sehr gering ist und dass

trainierte Modelle als Datei vorliegen wodurch sie auf Github oder ähnlichem einfach veröffentlicht werden können.

Das eigentliche Speichern soll auf jeden Fall am Ende das Trainingsverfahrens durchgeführt werden. Allerdings kann es zusätzlich sinnvoll sein, in regelmäßigen Abständen die bis dahin erhaltenen Zwischenergebnisse ebenfalls zu sichern. Hierfür gibt es verschiedene Gründe. Bei einer ungünstigen Konfiguration kann ein Trainingsverfahren eventuell niemals die Abbruchbedingung erfüllen. In diesem Fall müsste das Programm manuell abgebrochen werden wobei sämtliche Ergebnisse verloren gehen würden. Auch bei Implementierungsfehler können zu einem Absturz führen, bei welchem nicht die finalen Ergebnisse gespeichert werden. Bei der parallelisierten Implementierung können zusätzlich Hardware- und Netzwerkfehler auftreten für die MPI, wie in Kapitel 2.4.1 beschrieben, anfällig ist und ebenfalls zu einem Absturz der Anwendung führen. Aus diesem Grund wird die Klasse *CheckPointReporter* implementiert, welche ebenfalls das Interface *NeatReporter* umsetzt. Hierbei werden die Funktionen *on_generation_evaluation_end()* und *on_finish()* implementiert. Bei Aufruf von letzterem wird das Ergebnis immer gespeichert. Bei der *on_generation_evaluation_end()* ist es abhängig von der gewählten Konfiguration. In dieser wird spezifiziert, nach wie vielen Generationen ein Zwischenergebnis abgespeichert werden soll. Prinzipiell ist es möglich sowohl nach jeder Generation die Ergebnisse zu speichern oder gar nicht, wobei letzteres nicht zu empfehlen ist.

3.3 Sequenzielle Implementierung

Im vorherigen Kapitel sind die verschiedenen Komponenten dieser Arbeit vorgestellt, welche zur Implementierung des sequenziellen Verfahrens verwendet werden. Der grundsätzliche Ablauf entspricht hierbei den Erläuterungen der Kapiteln 2.2 und 2.3. Daher wird in diesem Kapitel hauptsächlich auf die technische Umsetzung sowie das Zusammenspiel der bereits vorgestellten Komponenten eingegangen.

Die *evaluate()* Funktion ist, wie in Kapitel 3.2 beschrieben, der Einstiegspunkt für die Bibliothek. Ein Überblick über diese ist im Sequenzdiagramm in Abbildung (TODO Abbildung) gegeben. Initial werden die bereits spezifizierten *NeatReporter* und der *NeatOptimizerCallback* mit der Funktion *on_initialization()* über den Beginn des Verfahrens informiert. Direkt im Anschluss wird auch das Optimierungsproblem initialisiert und die initiale Generation erstellt. Danach beginnt

der Zyklus aus Evaluation, Selektion, Rekombination und Mutation. Hierfür wird zuerst die *evaluate_generation()* Funktion aufgerufen, deren Implementierung im weiteren Verlauf noch genauer erläutert wird. Grundsätzlich werden die verschiedenen Agenten im Optimierungsproblem evaluiert und erhalten einen Fitnesswert. Nach Abschluss dieser Phase wird überprüft, ob das Verfahren terminiert werden soll, indem die Funktion *finish_evaluation()* des *NeatOptimizerCallbacks* aufgerufen wird. Ist dies der Fall, wird die Schleife abgebrochen, die finale Generation zurückgegeben und die *NeatReporter* sowie der *NeatOptimizerCallback* über hierüber informiert. Andernfalls wird mit der Funktion *build_new_generation()* eine neue Population durch die Phasen Selektion, Rekombination und Mutation erstellt. Auch hierauf wird in diesem Kapitel noch genauer eingegangen.

Der Ablauf der *evaluate_generation()* Funktion ist genauer in Abbildung (TODO ABBILDUNG) dargestellt. Am Anfang werden die *NeatReporter* und der *NeatOptimizerCallback* über den Beginn der Funktion informiert. Im Anschluss wird über alle Agenten iteriert und für jeden die Evaluation durchgeführt. Dies umfasst das Aufrufen der entsprechenden Funktionen im *NeatOptimizerCallback* und den *NeatReportern*, sowie das Erstellen und Evaluieren KNN.

Abbildung (TODO Abbildung) zeigt den Ablauf der *build_new_generation()* Funktion. In dieser sind die meisten Funktionen von NEAT enthalten und dementsprechend aufwendig ist die Implementierung. Zu Beginn werden wie bei der *evaluate_generation()* Funktion die *NeatReporter* und der *NeatOptimizerCallback* über den Beginn dieser Phase informiert. Danach werden entsprechend der originalen Publikation von NEAT die besten Genome jeder Spezies mit mehr als x Mitglieder selektiert und unverändert in die nächste Generation kopiert, wobei x ein konfigurierbarer Schwellwert ist. Im Anschluss wird über jede Spezies iteriert und der höchste erreichte Fitnesswert aktualisiert, sofern sich dieser in der letzten Generation geändert hat. Dies ist für den nächsten Schritt wichtig, da bei diesem die Spezies entfernt werden, welche keine Fitnesssteigerung in den letzten t Generationen erzielt haben, wobei auch t konfigurierbar ist. Die Mitglieder von diesen können in der nachfolgenden Selektion nicht als Elternteile ausgewählt werden.

Für die verbleibenden Spezies wird im Anschluss der angepasste Fitnesswert berechnet und auf Basis von diesem im darauffolgenden Schritt die Anzahl an Nachkommen bestimmt. Bevor es zur eigentlichen Selektion kommt, werden noch

die Agenten entfernt, welche im vorherigen Durchlauf schlechte Fitnesswerte erzielt haben. Diese können somit nicht für die Reproduktion ausgewählt werden. Nachdem diese Schritte durchgeführt sind, kann die eigentliche Selektion mit der Funktion *create_offspring_pairs()* beginnen. Im Rahmen dieser Funktion wird über jede Spezies iteriert und für jeden zu erzeugenden Nachkommen zwei Elternteile zufällig ausgewählt, welche als Paar in einer Liste zwischengespeichert werden. Anhand dieser Liste wird die Rekombination und Mutation durchgeführt. Durch die Funktionen *cross_over()* wird die Rekombination implementiert, welche das Genom für den Nachkommen erzeugt. Anschließend wird das Genom mutiert. Hierfür werden die Funktion *mutate_weights()*, *mutate_add_node()* und *mutate_add_connection()* nacheinander aufgerufen.

Bevor das Sortieren der neu erstellten Agenten in die Spezies beginnen kann, müssen noch zwei weitere Funktionen ausgeführt werden. Zuerst wird für jede Spezies die *select_new_representative()* Funktion aufgerufen, welche ein Mitglied zufällig auswählt und dessen Genom als Repräsentant setzt. Danach werden durch den Aufruf der Funktion *reset_species()* alle bisherigen Mitglieder entfernt. Erst an dieser werden die neuen Agenten auf Basis ihres Kompatibilitätswertes tatsächlich den verschiedenen Spezies zugeordnet. Als letzte Aktionen der *build_new_generation()* Funktion werden die Spezies herausgefiltert, welche keine Mitglieder erhalten haben. Danach wird die neue Generation erstellt und als Ergebnis der Funktion zurückgegeben.

4 Analyse

Auf Basis der in Kapitel 3.3 vorgestellten sequenziellen Implementierung wird eine Analyse des Verfahrens durchgeführt. Dafür wird in Kapitel 4.2 die korrekte Funktionalität der Implementierung überprüft. In Kapitel 4.3 wird das implementierte Verfahren auf verschiedene Optimierungsprobleme angewendet und ausgewertet. Hierbei wird insbesondere die Ausführungszeit betrachtet, da diese mit dem parallelisierten Verfahren reduziert werden soll. Zuletzt werden in Kapitel 4.4 die Ergebnisse zusammengefasst.

4.1 Testumgebung

Bevor die verschiedenen Optimierungsprobleme getestet werden, wird an dieser Stelle zunächst auf die Testumgebung eingegangen. Hierfür sind verschiedene Anforderungen zu definieren. Da in dieser Arbeit die Ausführung auf einem verteilten System betrachtet werden soll, müssen grundsätzlich mehrere Geräte zur Verfügung stehen, welche über ein Netzwerk miteinander verbunden sind. Ein aussagekräftiger Vergleich zwischen der parallelisierten und sequenziellen Implementierung ist einfacher, wenn alle Geräte des Clusters dieselbe Hardware verwenden. Bieten diese dieselbe Leistung, steht dem Cluster mit der doppelten Anzahl an Geräten auch die doppelte Rechenkapazität zur Verfügung. Als letzte Anforderung muss das System sowohl in der Anschaffung als auch im Betrieb kostengünstig sein.

Ein Ansatz zur Umsetzung einer solchen Testumgebung ist die Nutzung von mehreren virtuellen Servern. Für diese gibt es zahlreiche kostengünstige Angebote von verschiedenen Anbietern. Allerdings kann bei solchen Systemen nicht auf die zugrunde liegende Hardware Einfluss genommen werden, wodurch ein Vergleich der verschiedenen Implementierungen schwierig sein kann. Aus diesem Grund ist dieser Ansatz für den beschriebenen Anwendungskontext ungeeignet und wird nicht weiter verfolgt. Eine Alternative ist die Anschaffung mehrerer physischer Geräte, welche dieselben Komponenten verwenden. Wie in Kapitel 2.4.1 beschrieben, kann mit diesen ein Beowulf Cluster erstellt werden, der das

kostengünstige Testen von Anwendungen im Bereich HPC ermöglicht. Da in dieser Arbeit primär der Vergleich zwischen der sequenziellen und parallelisierten Implementierung untersucht wird, ist die absolute Leistung nicht das entscheidende Kriterium. Der Fokus liegt auf den Anschaffungs- und Betriebskosten sowie der platzsparenden Unterbringung. Hierfür bieten sich Raspberry Pis besonders gut an.

In diesem Projekt wird als Basis das Modell 4 mit insgesamt 4GB RAM verwendet, auf welchen standardmäßig ein 64bit Quad-core ARM Prozessor verbaut ist [52]. Zusätzlich kann der später erstellte Cluster von der Gigabit Ethernet Verbindung profitieren, welche in diesem Modell neu hinzugefügt wurde und eine ausreichend große Bandbreite für eine effiziente Kommunikation bietet. Die Gesamtkosten für einen Raspberry Pi mit dieser Konfiguration belaufen sich zum Zeitpunkt der Arbeit auf ca. 60 Euro. Als Betriebssystem wird Raspian (Version 10) verwendet. Zwar ist dieses Standardbetriebssystem nur in einer 32bit Variante verfügbar und dadurch in vielen Fällen langsamer als andere 64bit Betriebssysteme, dennoch überwiegen die Vorteile für diese Arbeit. Raspian ist weit verbreitet und bietet eine hohe Kompatibilität zu diversen Softwarepaketen, wie zum Beispiel Tensorflow, von welchen zukünftige Erweiterungen profitieren können. Für die eigentliche Ausführung des implementierten Verfahrens wird Python in der Version 3.7.3 verwendet. Die erforderlichen Bibliotheken sind in der Datei *requirements.txt* im Projekt spezifiziert und in einer virtuellen Umgebung installiert. Mit diesem Aufbau wird die Analyse des sequenziellen Verfahrens durchgeführt.

4.2 Verifizierung der Funktionalität

Bevor die Analyse aufwendiger Optimierungsprobleme durchgeführt wird, soll die korrekte Funktionalität der sequenziellen Implementierung anhand eines einfachen Beispiels getestet werden. Hierbei liegt der Fokus vor allem auf den strukturellen Mutationen, welche essenziell für größere Probleme sind. Durch eine fehlerhafte Implementierung kann beispielsweise das langfristige Integrieren von neuen erforderlichen Strukturen fehlschlagen. Auch besteht die Möglichkeit, dass ein lokales Maximum durch einen Agenten gefunden wird, dessen Genom dann die restliche Population dominiert, mit der Folge, dass keine neuen Lösungsansätze entwickelt werden können.

Um solche Fehler zu entdecken und die korrekte Funktionalität des Algorithmus zu verifizieren, bietet sich das XOR-Problem besonders gut an. Zusätzlich

kann ein Vergleich der erhaltenen Ergebnisse mit der originalen Implementierung durchgeführt werden, da auch die Funktionalität von dieser mit dem XOR-Problem verifiziert wurde und die Ergebnisse in Quelle [28] veröffentlicht sind. Die klassische XOR-Funktion erhält zwei binäre Eingabewerte und erzeugt einen Ausgabewert. Dieser nimmt den Wert 1 an, wenn genau einer der beiden Eingabewerte 1 ist. Andernfalls gibt die Funktion den Wert 0 zurück. Die Abbildung (TODO ABBILDUNG) zeigt links alle möglichen Kombinationen mit zwei Eingabewerten sowie die dazugehörigen Ergebnisse. Rechts in der Abbildung sind die Eingabe- und Ausgabewerte zweidimensional dargestellt. Ziel des XOR-Problems ist, dass ein neuronales Netz für jedes mögliche Paar der Eingabewerte den richtigen Ausgabewert berechnet bzw. den Klassen 0 und 1 zuordnet. Durch die Abbildung (TODO ABBILDUNG) ist erkennbar, dass das Unterteilen der Ausgabewerte in die zwei Klassen 0 und 1 nicht mit einer linearen Funktion möglich ist. Diese Eigenschaft macht die XOR-Funktion für die Verifizierung der Funktionalität von NEAT besonders geeignet. Wie in Kapitel 2.1.4 beschrieben, kann ein KNN ohne *Hidden*-Neuronen nur eine lineare Funktion abbilden und ist somit nicht in der Lage, das XOR-Problem zu lösen. Dies trifft auch auf die KNN der initialen Population von NEAT zu, welche mit einer minimalen Struktur beginnen, die nur aus *Input*- und *Output*-Neuronen besteht. Somit müssen für das erfolgreiche Lösen des XOR-Problems neue Neuronen und Verbindungen erfolgreich in die Population integriert und optimiert werden.

4.2.1 Implementierung

In Abbildung 4.2.1 ist die Implementierung dieses Optimierungsproblems abgebildet, welche im Folgenden genauer erläutert wird. Zu erkennen ist, dass die Implementierung nur wenige Zeilen Programmcode benötigt, was die einfache Nutzung der Bibliothek für verschiedene Optimierungsprobleme verdeutlicht. Zu Beginn des Optimierungsproblems ist eine Liste definiert, welche die verschiedenen Kombinationen der Ein- und Ausgabewerte des XOR-Problems enthält. Danach ist die *evaluate()* Funktion implementiert, welche als Parameter ein initialisiertes KNN übergeben bekommt und mit diesem versucht, das XOR-Problem zu lösen. Das KNN besitzt entsprechend der später vorgestellten Konfiguration zwei Eingabewerte und einen Ausgabewert. Zum Lösen des Optimierungsproblems wird über die verschiedenen Kombinationen von Eingabewerten iteriert und für jeden Eintrag das KNN einmal aktiviert. Das Ergebnis der Aktivierung ist eine Liste, welche in diesem Fall entsprechend der Anzahl an Ausgabeneuronen nur einen Wert enthält und zwar das Ergebnis des KNN für die XOR-Funktion. Bei dieser

```
class OptimizationProblemXOR(OptimizationProblem):

  xor_tuples = [[0, 0, 0], [1, 0, 1], [0, 1, 1], [1, 1, 0]]

  def evaluate(self, neural_network) -> (float, Dict[str,
                                object]):
    fitness_val = 4.0

    for xor_input in ChallengeXOR.xor_tuples:
      inputs = [xor_input[0], xor_input[1]]
      result_array = neural_network.activate(inputs)
      result = result_array[0]

      fitness_val -= (abs(xor_input[2] - result))


    # Square remaining fitness
    fitness_val = fitness_val ** 2

    return fitness_val, None
```

Abbildung 4.1: Implementierung des XOR-Problems in Python

Art der Implementierung wird davon ausgegangen, dass eine Sigmoidfunktion für die Neuronen verwendet wird, sodass das Ergebnis zwischen 0 und 1 liegt. Allerdings ist der Ausgabewert allein nicht ausreichend. Wie in den vorherigen Kapiteln beschrieben muss die *evaluate()* Funktion den Fitnesswert berechnen und diesen als Ergebnis zurückgeben. Im nächsten Schritt muss somit die Fitnessfunktion entwickelt werden, welche in diesem Beispiel der Funktion aus Quelle [28] entspricht. Zu Beginn wird der Fitnesswert entsprechend der Anzahl an Berechnungen auf 4 initialisiert. Nach jeder Aktivierung wird die Differenz zwischen dem erwarteten und tatsächlich erhaltenen Ergebnis berechnet und anschließend vom Fitnesswert subtrahiert. Da die Differenz aufgrund der Aktivierungsfunktion und der Ausgabewerte zwischen 0 und 1 liegen muss, ist der Fitnesswert am Ende der Schleife mindestens 0 und maximal 4. Der verbleibende Wert wird danach quadriert, sodass ein besserer Agent einen proportional höheren Fitnesswert erhält [28]. Am Ende wird dies als Ergebnis der Funktion zurückgegeben. Wie aus der Funktionssignatur zu entnehmen ist, kann optional zusätzlich ein *Dict* übergeben werden, welches Zusatzinformationen enthält. In diesem kann beispielsweise ein *boolean* Wert enthalten sein, welcher anzeigt, ob das KNN für alle Eingaben den korrekten Wert berechnet hat. Dieser Wert kann dann beispielsweise für die Abbruchbedingung verwendet werden.

4.2.2 Parametrisierung und Ergebnisse

Die grundsätzliche Implementierung für das XOR-Optimierungsproblem ist im vorherigen Kapitel beschrieben. Bevor das Verfahren durchgeführt wird, sind noch einige grundlegende Parameter zu spezifizieren, welche sich in diesem Beispiel an Quelle [28] orientieren. Wie bereits beschrieben besitzt jedes KNN entsprechend dem Optimierungsproblem zwei Input-Neuronen und ein Output-Neuron. Zusätzlich verwenden alle Neuronen als Aktivierungsfunktion eine modifizierte Sigmoidfunktion, deren Ausgabewert mit $f_{act}(net_j) = \frac{1}{1+e^{-4.9 \cdot net_j}}$ berechnet wird. Des Weiteren ist die Abbruchbedingung zu bestimmen. Das Optimierungsverfahren wird beendet, wenn ein KNN für alle vier Eingabekombinationen den richtigen Ausgabewert erzeugt. Hierbei wird das Ergebnis als korrekt gewertet, wenn der Ausgabewert o für alle Kombinationen, die 1 ergeben sollen, $o \geq 0.5$ ist. Dementsprechend muss für alle Kombinationen, die 0 ergeben sollen, die Bedingung $o < 0.5$ zutreffen. Zusätzlich wird in diesem Durchlauf eine Populationsgröße von 150 Agenten angenommen. Die Koeffizienten für die in Kapitel 2.3.4 vorgestellte Kompatibilitätsfunktion sind $c_1 = 1.0$, $c_2 = 1.0$ und $c_3 = 0.4$. Ein Genom wird einer Spezies zugeordnet, wenn die berechnete Kompatibilität kleiner als der Schwellwert $\delta_t = 3.0$ ist. Auch der in Kapitel 2.2.2 vorgestellte Elitismus wird in dieser Arbeit implementiert. Der beste Agent jeder Spezies, welche mehr als 5 Mitglieder besitzt, wird unverändert in die nächste Generation kopiert. Des Weiteren erhält eine Spezies nur Nachkommen zugewiesen, wenn in den letzten 15 Generationen eine Steigerung des maximal erreichten Fitnesswertes erzielt wurde. Zuletzt sind noch die Wahrscheinlichkeiten für die Mutation und Rekombination zu bestimmen. Es besteht für jedes Verbindungsgewicht eine Wahrscheinlichkeit von 80%, dass dieses mutiert wird. In diesem Fall wird das Gewicht in 10% der Fälle zufällig neu gewählt, andernfalls wird eine Gauss-Mutation durchgeführt, bei welcher ein Zufallswert auf das Gewicht addiert wird. Zusätzlich besteht für jedes Genom eine Wahrscheinlichkeit von 3%, dass ein neues Neuron hinzugefügt wird. Die Wahrscheinlichkeit für eine neue Verbindung liegt bei 5%. Bezüglich der Rekombination besteht eine Chance von 25%, dass eine Verbindung, welche in beiden Elternteilen deaktiviert ist, im Nachkommen wieder aktiviert wird.

Mit diesen Parametern wird das XOR-Problem 100 mal nacheinander durchgeführt und die Anzahl an Generationen gemessen, welche zum Lösen des Problems benötigt werden. Die Ergebnisse zeigen, dass im Durchschnitt nach 38 Generationen ein KNN das Optimierungsproblem erfolgreich lösen kann. Im besten Durchlauf wurden 5 Generationen, im längsten Durchlauf 98 Generationen benötigt. Aus

diesen Ergebnissen ist erkennbar, dass sich die Werte sehr unterscheiden können und dennoch sind in allen 100 Durchläufen gültige Lösungen gefunden worden. Dass dieses Verhalten nicht ungewöhnlich ist, zeigt ein Vergleich der Ergebnisse mit denen aus der Publikation in Quelle [28]. Mit der originalen Implementierung werden durchschnittlich 32 Generationen und im längsten Durchlauf 90 Generationen zum Lösen des Optimierungsproblems benötigt. Zwar sind die erhaltenen Werte in dieser Arbeit etwas schlechter, die Unterschiede sind jedoch minimal und aufgrund der hohen Varianz der Ergebnisse vernachlässigbar.

Das Python Paket mit dem Namen neat-python aus Quelle [30] ist eine weit verbreitete Implementierung des NEAT Algorithmus und verwendet eine andere Konfiguration für das XOR-Problem. Zusätzlich sind in dieser Implementierung einige Anpassungen durchgeführt worden, welche eine bessere Performanz bieten sollen. Beispielsweise wird bei der Berechnung des Kompatibilitätswertes standardmäßig die Anzahl an *excess genes* und *disjoint genes* durch den Wert N dividiert. Bei kleinen KNN ist dieser Wert 1 und andernfalls die Größe des KNN. Bei der neat-python Implementierung hingegen wird immer durch die Größe des KNN dividiert. Zusätzlich ermöglicht diese Implementierung das Nutzen von negativen Fitnesswerten und verwendet bedeutend höhere Wahrscheinlichkeiten für die strukturellen Mutationen. Diese Anpassungen werden für die Implementierung dieser Arbeit übernommen und das XOR-Problem erneut getestet. Die Wahrscheinlichkeit, dass eine neue Verbindung hinzugefügt wird, liegt bei 50% anstelle von 5% bei der originalen Publikation. Ein neues Neuron wird mit einer Wahrscheinlichkeit von 20% hinzugefügt. Zuletzt wird der Faktor c_3 der Kompatibilitätsfunktion auf 0.5 erhöht und die Aktivierungsfunktion geändert. Bei verschiedenen Tests wurden die besten Ergebnisse mit der tanh Funktion erzielt. Das Problem wird erneut 100 mal nacheinander evaluiert und die Anzahl an Generationen zur Lösung gemessen. Mit der vorgestellten Konfiguration werden durchschnittlich 18 Generationen benötigt bis ein Agent das XOR-Problem erfolgreich löst. Im besten Durchlauf hat ein Agent bereits nach 4 Generationen eine Lösung gefunden, beim schlechtesten erst nach Generation 49. Diese Ergebnisse sind im Vergleich zur originalen Implementierung besser. Daher werden die vorgenommenen Änderungen bei den nachfolgenden Verfahren beibehalten.

Zuletzt soll die tatsächlich erhaltene Lösung genauer betrachtet werden. Im Durchschnitt hat NEAT Lösungen mit 2 oder 3 *Hidden*-Neuronen entwickelt. Allerdings sind auch einige KNN entstanden, welche nur ein *Hidden*-Neuron und

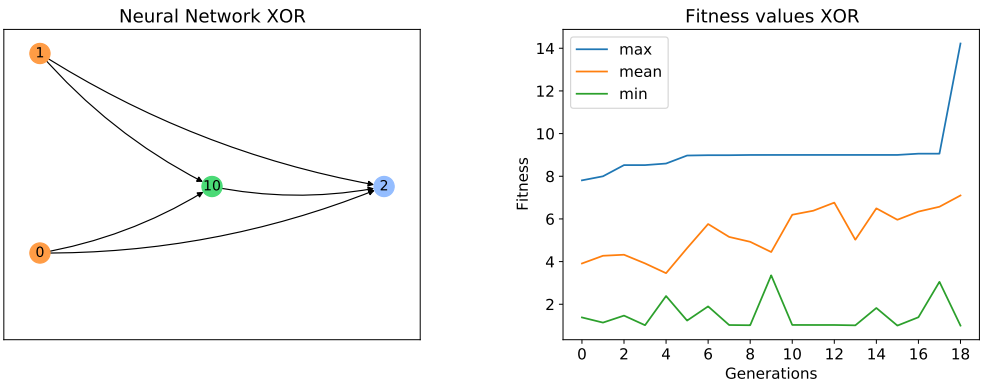

Abbildung 4.2: Links die Lösung für das XOR-Problem mit einem *Hidden*-Neuron, rechts die dazugehörigen Fitnesswerte pro Generation

somit eine minimale Struktur zum Lösen des XOR-Problems besitzen. Ein solch evaluiertes KNN ist in Abbildung 4.2 links dargestellt. Die eigentliche Abbildung ist mithilfe der implementierten Visualisierung erstellt worden, welche die Pakete NetworkX und Matplotlib nutzt. Bei solchen Darstellungen ist zu beachten, dass die orangenen Neuronen vom Typ *Input*, die grünen vom Typ *Hidden* und die blauen vom Typ *Output* sind. Die Zahlen auf den Neuronen repräsentieren die jeweilige ID, während die Pfeile die Verbindungen zwischen den Neuronen darstellen. Für eine bessere Übersichtlichkeit wird auf die Darstellung der Gewichte verzichtet. Prinzipiell kann eine solche Abbildung auch gestrichelte Verbindungen enthalten. Diese sind im Genom deaktiviert und werden nicht für die Berechnungen verwendet. In der Abbildung rechts ist der Verlauf des maximalen, minimalen und durchschnittlichen Fitnesswertes für jede Generation dargestellt. Auch diese Abbildung wird mit Matplotlib im Rahmen des implementierten *FitnessReporters* automatisch erstellt und zeigt einige interessante Eigenschaften. Der durchschnittliche Fitnesswert steigt trotz einiger Einbrüche relativ kontinuierlich an. Hieraus lässt sich schließen, dass auch die Population im ganzen Fortschritte erzielt. Der maximale Fitnesswert hingegen stagniert für einige Generationen beim Wert 9. Der Grund hierfür liegt in der Fitnessfunktion. Ein einfaches KNN ohne *Hidden*-Neuronen kann drei Werte des XOR-Problems richtig bestimmen. Ist dies der Fall, ergibt die Fitnessfunktion den Wert 9. Ab diesem Wert wird kein Fortschritt des Fitnesswertes erzielt bis das neue Neuron erfolgreich in die Struktur integriert ist. Der minimale Fitnesswert ist im gesamten Verlauf sehr gering. Der Grund hierfür ist, dass durch unpassende Mutationen oder Rekombinationen Genome entstehen können, bei denen die negativen Eigenschaften überwiegen.

Abbildung 4.3 zeigt die Ausführungszeit des Verfahrens in Sekunden für jede

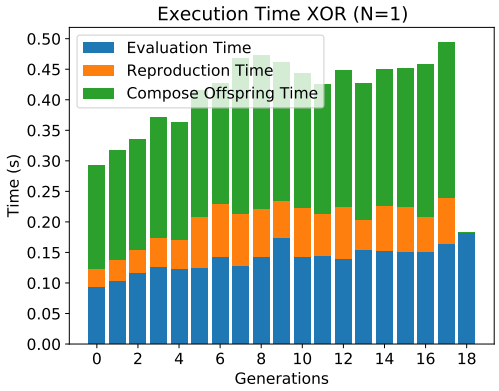

Abbildung 4.3: Ausführungszeiten des XOR-Problems auf einem Raspberry Pi 4 mit einem Prozess

Generation. Auch dieser Graph wird wieder automatisch mit dem Paket Matplotlib generiert und zeigt drei verschiedene Phasen. Der blaue Bereich zeigt die *Evaluation Time*, welche die benötigte Evaluationszeit für alle Agenten ist. Der grüne Balken repräsentiert die *Compose Offspring Time*, in welcher die Zeit für die Rekombination und Mutation aller Agenten gemessen wird. Die verbleibenden Aktionen bezüglich NEAT, wie zum Beispiel das Sortieren der Agenten in die verschiedenen Spezies, wird im Rahmen der *Reproduction Time* erfasst, welche mit dem orangenen Balken repräsentiert wird. Die benötigte Zeit zum Speichern von Zwischenergebnissen und Erzeugen von Log-Ausgaben wird nicht erfasst, da diese das Ergebnis verfälschen könnten. Insgesamt ist bezüglich des XOR-Problems zu erkennen, dass die Ausführungszeit im Verlauf der Generationen tendenziell ansteigt. Dies kann durch den erhöhten Rechenaufwand erklärt werden, welcher durch größere KNN entsteht. Des Weiteren ist auffällig, dass für die letzte Generation keine *Reproduction Time* oder *Compose Offspring Time* erfasst wird. Grund hierfür ist, dass nach der evaluierten Generation die Abbruchbedingung überprüft wird, welche in diesem Fall die Ausführung beendet. Daher wird keine neue Generation erstellt. Zuletzt soll die allgemeine Ausführungszeit und das Verhältnis der verschiedenen Phasen betrachtet werden. Zu erkennen ist, dass die Ausführungszeit pro Generation weniger als eine halbe Sekunde beträgt. Dies ist vor allem für Testzwecke ein großer Vorteil, da der Effekt von Änderungen schnell zu beobachten ist. Zusätzlich ist zu erkennen, dass die meiste Zeit für das Erstellen und Mutieren von neuen Genomen und Agenten benötigt wird. Allerdings ist diese Verteilung unter Umständen nicht repräsentativ, da die XOR-Funktion ein sehr einfaches Problem ist, welches nur wenige Instruktionen umfasst. Daher werden im nächsten Kapitel aufwendigere Optimierungsprobleme betrachtet.

4.3 Optimierungsprobleme

Nach erfolgreicher Verifizierung der Funktionalität wird in diesem Kapitel auf verschiedene andere Optimierungsprobleme eingegangen, anhand derer die Analyse durchgeführt werden soll. Grundsätzlich ist bei der Implementierung zu beachten, dass keine zu aufwendigen Optimierungsprobleme verwendet werden, da der Raspberry Pi 4 nicht so leistungsfähig ist und die benötigte Optimierungszeit sehr hoch sein kann. Daher werden im Folgenden hauptsächlich klassische Probleme des bestärkenden Lernens aus dem OpenAI Gym verwendet. Die ausgewählten Umgebungen sind das *Cartpole*, *MountainCar* und *Pendulum* Problem. Im Folgenden wird auf die entsprechenden Optimierungsprobleme genauer eingegangen und die Implementierung der Fitnessfunktion und Abbruchbedingung vorgestellt.

4.3.1 Cartpole

Die *Cartpole* Umgebung, auch als *Pole Balancing* bezeichnet, wurde bereits 1983 in Quelle [53] vorgestellt und ist auch heute noch ein bekanntes Optimierungsproblem, welches in vielen Publikationen verwendet wird. Auch im OpenAI Gym ist dieses Problem entsprechend der Beschreibung aus Quelle [53] enthalten und in Abbildung 4.4 dargestellt. In der Umgebung befinden sich zwei Gegenstände, ein Wagen und ein Balken. Der Wagen kann vom Agenten nach links und rechts bewegt werden. Hierauf befindet sich der Balken, der am unteren Ende mit dem Wagen verbunden ist und entsprechend seiner Position nach links oder rechts kippen kann. Ziel des Agenten ist, durch Steuerung des Wagens den Balken so lange wie möglich senkrecht zu balancieren. Bezüglich der Abbruchbedingung gilt, dass der Agent scheitert, wenn sich der Balken um mehr als 15 Grad zur Seite neigt oder der Wagen zu weit vom Zentrum entfernt ist. Als Eingabewerte für das KNN werden von der Umgebung vier Werte zur Verfügung gestellt, für welche je ein Eingabeneuron erstellt wird. Dies ist unter anderem die Position und Geschwindigkeit des Wagens, der aktuelle Winkel des Balkens und dessen Änderungsrate. Zusätzlich besitzt das erstellte KNN zwei Ausgabeneuronen, welche die jeweilige Richtung repräsentieren. Ist der Aktivierungsgrad des erstens *Output*-Neurons höher als der des zweiten, wird der Wagen nach links bewegt und andernfalls nach rechts. Bevor die Evaluation mit dieser Umgebung durchgeführt wird, müssen die Fitnessfunktion, Lösungsbedingung und Konfiguration festgelegt werden. Beim klassischen bestärkenden Lernen, wie in Kapitel 2.1.6 beschrieben, erhält der Agent nach jedem Zeitschritt einen *reward*. Da das OpenAI Gym primär für diese Art des Lernens konzipiert ist, wird auch hier nach jeder Aktion in der

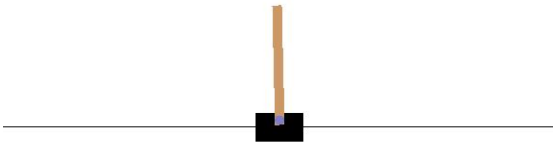

Abbildung 4.4: Darstellung der *Cartpole* Umgebung aus dem OpenAI Gym

Umgebung ein *reward* zurückgegeben. In diesem Fall erhält ein Agent bis zum Scheitern für jeden Zeitschritt einen *reward* mit der Wertigkeit 1. Mit diesen muss für neuroevolutionäre Algorithmen eine Fitnessfunktion definiert werden. Bei diesem Beispiel ist das Vorgehen einfach. Der Fitnesswert wird berechnet, indem die erhaltenen *rewards* aggregiert und der resultierende Wert am Ende der Evaluation quadriert wird. Somit soll wie zuvor beim XOR-Problem besseren Agenten ein proportional höherer Fitnesswert zugewiesen werden. Das Optimierungsverfahren wird beendet, wenn ein Agent den Balken 500 Zeitschritte balancieren kann. Die restlichen Parameter wurden aus dem vorherigen Beispiel übernommen und nicht geändert.

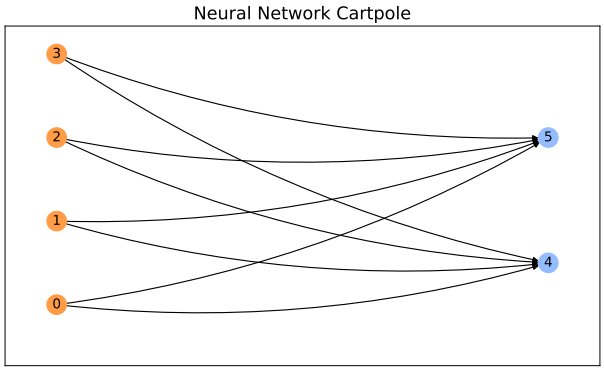

Abbildung 4.5: Struktur des finalen KNN im *Cartpole* Optimierungsproblem

Allerdings stellt sich beim Ausführen dieses Optimierungsproblems heraus, dass es für die Analyse ungeeignet ist. Bereits bei den zufällig erstellten Agenten der initialen Population sind KNN vorhanden, welche die Abbruchbedingung erfüllen und das Optimierungsproblem lösen. Ein solches ist in Abbildung 4.5 dargestellt. Wie die anderen KNN in der initialen Population besitzt auch dieses keine *Hidden*-Neuronen und zeigt, dass für das Lösen dieses Optimierungsproblems keine komplexen Entscheidungen notwendig sind. Es ist beispielsweise möglich, mit dem Winkel des Balkens auf die auszuführende Aktion zu schließen. Neigt sich der

Balken nach rechts, bewegt sich der Wagen in diese Richtung und umgekehrt. Dies ist einer der Gründe, warum dieses Optimierungsproblem im weiteren Verlauf der Arbeit nicht weiter verwendet wird. Ein weiterer Grund ist, dass aufgrund der fehlenden Optimierung keine Ausführungszeiten über mehrere Generationen gemessen werden können, welche eine notwendige Grundlage für den späteren Vergleich sind.

4.3.2 Mountain Car

Die Umgebung *Mountain Car* ist ein weiteres Optimierungsproblem, welches aus dem OpenAI Gym stammt und in Abbildung 4.6 dargestellt ist. Das Ziel für den Agenten ist, den Wagen auf den rechten Berg zu fahren, wo sich eine Fahne befindet. Hierfür stehen ihm Steuerungsoptionen für den Antrieb des Wagens zur Verfügung. Bei jedem Zeitschritt kann der Agent entscheiden, ob er nach links, rechts oder gar nicht beschleunigen möchte. Die Schwierigkeit dabei ist, dass der Antrieb nicht stark genug ist, um den Wagen direkt zum Ziel auf der rechten Bergspitze zu fahren. Das Ziel kann nur erreicht werden, wenn der Wagen zunächst ein Stück den linken Berg hochfährt, in einer gewissen Höhe die Richtung wechselt und beschleunigt. Mit dem zusätzlichen Schwung kann der Wagen dann die rechte Bergspitze erreichen. Für dieses Optimierungsproblem gibt es zwei Eingabewerte. Der erste ist die aktuelle Position des Wagens, der zweite seine Geschwindigkeit. Auf Basis von diesen muss der Agent seine Aktion wählen. Auch für diese Umgebung muss eine Abbruchbedingung und Fitnessfunktion festgelegt werden. Die Ausführung der Umgebung wird beendet, wenn der Agent entweder das Ziel auf der rechten Seite erreicht oder 200 Zeitschritte vergangen sind. Allerdings ist das Erreichen der Fahne allein nicht ausreichend, um das *Mountain Car* Problem erfolgreich zu lösen. Laut der Dokumentation des OpenAI Gyms muss der Agent dies in 100 aufeinander folgenden Evaluationen in durchschnittlich 110 Zeitschritten oder weniger schaffen.

Grundsätzlich kann dies implementiert werden, indem beispielsweise nach jeder Generation der beste Agent 100 mal in verschiedenen Umgebungen getestet wird. Aber da dies die Trainingszeit stark erhöhen würde und die Laufzeit auf dem Raspberry Pi 4 vergleichsweise hoch ist, wird eine einfachere Bedingung gewählt. Das Trainingsverfahren wird beendet, wenn es einem Agenten erstmalig gelingt, das *Mountain Car* Problem in weniger als 110 Zeitschritten erfolgreich zu beenden. Zuletzt muss die Fitnessfunktion implementiert werden. Die *Mountain Car* Umgebung gibt für jeden Zeitschritt einen *reward* von -1. Für eine bessere

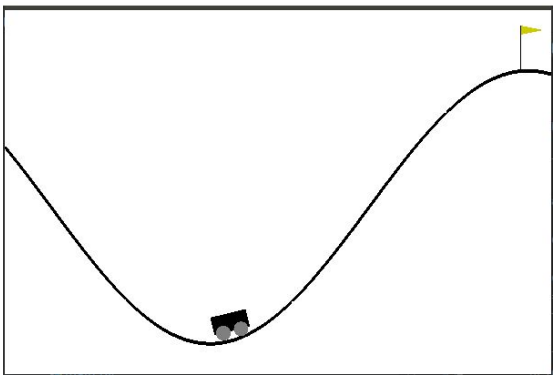

Abbildung 4.6: Darstellung der *Mountain Car* Umgebung aus dem OpenAI Gym

Übersichtlichkeit und zur Vermeidung negativer Fitnesswerte wird dieser auf den Wert 200 initialisiert. Für jeden benötigten Zeitschritt wird ein Punkt abgezogen. Bessere Agenten, die weniger Zeitschritte benötigten, erhalten so einen höheren Fitnesswert. Allerdings kann es vorkommen, dass kein Agent der initialen Population das Ziel auf der rechten Seite erreicht. In diesem Fall würde jeder Agent den Fitnesswert 0 haben, wodurch keine Selektion möglich ist. Aus diesem Grund wird die maximal erreichte X-Koordinate ebenfalls in den Fitnesswert miteinbezogen. Sollte kein Agent das Ziel erreichen, haben die Agenten, welche diesem am nächsten waren, einen Selektionsvorteil. Zuletzt wird der Fitnesswert wie auch zuvor quadriert, um besseren Agenten einen proportional höheren Fitnesswert zuzuweisen. Die restliche Konfiguration für diese Umgebung ist ähnlich zu den vorherigen Beispielen. Einzig die Populationsgröße wird auf 300 Agenten erhöht, da dieses Problem aufwendiger zu lösen ist als die vorherigen.

Prinzipiell kann das Verfahren ab diesem Punkt evaluiert werden, allerdings varieren die maximalen Fitnesswerte in vielen Fällen stark. Der Grund hierfür ist eine verrauschte Fitnessfunktion, wie es in Kapitel 2.2.4 erläutert ist. Die Startposition des Wagens ist zufällig durch die Umgebung gesetzt und kann einen großen Einfluss auf den Fitnesswert haben. So kann es vorkommen, dass ein Agent einen hohen Fitnesswert aufgrund einer günstigen Startposition erhält, ansonsten aber schlechte Ergebnisse erzielen würde. Ein Ansatz zur Minimierung dieses Effektes besteht darin, dass jeder Agent mehrfach die Evaluation durchführen muss und der mittlere Fitnesswert verwendet wird. Ein Vorteil dabei ist, dass der Agent mit verschiedenen Situationen konfrontiert wird und somit die Eigenschaft der Generalisierung erfüllt. Ein Nachteil ist, dass dadurch die Trainingszeit stark erhöht wird. Der Fokus in dieser Arbeit liegt auf dem Vergleich zwischen der

sequenziellen und parallelisierten Implementierung. Da dafür die Eigenschaft der Generalisierung nicht notwendig ist, wird ein anderer Ansatz gewählt. Die Umgebungen werden so konfiguriert, dass jeder Agent mit derselben Ausgangssituation beginnt, wodurch sich die Trainingszeiten verringern und ein Vergleich einfacher umzusetzen ist.

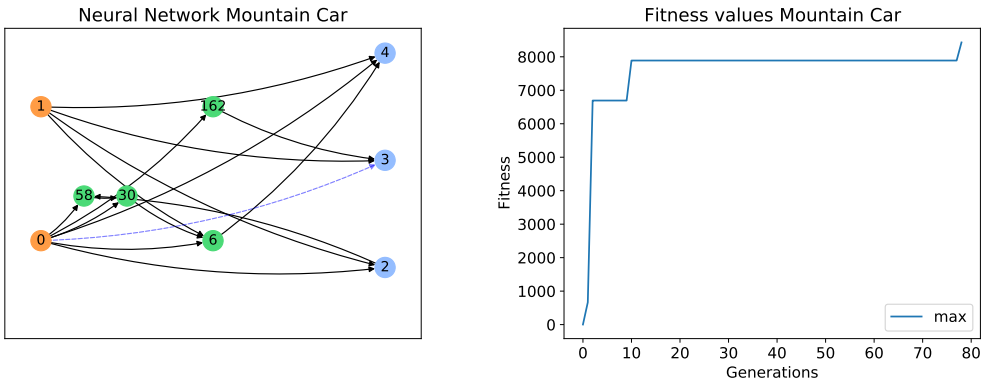

Abbildung 4.7: Links die Lösung für das Mountain Car Problem, rechts die dazugehörigen Fitnesswerte pro Generation

Mit der vorgestellten Konfiguration wird das Optimierungsproblem durchgeführt. Die Ergebnisse sind in Abbildung 4.7 und 4.8 dargestellt. Das KNN links in Abbildung 4.7 enthält vier *Hidden*-Neuronen. Zusätzlich sind sowohl zyklische als auch deaktivierte Verbindungen enthalten. In der Abbildung rechts ist der maximale Fitnesswert dargestellt. Innerhalb der ersten 10 Generationen steigt dieser schnell auf 7888 an, da der Agent 113 Zeitschritte zum Lösen gebraucht hat. Da für das erfolgreiche Lösen nicht mehr als 110 Zeitschritte genutzt werden dürfen, versucht der Algorithmus eine bessere Lösung zu finden. Allerdings stagniert der Fortschritt bis Generation 78, in der eine bessere Lösung gefunden wird, welche die Abbruchbedingung erfüllt. Abbildung 4.8 zeigt die benötigte Ausführungszeit für diesen Versuch. Insgesamt wurden ungefähr 105 Minuten für das Trainingsverfahren benötigt. Aus der Abbildung ist erkennbar, dass diese Zeit maßgeblich durch die *Evaluation Time* bestimmt ist. Insgesamt wurden 98% der Ausführungszeit für die Evaluation der Agenten verwendet. Die restlichen zwei Prozent teilen sich die *Compose Offspring Time* und *Reproduction Time*. Letztere hat mit insgesamt ca. 9 Sekunden die geringste Ausführungszeit benötigt. Für die Rekombination und Mutation von Agenten wurden insgesamt 113 Sekunden verwendet. Bezüglich der gesamten Ausführungszeit ist festzustellen, dass diese für jede Generation tendenziell ansteigt, auch wenn es einzelne Spitzen oder Einbrüche gibt. Hierfür kann es verschiedene Gründe geben. Die Ausführungszeit wird einer-

seits von der Größe der KNN beeinflusst, welche im Laufe des Verfahrens größer werden. Die Einbrüche in der Ausführungszeit können beispielsweise entstehen, wenn die Agenten großteils das Ziel früh erreichen und somit weniger Zeitschritte simuliert werden müssen. Alternativ kann die Ausführungszeit sinken, wenn große KNN schlechte Fitnesswerte erzielt haben und daher nicht für die Reproduktion ausgewählt werden.

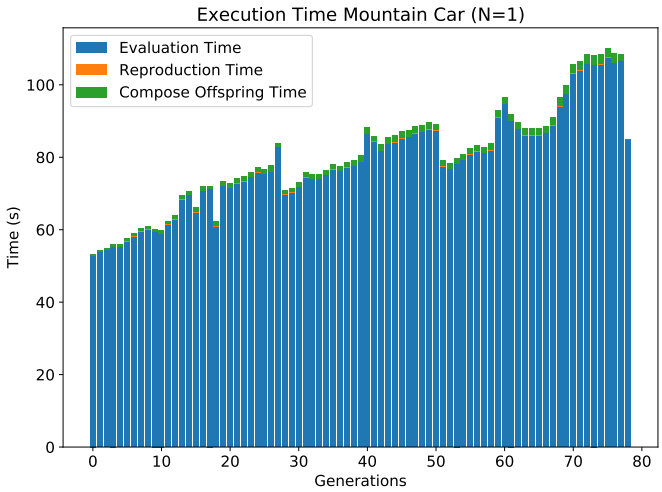

Abbildung 4.8: Ausführungszeiten des Mountain Car Problems auf einem Raspberry Pi 4 mit einem Prozess

4.3.3 Pendulum

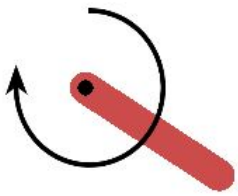

Abbildung 4.9: Darstellung der *Pendulum* Umgebung aus dem OpenAI Gym

Die letzte Umgebung, welche im Rahmen der Analyse betrachtet wird, ist das *Pendulum* aus dem OpenAI Gym, welches in Abbildung 4.9 dargestellt ist. In der Umgebung befindet sich ein Pendel mit einer initial zufälligen Position und Geschwindigkeit. Die Aufgabe des Agenten ist, dieses in eine senkrechte Position zu bringen und dort so lang wie möglich zu halten. Hierfür stehen ihm drei Eingabewerte zur Verfügung, von denen zwei die aktuellen Winkel des Pendels und

der dritte die Bewegungsgeschwindigkeit repräsentieren. Entsprechend besitzt das KNN drei Eingabeneuronen, auf Basis derer eine Aktion ausgewählt wird. Der Agent kann prinzipiell Kraft nach links oder rechts ausüben, um das Pendel zu bewegen. Allerdings gibt es einen Unterschied zu den vorherigen Testumgebungen. Bei diesen kann der Agent nur die Richtung entscheiden, in welche eine konstante Kraft wirken soll. Dies sind diskrete Entscheidungsprobleme. In dieser Umgebung muss der Agent nicht nur die Richtung, sondern auch die auszuübende Kraft bestimmen. Das bedeutet konkret für diese Umgebung, dass ein Wert zwischen -2 und 2 gewählt werden muss. Ist der Wert negativ, wird die Kraft nach links ausgeübt, andernfalls nach rechts. Die auszuübende Kraft wird durch die absolute Größe des Wertes bestimmt. Das KNN erhält einen Ausgabewert und verwendet die tanh Funktion für die Aktivierung, welche Werte zwischen -1 und 1 zurückgibt. Um die auszuführende Kraft zu berechnen, wird der Ausgabewert vor dem Ausführen der Aktion mit 2 multipliziert. Die Fitnessfunktion für diese Umgebung besteht wieder aus der Aggregation der einzelnen *rewards*, welche durch das OpenAI Gym für jeden Zeitschritt gegeben werden. Ein einzelner *reward* kann zwischen ≈ -16.2 und 0 liegen. Dies ist abhängig von der aktuellen Position und Geschwindigkeit des Pendels sowie der ausgeübten Kraft. Das Ziel ist, mit möglichst wenig Kraftaufwand das Pendel aufrecht zu halten. Das Trainingsverfahren wird beendet, wenn ein Agent einen Fitnesswert von -200 erreicht. Bezüglich der sonstigen Konfiguration ist anzumerken, dass bei diesem Optimierungsproblem keine zyklischen Verbindungen zugelassen werden und die Populationsgröße auf 1000 Agenten erhöht ist. Im Zuge dessen wird auch die Kompatibilitätsfunktion mit den Werten $c_3 = 4.0$ und $\delta_t = 3.5$ angepasst. Das Ziel dieser Änderung ist, dass den Gewichtsunterschieden eine höhere Priorität zugewiesen wird und unterschiedliche Strategien entwickelt werden können. Zuletzt ist bezüglich der Umgebung anzumerken, dass der Startzustand bei diesem Optimierungsproblem aus denselben Gründen wie in Kapitel 4.3.2 beschrieben für alle Agenten gleich ist.

Das Verfahren wird mit der spezifizierten Konfiguration durchgeführt, die erhaltenen Ergebnisse sind in Abbildung 4.10 und 4.11 dargestellt. Insgesamt hat der Algorithmus 55 Generationen evaluiert und einen finalen Fitnesswert von ca. -120 erreicht. Mit dem dargestellten KNN hat der Agent in der letzten Evaluation das Pendel beim ersten Versuch aufgerichtet und während der restlichen Simulation ausbalanciert. Bezüglich der Fitnesswerte zu erkennen, dass der Agent zwar immer wieder Fortschritte erzielt, aber auch einige Generationen lang stagniert. Dies ist ähnlich zu den vorherigen Beispielen. Die Ausführungszeiten hingegen

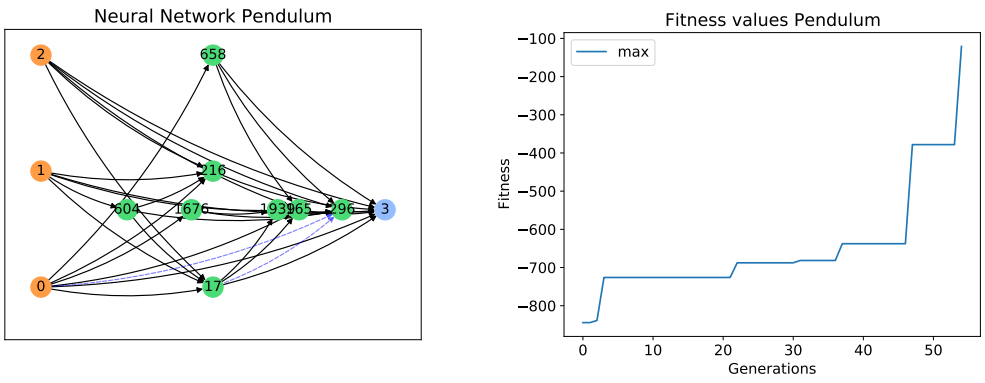

Abbildung 4.10: Links die Lösung für das Pendulum Problem, rechts die dazugehörigen Fitnesswerte pro Generation

haben sich etwas geändert. Insgesamt hat die Evaluation 125.8 Minuten gedauert. Wie zuvor beim *Mountain Car* Problem ist die Ausführungszeit während des Verfahrens insgesamt gestiegen. Im Gegensatz zum vorherigen Beispiel erfolgt der Zuwachs stetig und es entstehen keine großen Schwankungen der Laufzeit. Ein Grund hierfür ist, dass unabhängig der Leistung eines Agenten in der Evaluation immer 200 Zeitschritte ausgeführt werden. Die Evaluation wird nicht durch eine Fehlentscheidung oder dem erreichen eines Ziels vorzeitig beendet. Mit insgesamt 91% Anteil hat die *Evaluation Time* den größten Einfluss auf die Ausführungszeit. Die *Compose Offspring Time* hat bei diesem Durchlauf insgesamt 180 Sekunden und damit nur etwa 2% der Ausführungszeit benötigt. Die restlichen 7% werden für die *Reproduction Time* genutzt, welche insgesamt ungefähr 574 Sekunden angedauert hat. Der Grund hierfür ist, dass durch die veränderte Kompatibilitätsfunktion bedeutend mehr Spezies entstanden sind. Die Folge ist, dass für das Zuweisen von Nachkommen für jede Spezies und vor allem für das Sortieren der Agenten in diese bedeutend mehr Rechenaufwand entsteht.

4.4 Erkenntnisse

Nach Darstellung der Optimierungsprobleme erfolgt eine Zusammenfassung der Ergebnisse in diesem Kapitel. Alle vorgestellten Optimierungsverfahren konnten durch die Schnittstelle der Bibliothek schnell und einfach implementiert werden. Dies trifft sowohl auf individuell erstellte Optimierungsprobleme als auch auf Standardprobleme aus dem OpenAI Gym zu. Auch die restlichen Anforderungen aus Kapitel 3.1 sind vollständig implementiert. Der gesamte Ablauf des Algorithmus kann mithilfe eines spezifizierten *Seeds* beeinflusst und wiederholt werden. Die

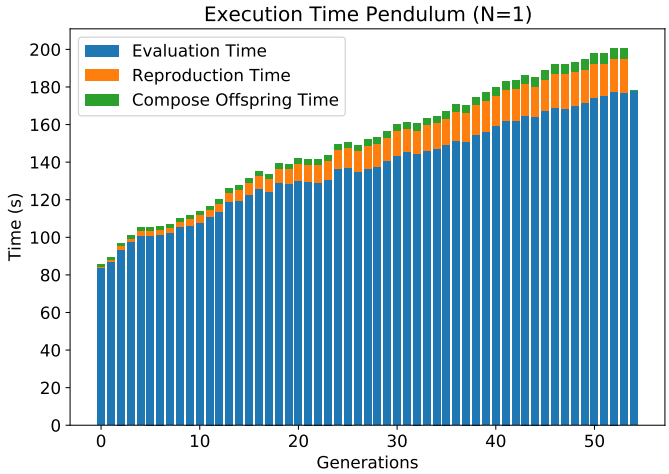

Abbildung 4.11: Ausführungszeiten des Pendulum Problems auf einem Raspberry Pi 4 mit einem Prozess

verschiedenen Abbildungen im vorherigen Kapitel zeigen, dass die Testergebnisse der Trainingsverfahren gemessen, visualisiert und gespeichert werden können, sodass ein späterer Vergleich mit der parallelisierten Implementierung möglich ist.

Die Funktionalität des Algorithmus, neue Strukturen zu entwickeln und zu optimieren, ist mit dem XOR-Problem bewiesen. Mit derselben Konfiguration werden im Vergleich zur originalen Implementierung ähnliche Ergebnisse gemessen. Durch verschiedene Anpassungen konnten bei nachfolgenden Versuchen sogar bedeutend bessere Ergebnisse erzielt werden. Bei Betrachtung der Ausführungszeit ist in diesem Beispiel die Zeit zum Erstellen und Mutieren von Nachkommen der größte Faktor. Aber da das XOR-Problem insgesamt sehr einfach ist und somit aufwendigere Probleme nicht richtig repräsentiert, finden die Ergebnisse im weiteren Verlauf keine Beachtung.

Im nächsten Schritt wurden daher die *Cartpole*, *Mountain Car* und *Pendulum* Umgebung des OpenAI Gyms implementiert, welche aufwendiger zu lösen sein sollten als das XOR-Problem. Die Tests haben ergeben, dass die erste Umgebung ungeeignet ist, da bereits die zufällig erstellten KNN der ersten Generation das Optimierungsproblem lösen können. Dies trifft nicht auf die zwei anderen Umgebungen zu. Im vorgestellten Szenario wurden für die erste Umgebung 105 Minuten, für die zweite Umgebung 125 Minuten zur erfolgreichen Optimierung benötigt. Zwar hat NEAT beide Probleme erfolgreich gelöst, dennoch werden die Grenzen des Algorithmus aufgezeigt. Vor allem auf dem Raspberry Pi sind die benötigten

Optimierungszeiten für verhältnismäßig kleine Probleme sehr hoch. Große Optimierungsprobleme, bei denen die Eingabevektoren aus mehreren tausend Werten bestehen, können aufgrund der begrenzten Rechenleistung nicht oder nur mit sehr hohem Zeitaufwand optimiert werden. Da der Fokus dieser Arbeit auf dem Vergleich zwischen einer sequenziellen und parallelisierten Implementierung liegt, ist dieser Umstand zu vernachlässigen.

Bei Betrachtung der Ausführungszeiten ist sowohl für das *Mountain Car* und *Pendulum* Problem ersichtlich, dass die Evaluationsphase der größte Faktor ist. In der ersten Umgebung werden 98%, in der zweiten 91% der Ausführungszeit für das Erstellen und Evaluieren der KNN verwendet. Da im Falle einer erfolgreichen Parallelisierung in dieser Phase die meiste Ausführungszeit eingespart werden kann, liegt der Fokus im folgenden Kapitel primär darauf. Für die eigentliche Parallelisierung ist wichtig, dass nicht nur die Berechnungen des KNN optimiert werden. Im Rahmen der *Evaluation Time* wird neben der benötigten Zeit zum Erstellen und Aktivieren eines KNN auch die benötigte Zeit zum Simulieren der Umgebung gemessen. Diese kann einen großen Einfluss auf das Gesamtergebnis haben. An zweiter und dritter Stelle der Priorisierung stehen die Parallelisierungen der Funktionen in den Phasen *Compose Offspring Time* und *Reproduction Time*. Beide haben einen bedeutend geringeren Anteil an der gesamten Ausführungszeit und dementsprechend weniger Zeit kann eingespart werden. Ob und wie diese implementiert werden können, wird in den Kapiteln (TODO KAPITEL!) erläutert.

Zuletzt soll betont werden, dass bei den vorgestellten Optimierungsproblemen keine generelle Lösungsstrategie entwickelt wird. Hierfür müssten verschiedene Startsituationen in der Evaluationsphase getestet werden, indem beispielsweise jeder Agent in x verschiedenen Umgebungen getestet und der mittlere Fitnesswert verwendet wird. Dies würde die Evaluationszeit um den Faktor x erhöhen, was im gegebenen Anwendungsszenario eine Analyse bedeutend aufwendiger gestalten würde. Mit der parallelisierten Version in Kapitel (TODO KAPITEL) kann ein solches Verfahren umgesetzt und ein entsprechendes KNN schneller entwickelt werden.

5 Optimierung

Grundsätzlich gibt es verschiedene Arten der Optimierung. In dieser Arbeit wird das sequenzielle Verfahren mit MPI parallelisiert. Hierzu wird in Kapitel 5.1 die ausgewählte Parallelisierungsstrategie vorgestellt. Im Anschluss dazu sind die konkreten Implementierungsdetails in Kapitel 5.2 erläutert. Als Testumgebung für das parallelisierte Verfahren wird ein Beowulf Cluster erstellt, dessen Aufbau in 5.3 beschrieben ist. Danach wird die eigentliche Evaluation mit der *Mountain Car* und *Pendulum* Umgebung in Kapitel 5.3 vorgestellt. Zuletzt wird noch ein KNN für die *Lunar Lander* Umgebung trainiert und die Ergebnisse der Parallelisierung zusammengefasst.

5.1 Strategien zur Parallelisierung

Es gibt mehrere Möglichkeiten zur Umsetzung einer Parallelisierung sowie verschiedene Schwerpunkte, welche diese haben kann. Die Analyse der sequenziellen Implementierung hat gezeigt, dass die *Evaluation Time* den größten Einfluss auf die Ausführungszeit hat. Dementsprechend kann bei einer erfolgreichen Parallelisierung dieser Phase die größte Reduktion der Ausführungszeit erzielt werden. Aus diesem Grund liegt der Fokus im Weiteren hierauf. Mit *Amdahl's Law*, welches in Kapitel 2.4.3 vorgestellt ist, kann der theoretisch erreichbare *SpeedUp* in Abhängigkeit von P Prozessoren berechnet werden. Dies ist in Abbildung 5.1 dargestellt, wobei entsprechend der Analyseergebnisse angenommen wird, dass die *Mountain Car* Umgebung zu 98% und die *Pendulum* Umgebung zu 91% parallelisiert werden können. Durch die Abbildung ist ersichtlich, dass beide Probleme anfänglich mit steigender Anzahl an Prozessen sehr gute *SpeedUp* Werte erzielen. Allerdings ist das Hinzufügen von weiterer Rechenleistung ab einem gewissen Punkt nicht mehr effektiv, da der Anstieg des *SpeedUps* zunehmend geringer wird und schlussendlich gegen einen gewissen Wert konvergiert. Für $P \to \infty$ liegt der maximale *SpeedUp* für die *Mountain Car* Umgebung bei Faktor 50 und für die *Pendulum* Umgebung bei Faktor $11.\overline{1}$. Auch wenn diese Ergebnisse eine ungefähre Richtlinie bieten, ist das Erreichen dieser Vorgaben in einer praktischen Umsetzung unwahrscheinlich.

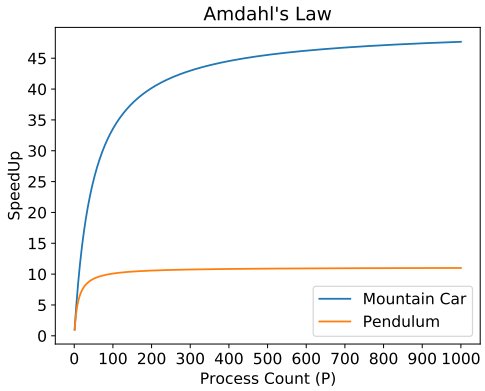

Abbildung 5.1: Durch *Amdahl's Law* berechnete theoretische *SpeedUp* für das *Mountain Car* und *Pendulum* Problem in Abhängigkeit der Anzahl an Prozessen

Durch eine hohe Anzahl an beteiligten Prozessen entsteht in der Regel ein hoher Kommunikationsaufwand, der sich negativ auf den *SpeedUp* Wert auswirkt. Im Folgenden werden verschiedene Strategien zur Umsetzung einer möglichst effizienten Parallelisierung aufgezeigt und die verschiedenen Vor- und Nachteile gegeneinander abgewogen.

Vor dem Auswählen einer geeigneten Strategie sollen die zu parallelisierenden Funktionen der *Evaluation Time* analysiert werden. Im sequenziellen Verfahren wird in dieser Phase über eine Liste mit allen Agenten iteriert. Für jeden von diesen wird die Umgebung einmal zurückgesetzt und ein KNN aus dem Genom des Agenten gebildet. Danach wird die Umgebung simuliert. Bei diesem Vorgang können beliebig viele Simulationsschritte und Aktivierungen des KNN ausgeführt werden.

Ein möglicher Ansatz besteht darin, die Berechnungen in einem KNN selbst zu parallelisieren. Dies ist eine valide Strategie, die unter anderem von Bibliotheken wie Tensorflow und PyTorch genutzt wird. Die einzelnen Neuronen einer Schicht können beispielsweise unabhängig voneinander aktiviert und somit auch parallelisiert werden. Zusätzlich kann eine solche Parallelisierung nicht nur mithilfe von CPUs, sondern auch auf GPUs umgesetzt werden. Diese haben in der Regel mehr unabhängige Prozessoren als normale CPUs, wodurch in vielen Fällen eine bessere Parallelisierung ermöglicht wird. Allerdings sprechen zwei Gründe gegen den Einsatz einer solchen Parallelisierungsstrategie in dieser Arbeit. Die vorgestellten Bibliotheken implementieren diese Funktionalitäten bereits, sind weit verbreitet und für verschiedene Plattformen optimiert. Zusätzlich ist es im

Rahmen einer zukünftigen Erweiterung möglich, diese einfach in dieses Projekt zu integrieren. Daher ist es nicht sinnvoll, Ressourcen für die Implementierung derselben Funktionalität zu verwenden. Der zweite Grund, der gegen diese Art der Parallelisierung spricht, besteht darin, dass zwar die Aktivierungszeit des KNN, jedoch nicht die Simulationszeit der Umgebung verkürzt wird. Diese kann je nach Komplexität des Problems sehr viel Rechenzeit in Anspruch nehmen.

Hier können neuroevolutionäre Algorithmen einen Vorteil bieten, da sowohl das KNN als auch die Umgebung parallelisierbar sind. Die Idee ist, dass jeder beteiligte Prozess eine Kopie des Optimierungsproblems initialisiert. Während des Verfahrens wird jeder Agent einem Prozess zugewiesen. Dieser kann unabhängig von anderen Prozessen ein KNN aus dem Genom erzeugen und mit diesem die gesamte Evaluation in der lokalen Umgebung durchführen. Am Ende wird nur der Fitnesswert als Ergebnis übertragen. Dieses Vorgehen bietet mehrere Vorteile. Mit dieser Strategie werden alle Funktionen der *Evaluation Time* parallelisiert, inklusive des Optimierungsproblems und der Berechnungen im KNN. In einer späteren Erweiterung kann zusätzlich eine Bibliothek wie Tensorflow integriert werden, welche die benötigte Zeit zum Berechnen des KNN reduziert und somit einen noch größeren *SpeedUp* ermöglicht. Ein weiterer Vorteil ist die geringe Anzahl an benötigten Nachrichten. Mit n Agenten müssen theoretisch nur $2 \cdot n$ Nachrichten pro Generation für die Parallelisierung der *Evaluation Time* ausgetauscht werden. Hiervon wird eine Hälfte zum Verteilen der Agenten und die andere Hälfte zum Sammeln der berechneten Fitnesswerte am Ende der Evaluation benötigt. Durch diese effiziente Kommunikation ist der zusätzlich entstehende Rechenaufwand sehr gering und das Erreichen einer effizienten Parallelisierung möglich.

5.2 Implementierung

Dieses Kapitel befasst sich mit der Implementierung der bereits vorgestellten Parallelisierungsstrategie. Das Ziel ist, die Ausführungszeit des Verfahrens in einem verteilten System durch Hinzufügen von weiteren Prozessoren zu reduzieren. Bei der Umsetzung sind drei Anforderungen zu beachten. Erstens muss es mit geringem Aufwand möglich sein, die sequenzielle Implementierung durch die parallelisierte auszutauschen und umgekehrt. Zweitens muss für eine Bewertung der Effizienz der Parallelisierung ein direkter Vergleich zwischen beiden Implementierungen möglich sein. Drittens ist der Standard MPI für die Kommunikation zu verwenden

Für die parallelisierte Implementierung wird eine neue Klasse angelegt, die von dem in Kapitel 3.2 vorgestellten Interface *NeatOptimizer* erbt. Da die sequenzielle Implementierung dasselbe Interface verwendet, ist die erste Anforderung bereits erfüllt. Beide Klassen stellen dieselben Funktionen zur Verfügung und können somit jederzeit gegeneinander ausgetauscht werden. Wie in Kapitel 3.2 erläutert, ist die *evaluate()* Funktion in diesem Interface der Einstiegspunkt für die Bibliothek und das gesamte Trainingsverfahren ist in dieser implementiert.

Um die zweite Anforderung zu erfüllen, ist der grundsätzliche Ablauf der *evaluate()* Funktion identisch zum sequenziellen Verfahren, sodass sich nur der parallelisierte Teil des Programms unterscheidet. Zu Beginn des Trainingsverfahrens wird eine initiale Population generiert. Danach beginnt eine Schleife, in welcher die Funktionen *build_new_generation()* und *evaluate_generation()* abwechselnd aufgerufen werden bis die Abbruchbedingung erfüllt ist. Da primär die Evaluationsphase parallelisiert wird, kann die *build_new_generation()* Funktion vom sequenziellen Verfahren unverändert übernommen werden. In dieser sind die Phasen Selektion, Rekombination und Mutation implementiert. Die Funktion *evaluate_generation()* umfasst die Evaluation der Agenten. Da die Analyse zeigt, dass sowohl in der *Mountain Car* als auch in der *Pendulum* Umgebung diese Funktion den größten Einfluss auf die benötigte Rechenzeit hat, wird sie im Rahmen der Parallelisierung neu implementiert.

Im sequenziellen Verfahren wird bei der Evaluationsphase über die Liste mit allen Agenten iteriert und für jeden nacheinander die Evaluation durchgeführt. Wie bereits in Kapitel 3.3 beschrieben, wird für jeden neuen Agenten die Umgebung einmalig zurückgesetzt, das KNN gebildet und die eigentliche Evaluation mit dem Ziel durchgeführt, am Ende einen Fitnesswert zurückzugeben. Bei der parallelisierten Implementierung werden die Agenten an alle beteiligten Prozesse verteilt, welche dann die Evaluation unabhängig voneinander durchführen.

Dafür muss im nächsten Schritt die Art der Kommunikation und Verteilung der Agenten festgelegt werden. In Kapitel 2.4.2 sind die *Scatter* und *Gather* Funktionen vorgestellt, welche sich prinzipiell für dieses Szenario anbieten. Die *Scatter* Funktion verteilt die Agenten gleichmäßig auf die Prozesse und die *Gather* Funktion sammelt die Ergebnisse. Zwar ist die Implementierung effizient, es ergibt sich aber ein gewichtiger Nachteil. Die *Scatter* Funktion erlaubt keine dynamische Verteilung von Lasten, was in diesem Projekt jedoch aus zwei

Gründen sinnvoll ist. Der erste Grund betrifft die Evaluationszeiten, der zweite die Struktur des Clusters. Die Evaluationszeiten der Agenten unterscheiden sich in vielen Fällen. Gründe hierfür können zum einen unterschiedlich große KNN sein, deren Berechnungszeit bzw. Aktivierungszeit variiert, zum anderen unterschiedliche Fortschritte im Optimierungsproblem. In der vorgestellten *Mountain Car* Umgebung wird beispielsweise die Evaluation beendet, wenn der Agent das Ziel erreicht. In anderen Optimierungsproblemen kann die Evaluation frühzeitig abgebrochen werden, wenn ein Agent eine bestimmte Fehlentscheidung getroffen hat. Die Folge in beiden Szenarien ist, dass im Vergleich zu anderen Prozessen weniger Rechenaufwand für die Evaluation des Agenten benötigt wird. Hierdurch entsteht ein Ungleichgewicht in der Lastenverteilung. Die Folge ist, dass am Ende einer Evaluationsphase die Prozesse aufeinander warten. Somit sinkt die Effizienz und auch der erreichte *SpeedUp* Wert der Parallelisierung. Auch wenn alle Agenten dieselbe Rechenlast erzeugen und diese gleichmäßig auf alle Prozesse verteilt ist, kann es in einem Beowulf Cluster zu Wartezeiten kommen. Grund hierfür ist, dass sich die Konfiguration der Hardware bei den einzelnen Geräten des Clusters und somit auch deren Leistung unterscheiden kann. Für eine gute Performance des parallelisierten Verfahrens müssen die Prozessoren bzw. Geräte des Clusters, welche mehr Rechenleistung bieten, auch proportional mehr Rechenlast zugewiesen bekommen. Dies ist mit der *Scatter* Funktion nicht möglich.

Daher soll in dieser Arbeit die *Point-to-Point* Kommunikation verwendet werden, mit welcher ein dynamisches Zuteilen möglich ist. Um mögliche *Deadlocks* zu vermeiden, wird eine *Master-Slave* Architektur gewählt. Zu Beginn der Evaluationsphase erstellt der *Master* Prozess verschiedene Arbeitspakete, welche jeweils einen zu evaluierenden Agenten enthalten. Initial wird an jeden *Slave* Prozess asynchron ein Paket gesendet, welche hierauf warten. Ist dieses vollständig empfangen, wird der darin enthaltene Agent im lokalen Optimierungsproblem evaluiert und der berechnete Fitnesswert als Ergebnis an den *Master* Prozess zurückgesendet. Dieser ist für die Speicherung zuständig. Danach wird überprüft, ob noch weitere Arbeitspakete zur Abarbeitung ausstehen. Ist dies der Fall, wird ein neues Paket an den *Slave* übermittelt. Erst wenn alle Pakete abgearbeitet sind, endet die Evaluation der Generation. Danach führt der *Master* Prozess die erforderlichen Operationen zum Erzeugen der nächsten Generation aus. Dabei werden die gespeicherten Ergebnisse der *Slaves* verwendet. Mit der neuen Generation beginnt die Evaluation erneut. Ein großer Vorteil dieses Verfahrens ist, dass Prozesse automatisch mehr Arbeitspakete zugeteilt bekommen, wenn sie entweder

leistungsfähiger oder die Evaluationszeiten der Agenten kürzer sind. Zusätzlich ist die Kommunikationsstruktur strikt geregelt, sodass das Auftreten von *Deadlocks* unwahrscheinlich ist.

Für die praktische Umsetzung muss der Implementierung noch die Klasse *Slave* hinzugefügt werden, welche die notwendigen Operationen enthält. Dies umfasst die Funktion *setup()*, welche zum Erstellen des Optimierungsproblems bzw. der Umgebung benötigt wird und die Funktion *evaluate_agent()*. Als Parameter wird dieser Funktion ein Agent übergeben, der dann evaluiert wird. Im Rahmen dessen wird die Umgebung auf den Startzustand gesetzt, ein KNN mit dem Genom des Agenten erzeugt und die eigentliche Evaluation in der lokalen Umgebung durchgeführt. Die Funktion gibt am Ende den Fitnesswert sowie optionale Zusatzinformationen der Umgebung zurück. Insgesamt entspricht diese Funktionalität der des sequenziellen Verfahrens. Die *setup()* Funktion erstellt die Umgebung, benötigt aber einen anderen Implementierungsansatz als die *evaluate_agent()* Funktion. Der Grund hierfür ist der Parameter. Ein Agent enthält nur Daten, aber keine Funktionalität. Aus diesem Grund kann er einfach mit MPI serialisiert und versendet werden. Ein Optimierungsproblem hingegen kann eine komplexe Klasse mit unterschiedlichen Funktionen sein, die unter Umständen nicht serialisiert werden können. Zusätzlich sind in der implementierten Bibliothek keine Informationen über die Funktionalitäten, Architektur und den internen Zustand der Umgebung enthalten. Grund hierfür ist, dass neue Optimierungsprobleme von Nutzern jederzeit hinzugefügt werden können und es diesbezüglich keine Einschränkungen geben soll. Daher kann das Optimierungsproblem nicht mit MPI serialisiert werden. Stattdessen wird der Umstand genutzt, dass der Programmcode auf jedem beteiligten Gerät bzw. Prozessor vorliegen muss. Beim Starten einer MPI Anwendung wird auf jedem beteiligten Prozess eine Kopie des Programms ausgeführt. Hierbei kann jeder Prozess auf Basis des zugewiesenen Rangs entscheiden, ob er die Funktion eines *Masters* oder *Slaves* einnimmt. Der *Master* führt die notwendigen Operationen zum Erstellen der initialen Generation durch und startet den Optimierungsprozess. Die *Slaves* greifen auf das im Programmcode vorliegende Optimierungsproblem zu und initialisieren es. Bei einem solchen Vorgehen ist es nicht erforderlich, das Optimierungsproblem zu serialisieren und zu versenden.

Der letzte hier vorgestellte Implementierungsaspekt bezieht sich auf die Umsetzung der *Point-to-Point* Kommunikation. Hierfür wird das Paket *mpi4py* verwendet, welches eine Python Schnittstelle zum Nutzen von MPI Funktionen bietet. Das

Paket selbst ist *open source* verfügbar und im Vergleich zu anderen Implementierungen weit verbreitet. Zusätzlich bietet es eine gute Performance welche in Quelle [41] gemessen wurde und als nur geringfügig langsamer als eine Implementierung in der Programmiersprache C beschrieben ist. Ein letzter entscheidender Vorteil sind verschiedene in der Bibliothek implementierte *high-level* Funktionen. Diese ermöglichen unter anderem das einfache Umsetzen der oben beschriebenen *Master-Slave* Kommunikation. Diese Funktionen werden in dieser Arbeit für die Kommunikation verwendet.

5.3 Testumgebung

Das nächste Kapitel befasst sich mit der Evaluation des parallelisierten Verfahrens und führt einen Vergleich mit der sequenziellen Implementierung durch. Zuvor wird in diesem Kapitel auf die verwendete Testumgebung sowie die Installation der notwendigen Software eingegangen.

Der größte Unterschied dieser Testumgebung im Vergleich zu der des sequenziellen Verfahrens besteht darin, dass das parallelisierte Verfahren nicht auf einem Raspberry Pi 4, sondern auf einem Beowulf-Cluster ausgeführt wird, welcher schematisch in Abbildung (TODO ABBILDUNG) dargestellt ist. Insgesamt stehen zehn identisch konfigurierte Raspberry Pi 4 *Nodes* zur Verfügung. Die Hardware von diesen entspricht der Beschreibung in Kapitel 4.1. Somit besitzt der entstehende Cluster die zehnfache Rechenleistung im Vergleich zu der Testumgebung des sequenziellen Verfahrens.

Entsprechend der in Kapitel 2.4.1 definierten Anforderungen an einen Beowulf Cluster ist jeder Raspberry Pi durch eine Ethernet Verbindung mit einem Netzwerkswitch verbunden. Die maximale Bandbreite für jeden Raspberry Pi beträgt ein Gigabit. Hinsichtlich der Software wird das bisher verwendete Betriebssystem sowie alle Softwarepakete von der sequenziellen Testumgebung übernommen. Allerdings werden für die Kommunikation im Beowulf Cluster noch weitere Softwarekomponenten benötigt. Zuerst wird das MPICH installiert. Wie in Kapitel 2.4.2 beschrieben, ist dies eine weit verbreitete *open source* Implementierung des MPI Standards. Zusätzlich wird in der Python Umgebung das bereits vorgestellte Paket *mpi4py* installiert, welches den Zugriff auf MPI Funktionen aus Python ermöglicht. Ab diesem Zeitpunkt kann die Implementierung lokal getestet werden. Dennoch kann das Verfahren noch nicht auf dem gesamten Beowulf Cluster ausgeführt

werden, da die einzelnen *Nodes* noch nicht miteinander kommunizieren können. Hierfür wird zusätzlich das *Secure Shell* (SSH) Protokoll benötigt. Beim Beginn der Ausführung eines MPI Programms wird eine SSH Verbindung von jedem teilnehmenden Prozess zu allen anderen erstellt, über welche dann die spätere MPI Kommunikation erfolgt. Für die Authentifizierung im SSH Protokoll wird in diesem Fall ein asymmetrischer kryptographischer Schlüssel benötigt. Auf jedem Raspberry Pi wird hiervon einer erzeugt, welcher aus einem öffentlichen und privaten Teil besteht. Von jedem Raspberry Pi wird der öffentliche Teil des Schlüssels auf den anderen Geräten des Clusters hinterlegt, sodass eine Authentifizierung ohne Passworteingabe möglich ist.

Nach Konfiguration des SSH Protokolls kann das Optimierungsverfahren gestartet werden, sobald der Programmcode auf allen beteiligten Geräten vorliegt. Prinzipiell kann dieser manuell an jede einzelne *Node* verteilt werden. Dies ist zeitaufwendig und nicht für die aktive Entwicklung geeignet, daher wird in dieser Testumgebung eine automatisierte Lösung angestrebt. Wie in Kapitel 2.4.1 beschrieben, kann die *Master Node* eines Beowulf Clusters verschiedene organisatorische Aufgaben übernehmen. In diesem Projekt gehört hierzu unter anderem eine Schnittstelle zu den externen Umgebungen, über welche das Optimierungsverfahren gestartet werden kann. Zusätzlich wird für die automatisierte Verteilung des Programmcodes ein Ordner im Netzwerk des Beowulf Clusters freigegeben. Dieser kann von den *Slaves* lokal hinzufügt werden. Der Programmcode bzw. die benötigten Dateien müssen auf dem *Master* im freigegebenen Ordner vorhanden sein. Mit entsprechender Standardsoftware kann eine automatisierte Verteilung und Synchronisation der Dateien umgesetzt werden. Änderungen sind direkt auf allen *Slaves* verfügbar und fehlende oder inkonsistente Dateien werden vermieden.

5.4 Evaluation

Nachdem die Parallelisierung der Evaluationsphase durchgeführt und die Testumgebung spezifiziert ist, wird in diesem Kapitel die Performance des Verfahrens gemessen. Die hierbei erhaltenen Ergebnisse werden mit denen der sequenziellen Implementierung verglichen. Im letzten Schritt wird eine Bewertung bezüglich der Effizienz und des *SpeedUps* abgegeben.

5.4.1 Mountain Car

Das parallelisierte Verfahren wird zuerst in der *Mountain Car* Umgebung getestet, die aus Kapitel 4.3.2 bekannt ist. Mit dem Ziel, einen einfachen Vergleich zwischen der sequenziellen und parallelisierten Implementierung zu ermöglichen, wird die zuvor verwendete Konfiguration des Verfahrens vollständig übernommen. Selbiges gilt für den *Seed*. Bei korrekter Implementierung des parallelisierten Verfahrens werden dieselben Zwischenergebnisse nach jeder Generation generiert und das finale Ergebnis stimmt mit dem des sequenziellen Verfahrens überein. Somit kann keine Implementierung einen zeitlichen Vorteil erlagen, der durch bessere Agenten oder kürzere Evaluationszeiten entsteht. Beide Implementierungen werden dieselben Berechnungen und Evaluationen durchführen.

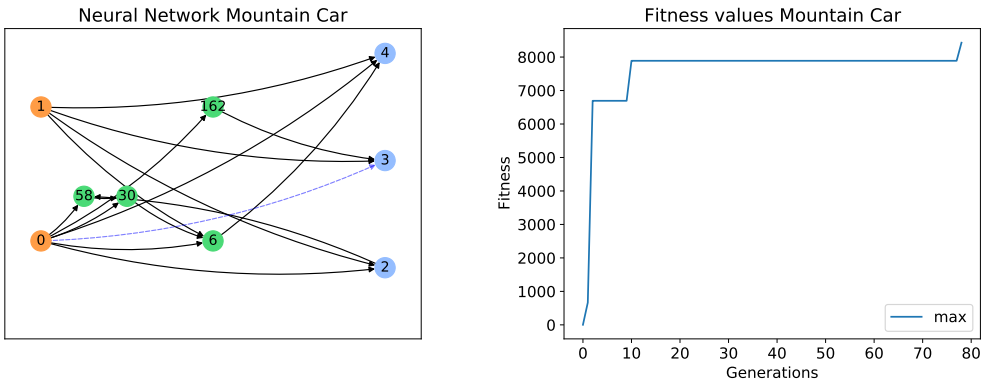

Abbildung 5.2: Links die Lösung für das *Mountain Car* Problem, rechts die dazugehörigen Fitnesswerte pro Generation mit 10 Prozessen

Im ersten Durchlauf wird der parallelisierte Algorithmus mit zehn Prozessen auf zehn Raspberry Pis ausgeführt. Das Verfahren ist so konfiguriert, dass auf jedem Raspberry Pi ein Prozess gestartet wird. Abbildung 5.2 zeigt das hierbei entstandene KNN und den maximal erreichten Fitnesswert. Beide Darstellungen sind identisch mit denen des sequenziellen Verfahrens. Da auch die restlichen Zwischenergebnisse übereinstimmen, wie beispielsweise die Anzahl an verschiedenen Spezies pro Generation, ist die Anforderung bezüglich des *Seeds* erfüllt. Sowohl die parallelisierte als auch die sequenzielle Implementierung führen dieselben Rechenschritte aus, wodurch ein direkter Vergleich der Laufzeiten möglich ist. Die Ausführungszeit für die vorgestellte Konfiguration ist in Abbildung 5.3 dargestellt. Wie beim sequenziellen Verfahren wird diese in die Phasen *Evaluation Time*, *Reproduction Time* und *Compose Offspring Time* unterteilt. Grundsätzlich ist festzustellen, dass die insgesamt benötigte Laufzeit stark verringert ist. Mit zehn

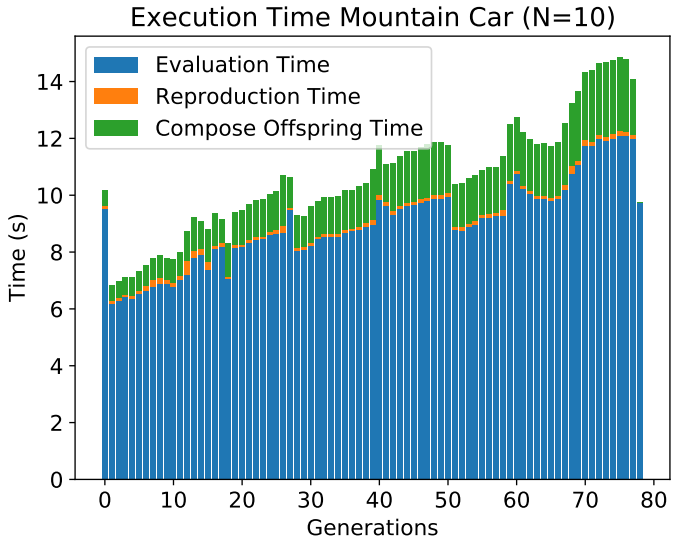

Abbildung 5.3: Ausführungszeit des *Mountain Car* Problems auf 10 *Raspberry Pis* mit 10 Prozessen

Prozessen in der vorgestellten Konfiguration sinkt die durchschnittliche Rechenzeit auf ungefähr 10.5 Sekunden pro Generation. Die sequenzielle Implementierung hat im Vergleich dazu etwa 80 Sekunden pro Generation benötigt. Für die gesamte Ausführungszeit des Verfahrens bedeutet dies, dass das parallelisierte Verfahren bereits nach 14 Minuten beendet ist. Im Vergleich zur Laufzeit der sequenziellen Implementierung mit 105 Minuten ist das parallelisierte Verfahren 91 Minuten schneller.

Eine Besonderheit des Graphen zeigt sich in der ersten Generation. In dieser ist die Ausführungszeit der *Evaluation Time* vergleichsweise hoch. Der Grund hierfür ist, dass das Starten und Initialisieren der beteiligten Prozesse Zeit benötigt, welche die Evaluation der Agenten verzögert. Da dies im Rahmen der *Evaluation Time* erfasst wird, erhöht sich die gemessene Ausführungszeit in der ersten Generation. Für die nachfolgenden Generationen trifft dies nicht mehr zu, da die Initialisierung nur einmalig zu Beginn erfolgt. Eine weitere Besonderheit betrifft die Form des Graphen. Wie bei der sequenziellen Implementierung gibt es auch in der parallelisierten Implementierung Generationen, in denen die Ausführungszeit stark sinkt oder ansteigt. Mögliche Gründe hierfür sind in Kapitel 4.3.2 erörtert. Bei einem direkten Vergleich der Ausführungszeiten ist festzustellen, dass diese Änderungen meistens in denselben Generationen auftreten. Diese Eigenschaft ist wichtig, da ein Vergleich der Laufzeiten nur möglich ist, wenn die Messergebnisse, von kleinen Abweichungen abgesehen, konsistent sind. Diese Eigenschaft kann

zusätzlich mit der *Reproduction Time* und *Compose Offspring Time* verifiziert werden. Diese Phasen sind nicht parallelisiert und daher sollte die Ausführungszeit identisch sein. Insgesamt sind die gemessenen Laufzeiten in diesen Phasen etwas höher als bei der sequenziellen Implementierung. Über alle 78 Generationen entsteht eine Differenz von ungefähr 5 Sekunden beziehungsweise 4%. Diese Unterschiede können beispielsweise durch Hintergrundprozesse im Betriebssystem entstehen, auf welche kein Einfluss genommen werden kann. Da die Abweichungen insgesamt gering sind, kann dennoch ein Vergleich der Laufzeiten vorgenommen werden. Die letzte zu nennende Eigenschaft bezüglich der Abbildung 5.3 ist der prozentuale Anteil der *Reproduction Time* und *Compose Offspring Time*. Bei der sequenziellen Implementierung haben diese beiden Phasen nur etwa 2% der Ausführungszeit in Anspruch genommen. Im aktuellen Testdurchlauf sind dies 15%, da durch die Parallelisierung der Anteil der *Evaluation Time* sinkt. Aufgrund *Amdahl's Law* ist anzunehmen, dass durch das Hinzufügen von weiteren Prozessen der prozentuale Anteil der *Reproduction Time* und *Compose Offspring Time* weiter ansteigt.

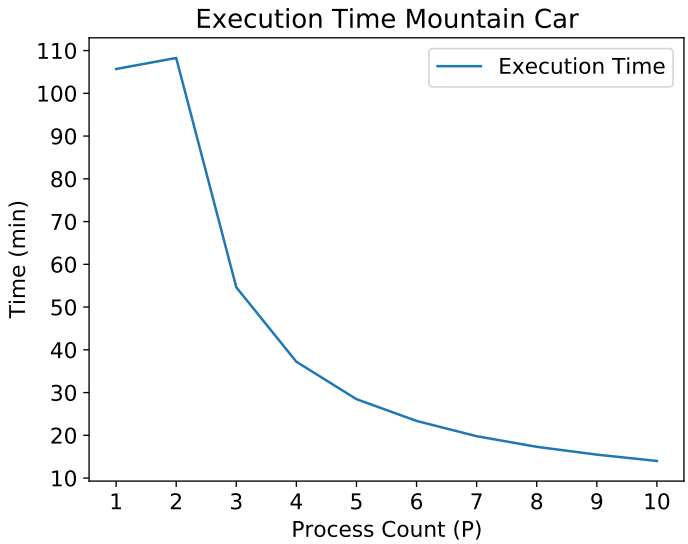

Abbildung 5.4: Ausführungszeit des parallelisierten Verfahrens in der *Mountain Car* Umgebung in Abhängigkeit zur Prozessanzahl

Für eine vollständige Bewertung des parallelisierten Verfahrens werden die in Kapitel 2.4.3 vorgestellten Metriken *SpeedUp* und Effizienz benötigt. Für eine bessere Einordnung der Ergebnisse werden diese nicht nur für die oben vorgestellte Konfiguration mit zehn Prozessen, sondern auch für Testdurchläufe mit zwei bis neun Prozessen berechnet. Diese werden im Folgenden mit derselben Konfiguration durchgeführt. Die hierbei entstehenden KNN und Fitnesswerte sind identisch mit

den vorherigen und sind daher nicht abgebildet. Die benötigte Ausführungszeit für das gesamte Verfahren in Abhängigkeit zur Anzahl an Prozessen ist in Abbildung 5.4 dargestellt. Eine Besonderheit hierbei ist, dass die benötigte Ausführungszeit mit zwei Prozessen länger ist als die mit einem Prozess. Der Grund hierfür liegt in der gewählten *Master-Slave* Architektur des parallelisierten Verfahrens. Mit einem Prozess wird das sequenzielle und andernfalls das parallelisierte Verfahren ausgeführt. Werden genau zwei Prozesse verwendet, gibt es einen *Master* und einen *Slave*. In diesem Fall ist die Ausführung langsamer als beim sequenziellen Verfahren, da der *Master* im Rahmen der Parallelisierung nur die Kommunikation koordiniert, selbst aber keine Aufgabenpakete abarbeitet. Der *Slave* führt alleine die Evaluation der Agenten durch. Dafür benötigt er dieselbe Zeit wie das sequenzielle Verfahren. Hinzu kommt der benötigte Kommunikationsaufwand für das Verteilen der Agenten und Sammeln von Fitnesswerten. Aufgrund der hierfür aufgewendeten Zeit ist das parallelisierte Verfahren mit zwei Prozessen langsamer als das sequenzielle Verfahren. Mit drei oder mehr beteiligten Prozessen sinkt die Ausführungszeit kontinuierlich.

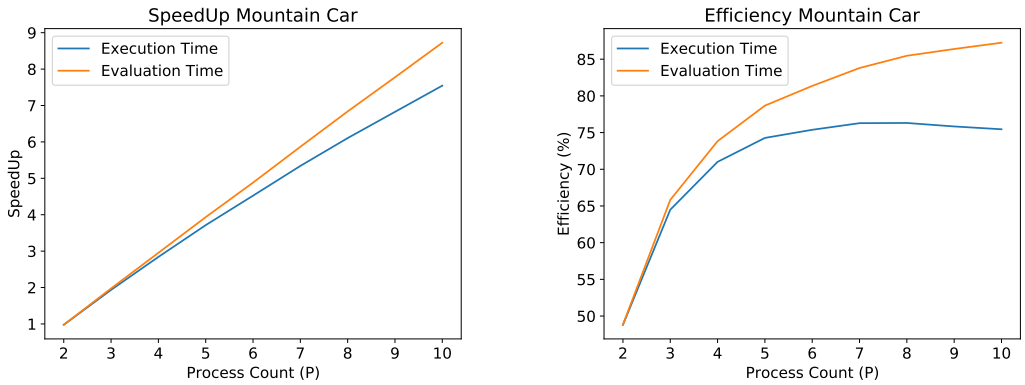

Abbildung 5.5: Links der *SpeedUp*, rechts die dazugehörigen Effizienzwerte für die *Mountain Car* Umgebung in Abhängigkeit zur Prozessanzahl

Anhand der Ausführungszeiten wird der *SpeedUp* und die Effizienz des parallelisierten Verfahrens gemessen, welche eine Bewertung der Implementierung ermöglichen. Abbildung 5.5 zeigt links die erreichten *SpeedUps* und rechts die dazugehörigen Effizienzwerte in Abhängigkeit zur Anzahl an verwendeten Prozessen. In beiden Graphen sind jeweils die Metriken für die Evaluationsphase und das gesamte Verfahren dargestellt. Beim Testdurchlauf mit zehn Prozessen ist das parallelisierte Verfahren ungefähr um den Faktor 7.6 schneller als das sequenzielle Verfahren. Dieses Ergebnis ist grundsätzlich höher als bei den vorherigen Testdurchläufen mit weniger Prozessen, entspricht aber nicht der Vorhersage von

Amdahl's Law. Laut diesem liegt der maximal erreichbare *SpeedUp* mit zehn Prozessen in einem zu 98% parallelisierbaren Programm bei ungefähr 8.5. Durch die gewählte *Master-Slave* Kommunikationsarchitektur ist zwischen dem erwarteten und tatsächlich erhaltenen Wert eine Differenz von 0.9. Wie bereits beschrieben, unterstützt der *Master* die *Slaves* nicht bei der Evaluation, wird aber dennoch bei den Berechnungen berücksichtigt. Wird nur die Anzahl der *Slave* Prozesse bei der Berechnung von *Amdahl's Law* verwendet, ergibt sich ein maximaler *SpeedUp* von 7.8. Die restliche Differenz von 0.2 entsteht durch den Kommunikationsaufwand.

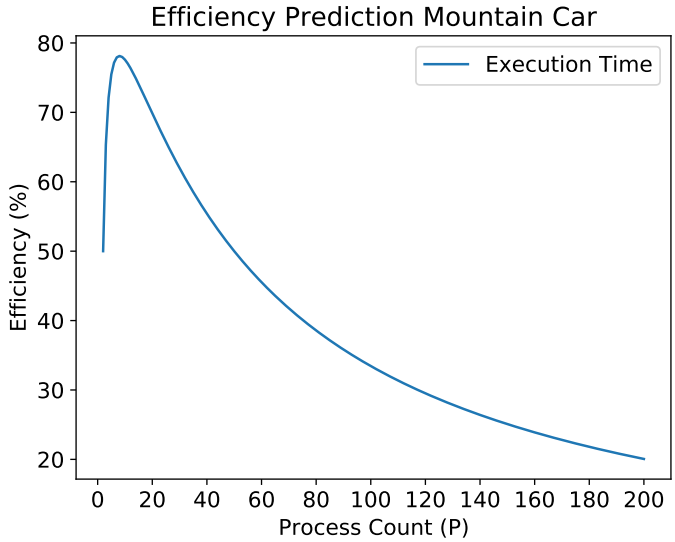

Abbildung 5.6: Erwartete Effizienz in der *Mountain Car* Umgebung in Abhängigkeit zur Anzahl an Prozessen

Die in Abbildung 5.5 rechts dargestellten Effizienzwerte beurteilen den *SpeedUp* anhand der Prozessanzahl. Initial erzielt das parallelisierte Verfahren mit zwei Prozessen aufgrund der *Master-Slave* Architektur eine geringe Effizienz. Mit je einem *Master* und *Slave* wird entsprechend der Abbildung 5.4 eine etwas höhere Ausführungszeit im Vergleich zur sequenziellen Implementierung benötigt. Daher liegt der *SpeedUp* bei ungefähr 0.98. Entsprechend der Formel aus Kapitel 2.4.3 ergibt die Berechnung der Effizienz einen Wert von 49%. Beim Hinzufügen weiterer Prozesse sinkt der Einfluss des *Masters* auf das Ergebnis und die Effizienz steigt. Mit sechs bis zehn Prozessen wird ein Wert von über 75% erreicht. Aufgrund des Graphen und *Amdahl's Law* ist davon auszugehen, dass der Wert nicht weiter steigen, sondern durch Hinzufügen weiterer Prozesse letztendlich sinken wird. Abbildung 5.6 zeigt die erwarteten Effizienzwerte für die *Mountain Car* Umgebung mit einer *Master-Slave* Architektur, wobei die Werte mit *Amdahl's Law*

berechnet worden sind. Auch in diesem Fall steigt die Effizienz anfangs stark an. Der höchste Wert von ungefähr 78% wird mit acht Prozessen erreicht. Dies entspricht etwa dem tatsächlich erhaltenen Ergebnis. Hierbei wird die höchste Effizienz von 76% ebenfalls mit acht Prozessen erreicht. Der Grund für das starke Absinken der Effizienz im weiteren Verlauf ist darin begründet, dass der *SpeedUp* durch den sequenziellen Anteil von *Amdahl's Law* immer gegen einen bestimmten Wert konvergiert. Die niedrigen Effizienzwerte entstehen, da der *SpeedUp* nicht proportional zur Anzahl der Prozesse ansteigt.

Es ist möglich, den *SpeedUp* und die Effizienz nur für die parallelisierte *Evaluation Time* zu berechnen, sodass der sequenzielle Teil des Verfahrens keinen Einfluss auf das Ergebnis hat. Die Ergebnisse hiervon sowie die des gesamten Verfahrens sind in Abbildung 5.5 dargestellt. Der *SpeedUp* der *Evaluation Time* ist nahezu linear. Mit neun *Slaves* bzw. zehn Prozessen wird ein *SpeedUp* von 8.7 erreicht. Diese guten Ergebnisse spiegeln sich in der Effizienz wieder. Diese ist anfänglich durch die *Master-Slave* Architektur bei 49%, steigt aber stetig an. Mit zehn Prozessen liegt sie bei 87%. Prinzipiell kann der Wert weiter ansteigen, da der Einfluss des *Masters* auf das Ergebnis mit zunehmender Prozessanzahl sinkt. Dennoch ist aufgrund der entstehenden Kommunikationszeit keine Effizienz von 100% möglich. Bleibt bei der Effizienzberechnung der *Master* Prozess unberücksichtigt, werden Ergebnisse zwischen 97% und fast 99% erreicht. Dies entspricht nahezu dem idealen Wert.

5.4.2 Pendulum

In diesem Kapitel wird die *Pendulum* Umgebung mit dem parallelisierten Verfahren evaluiert und bewertet. Zusätzlich erfolgt ein Vergleich mit den Ergebnissen des vorherigen Kapitels. Hierzu wird das Optimierungsproblem zunächst in verschiedenen Testdurchläufen mit zwei bis zehn Prozessen optimiert und jeweils die *Evaluation Time*, *Reproduction Time* und *Compose Offspring Time* erfasst. Die Konfiguration und der *Seed* werden vom sequenziellen Verfahren übernommen, sodass alle Testdurchläufe dieselben Lösungen produzieren und ein Vergleich der Implementierungen möglich ist. Wie bei der *Mountain Car* Umgebung wird von jedem Raspberry Pi maximal ein Prozess ausgeführt.

Die Ergebnisse zeigen, dass die Fitnesswerte und die finalen KNN identisch mit denen des sequenziellen Verfahrens sind, daher wird auf eine Abbildung verzichtet. Der Schwerpunkt der Analyse liegt auf den Ausführungszeiten. Diese sind von dem

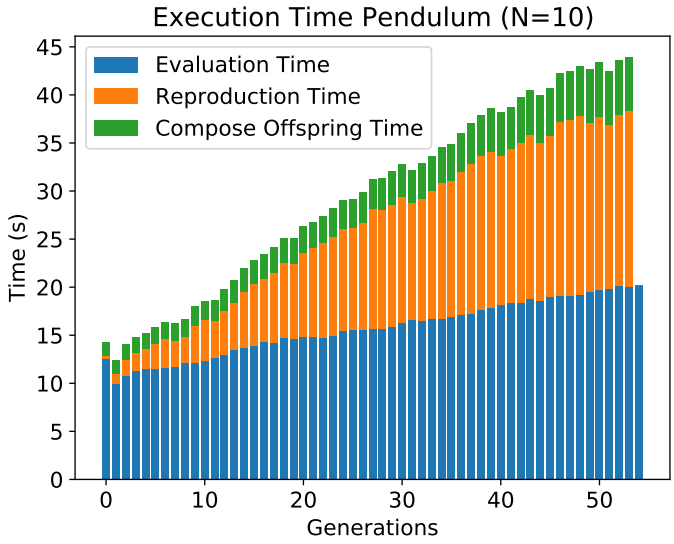

Abbildung 5.7: Ausführungszeit des *Pendulum* Problems auf 10 *Raspberry Pis* mit 10 Prozessen

zuletzt durchgeführten Test mit zehn Prozessen in Abbildung 5.8 dargestellt. Wie bei der *Mountain Car* Umgebung ist die *Evaluation Time* der ersten Generation höher als die der zweiten. Der Grund hierfür liegt in der Initialisierung der einzelnen Prozesse. Die gesamte Ausführungszeit in diesem Test beträgt 27 Minuten. Verglichen mit der Ausführungszeit der sequenziellen Implementierung von 138 Minuten ist dies eine Zeitersparnis von 111 Minuten. Zuletzt ist bezüglich dieses Graphen hervorzuheben, dass die *Evaluation Time* im Verlauf der Generationen nur leicht ansteigt und insgesamt 53% der Ausführungszeit ausmacht. Der Anteil der beiden anderen Phasen ist anfänglich gering, steigt aber stark an und benötigt zusammen in den letzten Generationen mehr Zeit als die eigentliche Evaluation. Wie bereits beim sequenziellen Verfahren beschrieben, ist die hohe Anzahl an verschiedenen Spezies der Grund für die vergleichsweise hohe *Reproduction Time*.

Als nächstes wird der Verlauf der gesamten Ausführungszeit in Abhängigkeit zur Anzahl an Prozessen betrachtet. Diese sind in Abbildung 5.8 dargestellt. Prinzipiell ähnelt der Graph dem der *Mountain Car* Evaluation. Mit zwei Prozessen steigt die Ausführungszeit im Vergleich zum sequenziellen Verfahren an. Wie in Kapitel 5.4.1 beschrieben, ist der Grund hierfür die gewählte *Master-Slave* Architektur. Mit steigender Anzahl an Prozessen nimmt die Ausführungszeit kontinuierlich ab. Gegen Ende wird die Zeitersparnis mit zunehmender Anzahl an Prozessen immer geringer. Grund hierfür sind die sequenziellen Phasen des Verfahrens.

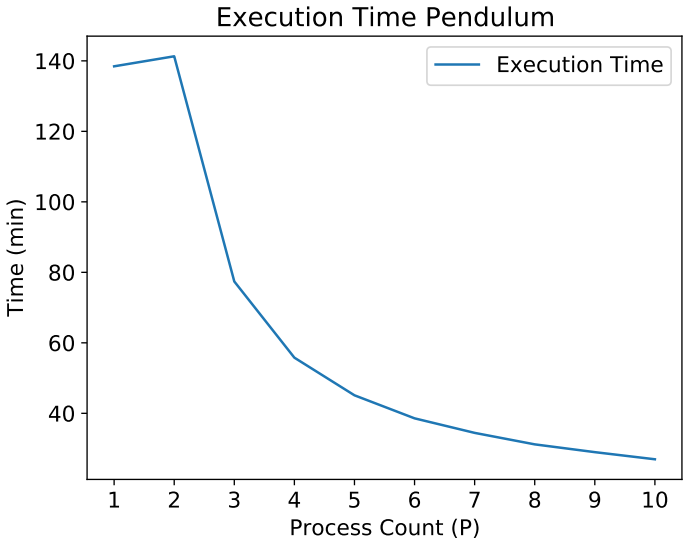

Abbildung 5.8: Ausführungszeit des parallelisierten Verfahrens in der *Pendulum* Umgebung in Abhängigkeit zur Prozessanzahl

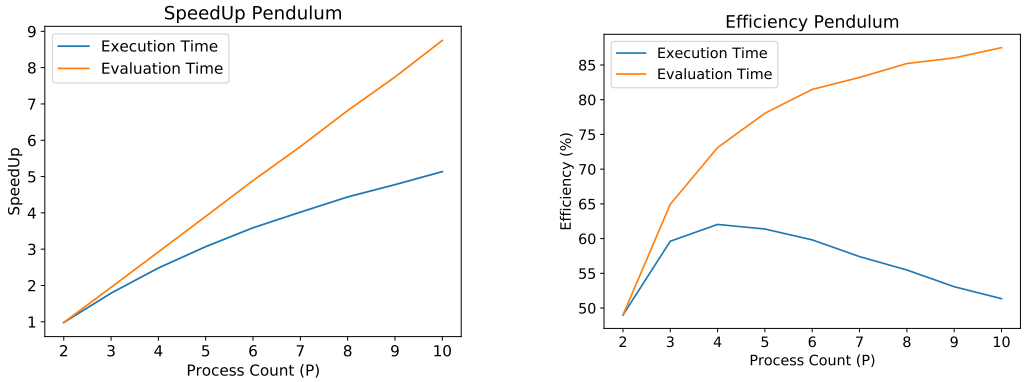

Abbildung 5.9: Links der *SpeedUp*, rechts die dazugehörigen Effizienzwerte für die *Pendulum* Umgebung in Abhängigkeit zur Prozessanzahl

Um die Ausführungszeiten und den Grad der Parallelisierung besser bewerten zu können, wird der *SpeedUp* und die Effizienz für jeden Durchlauf berechnet. Die Ergebnisse sind in Abbildung 5.9 dargestellt, wobei jeweils die Werte für das gesamte Verfahren und die parallelisierte *Evaluation Time* dargestellt sind. Auf Letzteres wird zuerst eingegangen. Der *SpeedUp* für die Evaluationsphase ist nahezu konstant. Mit zehn Prozessen wird ein Faktor von 8.75 erzielt. Dieses Ergebnis ähnelt dem der *Mountain Car* Umgebung, bei der in dieser Phase ein Faktor von 8.7 erreicht wird. Das Ergebnis ist insgesamt als sehr gut zu bewerten, da in der *Master-Slave* Architektur mit zehn Prozessen bzw. neun *Slaves* der maximal erreichbare *SpeedUp* 9 beträgt. Dies spiegelt sich auch in der Effizienz wieder. Bei

der *Master-Slave* Architektur liegt diese initial bei 49%, steigt aber stetig an. Mit zehn Prozessen wird ein Wert von 87% für die Parallelisierung der *Evaluation Time* erzielt. Die vergleichsweise schlechten initialen Messwerte sind durch die Berücksichtigung des *Masters* bei der Berechnung zu erklären, da er selbst keine Agenten evaluiert. Derselbe Umstand tritt in der *Mountain Car* Umgebung auf. Werden bei der Berechnung nur die zur Verfügung stehenden *Slaves* betrachtet, liegt die Effizienz in allen Testdurchläufen bei ungefähr 97%, maximal bei 98%. Aufgrund der notwendigen Kommunikation wird keine Effizienz von 100% erreicht. Zusammengefasst zeigen diese Ergebnisse, dass die Parallelisierung der *Evaluation Time* in beiden Testdurchläufen sehr gute und konstante Ergebnisse liefert. Es ist davon auszugehen, dass trotz Hinzufügen von weiteren Prozessen die Effizienz bei einer entsprechenden Populationsgröße weiterhin hoch ist. Hierauf wird im Rahmen der Ergebnisse in Kapitel 5.6 genauer eingegangen. In Abbildung 5.9 sind nicht nur der *SpeedUp* und die Effizienz für die Evaluationsphase, sondern auch das gesamten Verfahrens dargestellt. Die hierbei erhaltenen Werte sind nicht so gut wie in der *Mountain Car* Umgebung. Mit zehn Prozessen wird ein *SpeedUp* von 5.1 erreicht. Die *Mountain Car* Umgebung erzielt im Vergleich hierzu noch einen Wert von 7.6. Auch der Anstieg des *SpeedUps* sinkt innerhalb der ersten zehn Generationen drastisch. Grund hierfür ist der Anteil der nicht parallelisierten *Reproduction Time* und *Compose Offspring Time*. Wie in Kapitel 5.1 veranschaulicht, ist in der *Pendulum* Umgebung ein maximaler *SpeedUp* von $11.\overline{1}$ möglich. Mit steigender Anzahl an Prozessen nähert sich das Ergebnis diesem Wert an.

Entsprechend des *SpeedUps* sind auch die Effizienzwerte weniger gut als in der *Mountain Car* Umgebung. Anfangs liegt diese bei 42%, steigt danach an, erreicht mit vier Prozessen das Maximum von 62% und sinkt danach kontinuierlich. Hierfür sind dieselben Gründe maßgeblich wie in der *Mountain Car* Umgebung. Um die erhaltenen Ergebnisse besser einordnen zu können, werden mit *Amdahl's Law* die zu erwartenden Effizienzwerte berechnet. Diese sind in Abbildung 5.10 dargestellt. Aufgrund der unterschiedlichen Skalierung an den Achsen ist nicht offensichtlich, dass die gemessenen und idealen Ergebnisse sehr ähnlich sind. Die maximal erreichbare Effizienz von ungefähr 63.5% tritt ebenfalls mit vier Prozessen auf. Die übrigen Effizienzwerte für zwei bis zehn Prozesse weichen weniger als zwei Prozentpunkte vom tatsächlich erhaltenen Ergebnis ab.

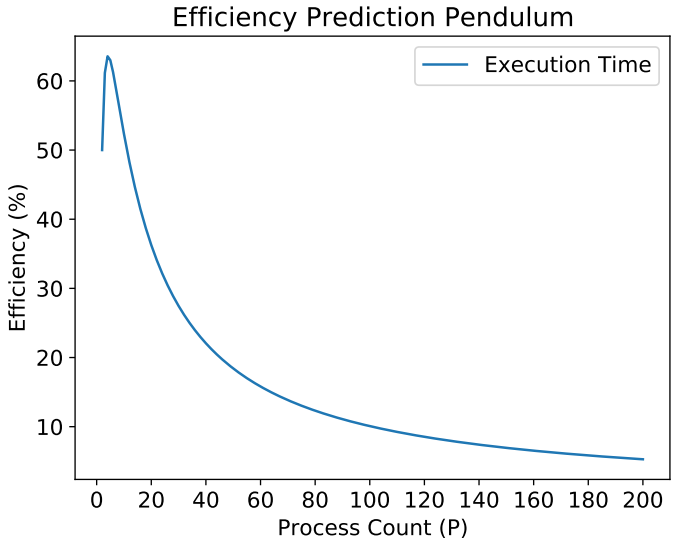

Abbildung 5.10: Erwartete Effizienz in der *Pendulum* Umgebung in Abhängigkeit zur Anzahl an Prozessen

5.4.3 Multi-Core CPUs

Die bisherigen Testdurchläufe nutzen maximal zehn Prozesse gleichzeitig, und zwar einen auf jedem Raspberry Pi. Mit dieser Konfiguration werden insgesamt sehr gute *SpeedUp* Ergebnisse erzielt. Prinzipiell ist es mit der bereits vorhandenen Hardware möglich, die Ausführungszeit weiter zu reduzieren, da die CPU der einzelnen Raspberry Pis nicht vollständig ausgelastet ist. Grund hierfür sind die Hardware und Funktionsweise der Programmiersprache Python. Wie in Kapitel 4.1 beschrieben, verfügt der Raspberry Pi 4 über einen Quad-core ARM Prozessor. Das bedeutet, dass sich vier unabhängige Prozessoren auf derselben CPU befinden, die parallel Instruktionen ausführen können. Allerdings kann der Standard Python Interpreter dies nicht nutzen, da ein *Global Interpreter Lock* (GIL) verwendet wird, um die Sprache *thread-safe* zu machen. Das bedeutet, dass nur ein *Thread* gleichzeitig ausgeführt werden kann. Existieren mehrere *Threads* wechselt der Python Interpreter in regelmäßigen Abständen zwischen diesen, sodass jedem eine gewisse Ausführungszeit zukommt. Die Folge ist, dass alle *Threads*, eines Programms auf demselben Prozessor abgearbeitet werden und keine Reduzierung der Ausführungszeit möglich ist [54]. Für das Umgehen des GILs gibt es verschiedene Ansätze, wobei eine Möglichkeit das Nutzen von MPI ist. Die parallelisierte Implementierung kann in der vorgestellten Testumgebung auf zehn Raspberry Pis mit insgesamt 40 Prozessen gestartet werden, sodass jeder Raspberry Pi vier Prozesse zugewiesen bekommt. Das Betriebssystem verteilt diese auf die vier verfügbaren Prozessoren und erzielt so eine parallele Ausführung. Bei dieser

Umsetzung verfügen die einzelnen Prozesse jeweils über einen eigenen GIL, der jedoch nicht mit denen von anderen Prozessen interferiert.

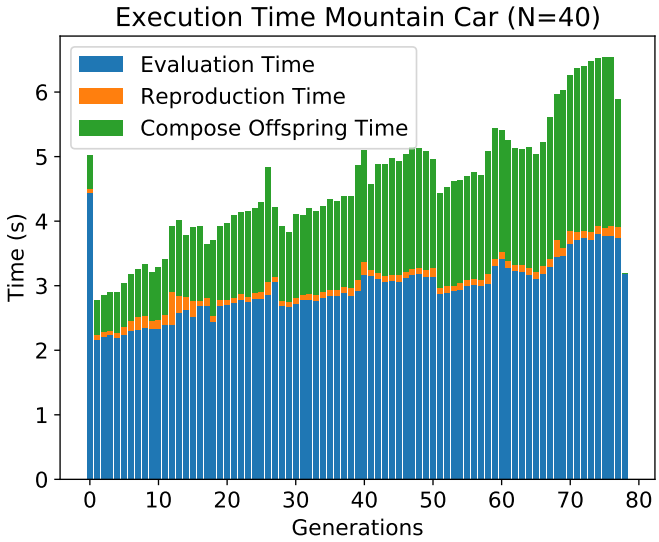

Abbildung 5.11: Ausführungszeit des *Moutain Car* Problems auf 10 *Raspberry Pis* mit 40 Prozessen

Um die Performance von diesem Szenario zu testen, werden die *Mountain Car* und *Pendulum* Umgebung erneut ausgeführt. Aufgrund der vorherigen Ergebnisse ist zu erwarten, dass die Ausführungszeit der *Evaluation Time* ungefähr um den Faktor vier im Vergleich zu zehn Prozessen verringert wird. Abweichungen können durch andere Hintergrundprozesse oder das Betriebssystem entstehen, mit denen die verfügbaren Rechenkapazitäten geteilt werden müssen. Abbildung 5.11 zeigt die gemessenen Ausführungszeiten für die *Mountain Car* Umgebung mit 40 Prozessen. Diese ist grundsätzlich im Vergleich zu den vorherigen Testdurchläufen gesunken. Insgesamt werden etwa sechs Minuten für die gesamte Ausführung benötigt, dies entspricht einer Zeitersparnis von 99 Minuten gegenüber dem sequenziellen Verfahren. Im Vergleich zu dem Testdurchlauf mit zehn Prozessen ist die Ausführungszeit um 8 Minuten gesunken. Um eine Bewertung der Parallelisierung zu ermöglichen, wird der *SpeedUp* und die Effizienz für das gesamte Verfahren sowie die *Evaluation Time* berechnet. Der *SpeedUp* für das gesamte Verfahren beträgt im Vergleich zur sequenziellen Implementierung ungefähr 17.6, die Effizienz liegt bei 44%. Diese Ergebnisse entsprechen nicht den Erwartungen. Nach *Amdahl's Law* soll ein *SpeedUp* von ungefähr 22.5 erzielt werden, beziehungsweise von 22.1, wenn der *Master* Prozess bei der Berechnung nicht berücksichtigt wird. Bei Betrachtung der *Evaluation Time* sind die erhaltenen Ergebnisse ebenfalls geringer als erwartet.

Für diese Phase wird ein *SpeedUp* von 26.7 erzielt, was einer Effizienz von ungefähr 67% entspricht. Selbst wenn der *Master* Prozess keine Berücksichtigung findet und ausschließlich die *Slaves* betrachtet werden, liegt die Effizienz nur bei 68.4%. Dieses Ergebnis ist bedeutend geringer als die mit zwei bis zehn Prozessen erreichte Effizienz 97% bis 99%. Auch wenn bei der *Pendulum* Umgebung etwas bessere Ergebnisse erzielt werden, skaliert die Leistung nicht wie erwartet. Der *SpeedUp* für die *Evaluation Time* beträgt ungefähr 29.6 und ergibt eine Effizienz von 74%, beziehungsweise 76%, wenn der *Master* nicht in die Berechnung miteinbezogen wird. Auch dies liegt über 20 Prozentpunkte unter den vorherigen Ergebnissen.

Für den Leistungseinbruch können verschiedene Faktoren ursächlich sein. Offensichtlich ist, dass an einer Stelle des Systems ein Ressourcenengpass entsteht. Um den Grund hierfür zu lokalisieren, werden verschiedene Tests durchgeführt, auf deren Ergebnisse im Folgenden eingegangen wird. Zuerst wird der *Master* Prozess untersucht. Dieser muss die Arbeitspakete mit einer geringen Latenz verteilen, sodass die *Slaves* geringe Wartezeiten haben. Aufgrund der hohen Anzahl an Prozessen ist dies unter Umständen nicht gewährleistet. Um diese These zu prüfen, wird das Optimierungsproblem angepasst. Jeder Agent muss bei der Evaluierung dieselbe Umgebung zehnmal absolvieren. Dies ändert nicht das Ergebnis, erhöht aber die Ausführungszeit stark. Zwar ist die hierbei entstehende Last auf den *Master* unverändert, sie wird jedoch über eine größere Zeitspanne verteilt. Falls beim Master Prozess der Ressourcenengpass auftritt, müsste durch diese Maßnahme eine Steigerung der Effizienz erkennbar sein. Dies ist jedoch nicht der Fall. Die Ergebnisse zeigen nur eine geringfügige Steigerung. Daher kann der *Master* Prozess als Engpass der Implementierung ausgeschlossen werden.

Als nächstes wird die Hard- bzw. Software des Beowulf Clusters überprüft, wobei der nächste Test verifiziert, ob der Ressourcenengpass bei den einzelnen *Nodes* entsteht. Hierzu werden neun Prozesse auf drei Raspberry Pis ausgeführt, wobei auf einem Gerät der *Master* Prozess und auf den anderen jeweils vier *Slave* Prozesse gestartet werden. Mit dieser Konfiguration beträgt die Ausführungszeit des Verfahrens fast 20 Minuten und ein *SpeedUp* von 5.3 wird erzielt. Der Anteil der *Evaluation Time* beträgt hierbei 17.7 Minuten. Diese Werte sind im Vergleich zum Durchlauf auf neun Raspberry Pis substanziell schlechter. Trotz derselben Anzahl an Prozessen ist die benötigte Ausführungszeit für das gesamte Verfahren um 28% höher und somit um 4.4 Minuten langsamer. Bei der Pendulum Umgebung treten ähnliche Ergebnisse auf. Daraus lässt sich folgern, dass mehrere Prozesse

```
import gym
import time

env = gym.make("Pendulum-v0")
env.reset()

start_time = time.time()
for _ in range(160000):
  env.step(env.action_space.sample())

time_required = time.time() - start_time
print(time_required)
```

Abbildung 5.12: Python Programmcode zum Überprüfen der Parallelisierung des OpenAI Gyms

auf demselben Raspberry Pi sich teilweise gegenseitig blockieren und dadurch eine längere Ausführungszeit entsteht.

Zuletzt ist festzustellen, durch welche Komponenten des Algorithmus die Verzögerung entsteht. In einem ersten Schritt wird eine Umgebung des OpenAI Gyms untersucht. Abbildung 5.4.3 zeigt den erstellten Test. Es wird zuerst die *Pendulum* Umgebung initialisiert und danach 160.000 Zufallsaktionen ausgeführt. Die hierfür benötigte Zeit wird gemessen und am Ende dargestellt. Der Test wird zunächst mit einem Prozess ausgeführt, dabei wird eine Laufzeit von 75.6 Sekunden gemessen. Danach wird der Test mit vier Prozessen auf einem Raspberry Pi wiederholt. Der zu leistende Rechenaufwand wird in diesem Fall nicht geteilt. Daher sollte jeder Prozess 160.000 Zufallsaktionen in einer lokalen Umgebung ausführen und dafür dieselbe Zeit benötigten. Allerdings zeigen die Messergebnisse einen Anstieg von 27% auf 95.5 Sekunden. Ein vergleichbarer Test wird mit einem KNN durchgeführt. Mit einem Prozess werden hierfür 73 Sekunden benötigt. Beim der Ausführung mit vier Prozessen steigt die Laufzeit um ungefähr 9.5% auf 80 Sekunden an. Da in den letzten beiden Tests keine Nachrichten ausgetauscht werden, ist zu schlussfolgern, dass die zuvor festgestellten Leistungseinbrüche mit 40 Prozessen nicht durch einen zu großen Kommunikationsaufwand, sondern vor allem durch die verwendeten Optimierungsprobleme entstehen.

5.5 Lunar Lander

Die vorherigen Kapitel beschreiben die Ergebnisse des parallelisierten Verfahrens. Mit 40 Prozessen wird ein *SpeedUp* für die *EvaluationTime* von 26.7 und 29.6

gemessen. Allerdings erfüllen die in diesen Umgebungen optimierten KNN nicht die Eigenschaft der Generalisierung. Jedes KNN beginnt die Evaluation immer in derselben Startposition und erzielt mit dieser gute Optimierungsergebnisse. Wenn diese KNN in einer Umgebung mit einer abweichenden Startposition starten, besteht eine hohe Wahrscheinlichkeit, dass nicht das gewünschte Ergebnis erreicht wird. Bei der durchgeführten Analyse wird keine generelle Lösung für alle Startpositionen benötigt. Diese wäre zudem aufgrund der hohen Trainingszeiten ungeeignet. In diesem Kapitel wird ein letztes Optimierungsproblem aus dem OpenAI Gym mit dem Namen *Lunar Lander* vorgestellt. Dieses Beispiel zeigt, wie eine generelle Lösung für alle Startsituation entwickelt werden kann und die hierfür benötigten Laufzeiten. Das Verfahren wird ausschließlich mit dem parallelisierten Algorithmus durchgeführt. Mit den zuvor gemessenen *SpeedUp* Werten kann jedoch auf die Laufzeit des sequenziellen Verfahrens geschlossen werden.

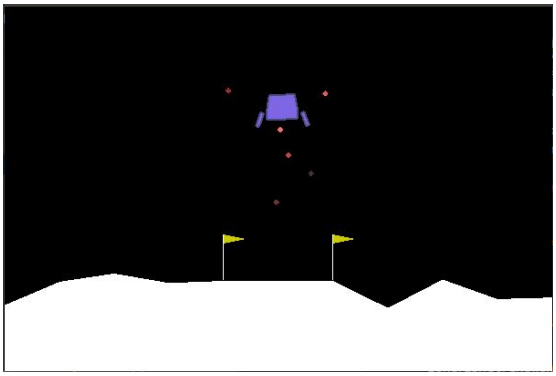

Abbildung 5.13: Darstellung der *Lunar Lander* Umgebung aus dem OpenAI Gym

Abbildung 5.13 zeigt ein Raumschiff und eine Landeplattform, die sich zwischen zwei Fahnen befindet. Ziel der Umgebung ist das Landen des Raumschiffs auf der Plattform. Für das Optimierungsproblem stehen insgesamt 8 Eingabewerte zur Verfügung, welche die Position, Geschwindigkeit und den Winkel des Raumschiffs beschreiben. Die Koordinaten der Landeplattform sind nicht enthalten, da sich diese immer an derselben Position befindet. Als Aktion kann entweder die sich unten am Raumschiff befindende Hauptdüse oder eine Steuerdüse an der linken oder rechten Seite des Raumschiffs aktiviert werden. Ebenfalls ist es möglich, den Antrieb nicht zu aktivieren und Treibstoff zu sparen. Die Simulation der Umgebung wird beendet, wenn das Raumschiff entweder abstürzt oder erfolgreich landet. Wie bei den anderen Umgebungen des OpenAI Gyms wird für jeden Zeitschritt ein *reward* vergeben. Diese werden während der Simulation summiert

und bilden den Fitnesswert. Ein positiver *reward* wird vergeben, wenn das Raumschiff sinkt und sich der Plattform nähert. Zusätzlich gibt es einen Bonus, wenn das Raumschiff sich im Ziel befindet oder die Landefüße den Boden berühren. Für jeden Zeitschritt, in dem ein Antrieb aktiviert ist, wird ein kleiner Betrag vom *reward* subtrahiert. Somit muss für die Maximierung des Fitnesswertes das Raumschiff so wenig Treibstoff wie möglich verbrauchen. Im Falle eines Absturzes gibt es einen negativen *reward*.

Für die Optimierung werden die Parameter der *Pendulum* Umgebung übernommen. Somit steht eine Population von 1000 Agenten zur Verfügung. Jedes KNN besitzt entsprechend der Ein- und Ausgabewerte acht *Input*- und vier *Output*-Neuronen. Das Optimierungsproblem ist so konfiguriert, dass jeder Startzustand zufällig gewählt wird. Um eine generelle Lösungsstrategie zu finden, muss jeder Agent zehn Durchläufe in der Umgebung absolvieren. Für den finalen Fitnesswert werden die Ergebnisse der einzelnen Durchläufe summiert. Das Verfahren wird beendet, wenn ein Agent in jedem der zehn Durchläufe einen Fitnesswert von über 200 Punkten erreicht. So ist sichergestellt, dass der Agent auf verschiedene Startsituationen entsprechend reagieren kann. Abbildung 5.14 zeigt das finale KNN und den Verlauf

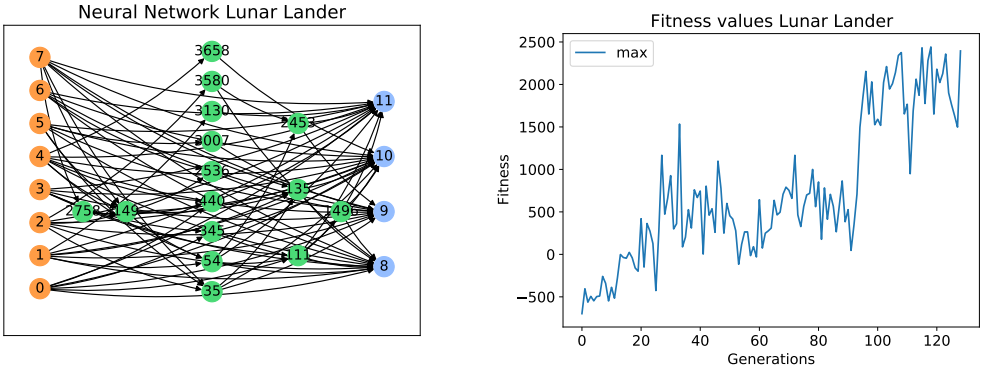

Abbildung 5.14: Links die Lösung für das *Lunar Lander* Problem, rechts die dazugehörigen Fitnesswerte pro Generation

des maximalen Fitnesswertes. Das KNN hat 15 *Hidden*-Neuronen entwickelt und besitzt insgesamt 88 Verbindungen. Die Fitnesswerte sind anfänglich negativ, da das Raumschiff häufig abstürzt. Im weiteren Verlauf steigt der Fitnesswert zwar an, zeigt aber große Schwankungen aufgrund einer verrauschten Fitnessfunktion. Der Grund hierfür ist, dass nicht alle, sondern nur ein Teil der Startzustände evaluiert werden. So kann ein Agent hohe Fitnesswerte aufgrund günstiger Startpositionen erhalten, wie es zum Beispiel in Generation 33 geschehen ist. In dieser hat der beste Agent einen Fitnesswert von 1533 Punkten erzielt. Obwohl er unverändert

in die nächste Generation kopiert wird, sinkt der maximale Fitnesswert auf 88 Punkte. In Generation 94 wird ein große Steigerung des Fitnesswertes auf 1495 Punkte erzielt. Das Verfahren endet nach 128 Generationen mit einem Fitnesswert von 2394 Punkten. Eine Besonderheit ist, dass der maximale Fitnesswert von 2441 Punkten nicht am Ende des Verfahrens, sondern in Generation 118 erzielt wird. Da jedoch die Abbruchbedingung nicht erfüllt ist, wird das Verfahren fortgeführt. Bei der Visualisierung des final entwickelten KNN wird ersichtlich, dass das Verfahren grundsätzlich erfolgreich ist. Der Agent landet in vielen Fällen direkt auf der Zielplattform und erreicht Fitnesswerte zwischen 240 bis 280 Punkten. Allerdings gibt es auch Durchläufe, in denen der Agent abdriftet. In diesen Fällen landet das Raumschiff nicht auf der Plattform, dennoch werden Fitnesswerte von ungefähr 200 Punkten erreicht. Insgesamt hat das Optimierungsverfahren eine generelle Lösungsstrategie entwickelt, aber die erreichten Fitnesswerte können aufgrund der unterschiedlichen Startpositionen dennoch abweichen. Eine bessere Leistung kann durch die Evaluation von weiteren Startzuständen oder eine höhere Trainingszeit erzielt werden.

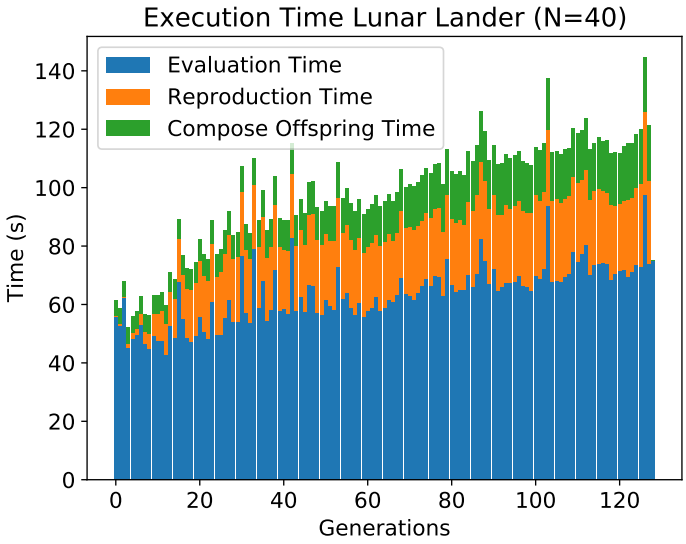

Abbildung 5.15: Ausführungszeit des *Lunar Lander* Problems auf 10 *Raspberry Pis* mit 40 Prozessen

Abbildung 5.15 zeigt die gemessenen Ausführungszeiten mit 40 Prozessen auf zehn Raspberry Pis. Insgesamt hat das Verfahren etwa 3.5 Stunden benötigt. Hiervon werden ungefähr 65% für die *Evaluation Time* verwendet. Der zweitgrößte Faktor der Ausführungszeit ist die *Reproduction Time* mit ungefähr 22%, welcher wie bei der *Pendulum* Umgebung durch die vielen verschiedenen Spezies zu erklären ist.

Wie bei der *Mountain Car* Umgebung unterliegt die Ausführungszeit der einzelnen Generationen einigen Schwankungen, da die Evaluationszeit je nach Agent stark abweichen kann. Stürzt das Raumschiff direkt ab oder landet schnell, ist die Evaluationszeit kurz. Evaluationen mit einer hohen Flugzeit können hingegen vergleichsweise lange andauern. Auch die Größe des KNN kann ein entscheidender Faktor sein, sowohl bei der *Evaluation* als auch in den Phasen *Mutation* und *Rekombination*. Um in solchen Szenarien gute Effizienzwerte zu erhalten, ist die dynamische Zuteilung von Arbeitspaketen durch die *Master-Slave* Architektur besonders wichtig. Prinzipiell hätte diese Umgebung auch bei der Analyse in Kapitel 4 verwendet werden können, die lange Ausführungszeit ist jedoch nicht praktikabel. Mit den *SpeedUp* Werten aus der *Mountain Car* und *Pendulum* Umgebung ist auf die sequenzielle Ausführungszeit zu schließen. Bei einem *SpeedUp* Faktor von 26.7 bzw. 29.6 liegt die erwartete Ausführungszeit zwischen 62.1 und 68.7 Stunden.

5.6 Ergebnisse

Auf Basis der Analyse ist das parallelisierte Verfahren implementiert und getestet worden. Die hierbei erhaltenen Ergebnisse sind im Folgenden zusammengefasst. Der größte Einfluss bezüglich der Ausführungszeit ist die *Evaluation Time*, deren Anteil in beiden Verfahren über 90% beträgt. Somit kann durch eine Parallelisierung dieser Phase die größte Zeitersparnis erzielt werden. *Amdahl's Law* zeigt, dass durch den sequenziellen Anteil des Programmcodes trotz unendlich vieler Prozesse ein maximaler *SpeedUp* von 50 für die *Moutain Car* Umgebung und von $11.\overline{1}$ für die *Pendulum* Umgebung möglich ist. Diese Werte werden für einen Vergleich mit den tatsächlich erhaltenen Ergebnissen genutzt.

Für eine maximale Zeitersparnis wird die Evaluation der Agenten als Ganzes parallelisiert. Ein Vorteil ist, dass sowohl das Optimierungsproblem als auch die Funktionalität des KNN parallel ausgeführt werden. Zusätzlich ist bei dieser Umsetzung der entstehende Kommunikationsaufwand gering und das nachträgliche Integrieren von Bibliotheken wie Tensorflow und PyTorch möglich. Für die eigentliche Implementierung sind verschiedene Anforderungen definiert, die vollständig umgesetzt sind. Es wird eine *Master-Slave* Architektur für die Kommunikation verwendet. Diese ist einfach zu implementieren, vermeidet *Deadlocks* und ermöglicht eine dynamische Lastenverteilung. Letzteres ist insbesondere wichtig, wenn das parallelisierte Verfahren auf unterschiedlich leistungsfähigen Geräten ausgeführt wird oder die Evaluationszeit von Agenten variiert. Ein Nachteil dieser Umset-

zung ist, dass der *Master* nur die Koordination der Kommunikation übernimmt und nicht die *Slaves* bei der Evaluation unterstützt. Mit einer geringen Anzahl an Prozessen macht dieser Umstand bei der Effizienzberechnung einen großen Unterschied aus. In größeren Systemen mit hunderten Prozessen ist der Einfluss hingegen vernachlässigbar gering und daher nicht relevant. Bei der Implementierung des parallelisierten Verfahrens werden die Komponenten des sequenziellen Verfahrens überwiegend wiederverwendet. Hierdurch ist eine schnelle und effiziente Entwicklung möglich. Entscheidend ist, dass beide Verfahren einfach untereinander austauschbar sind und ein direkter Vergleich möglich ist. Um Letzteres zu gewährleisten, basiert auch das parallelisierte Verfahren auf einem *Seed*. Ist dieser identisch mit dem des sequenziellen Verfahrens, werden dieselben Ergebnisse berechnet. Für die Kommunikation wird der Standard MPI eingesetzt, da dieser im Bereich HPC weit verbreitet ist und zusätzlich das Umgehen des GILs von Python ermöglicht.

Für die Tests wird ein Beowulf Cluster bestehend aus zehn Raspberry Pis erstellt. Auf diesem wird das parallelisierte Verfahren durchgeführt und mit den Laufzeiten des sequenziellen Verfahrens verglichen. Zu Anfang wird das Verfahren mit jeweils einem Prozess pro Raspberry Pi durchgeführt. Insgesamt sind die hierbei erhaltenen Werte für den *SpeedUp* und die Effizienz nur geringfügig niedriger als die Vorhersage nach *Amdahl's Law*. Mit zehn Prozessen wird für das gesamte Verfahren ein *SpeedUp* für die *Mountain Car* und *Pendulum* Umgebung von 7.6 bzw. 5.1 erzielt. Da diese Werte durch den sequenziellen Anteil des Algorithmus beeinflusst sind, wird die parallelisierte *Evaluation Time* gesondert bewertet. Mit zehn Prozessen liegen die erzielten *SpeedUp* Werte bei 8.5 und 8.7. Werden ausschließlich die *Slaves* bei der Berechnung der Effizienz berücksichtigt, beträgt diese in jedem Testdurchlauf mindestens 97%. Dies verdeutlicht die Effizienz der Kommunikation und den Erfolg der Parallelisierung.

Mit MPI ist es möglich, das GIL von Python zu umgehen. Damit kann das parallelisierte Verfahren alle 40 Prozessoren des erstellten Beowulf Clusters vollständig auslasten. Zwar ist das Verfahren grundsätzlich schneller als mit zehn Prozessen, allerdings skaliert der *SpeedUp* in diesem Fall nicht so gut wie erwartet. Verschiedene Tests belegen, dass nicht die höhere Anzahl der beteiligten Prozesse hierfür ursächlich ist. Diese Erkenntnis ist entscheidend für die Ausführung des Algorithmus in größeren und leistungsfähigeren Clustern. Auf Basis der durchgeführten Tests ist stattdessen zu schlussfolgern, dass mehrere Prozesse

auf demselben Raspberry Pi sich gegenseitig blockieren. Dieses Phänomen ist unabhängig von der Anzahl der beteiligten Prozesse. Zur Feststellung der exakten Ursache sind weitere Tests erforderlich, die nicht Teil dieser Arbeit sind. Ein Ressourcenengpass kann möglicherweise beim Zugriff auf Systemressourcen wie dem RAM entstehen. Ist die hierfür bereitgestellte Bandbreite nicht für alle vier Prozessoren gleichzeitig ausreichend, kommt es zu Verzögerungen und somit niedrigeren *SpeedUp* Werten.

Die zuletzt vorgestellte *Lunar Lander* Umgebung dient als Beispiel für die Entwicklung einer generellen Lösungsstrategie. Im Vergleich zu den vorherigen Umgebungen verwendet diese zufällige Startzustände und das KNN muss lernen, auf diese zu reagieren. Dies ist anspruchsvoller als die Entwicklung einer Lösung für einen einzelnen Startzustand. Hierbei ist ein grundsätzliches Problem, dass die Fitnessfunktion nicht alle Startzustände miteinbezieht. Schlechte KNN mit einfachen Startzuständen können gute Fitnesswerte erzielen, vielversprechende KNN in schwierigen Umgebungen hingegen nur niedrige Fitnesswerte. Die Gefahr hierbei ist, dass vielversprechende Lösungsansätze aufgrund niedriger Fitnesswerte bei der Selektion nicht ausgewählt werden und verloren gehen. Um dies zu verhindern und einen aussagekräftigen Fitnesswert zu erhalten, wird in diesem Beispiel die Umgebung zehnmal nacheinander mit unterschiedlichen Startzuständen evaluiert. Das Verfahren wird erst dann beendet, wenn für jeden Durchlauf eine gültige Lösung gefunden ist. Wie das Beispiel zeigt, ist dieses Vorgehen erfolgreich. Ein großer Nachteil besteht darin, dass die Evaluationszeit stark erhöht wird. Hier zeigt sich der große Vorteil der Parallelisierung. Mit den zuvor gemessenen *SpeedUp* Werten wird angenommen, dass die Ausführungszeit mit dem sequenziellen Verfahren zwischen 62.1 und 68.7 Stunden liegt. In der parallelisierten Umgebung wird das Verfahren hingegen bereits nach 3.5 Stunden beendet. Es ist davon auszugehen, dass durch Hinzufügen von weiteren Prozessoren die Ausführungszeit entsprechend *Amdahl's Law* weiter gesenkt werden kann. Dieses Beispielproblem verdeutlicht erneut den Erfolg der Parallelisierung.

6 Zusammenfassung und Ausblick

In dieser Arbeit ist das Verfahren NEAT analysiert und für ein verteiltes System parallelisiert worden. Die Ergebnisse der vorherigen Kapitel werden im Folgenden zusammengefasst. Zusätzlich sind einige Vorschläge für zukünftige Weiterentwicklungen gegeben, die nicht im Rahmen dieser Arbeit umgesetzt sind.

6.1 Ergebnis

Die erste Anforderung an diese Arbeit ist die Implementierung des sequenziellen Verfahrens. Hierfür ist die Sprache Python gewählt worden, da sie im Bereich maschinelles Lernen weit verbreitet ist und viele Bibliotheken bietet. Eine hiervon ist das OpenAI Gym, welches bei der Analyse und Evaluation genutzt wird. Die generelle Softwarearchitektur ist in Kapitel 3 beschrieben. Sie ist darauf ausgelegt, eine einfach zu nutzende Schnittstelle zu bieten. Dadurch können verschiedene Optimierungsprobleme schnell und effizient implementiert werden. Ein weiterer wichtiger Aspekt ist, dass das Ergebnis des Verfahrens durch einen *Seed* beeinflusst wird. Dadurch ist das Wiederholen und Vergleichen von Testergebnisse mit der später erstellten parallelisierten Implementierung möglich. Zuletzt ist bei der Softwarearchitektur zu verdeutlichen, dass das optimierte KNN und die Testergebnisse gespeichert und visualisiert werden können. Dies ist unter anderem notwendig, wenn das KNN in einer produktiven Umgebung eingesetzt werden soll.

Kapitel 4 beschreibt die Analyse des sequenziellen Verfahrens. Dies entspricht der zweiten Anforderung an diese Arbeit. Zuerst wird die korrekte Funktionalität des implementierten Algorithmus mit dem XOR-Problem überprüft. Mit der Konfiguration aus der originalen Publikation werden ähnliche Ergebnisse erzielt, wie sie von den Autoren beschrieben sind. Inspiriert durch das Paket *neat-python* werden die Konfiguration und das Verfahren modifiziert. Dadurch sinkt die Anzahl der benötigten Generationen zum Lösen des XOR-Problems beträchtlich. Im nächsten Schritt der Analyse wird die Ausführungszeit des sequenziellen Verfahrens in verschiedenen Beispielen erfasst. Das XOR-Problem wird wegen einer zu geringen

Komplexität hierfür nicht verwendet. Es werden die *CartPole*, *MountainCar* und *Pendulum* Umgebung des OpenAI Gym ausgewählt. Bei allen drei Umgebungen kann die Fitnessfunktion mit den zur Verfügung gestellten *rewards* implementiert werden. Die Testdurchläufe zeigen, dass die initiale Generation die *CartPole* Umgebung bereits lösen kann und sie daher für die weiteren Tests ungeeignet ist. Bei den anderen beiden Umgebungen werden die Ausführungszeiten gemessen. Hierbei wird zwischen der *Evaluation Time*, *Reproduction Time* oder *Compose Offspring Time* unterschieden. Die *Evaluation Time* misst die benötigte Zeit zum Evaluieren aller Agenten. Diese macht in beiden Testumgebungen mit Abstand den größten Anteil an der gesamten Laufzeit des Algorithmus aus.

Die letzte Anforderung an diese Arbeit ist die Optimierung, welche in Kapitel 5 beschrieben ist. Hierbei wird der Fokus auf die Parallelisierung der *Evaluation Time* gelegt, da hierbei die größte Zeiteinsparung möglich ist. Es wird eine *Master-Slave* Architektur mit MPI implementiert. Der *Master* verteilt Arbeitspakete, die je einen Agenten enthalten und von den *Slaves* evaluiert werden. Der Vorteil von diesem Ansatz ist, dass sowohl das KNN als auch das Optimierungsproblem parallelisiert ausgeführt werden. Der dabei entstehende Kommunikationsaufwand ist sehr gering und ermöglicht insgesamt eine gute Parallelisierung. Zusätzlich kann die bereits vorhandene Struktur einfach erweitert werden, sodass auch die Parallelisierung der *Reproduction Time* oder *Compose Offspring Time* möglich ist. Das Verfahren wird auf einem verteilten System ausgeführt. Hierfür wird ein Beowulf Cluster aus zehn Raspberry Pis erstellt. Neben geringen Kosten und einer platzsparenden Unterbringung bietet dies den Vorteil, dass alle Geräte identisch konfiguriert sind. Das vereinfacht den Vergleich zwischen der sequenziellen und parallelisierten Implementierung. Um die Effizienz der Parallelisierung zu messen, werden die zuvor durchgeführten Tests wiederholt und die Ausführungszeit erfasst. Auf jedem beteiligten Raspberry Pi wird hierzu ein Proezss ausgeführt. Bei Verwendung von zwei oder mehr *Slaves* wird die Ausführungszeit stark reduziert. Für eine Bewertung der Parallelisierung sind der *SpeedUp* und Effizienzwert berechnet worden. Die erhaltenen Ergebnisse entsprechen nahezu den idealen Werten, die mit *Amdahl's Law* berechnet sind. Werden nur die *Slaves* betrachtet, ist die *Evaluation Time* mit einer Effizienz von 97% parallelisiert. Dies ist ein sehr gutes Ergebnis und wird unter anderem durch die effiziente Kommunikation ermöglicht. Für ein Population mit n Mitgliedern müssen nur $2 \cdot n$ Nachrichten versendet werden. Zusätzlich ist in Kapitel 5 demonstriert, dass mit MPI das GIL von Python umgegangen werden kann. Es können alle 40 Prozessoren des erstellten Beowulf

Clusters vollständig ausgelastet werden. Allerdings sind die dabei entstehenden *SpeedUp* Werte niedriger als durch die vorherigen Tests angenommen. Der Grund ist, dass sich mehrere Prozesse auf einem Gerät sich gegenseitig blockieren bzw. behindern. Die genaue Ursache hiervon kann im Rahmen dieser Arbeit nicht untersucht werden. Die Tests zeigen jedoch, dass dies nicht primär durch das implementierte Verfahren entsteht sondern unter anderem durch das OpenAI Gym.

Insgesamt haben die erzielten *SpeedUp* und Effizienzwerte die Erwartungen übertroffen. Durch eine gute Parallelisierung können neuroevolutionäre Algorithmen einen entscheidenden Vorteil gegenüber dem Backpropagation Algorithmus und seinen Derivaten bieten. Der große Vorteil ist, dass sowohl das KNN als auch das Optimierungsproblem parallelisierbar sind. Durch die erfassten Ergebnisse ist zu schließen, dass mit steigender Anzahl an Prozessoren die Ausführungszeit weiter sinken wird. Die zuletzt vorgestellte *Lunar Lander* Umgebung verdeutlicht nochmals, wie groß die tatsächliche Einsparung ist. Verfahren die auf einem Raspberry Pi 4 mehrere Tage dauern, können auf zehn Geräten in wenigen Stunden beendet werden. Für eine vollständige Bewertung ist auch zwei Nachteile zu nennen. Gibt es mehr Prozesse als Agenten in einer Generation, kann nicht jedem ein Arbeitspaket zugewiesen werden. In diesem Fall kann durch Hinzufügen von weiteren Rechenressourcen keine Reduzierung der Ausführungszeit erreicht werden. Jedoch ist es in einer solchen Situation möglich die Populationsgröße entsprechend anzupassen. Mit einer größeren Population können mehr Lösungsansätze gleichzeitig evaluiert werden was indirekt schneller zu einer Lösung führen kann. Der zweite Nachteil betrifft NEAT direkt. Grundsätzlich werden mit diesem Algorithmus beeindruckende Ergebnisse erzielt. Jedoch können die vorgestellten Arten der strukturellen Mutation nicht ausreichend sein um große KNN in einer kurzen Zeit zu entwickeln. In diesem Fall kann das HyperNEAT Verfahren aus Quelle [55] verwendet werden. Dieses basiert auf NEAT und ermöglicht die Entwicklung von größeren KNN. Die in dieser Arbeit vorstellten Parallelisierungsstrategie bezieht sich vor allem auf die Evaluationsphase und kann daher auf verschiedene neuroevolutionäre Algorithmen angewendet werden.

6.2 Weiterentwicklung

Das implementierte Projekt bietet die Grundlage für verschiedene Erweiterungen. In Kapitel 5.1 ist bereits die Integration von Bibliotheken wie Tensorflow und PyTorch vorgestellt. Mit diesen kann die Aktivierungszeit von KNN stark reduziert

werden und auch die Nutzung von GPUs wird ermöglicht. Für einen noch besseren *SpeedUp* müssen die Funktionen der *Reproduction Time* und *Compose Offspring Time* ebenfalls parallelisiert werden. Denn durch *Amdahl's Law* ist erkennbar, dass die Phasen zwar beim sequenziellen Verfahren nur einen geringen Prozentsatz der Ausführungszeit benötigen, aber bei steigender Parallelisierung zunehmend zum limitierenden Faktor werden. Prinzipiell ist es mit der *Master-Slave* Architektur möglich beliebig viele weitere Funktionen mit geringem Aufwand zu parallelisieren. Die Funktionen für die Mutation der Gewichte und die Rekombination zum Erzeugen von Nachkommen bieten sich für eine Parallelisierung an. Für letzteres kann der *Master* Arbeitspakete mit je zwei Agenten erstellen. Die *Slaves* führen für diese die Rekombination aus und geben den neu erstellten Agenten als Ergebnis zurück. Selbes Prinzip kann auf die Mutation der Gewichte angewendet werden. Andere Funktionen hingegen können nur schlecht oder teilweise parallelisiert werden. Ein Grund hierfür können globale Variablen sein, wie es im Fall der strukturellen Mutationen mit den Innovationsnummern ist. Die Funktion zum sortieren der Agenten in die Spezies kann teilweise parallelisiert werden. Die Zuweisung der Agenten in die bestehenden Spezies kann parallel geschehen, das Erzeugen von neuen Spezies hingegen muss sequenziell durchgeführt werden. Insgesamt ist der Aufwand für die Parallelisierung dieser Phasen bedeutend höher und wird wahrscheinlich dieselben *SpeedUp* Ergebnisse liefern wie die *Evaluation Time*. Daher ist die Parallelisierung dieser Phasen häufig erst sinnvoll, wenn durch hinzufügen von weiteren Prozessen die Ausführungszeit der *Evaluation Time* nicht weiter reduziert werden kann.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Thesis ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen benutzt habe.

Ort und Datum	Unterschrift