HAW HAMBURG

BACHELOR THESIS
Jan Klett, MatrNr. [redacted]

Analyse des Einflusses von Zufallszahlengeneratoren auf den NeuroEvolution of Augmenting Topologies Algorithmus

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jan Klett, [redacted]

Analyse des Einflusses von Zufallszahlengeneratoren auf den NeuroEvolution of Augmenting Topologies Algorithmus

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Christian Lins
Zweitgutachter: Prof. Dr. Thomas Clemen

Eingereicht am: 14. Feb 2025

Jan Klett, MatrNr. [redacted]

Thema der Arbeit

Analyse des Einflusses von Zufallszahlengeneratoren auf den NeuroEvolution of Augmenting Topologies Algorithmus

Stichworte

Künstliche Intelligenz, Evolutionäre Algorithmen, NEAT, Zufallszahlengenerator, Pseudozufallszahlengenerator

Kurzzusammenfassung

In dieser Arbeit wird untersucht, wie sich unterschiedliche Zufallszahlengeneratoren auf den NeuroEvolution of Augmenting Topologies (NEAT) Algorithmus auswirken, indem die Pseudozufallszahlengeneratoren Xorshift, Mersenne Twister und Complementary-Multiply-With-Carry hinsichtlich ihrer Auswirkungen auf drei klassische Problemstellungen evaluiert werden: k-XOR, Spiraldaten-Klassifikation und Two-Pole-Balancing. Die Experimente analysieren, inwiefern Unterschiede in der Zufallszahlenerzeugung die Netzwerkevolution beeinflussen. Die resultierenden Leistungsmetriken werden mittels einer ANOVA-Analyse statistisch ausgewertet, um mögliche Unterschiede zwischen den PRNGs zu identifizieren. Die Ergebnisse liefern wertvolle Erkenntnisse über die Robustheit von NEAT gegenüber der Wahl des Zufallszahlengenerators und tragen zur besseren Charakterisierung evolutionärer Algorithmen bei.

Jan Klett, MatrNr. [redacted]

Title of Thesis

Analyzing the Impact of Random Number Generators on NeuroEvolution of Augmenting Topologies Algorithm

Keywords

Artifical Intelligence, Evolutionary Algorithms, NEAT, Random Number Generator, Pseudo Random Number Generator

Abstract

This study examines the effects of different random number generators on the NeuroEvolution of Augmenting Topologies (NEAT) algorithm by evaluating the pseudo-random number generators Xorshift, Mersenne Twister, and Complementary-Multiply-With-Carry in three benchmark problems: k-XOR, spiral data classification and two-pole balancing. The experiments analyze, whether differences in random number generation influence network evolution. The resulting performance metrics are statistically evaluated using an ANOVA to identify potential differences between the PRNGs. The findings provide valuable insights into the robustness of NEAT concerning the choice of PRNG and contribute to a better understanding of evolutionary algorithms.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Pseudocode-verzeichnis

Abkürzungen

ANOVA Analysis of Variance.

BCE Binary Cross Entropy.

CMWC Complemetary-Multiply-With-Carry.

DE Differential Evolution.

EA Evolutionäre Algorithmen.

KI Künstliche Intelligenz.

ML Maschinelles Lernen.

MT Mersenne Twister.

NEAT NeuroEvolution of Augmenting Topologies.

NFL No-Free-Lunch.

PRNG Pseudo-Random Number Generator.

RNG Random Number Generator.

XOR Exklusives Oder.

1 Einleitung

1.1 Motivation

Das wissenschaftliche Feld Künstliche Intelligenz (KI) und besonders des maschinellen Lernens (ML) gewinnt in der Welt immer mehr Bedeutung [12]. Die Fähigkeit von Systemen, aus bestehenden Daten zu lernen und dies auf neue Daten anzuwenden, findet heutzutage in allen möglichen Branchen und Bereichen Gebrauch. In diesem Kontext der KI zeigen sich Evolutionäre Algorithmen (EA) als leistungsstarke Werkzeuge der künstlichen Intelligenz, da sie auf den Prinzipien der biologischen Evolution basieren und sich als effektive Mittel zur Lösung komplexer Optimierungsprobleme bewährt haben [18].

Eine der Grundlagen von EAs sind die Verwendung von (Random Number Generators, RNGs). Sie werden in mehreren Aspekten der Algorithmen genutzt, darunter liegen meist die Rekombination, Mutation und die Initialisierung der Population [18]. Durch viele Jahre der Enwicklung gibt es nun die verschiedensten RNGs die sich in ihren statistischen Eigenschaften und ihrem Verhalten unterscheiden. Dies führt zu der Frage, ob die Wahl des RNGs einen Einfluss auf die Ergebnisse von EAs hat.

Während jedoch zahlreiche Studien den Einfluss von RNG auf andere EAs, wie DE [1, 5], untersucht haben, bleibt ein Algorithmus bisher weniger erforscht: der NEAT-Algorithmus.

Der NeuroEvolution of Augmenting Topologies (NEAT) Algorithmus ist hierbei eine Anwendung der EA im Bereich des ML. NEAT ermöglicht es, neuronale Netzwerke mit zunehmender Komplexität zu erkunden und zu optimieren. Eine Besonderheit des NEAT Algorithmus ist hierbei, das er keine randomisierte Initialpopulation besitzt. Somit fehlt eine mögliche Einflussquelle des RNGs, jedoch könnte dies auch den Einfluss in den anderen Quellen hervorheben. [26]

1.2 Ziel

Diesen Einfluss der RNGs zu untersuchen und diesen zu quantifiziern ist das Ziel dieser Thesis. Für diese Analyse der Effekte von RNGs soll, basierend auf der Fitness und der benötigenten Zeit, verglichen werden.

Diese Kennzahlen wurden gewählt, um die wichtigsten Aspekte des Algorithmus wiederzuspiegeln. Die Fitness bezieht sich hierbei sowohl auf die Entwicklung der Fitness über die Generationen, als auch auf die final erreichte Fitness des trainierten Modells. Die benötigte Zeit ist für diese Untersuchung in zwei Aspekte aufgeteilt: Die benötigten Generationen und die Ausführungszeit. Diese Aufteilung wurde gewählt, da die Ausführungszeit Hardware- und Situationsabhängig sein kann und somit ein eher subjektives Kriterium ist. Die Generationszahl hingegen ist hardwareunabhängig und objektiv, gibt aber keinen Aufschluss darüber, wie sich ein komplexerer RNG auf die Dauer des Trainings auswirkt.

Als Benchmark für diese Hypothese sollen im Rahmen dieser Untersuchung mehrere Problemstellungen geprüft werden. Als erste Problemstellung steht hierbei die Simulation eines exklusiven Oder-Gates (XOR), da es ein weit bekanntes, nicht-linear lösbares Problem darstellt und deshalb für viele Forscher als erster Test für neue Ansätze gilt [17, 26].

Die zweite Problemstellung soll die Klassifikation bieten. Angelehnt an die Untersuchungen von Evgenia Papavasileiou et al. [17] und Maxine Tan et al. [27] sollen Datenpunkte aus Spiral-Plot-Daten (Abbildung 1.1) klassifiziert werden.

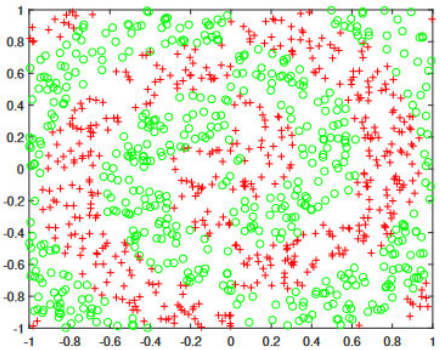

Abbildung 1.1: Ein Beispiel für Spiral-Plot-Daten [17]

Als letzte Problemstellung steht das Pole-Balancing, das bereits im originalen NEAT-Paper als Benchmark-Funktion Verwendung fand, getestet werden. Wie auch Kenneth O. Stanley et al. [26] soll die Variante mit zwei Poles getestet werden.

Als letztes ist noch die Wahl der RNGs entscheidend. In dieser Analyse werden MT, XORShift und CMWC untersucht, weil sie zu den meistgenutzten RNGs zählen, verschiedene Periodenlängen und verschieden Art der Zahlengenerierung aufweisen.

1.3 Aufbau der Arbeit

Im nächsten Abschnitt werden zunächst die Grundlagen von EA, der NEAT Algorithmus, die verwendeten RNGs und die Problemstellungen ausführlich vorgestellt. Des Weiteren wird auch auf die bestehende Forschung, im Bereich des RNG-Einflusses auf EA, und deren Verhältniss zu dieser Studie eingegangen.

Anschließend folgt in Abschnitt 3 eine Erläuterung des praktischen Aufbaus. Dies beinhaltet unter anderem die Implementation und Konfiguration der Experimente. In Abschnitt 4 folgt darauf die Analyse der Daten, die sich aus den Experimenten ergeben. Hierzu werden zunächst die verwendeten Analysemethoden und deren Anwendung erklärt und anschließend die graphischen und tabellarischen Ergebnisse vorgestellt.

In Abschnitt 5 wird daraufhin, basierend auf den Daten von Abschnitt 4, ein Urteil über den Effekt der RNGs auf den NEAT Algorithmus zu schließen und mögliche Konsequenzen dieses Urteils zu betrachten. Zuletzt wird in Abschnitt 6 noch ein Fazit zu der Forschungsfrage, der Durchführung und mögliche Erweiterungen der Forschung gegeben.

2 Stand der Technik

2.1 Was sind Evolutionäre Algorithmen

Evolutionäre Algorithmen (EA) stellen eine Klasse von Optimierungs- und Suchverfahren dar, die sich an den Prinzipien der natürlichen Evolution orientieren. Sie basieren auf zentralen Mechanismen wie Selektion, Mutation, Rekombination und dem Prinzip des Überlebens des best Angepassten, um robuste und oft überraschend effiziente Lösungen für komplexe Probleme zu finden. [2]

Der Prozess beginnt typischerweise mit einer Population von meist zufällig erzeugten Kandidatenlösungen. Diese Population wird über mehrere Iterationen hinweg, die üblicherweise als Generationen bezeichnet werden, durch die Anwendung genetischer Operatoren fortentwickelt. Ziel ist es, innerhalb dieses iterativen Prozesses entweder eine optimale oder zumindest eine zufriedenstellende Lösung zu identifizieren. [2]

Die Kernprinzipien der EA lassen sich auf die Evolutionstheorie von Darwin zurückführen. Durch Selektion werden die Kandidatenlösungen mit der höchsten Anpassungsfähigkeit (Fitness) bevorzugt in die nächste Generation übernommen. Mutation und Rekombination ermöglichen es, neue Lösungen zu generieren und so den Suchraum effektiver zu explorieren. [2]

Evolutionäre Algorithmen haben sich als besonders leistungsfähig bei der Lösung von Problemen erwiesen, die für klassische, mathematische oder deterministische Ansätze nur schwer zugänglich sind. Dazu gehören beispielsweise hochdimensionale Optimierungsprobleme, kombinatorische Herausforderungen oder Multi-Objective-Probleme. In der Praxis finden sie Anwendung in diversen Bereichen, darunter Maschinenlernen, Ingenieurwissenschaften und Betriebswirtschaft. [2]

Durch ihre Fähigkeit, globale Lösungen auch in hochkomplexen Suchräumen zu identifizieren, haben EA eine große Relevanz in der Wissenschaft und Technik. Insbesondere der Ansatz, biologische Prozesse zu abstrahieren und algorithmisch umzusetzen, hat

zur Weiterentwicklung vieler verwandter Methoden, wie z. B. genetischer Algorithmen, Differential Evolution und evolutionärer Strategien, beigetragen. Dies unterstreicht die Flexibilität und die vielseitige Einsetzbarkeit dieser Verfahren. [16]

2.1.1 Grundlegende Funktionsweise

Evolutionäre Algorithmen basieren, wie bereits erwähnt, auf der natürlichen Evolution. Ein EA arbeitet mit zwei Typen von Objekten, die Genotypen und Phenotypen.

Genotypen beschreiben eine maschinenlesbar und -bearbeitbare Darstellung eines Lösungansatzes. Häufig enthalten sie neben den Informationen für den Phenotypen noch zusätzlich Informationen, die für die evolutionären Mechanismen notwendig sind. Sie ensprechen der DNA in der Natur und bilden einen Bauplan für einen Phenotypen. Die Anzahl von Genotypen wird in der Initialisierung festgelegt und wird auch als Populationsgröße bezeichnet. Diese Populationsgröße bleibt während des gesamten Ablaufs des Algorithmus konstant. Deshalb ist die Populationsgröße ein wichtiger Parameter für die Optimierung des Algorithmus. Zu große Populationen können zu einer langsamen Konvergenz und langer Ausführung führen, während zu kleine Populationen zu einer schlechten Konvergenz und schlechten Ergebnissen führen können. [2]

Phenotypen wiederrum sind die nutzbaren Ausprägungen eines Genotypen. Sie sind die Lösungen, die während des Algorithmus getestet werden und das Resultat. Dabei ensprechen sie dem Körper, der basierend auf der DNA gebaut wurde.

Die Phenotypen können entweder basierend auf einem Fitnesswert oder einem Losswert bewertet werden. Wie diese Fitness oder das Loss berechnet wird, unterscheidet sich zwischen jeder Anwendnung der Algorithmen. Sie beschreiben wie weit eine Lösung vom richtigen Ergebnis entfernt ist. Wenn beispielsweise, der höchste Punkt auf einer Ebene gesucht wird, könnte eine Fitnessberechnung wie folgt aussehen: Die Fitness eines Punktes wird berechnet, indem für jeden Punkt in einem Radius der höher ist als der Punkt, die vertikale Distanz von der Fitness abgezogen wird. Derartige Berechnungsforschriften für die Fitness wird auch als Fitnessfunktion. Dieses Beispiel ist vereinfacht und keine optimale Fitnessfunktion, es soll lediglich verdeutlichen wie eine Fitnessfunktion aussehen kann.

Der Ablauf von EA findet in einer Wiederholung, meist als Generationen bezeichnet, statt. Für jede dieser Generationen, werden mehrere evolutionäre Mechanismen verwen-

det. Darunter fallen: Selektion, Rekombination, Mutation und Überleben der Fittesten. Diese Mechanismen werden in der Regel erst nach der Berechnung der Fitness der Phenotypen durchgeführt.

Bevor die erste Generation beginnt, muss die Initialisierung durchgeführt werden. Hierbei werden die Genotypen, je nach Algorithmus, zufällig oder basierend auf einer Vorlage generiert. Das Ziel der Initialisierung ist es meist, eine möglichst breite Menge an möglichen Lösungen zu generieren, um den Suchraum möglichst effizient zu erkunden und die Chancen ein globales Optimum zu finden, zu erhöhen.

Grundsätzlich sind EA jedoch meist sehr ähnlich aufgebaut. Der Ablauf ist wie folgt:

Algorithm 1 Aufbau eines EAs [8]

```
Parents ← Initialpopulation
while Abbruchbedingung = False do
    Fitness ← BerechneFitness(Parents)
    Children ← ∅
    while |Children| < |Parents| do
        Wähle 2 Parents                                  ▷ Selektion
        Kombiniere Sie zu neuen Parents                  ▷ Rekombination
        Children ← Children ∪ {Child1, Child2}
    end while
    Children ← Mutiere(Children)                         ▷ Mutation
    Parents ← Überleben(Parents, Children)               ▷ Überleben der Fittesten
end while                                                ▷ Nächste Generation
```

Evolutionäre Mechanismen

Die Selektion beschreibt die Auswahl von Genotypen für die Rekombination. Sie ist meist der erste Schritt innerhalb einer Generation. Hierbei werden meistens die Genotypen basierend auf ihrer Fitness bevorzugt. Somit besitzen Genotypen mit einer höheren Fitness eine höhere Wahrscheinlichkeit ausgewählt zu werden und ihre Eigenschaften an die nächste Generation weiter zu vererben.

Nachdem die Genotypen über Selektion gewählt wurden, folgt die Rekombination. Hier unterscheidet sich die Vorgehensweise stark je nachdem, wie die Genotypen aufgebaut sind. Sogar die Anzahl der kombinierten Genotypen kann sich unterscheiden, meist zwei Genotypen in der Kombination aber auch drei oder mehr sind möglich. Auch die Anzahl der resultierenden Genotypen kann variieren, in den meisten Fällen entsteht jedoch ein

Genotyp. Das Grundprinzip bleibt jedoch immer das selbe, die einzelne Eigenschaften der Genotypen werden kombiniert oder übernommen um einen neuen Genotypen zu bilden. Diese Kombination oder Übernahme ist meist zufällig gewichtet, wie stark die Genotypen Einfluss auf den entstehenden Genotypen haben. Somit liegt auch hier ein wichtiger Punkt für diese Untersuchung. [16]

Als nächstes kommt die Mutation, hier können basierend auf einer zufälligen Chance zu einem zufälligen Ausmaß, Eigenschaften der Genotypen verändert werden. Wie sie Gene beeinflussen und wie stark diese Mutationen sind, ist meist ein wichtiger Parameter für den Algorithmus. Für einfache numerische Gene kann die Mutation simpel sein, wie zum Beispiel das Addieren oder Subtrahieren eines zufälligen Wertes oder selten auch Werte vollkommen zu erstetzen. Gene die nur aus einer Liste von Werten bestehen, können auch einfach einen Wert aus der Liste zufällig auswählen. Für komplexere Gene, wie Listen, Matrizen oder Objektstrukturen, kann die Mutation jedoch schwieriger sein. Besonders bei Objektstrukturen kann die Mutation sehr komplex sein und mehrere Eigenschaften auf einmal verändern. Häufig gibt es hier einen Elitismus, um zu verhindern, dass die besten Ergebnisse verändert werden. Da die Wahl der betroffenen Gene und das Ausmaß der Mutationen zufällig ist, liegt hier ebenfalls ein besonderer Fokus dieser Untersuchung. [2]

2.1.2 Parameter Optimierung

Eine Komplipikation der EA ist die Wahl der korrekten Parameter, auch Parameter Optimierung genannt. Dies ist häufig eine große Aufgabe, da EAs häufig eine große Anzahl an verschiedenen Parametern besitzen und der Effekt durch Änderungen meist nur schwer einzuschätzen ist. Häufige Parameter sind hierbei, Selektions- und Rekombinationschance, Mutationschance, Mutationsstärke, Grenzen für Ausgabewerte, Interval/Liste der möglichen Werte für Genotypeeigenschaften und weitere.

Ein weiteres Problem kann sein, dass bei geringfügigen Änderungen an den Parametern, die Effekte aufgrund der zufallsbasierten Arbeitsweise nur bei mehrfachem Ausführen bemerkbar ist. Ebenfalls kann das Gegenteil das Problem sein, das eine kleine Änderung zu einem stark veränderten Ergebnis führen kann.

Verfahren für die Parameter Optimierung

Da die Parameter ein derart wichtiger Bestandteil für den Erfolg des Algorithmus sind, gibt es verschiedene Ansätze diese möglichst optimal zu wählen.

Die simplste Methode ist die Parameter manuell anzupassen und die Ergebnisse zu vergleichen. Während diese Methode simpel in der Ausführung ist, birgt sie auch einige Nachteile. Durch die manuelle Prüfung der Parameter wird meist nur ein sehr kleiner Wertebereich für die Parameter geprüft. Dies kann dazu führen das nur ein lokales Optima gefunden wird, statt dem eigentlich gesuchten globalen Optima. Des Weiteren ist diese Methode sehr zeitaufwendig, da die Parameter meist mehrfach getestet werden müssen, um ein aussagekräftiges Ergebnis zu erhalten. Besonders bei EAs, kann durch ihre zufallsbasierte Arbeitsweise, möglicherweise suboptimale Parameter, bei einmaliger Ausführung, besseres aber nicht reproduzierbares Ergebniss erziehlen. So können Parameterkombinationen die Konsistent auf gute Ergebnisse kommen, ersetzt werden mit Kombinationen die starke Schwankungen erzeugen. Außerdem kann für Probleme mit vielen Parametern, die manuelle Anpassung sehr schwierig sein.

Eine simple automatisierte Methode ist die Gridsearch, die den gesamten Suchraum systematisch abtastet, indem sie eine vordefinierte Menge von Parameterkombinationen testet. Hierfür werden für alle Parameter Intervale oder Listen valider Werte angegeben und anschließend automatisiert alle möglichen Kombinationen getestet und die Beste als Ergebnis verwendet. Während diese Methode garantiert, dass das globale Optima innerhalb der gegebenen Parameter gefunden wird und meist auch sehr einfach zu implementieren ist, benötigt es je nach Parameteranzahl und Größe der Intervale und Listen sehr viel Zeit. Des Weiteren besteht das Problem wie die Intervale und Listen festgelegt werden. Zu große Intervale und Listen können die Ausführungzeit stark verlängern und zu kleine Intervale und Listen können potenziell keine zufriedenstellende Kombination enthalten. [3]

Eine weitere simple Methode ist die Randomsearch, die eine zufällige Auswahl von Parameterkombinationen testet. Somit wird nicht der gesamte Suchraum abgetastet, sondern nur eine zufällige Auswahl der möglichen Parameterkombinationen. Dadurch werden mit weniger Evaluierungen potenziell ähnliche Ergebnisse erzielt, insbesondere bei hochdimensionalen Problemen. Damit ist die Randomsearch meist effizienter und besser für große Suchbereiche geeignet als die Gridsearch. Jedoch besteht auch hier das Problem,

dass die Parameterwahl zufällig ist und somit auch suboptimale Parameter gewählt werden können. Im Gegenteil zur Gridsearch, kann Randomsearch auch nicht garantieren, dass das globale Optima gefunden wird. [3]

Natürlich gibt es auch komplexere Methoden, wie die Bayesian Optimization. Diese versucht, mit Hilfe von probabilistischen Modellen, wie Gaussian Processes, die Funktion zu approximieren, um die Leistung eines Parametersatzes bewertet [24]. Basierend auf diesen Modellen wird der Suchraum iterativ effizient erkundet. Auch EAs können hierbei verwendet werden, um die Parameter zu optimieren, wie beispielsweise Differential Evolution oder Particle Swarm Optimization.

2.1.3 No-Free-Lunch Theorem

Neben den Problemen der Parameter Optimierung kommt noch hinzu, dass zuerst der passende Algorithmus gewählt werden muss. Das "No-Free-Lunch"-Theorem (NFL) befasst sich genau hiermit. Es besagt, dass es keinen universellen Algorithmus gibt, der für alle Optimierungsprobleme gleich gut geeignet ist. Dieses Prinzip wurde ursprünglich von David Wolpert und William G. Macready formuliert und hat weitreichende Konsequenzen für die Entwicklung und Anwendung von Algorithmen. Das Theorem beruht auf der Idee, dass, wenn man die Leistung eines Algorithmus über alle möglichen Optimierungsprobleme mittelt, die Ergebnisse für alle Algorithmen gleich gut oder schlecht ausfallen. Mit anderen Worten: Wenn ein Algorithmus bei einer bestimmten Problemklasse besonders gut abschneidet, muss es zwangsläufig andere Problemklassen geben, bei denen seine Leistung schlechter ist. In der Praxis bedeutet dies, dass ein Algorithmus, der für ein spezifisches Problem hervorragend funktioniert, möglicherweise für ein anderes Problem ungeeignet ist. Diese Erkenntnis stellt die Annahme infrage, dass es eine universelle Optimierungsmethode gibt, die immer besser oder gleich gut wie alle anderen ist. [30, 29]

Somit muss bei der Auswahl des Algorithmus für ein Problem erst untersucht werden, um was für eine Art von Problem es sich handelt: ein Such- oder ein Optimierungsproblem. Anschließend muss festgestellt werden, welche Eigenschaften benötigt der Genotype bzw. wie muss dieser aufgebaut sein. Sobald diese Informationen feststehen kann die Wahl des korrekten Algorithmus beginnnen.

2.2 Die Funktionsweise von NEAT

2.2.1 Grundlagen

Der NeuroEvolution of Augmenting Topologies (NEAT) Algorithmus ist eine Anwendung des Evolutionäre Algorithmen Formats auf den Bereich des ML. Im Gegensatz zu den Meisten anderen Ansätzen, die versuchen in einem bestehenden neuronalen Netz die Gewichte der Kanten zu optimieren, ist NEAT dafür ausgelegt neben den Gewichten auch die Struktur des finalen Netzwerkes zu optimieren.

Um diesen Effekt zu ermöglichen, startet NEAT für gewöhnlich nur mit einem initialen neuronalen Netz, dass nur vollvernetzte Eingabe- und Ausgabeneuronen enthält. Dieses Netz ist auch für die gesammte Initialpopulation gleich, somit besitzt NEAT im Gegensatz den meisten anderen EA keine zufällige Initialpopulation. Von dieser Initialpopulation wird dann im Lauf des Algorithmus, über die evolutionären Mechanismen Selektion, Rekombination und Mutation, langsam erweitert.

Der Genotyp des NEAT Algorithmus entspricht einer Liste von Neuronen und einer Liste deren Verbingungen. Die Neuronen bestehen hierbei aus folgenden Attributen:

- ID: Eindeutige Identifikation der Neuronen
- TYPE: Bestimmt ob das Neuron ein Input-, Output- oder Hiddenneuron ist

Wichtiger ist jedoch der Aufbau der Verbindungen. Sie enthalten folgende Attribute:

- Eingabeneuron: Die ID des Neurons, welches den Wert liefert
- Ausgabeneuron: Die ID des Neurons, welches den Wert empfängt
- Gewichtung: Wie stark wird die Eingabe gewertetet
- Status: Ist die Verbindung aktiv oder deaktiviert
- Innovation: Wann wurde diese Verbindung erstellt

Mit diesen Informationen lässt sich dann der Phenotyp, das neuronale Netzwerk, aufbauen. Deaktivierte Verbindungen und Knoten ohne aktive Verbindungen werden hierbei ignoriert.

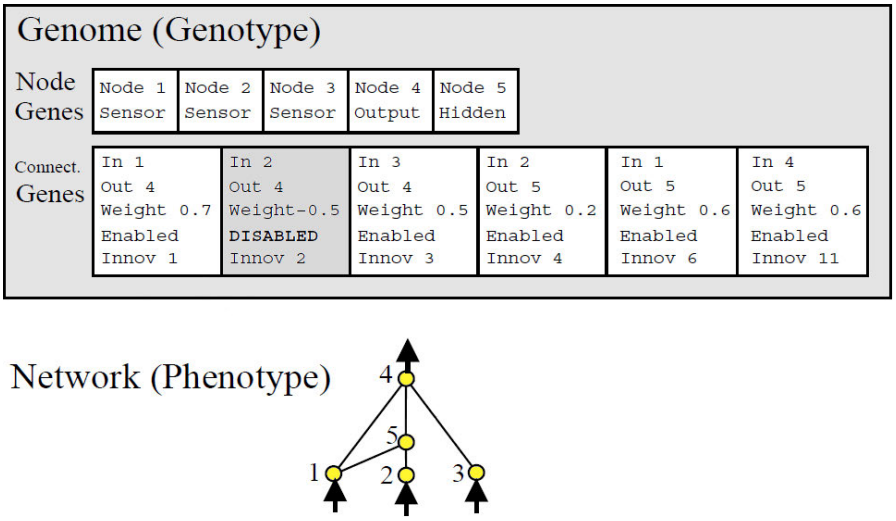

Abbildung 2.1: Genotyp und Phenotyp eines NEAT-Algorithmus [26]

Die Fitnessfunktion nutzt dann diesen Phenotyp um die Testdaten auf ihn anzuwenden. Anschließend werden die Ergebnisse des Neuronalen Netzwerkes mit den erwarteten Ergebnissen verglichen und daraus die Fitness des Netzes berechnet.

2.2.2 Selektion

Die Selektion im NEAT-Algorithmus basiert auf der Fitnessbewertung der Individuen. Individuen mit höherer Fitness haben eine höhere Wahrscheinlichkeit, Eltern für die nächste Generation zu werden. Dabei berücksichtigt NEAT spezifische Mechanismen, die über klassische Selektionsmethoden hinausgehen, um die Evolution sowohl effizient als auch divers zu gestalten.

2.2.3 Rekombination

Die Rekombination ist der erste Schritt bei dem die Innovationsnummer aus der Mutation Verwendung findet. In der Rekombination werden die zwei Genotypen nebeneinander gestellt und aus ihnen ein neuer Genotyp erstellt. Hierfür werden die Verbindungslisten der beiden Genotypen verwendet. Die einzelnen Verbindungen werden, basierend auf ihren gespeicherten Innovationsnummern sotiert und falls identisch zusammengestellt. Nun werden alle Verbindungen durchgegangen und eine von drei Aktionen versucht.

Die erste mögliche Aktion tritt auf, wenn beide Genotypen eine Verbindung der selben Innovationsnummer besitzen. In diesem Fall wird per Zufall entschieden, aus welchen Genotype die Attribute der Verbindung übernommen werden.

Für die zweite mögliche Aktion darf nur einer der beiden Genotypen eine Verbindung mit dieser Innovationsnummer besitzen und der andere Elternteil noch Verbindungen mit höherer Innovationsnummer besitzt. Wenn dies der Fall ist, spricht man auch von einer disjunkten ("disjoint") Verbindung und es wird per Zufall entschieden, ob diese Verbindung in den neuen Genotypen kopiert wird.

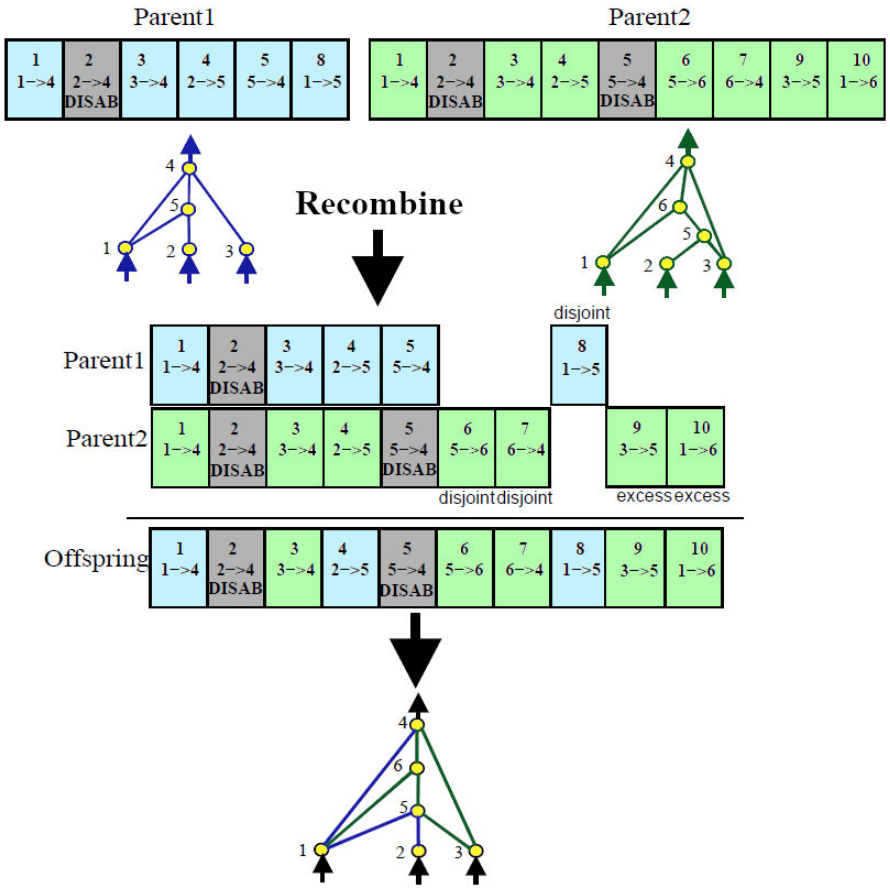

Abbildung 2.2: Rekombination von zwei Genotypen [26]

Zuletzt gibt es noch die Aktion, wenn eine Verbindung mit dieser Innovationsnummer nur in einem Genotyp vorhanden ist und der andere Genotyp keine Verbindungen mit höheren Innovationsnummern besitzt. Verbindungen dieser Art werden auch überschüssige

("excess") Verbindungen genannt und, wie für disjunkte Verbindungen, wird per Zufall entschieden, ob die Verbindung in den neuen Genotyp übernommen wird.

Zuletzt wird einmal über die Liste von Verbindungen des neuen Genotype iteriert und alle benötigten Neuronen, neben den Ein- und Ausgabeneuronen, werden der Neuronenliste hinzugefügt.

2.2.4 Artbildung (Speciation)

Der NEAT-Algorithmus nutzt einen innovativen Mechanismus namens Speciation, um genetische Vielfalt während des Evolutionsprozesses zu bewahren. Dieser Ansatz basiert auf der Idee, Individuen einer Population in verschiedene Spezies zu unterteilen, die jeweils auf strukturellen Gemeinsamkeiten ihrer neuronalen Netzwerke beruhen. Speciation spielt eine zentrale Rolle, da sie sowohl die Erforschung neuer genetischer Möglichkeiten als auch die Bewahrung erfolgversprechender genetischer Linien unterstützt. [26]

Im Kern wird die Einteilung in Spezies durch ein strukturelles Abstandsmaß (δ) gesteuert, das Unterschiede zwischen den Genotypen zweier neuronaler Netzwerke quantifiziert. Dieses Maß berücksichtigt Variationen wie das Hinzufügen neuer Knoten oder Verbindungen sowie die Veränderung bestehender Gewichte. [26]

$$\delta = \frac{c_1 \cdot E}{N} + \frac{c_2 \cdot D}{N} + c_3 \cdot \bar{W} \tag{2.1}$$

Dabei sind E die Anzahl der überschüssigen Verbindungen ("excess"), D die Anzahl der disjunkten Verbindungen ("disjoint"), N die Anzahl der Verbindungen im größeren Netzwerk und $\bar{W}$ die durchschnittliche Gewichtungsdifferenz der gemeinsamen Verbindungen. c_1, c_2 und c_3 sind hierbei Hyperparameter, die die Gewichtung der einzelnen Komponenten des Abstandsmaßes steuern. Zwei Individuen werden derselben Spezies zugeordnet, wenn der berechnete Abstand unterhalb eines vordefinierten Schwellenwerts liegt. Durch diesen Ansatz entstehen Gruppen von Netzwerken mit ähnlicher genetischer Struktur, die voneinander isoliert weiterentwickelt werden können. [26]

Ein zentraler Vorteil dieses Mechanismus ist die Fähigkeit, Innovationen zu fördern. Neue Mutationen, insbesondere strukturelle Änderungen wie das Hinzufügen neuer Knoten oder Verbindungen, können die Leistungsfähigkeit eines Netzwerks kurzfristig verringern, bieten jedoch langfristig das Potenzial für signifikante Verbesserungen. Innerhalb einer Spezies konkurrieren Individuen nur mit anderen Netzwerken ähnlicher Struktur.

Dies verhindert, dass innovative Mutationen von bereits ausgereiften Netzwerken dominiert werden und ermöglicht es, dass sie über mehrere Generationen hinweg evaluiert werden. Diese Funktion wird unterstützt durch die Tatsache, dass die Fitnessberechnung innerhalb einer Spezies normalisiert wird, um die Leistungsfähigkeit von Netzwerken unterschiedlicher Größe und Komplexität vergleichbar zu machen. [26]

$$f'_i = \frac{f_i}{\sum_{j=1}^{N} sh(\delta(i,j))} \tag{2.2}$$

Dabei ist f_i die Fitness des Individuums i und $\delta(i,j)$ die Ähnlichkeit zwischen den Individuen i und j. Wenn $\delta(i,j)$ innerhalb des Schwellenwerts liegt, wird der Wert von $sh(\delta(i,j))$ auf 1 gesetzt, ansonsten auf 0. N ist die Anzahl der Individuen in der Spezies. Basierend auf dieser angepassten Fitnessfunktion wird die Größe der Spezies in der nächsten Generation entschieden. [26]

Trotz seiner Vorteile bringt Speciation auch Herausforderungen mit sich. Der zusätzliche Rechenaufwand, der durch die Berechnung des Abstandsmaßes und die Zuordnung zu Spezies entsteht, kann bei großen Populationen oder hochkomplexen Netzwerken erheblich sein. Zudem ist die Effektivität von Speciation stark von der Wahl des Schwellenwerts für die Speziesbildung abhängig, was eine sorgfältige Anpassung der Hyperparameter erfordert. [26]

2.2.5 Mutation

Für die Mutation gibt es zwei Mechanismen: Änderungen der Verbindungsgewichte oder Änderungen an der Netzstruktur/Topologie.

Mutation der Verbindungsgewichte ist der einfachere Mechanismus. Sie iteriert über die Verbindungsliste und ändert die Gewichtung der Verbindungen mit einer Wahrscheinlichkeit um einen zufälligen Wert. [26]

Änderungen an der Netzstruktur sind komplexer. Es gibt zwei Arten von Netzstrukturmutationen: Hinzufügen einer Verbindung oder Hinzufügen eines Knotens. Sie sind auch ein Beispiel für Mutationen, die mehrere Gene gleichzeitig verändern. [26]

Für das Hinzufügen einer Verbindung muss zunächst geprüft werden, ob die gewählten Neuronen bereits verbunden sind. Wenn dies der Fall ist, muss geprüft werden ob die Verbindung aktiv ist und wenn nicht, wird sie aktiviert. Bei aktiver Verbindung wird das

nächste Neuronenpaar geprüft. Wenn keine Verbindung existiert, dann wird eine neue Verbindung erstellt, die Innovationsnummer wird gespeichert, anschließend um 1 erhöht und die Verbindung zur Verbindungsliste hinzugefügt. [26]

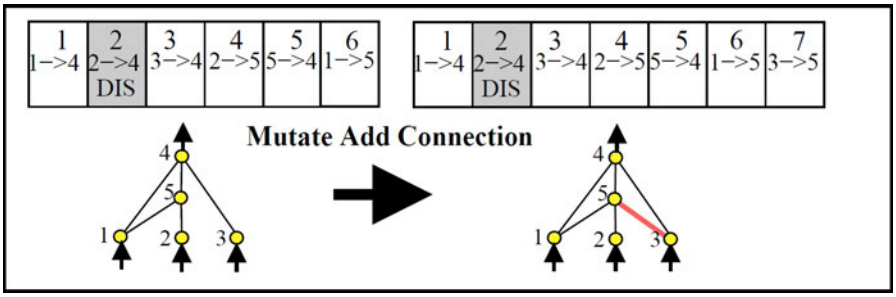

Abbildung 2.3: Mutation: Hinzufügen einer neuen Verbindung [26]

Das Hinzufügen eines Knoten wählt eine bestehende aktive Verbindung und deaktiviert diese. Anschließend wird ein neues Neuron der Neuronenliste hinzugefügt. Für dieses Neuron werden dann jeweils eine Verbindung von der Quelle der ursprünglichen Verbindung und zu dem Ziel der Verbindung erstellt und der Verbindungsliste hinzugefügt. [26]

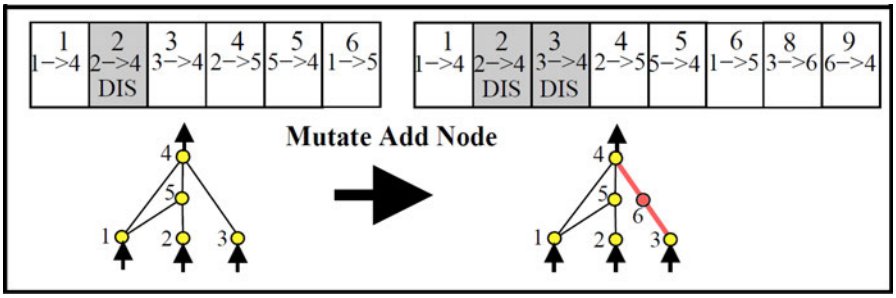

Abbildung 2.4: Mutation: Hinzufügen eines neuen Neuron [26]

Die in dieser Arbeit verwendete Implementierung von NEAT (neat-python) nutzt zusätzlich weiter Mutationen, wie die Änderung der Aktivierungsfunktionen. Hierbei wird die Aktivierungsfunktion des Neurons, mit einer geringen Wahrscheinlichkeit, ausgetauscht mit einer anderen Aktivierungfunktion aus der Liste der möglichen Aktivierungsfunktionen.

2.3 Random Number Generators (RNGs)

Zufallszahlengeneratoren (Random Number Generators, RNGs) sind essenzielle Werkzeuge in zahlreichen wissenschaftlichen und technischen Anwendungen, da sie die Erzeugung von Zahlen ermöglichen, die bestimmte Zufallseigenschaften aufweisen. Diese Zahlen finden in unterschiedlichsten Bereichen Anwendung, darunter Simulationen, Kryptografie, Optimierung, Maschinelles Lernen und statistische Modellierung. Der Begriff "Zufall" kann dabei, je nach Kontext, unterschiedliche Bedeutungen haben, die von der Simulation echter physikalischer Zufälligkeit bis hin zur Erzeugung pseudorandomisierter Sequenzen reichen. [9]

Ein wesentliches Merkmal eines guten RNG ist seine Fähigkeit, Sequenzen zu erzeugen, die statistische Tests auf Zufälligkeit bestehen. Dies umfasst Gleichverteilung, Unabhängigkeit und andere spezifische Eigenschaften, die je nach Anwendung erforderlich sein können. Beispielsweise sollte in der Kryptografie die Unvorhersehbarkeit von zentraler Bedeutung sein, da ein vorhersehbarer RNG die Sicherheit eines kryptografischen Systems gefährden würde. [9]

Grundsätzlich werden RNGs in zwei Kategorien unterteilt: echte Zufallszahlengeneratoren und Pseudozufallszahlengeneratoren (PRNGs). Echte RNGs basieren auf physikalischen Prozessen wie thermischem Rauschen, Quantenphänomenen oder anderen schwer vorhersagbaren natürlichen Quellen. Diese Methoden liefern authentische Zufallszahlen, die von Natur aus unvorhersehbar sind. Die Komplexität und der physikalische Aufwand bei der Implementierung solcher Systeme machen sie jedoch in vielen praktischen Anwendungen unhandlich. [9]

2.3.1 Pseudo-Random Number Generators (PRNGs)

Pseudozufallszahlengeneratoren (PRNGs) hingegen erzeugen Zahlen, die statistisch wie echte Zufallszahlen erscheinen, aber deterministisch durch mathematische Algorithmen generiert werden. PRNGs nutzen einen Startwert, den sogenannten "Seed", und berechnen mit Hilfe rekursiver Algorithmen nachfolgende Zahlen. Da der Seed den gesamten Zahlenstrom eindeutig bestimmt, sind PRNGs inhärent deterministisch. Diese Eigenschaft ist einerseits nützlich, da sie Reproduzierbarkeit ermöglicht, stellt jedoch in der Kryptografie eine Schwachstelle dar, wenn der Seed erraten oder rekonstruiert werden kann. [9]

Ein zentrales Ziel bei der Entwicklung von PRNGs ist es, Sequenzen zu erzeugen, die für praktische Zwecke als zufällig gelten. Die Periodenlänge beschreibt hierbei, nach wie vielen Schritten sich die Sequenz wiederholt. Moderne PRNGs wie der Mersenne Twister haben extrem lange Perioden, die sicherstellen, dass Wiederholungen in praktisch relevanten Anwendungen ausgeschlossen sind. [9]

2.3.2 Mersenne Twister

Der Mersenne Twister (MT) ist ein weit verbreiteter PRNG, der 1997 von Makoto Matsumoto und Takuji Nishimura entwickelt wurde. Der Algorithmus wurde entwickelt, um die Herausforderungen früherer PRNGs wie kurze Perioden, schlechte Verteilungseigenschaften und Korrelationen in der erzeugten Sequenz zu adressieren. Der Algorithmus zeichnet sich durch eine sehr lange Periode von $2^{19937} - 1$ aus, was auch namensgebend für den Twister ist, da es sich hierbei um eine sogenannte Mersenne-Primzahl handelt. Mersenne-Primzahlen sind jene Primzahlen die der Struktur $2^n - 1$, wobei n eine natürliche Primzahl ist, folgen [21]. [15]

Viele Programmiersprachen, darunter Python, MATLAB und R, und Bibliotheken implementieren den MT als Standard-RNG. Trotz seiner zahlreichen Stärken ist der MT nicht ohne Einschränkungen. Eine Schwäche liegt in der Tatsache, dass der Algorithmus deterministisch ist und daher nicht für kryptografische Anwendungen geeignet ist, bei denen echte Unvorhersehbarkeit erforderlich ist. [15]

MT verwendet ein Zustandsarray, das eine Reihe von 32-Bit-Werten speichert und als Grundlage für die Erzeugung der Zufallszahlen dient. Das Array hat eine feste Größe, die für die Standardimplementierung MT19937 auf 624 Werte festgelegt ist. Die große Größe des Zustandsarrays ermöglicht eine Periodenlänge von $2^{19937} - 1$. Der Algorithmus verwendet diese Werte später, um eine neue Zufallszahl zu generieren. [15]

Bevor der MT Zufallszahlen generieren kann, muss das Zustandsarray initialisiert werden. Hierfür wird ein Seed-Wert benötigt, der die Startwerte des Zustandsarrays bestimmt. Der Seed-Wert wird in der Regel aus der Systemzeit oder anderen Quellen erzeugt, die als zufällig gelten. Die Initialisierung des Zustandsarrays erfolgt durch eine iterative Transformation, die den Seed-Wert in die 624 Werte des Arrays umwandelt. Der Seed-Wert wird hierfür als erstes Element des Zustandsarrays eingetragen. Anschließend werden die

restlichen Einträge basierend auf folgender Formel:[15]

$$MT[i] = (f \cdot (MT[i-1] \oplus (MT[i-1] \gg (w-2))) + i) \ \& \ \text{0xFFFFFFFF} \tag{2.3}$$

berechnet, wobei f eine Konstante, w die Bitbreite der Werte und i der Index des aktuellen Werts im Zustandsarray ist. Für MT19937 ist f als 1812433253 festgelegt und w als 32. [15]

Die Aktualisierung des Zustandsarrays, auch `twist` genannt, erfolgt durch eine iterative Transformation. Dabei wird der aktuelle Werte aus dem Zustandsarray mit dem darauf folgenden kombiniert, wobei eine Mischung aus XOR und modulo Operationen angewandt wird. Ein wichtiger Bestandteil dieser Transformation ist die Verwendung von Masken, die die oberen und unteren Bits der Werte trennen. Im Fall von 32-Bit-Werten wird das 32. Bit des aktuellen Werts mit den restlichen 31 Bits des folgenden Werts kombiniert, um einen neuen Wert zu erzeugen. Für das letzte Element des Zustandsarrays wird das Erste als nachfolgender Wert verwendet. Diese maskierten Werte werden dann addiert und um ein Bit nach rechts verschoben, um den neuen Wert zu erhalten. Anschließend wird geprüft, ob der Wert ungerade ist und gegebenenfalls mit einer Konstanten (a) via XOR verknüpft. Final wird der Wert mit einem weiteren Wert aus dem Zustandsarray verknüpft, der um eine feste Anzahl an Schritten (m) verschoben ist. [15]

Algorithm 2 Aktualisierung des Zustandsarrays (`twist`)[15]

Require: $a \leftarrow$ 0x9908B0DF, $m \leftarrow 397$
  for $i \leftarrow 0$ to 624 **do**
    $x \leftarrow (MT[i] \ \& \ \text{0x80000000}) + (MT[(i+1) \mod 624] \ \& \ \text{0x7FFFFFFF})$
    $xA \leftarrow x \gg 1$
    if $x \mod 2 \neq 0$ **then** ▷ Prüfen ob x ungerade ist
      $MT[i] \leftarrow MT[i] \oplus a$
    end if
    $MT[i] \leftarrow MT[(i+m) \mod 624] \oplus xA$
  end for

Um eine Zufallszahl zu generieren, wird ein weiterer Algorithmus verwendet, der auf dem Zustandsarray basiert, die Temperierung. Hierfür werden weitere Konstanten und Bitmasken verwendet, um die Werte zu transformieren. Zunächst wird eine Kopie des aktuelle Werts aus dem Zustandsarray um einem konstanten Wert u bitweise nach rechts verschoben. Anschließend wird der neue Wert mit der Konstante d maskiert und mit dem ursprünglichen Wert XOR verknüpft. Der resultierende Wert wird mit s bitweise

nach rechts verschoben, mit der Konstanten b maskiert und wieder mit sich selbst XOR verknüpft. Dieser Ablauf wird mit dem Paar (t, c), wobei t die Verschiebung nach links und c die Bitmaske darstellt, wiederholt. Der erzeugte Wert wird ein letztes Mal mit einer Konstanten l nach rechts verschoben und mit sich selbst XOR verknüpft. Der resultierende Wert wird auf die Wortgröße, meist 32-Bit, beschränkt und als Zufallszahl zurückgegeben. [15]

Algorithm 3 Erzeugung einer Zufallszahl (`extract_number`)[15]

Require: $index, u \leftarrow 11, d \leftarrow$ 0xFFFFFFFF, $s \leftarrow 7, b \leftarrow$ 0x9D2C5680,
Require: $t \leftarrow 15, c \leftarrow$ 0xEFC60000, $l \leftarrow 18$
  $y \leftarrow MT[index]$
  $index \leftarrow (index + 1) \mod 624$
  $y \leftarrow y \oplus ((y \gg u)\ \&\ d)$
  $y \leftarrow y \oplus ((y \ll s)\ \&\ b)$
  $y \leftarrow y \oplus ((y \ll t)\ \&\ c)$
  $y \leftarrow y \oplus (y \gg l)$
  return y & 0xFFFFFFFF

2.3.3 Complementary Multiply With Carry (CMWC)

Der Complemetary-Multiply-With-Carry (CMWC) ist ein PRNG, der auf der Klasse der Multiply-With-Carry-Generatoren basiert, welche von George Marsaglia entwickelt wurden. Der CMWC-Ansatz erweitert das Konzept der MWC-Generatoren, indem er das Prinzip der Multiplikation mit einem zusätzlichen Komplementärmechanismus kombiniert, um die Qualität der Zufallszahlen zu verbessern. Dieser Generator zeichnet sich durch seine Effizienz und seine Fähigkeit aus, große Periodenlängen zu erreichen, während er gleichzeitig mit einfachen mathematischen Operationen arbeitet. [13]

Der CMWC-Algorithmus verwendet eine Reihe von Parametern, die die Qualität der erzeugten Zufallszahlen beeinflussen. Dazu gehören die Multiplikationskonstante a und die Größe des Zustandsarrays n. Die Wahl dieser Parameter ist entscheidend für die Leistung des Algorithmus und sollte sorgfältig getroffen werden. Marsaglia empfiehlt die Verwendung von $a = 18782$ und $n = 4096$ als gute Werte für den CMWC-Algorithmus. [13]

Der CMWC-Algorithmus, wie in Algorithmus 4 dargestellt, erreicht eine Periode von $2^{131086} \approx 10^{39424}$ und zeichnet sich durch seine Einfachheit und Effizienz aus.

Algorithm 4 Complemetary-Multiply-With-Carry Algorithmus[13]

Require: $n \leftarrow 4096, Q \leftarrow [n], a \leftarrow 18782, m \leftarrow 0\text{xFFFFFFFE}, i, c$
  $i = (i + 1) \mod n$
  $t = a \cdot Q[i] + c$
  $c = t \gg 32$
  $x = t + c$
  if $x < c$ **then** ▷ Überlauf prüfen
    $x = x + 1$
    $c = c + 1$
  end if
  $Q[i] = m - x$
  return $Q[i]$

2.3.4 Xorshift

Der Xorshift-Algorithmus, der von George Marsaglia im Jahr 2003 vorgestellt wurde, basiert auf einfachen bitweisen Operationen, insbesondere XOR-Verschiebungen, und gehört zur Klasse der linearen PRNGs. Er wurde entwickelt, um eine schnelle und effiziente Methode zur Generierung von Zufallszahlen zu bieten, die dennoch eine akzeptable statistische Qualität aufweist. Mit seiner Periode von $2^{32} - 1$ ist er der kürzeste PRNG in dieser Untersuchung und soll einen guten Vergleichwert bieten. [14]

Die Funktionsweise des Xorshift-Algorithmus ist einfach und effizient. Der Algorithmus verwendet drei Konstanten, die als Parameter für die bitweisen Operationen dienen. Die Konstanten a, b und c werden in jedem Schritt verwendet, um den aktuellen Zustand zu transformieren und die nächste Zufallszahl zu generieren. Der Algorithmus verwendet eine interne Zustandsvariable, die als Seed-Wert initialisiert wird und in jedem Schritt aktualisiert wird. Die Generierung einer Zufallszahl erfolgt durch eine Kombination aus XOR- und Bitverschiebungsoperationen, die den aktuellen Zustand transformieren und die nächste Zufallszahl erzeugen. [14]

Algorithm 5 Erzeugung einer Zufallszahl (`xorshift`)[14]

Require: $a \leftarrow 13, b \leftarrow 17, c \leftarrow 5$
  $y \leftarrow x$
  $y \leftarrow y \oplus (y \ll a)$
  $y \leftarrow y \oplus (y \gg b)$
  $y \leftarrow y \oplus (y \ll c)$
  return y

Die Konstanten a, b und c bestimmen die Anzahl der Bits, die in jedem Schritt verschoben werden. Die Wahl dieser Konstanten beeinflusst die Qualität der erzeugten Zufallszahlen und kann die Periodenlänge und statistische Eigenschaften des Algorithmus beeinflussen. Die Wahl der Konstanten ist daher entscheidend für die Leistung des Algorithmus und sollte sorgfältig getroffen werden. Die Werte $a = 13$, $b = 17$ und $c = 5$ wurden von Marsaglia als gute Werte für den Xorshift-Algorithmus vorgeschlagen und haben sich in der Praxis bewährt. [14]

2.4 Verwandte Arbeiten

Bereits eine Vielzahl an Untersuchungen zu Evolutionäre Algorithmens haben gezeigt, dass sie selbst bei Verwendung von schlechten Pseudo-Random Number Generator (PRNGs) in der Lage sind, gute Lösungen zu finden. Allerdings sind die Ergebnisse oft deutlich inkonsistenter im Vergleich zu den Ergebnissen, die mit hochwertigen PRNGs erzielt werden. Interessanterweise konnte festgestellt werden, dass in einigen Ausnahmefällen die Verwendung schlechter PRNGs sogar zu besseren Ergebnissen führen kann. [1, 5, 10]

Pavel Krömer et al. zeigten in ihrer Untersuchung zusätzlich die Effekte, durch Verwendung "deterministic chaos"-basierten Generatoren. Hierbei ergab sich, dass zumindest für Differential Evolution (DE), die Auswahl des Generators keinen Einfluss Ergebnisse aufweist. [10]

Erick Cantú-Paz gibt in seiner Untersuchung des genetischen Algorithmus des Weiteren noch an, dass die Wahl des PRNG besonders in der Initialisierung der Population entscheidend ist, während sie in der späteren Verwendung das Ergebnis nicht beeinflussen [5]. Dies wird einen interessanten Vergleich für NEAT bilden, da dieser Algorithmus keine randomisierte Initialpopulation besitzt [26].

Lekshmi Rajashekharan et al. führten Tests mit 14 verschiedenen Different DE Varianten, mit 6 PRNGs auf 19 10-dimensionale Mess-Funktionen durch. Jedoch selbst mit dieser großen Auswahl an Kriterien wurden nur vereinzelt bessere Ergebnisse erzielt, was in Anbetracht der großen Sample-Menge keinen signifikanten Anteil ausmacht. [20]

Die Untersuchung von Miguel Cárdenas-Montes et al. auf die Effekte von RNG auf die Algorithmen Particle Swarm, DE, Genetic Algorithm und Firefly Algorithm wiederum hat gezeigt, dass die Wahl des RNG doch einen Einfluss auf die Leistung haben kann. [6]

Für die Untersuchungen werden meist eine oder mehrere Versionen von MT und PRNGs mit deutlich stärker begrenzten Periodenlängen verglichen [1, 5, 10]. MT besitzt mehrere Gründe für seine bevorzugte Nutzung; er hat eine riesige Periodenlänge von $2^{19937} - 1$ [15], war für lange Zeit als einer der besten PRNG bekannt und ist der Standard RNG für mehrere Programmiersprachen, unter anderem für Python [19]. Für PRNGs mit begrenzter Periodenlänge kommen Xorshift, mit einer Periodenlänge von $2^{32} - 1$ [14], oder auch Periodenbegrenzte Versionen von MT in Frage. In dieser Untersuchung wurde der Xorshift gewählt.

3 Methodik

3.1 Aufbau

Dieses Kapitel wird sich mit der Durchführung des Versuchs beschäftigen. Hierbei wird zuerst auf die verwendeten Problemstellungen eingegangen, anschließend wird die Implementierung des Versuchs und die Konfiguration der Experimente erläutert. Zum Schluss wird die Durchführung der Experimente beschrieben.

3.2 Problemstellungen

3.2.1 k-XOR Simulation

In dieser Arbeit werden drei Problemstellungen untersucht, um den Einfluss der RNGs auf den NEAT-Algorithmus zu prüfen. Die erste Problemstellung ist die Simulation eines exklusiven Oder-Gates (XOR). Ein XOR-Gate ist ein einfaches logisches Gatter, welches zwei Eingänge hat und nur dann ein Signal ausgibt, wenn die Eingänge unterschiedlich sind. Dies ist ein weit bekanntes, nicht-linear lösbares Problem und wird deshalb oft als erster Test für neue Ansätze verwendet [17, 11].

Da es sich hierbei aber um ein sehr simples Problem handelt, wird die etwas komplexere Version des k-XOR-Problems verwendet. Hierbei wird ein XOR-Gate mit k Eingängen simuliert, wobei k eine natürliche Zahl größer zwei ist. Es existieren zwei Definitionen für das Ergebnis des k-XOR-Problems, die Variante, welche genau einen positiven Eingang erwartet, häufig "one-hot detector" genannt, und die Variante, die eine ungerade Anzahl an positiven Eingängen erwartet. Für die "one-hot detector" Variante wird die Ausgabe 1, wenn genau ein Eingang 1 ist, und 0, wenn alle Eingänge 0 sind. Daraus ergibt sich, dass es aus den 2^k mögliche Eingabekombinationen, k positive und $2^k - k$ negative Ergebnisse gibt. Die andere Variante, die ungerade Anzahl an positiven Eingängen erwartet, gibt 1

aus, wenn die Anzahl der positiven Eingaben ungerade ist, und 0, wenn sie gerade ist. Dies ergibt 2^{k-1} positive und 2^{k-1} negative Ergebnisse.

Ein Beispiel für das 3-XOR-Problem ist in Tabelle 3.1 dargestellt.

Eingang 1	Eingang 2	Eingang 3	"one-hot detector"	Ungerade Eingänge
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	1

Tabelle 3.1: Beispiel für das 3-XOR-Problem

Im Rahmen dieser Arbeit wird das 6-XOR-Problem verwendet.

3.2.2 Klassifikation

Die zweite Problemstellung ist die Klassifikation von Daten in einem Spiralplot. Hierbei wird ein Datenset verwendet, welches aus Datenpunkten besteht, die jeweils zwei Eingaben und eine Ausgabe haben. Die Ausgabe ist entweder 0 oder 1, was der Gruppe des Datenpunkts entspricht. Die Datenpunkte sind in einem Spiralplot angeordnet, wobei die Datenpunkte der Gruppe 0 und 1 sich spiralförmig von der Mitte entfernen.

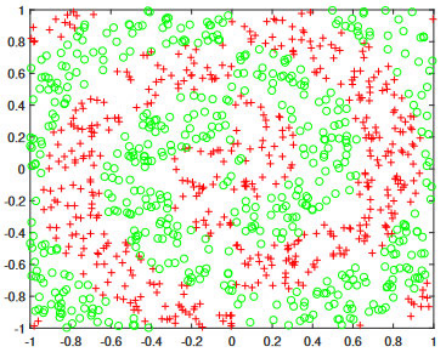

Abbildung 3.1: Ein Beispiel für Spiral-Plot-Daten [17]

3.2.3 Pole-Balancing

Die dritte Problemstellung ist das Pole-Balancing-Problem, auch bekannt als Cart-Pole-Problem. Es ist ein klassisches Steuerungsproblem in der Robotik und Regelungstechnik, das häufig als Benchmark für Algorithmen in den Bereichen ML und Verstärkungslernen verwendet wird. Es besteht darin, eine oder mehrere Stäbe, die auf einem Wagen montiert sind, durch geeignete Bewegungen des Wagens in einer aufrechten Position zu halten. Das Ziel ist es, die Stäbe stabil zu balancieren, indem der Wagen auf einer horizontalen Schiene bewegt wird. [4]

Für das Pole-Balancing-Problem werden eine Reihe von Parametern definiert, die die Dynamik des Systems beeinflussen. Dazu gehören die Länge l und Masse m_p des Stabs, die Masse des Wagens m_c, die Gravitationskonstante g, die Breitenbegrenzung der Strecke h und der maximale Winkel r, den der Stab relativ zur vertikalen Achse einnehmen kann. Bei Verlassen der Intervale h oder r gilt der Versuch als Fehlgeschlagen. Neben diesen Parametern gibt es auch eine Reihe von Variablen, die den Zustand des Systems beschreiben, wie die Position x, Geschwindigkeit $\dot{x}$ und Beschleunigung $\ddot{x}$ des Wagens, der Winkel θ, die Winkelgeschwindigkeit $\dot{\theta}$ und die Winkelbeschleunigung $\ddot{\theta}$ des Stabs und der Impulse der auf den Wagen wirkt F. Das Ziel des Algorithmus ist es, eine geeignete Steuerungsstrategie über den Impulse zu entwickeln, die es ermöglicht, den Stab in einer aufrechten Position zu halten, während der Wagen entlang der Schiene bewegt wird. Ein Beispiel für das Pole-Balancing-Problem ist in Abbildung 3.2 dargestellt. [4]

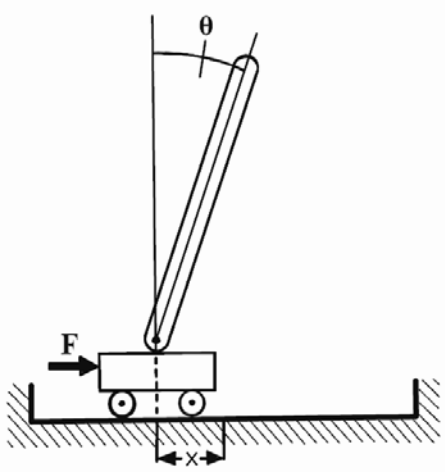

Abbildung 3.2: Ein Beispiel für das Pole-Balancing-Problem [28]

Die Two-Pole-Variante des Problems erweitert das klassische Szenario durch die Einführung eines zweiten Stabs. Dieser zusätzliche Stab unterscheidet sich typischerweise vom ersten in Eigenschaften wie Länge, Masse oder Reibungskoeffizient. Die Dynamik des Systems wird dadurch erheblich komplexer, da die Stabilität eines Stabs direkt von der Bewegung des anderen beeinflusst wird. Dies führt zu einer Kopplung der Bewegungen, bei der der Algorithmus simultan beide Stäbe balancieren muss, während er den Wagen innerhalb eines vorgegebenen Bereichs hält. [4]

Die Two-Pole-Variante wird häufig verwendet, um die Leistungsfähigkeit von Algorithmen in hochdimensionalen und nichtlinearen Problembereichen zu testen. Insbesondere im Bereich des Verstärkungslernens dient es als anspruchsvolle Herausforderung, die Algorithmen zwingt, robuste Strategien zu entwickeln, die sowohl präzise Steuerungen als auch eine effiziente Erkundung des Zustandsraums ermöglichen. Die Kombination aus Determinismus und Chaos in der Dynamik des Systems macht dieses Problem zu einem idealen Testfall für Algorithmen, die in realen Anwendungen wie der Robotik, der autonomen Fahrzeugsteuerung oder der Regelung instabiler Systeme eingesetzt werden sollen.

3.3 Implementierung

Die Implementierung der Experimente erfolgt in Python, unter Verwendung der NEAT-Bibliothek (neat-python). Python wurde für die Implementierung gewählt, da es eine hohe Flexibilität und schnelle Entwicklung ermöglicht. Die NEAT-Bibliothek wurde gewählt, da sie eine der bekanntesten Implementierungen des NEAT-Algorithmus ist und eine hohe Anpassbarkeit bietet. Auch für die Datenverarbeitung wird Python verwendet, da es eine Vielzahl von Bibliotheken für die Datenverarbeitung bietet, wie z.B. numpy und pandas. Die Visualisierung wurde ebenfalls mit Python gelöst, da es mit Matplotlib eine sehr mächtige Visualisierungsbibliothek bietet.

3.3.1 Datensammlung

Der erste Schritt der Implementierung war es, eine Möglichkeit die Daten aus den Versuchen effektiv sammeln und speichern zu können. Glücklicherweise bietet die NEAT-Bibliothek eine Möglichkeit, während des Trainings über sogenannte `reporters` Daten auf Events im Training zu reagieren und dessen Daten abzufragen.

Somit wurde ein Reporter implementiert, der basierend auf seinen Parametern die Daten in einer CSV-Datei speichert. Die Parameter des Reporters sind die Anzahl der Generationen, die zwischen den Speicherungen liegen (stepSize), der Pfad zur Datei, in der die Daten gespeichert werden sollen (path), und die Namen der Spalten die gespeichert werden sollen (columns).

Während des Trainings werden zwei dieser Reporter verwendet, einer der die Daten jeder Generation eines einzelnen Versuchs speichert und einer der nur die beste Generation aus jedem Versuch speichert. Die Daten, die gespeichert werden, sind die Generationsnummer, die Fitness des besten Individuums und die benötigte Zeit in Sekunden. Diese Daten werden dann in einer CSV-Datei gespeichert, die dann für die Analyse verwendet wird.

Die Generationsnummer wird als Index der Daten verwendet, da sie es erlaubt, die Daten chronologisch einzuordnen und somit den Vergleich mit anderen Versuchen ermöglicht. Neben diesem Effekt erlaubt es auch, die Daten in einem Diagramm darzustellen und besonders für die Daten des zweiten Reporters ermöglicht es, einen Vergleich über die benötigten Generation um der Fitnessschwellwert zu erreichen.

Der Fitnesswert des besten Individuums wird als Maß für die Qualität des Versuches verwendet. Hierbei wird nur der Fitnesswert des besten Individuums verwendet, da dieser für Nutzer des Algorithmus der Relevanteste ist. Des Weiteren erlaubt der Fitnesswert, im Fall, dass der Fitnessschwellwert nicht vor der maximalen Generation erreicht wird, die Versuche miteinander zu vergleichen.

Die benötigte Zeit wird als zweites Maß für die Geschwindigkeit des Versuches verwendet. Die Generationsnummer gibt zwar eine grobe Einschätzung über die Geschwindigkeit des Versuches, nimmt jedoch keine Rücksicht auf die extra Zeit, die komplexere RNGs benötigen. Dieser Wert ist dennoch ein sehr subjektives Maß, da er von der Hardware, der Software und der aktuellen Auslastung des Systems abhängt. Er soll somit nur ein grober Richtwert sein.

3.3.2 Anpassung der NEAT-Bibliothek

Nach dem die Datensammlung implementiert wurde, musste die NEAT-Bibliothek angepasst werden, um die Verwendung verschiedener PRNGs zu ermöglichen. Hierfür wurde vorerst geprüft an welchen Stellen der Code auf den PRNG zugreift und in welcher Form

dies geschieht. Es wurde festgestellt, dass die PRNG-Funktionen in der (neat-python) Bibliothek innerhalb von vier Dateien verwendung findet.

Innerhalb dieser Dateien werden insgesamt Sechs Funktionen des PRNGs verwendet. Beginnend hierbei mit der `random` Funktion, die eine zufällige Gleitkommazahl in dem Interval $[0.0, 1.0)$, bzw. $0.0 <= N < 1.0$, zurückgibt. Auf dieser Funktion aufbauend wird die `uniform` Funktion implementiert, die, basierend auf den Parametern `a` und `b`, eine zufällige Gleitkommazahl im Interval $[a, b]$, bzw. $a <= N <= b$, zurückgibt. Die `randint` Funktion ist das Ganzzahl Äquivalent zu der `uniform` Funktion. Für die `gauss` Funktion wird eine zufällige Gleitkommazahl aus einer Normalverteilung mit dem Mittelwert `mu` und der Standardabweichung `sigma` zurückgegeben. Diese Abbildung aus der `random` Funktion wird durch die Box-Muller Transformation erreicht [23]. Zuletzt wurden noch die Listenfunktionen `shuffle` und `choice` verwendet, die die Elemente einer Liste zufällig umsortieren und ein zufälliges Element aus der Liste zurückgeben.

Box-Muller Transformation

Die Box-Muller-Transformation ist hierbei ein mathematisches Verfahren, das zwei unabhängige, gleichverteilte Zufallszahlen im Intervall $[0.0, 1.0)$ in zwei unabhängige, normalverteilte Zufallszahlen mit Mittelwert (μ) 0 und Standardabweichung (σ) 1 transformiert. Sie kann so erweitert werden, dass die erzeugten Zufallszahlen einer Normalverteilung mit beliebigem Mittelwert μ und Standardabweichung σ folgen. Ausgangspunkt der Methode sind zwei gleichverteilte Zufallszahlen U_1 und U_2. Zunächst werden diese in Polarkoordinaten umgerechnet. [23] Der Radius R wird als

$$R = \sqrt{-2 \cdot \ln(U_1)} \tag{3.1}$$

und der Winkel Θ als

$$\Theta = 2 \cdot \pi \cdot U_2 \tag{3.2}$$

berechnet. Mit diesen Größen lassen sich zwei standardnormalverteilte Zufallszahlen Z_0 und Z_1 erzeugen, indem die trigonometrischen Beziehungen [23]

$$Z_0 = R \cdot \cos(\Theta) \tag{3.3}$$

$$Z_1 = R \cdot \sin(\Theta) \tag{3.4}$$

verwendet werden. Die Zufallsvariablen Z_0 und Z_1 folgen der Standardnormalverteilung mit Mittelwert 0 und Standardabweichung 1. Um diese Zufallszahlen auf eine Normalverteilung mit einem beliebigen Mittelwert μ und Standardabweichung σ abzubilden, werden sie skaliert und verschoben. Die transformierten Zufallszahlen X_0 und X_1 ergeben sich durch die Beziehungen [23]

$$X_0 = \mu + \sigma \cdot Z_0 \tag{3.5}$$

$$X_1 = \mu + \sigma \cdot Z_1 \tag{3.6}$$

Die so erzeugten Zufallszahlen X_0 und X_1 folgen der gewünschten Normalverteilung $N(\mu, \sigma^2)$. [23]

Algorithm 6 Box-Muller-Transformation [23]

Require: Z_0, Z_1, μ, σ
  if $Z_1 = None$ **then**
    $U_1 \leftarrow random()$
    $U_2 \leftarrow random()$
    $R \leftarrow \sqrt{-2 \cdot \ln U_1}$
    $\Theta \leftarrow 2 \cdot \pi \cdot U_2$
    $Z_0 \leftarrow R \cdot \cos \Theta$
    $Z_1 \leftarrow R \cdot \sin \Theta$
    return $\mu + \sigma \cdot Z_0$
  else
    $x \leftarrow Z_1$ ▷ Zwischenspeichern
    $Z_1 \leftarrow None$
    return $\mu + \sigma \cdot x$
  end if

Anpassung der betroffenen Dateien

Nachdem die Funktionen identifiziert wurden, wurde eine neue Klasse `RNG` implementiert, die diese Funktionen mit einem gegebenen PRNG umsetzt. Hierfür wird eine Funktion `set_rng` implementiert, die eine statische Variable `rng` auf den gegebenen PRNG setzt. Diese Variable wird dann in den Funktionen, die den PRNG verwenden, anstatt der `random` Funktion verwendet. Um für Kompabilität zu sorgen, wurden alle getesteten PRNGs angepasst, um eine Ganzzahl zwischen 0 und dem Integer-Maximum zurückzugeben.

Um die PRNG-Funktionen auf Korrektheit zu prüfen, wurden für jeden PRNG die Funktionen getestet, indem jede Funktion 1000-mal aufgerufen wurde und die Ergebnisse in

einem Histogramm dargestellt wurden. Die Ergebnisse wurden dann mit den erwarteten Ergebnissen verglichen und auf Korrektheit geprüft.

Mit diesen Funktionen implementiert und auf Korrektheit geprüft, wurden alle Zugriffe auf den PRNG in den betroffenen Dateien mit aufrufen auf die neue `RNG`-Klasse ersetzt. Die hierbei betroffenen Dateien sind `neat/attributes.py`, `neat/genes.py`, `neat/genome.py` und `neat/reproduction.py`. Diese Auswahl an Dateien stimmt mit den vorherigen Beobachtungen überein, an welchen Stellen der NEAT-Algorithmus PRNGs verwendet.

Die `neat/attributes.py`-Datei bietet Klassen für die Attribute der Gene an und damit gleichzeitig auch die Initialisierung und Mutation dieser Attribute. Wie zuvor beobachtet, sind die Initialisierung und Mutation der Attribute die Bereiche, in denen der PRNG zum Einsatz kommt und daher angepasst werden mussten..

Die `neat/genes.py`-Datei bietet die Klassen für die Gene, von Knoten und Verbindungen, an und damit auch die `crossover`-Funktion für die Repopulation. Erneut stimmt dies mit den Beobachtungen überein, dass die Repopulation eine der Stellen ist, an denen der PRNG verwendet wird.

Die `neat/genome.py`-Datei bietet die Klasse für die Genome an und damit auch die Mutationsfunktionen für die Genome. Auch hier stimmt dies mit den Beobachtungen überein, dass die Mutationen eine der Stellen sind, an denen der PRNG verwendet wird.

Zuletzt wurde die `neat/reproduction.py`-Datei angepasst, die die Funktionen für die Repopulation und Selektion der Genome anbietet. Hierbei wird der PRNG jedoch nur in der Selektion benötigt, da die Reproduktion bereits in den Genen implementiert ist. Dennoch stimmt dies mit den Beobachtungen überein, dass die Selektion eine der Stellen ist, an denen der PRNG verwendet wird.

3.3.3 Implementierung der Problemstellungen

k-XOR

Nachdem die NEAT-Bibliothek angepasst wurde, wurden die Problemstellungen implementiert. Hierbei wurde für jede Problemstellung eine eigene Datei angelegt, die sowohl die Eingabe- und Ausgabedaten als auch die Fitnessfunktionen enthält.

Für das k-XOR-Problem wurde ursprünglich die "one-hot detector" Variante implementiert. Die 1024 Eingabekombinationen wurden mit Hilfe der `itertools` Bibliothek und dessen `product`-Funktion generiert. Die `product`-Funktion generiert für eine gegebene Liste von Eingaben alle möglichen Kombinationen dieser Eingaben. In diesem Fall wurde die Liste [0, 1] verwendet, um die Eingaben zu erzeugen. Mit dem `repeat`-Parameter wird festgelegt, wie viele Elemente die einzelnen Permutationen enthalten. Für das k-XOR-Problem mit einem k von 6 wurde somit eine Liste von 64 Permutationen mit je 6 Elementen erstellt. Basierend auf dieser Liste von Permutationen wurde dann mit einer einfachen Schleife eine zugehörige Liste von Ausgaben generiert.

Algorithm 7 k-XOR-Daten Erzeugung

```
Require: k ≥ 1
  bit_permutations ← list(product([0, 1], product = k))      ▷ Eingabedaten
  xor_results ← ∅                                            ▷ Ausgabedaten
  for permutation in bit_permutations do
      isTrue ← false
      for bit in permutation do
          if bit = 1 and not isTrue then
              isTrue ← not isTrue
          else if bit = 1 then
              isTrue ← true
              break
          end if
      end for
      xor_results.append(isTrue)
  end for
```

Als Lossfunktion wurde die Binary Cross Entropy (BCE) verwendet. BCE ist eine weit verbreitete Lossfunktion, die speziell für binäre Klassifikationsaufgaben entwickelt wurde. Sie misst die Differenz zwischen den vorhergesagten Wahrscheinlichkeiten eines Modells und den tatsächlichen Zielwerten und dient als zentraler Bestandteil zur Optimierung von Modellen, die auf Wahrscheinlichkeitsausgaben basieren. BCE wird häufig in neuronalen Netzwerken verwendet, insbesondere in Szenarien, bei denen der Ausgang des Modells durch eine Sigmoid-Aktivierungsfunktion beschränkt wird, um Werte im Bereich zwischen 0 und 1 zu erzeugen. [22]

$$BCE(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \tag{3.7}$$

y entspricht hierbei der Liste der erwarteten Ergebnissen und $\hat{y}$ den Ergebnissen die NEAT erzeugt. y_i und $\hat{y}_i$ entsprechen dabei dem Ergebniss für die i-te Permutation der insgesamt N Permutationen.

Eine binäre Klassifikationsaufgabe besteht darin, Eingabedaten in eine von zwei möglichen Kategorien zu klassifizieren. Beispiele hierfür sind die Entscheidung, ob eine E-Mail als Spam oder nicht zu klassifizieren ist, oder die Vorhersage, ob ein Kunde ein Produkt kaufen wird oder nicht. Die Zielvariablen in binären Klassifikationsaufgaben sind dichotom, das heißt, sie nehmen nur zwei Werte an, die üblicherweise durch 0 und 1 dargestellt werden. Die Aufgabe des Modells ist es, für jede Eingabe eine Wahrscheinlichkeit vorherzusagen, dass diese zur Klasse 1 gehört.

Das k-XOR-Problem besitzt nur zwei mögliche Ausgaben, 0 und 1, wodurch es sich für die Verwendung der BCE eignet. Die BCE wird hierbei für jede Eingabepermutation berechnet und dann aufsummiert. Die Summe wird durch die Anzahl der Permutationen geteilt, um den Durchschnitt zu berechnen. Im Fall des "one-hot detector"-XOR-Problems werden die Fehlerwerte von false positives und false negatives noch getrennt gewichtet. Dies soll bei der sehr ungleichen Verteilung von 0 und 1 in den Ausgaben helfen und verhindern, dass die wenigen true positive-Fälle ignoriert werden.

Bevor die Ausgaben an die Fitnessfunktionen übergeben werden, werden sie mit Hilfe einer Sigmoid-Funktion in den Bereich $[0, 1]$ transformiert. Die Sigmoid-Funktion ist eine Aktivierungsfunktion, die in neuronalen Netzwerken verwendet wird, um die Ausgabe eines Neurons in einen Wert zwischen 0 und 1 zu transformieren. Sie wird häufig in binären Klassifikationsproblemen verwendet, da sie die Wahrscheinlichkeit angibt, dass eine Eingabe zur Klasse 1 gehört. Die Sigmoid-Funktion wird durch die Formel definiert:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.8}$$

Nach mehreren Versuchen wurde festgestellt, dass das k-XOR-Problem mit einem k von Zehn und der "one-hot detector" Variante zu leicht war und meist innerhalb von wenigen Generationen eine Lösung gefunden wurde. Somit wurde eine neue Variante des k-XOR-Problems implementiert, die für ungerade Anzahlen von 1 in den Eingaben positive Ausgaben angibt. Diese Variante ist deutlich schwieriger, da nun nicht mehr auf nur eine 1 sondern auf deren Anzahl geachtet werden muss. Mit dieser neuen Variante wurde das k-XOR-Problem nun mit einem k von Sechs erneut getestet. Hierbei zeigte sich die Lösungsfindung als deutlich schwieriger mit meist nur sehr späten Lösungen. Dies

eignet sich für die Analyse besser, da bei der allgemein längeren Laufzeit des Problems Unterschiede in der Performance der PRNGs besser sichtbar werden sollte.

Spiraldaten

Die Spiraldaten, wurden basierend auf einem einfachen logarithmischen Algorithmus implementiert, der es erlaubt, eine beliebige Anzahl von Spiralen zu generieren. Hierbei wurde über den Parameter n die Anzahl der Spiralen pro Klasse, r der maximale Radius der Spiralen, *step* die Schrittweite vom Ursprung und *resolution* die Schrittweite des Winkels festgelegt. Zusätzlich wurde ein Parameter *noise* hinzugefügt, der jeden Punkt um zufällige Werte horizontal und vertikal verschieben kann. Dieser soll eine weniger lineare Trennung der Klassen ermöglichen. Diese Spiralen wurden um den Ursprung verteilt und ihren Klassen zugeordnet.

Algorithm 8 Spiral-Daten Erzeugung 1

Require: $r > 0, step > 0, resolution > 0, noise \geq 0, angle$
  $data \leftarrow \emptyset$
  while $dist \cdot hypot(cos(angle), sin(angle)) < radius$ **do**
    $dist \leftarrow dist + step$
    $angle \leftarrow angle + resolution$
    $x \leftarrow dist \cdot cos(angle) + random(-noise, noise)$
    $y \leftarrow dist \cdot sin(angle) + random(-noise, noise)$
    $data.append((x, y))$
  end while
  return *data*

Algorithm 9 Spiral-Daten Erzeugung 2

Require: $n \geq 1, r > 0, step > 0, resolution > 0, noise \geq 0$
  $data \leftarrow \emptyset$
  for $igets0ton$ **do**
    $angle \leftarrow 2 \cdot \pi \cdot i/n$
    $spiral \leftarrow generateSpiral(r, step, resolution, noise, angle)$
    Weise jedem Punkt in *spiral* eine Klasse zu
    $data \leftarrow data + spiral$
  end for
  return *data*

Auf diese Weise wurden zwei Klassen von Spiralen generiert, die das Datenset für die zweite Problemstellung darbieten.

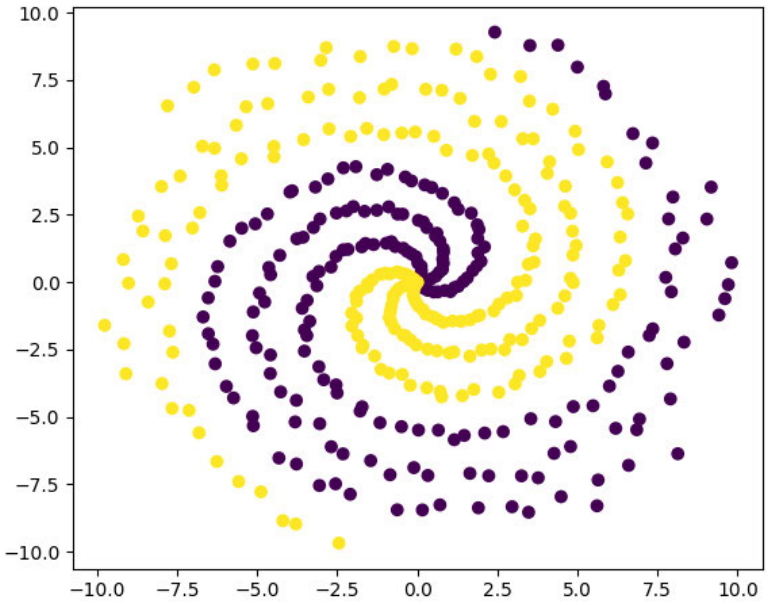

Abbildung 3.3: Generierte Spiraldaten

Bei dieser Implementierung handelt es sich ebenfalls um eine binäre Klassifikationsaufgabe, da nur zwei Klassen existieren. Somit ist die Lossfunktion ebenfalls die BCE.

Pole-balancing

Das Pole-balancing-Problem wurde basierend auf dem bestehenden Code der NEAT-Bibliothek implementiert. Diese stellt bereits eine Umgebung für das Pole-Balancing-Problem jedoch nur eine Version mit einem Stab bereit. Somit wurde die Umgebung so angepasst das sie mit mehreren Stäben umgehen kann. Dabei wurden die Berechnungen für den einzelnen Stab erweitert, um zwei Stäbe zu unterstützen. Die Berechnungen für die Fitnessfunktion wurden ebenfalls erweitert, um das Verhalten für beide Stäben zu beachten.

Wie bereits in der Beschreibung des Pole-Balancing-Problems erwähnt, gibt es eine Reihe von Parametern, die die Dynamik des Systems beeinflussen. Diese Parameter wurden

in der Umgebung als Konstanten definiert und können somit nicht verändert werden. Folgende Parameter wurden definiert:

Parameter	Wert	Bedeutung
g	$9.8m/s^2$	Erdbeschleunigung
l_1	$2m$	Länge des ersten Stabs
l_2	$1.2m$	Länge des zweiten Stabs
m_c	$1kg$	Masse des Wagens
m_{p_1}	$0.1kg$	Masse des Stabs
m_{p_2}	$0.075kg$	Masse des zweiten Stabs
h	$\pm 2.4m$	Breitenbegrenzung
r	$45°$	Winkelbegrenzung

Tabelle 3.2: Parameter des Pole-Balancing-Problems

Der nächste Schritt ist die Funktion, die für den zeitlichen Ablauf des Problems zuständig ist, für zwei Stäbe anzupassen. Diese Funktion berechnet die Änderungen des Systems basierend auf den gegebenen Eingaben und der vergangenen Zeit Δt. Zunächst wird der Winkel der Stäbe $\theta_{1|2}$ berechnet, basierend auf der Geschwindigkeit $\dot{\theta}_{1|2}$ der Beschleunigung $\ddot{\theta}_{1|2}$ und dem vorherigen Winkel $\theta_{1|2}$.

$$\theta_{1|2} = \theta_{1|2} + \Delta t \cdot \dot{\theta}_{1|2} + 0.5 \cdot \ddot{\theta}_{1|2} \cdot \Delta t^2 \tag{3.9}$$

Auf dem selben Weg wird auch die Position des Wagens x berechnet, basierend auf der Geschwindigkeit $\dot{x}$ und der Beschleunigung $\ddot{x}$.

$$x = x + \Delta t \cdot \dot{x} + 0.5 \cdot \ddot{x} \cdot \Delta t^2 \tag{3.10}$$

Anschließend wird die neue Beschleunigung der Stäbe $\ddot{\theta}'_{1|2}$ berechnet, basierend auf dem Impuls F, der auf den Wagen wirkt. Die Formel von $\ddot{\theta}'_{1|2}$ basiert auf der, von Răzvan V. Florian [7] erforschten, Formel und wurde für zwei Stäbe angepasst. Im Fall dieser Arbeit wurde die Formel ohne Reibung und die Trägheit des Wagens gewählt, da diese

den Rahmen der Arbeit sprengen würden.

$$\ddot{\theta}'_{1|2} = \frac{g \cdot sin(\theta_{1|2}) + cos(\theta_{1|2}) \cdot \left(\frac{-F - m_{p_{1|2}} \cdot l_{1|2} \cdot \dot{\theta}^2_{1|2} \cdot sin(\theta_{1|2})}{m_c + m_{p_1} + m_{p_2}} \right)}{l_{1|2} \cdot \left(\frac{4}{3} - \frac{m_{p_{1|2}} \cdot cos(\theta_{1|2})^2}{m_c + m_{p_1} + m_{p_2}} \right)} \tag{3.11}$$

Auch die Beschleunigung des Wagens $\ddot{x}'$ wird basierend auf dem Impuls F berechnet:

$$\ddot{x}' = \frac{F + m_{p_1} \cdot l_1 \cdot \left(\dot{\theta}^2_1 \cdot sin(\theta_1) - \ddot{\theta_1} \cdot cos(\theta_1) \right) + m_{p_2} \cdot l_2 \left(\dot{\theta}^2_2 \cdot sin(\theta_2) - \ddot{\theta_2} \cdot cos(\theta_2) \right)}{m_c + m_{p_1} + m_{p_2}} \tag{3.12}$$

Mit den Beschleunigungen kann nun die Geschwindigkeiten der Stäbe $\dot{\theta}'_{1|2}$ und des Wagens $\dot{x}'$ berechnet werden. Die Geschwindigkeiten der Stäbe basieren auf der alten Beschleunigung $\ddot{\theta_{1|2}}$ und neuen Beschleunigung $\ddot{\theta_{1|2}}'$, der alten Geschwindigkeit $\dot{\theta_{1|2}}$ und der vergangenen Zeit Δt.

$$\dot{\theta}'_{1|2} = \dot{\theta}_{1|2} + 0.5 \cdot (\ddot{\theta}_{1|2} + \ddot{\theta}'_{1|2}) \cdot \Delta t \tag{3.13}$$

Dem entsprechend werden auch die Geschwindigkeiten des Wagens $\dot{x}'$ berechnet.

$$\dot{x}' = \dot{x} + 0.5 \cdot (\ddot{x} + \ddot{x}') \cdot \Delta t \tag{3.14}$$

Die somit ermittelten Werte werden in die Umgebung übernommen und die Funktion gibt die neuen Werte für $\theta_{1|2}$, $\dot{\theta}_{1|2}$, x und $\dot{x}$ zurück.

Die Fitnessfunktion für das Pole-Balancing-Problem ist wohl die komplexeste und zeitaufwendigste der drei Problemstellungen. Grundsätzlich basiert die Fitness des Genoms auf der Zeit, die das Netzwerk schafft, die Stäbe zu balancieren. Da die Stäbe jedes mal anders starten, wird die Fitness pro Genom insgesammt fünf mal berechnet. Aus diesen fünf Werten wird dann der schlechteste Wert als Fitnesswert verwendet. Dies soll Genome bevorzugen, die Konsistent sind und nicht nur zufällig eine gute Lösung gefunden haben. Neben der Balancezeit werden weitere Werte in der Fitnessfunktion berücksichtigt. So wird die Fitness durch die Position des Wagens und der Stäbe beeinflusst. Für den Wagen wird die Position in der Mitte des Bildschirms bevorzugt und eine Strafe basierend auf

der Entfernung zur Mitte vergeben.

$$positionPenalty = 0.2 \cdot \left(\frac{|sim.x|}{sim.position\_Limit} \right) \tag{3.15}$$

Die Stäbe werden ebenfalls bevorzugt, wenn sie Senkrecht auf dem Wagen stehen und eine Strafe basierend auf dem Winkel vergeben.

$$anglePenalty = 0.5 \cdot \left(\left(\frac{|sim.\theta_1|}{sim.angle\_limit} \right) + \left(\frac{|sim.\theta_2|}{sim.angle\_limit} \right) \right) \tag{3.16}$$

Zusätzlich wird eine Strafe für die Geschwindigkeit der Stäbe und des Wagen vergeben, um Genome zu bevorzugen, die eine langsame und stabile Lösung gefunden haben.

$$velocityPenalty = 0.1 \cdot \left(\left(\frac{|sim.\dot{x}|}{1.5} \right) + \left(\frac{|sim.\dot{\theta_1}|}{2} \right) + \left(\frac{|sim.\dot{\theta_2}|}{2} \right) \right) \tag{3.17}$$

Zuletzt wird eine Strafe vergeben wenn der Wagen die Begrenzungen der Strecke erreicht oder die Stäbe über die Winkelgrenzen hinausgehen. Diese Strafe beendet den Versuch und wird mit der verbleibenden Zeit multipliziert.

Für diese Implementierung wurde die Balancezeit auf 30 Sekunden festgelegt, da dies ein guter Kompromiss zwischen Geschwindigkeit und Genauigkeit ist.

Um NEAT eine bessere Chance zu geben, eine Lösung zu finden, wurde das Netzwerk zuerst seperat trainiert, mit jeweils nur einem der Stäbe in Bewegung. Erst nachdem das Netzwerk mit beiden Stäben allein eine Lösung gefunden hat, wurde es mit beiden Stäben zusammen trainiert. Dies soll die Komplexität des Problems reduzieren und dem Netzwerk erlauben, die Auswirkungen seiner Aktionen auf die Stäbe und den Wagen zu verstehen, bevor es mit der gesammten Komplexität des Problems umgehen muss. Hierzu wurde die Implementierung des Wagens und der Stäbe so angepasst, dass die Berechnung des nächsten Simulationschrittes sich durch die aktuelle Generationsanzahl steuern lässt. So kann in den ersten Generationen nur ein Stab bewegt werden und in den späteren Generationen beide Stäbe. Nach mehreren Versuchen wurde festgestellt, dass sich nach den ersten 50 Generationen eine gute Lösung für das Balancieren des ersten Stabs gefunden wurde. Anschließend wurde das Netzwerk für den zweiten Stab trainiert, um sicherzustellen, dass das Netzwerk auch die nötigen Verbindungen für diesen Stab gelernt hat. Häufig konnte bereits nach 15 weiteren Generationen ein Ansatz zur Problemlösung/Balancieren des Stabs gefunden werden. Um zu verhindern, das sich das

Algorithm 10 Two-Pole-Balancing Fitnessfunktion

Require: $genome, repeats \leftarrow 5, simulationTime \leftarrow 30$
  $fitnesses \leftarrow \emptyset$
  for $i \leftarrow 0$ **to** $repeats$ **do**
    $fitness \leftarrow simulationTime$
    $time \leftarrow 0$
    $sim \leftarrow new\ PoleCart$
    while $time < simulationTime$ **do**
      $input \leftarrow sim.state()$
      $action \leftarrow genome$predicts
      $sim.step(action)$
      $time \leftarrow time + 1$
      Calculate $positionPenalty$
      Calculate $anglePenalty$
      Calculate $velocityPenalty$
      $outOfBoundsPenalty \leftarrow 1$
      if $|sim.x| > sim.position_limit$ **then**
        $fitness \leftarrow fitness - outOfBoundsPenalty \cdot positionPenalty$
      end if
      if $|sim.\theta_1| > sim.angle_limit$ **or** $|sim.\theta_2| > sim.angle_limit$ **then**
        $fitness \leftarrow fitness - outOfBoundsPenalty \cdot anglePenalty$
      end if
      $fitness \leftarrow fitness - positionPenalty - anglePenalty - velocityPenalty$
    end while
    $fitnesses.append(fitness)$
  end for
  return $min(fitnesses)$

Netzwerk zu sehr auf den zweiten Stab spezialisiert, wurde nach diesen Generationen mit beiden Stäben weiter trainiert.

3.3.4 Konfiguration

Die NEAT-Bibliothek bietet eine Vielzahl an Konfigurationsmöglichkeiten, um den Algorithmus an die Problemstellung anzupassen. Hierbei wurden die Konfigurationen möglichst für die drei Problemstellungen optimiert, allerdings nicht für die PRNGs. Dies wurde bewusst nicht gemacht, um die Ergebnisse der verschiedenen PRNGs vergleichen zu können.

Für die grobe Konfiguration wurden vorerst manuelle Änderungen an den einzelnen Konfigurationsdateien vorgenommen. Diese Voreinstellungen wurden dann mittels Gridsearch weiter optimiert. An diesem Punkt wurden die Konfigurationen für die Problemstellungen so angepasst, dass die besten Ergebnisse erzielt wurden. Da Gridsearch sehr rechenintensiv ist, wurden nur bestimmte Parameter angepasst und die anderen auf den Standardwerten belassen. Folgende Parameter wurden angepasst:

Parameter	Mögliche Werte
Aktivierungsfunktion	ReLU, Sigmoid, Tanh
Neue Verbindungs-Wahrscheinlichkeit	0.1, 0.2, 0.3
Neue Knoten-Wahrscheinlichkeit	0.2, 0.3, 0.4
Verbindungsgewicht Mutationsstärke	0.3, 0.4, 0.6, 1.0
Verbindungsgewicht Mutationsrate	0.3, 0.4, 0.6, 0.8
Ähnlichkeitsgrenze	2.0, 2.5, 3.0
Elitisimus	2, 5
Überlebensrate	0.2, 0.3, 0.4

Tabelle 3.3: Gridsearch Parameter

Diese Auswahl ist sehr klein und auch nicht optimal, jedoch war dies die einzige Möglichkeit, die Konfigurationen zu optimieren, ohne die Rechenzeit zu sehr zu erhöhen. Selbst mit diesen wenigen Parametern dauerte die Gridsearch mehrere Stunden, um alle möglichen Kombinationen zu testen. Diese kleine Auswahl an Parametern ist der Grund warum die Konfigurationen zuerst manuell angepasst wurden, um einen guten Ausgangspunkt

für die Gridsearch zu haben. Nach Durchführung der Gridsearch wurden die Konfigurationen erneut manuell angepasst, um bei vereinzelten Parametern Zwischenwerte zu testen.

Für die Problemstellungen ergaben sich folgende Konfigurationen:

Parameter	k-XOR	Spiraldaten	Pole-Balancing
Aktivierungsfunktion	Tanh	Tanh	Tanh
Neue Verbindungs-Wahrscheinlichkeit	0.2	0.2	0.2
Neue Knoten-Wahrscheinlichkeit	0.2	0.2	0.4
Verbindungsgewicht Mutationsstärke	0.4	1.0	1.0
Verbindungsgewicht Mutationsrate	0.3	0.8	0.8
Ähnlichkeitsgrenze	3.0	2.0	2.5
Elitisimus	2	5	5
Überlebensrate	0.2	0.4	0.2

Tabelle 3.4: Problemstellungs-Konfigurationen nach Gridsearch und manueller Anpassung

Die maximale Anzahl der Generationen für die drei Problemstellungen wurden basierend auf deren Fitnessverlauf gewählt. Um dies zu erreichen wurden zuerst Testdurchläufe mit einer maximalen Generationenanzahl von 1000 durchgeführt. Die Generationenanzahl wurde dann dort angesetzt wo die durchschnittliche Fitnesssteigung stark Nachläst. Unter diesen Umständen ergaben sich folgende Ergebnisse:

	k-XOR	Spiraldaten	Pole-Balancing
Max. Generationen	400	300	300

Tabelle 3.5: Problemstellungs-Konfigurationen nach Gridsearch und manueller Anpassung

Die vollständigen Konfigurationen sind im Anhang A.1 zu finden.

3.3.5 Durchführung

Mit den implementierten Problemstellungen und den angepassten Konfigurationen wurde der NEAT-Algorithmus für die drei Problemstellungen durchgeführt. Der Algorithmus wurde pro Problemstellung 100 mal durchgeführt, um die Ergebnisse zu stabilisieren.

4 Analyse

Mit den implementierten und konfigurierten Problemstellungen fertig, kann die Analyse der Ergebnisse beginnen. In diesem Kapitel werden die Ergebnisse der Versuche vorgestellt und diskutiert. Der Ablauf sieht wie folgt aus: Zuerst wird die Performance der PRNG-Algorithmen für die Problemstellung betrachtet. Dies wird durch eine graphische Darstellung der Fitnesswerte und der benötigten Generationen für die Konvergenz erreicht. Anschließend werden die wichtigsten Werte in einer Tabelle angegeben.

Mit diesen Informationen kann dann eine Analysis of Variance (ANOVA) durchgeführt werden, um die statistisch signifikanten Unterschiede der Ergebnisse zu bestimmen. Sofern die Ergebnisse Unterschiede aufweisen, wird eine Post-Hoc-Analyse durchgeführt, um die Unterschiede genauer zu bestimmen.

Dies wird für alle drei Problemstellungen durchgeführt.

4.1 ANOVA

Für die Analyse der Ergebnisse der Algorithmen für die verschiedenen Probleme wurde eine Analysis of Variance (ANOVA) durchgeführt. Das ANOVA-Verfahren ist ein statistisches Verfahren, welches die Varianz der abhängigen Variable in Gruppen aufteilt und überprüft, ob die Varianz innerhalb der Gruppen signifikant von der Varianz zwischen den Gruppen abweicht. [25]

Bei der Varianzanalyse wird eine Nullhypothese und eine Alternativhypothese aufgestellt. Für diese Untersuchung lautet die Nullhypothese: Die Fitness der Algorithmen weichen nicht signifikant voneinander ab. Die Alternativhypothese hingegen besagt, dass die Fitness der Algorithmen signifikant voneinander abweicht. Ziel der ANOVA ist es, die Nullhypothese zu widerlegen oder zu bestätigen und somit zu zeigen, ob die Fitness der Algorithmen signifikant voneinander abweicht. [25]

Ein zentraler Bestandteil der ANOVA ist die Berechnung der F-Statistik, die das Verhältnis der Varianz zwischen den Gruppen zur Varianz innerhalb der Gruppen angibt. Eine hohe F-Statistik deutet darauf hin, dass die Unterschiede zwischen den Gruppen nicht nur durch Zufall entstehen, sondern tatsächlich systematische Effekte bestehen. [25]

Eng mit der F-Statistik verknüpft ist der p-Wert, der angibt, mit welcher Wahrscheinlichkeit ein derartiger Unterschied zwischen den Gruppen unter der Nullhypothese beobachtet würde. Ein niedriger p-Wert, typischerweise unter einem Signifikanzniveau von 0,05, weist darauf hin, dass die Nullhypothese verworfen werden kann und signifikante Unterschiede zwischen den Algorithmen bestehen. Ist der p-Wert hingegen über diesem Schwellwert, gibt es keine ausreichende Evidenz, um auf eine tatsächliche Differenz zwischen den Algorithmen zu schließen. [25]

Ein wesentlicher Vorteil des ANOVA-Verfahrens ist, dass mehrere Algorithmen gleichzeitig verglichen werden können, ohne dass wiederholte Paarvergleiche erforderlich sind. Darüber hinaus ist die ANOVA zuverlässiger gegenüber kleinen Abweichungen von der Normalverteilungsannahme, insbesondere wenn die Stichprobengrößen pro Gruppe annähernd gleich groß sind. [25]

ANOVA wurde für diese Arbeit gewählt, da es sich um ein einfaches und effektives Verfahren handelt, um die Performance von mehreren Algorithmen statistisch zu überprüfen. Die Ergebnisse der ANOVA-Analyse für die verschiedenen Probleme sind in den folgenden Abschnitten dargestellt.

4.2 k-XOR Analyse

In diesem Abschnitt wird die Leistung der Algorithmen für das k-XOR Problem betrachtet. Für das k-XOR Problem wurden leider keine perfekten Lösungen gefunden. Dies ist für den Zweck der Analyse jedoch nicht weiter schlimm, da die Leistung der PRNG-Algorithmen dennoch verglichen werden kann.

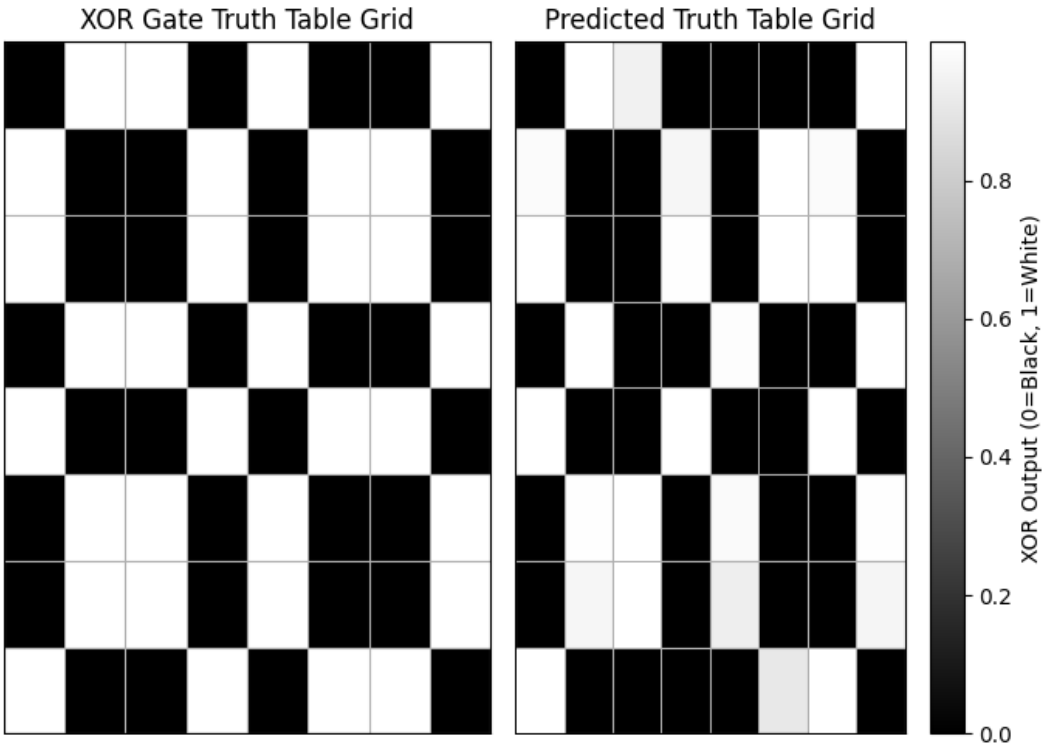

Abbildung 4.1: Visualisierung des Erwartungswertes und des Ergebnisses für das k-XOR Problem

In Abbildung 4.1 wird die durchschnittliche Leistung der Algorithmen für das k-XOR Problem dargestellt. Hierbei ist die Darstellung der Wahrheitswerte so geordnet, dass oben Links der Wert für die Eingabe $[0, 0, 0, 0, 0, 0]$ steht. Von dort wird nach rechts weitergezählt, bis der Wert $[0, 0, 0, 1, 1, 1]$ erreicht ist. Anschließend wird eine Zeile nach unten gegangen und der Prozess wiederholt. Diese Darstellung wurde gewählt, um die Ergebnisse besser vergleichbar zu machen und übersichtlicher als mit einer vollen Wahrheitstabellen-Darstellung.

4.2.1 Testauswertung

Zuerst die Fitness der Algorithmen betrachtet. In den folgenden Grafiken werden für jeden PRNG-Algorithmus der Fitnessverlauf der folgenden Durchläufe dargestellt. Diese sind der Durchlauf mit der besten Fitness, der Durchlauf mit der schlechtesten Fitness,

der Durchschnitt der Fitnesswerte aller Durchläufe und jeweils das 25. und 75. Perzentil der Fitnesswerte.

Diese Werte wurden gewählt, um die Leistung der Algorithmen besser vergleichbar zu machen. So kann beispielsweise der Durchschnitt der Fitnesswerte als Maß für die generelle Leistung eines Algorithmus genutzt werden. Das 25. und 75. Perzentil hingegen geben an, wie stabil die Fitnesswerte sind. Ein Algorithmus, der in der Lage ist, in vielen Durchläufen eine hohe Fitness zu erreichen, wird eine geringe Differenz zwischen dem 25. und 75. Perzentil haben. Die extremen Fitnesswerte verdeutlichen, wie effektiv der Algorithmus darin ist, ein hohes Leistungsniveau zu erzielen.

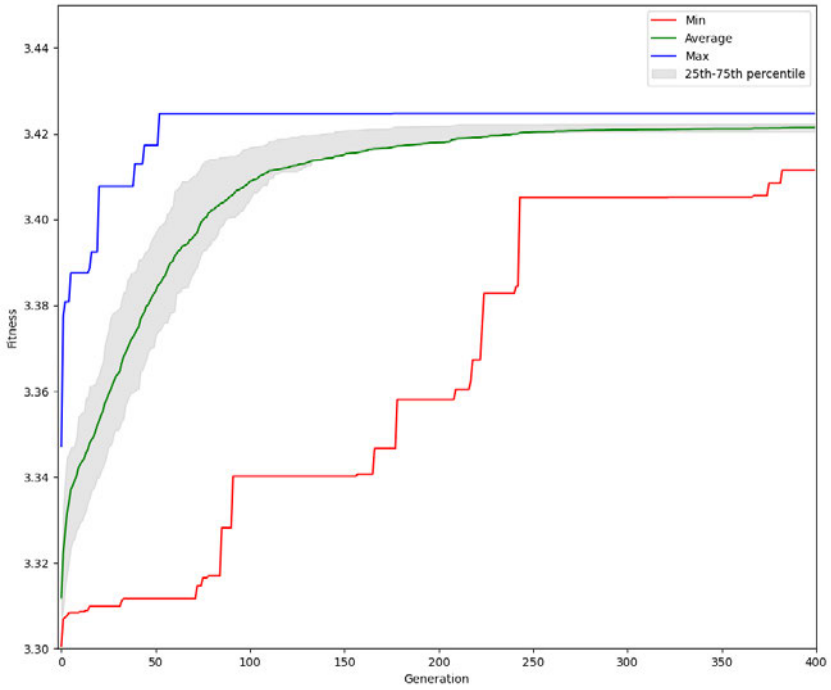

Abbildung 4.2: Fitnesswerte des Xorshift-Algorithmus für das k-XOR Problem

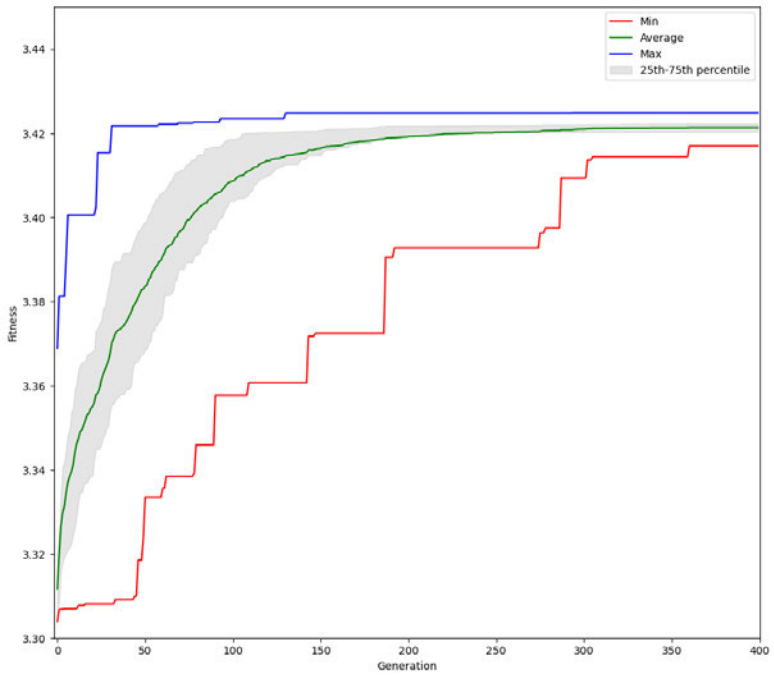

Abbildung 4.3: Fitnesswerte des MT-Algorithmus für das k-XOR Problem

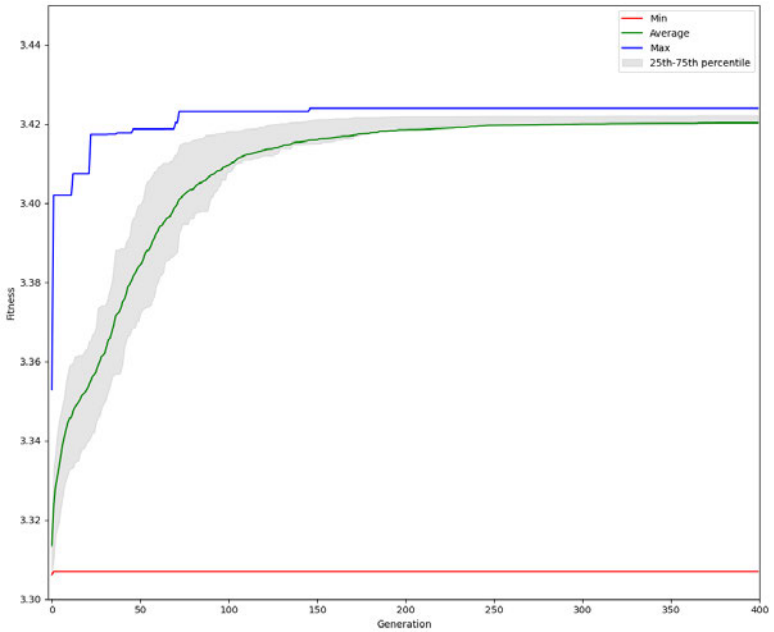

Abbildung 4.4: Fitnesswerte des CMWC-Algorithmus für das k-XOR Problem

In den Abbildungen 4.2, 4.3 und 4.4 wird der Fitnessverlauf der Algorithmen für das k-XOR Problem dargestellt. Zusätzlich zu diesen Fitnessverläufen wurden die finalen Fitnesswerte, die benötigten Generationen und die benötigte Zeit der Durchläufe pro PRNG-Algorithmus in den Tabellen 4.1, 4.2 und 4.3 zusammengefasst.

	Xorshift	MT	CMWC
Minimale Fitness	3.4114	3.4170	3.3068
Durchschnittliche Fitness	3.4214	3.4214	3.4202
Maximale Fitness	3.4247	3.4248	3.4239
25. Perzentil Fitness	3.4205	3.4203	3.4205
75. Perzentil Fitness	3.4223	3.4222	3.4222

Tabelle 4.1: Fitnesswerte der PRNG-Algorithmen für das k-XOR Problem

	Xorshift	MT	CMWC
Minimale Generationen	53	32	71
Durchschnittliche Generationen	193	176	197
Maximale Generationen	400	400	400
25. Perzentil Generationen	139	107	129
75. Perzentil Generationen	236	226	244

Tabelle 4.2: Benötigte Generationen der PRNG-Algorithmen für das k-XOR Problem

	Xorshift	MT	CMWC
Minimale Zeit	2.68s	1.56s	5.65s
Durchschnittliche Zeit	10.96s	10.08s	16.65s
Maximale Zeit	29.67s	24.10s	41.42s
25. Perzentil Zeit	7.70s	6.20s	10.57s
75. Perzentil Zeit	12.96s	12.94s	20.46s

Tabelle 4.3: Benötigte Zeit der PRNG-Algorithmen für das k-XOR Problem

Bereits anhand der Fitnesswerte in Tabellen 4.1 ist zu erkennen, dass die Algorithmen Xorshift und MT für das k-XOR Problem eine sehr ähnliche Leistung haben. Der CMWC-Algorithmus hingegen hat eine leicht schlechtere Leistung. Dies zeigt sich auch in

den benötigten Generationen und der benötigten Zeit. Jedoch zeigt sich hier, ein Unterschied zwischen den Algorithmen Xorshift und MT. Der Xorshift-Algorithmus benötigt im Durchschnitt mehr Generationen und Zeit als der MT-Algorithmus. Dies ist ein interessantes Ergebnis, da der Xorshift-Algorithmus in der Theorie schneller sein sollte als der MT-Algorithmus. Dies zeigt, dass die Leistung eines PRNG-Algorithmus nicht nur von der Theorie abhängt, sondern auch von der konkreten Implementierung. Hierbei profitiert der MT-Algorithmus davon, dass er bereits eine sehr gute Implementierung in der Standardbibliothek von Python hat. Der Xorshift-Algorithmus hingegen wurde für diese Arbeit selbst implementiert und ist daher nicht so optimiert wie der MT-Algorithmus.

Auch wenn die Leistung der Algorithmen für das k-XOR Problem, bereits unterschiede aufzeigt, ist es dennoch wichtig, dies statistisch zu überprüfen.

4.2.2 Statistische Analyse

Um die Leistung der Algorithmen statistisch zu analysieren, wird eine ANOVA-Analyse durchgeführt. Hierbei wird die Fitness der Algorithmen als abhängige Variable betrachtet. Mit der ANOVA wird überprüft, ob die Fitness der Algorithmen signifikant voneinander abweicht.

	Fitness	Generationen
F-Statisik	0.9072	1.799
p-Wert	0.4047	0.1671

Tabelle 4.4: Die p-Werte der ANOVA-Analyse für das k-XOR Problem

In Tabelle 4.4 sind die p-Werte der ANOVA-Analyse für das k-XOR Problem dargestellt. Für ANOVA wird ein Signifikanzniveau von 0.05 gewählt. Somit wird die Nullhypothese abgelehnt, wenn der p-Wert kleiner als 0.05 ist. Da die p-Werte für die Fitness und die benötigten Generationen größer als 0.05 sind, wird die Nullhypothese nicht abgelehnt. Dies bedeutet, dass die Fitness und die benötigten Generationen der Algorithmen für das k-XOR Problem nicht signifikant voneinander abweichen.

Dieses Ergebnis zeigt, dass die Algorithmen für das k-XOR Problem eine ähnliche Leistung haben. Dies deckt sich mit den Ergebnissen der Testauswertung, in der bereits gezeigt wurde, dass die Algorithmen für das k-XOR Problem eine ähnliche Leistung haben.

4.3 Spiraldaten Analyse

Nun wird die Performance der Algorithmen für das Spiraldaten Problem betrachtet. Die Vorgehensweise ist hierbei die Gleiche, wie bei der k-XOR-Analyse.

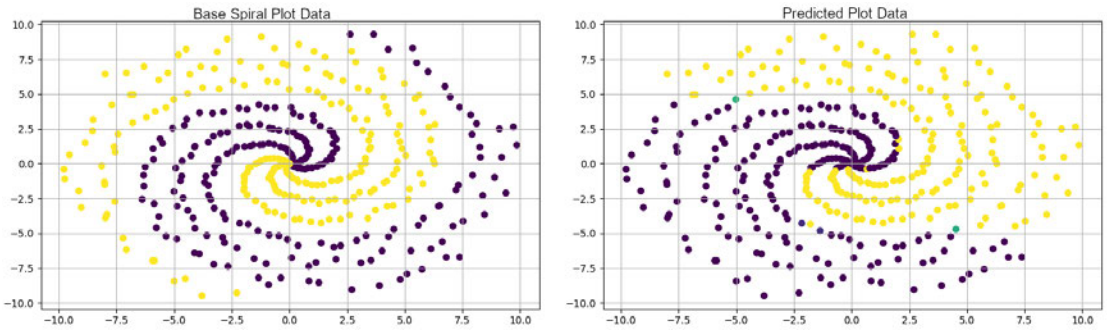

Abbildung 4.5: Visualisierung des Erwartungswertes und des Ergebnisses für das Spiraldaten Problem

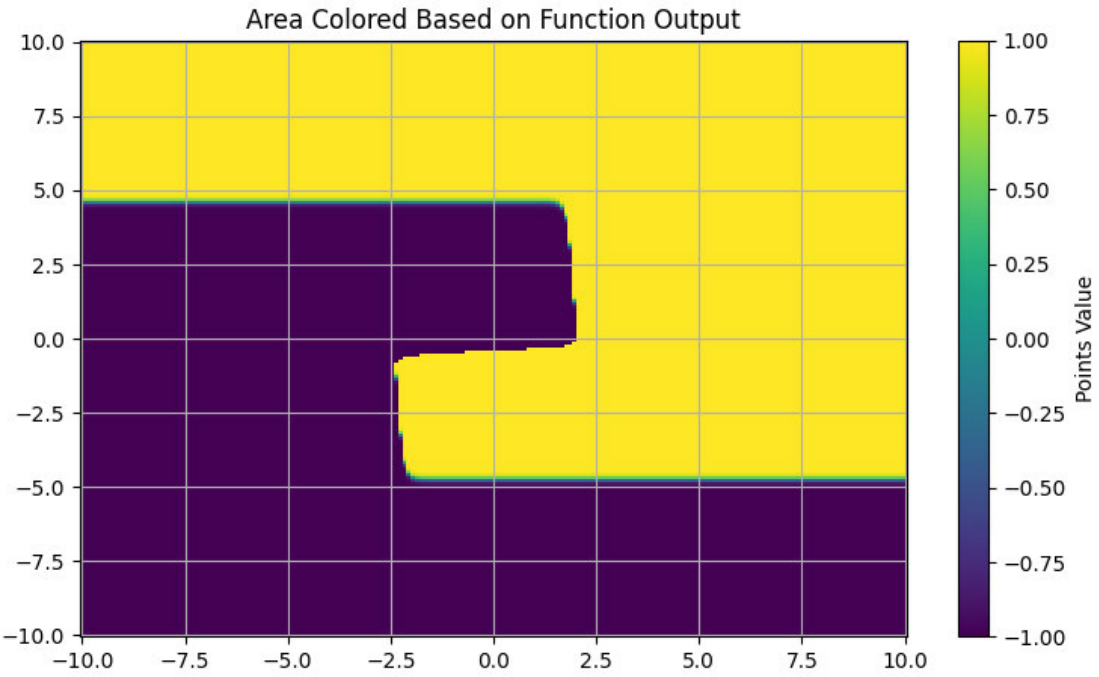

Abbildung 4.6: Visualisierung der Klassifikationsbereiche des Ergebnisses

In Abbildung 4.5 wird die durchschnittliche Performance der Algorithmen für das Spiraldaten Problem dargestellt. Zusätzlich wurde in Abbildung 4.6 das Ergebniss als Klas-

sifikationsbereiche dargestellt. Hierbei wurde die Farbe der Klassifikationsbereiche so gewählt, dass die Farbe der Klasse entspricht, die der Algorithmus für den Bereich vorhergesagt hat. Die grünen Bereiche entsprechen hierbei der Unsicherheitszone, in der der Algorithmus keine klare Entscheidung treffen konnte.

4.3.1 Testauswertung

Wie auch für das k-XOR-Problem, wird zuerst die Fitness der Algorithmen betrachtet. In den folgenden Grafiken werden für jeden PRNG-Algorithmus der Fitnessverlauf der folgenden Durchläufe dargestellt. Diese sind der Durchlauf mit der besten Fitness, der Durchlauf mit der schlechtesten Fitness, der Durchschnitt der Fitnesswerte aller Durchläufe und jeweils das 25. und 75. Perzentil der Fitnesswerte.

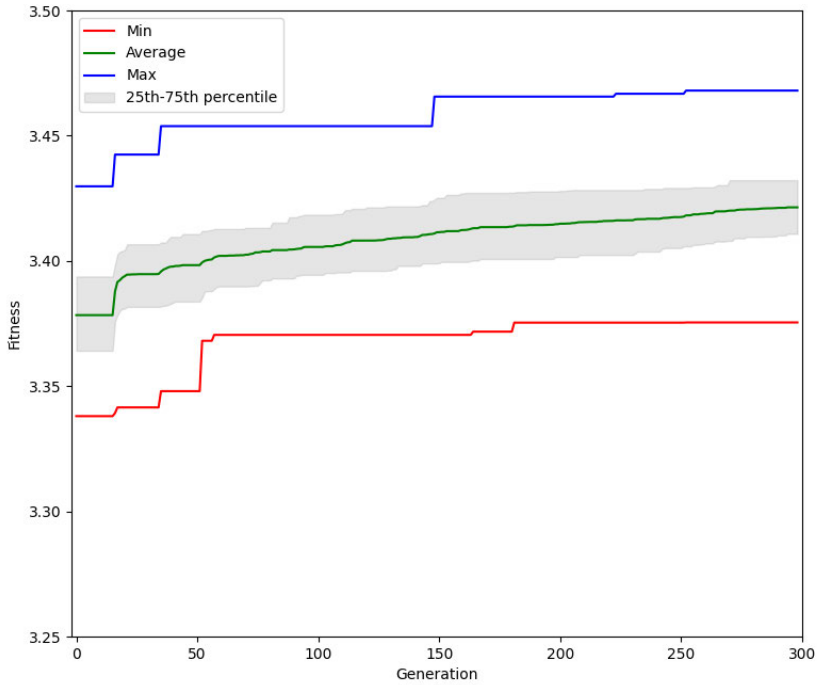

Abbildung 4.7: Fitnesswerte des Xorshift-Algorithmus für das Spiraldaten Problem

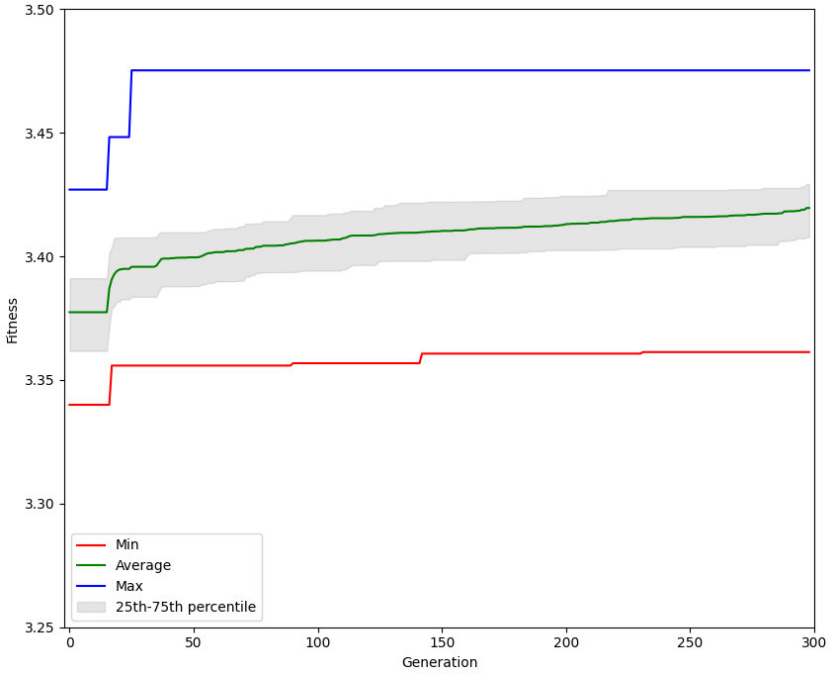

Abbildung 4.8: Fitnesswerte des MT-Algorithmus für das Spiraldaten Problem

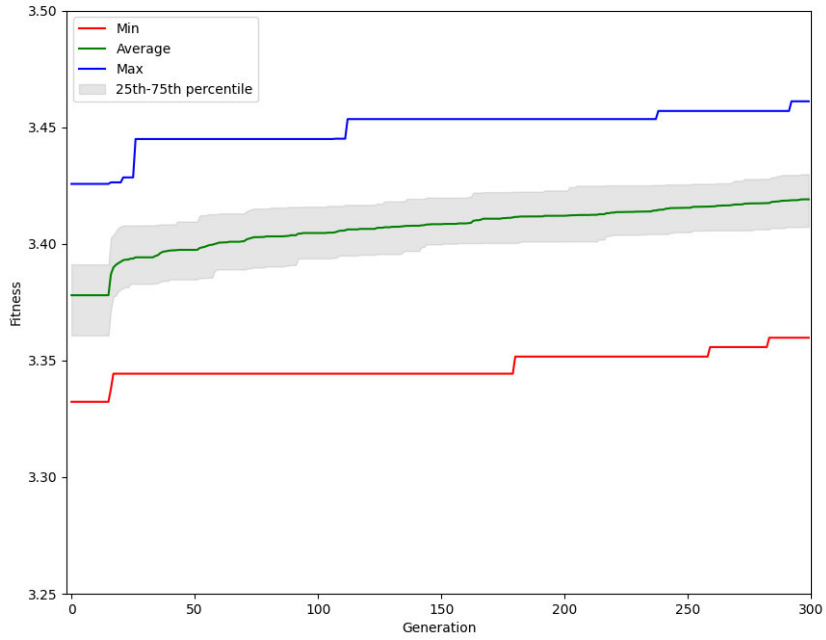

Abbildung 4.9: Fitnesswerte des CMWC-Algorithmus für das Spiraldaten Problem

In den Abbildungen 4.7, 4.8 und 4.9 wird der Fitnessverlauf der Algorithmen für das Spiraldaten Problem dargestellt. Zusätzlich zu diesen Fitnessverläufen wurden die finalen Fitnesswerte, die benötigten Generationen und die benötigte Zeit der Durchläufe pro PRNG-Algorithmus in den Tabellen 4.5, 4.6 und 4.7 zusammengefasst.

	Xorshift	Mersenne Twister	CMWC
Minimale Fitness	3.3754	3.3613	3.3597
Durchschnittliche Fitness	3.4213	3.4196	3.4188
Maximale Fitness	3.4680	3.4752	3.4610
25. Perzentil Fitness	3.4108	3.4077	3.4071
75. Perzentil Fitness	3.4322	3.4293	3.4297

Tabelle 4.5: Fitnesswerte der PRNG-Algorithmen für das Spiraldaten Problem

	Xorshift	MT	CMWC
Minimale Generationen	36	26	113
Durchschnittliche Generationen	290.21	292.73	295.18
Maximale Generationen	300	300	300
25. Perzentil Generationen	300	300	300
75. Perzentil Generationen	300	300	300

Tabelle 4.6: Benötigte Generationen der PRNG-Algorithmen für das Spiraldaten Problem

	Xorshift	MT	CMWC
Minimale Zeit	43.05s	38.53s	68.01s
Durchschnittliche Zeit	95.73s	96.87s	104.09s
Maximale Zeit	115.66s	128.27s	138.93s
25. Perzentil Zeit	91.56s	92.50s	98.55s
75. Perzentil Zeit	101.23s	101.11s	109.17s

Tabelle 4.7: Benötigte Zeit der PRNG-Algorithmen für das Spiraldaten Problem

In diesen Daten ist zu beachten, dass die benötigten Generationen für alle Algorithmen nahezu identisch sind. Die Fitnesswerte und die benötigte Zeit sind ebenfalls sehr ähnlich. Dies deutet darauf hin, dass die Algorithmen für das Spiraldaten Problem sehr ähnliche Ergebnisse liefern.

4.3.2 Statistische Analyse

Auch für das Spiraldaten Problem wird eine ANOVA durchgeführt.

	Fitness	Generationen
F-Statisik	0.4536	0.5773
p-Wert	0.6357	0.5620

Tabelle 4.8: Die p-Werte der ANOVA-Analyse für das Spiraldaten Problem

In Tabelle 4.8 sind die p-Werte der ANOVA für das Spiraldaten Problem dargestellt. Auch für das Spiraldaten Problem zeigt sich, dass die Fitness und die benötigten Generationen der Algorithmen keine signifikanten Unterschiede aufweisen. Die Daten für das Spiraldaten Problem sind sogar noch ähnlicher als die Daten für das k-XOR-Problem. Somit kann auch für das Spiraldaten Problem die Nullhypothese nicht abgelehnt werden. Dies bedeutet, dass die Fitness und die benötigten Generationen der Algorithmen für das Spiraldaten Problem nicht signifikant voneinander abweichen.

4.4 Two-Pole-Balancing Analyse

Zuletzt wird die Performance der Algorithmen für das Two-Pole-Balancing Problem betrachtet. Das Two-Pole-Balancing Problem hatte jedoch starke Schwierigkeiten, konvergente Ergebnisse zu erzielen. Dies ist darauf zurückzuführen, dass das Problem sehr komplex ist und die Algorithmen Schwierigkeiten haben, eine Lösung zu finden. Selbst mit mehren Schritten um die Konsistenz der Ergebnisse zu erhöhen, war es nicht möglich, konvergente Ergebnisse zu erzielen. Änderungen, wie die Anpassung der Populationsgröße oder der Mutationsrate haben keine signifikanten Verbesserungen gebracht. Auch die Aufteilung des Lernporzesses in mehrere Phasen hat keine Verbesserungen gebracht. Weder die Anpassung der Simulationsparameter, wie die Länge der Stäbe, noch die Erhöhung der maximalen Generationenzahl führten zu Verbesserungen. Auch mehrere Modifikationen der Fitnessfunktion zeigten keinen positiven Effekt. Trotz all dieser Änderungen konnten keine konvergenten Ergebnisse erzielt werden.

Aus diesem Grund wird die Analyse der Ergebnisse für das Two-Pole-Balancing Problem, stattdessen auf das Single-Pole-Balancing Problem fokussiert. Dieses Problem ist einfacher und die Algorithmen können konvergente Ergebnisse erzielen.

4.4.1 Testauswertung

Das Single-Pole-Balancing Problem ist eine vereinfachte Version des Two-Pole-Balancing Problems. Diese Einfachheit spiegelt sich auch stark in den Ergebnissen wieder. Die Algorithmen konnten konvergente Ergebnisse erzielen und die Fitnesswerte konnten stabilisiert werden. Dies meist sogar in ein bis zwei Generationen. Selbst die langsamsten Durchläufe konnten in unter 100 Generationen konvergente Ergebnisse erzielen. Eine weitere Besonderheit ist der Verlauf der Fitnesswerte in den Abbildungen 4.10, 4.11 und 4.12. Durch die starke Bestrafung für das Umkippen der Stäbe, sind die Fitnesswerte anfangs sehr niedrig und steigen sehr stark mit dem Lernfortschritt an. Dies ist ein deutlicher Unterschied zu den anderen Problemstellungen, bei denen die Fitnesswerte meistens schon zu Beginn relativ hoch sind und nur langsam ansteigen.

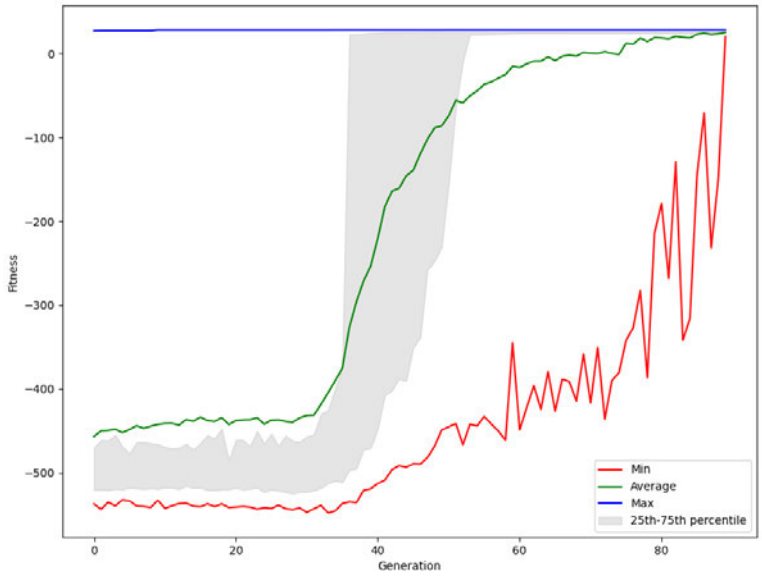

Abbildung 4.10: Fitnesswerte des Xorshift-Algorithmus für das Single-Pole-Balancing Problem

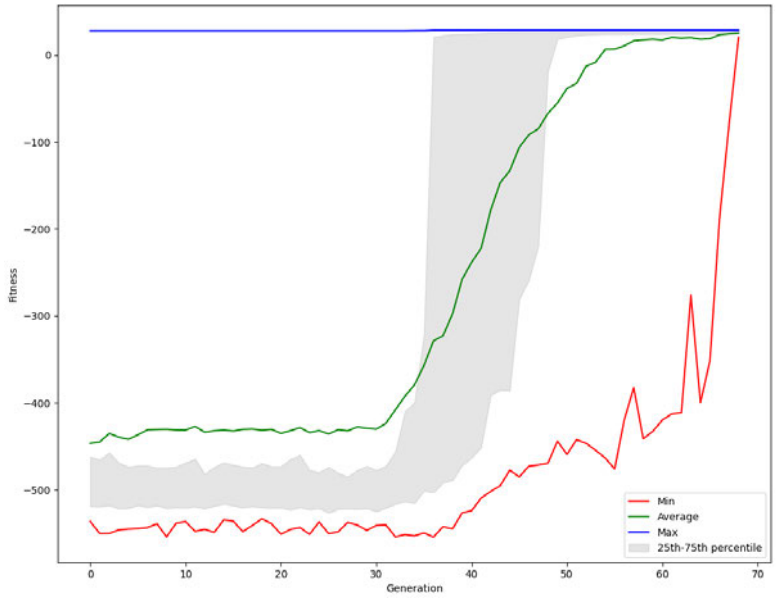

Abbildung 4.11: Fitnesswerte des MT-Algorithmus für das Single-Pole-Balancing Problem

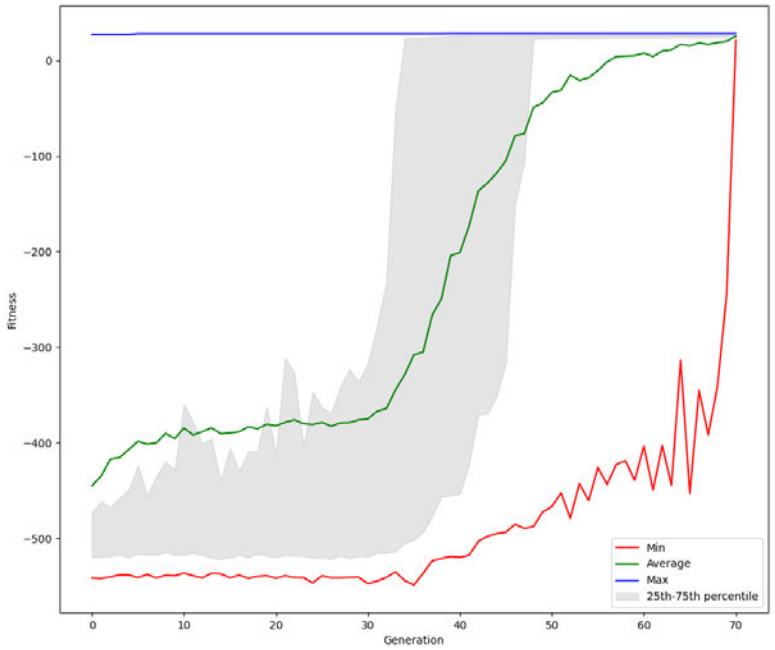

Abbildung 4.12: Fitnesswerte des CMWC-Algorithmus für das Single-Pole-Balancing Problem

Aufgrund der leichten Randomisierung der Initialisierung der Simulation sind starke Schwankungen in den Fitnesswerten zu erkennen. Dies ist darauf zurückzuführen, dass die Simulationen nicht deterministisch sind und die Ergebnisse von den Startwerten abhängen. Dieser Zufall der Initialisierung ist jedoch notwendig, um die Generalisierung der Algorithmen zu gewährleisten.

In den Abbildungen 4.10, 4.11 und 4.12 wird der Fitnessverlauf der Algorithmen für das Single-Pole-Balancing Problem dargestellt. Zusätzlich zu diesen Fitnessverläufen wurden die finalen Fitnesswerte, die benötigten Generationen und die benötigte Zeit der Durchläufe pro PRNG-Algorithmus in den Tabellen 4.9, 4.10 und 4.11 zusammengefasst.

	Xorshift	MT	CMWCC
Minimale Fitness	20.1542	20.1563	20.9874
Durchschnittliche Fitness	25.5648	25.2280	25.3951
Maximale Fitness	28.4886	28.3484	28.1894
25. Perzentil Fitness	24.6599	23.9227	24.3287
75. Perzentil Fitness	26.5827	26.5853	26.6137

Tabelle 4.9: Fitnesswerte der PRNG-Algorithmen für das Single-Pole-Balancing Problem

	Xorshift	MT	CMWC
Minimale Generationen	1	1	1
Durchschnittliche Generationen	43.92	40.15	36.94
Maximale Generationen	90	69	71
25. Perzentil Generationen	37	37	34
75. Perzentil Generationen	53	50	49

Tabelle 4.10: Benötigte Generationen der PRNG-Algorithmen für das Single-Pole-Balancing Problem

	Xorshift	MT	CMWC
Minimale Zeit	4.38s	4.25s	5.44s
Durchschnittliche Zeit	84.53s	80.65s	94.22s
Maximale Zeit	120.74s	107.28s	142.52s
25. Perzentil Zeit	85.91s	85.91s	101.80s
75. Perzentil Zeit	96.45s	93.84s	117.83s

Tabelle 4.11: Benötigte Zeit der PRNG-Algorithmen für das Single-Pole-Balancing Problem

Wie auch bei den anderen Problemstellungen, sind die Ergebnisse der PRNG-Algorithmen sehr ähnlich. Die Fitnesswerte und die benötigten Generationen sind sehr ähnlich und es gibt keine signifikanten Unterschiede. Die benötigte Zeit ist ebenfalls sehr ähnlich und es gibt keine signifikanten Unterschiede. Somit scheint es, dass die PRNG-Algorithmen auch für das Single-Pole-Balancing Problem sehr ähnliche Ergebnisse liefern.

4.4.2 Statistische Analyse

Bevor jedoch eine finale Aussage getroffen werden kann, wird eine ANOVA durchgeführt.

	Fitness	Generationen
F-Statisik	0.8792	3.4675
p-Wert	0.4162	0.0324

Tabelle 4.12: Die p-Werte der ANOVA für das Single-Pole-Balancing Problem

Für die Fitnesswerte des Single-Pole-Balancing Problems zeigt sich, dass die Algorithmen keine signifikanten Unterschiede aufweisen. Der p-Wert beträgt 0.4162 und somit kann die Nullhypothese nicht abgelehnt werden. Für die benötigten Generationen zeigt sich jedoch ein signifikanter Unterschied. Der p-Wert beträgt 0.0324 und somit kann die Nullhypothese abgelehnt werden. Dies bedeutet, dass die Algorithmen signifikante Unterschiede in der benötigten Generationen aufweisen.

Wenn dieses Ergebniss in den Kontext der vorherigen Ergebnisse gesetzt wird, ist diese jedoch die einzige signifikante Abweichung. Für alle anderen Problemstellungen konnten

keine signifikanten Unterschiede festgestellt werden und die p-Werte waren weit über 0.05. Somit wird in der Gesamtbetrachtung der Ergebnisse davon ausgegangen, dass die PRNG-Algorithmen für die Problemstellungen keine statistischrelevanten Unterschiede in den Ergebnissen liefern.

5 Auswertung und Ausblick

5.1 Auswertung

Mit all diesen Daten und den Ergebnissen der Tests, die in den vorherigen Kapiteln beschrieben wurden, können nun die Ergebnisse analysiert und diskutiert werden. Zuvor sollte darauf hingewiesen werden, dass die Ergebnisse der Tests nicht immer eindeutig sind. Die kalkulierten p-Werte sind nur eine Wahrscheinlichkeit, dass die Nullhypothese wahr ist. Es ist also möglich, dass die Nullhypothese fälschlicherweise abgelehnt oder angenommen wird. In den Test wurden insgesamt sechs ANOVA Tests durchgeführt, wobei fünf der Tests die Nullhypothese annahmen und nur ein Test die Nullhypothese ablehnt. Simple Wahrscheinlichkeitsrechnung zeigt, dass die Wahrscheinlichkeit P, dass fünf von sechs Tests die Nullhypothese fälschlicherweise angenommen werden, bei 0.015625 oder 1.5625% liegt.

$$P = \binom{6}{5} \times 0.05^5 \times 0.95 = 0.015625 \tag{5.1}$$

Andereseits liegt die Wahrscheinlichkeit, dass ein von sechs Tests die Nullhypothese fälschlicherweise ablehnt, bei 0.09375 oder 9.375%.

$$P = \binom{6}{1} \times 0.05 \times 0.95^5 = 0.09375 \tag{5.2}$$

Somit ist es deutlich wahrscheinlicher, dass die Nullhypothese einmal fälschlicherweise abgelehnt wird als fünfmal angenommen. Aus diesem Grund wird fortführend davon ausgegangen, dass es sich bei der einen Ablehnung um eine Ausnahme handelt und die Nullhypothese korrekt ist.

Wenn also davon ausgegangen wird, dass die Nullhypothese korrekt ist, lässt sich sagen, dass die Wahl des PRNG keinen signifikanten Einfluss auf die Ergebnisse hat. Dies bedeutet, dass bei der Verwendung von NEAT der PRNG keine Rolle spielt. Diese Ergebnisse decken sich mit den Ergebnissen von Zelinka et al. [31],Lekshmi Rajashekharan et al. [20]

und Miguel Cárdenas-Montes et al. [6], die festgestellt haben, dass die Wahl des PRNG keinen signifikanten Einfluss auf die Ergebnisse der Algorithmenklasse EAs hat.

In der Praxis bedeutet dies, dass Entwickler, die NEAT verwenden, sich keine Gedanken über die Wahl des PRNGs machen müssen. Sie können einfach den Standard-PRNG ihrer Programmiersprache verwenden und sich auf die Entwicklung ihrer Anwendung konzentrieren.

5.2 Ausblick für zukünftige Arbeiten

Diese Arbeit hat gezeigt, dass die Wahl des PRNG keinen signifikanten Einfluss auf die Ergebnisse von NEAT hat. Allerdings ist es wichtig zu beachten, dass dies nur für ein sehr kleines Spektrum von PRNGs und Problemstellungen getestet wurde. In dieser Arbeit wurden nur drei der bekanntesten PRNGs getestet, und es wurden nur drei verschiedene Problemstellungen getestet. Es ist also möglich, dass die Ergebnisse für andere PRNGs und Problemstellungen anders aussehen. Besonders die Problemstellungen der Two-Pole-Balancing, an dessen Implementierung diese Arbeit gescheitert ist, lässt viel Raum für weitere Untersuchungen. Somit ist es möglich, dass eine andere Problemstellung mit anderen PRNGs andere Ergebnisse liefert.

Des Weiteren könnte das Ergebnis, dass die Wahl des PRNG keinen Einfluss hat, noch weiter eskaliert werden. Es könnte Untersucht werden, ob ein PRNG überhaupt notwendig ist, oder ob es möglich ist, NEAT ohne PRNG zu verwenden. Oder wie kurz die Periodenlänge eines PRNGs sein kann, bevor es einen Einfluss auf die Ergebnisse hat.

Zuletzt ist NEAT nicht der letzte Stand der Technik. Es gibt bereits viele Weiterentwicklungen von NEAT, die möglicherweise von der Wahl des PRNG beeinflusst werden. Es wäre also interessant zu untersuchen, ob die Wahl des PRNG bei diesen Weiterentwicklungen einen Einfluss hat. Beispielsweise könnte untersucht werden, ob die Wahl des PRNG bei HyperNEAT einen Einfluss hat. Hinzu kommt, dass NEAT auch nicht der einzige Algorithmus ist, der versucht den Prozess des MLs mittels der Ideen hinter EAs zu unterstützen oder zu implementieren. Es wäre also interessant zu untersuchen, ob die Wahl des PRNG bei anderen Algorithmen, die auf EAs basieren, einen Einfluss hat.

6 Fazit

Ziel dieser Arbeit war es, zu untersuchen, ob die Wahl des PRNG einen Einfluss auf die Ergebnisse von NEAT hat. Dazu wurden drei verschiedene PRNGs und drei verschiedene Problemstellungen getestet. Für jede Kombination aus PRNG und Problemstellung wurden 100 Durchläufe mit je 300 bis 400 Generationen durchgeführt, um die Ergebnisse zu ermitteln. Diese Ergebnisse wurden dann mittels ANOVA Tests analysiert. Die Ergebnisse dieser Tests zeigen, dass die Wahl des PRNG keinen signifikanten Einfluss auf die Ergebnisse von NEAT hat.

Während es wünschenswert gewesen wäre, jeweils ein größeres Spektrum an PRNGs und Problemstellungen zu testen, ist die Anpassung der Problemstellungen und deren Konfigurationen sehr zeitaufwendig. Besonders die mehrfachen Durchläufe die jeder PRNG benötigt um die Daten zu generieren, haben den Testprozess sehr verlangsamt. Trotzdem wurde eine möglichst breite Auswahl an PRNGs und Problemstellungen getestet, um die Ergebnisse so allgemein wie möglich zu halten.

Neben der Analyse der Ergebnisse wurde auch die Implementierung von NEAT, der Problemstellungen und Methoden zur Optimierung von Parametern diskutiert. Auch wenn die Problemstellungen keine perfekten Lösungen ergaben, so ließ sich doch erkennen, dass der durchschnittliche Verlauf der Fitnesswerte und die Anzahl der Generationen, die benötigt wurden, um eine Lösung zu finden, von der Wahl des PRNG nicht beeinflusst wurden.

Literaturverzeichnis

[1] Ahmad, Adel T.: Einfluss von RNGs auf die Effizienz von evolutionären Algorithmen. In: *Hamburg University of Applied Sciences* (2023)

[2] Back, Thomas: *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996. – 7–9 S. – URL https://books.google.de/books?id=htJHI1UrL7IC. – ISBN 9780195356700

[3] Bergstra, James ; Bengio, Yoshua: Random search for hyper-parameter optimization. In: *Journal of machine learning research* 13 (2012), Nr. 2. – URL https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

[4] Brownlee, Jason: The pole balancing problem: A benchmark control theory problem. (2005). – URL https://www.idi.ntnu.no/emner/it3105/materials/rl/cartpole-brownlee-2005.pdf

[5] Cantú-Paz, Erick: On random numbers and the performance of genetic algorithms. In: *Science Direct Working Paper No S1574-034X(04)70205-7* (2002). – URL https://ssrn.com/abstract=3125461

[6] Cárdenas-Montes, Miguel ; Vega-Rodríguez, Miguel A. ; Gómez-Iglesias, Antonio: Sensitiveness of Evolutionary Algorithms to the Random Number Generator. In: Dobnikar, Andrej (Hrsg.) ; Lotrič, Uroš (Hrsg.) ; Šter, Branko (Hrsg.): *Adaptive and Natural Computing Algorithms*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, S. 371–380. – ISBN 978-3-642-20282-7

[7] Florian, Răzvan V: Correct equations for the dynamics of the cart-pole system. In: *Center for Cognitive and Neural Studies (Coneural), Romania* (2007), S. 63. – URL https://coneural.org/florian/papers/05_cart_pole.pdf

[8] Ingrid Gerdes, Rudolf K.: *Evolutionäre Algorithmen: Genetische Algorithmen — Strategien und Optimierungsverfahren — Beispielanwendungen*. Vieweg+Teubner Verlag Wiesbaden, 2004. – 1–14 S. – URL https://doi.org/10.1007/978-3-322-86839-8. – ISBN 978-3-528-05570-7

[9] Knuth, Donald E.: *The art of computer programming*. Bd. 3. Addison Wesley Longman, 1997. – 1–4 S. – ISBN 0-201-89684-2

[10] Krömer, Pavel ; Snáel, Václav ; Zelinka, Ivan: Randomness and chaos in genetic algorithms and differential evolution. In: *2013 5th International Conference on Intelligent Networking and Collaborative Systems* IEEE (Veranst.), 2013 5th International Conference on Intelligent Networking and Collaborative Systems, 2013, S. 196–201. – URL https://doi.org/10.1109/INCoS.2013.36

[11] Lewin, M: All about XOR. In: *For details of ACCU, our publications and activities, visit the ACCU website: www. accu. org* (2012), S. 14

[12] Lynch, Shana: The state of AI in 9 charts. In: *Stanford HAI* (2022), Mar. – URL https://hai.stanford.edu/news/state-ai-9-charts

[13] Marsaglia, George: Random number generators. In: *Journal of Modern Applied Statistical Methods* 2 (2003), Nr. 1, S. 2. – URL https://digitalcommons.wayne.edu/jmasm/vol2/iss1/2

[14] Marsaglia, George: Xorshift RNGs. In: *Journal of Statistical Software* 8 (2003), Nr. 14, S. 1–6. – URL https://www.jstatsoft.org/index.php/jss/article/view/v008i14

[15] Matsumoto, Makoto ; Nishimura, Takuji: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. In: *Association for Computing Machinery* 8 (1998), jan, Nr. 1, S. 3–30. – URL https://doi.org/10.1145/272991.272995. – ISSN 1049-3301

[16] Nissen, V. ; Bibel, W. ; Kruse, R.: *Einführung in Evolutionäre Algorithmen: Optimierung nach dem Vorbild der Evolution*. Vieweg+Teubner Verlag, 2013 (Computational Intelligence). – 1–14 S. – URL https://books.google.de/books?id=FLzOBgAAQBAJ. – ISBN 9783322938619

[17] Papavasileiou, Evgenia ; Cotnelis, Jan ; Jansen, Bart: Behavior-based Speciation in Classification with NeuroEvolution. In: *2020 IEEE Congress on Evolutionary*

Computation (CEC), 2020 IEEE Congress on Evolutionary Computation (CEC), 2020, S. 1–7. – URL https://doi.org/10.1109/CEC48606.2020.9185720

[18] Petrowski, Alain ; Ben-Hamida, Sana: *Evolutionary Algorithms*. S. 1–5. In: *Evolutionary Algorithms*, John Wiley & Sons, April 2017. – URL https://books.google.de/books?id=GlGpDgAAQBAJ

[19] Python, Foundation: random — Generate pseudo-random numbers. In: *Python documentation* (2024). – URL https://docs.python.org/3/library/random.html

[20] Rajashekharan, Lekshmi ; Shunmuga Velayutham, C.: Is Differential Evolution Sensitive to Pseudo Random Number Generator Quality? – An Investigation. In: Berretti, Stefano (Hrsg.) ; Thampi, Sabu M. (Hrsg.) ; Srivastava, Praveen R. (Hrsg.): *Intelligent Systems Technologies and Applications*. Cham : Springer International Publishing, 2016, S. 305–313. – ISBN 978-3-319-23036-8

[21] Ribenboim, Paulo ; Ribenboim, Paulo: Welche besonderen Arten von Primzahlen wurden untersucht? In: *Die Welt der Primzahlen: Geheimnisse und Rekorde* (2011), S. 233–261. – URL https://doi.org/10.1007/978-3-642-18079-8_5

[22] Ruby, Usha ; Yendapalli, Vamsidhar: Binary cross entropy with deep learning technique for image classification. In: *Int. J. Adv. Trends Comput. Sci. Eng* 9 (2020), Nr. 10

[23] Scott, David W.: Box–Muller transformation. In: *WIREs Computational Statistics* 3 (2011), Nr. 2, S. 177–179. – URL https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.148

[24] Snoek, Jasper ; Larochelle, Hugo ; Adams, Ryan P.: Practical Bayesian Optimization of Machine Learning Algorithms. In: Pereira, F. (Hrsg.) ; Burges, C.J. (Hrsg.) ; Bottou, L. (Hrsg.) ; Weinberger, K.Q. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 25, Curran Associates, Inc., 2012. – URL https://proceedings.neurips.cc/paper_files/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf

[25] Ståhle, Lars ; Wold, Svante: Analysis of variance (ANOVA). In: *Chemometrics and Intelligent Laboratory Systems* 6 (1989), Nr. 4, S. 259–272. – URL https://www.sciencedirect.com/science/article/pii/0169743989800954. – ISSN 0169-7439

[26] Stanley, Kenneth O. ; Miikkulainen, Risto: Evolving Neural Networks through Augmenting Topologies. In: *Evolutionary Computation* 10 (2002), Nr. 2, S. 99–127. – URL https://doi.org/10.1162/106365602320169811

[27] Tan, Maxine ; Hartley, Michael ; Bister, Michel ; Deklerck, Rudi: Automated feature selection in neuroevolution. In: *Evolutionary Intelligence* 1 (2009), Mar, Nr. 4, S. 271–292. – URL https://doi.org/10.1007/s12065-009-0018-z. – ISSN 1864-5917

[28] Wieland, A.P.: Evolving neural network controllers for unstable systems. In: *IJCNN-91-Seattle International Joint Conference on Neural Networks* Bd. ii, 1991, S. 667–673 vol.2

[29] Wolpert, David H. ; Macready, William G.: No free lunch theorems for optimization. In: *IEEE transactions on evolutionary computation* 1 (1997), Nr. 1, S. 67–82. – URL https://ieeexplore.ieee.org/abstract/document/585893

[30] Wolpert, David H. ; Macready, William G. u. a.: No free lunch theorems for search / Citeseer. Curran Associates, Inc., 1995. – Forschungsbericht. – URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bfd288d7adb887ea0c91d14131f4cb78d675f0a4

[31] Zelinka, Ivan ; Senkerik, Roman ; Pluhacek, Michal: Do evolutionary algorithms indeed require randomness? In: *2013 IEEE Congress on Evolutionary Computation*, 2013, S. 2283–2289

A Anhang

A.1 NEAT-Konfigurationen

Parameter	k-XOR	Spiraldaten	Pole-Balancing
generations	400	300	300
fitness_criterion	max	max	max
fitness_threshold	3.42	3.45	57.0
pop_size	300	150	300
reset_on_extinction	False	False	False
num_hidden	0	6	2
num_inputs	6	2	6
num_outputs	1	1	1
feed_forward	True	True	True
initial_connection	full	full_direct	full
activation_default	tanh	tanh	sigmoid
activation_mutate_rate	0.1	0.1	0.1
activation_options	sigmoid tanh relu	sigmoid tanh relu	sigmoid tanh relu
aggregation_default	sum	sum	sum
aggregation_mutate_rate	0.0	0.0	0.0
aggregation_options	sum	sum	sum
bias_init_mean	0.0	0.0	0.0
bias_init_stdev	1.0	1.0	1.0
bias_max_value	30.0	30.0	30.0
bias_min_value	-30.0	-30.0	-30.0
bias_mutate_power	0.5	1.0	1.0
bias_mutate_rate	0.75	0.8	0.8
bias_replace_rate	0.1	0.1	0.1
conn_add_prob	0.2	0.2	0.5

conn_delete_prob	0.15	0.15	0.5
enabled_default	True	True	True
enabled_mutate_rate	0.055	0.05	0.05
node_add_prob	0.2	0.2	0.4
node_delete_prob	0.15	0.15	0.2
response_init_mean	1.0	1.0	1.0
response_init_stdev	0.0	0.0	0.0
response_max_value	30.0	30.0	30.0
response_min_value	-30.0	-30.0	-30.0
response_mutate_power	0.0	0.1	0.1
response_mutate_rate	0.0	0.1	0.1
response_replace_rate	0.0	0.0	0.0
weight_init_mean	0.0	0.0	0.0
weight_init_stdev	1.0	1.0	1.0
weight_max_value	30.0	30.0	30.0
weight_min_value	-30.0	-30.0	-30.0
weight_mutate_power	0.4	1.0	1.0
weight_mutate_rate	0.3	0.8	0.8
weight_replace_rate	0.1	0.1	0.1
compatibility_disjoint_coefficient	1.0	1.0	1.0
compatibility_weight_coefficient	0.5	0.5	0.5
compatibility_threshold	3.0	2.5	2.5
species_fitness_func	max	max	max
max_stagnation	25	15	15
species_elitism	3	2	2
elitism	2	5	5
survival_threshold	0.2	0.4	0.2

NEAT-Konfigurationen für die verschiedenen Experimente. Die Namengebung der Parameter basiert auf der Implementierung von *neat-python*.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original