Evolution neuronaler Netze mittels augmentierender Topologien: Eine Analyse des NEAT-Algorithmus

Niklas Viertel

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
an der Fakultät für Mathematik und Informatik
an der Fernuniversität in Hagen

Matrikel-Nr.: 3977315

Erstgutachter: Dr. Fabio Valdés

2. Dezember 2024

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund und Motivation

Künstliche Intelligenz (KI) hat sich in den letzten Jahrzehnten zu einem Schlüsselbereich der Forschung entwickelt. Eines der wichtigsten und leistungsfähigsten Werkzeuge in diesem Bereich sind neuronale Netze, die sich durch ihre Flexibilität und Leistungsfähigkeit auszeichnen. Traditionell wurden die Topologien dieser Netze manuell entworfen und mit klassischen Algorithmen wie Backpropagation trainiert. In vielen Fällen ist es aus menschlicher Sicht nahezu unmöglich, im Voraus eine optimale Topologie zu wählen, ganz zu schweigen von der Wahl der Verbindungen zwischen den einzelnen Schichten. Dies führt dazu, dass das Training neuronaler Netze häufig neu gestartet werden muss, um die Topologie anzupassen. Außerdem wird durch die überzähligen Knoten, Verbindungen und Schichten wertvolle Rechenzeit verschwendet. Hier setzt die Neuroevolution an, die mit genetischen Algorithmen und ihren biologisch inspirierten Ansätzen die Topologie und Parameter von Netzen parallel optimiert. Dies bringt jedoch eine Vielzahl bisher unbekannter Herausforderungen und Probleme mit sich.

Ein Algorithmus, der sich als Meilenstein der Neuroevolution erwiesen hat, ist der NEAT-Algorithmus (NeuroEvolution of Augmenting Topologies) von Kenneth Stanley und Risto Miikkulainen. Er bot eine Vielzahl von Lösungen für die vorherrschenden Probleme und ermöglichte wesentliche Fortschritte in der Neuroevolution. NEAT ist in der Lage, sowohl Netztopologien als auch deren Parameter parallel zu optimieren und dabei die Topologien zu minimieren. Damit hat sich NEAT als Meilenstein in der Neuroevolution etabliert und findet auch 20 Jahre nach seiner Veröffentlichung, insbesondere in Form seiner zahlreichen Weiterentwicklungen, eine Vielzahl von Anwendungen.

Das Hauptziel dieser Arbeit ist es, den NEAT-Algorithmus (NeuroEvolution of Augmenting Topologies) im Detail zu analysieren und seine Fähigkeiten zur Optimierung neuronaler Netze durch evolutionäre Prozesse darzustellen und zu bewerten. Um dies zu erreichen, beginnt die Arbeit mit kurzen Einführungen in genetische Algorithmen und neuronale Netze, welche die Grundlage für die später behandelte Neuroevolution bilden. Anschließend werden einige Algorithmen der Neuroevolution vorgestellt und ein Blick auf deren Kodierung geworfen, die eine der Hauptschwierigkeiten beim Entwurf von Neuroevolutionsalgorithmen darstellt.

Die Suche nach einer geeigneten Kodierung ist in der Neuroevolution umso schwieriger, da es in diesem Bereich viele weitere Herausforderungen gibt. Nach einer Diskussion der Grundlagen der Kodierungswahl werden die anderen in der Neuroevolution vorherrschenden Probleme vorgestellt und diskutiert. Anschließend wird der NEAT-Algorithmus in seiner grundlegenden Version vorgestellt und erläutert. Besonderes Augenmerk wird darauf gelegt, wie NEAT erstmals viele der zuvor diskutierten Probleme gleichzeitig auf kreative Weise löst. Im Folgenden wird die Leistungsfähigkeit von NEAT anhand praktischer Projekte demonstriert. In diesen Projekten wird untersucht, inwieweit die einzelnen Komponenten des NEAT-Algorithmus positiv zum Gesamtprodukt beitragen und wie die Leistungsfähigkeit des gesamten NEAT-Algorithmus im Vergleich zu anderen modernen

Algorithmen abschneidet. Zusätzlich werden Erweiterungen und weitere Anwendungsfälle von NEAT vorgestellt, um die Stärken und Schwächen von NEAT im Vergleich zu anderen Algorithmen innerhalb und außerhalb der Neuroevolution aufzuzeigen. Diese werden dann in der Zusammenfassung diskutiert.

2 Grundlagen

2.1 Genetische Algorithmen

Genetische Algorithmen (GA), die von John Holland in den 1960er Jahren entwickelt [1] und in seinem Buch „Adaptation in Natural and Artificial Systems“ von 1975 veröffentlicht wurden [Hol92], sind eine Klasse von Such- und Optimierungsverfahren, die auf den Mechanismen der natürlichen Selektion und der genetischen Vererbung basieren. Inspiriert von der Evolutionstheorie Charles Darwins, nutzen GAs eine Population von Individuen, die mögliche Lösungen für ein gegebenes Problem darstellen. Jede Lösung wird durch einen Satz von Parametern beschrieben, der als **Genotyp** bezeichnet wird. Der Genotyp ist eine Darstellung in Form einer Zeichenkette, die die verschiedenen Parameter einer möglichen Lösung kodiert. Diese Parameter, die als **Allele** bezeichnet werden, können binär, diskret, real oder in einem anderen durchsuchbaren Kodierungsformat sein. Ein Genotyp wird durch einen Geneseprozess in einen **Phänotyp** umgewandelt. Der Phänotyp stellt die tatsächliche Lösung des durch den Genotyp beschriebenen Problems dar. Dies kann ein bestimmtes Verhalten, eine bestimmte Struktur oder eine andere Form der Lösung sein, die sich aus dem Genotyp ergibt. Wenn es notwendig ist, sich auf eine bestimmte Position innerhalb des Genotyps zu beziehen, werden diese als **Locus** (Plural: **Loci**) bezeichnet. Durch die Anwendung von Operatoren wie Selektion, Kreuzung und Mutation wird die Population iterativ weiterentwickelt. Bessere Lösungen werden bevorzugt und die Ergebnisse von Generation zu Generation optimiert.

Goldberg [Gol89, S.1-25] beschreibt GAs als robuste Problemlösungsverfahren, die sich besonders für komplexe, nichtlineare und hochdimensionale Suchräume eignen, in denen herkömmliche Algorithmen oft an ihre Grenzen stoßen. Besonders wertvoll sind diese Algorithmen in Bereichen, in denen eine explizite mathematische Beschreibung des Problems schwierig zu formulieren ist oder die Lösungssuche durch lokale Optima behindert wird. Durch den Einsatz von Genetischen Algorithmen kann eine Balance zwischen Exploitation (Nutzung des bereits bekannten besten Wissens) und Exploration (Erkundung neuer Lösungen) erreicht werden, die zu einer effektiven und effizienten Optimierung führt.

Genetische Algorithmen folgen typischerweise einem iterativen Ablauf, der wie folgt strukturiert ist:

1. **Initialisierung**: Zu Beginn $t = 0$ wird eine Population N zufälliger Individuen erzeugt, dargestellt durch:

$$\mathbf{P}(t) = \{\mathbf{x}_1(t), \mathbf{x}_2(t), \ldots, \mathbf{x}_N(t)\}$$

[1] Genetische Algorithmen wurden zudem unabhängig von William Fraser und Heinz von Bremermann entwickelt [Fog06, S.75]

Die Variabel $t \in \mathbb{N}$ steht dabei für die betrachtete Generation. Jedes einzelne Individuum $\mathbf{x}_i(t) = (g_1, g_2, \ldots, g_l)$, $i \in \mathbb{N}_{\leq N}$ bestehend aus den vorher erwähnten Allelen stellt den Genotyp dar.

2. **Fitnessbewertung**: Jedes Individuum wird dann mit einer Fitnessfunktion bewertet, die angibt, wie gut ein Individuum das zugrunde liegende Problem löst:

$$\text{Fitness}(\mathbf{x}_i) = f(\mathbf{x}_i), \quad f : \mathbf{P}(t) \to \mathbb{R}$$

   Eine höhere Fitness steht für eine bessere Lösung.

3. **Selektion**: Aufgrund der Fitnesswerte werden Individuen als Eltern für die nächste Generation ausgewählt. Die Wahrscheinlichkeit $p(\mathbf{x}_i)$, dass ein Individuum x_i ausgewählt wird, hängt meist von seiner eigenen Fitness ab. Eine Möglichkeit für das Auswahlverfahren besteht darin, die Wahrscheinlichkeit, ausgewählt zu werden, proportional zur Fitness zu machen, d. h. je besser die Fitness, desto größer die Wahrscheinlichkeit, dass dieses Individuum ausgewählt wird.

$$p(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{N} f(\mathbf{x}_j)}$$

4. **Kreuzung (Crossover)**: In diesem Schritt werden ausgewählte Elternpaare $\mathbf{x}_p$ und $\mathbf{x}_q$ kombiniert [2], um einen Nachkommen $\mathbf{x}_c$ zu erzeugen. Eine Möglichkeit ist die **1-Punkt-Kreuzung** mit einem zufällig gewählten Kreuzungspunkt k, an dem sich die genetische Information der Eltern vermischt. Der Nachkomme wird wie folgt erzeugt

$$\mathbf{x}_c = (x_{p[1]}, \ldots, x_{p[k]}, x_{q[k+1]}, \ldots, x_{q[n]}).$$

   Hierbei werden die Gene des Elternteils $\mathbf{x}_p$ bis zum Punkt k übernommen, während die Gene von $\mathbf{x}_q$ ab dem Punkt $k+1$ bis zum Ende des Genotyps übernommen werden. Diese Methode erzeugt neue Lösungen, die Merkmale beider Eltern enthalten und somit die Vielfalt in der Population erhöhen.

5. **Mutation**: Bei jedem Nachkommen können einzelne Gene mit einer bestimmten Wahrscheinlichkeit P_{mut} zufällig verändert sein. Eine Möglichkeit ist das Hinzufügen eines kleinen Zufallswertes:

$$\mathbf{x}_c[i] = \mathbf{x}_c[i] + \epsilon, \quad \text{mit Wahrscheinlichkeit } P_{\text{mut}}$$

   Der Zufallswert ϵ wird üblicherweise mithilfe einer Normalverteilung $\mathcal{N}(0, \sigma)$ generiert. Diese Mutation bringt zusätzliche Variation in die Population und hilft, neue Bereiche des Suchraums zu erforschen.

[2] in vielen Algorithmen geschieht ebenfalls mit einer bestimmten Wahrscheinlichkeit P_{cross}, ansonsten wird eine Kopie des ersten Elternteil erstellt

6. **Reproduktion**: Nach der Anwendung von Selektion, Kreuzung und Mutation wird eine neue Population gebildet welche die vorherige ganz oder teilweise ersetzt:

$$\mathbf{P}(t+1) = \{\mathbf{x}_1(t+1), \mathbf{x}_2(t+1), \ldots, \mathbf{x}_N(t+1)\}$$

Dieser Prozess wird so lange wiederholt, bis ein Abbruchkriterium erreicht ist, z.B. eine maximale Anzahl von Generationen t_{max} oder eine zufriedenstellende Fitness:

$$f(\mathbf{x}_{\text{best}}) \geq f_{\text{target}}$$

Dieser Prozess zeigt, wie Genetische Algorithmen durch Iteration eine kontinuierliche Verbesserung der Population ermöglichen, um schließlich eine optimale oder zumindest sehr gute Lösung zu finden. Dies hat Holland [Hol92] mit seinem „Schema Theorem“ für eine binäre Kodierung gezeigt, welches dann von Wright [Wri91] auf reelle Zahlen übertragen wurde. Dies ist wichtig, da in NEAT die Verbindungsgewichte im Genotyp in reellen Zahlen kodiert werden.

Dazu wird ein einfaches Beispiel betrachtet.

Beispiel. *In diesem Beispiel besteht die Population aus 5-stelligen Bitstrings. Als Populationsgröße wurde hier die überschaubare Größe $N = 5$ gewählt, die bei richtiger Anwendung in der Regel deutlich größer ist. Die Aufgabe besteht darin, mit den Bitstrings eine möglichst hohe Zahl in dezimaler Form zu erhalten. Die Fitnessfunktion $f(\mathbf{x})$ liefert dann die entsprechende Dezimalzahl, die durch die Umwandlung der Bitstrings in Dezimalzahlen bestimmt wird.*
Es wird eine Anfangspopulation erzeugt und deren Fitness bewertet:

Individuum	**Fitness**	$P(\mathbf{x}_i)$ **= Prozentualer Anteil (%)**
10101	21	34%
01100	12	19%
11011	27	44%
00010	2	3%
Gesamtfitness	62	100%

Tabelle 1: 0-te Generation

Der nächste Schritt ist die Auswahl der Eltern für die nächste Generation. Dabei ist die Wahrscheinlichkeit, als Elternteil ausgewählt zu werden, proportional zur Fitness des betreffenden Individuums. Je höher die Fitness, desto höher ist die Wahrscheinlichkeit, als Elternteil ausgewählt zu werden. Dazu werden vier Zufallszahlen zwischen 0 und 100 erzeugt, die darüber entscheiden, welches Individuum als Elternteil ausgewählt wird. Die Zufallszahlen werden mit den kumulierten Wahrscheinlichkeiten verglichen, um die Auswahl zu treffen. Beispielsweise sind alle Zufallszahlen im Bereich $[0, 34]$ korrespondieren zum ersten Individuen, während alle Zufallszahlen im Bereich $[35, 54]$ zum zweiten Individuen gehören würden.

Zufallszahl 26 → Individuum 10101 (Fitnessanteil: 34%)
Zufallszahl 56 → Individuum 11011 (Fitnessanteil: 44%)
Zufallszahl 76 → Individuum 11011 (Fitnessanteil: 44%)
Zufallszahl 20 → Individuum 01100 (Fitnessanteil: 19%)

Dadurch werden fittere Individuen bevorzugt, aber auch weniger fitte Individuen haben eine Chance, selektiert zu werden, wodurch die genetische Vielfalt erhalten bleibt.

Im Folgenden wird eine Kreuzung zwischen den neuen Eltern durchgeführt. Für jedes Individuum wird zufällig ein anderes Individuum aus der Elternpopulation ausgewählt, und ein zufälliger Locus l_i wird bestimmt, an dem die Kreuzung stattfinden soll. Die mögliche Loci sind anhand des ersten Individuum dargestellt.

$$l_1\ 1\ l_2\ 0\ l_3\ 1\ l_4\ 0\ l_5\ 1\ l_6$$

Danach werden alle Bits vor diesem Locus vom ersten Elternteil und alle Bits nach diesem Locus vom zweiten Elternteil übernommen.

Elternteil 1	**Zufall Elt.**	**Elternteil 2**	**Zufall Stelle**	**Kreuzung**
10101	2	11011	4	101 | 11
11011	4	01100	5	1101 | 0
11011	2	11011	2	1 | 1011
01100	3	10101	4	011| 01

Tabelle 2: Kreuzung-Operator

Der nächste Schritt ist die Mutation. Dabei hat jedes Individuum eine Chance von $P_{mut} = 20\%$, ein zufällig ausgewähltes Bit zu mutieren. Die Mutationsrate ist wichtig, um eine Balance zwischen der Erhaltung der genetischen Vielfalt in der Population und der Vermeidung der Zerstörung bereits guter Lösungen zu finden, sodass der Algorithmus sowohl die Entdeckung neuer potentieller Lösungen fördert als auch das Risiko des Verwerfens vielversprechender Lösungen minimiert. Da es für jede Mutation nur eine Möglichkeit gibt, wird das zu mutierende Bit geflippt.

$$x_c[i] = \begin{cases} 1 - x_c[i] & \textit{mit Wahrscheinlichkeit } P_{mut} \\ x_c[i] & \textit{mit Wahrscheinlichkeit } 1 - P_{mut} \end{cases}$$

Die Loci sind hier die Position der Allelen.

Individuum	**Zufallswert**	**Bedingung**	**Locus**	**Mutation**
10111	5	False		10111
11010	3	False		11010
11011	2	False		11011
01101	1	**True**	4	01111

Tabelle 3: Mutation-Schritt

Anschließend wird die neue Generation erzeugt.

Individuum	Fitness	Prozentualer Anteil (%)
10111	23	36.51%
11010	26	41.94%
11011	27	43.55%
01111	15	24.19%
Gesamtfitness	91	100%

Tabelle 4: 1-te Generation

Bereits nach einer Generation ist oft eine deutliche Verbesserung der Fitness zu erkennen. Das Beispiel veranschaulicht somit den kontinuierlichen Prozess, in dem genetische Algorithmen über mehrere Generationen immer bessere Lösungen finden.

2.2 Künstliche Neuronale Netze

Künstliche Neuronale Netze (KNN) sind Rechenmodelle, die von der Struktur und Funktion biologischer Nervensysteme inspiriert sind. Wie Simon Haykin in seinem umfassenden Werk *Neural Networks: A Comprehensive Foundation* [Hay98] beschrieben hat, bestehen diese Netze aus einer Vielzahl miteinander verbundener Einheiten, den so genannten **Neuronen**. Diese Neuronen werden in der Regel in Schichten angeordnet. Die Hauptidee hinter neuronalen Netzen ist, dass sie durch das Lernen von Mustern in den Daten komplexe Aufgaben wie Klassifikation, Mustererkennung und Vorhersage bewältigen können. Die Leistungsfähigkeit von KNNs zeigt sich somit in ihrer Fähigkeit, beliebige stetige Funktionen zu approximieren, wie Cybenko [Cyb89] gezeigt hat.

Die erste Schicht, die Eingabeschicht, besteht aus Neuronen, die als „Sensoren“ fungieren und Informationen aus der Außenwelt aufnehmen. Diese Eingangssignale werden dann über gewichtete Verbindungen an andere Neuronen weitergeleitet. Die letzte Schicht eines neuronalen Netzes ist die Ausgabeschicht, deren Ergebnisse von dem System verwendet werden, das das neuronale Netz enthält. Zwischen der Eingabe- und der Ausgabeschicht liegen normalerweise die so genannten „verborgenen Schichten“, die aus Neuronen bestehen und für die Weiterverarbeitung der eingehenden Informationen zuständig sind. Bei anderen Arten von Netzen gibt es keine klare Struktur der inneren Neuronen; diese Netze werden als **Nicht-Standard**-Netze bezeichnet, während die zuvor genannten Netze als **Standard**-Netze bezeichnet werden (siehe Abbildung 1).

Mathematisch kann ein KNN als Graph $G = (V, E)$ aufgefasst werden, wobei V eine Menge von Knoten (Neuronen) und E eine Menge von Kanten (Verbindungen) darstellt. Dieser Graph beschreibt eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$, die den Prozess der Transformation von Eingaben in Ausgaben darstellt, wobei $\mathbb{R}^n$ den Eingaberaum und $\mathbb{R}^m$ den Ausgaberaum repräsentiert. Jedes Neuron j in einem neuronalen Netz berechnet einen Ausgabewert y_j. Dieser Ausgabewert ist typischerweise eine gewichtete Summe seiner Eingaben und wird somit nach folgender Formel berechnet:

$$y_j = \sigma \left(\sum_{k=1}^{n} w_{jk} x_k + b_j \right)$$

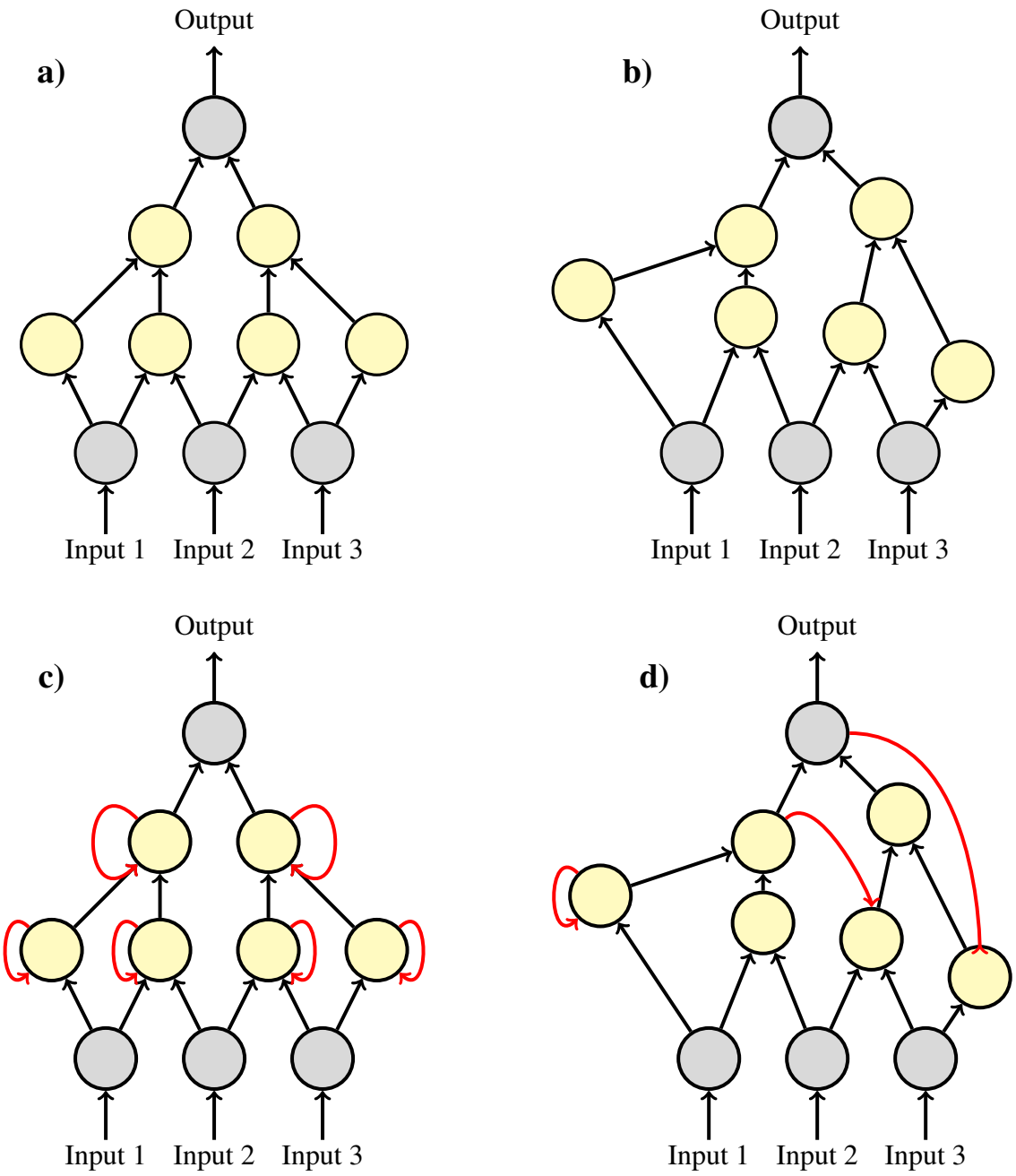


Abbildung 1: 4. Arten von KNN: a) Feedforward-Standard , b) Feedforward-nicht-Standard , c) Rekurrent-Standart, d) Rekurrent-nicht-Standard

Diese Summe wird dann durch eine **Aktivierungsfunktion**, häufig als σ bezeichnet, geleitet, die den Aktivierungswert in einen bestimmten Bereich, typischerweise zwischen 0 und 1, transformiert. wobei w_{jk} das Verbindungsgewicht zwischen dem Neuron j und dem Eingangsneuron k darstellt. Die Neuronen k sind alle Neuronen, die eine Verbindung zu dem betrachteten Neuron j haben. Zusätzlich gibt es für jedes Neuron einen Bias b_j, der die Ausgabe leicht verschiebt. Es gibt eine Reihe verschiedener Aktivierungsfunktionen, eine typische ist die nichtlineare Sigmoidfunktion:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad \sigma : \mathbb{R} \to [0, 1]$$

Weitere häufige Aktivierungsfunktionen sind die ReLU-Funktion (Rectified Linear Unit), definiert als:

$$\text{ReLU}(x) = \max(0, x), \quad \text{ReLU} : \mathbb{R} \to [0, \infty)$$

und die Tanh-Funktion (hyperbolischer Tangens), definiert als:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \tanh : \mathbb{R} \to [-1, 1]$$

Die Wahl der Aktivierungsfunktion hängt von verschiedenen Faktoren ab, wie der Art der Aufgabe, der Netzstruktur, dem Gradientenverhalten, der Trainingsgeschwindigkeit, den Dateneigenschaften, der Gefahr des Overfittings und den verfügbaren Rechenressourcen. So eignet sich die Sigmoid-Funktion aufgrund ihres Wertebereichs besonders gut für Klassifikationsprobleme, während die ReLU-Funktion ihre Stärke bei tiefen Netzen zeigt, unter anderem weil sie sehr recheneffizient ist. Die tanh-Funktion hingegen normiert die Ausgabe auf -1 bis 1 und zentriert die Werte um Null, was sie ideal für symmetrische Datenverteilungen und ein stabileres Training in flachen Netzen macht.

Ein Netz, bei dem die Aktivierung von den Eingängen zu den Ausgängen erfolgt, wird als **Feed-Forward-Netz** bezeichnet. Es gibt jedoch auch Netze mit Rückkopplung, bei denen die Signale nicht nur vorwärts, sondern auch rückwärts fließen. Solche Netze werden als **rekurrente Netze** bezeichnet (siehe Abbildung 1). Rekurrente Netze sind besonders nützlich, um zeitliche Abhängigkeiten zu lernen, wie es Ring [Rin94] beschreibt. Diese Netze können Informationen aus früheren Zeitschritten berücksichtigen und so Muster erkennen, die sich im Laufe der Zeit entwickeln. Allerdings ist das Training von rekurrenten neuronalen Netzen (RNNs) oft langsamer und weniger zuverlässig als das von Feedforward-Netzen, insbesondere aufgrund von Problemen wie dem Vanishing Gradient Problem [3], das die Lernfähigkeit von RNNs bei langen Abhängigkeiten einschränkt [BSF94].

Neuronale Netze können mit einer Vielzahl unterschiedlicher Trainingsregeln trainiert werden, z.B. mit gradientenbasierten Verfahren. Ein sehr bekanntes gradientenbasiertes Verfahren ist die Rückpropagation nach Rumelhart et al. [RHW86]. Das Verfahren besteht aus zwei Schritten: dem Vorwärtslauf und dem Rückwärtslauf. Im Vorwärtslauf werden

[3] Das Vanishing Gradient Problem beschreibt das Phänomen, dass Gradienten in tiefen KNNs während der Backpropagation immer kleiner werden, so dass frühe Schichten kaum noch lernen können.

die Eingabewerte x durch das Netz mit der Gewichtsmatrix W und dem Bias b geschickt, um eine Ausgabe y zu berechnen:

$$y = \sigma(W \cdot x + b)$$

Anschließend wird der Verlust / Fehler berechnet, um die Leistung des Netzes zu bewerten. Der Verlust wird typischerweise als Differenz zwischen der tatsächlichen Ausgabe y_{true} und der vorhergesagten Ausgabe y berechnet. Eine gebräuchliche Verlustfunktion ist die mittlere quadratische Fehlerfunktion, definiert als:

$$\mathcal{L} = \frac{1}{2}(y_{\text{true}} - y)^2$$

Rückwärts wird die Steigung der Verlustfunktion als Funktion der Gewichte berechnet. Dies geschieht durch Anwendung der Kettenregel der Differentialrechnung. Die Steigung in Abhängigkeit der Gewichte W für die Verlustfunktion L ergibt sich zu:

$$\frac{\partial \mathcal{L}}{\partial W} = -(y_{\text{true}} - y) \cdot \frac{\partial y}{\partial W}$$

Die Gewichte werden dann aktualisiert, um den Fehler zu minimieren, indem sie in Richtung des negativen Gradienten angepasst werden:

$$W_{\text{neu}} = W - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$$

wobei η die Lernrate ist, die die Schrittweite bei der Aktualisierung der Gewichte bestimmt. Dieser Vorgang wird so lange wiederholt, bis die gewünschte Netzleistung oder Trainingszeit erreicht ist. Gradientenbasierte Methoden bringen jedoch auch einige Nachteile mit sich. Zum einen laufen diese der Gefahr, in lokalen Minima stecken zu bleiben, was die Suche nach globalen Optima erschwert. Darüber hinaus sind solche Methoden nicht immer ideal, da (1) oft kein unmittelbares Feedback für jede Iteration der Ausgabe verfügbar ist und (2) die Rückpropagation Zielwerte benötigt, die nicht immer verfügbar sind [Sut18, S.15-16].

Methoden wie die Backpropagation werden auch oft kritisiert, weil sie nicht den natürlichen Lernprinzipien in biologischen Systemen entsprechen, was problematisch ist, da es ihre Fähigkeit einschränkt, die Funktionsweise des menschlichen Gehirns (die Idee von KNNs) genau zu imitieren und somit die Effizienz und Robustheit des Lernens beeinträchtigt [BLB$^+$15]. Ein weiterer wesentlicher Unterschied besteht darin, dass in einem statischen KNN die Gewichte der Verbindungen zwischen den Neuronen während der Laufzeit des Netzes konstant bleiben. Das bedeutet, dass sich die Stärke dieser Verbindungen nach dem anfänglichen Training des Netzes nicht mehr ändert. Diese statische Natur ermöglicht eine stabile und vorhersagbare Leistung des Netzes, schränkt jedoch seine Fähigkeit ein, sich an neue oder sich verändernde Umgebungen anzupassen.

Im Gegensatz dazu zeigen biologische Nervensysteme eine bemerkenswerte Flexibilität bei der Anpassung ihrer Verbindungen. Im lebenden Gehirn sind die Verbindungen zwischen den Nervenzellen ständigen Veränderungen unterworfen, ein Prozess, der als synaptische Plastizität bezeichnet wird. Diese dynamische Anpassungsfähigkeit ermöglicht es dem Gehirn, ständig zu lernen und sich an neue Informationen oder veränderte

Bedingungen anzupassen. Forscher wie Floreano und Urzelai [FU00] haben gezeigt, dass adaptive Netze, die diese Art der synaptischen Plastizität nachahmen, weit über einfache Topologie- und Gewichtsanpassungen hinausgehen und auch Regeln zur ständigen Anpassung der Verbindungen enthalten.
Ein Beispiel für eine Trainingsregel welche versucht das Prinzip der synaptischen Plastizität in adaptiven Netzen nachzuahmen ist die Hebb'sche Regel:

$$\Delta w = \eta(1 - w)xy$$

wobei x die Aktivität des eingehenden Neurons, y die Aktivität des ausgehenden Neurons und η die Lernrate ist. Die Hebb'sche Regel besagt, dass Verbindungen zwischen Neuronen, die gleichzeitig aktiv sind, verstärkt werden. Das bedeutet, dass Verbindungen, die häufiger zusammen feuern, verstärkt werden, um die Lern- und Anpassungsfähigkeit des Netzes zu verbessern. Ein neuronales Netz, das aus einer Vielzahl von Hebbian-Synapsen mit unterschiedlichen Lernraten besteht, kann flexibel auf Veränderungen in der Umwelt reagieren. Neben den Hebbian-Synapsen können auch andere Anpassungsregeln auf bestimmte Synapsen angewendet werden, während einige Verbindungen als statische Feedforward-Gewichte erhalten bleiben.

Seit der Veröffentlichung von NEAT haben sich zahlreiche alternative Ansätze entwickelt, die viele der Schwächen der Rückpropagation adressieren. Ein alternativer Ansatz ist die sogenannte **Neuroevolution** (NE), die auf evolutionären Prinzipien zur Optimierung neuronaler Netze basiert.

2.3 Neuroevolution und ihre Kodierung

Neuroevolution, der Prozess der Evolution neuronaler Netze mittels genetischer Algorithmen, hat sich als leistungsfähige Methode zur Entwicklung und Optimierung komplexer neuronaler Architekturen erwiesen [FU00]. Diese Technik wird auch heute noch intensiv genutzt, um sowohl die Struktur als auch die Gewichtung von Netzen durch evolutionäre Prozesse anzupassen und effiziente Lösungen für eine Vielzahl von Aufgaben zu finden [SMC+17]. Trotz ihrer Erfolge sieht sich die Neuroevolution mit einer Reihe von Herausforderungen konfrontiert, insbesondere bei der Auswahl geeigneter Kodierungen, dem Umgang mit dem „Competing Conventions"-Problem während des Kreuzung-Prozesses und der Vermeidung des vorzeitigen Verlusts von topologischen Innovationen. In den folgenden Abschnitten werden diese Herausforderungen näher beleuchtet und Strategien zur effektiven Initialisierung der Ausgangspopulation und zur Bewältigung dieser Probleme vorgestellt.

Eine zentrale Entscheidung, die bei allen **TWEANNs** (Topologies and Weight Evolving Artificial Neural Networks, Netze mit topologie- und gewichtsbasierter Evolution) getroffen werden muss, ist die Wahl einer geeigneten Kodierung der Netze. Grundsätzlich lassen sich TWEANNs in zwei Kategorien einteilen: solche, die eine **direkte Kodierung** verwenden und solche, die eine **indirekte Kodierung** verwenden.

2.3.1 Direkte Kodierungen

Direkte Kodierungsschemata, die von den meisten TWEANNs verwendet werden, spezifizieren explizit im Genotyp jede Verbindung und jeden Knoten, die im Phänotyp erscheinen sollen [SM02]. Diese Methode führt zu einer genauen und detaillierten Darstellung des Netzes, die eine klare Beziehung zwischen Kodierung und Netzleistung ermöglicht. In großen Systemen kann dies jedoch zu einem sehr großen Genotyp führen, was die Effizienz und Skalierbarkeit beeinträchtigen kann. Ein gutes Kodierungsschema sollte daher die Eigenschaften Eindeutigkeit, Einfachheit, Effizienz, Flexibilität und Rekombinierbarkeit erfüllen.

Eine einfache Form der direkten Kodierung ist die binäre Kodierung, wie sie z.B. von Dasgupta und McGregor in ihrem Structured Genetic Algorithm (sGA) verwendet wird [DM92]. Diese Kodierung verwendet die traditionelle Bitstring-Darstellung, bei der ein Bitstring die Verbindungsmatrix eines Netzes darstellt. Aufgrund seiner Einfachheit bietet der sGA fast die gleiche Funktionalität wie traditionelle genetische Algorithmen. Binäre Daten können dafür effizient verarbeitet werden, da digitale Systeme gut für binäre Operationen geeignet sind, was die Geschwindigkeit und Effizienz genetischer Algorithmen zur Evolution des Netzes verbessern kann [Gol89].

Diese Einfachheit bringt jedoch auch einige Einschränkungen mit sich. Erstens ist die Größe der Verbindungsmatrix das Quadrat der Anzahl der Knoten, was bei vielen Knoten zu einer erheblichen Vergrößerung der Bitfolge führt. Zweitens muss die Bitfolge für alle Individuen gleich groß sein, was bedeutet, dass die maximale Anzahl der Knoten (und damit der Verbindungen) im Voraus festgelegt werden muss. Diese Größe muss sorgfältig gewählt werden, da eine Überschreitung zu einer Neuberechnung des Algorithmus führt. Drittens kann die lineare Bitfolge, die zur Darstellung der Graphenstruktur verwendet wird, es schwierig machen, sicherzustellen, dass der Kreuzung-Operation sinnvolle und funktionale Kombinationen erzeugt.

Die Bewältigung dieser Herausforderungen erfordert eine Verlängerung der Genotypen, was zu den anderen oben genannten Nachteilen führt.

Evolution ohne Kreuzung

Die vorangegangenen Diskussion zur direkten Kodierung zeigt, wie schwierig es ist, den Kreuzungsoperator für Netze mit unterschiedlichen Topologien sinnvoll zu implementieren, um seine Existenz zu rechtfertigen. Die Binärkodierung weist beispielsweise erhebliche Probleme hinsichtlich der Flexibilität auf, da sie auf Kosten der Recheneffizienz geht. Umgekehrt kann der Kreuzung-Operator bei unterschiedlichen Topologien oft zu Funktionsverlusten führen kann, weshalb einige Forscher den Kreuzung-Operator ganz aufgegeben haben, was im weiteren Sinne als Evolutionäre Programmierung ([YL96]) bezeichnet wird. Ein Beispiel hierfür ist die Arbeit von Angeline et al. [ASP93], die ein System namens GeNeralized Acquisition of Recurrent Links (GNARL) entwickelten. Die Autoren kommentierten, dass „die Aussichten für die Entwicklung von Verknüpfungssystemen mit Kreuzung im Allgemeinen begrenzt zu sein scheinen“. GNARL verzichet dementsprechend vollständig auf den Kreuzung-Operator. GNARL zeigt, dass ein TWEANN nicht

auf Kreuzung angewiesen ist, um erfolgreich zu sein, und überlässt es anderen Methoden, die Vorteile vom Kreuzung-Operator aufzuzeigen.

In gewisser Weise hat die Zukunft Angeline et al. Recht gegeben, wie ein kurzer Blick auf die Größe der Evolutionären Programmierung zeigt. Dies wird zudem Beispiel deutlich an der Performanz von Algorithmen wie „Evolutionary acquisition of neural topologies“ [KS05]. Gleichzeitig haben Methoden wie NEAT jedoch gezeigt, dass der Kreuzung-Operator seine Existenzberechtigung mehr als gerechtfertigt hat, wie später zu sehen sein wird.

2.3.2 Indirekte Kodierung

Im Gegensatz zu einer direkten Kodierung spezifizieren indirekte Kodierungen in der Regel nur Regeln für die Strukturierung des Phänotyps [SM02]. Solche Regeln könnten z.B. Schichtspezifikationen oder Wachstumsregeln durch Zellteilung umfassen, wobei letztere die dynamische Erzeugung neuer Knoten und Verbindungen im Netz durch einen biologisch inspirierten Teilungsprozess beschreiben. Indirekte Kodierungen ermöglichen eine kompaktere Repräsentation, da nicht jede einzelne Verbindung und jeder Knoten im Genotyp explizit definiert werden muss, sondern stattdessen aus den definierten Regeln abgeleitet werden kann. Dies führt insbesondere bei komplexen Netzen zu einer effizienteren Darstellung.

Entwicklungskodierung

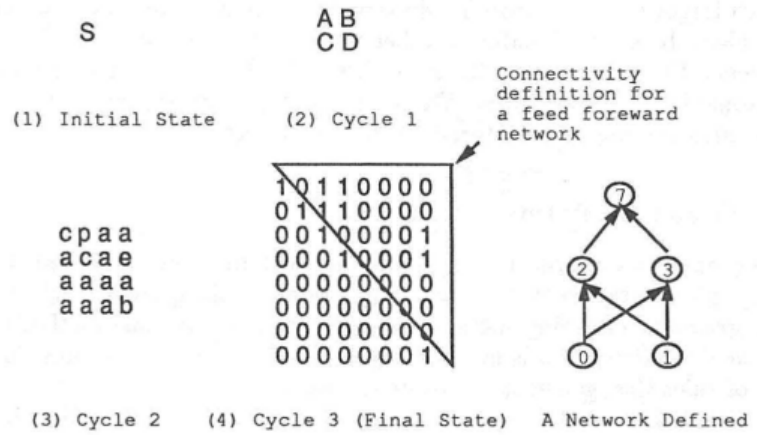
Ein einfaches Beispiel für eine indirekte Kodierung wird von Kitano [Kit90] verwendet. Er verwendet eine Entwicklungsregel zur Konstruktion der Netzstruktur. Bei diesem Ansatz wird der Genotyp in Blöcke von fünf Elementen unterteilt, wobei jeder Block eine Regel darstellt, die definiert, wie die Symbole in eine Matrix umgewandelt werden. Terminalsymbole entwickeln sich zu vordefinierten 2x2-Matrizen aus 0s und 1s, während Nichtterminalsymbole weitere Symbole erzeugen, die rekursiv entwickelt werden. Dieser Entwicklungsprozess führt zu einer finalen Matrix, die die Netzarchitektur und die Verbindungsstruktur repräsentiert. Kitano zeigte, dass diese Methode oft bessere Ergebnisse liefert als die direkte Kodierung, da sie komplexe Netze aus kompakten genetischen Repräsentationen erzeugt. Es wurde jedoch festgestellt, dass die Leistungseinbußen bei der direkten Kodierung eher auf Unterschiede in den Ausgangspopulationen zurückzuführen sind. Dies wird in einem späteren Kapitel näher untersucht. Die Entwicklungskodierung ermöglicht somit einen flexiblen und effizienten Aufbau von Netzen, hat jedoch den Nachteil, dass die Entwicklungsregeln komplex sein können und möglicherweise zusätzliche Mechanismen zur Feinabstimmung asymmetrischer Architekturen und spezifischer Parameter erfordern.

Beispiel. *Es wird der beispielhafte Ablauf der Erstellung eines Netzes mit der von Kitano vorgeschlagenen Kodierung betrachtet. Die genauen Entwicklungsregeln sind immer problemspezifisch, hier für das XOR-Problem. Die genauere Erläuterung des Problems ist hier nicht relevant. Beide Abbildungen sind [Kit90] entnommen. Zunächst werden die Entwicklungsregeln gezeigt:*

$$S \rightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad A \rightarrow \begin{bmatrix} c & p \\ a & c \end{bmatrix} \quad B \rightarrow \begin{bmatrix} a & a \\ a & e \end{bmatrix} \quad C \rightarrow \begin{bmatrix} a & a \\ a & a \end{bmatrix} \quad D \rightarrow \begin{bmatrix} a & a \\ a & d \end{bmatrix}$$

$$a \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \quad c \rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad e \rightarrow \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \quad p \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Mit Hilfe dieser Regeln ist es nun möglich, ein neuronales Netz zu erstellen:



Die von Kitano verwendete Entwicklungskodierung ist heute sehr veraltet, demonstriert aber dennoch gut die grundlegende Funktionsweise der indirekten Kodierung.

2.3.3 Abschließende Bemerkung zu Koderierungswahl

Während die direkte Kodierung eine detaillierte und genaue Darstellung der Netze gewährleistet, bietet die indirekte Kodierung den Vorteil, die Komplexität des Genotypes zu reduzieren, was insbesondere bei der Modellierung großer und komplexer Netze von Vorteil ist. Die direkte Kodierung ist vorteilhaft für kleinere Netze und wenn eine genaue Kontrolle der Struktur erforderlich ist. Die indirekte Kodierung eignet sich besser für große und komplexe Netze, bei denen Skalierbarkeit und Effizienz im Vordergrund stehen. Der Preis für die kompaktere Darstellung und bessere Skalierbarkeit indirekter Kodierungen ist jedoch eine erhöhte Komplexität und geringere Vorhersagbarkeit der resultierenden Netz wie Hornby in [HP04] beschreibt.

Basierend darauf entschieden sich Stanley und Miikkulainen bei der Entwicklung des NEAT-Algorithmus für eine direkte Kodierung [SM02]. Der Hauptgrund für diese Entscheidung ist die Tatsache, dass indirekte Kodierungen, wie sie in Methoden wie CE verwendet werden, ein tieferes Verständnis der genetischen und neuronalen Mechanismen erfordern [BW93]. Da indirekte Kodierungen nicht direkt mit den resultierenden Netzstrukturen übereinstimmen, besteht die Gefahr, dass die Suche nach optimalen Topologien in unvorhersehbarer Weise verzerrt wird. Um indirekte Kodierungen effektiv nutzen zu

können, müsste sichergestellt werden, dass sie die Suche nicht auf suboptimale Strukturen lenken [SM02].

In ihrem späteren HyperNEAT-Algorithmus (Stanley, K. O., D'Ambrosio, D. B. und Gauci, J., 2009), der als Erweiterung des ursprünglichen NEAT-Algorithmus angesehen werden kann, entschieden sich die Autoren jedoch für eine indirekte Kodierung in Form von Compositional Pattern-Producing Networks (CPPNs) [SDG09]. Diese Entscheidung ermöglichte die effiziente Entwicklung komplexer und großer neuronaler Netze, indem die Topologie des Netzes durch die Erzeugung wiederkehrender Muster und symmetrischer Strukturen im Verbindungsraum definiert wurde. Diese Art der Kodierung erweist sich als besonders vorteilhaft für Aufgaben, die eine regelmäßige und strukturierte Netzanordnung erfordern. Wie im späteren Kapitel über HyperNEAT noch genauer untersucht wird, eröffnet diese Methode neue Möglichkeiten für die Evolution von Netzen in komplexen Domänen, in denen traditionelle direkte Kodierungen an ihre Grenzen stoßen.

2.4 Weitere Probleme in Neuroevolution

2.4.1 Competing Conventions-Problem

Das „Competing Conventions"-Problem (auch Permutationsproblem genannt) ist ein zentrales Problem der Neuroevolution und genetischer Algorithmen, das erstmals Ende der 1980er Jahre erkannt wurde. Es beschreibt die Schwierigkeit, dass unterschiedliche genetische Repräsentationen den gleichen phänotypischen Output erzeugen können, was die Effizienz der Kreuzung-Operation beeinträchtigt. Besonders betroffen sind symmetrische Strukturen wie neuronale Netze, in denen Neuronen funktionell identisch, aber genetisch unterschiedlich angeordnet sein können. Dieses Problem wurde erstmals in der Arbeit von Whitley et al. [WSB90] analysiert und als wichtiger Faktor für den Fortschritt genetischer Algorithmen erkannt.

Eine wichtige praktische Konsequenz des „Competing Conventions"-Problems ist der potentielle Informationsverlust während des Kreuzungsprozesses. Wenn funktionell identische, aber genetisch unterschiedliche Lösungen kombiniert werden, besteht die Gefahr, dass bei der Kreuzung wichtige Informationen verloren gehen. Beispielsweise können verschiedene Permutationen von versteckten Neuronen in einem neuronalen Netz äquivalente Lösungen darstellen. Bei der Kreuzung dieser Netze kann eine falsche Anordnung der Neuronen jedoch zu Performanceverlusten führen, da wesentliche Teile der Lösung nicht miteinander kompatibel sind. Dies kann dazu führen, dass bei der Kreuzung von Elternstrukturen häufig inkompatible Nachkommen entstehen, die eine schlechtere Leistung erbringen als ihre Eltern.

Peter Radcliffe (1993) hat dieses Problem in seiner Arbeit „Genetic Set Recombination and its Application to Neural Network Topology Optimisation" weiter vertieft und als Kernproblem der genetischen Kreuzung beschrieben [Rad93]. Er wies darauf hin, dass genetische Algorithmen häufig versagen, wenn sie funktional identische, aber genetisch unterschiedliche Teile von Lösungen kreuzen. Diese Inkompatibilität führe häufig zur Zerstörung nützlicher Strukturen und behindere damit den Fortschritt des evolutionären

Prozesses.

Beispiel. *Ein einfaches Beispiel [Sta04] verdeutlicht dieses Problem: Betrachtet wird ein einfaches neuronales Netz mit drei versteckten Neuronen A, B und C. Die Gewichte, die die Verbindungen zwischen diesen Neuronen und den Eingängen sowie Ausgängen darstellen, können in unterschiedlichen Reihenfolgen angeordnet werden, ohne dass sich die Funktion des Netz ändert.*

$$[A, B, C], [A, C, B], [B, A, C], [B, C, A], [C, A, B], [C, B, A],$$

Diese unterschiedlichen Anordnungen repräsentieren die gleiche Lösung, aber ihre verschiedenen Kodierungen sind nicht immer vereinbar. Wenn man zwei solche Kodierungen kreuzt, kann es leicht zu einer Beschädigung der Information kommen, wie etwa dem Verlust von Informationen und Verbindungen. Zum Beispiel kann die Kreuzung von $[A, B, C]$ *und* $[C, B, A]$ *zu* $[C, B, C]$ *führen, einer Darstellung, die ein Drittel der Informationen verloren hat, die beide Elternteile hatten.*

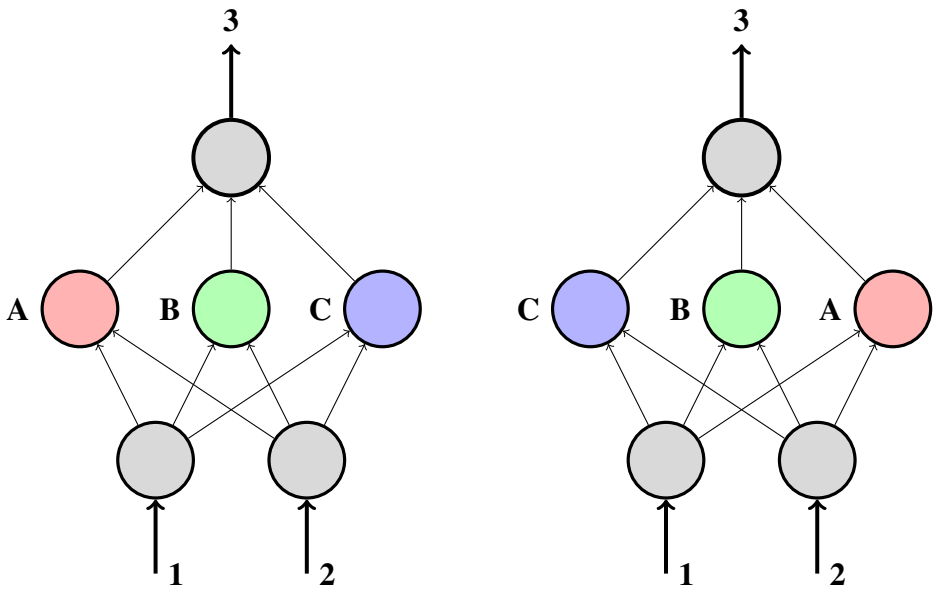


Abbildung 2: „Competing Convention“ Problem

Um die Ernsthaftigkeit des Problems zu verdeutlichen, wird das Beispiel näher betrachtet. Es gibt $3! = 6$ *Permutationen der drei versteckten Knoten, die jeweils die gleiche Lösung darstellen. Alle möglichen Kreuzungen zwischen diesen 6 Permutationen werden gezählt:*

- *Da jede Permutation mit jeder anderen Permutation (einschließlich sich selbst) kreuzen kann, gibt es* $6 \times 6 = 36$ *mögliche Paarungen von permutierten Strukturen, die konkurrierende Konventionen darstellen.*
- *Bei einer Darstellung mit 3 Knoten gibt es 4 mögliche Kreuz-Punkte bei der üblichen Ein-Punkt-Kreuzung:* $[1A2B3C4]$.
- *Da es 36 mögliche Paarungen gibt und jede an 4 verschiedenen Punkten kreuzen kann, gibt es insgesamt* $36 \times 4 = 144$ *mögliche Kreuz-Produkte (Nachkommen).*

Wenn jedes dieser 144 möglichen Produkte aufgezählt wird, zeigt sich, dass 48 von ihnen einen starken Verlust an genetischer Information und Redundanz aufweisen. Mit anderen Worten: 48 von 144 möglichen Nachkommen fehlen einer der versteckten Knoten A, B *oder* C.
Zusätzlich sind 72 der 144 Produkte trivial, das heißt, der Nachkomme ist ein Duplikat eines der beiden Elternteile, weil der Kreuz-Punkt am Ende des Genotypes liegt. Wenn nur die nicht-trivialen Kreuzungen betrachtet werden, bei denen die Nachkommen nicht einfach ein Duplikat eines Elternteils sind, beträgt die Wahrscheinlichkeit, dass ein Nachkomme einen schweren genetischen Schaden aufweist, $\frac{48}{72} = 66{,}6\%$.

Diese hohe Wahrscheinlichkeit genetischer Schäden unterstreicht die Bedeutung robuster Mechanismen in evolutionären Algorithmen, um den Verlust wertvoller genetischer Information zu verhindern, unnötige Rechenzeit zu vermeiden und die erfolgreiche Evolution komplexer Strukturen sicherzustellen.

Dieses Problem wird bei TWEANNs noch verschärft, da die Netze oft sehr unterschiedliche und komplexe Topologien aufweisen, die nicht auf eine einheitliche Struktur beschränkt sind und sich oft sogar in der Größe des Genotypes unterscheiden.
Da TWEANNs keine strikten Restriktionen bezüglich der Art der erzeugten Topologien haben, sind Lösungsvorschläge zur Bewältigung des „Competing Convention"-Problem, die für Netze mit festen oder eingeschränkten Topologien entwickelt wurden, wie z.B. die nicht-redundante genetische Kodierung [Thi96], hier nicht anwendbar.

Radcliffe schlug vor, Mapping-Mechanismen zu entwickeln, um die genetischen Strukturen vor der Kreuzung besser aufeinander abzustimmen und so die negativen Auswirkungen des „Competing Conventions"-Problems zu minimieren; NEAT verfolgt einen ähnlichen Ansatz, indem es eine von der Natur inspirierte Methode zur Lösung dieses Problems einsetzt, welche später im Detail betrachtet wird.

2.4.2 Schutz von Innovationen durch Speziation

Das Ziel jedes Neuroevolution-Algorithmus ist es mehr oder weniger, einen möglichst hohen Wert mit der Fitness-Funktion zu erzielen. Allerdings kann dieser Fokus auf eine hohe Fitness häufig zu Problemen führen, da der Suchalgorithmus in lokalen Minima steckenbleiben oder täuschenden Fitness-Landschaften erliegen kann. Diese Probleme können dazu führen, dass potenziell bessere, aber weniger offensichtliche Lösungen übersehen werden. Eine offenere Herangehensweise, wie die Suche nach Neuartigkeit, kann oft effektiver sein, da sie nicht durch diese Einschränkungen beeinträchtigt wird [VL$^+$91].

In TWEANNs entsteht Innovation durch das Hinzufügen neuer Strukturen zu den Netzen mittels Mutationen. Diese neuen Strukturen können anfänglich die Fitness eines Netzes verringern, da sie möglicherweise zu Nichtlinearitäten führen oder die Gewichtsanpassungen noch nicht abgeschlossen sind. Beispielsweise kann das Hinzufügen eines neuen Knotens die Fitness vorübergehend reduzieren, bis sich die neuen Verbindungen und deren Gewichte optimieren. Da ein neuer Knoten oder eine neue Verbindung oft nicht sofort nützlich ist, sind häufig mehrere Generationen erforderlich, um diese Strukturen zu optimieren und sinnvoll zu integrieren. Daher besteht die Gefahr, dass Innovationen aufgrund ihres anfänglichen Fitnessverlustes nicht lange genug in der Population überleben, um weiterentwickelt zu werden. Um diesen Prozess zu unterstützen, ist es entscheidend, Netze mit strukturellen Innovationen zu schützen, um ihnen die notwendige Zeit zu geben, ihre neuen Strukturen zu entwickeln und ihre Leistungsfähigkeit zu entfalten [SM02].

In unserem zuvor betrachteten GNARL-Algorithmus wurden neue Innovationen durch das Hinzufügen einer nicht-funktionalen Struktur geschützt. Ein Knoten wird zu einem Genotyp ohne Verbindungen hinzugefügt, in der Hoffnung, dass sich in der Zukunft nützliche Verbindungen entwickeln. Allerdings kann es vorkommen, dass nicht-funktionale Strukturen nie an das funktionale Netz angeschlossen werden, so dass die Suche mit zusätzlichen Parametern belastet wird [SM02].

In der Natur zeichnen sich verschiedene Arten oft durch spezifische Strukturen aus, die es ihnen ermöglichen, in unterschiedlichen ökologischen Nischen zu gedeihen. Diese Differenzierung minimiert die Konkurrenz, da sich die Arten an spezifische physikalische Bedingungen wie Temperatur und Niederschlag anpassen. Ricklef et al. beschreibt in „The Economy of Nature“ [RRR14, S. 204–207], dass diese spezialisierte Anpassung es den Arten ermöglicht, Ressourcen effizient zu nutzen und stabilere Populationen zu bilden. So sind neue Entwicklungen innerhalb einer Nische weitgehend vor Konkurrenz geschützt, was zur Artenvielfalt und zur Stabilität von Ökosystemen beiträgt. Ein ähnliches Prinzip kann bei genetischen Algorithmen auf die Population angewendet werden. Dabei werden die Genotypen in verschiedene Spezies eingeteilt, die sich aufgrund spezifischer Merkmale oder Strukturen voneinander unterscheiden. Innerhalb dieser Spezies konkurrieren die Genotypen zunächst nur miteinander, was es neuen Varianten ermöglicht, sich zu entwickeln und zu optimieren, ohne sofort dem gesamten Druck der gesamten Population ausgesetzt zu sein. Dieser Ansatz fördert nicht nur die Vielfalt innerhalb der Spezies, sondern ermöglicht auch die Entstehung stabiler und anpassungsfähiger Lösungen, bevor sie mit anderen Spezies um begrenzte Plätze in der Population konkurrieren müssen.

Bereits vor der Veröffentlichung von NEAT wurde Speziation, auch bekannt als Niching, in verschiedenen genetischen Algorithmen untersucht, aber noch nicht auf die Neuroevolution angewendet. Die Speziation wurde zu dieser Zeit am häufigsten bei der multimodalen Funktionsoptimierung verwendet [Mah95], bei der eine Funktion mehrere Optima hat und ein GA mit mehreren Arten eingesetzt wird, um diese Optima zu finden.

Ein möglicher Grund, warum Speziation bis dahin nicht erfolgreich in die Neuroevolution implementiert wurde, ist die Schwierigkeit zu bestimmen, ob zwei Genotypen zur selben Art gehören. Normalerweise wird dies durch eine Kompatibilitätsfunktion entschieden,

aber es ist schwierig, eine solche Funktion für Netze mit unterschiedlichen Topologien zu formulieren. Darüber hinaus wurde im vorhergehenden Kapitel über das „Competing Conventions“-Problems gezeigt, dass funktionell identische Netze oft unterschiedliche Genotypen aufweisen, was die Zuordnung von Netzen zur gleichen Spezies erschwert. Speziation ist nicht desto trotz eine gute Möglichkeit, topologische Innovationen durch Speziation zu schützen, solange die oben beschriebenen Probleme effizient gelöst werden können, wie NEAT später demonstriert.

2.4.3 Initiale Population und topologische Innovation

Ein weiteres Problem bei TWEANN-Systemen ist die Wahl der Topologie in der initialen Population. Die meisten TWEANN-Systeme beginnen die initiale Population mit zufälligen Topologien. Stanley und Miikkulainen erkannten jedoch, dass eine solche Initialisierung mehrere Probleme mit sich bringt [SM02]. Zum Beispiel kann es bei vielen direkten Kodierungen vorkommen, dass die Netze keine vollständigen Pfade von den Eingängen zu den Ausgängen haben, wodurch sie von Anfang an nicht funktionsfähig sind und wertvolle Rechenzeit verschwendet wird, um sie aus- oder anzuschließen. Ein weiteres großes Problem ist die Schwierigkeit, eine minimale Lösung zu finden. Zufällige Topologien enthalten oft viele unnötige Knoten und Verbindungen, die nicht zur Lösung des Problems beitragen. Diese überflüssigen Strukturen müssen im Laufe der Evolution entfernt werden, was zusätzlichen Aufwand bedeutet. Größere Netze mit hoher Fitness könnten ihre überflüssigen Komponenten nie loswerden, da sie nicht für ihre Größe bestraft werden.

Es ist daher wichtig, minimale Topologien anzustreben, da diese die Komplexität der neuronalen Netze reduzieren, was zu schnelleren Lernzeiten, einer besseren Generalisierungsfähigkeit und einer geringeren Gefahr der Überanpassung an die Trainingsdaten führt [ZM$^+$93]. Eine Möglichkeit, dies zu erreichen, ist die Einbeziehung der Netzgröße in die Fitnessfunktion. Einige TWEANNs, wie von Zhang und Muhlenbein (1993) vorgeschlagen [ZM$^+$93], tun dies, indem sie die Fitness in Abhängigkeit von der Netzgröße reduzieren. Dies erfordert jedoch eine Anpassung des TWEANN-Algorithmus an das jeweilige Problem und kann zu unvorhersehbarem Verhalten führen, insbesondere in Situationen, in denen die optimale Netztopologie a priori unbekannt ist - eine Situation, die typischerweise auftritt, wenn TWEANNs verwendet werden [Yao99].

Eine Alternative besteht darin, die Evolution von Netzen mit einer minimalen Struktur zu beginnen und diese nur so weit wachsen zu lassen, wie es der Lösung dient. Dies ist das grundlegende Entwurfsprinzip von NEAT [SM02]. Durch den Start mit minimalen Strukturen wird der Suchraum klein gehalten, was die Optimierung und die Rechenzeit über alle Generationen verbessert.

Ein Grund, warum frühere TWEANNs meist nicht mit minimalen Topologien begonnen haben, ist, dass ohne topologische Vielfalt in der Ausgangspopulation Innovationen möglicherweise nicht überleben. Das Problem des Schutzes von Innovationen wird von diesen Methoden oft nicht ausreichend berücksichtigt, was dazu führt, dass Netze mit neuen strukturellen Ergänzungen nicht reproduziert werden. Im folgendes wird betrachtet wie der NEAT Algorithmus diese Problem durch eine Populationsspeziation löst, die eine

minimale Ausgangsbasis ermöglicht, ohne die Innovation zu gefährden. Darüber hinaus wird dargestellt, wie dieser auch mit den anderen zuvor betrachteten Problemen umgeht.

3 NEAT-Algorithmus

3.1 NEAT im Überblick

Der NEAT-Algorithmus folgt einem ähnlichen Ablauf wie ein typischer genetischer Algorithmus. Viele grundlegende Aspekte, wie die Fitnessbewertung, sind bereits bekannt und müssen daher hier nicht erneut detailliert erläutert werden.

Der allgemeine Ablauf des NEAT-Algorithmus wird der Übersichtlichkeit halber in folgendem Pseudocode zusammengefasst:

```
def NEAT(config):
    population = initialize_population(Population_Size)

    for generation in range(Generations):
        evaluate_fitness(population)
        speciate(population)

        new_population = []

        while len(new_population) < Population_Size:
            parent1 = select_parent(species)
            if random() < Crossover_Probability:
                parent2 = select_parent(species)
                offspring = crossover(parent1, parent2)
            else:
                offspring = clone(parent1)

            mutate(offspring)
            new_population.append(offspring)

        population = new_population
```

Im Folgenden werden die spezifischen Komponenten des NEAT-Algorithmus betrachtet, die als Lösungen für die zuvor diskutierten Herausforderungen dienen.

3.2 Genetische Kodierung

Wie bereits erwähnt, verwendet NEAT eine direkte Kodierung. Dies bedeutet, dass in jedem Genotyp jede Verbindung und jeder Knoten, der im Phänotyp erscheinen soll, explizit spezifiziert ist. Das genetische Kodierungsschema von NEAT ist so ausgelegt, dass die Gene durch die Innovationsnummern korrekt zugeordnet und bei der Kreuzung zweier Genotypen kombiniert werden können. Die Genotypen sind Repräsentationen der Netzkonnektivität.

Im NEAT-Algorithmus wird jeder Genotyp in zwei Hauptkategorien von Genen eingeteilt: Knotengene N und Verbindungsgene V. Diese Struktur $G = (N, V)$ ermöglicht eine detaillierte und genaue Darstellung der Netztopologie. Knotengene repräsentieren die Neuronen im Netz und werden durch eine Liste

$$N = \{n_1, n_2, n_3, \ldots, n_k\}$$

der einzelnen Knoten beschrieben. Jedes Knotengen $n_i = (id, t)$ hat eine eindeutige globale $id \in \mathbb{N}$ und einen Typ

$$t \in \{\text{Input}, \text{Output}, \text{Hidden}\}$$

d.h., ob es sich um einen Eingangsknoten, einen Ausgangsknoten oder ein versteckten Knoten handelt.
Zudem besitzt der Genotyp eine Liste

$$V = \{v_1, v_2, v_3, \ldots, v_m\}$$

von Verbindungsgenen. Verbindungsgene stellen die synaptischen Verbindungen zwischen den Neuronen dar. Jedes Verbindungsgen v_j wird wie folgt dargestellt:

$$v_j = (s_j, e_j, w_j, a_j, \iota_j)$$

wobei:

- $s_j \in \mathbb{N}_{\leq k}$: ID des Startknoten (das Neuron, von dem die Verbindung ausgeht),
- $e_j \in \mathbb{N}_{\leq k}$: ID des Endknoten (das Neuron, zu dem die Verbindung führt),
- $w_j \in \mathbb{R}$: Gewicht der Verbindung, das die Stärke des Signals bestimmt,
- $a_j \in \{0, 1\}$: Aktivierungsstatus (deaktiviert oder aktiviert),
- $\iota_j \in \mathbb{N}$: Innovationsnummer, eine eindeutige Kennung für die Evolutionshistorie der Verbindung.

Die Abbildung 3 zeigt diese Kodierung im Genotyp und Phänotyp.

Genome

Neuronen-Gene

Node ID	Typ
1	Input
2	Input
3	Hidden
4	Hidden
5	Output

Verbindungs-Gene

Start	Ende	Gewicht	Aktiv	Innov.
1	3	0.5	1	1
1	4	-0.3	1	2
2	3	0.8	0	4
2	4	-0.1	1	6
3	5	0.6	1	8
4	5	0.2	1	9

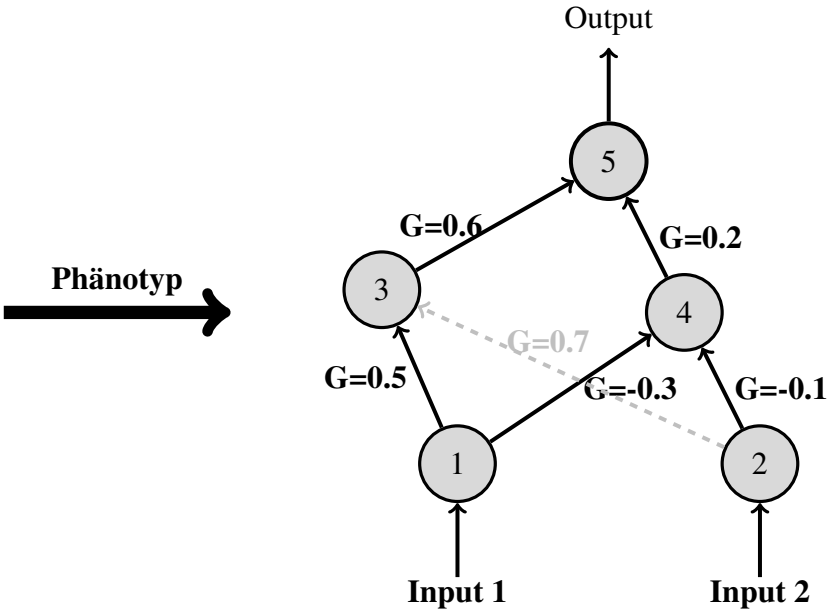


Abbildung 3: NEAT Genotyp Kodierung und der zugehörige Phänotyp

3.3 Mutationen in NEAT

Diese genetische Kodierung erlaubt die grundlegenden Arten von Mutationen, die man erwarten kann. Mutationen in NEAT können sowohl die Verbindunsgewichte als auch die topologische Struktur verändern. Die Gewichte der Verbindungen ändern sich wie in jedem NE-System, wobei jede Verbindung in jeder Generation entweder mutiert oder unverändert bleibt. Topologische Änderungen können in NEAT in zwei verschiedenen Mutationen auftreten, dem Hinzufügen einer Verbindung oder dem Hinzufügen eines Knotens. Im ersten Fall wird eine einzelne neue Verbindung

$$V_{m+1} = \{s_{K_1},\ e_{K_2},\ w,\ 1,\ \max(\iota) + 1\}$$

mit zufälligem Gewicht w zu zwei zuvor unverbundenen Knoten K_1, K_2 hinzugefügt. Bei der zweiten Mutation wird eine bestehende Verbindung $V_j = (s_j, e_j, w_j, a_j, \iota_j)$ in zwei neue Verbindungen aufgeteilt. Das bestehende Verbindungsgen wird deaktiviert $a_j = 0$ und zwei neue Verbindungsgene v_{j1}, v_{j2} werden erzeugt. Die neue Verbindung v_{j1}, die in den neuen Knoten n_{k+1} eintritt, wird mit einem Gewicht von 1 initialisiert, während

die Verbindung, die aus dem neuen Knoten austritt, dasselbe Gewicht w_j wie die zuvor existierende Verbindung erhält.

$$v_{j1} = (s_j,\ k+1,\ 1.0,\ 1,\ \max(\iota)+1)$$

$$v_{j2} = (k+1,\ e_j,\ w_j,\ 1,\ \max(\iota)+2)$$

Diese Methode wurde gewählt, um den Anfangseffekt durch das Hinzufügen eines neuen Knotens zu minimieren. Die neue Nichtlinearität in der Verbindung verändert die Funktion geringfügig, aber neue Knoten können sofort in das Netz integriert werden, im Gegensatz zum Hinzufügen einer größeren fremden Struktur, die später im Netz entwickelt werden müsste. Zusammen mit dem Mechanismus der Speziation, der später genauer betrachtet wird, wird sichergestellt, dass neue topologische Innovationen genügend Zeit haben, sich zu optimieren.

Beispiel. *Die möglichen Mutationen des KNN in Abbildung 3 werden visualisiert. Wenn ein Netz eine neue Verbindung einfügen möchte, werden zunächst zwei zufällige Knoten als Start- und Zielknoten ausgewählt. Zusätzlich wird geprüft, ob durch das Einfügen der neuen Verbindung kein Zyklus entsteht und das Netz somit rekursiv wird. In einigen modernen NEAT-Versionen wie z.B. [LHPS22], in denen rekursive Verbindungen erlaubt sind, kann dieser Schritt übersprungen werden. Diese Mutation ist dargestellt in Abbildung 4. Die neuen Änderungen sind grün markiert.*

Genome

N.-Gene

ID	Typ
1	Input
2	Input
3	Hidden
4	Hidden
5	Output

V.-Gene

I.	O.	Wght.	A.	IN.
1	3	0.5	1	1
1	4	-0.3	1	2
2	3	0.8	0	4
2	4	-0.1	1	6
3	5	0.6	1	8
4	5	0.2	1	9
1	5	0.4	1	10

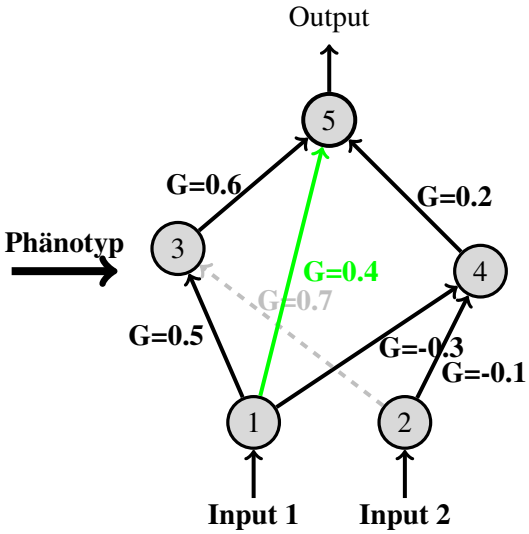


Abbildung 4: Mutation neue Verbindung

Wenn NEAT einen neuen Knoten durch Mutation einfügen will, wird zunächst eine zufällige Verbindung ausgewählt, die geteilt werden soll. In unserem Beispiel wurde die Verbindung von Knoten 1 *zu Knoten* 4 *ausgewählt. Die alte Verbindung wird dann deaktiviert und zwei neue Verbindungen werden wie oben beschrieben hinzugefügt (siehe Abbildung 5).*

Genome

N.-Gene

ID	Typ
1	Input
2	Input
3	Hidden
4	Hidden
5	Output
6	Hidden

V.-Gene

I.	O.	Wght.	A.	IN.
1	3	0.5	1	1
1	4	-0.3	**0**	2
2	3	0.8	0	4
2	4	-0.1	1	6
3	5	0.6	1	8
4	5	0.2	1	9
1	4	1.0	1	10
4	6	-0.3	1	11

Phänotyp

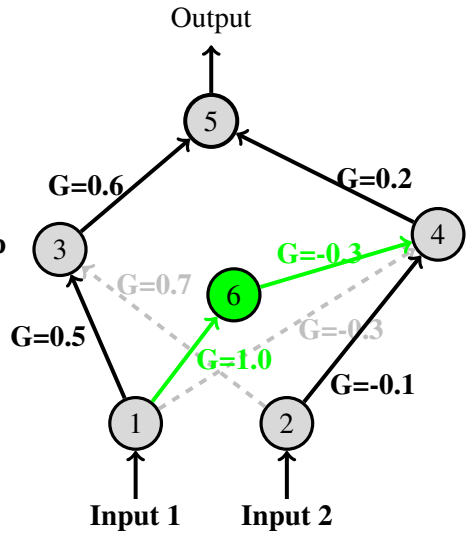


Abbildung 5: Mutation neuer Knoten

Bei den anderen Mutationen kann es sich entweder um die Deaktivierung einer Verbindung $a_j = 0$ handeln, wie hier $(1, 4)$, oder um die Reaktivierung einer alten Verbindung $a_4 = 1$, wie in $(3, 3)$ (dies kann auch beim Versuch geschehen, eine zuvor deaktivierte Verbindung wieder hinzuzufügen). Außerdem ist es immer möglich, eine zufällige Verbindung aufgrund ihrer Eigenschaften zu mutieren. Im grundlegenden NEAT-Algorithmus würde dies nur bedeuten, dass die Gewichtung wie in $(3, 5)$ mutiert [4] *wird. In modernen Versionen kann z.B. auch der Typ der Aktivierungsfunktion mutiert werden, wie in [MLM+17]. Die grundlegenden Mutation sind dargestellt in Abbildung 6.*

Genome

N.-Gene

ID	Typ
1	Input
2	Input
3	Hidden
4	Hidden
5	Output

V.-Gene

I.	O.	Wght.	A.	IN.
1	3	0.5	1	1
1	4	-0.3	**0**	2
2	3	0.8	**1**	4
2	4	-0.1	1	6
3	5	**0.1**	1	8
4	5	0.2	1	9

Phänotyp

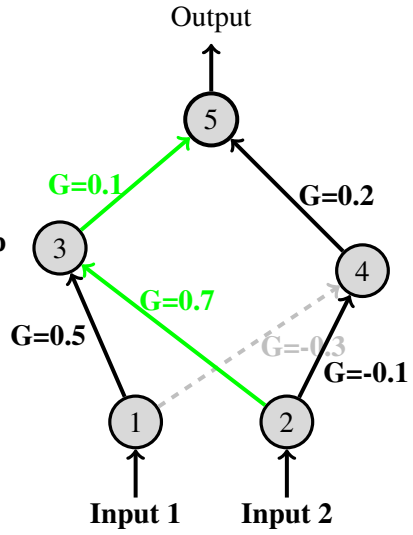


Abbildung 6: Sonstige Mutationen

Wie in unser Beispiel werden die Genotypen in NEAT immer größer da alte Verbindungen und nicht funktionellen Knoten nicht vollständig aus den Genotypen entfernt werden

[4] Siehe dazu das Kapitel zu Genetischen Algorithmen 2.1

sondern lediglich deaktiviert werden. Dies führt dazu dass auf längere Zeit die Genotypen immer größer werden, was zu unterschiedlich großen Genotypen führt, die teilweise sogar unterschiedliche Verbindungen an den gleichen Positionen aufweisen. Dies geschieht jedoch bewusst da diese mit sogenannten Innovationsnummern versehen sind welche erlauben das Problem der „Competing Conventions“ selbst für unterschiedliche Topologien und Gewichtskombinationen zu lösen.

3.4 Innovationsnummern: Schlüssel zur genetischen Kompatibilität

In der Evolution gibt es Informationen, die uns helfen können herauszufinden, welche Gene bei verschiedenen Individuen übereinstimmen. Diese Information beruht auf dem Ursprung der Gene. Wenn zwei Gene denselben Ursprung haben, handelt es sich um homologe Gene, die von einem gemeinsamen Vorfahren abstammen und daher ähnliche Eigenschaften haben. Ein System könnte also einfach herausfinden, welche Gene zusammengehören, indem es den Ursprung jedes Gens verfolgt [Koo05]. Auf der Grundlage dieser Idee funktioniert die oben erwähnte Innovationsnummer in den Verbindungsgenen [SM02]. Bei jedem Auftreten eines neuen Gens wird eine globale Innovationsnummer inkrementiert und den neuen Genen zugeordnet. Mit Hilfe der Innovationsnummer kann somit eine chronologische Reihenfolge des erstmaligen Auftretens der Gene erstellt werden. Dies lässt sich in unseren vorherigen Abbildungen 3, 4, 5 und 6 einsehen.

Um zu verhindern, dass identische topologische Innovationen innerhalb derselben Generation unterschiedliche Innovationsnummern erhalten, wird in jeder Generation eine Liste der bereits aufgetretenen Innovationen geführt. Auf diese Weise kann sichergestellt werden, dass zwei identische topologische Innovationen, die unabhängig voneinander in derselben Generation auftreten, dieselbe Innovationsnummer erhalten. Dadurch wird vermieden, dass die Anzahl der Innovationsnummern unnötig ansteigt.

Mit Hilfe der historischen Markierungen ist der NEAT-Algorithmus in der Lage, eine Kreuzung zwischen Genotypen mit unterschiedlichen Strukturen durchzuführen [Sta04]. Anhand der Innovationsnummern kann schnell festgestellt werden, welche Gene übereinstimmen, d.h. als „matching genes“ klassifiziert werden.
Es werden zwei Elternteile G_A und G_B mit ihren Verbindungsgene V_A und V_B betrachtet. *Matching genes* werden definiert als:

$$M = \{v_j \mid v_j \in V_A \cap V_B\}$$

Gene, die nur bei einem Elternteil vorhanden sind, werden als „disjoint“ (nicht übereinstimmend) oder „excess“ (überschüssig) klassifiziert, je nachdem, ob sie innerhalb oder außerhalb des Bereichs der höchsten geteilten Innovationsnummer des anderen Elternteils liegen.

$$D_A = \{v_j \mid v_j \in (V_A \setminus V_B) \text{ und } n_j \leq \max(\{n_i \mid v_i \in M\})$$
$$D_B = \{v_j \mid v_j \in (V_B \setminus V_A) \text{ und } n_j \leq \max(\{n_i \mid v_i \in M\})$$

$$E_A = \{v_j \mid v_j \in (V_A \setminus V_B) \text{ und } n_j > \max(\{n_i \mid v_i \in M\})$$
$$E_B = \{v_j \mid v_j \in (V_B \setminus V_A) \text{ und } n_j > \max(\{n_i \mid v_i \in M\})$$

Das Produkt der Kreuzung V_C von den Eltern (V_A, V_B) sind dabei die Verbindungsgene des erzeugtes Kindes. Alle „excess“ und „disjoint“ Gene werden immer vom fitteren Elternteil vererbt. Zur Entscheidung, welche Gene bei gleicher Fitness der Eltern G_A und G_B in das Kindergenom übernommen werden, wird eine Zufallsvariable ω eingeführt. Diese ist gleichmäßig in $[0, 1]$ verteilt und bestimmt, ob ein Gen aus G_A oder G_B übernommen wird.

$$v_j \in V_C, \quad \text{falls} \begin{cases} v_j \in M \\ v_j \in (D_A \cup E_A) \text{ und Fitness}(G_A) > \text{Fitness}(G_B) \\ v_j \in (D_B \cup E_B) \text{ und Fitness}(G_B) > \text{Fitness}(G_A) \\ v_j \in (D_A \cup E_A) \text{ und Fitness}(G_A) = \text{Fitness}(G_B) \text{ und } \omega \leq 0.5 \\ v_j \in (D_B \cup E_B) \text{ und Fitness}(G_A) = \text{Fitness}(G_B) \text{ und } \omega > 0.5 \end{cases}$$

Zusätzlich verwendet zwei Varianten für die Gewicht-Vererbung der matching Gene welche beiden zur Anwendung kommen. Um die Attribute wie Verbindungsgewichte w_j und Aktivierungswerte a_j den jeweiligen Genomen G_A, G_B und G_C zuzuordnen, werden Exponenten wie A, B und C verwendet. Zum Beispiel beschreibt w_j^A das Gewicht von v_j in G_A, w_j^B das Gewicht in G_B und w_j^C das entsprechende Gewicht im Kindergenom G_C.
Bei der ersten Variante werden die Gewichte zufällig von einem Elternteil ausgewählt.

$$w_j^C = \begin{cases} w_j^A, & \text{falls } \omega \leq 0.5 \\ w_j^B, & \text{falls } \omega > 0.5 \end{cases}$$

Bei der zweiten Variante wird das Gewicht des Durchschnitts der übereinstimmenden Gene verwendet.

$$w_j^C = \frac{w_j^A + w_j^B}{2}$$

Für beide Varianten gilt, dass Gene, die bei einem Elternteil deaktiviert sind, mit hoher Wahrscheinlichkeit auch beim Kind deaktiviert sind [5] [Sta13].

$$a_j^C = \begin{cases} 0 & \text{mit Wahrscheinlichkeit } p, \text{ wenn } a_j^A = 0 \text{ oder } a_j^B = 0 \\ 1 & \text{mit Wahrscheinlichkeit } (1-p), \text{ sonst} \end{cases}$$

Die Knotengene des Kindes N_C sind alle Gene welche in einer beliebigen Verbindung von V_C vorkommen:

$$N_C = \{s_j, e_j | \ \exists v_j \in V_C\}$$

Abschließend ergibt sich somit $G_C = (N_C, V_C)$.

Beispiel. *Die Farbmarkierungen in der endgültigen Grafik (Abbildung 7) zeigen an, von welchem Elternteil die Verbindung vererbt wurde. In diesem Beispiel haben beide Individuen die gleiche Fitness für jede topologische Struktur, d. h. für alle „disjoint“ oder*

[5] Im Original beträgt die Wahrscheinlichkeit $p = 75\%$

„excess“ Gene wird zufällig entschieden ob diese vererbt werden soll. Zu Demonstrationszwecken wurde hier jede dieser Strukturen vererbt. Außerdem wurde für die Wahl der Gewichte die erste Variante gewählt, d.h. jedes Gewicht stammt zufällig von einem der beiden Elterngewichte ab (dies wäre auch der Fall, wenn die Fitnesswerte unterschiedlich wären!)
In diesem Beispiel können einige wichtige Beobachtungen gemacht werden. Die globale Knoten-ID und die Innovationsnummer jeder Verbindung ermöglichen eine sinnvolle topologische Kreuzung der beiden Graphen. Zum Beispiel integrieren die Knoten $2, 5, 6$ *und* $9, 7, 4$ *sinnvoll die Strukturen des jeweils anderen Graphen.*
Verbindungen wie $(1, 5)$ *und* $(5, 7)$*, die in einem der beiden Elterngraphen aktiv und im anderen deaktiviert waren, sind hier zufällig aktiviert* $(5, 7)$ *bzw. deaktiviert* $(1, 5)$*. Außerdem ist zu erkennen, dass jede Gewichtung der entsprechenden Gene zufällig von einem der Elterngraphen übernommen wurde. So stammen die Gewichte der Gene mit den Nummern* $1, 2, 6, 8, 9$ *vom ersten Elternteil, während die Gene mit den Nummern* $3, 4, 11, 12$ *vom zweiten Elternteil stammen.*
Der erzeugte Nachkomme zeigt auch, welche Gene als „disjoint“ oder „excess“ betrachtet werden:

- *disjoint: Gene, die in beiden Eltern vorkommen und eine Innovationsnummer von maximal* 15 *(höchste IN. des kleineren Graphen) haben.*
- *Excess: Gene, die nur in einem der beiden Elternteile vorkommen und eine Innovationsnummer größer als* 15 *haben.*

Beispiel. *Es wird ein weiteres Beispiel (Abbildung 8) mit denselben beiden Elterngraphen aus der Abbildung 7 betrachtet. Diesmal wird die zweite Variante zur Wahl der Gewichtung verwendet. Es wird jedoch angenommen, dass der rechte Elterngraph eine höhere Fitness hat als der linke.*
Erste Beobachtungen zeigen sofort, dass alle topologischen Strukturen, die ausschließlich beim schwächeren Elternteil vorkommen, nicht vererbt werden. Dennoch gibt das schwächere Elternteil einige genetische Informationen an das Kind weiter. Unter anderem wird für alle Gewichte in den übereinstimmenden Genen der Durchschnitt der Gewichte beider Eltern genommen (2. Variante). Außerdem werden Verbindungen, die bei einem Elternteil deaktiviert wurden, auch beim Kind deaktiviert. In diesem Fall wurden also sowohl die Verbindung $(5, 7)$ *als auch* $(1, 5)$ *beim Kind deaktiviert. Für die Berechnung der Gewichtung der Gene ist der Aktivierungsstatus im Elternteil irrelevant, solange sie vorhanden sind.*

Nachdem erläutert wurde, wie der NEAT-Algorithmus durch Kreuzung und anschließende Mutationen zur Vielfalt der Topologien und damit zur Suche nach einer guten Lösung beiträgt, ist es wichtig sicherzustellen, dass die daraus resultierenden Innovationen nicht verloren gehen. Die Kreuzung erzeugt verschiedene Kombinationen von Netzstrukturen, während Mutationen zusätzliche Variationen hinzufügen. Damit neue und potenziell wertvolle Strukturen genügend Zeit haben, sich zu entwickeln und zu optimieren, spielt die Speziation eine zentrale Rolle. Sie stellt sicher, dass Innovationen geschützt bleiben und dass sich die Population sowohl vielfältig als auch effizient entwickelt.

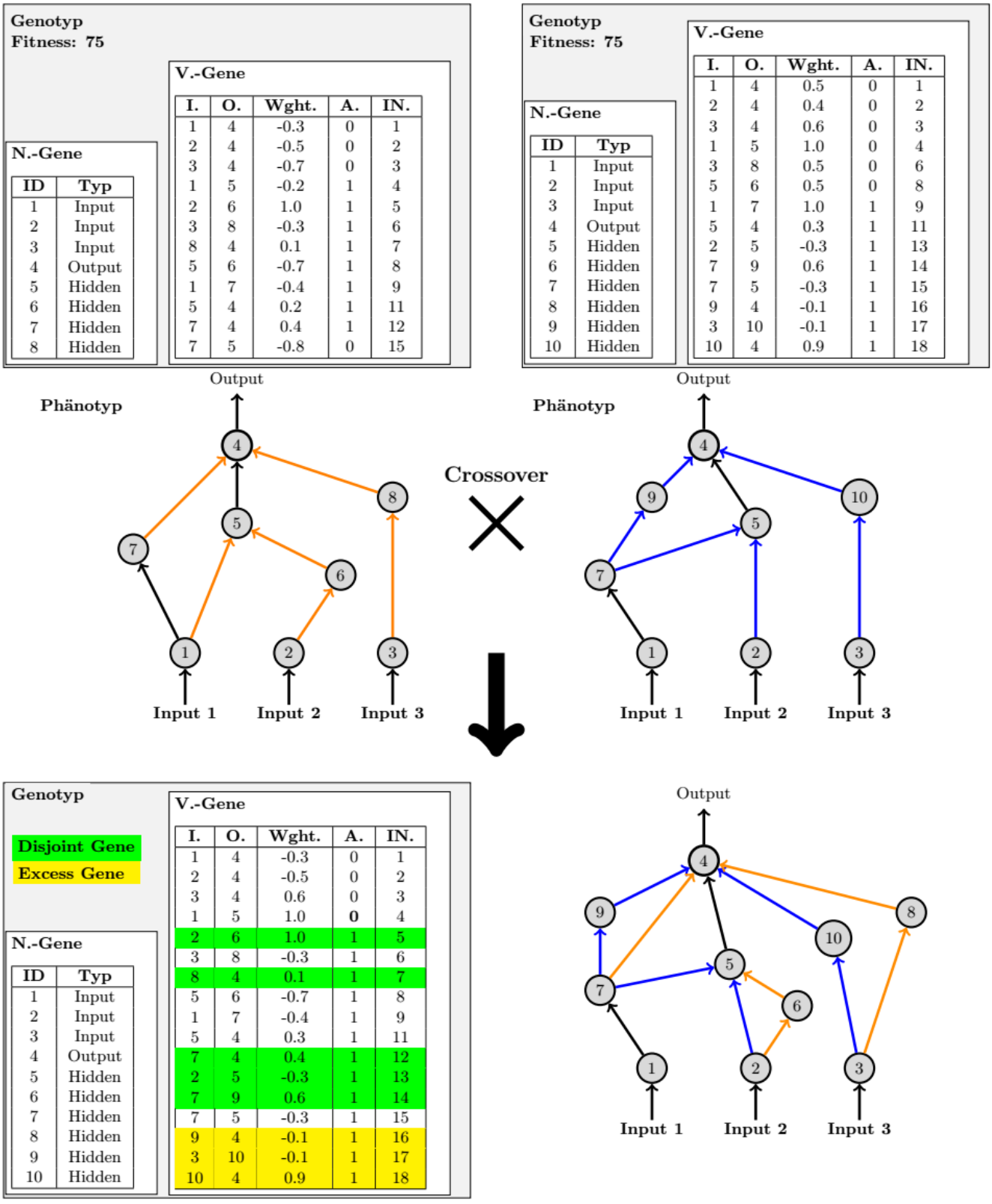


Abbildung 7: Kreuzung zweier Individuen

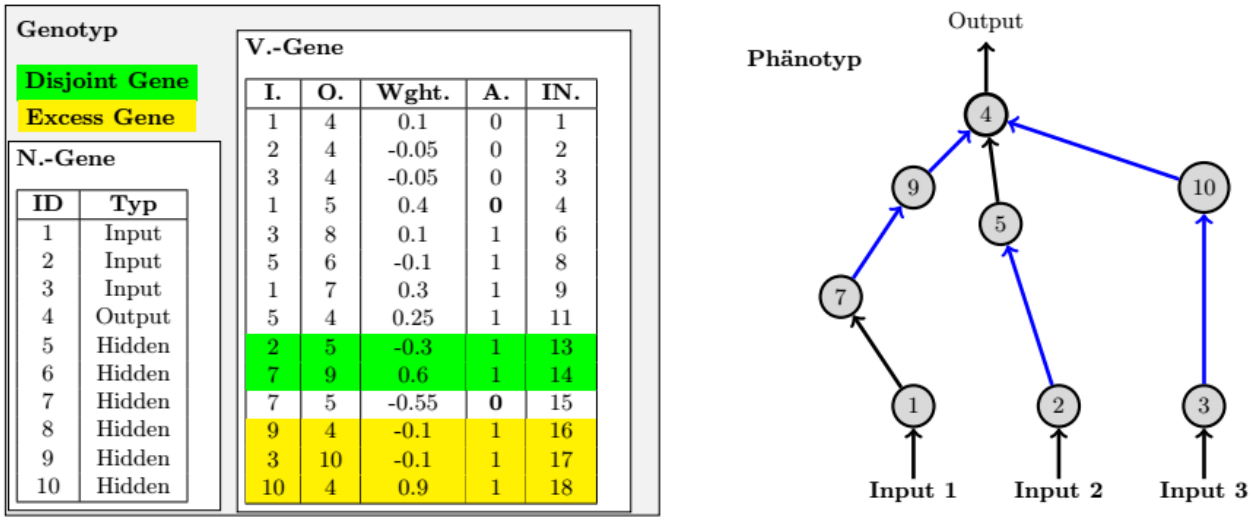


Abbildung 8: Kreuzung zweier Individuen

3.5 Speziation: Schutz vor genetischer Dominanz

Die Aufteilung der Population in verschiedene Spezies ermöglicht es den Individuen, vor allem innerhalb ihrer eigenen Spezi zu konkurrieren und nicht mit der Gesamtpopulation. Dadurch werden neue topologische Innovationen geschützt, indem sie sich in ihrer Spezi entwickeln und optimieren können. Die Grundidee besteht darin, die Population in Gruppen einzuteilen, so dass Netze mit ähnlichen Topologien in derselben Gruppe zusammengefasst werden. Obwohl dies wie ein Problem der Topologiezuordnung aussieht, bieten historische Marker eine effiziente Lösung für diese Aufgabe [SM02].

Die Kompatibilität von zwei verschiedenen Genen wird in NEAT mit der folgenden Funktion berechnet:

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

E und D geben die Anzahl der „excess“ und „disjoint“ Gene in den zu vergleichenden Genotypen an. $\overline{W}$ steht für die durchschnittliche Gewichtsdifferenz in den „matching“ Genen. Die zugehörigen Koeffizienten c_1, c_2 und c_3 erlauben es, die Wichtigkeit der einzelnen Faktoren anzupassen. Schließlich steht N für die Anzahl der Gene im größeren der beiden Genotypen und dient zur Normierung des Distanzmaßes. In Fällen, in denen N sehr klein ist, z.B. unter 20, kann es standardmäßig auf 1 gesetzt werden. Das Distanzmaß δ erlaubt es, mit Hilfe einer Kompatibilitätsschwelle δ_t die einzelnen Genotypen in verschiedene Spezies einzuteilen. Dazu wird für jede Spezi eine geordnete Liste geführt. Der Repräsentant einer Spezies ist immer das erste Individuum, das dieser Spezies zugeordnet wird. In jeder neuen Generation werden die Genotypen nacheinander den existierenden Spezies zugeordnet. Ein Genotyp g der aktuellen Generation wird der ersten Spezi zugeordnet, mit der es aufgrund von δ_t kompatibel ist. Wenn g mit keiner der vorhandenen Spezies kompatibel ist, wird eine neue Spezi erzeugt und g als deren repräsentativer Genotyp ausgewählt. Es ist anzumerken, dass es genauere Methoden zur Klassifizierung von Genotypen gibt, aber Stanley und Miikkulainen argumentieren, dass diese Methode

genau genug ist und der zusätzliche Rechenaufwand für komplexere Ansätze nicht gerechtfertigt wäre. [SM02].

Beispiel. *Zur Veranschaulichung wird die Formel auf die beiden Genotypen des oben beschriebenen Kreuzung-Beispiels 7 angewendet. In diesem Beispiel sind* 3 *„excess“ Gene,* 5 *„disjoint“ Gene und* 9 *„matching“ Gene vorhanden.*

Tabelle 5: „matching Gene“ und ihre Gewichte

IN	Wght. 1	Wght. 2	Wght. Differenz
1	-0.3	0.5	0.8
2	-0.5	0.4	0.9
3	-0.7	0.6	1.3
4	-0.2	1.0	1.2
6	-0.3	0.5	0.8
8	-0.7	0.5	1.2
9	-0.4	1.0	1.4
11	0.2	0.3	0.1
15	-0.8	-0.3	0.5
Summe			**7.2**

Es ergibt sich $\overline{W} = \frac{7.2}{9} = 0.8$*. Nun kann* δ *berechnet werden. Da die Koeffizienten problemabhängig sind, werden der Einfachheit halber* $c_1, c_2 = 1$ *und* $c_3 = 0.4$ *gewählt. Weiterhin wird die Funktion mit* $N = 14$ *betrachtet. Es ergibt sich:*

$$\delta = \frac{3}{14} + \frac{5}{14} + *0.4 * 0.8 = 0.51$$

Da die Kompatibilitätsschwelle δ_t *oft experimentell bestimmt wird, sind diese Zahlen allein nicht aussagekräftig. In Abhängigkeit von ihnen kann jedoch festgestellt werden, ob die beiden Graphen zur gleichen Spezies gehören. Wäre* $\delta_t = 1.0$*, würden unsere Genotypen zu derselben Spezies gehören.*

Um die genetische Vielfalt in der Population von NEAT zu erhalten und sicherzustellen, dass verschiedene Spezies nebeneinander existieren können, verwendet NEAT explizites Fitness-Sharing. Im Gegensatz zur ursprünglichen impliziten Version des Fitness Sharing, die von Holland [Hol92] eingeführt wurde und Individuen nach ihrer Leistungsähnlichkeit gruppiert, basiert die explizite Version auf der genetischen Ähnlichkeit der Netze [Gol89]. Diese Methode eignet sich besonders gut für TWEANN, da sie die Gruppierung von Netzen auf der Grundlage ihrer Topologie und ihrer Gewichtskonfigurationen ermöglicht. Fitness-Sharing begrenzt die Anzahl der Netze, die sich auf einem einzelnen Fitness-Peak entwickeln können, indem die Population in verschiedene Spezies aufgeteilt wird, die sich jeweils auf einem anderen Peak befinden. Dadurch wird verhindert, dass eine einzige Spezies die Oberhand gewinnt, und Innovationen bleiben innerhalb der eigenen Spezies geschützt.

Die angepasste Fitness f_i' für jeden Genotyp i wird berechnetet mithilfe des Distanzmaßes δ zu jeden Genotyp j in der Population:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}$$

Die Sharing-Funktion *sh* wird auf 0 gesetzt, wenn der Abstand $\delta(i, j)$ zwischen zwei Individuen den festgelegten Schwellenwert δ_t überschreitet. Liegt der Abstand jedoch unter diesem Schwellenwert, wird sh$(\delta(i, j))$ auf 1 gesetzt [SGH95]. Dies bedeutet, dass die Funktion *sh* im Wesentlichen die Anzahl der Organismen innerhalb derselben Spezies wie das betrachtete Individuum reduziert. Diese Reduktion ist natürlich, da die Spezies bereits durch den Kompatibilitätsschwellenwert δ_t gruppiert werden. Jede Spezies k erhält eine unterschiedliche Anzahl von Nachkommen n_k, die proportional zur Summe der angepassten Fitnesswerte f_i' ihrer Mitglieder ist, dies geschieht mithilfe folgender Formel [Sta04] :

$$n_k = \frac{\overline{F}_k}{\overline{F}_{tot}}|P|$$

Wobei $\overline{F}_k$ die durchschnittliche Fitness der Spezies k ist, $|P|$ die Größe der Population und $\overline{F_{tot}} = \sum_k \overline{F}_k$ die Summe über die durchschnittliche Fitness für alle Spezien.
Die Reproduktion erfolgt, indem die schlechtesten Individuen aus der Population entfernt werden. Die restlichen Individuen werden uniform als Elternteil für die nächste Generation gewählt. Eine Ausnahme bilden die Individuen mit der höchsten Fitness jeder Generation, die identisch an die nächste Generation weitergegeben werden. Die verbleibenden Individuen der nächsten Generation sind die Nachkommen der zuvor ausgewählten Eltern [Sta04]. In seltenen Fällen, in denen sich die Fitness der Population über eine bestimmte Anzahl von Generationen (typischerweise 20) nicht verbessert, werden nur die beiden fittesten Individuen zur Fortpflanzung zugelassen. Dies dient dazu, die Suche auf die vielversprechendsten Gebiete zu konzentrieren [SM02].

Beispiel. *Ein einfaches Beispiel mit* 10 *Individuen wird betrachtet. Der Einfachheit halber wird davon ausgegangen, dass diese bereits aufgrund ihrer Topologie in drei verschiedene Spezies unterteilt werden können. Außerdem wird angenommen, dass zwei beliebige Individuen aus verschiedenen Spezies stammen, deren Abstände immer den Schwellenwert* δ_t *überschreiten (was nicht immer der Fall ist!).*
Aus den Tabellem 6 und 7 ist ersichtlich, dass das Fitness-Sharing wie gewünscht funktioniert: So wurden z.B. alle Individuen der Spezies A *dafür bestraft, dass ihre Spezies die höchste durchschnittliche Fitness besitzt. Während die Individuen der anderen Spezies leicht dafür belohnt wurden, dass sie weniger Varianz in ihrer Spezies haben, wie z.B. in Spezies* B*. Zudem werden die Überflieger in schlechteren Spezies belohnt , siehe z.B.* $C2$*.*
Unsere Ergebnisse zeigen, wie das Fitness-Sharing leistungsfähigere Spezies wie Spezies A *durch eine höhere Anzahl von Nachkommen belohnt, während es gleichzeitig anderen Spezies die Chance gibt, sich fortzupflanzen, und so ein mögliches Aussterben von Spezies* B *verhindert, wenn nur die stärksten Individuen zur Fortpflanzung zugelassen wurden wären.*

Im letzten Abschnitt über den NEAT-Algorithmus wird kurz untersucht, wie die Suche möglichst effizient gestaltet werden kann. Dazu wird die initiale Population betrachtet.

Tabelle 6: Individuen mit aktualisierten Fitnesswerten

Individuum	f_i	$\sum_i f_i$	f_i'	Rang vorher	Rang nachher
A1	99	238	0.4159	1	2
A2	84	238	0.3529	4	5
A3	55	238	0.2311	6	8
B1	54	157	0.3439	7	6
B2	58	157	0.3694	5	4
B3	45	157	0.2866	8	7
C1	12	218	0.0550	10	10
C2	96	218	0.4404	2	1
C3	25	218	0.1147	9	9
C4	85	218	0.3899	3	3

Tabelle 7: Nachkommen jeder Spezies

Spezies	Summe Gesamtfitness	$\lvert P_{alt}\rvert$	$\overline{F}_k$	$\overline{F}_{tot}$	$n_k = \frac{\overline{F}_k}{\overline{F}_{tot}}\lvert P\rvert$
A	238	3	79.33	186.16	$\frac{79.33}{186.16} \cdot 10 \approx 4.26 \approx 4$
B	157	3	52.33	186.16	$\frac{52.33}{186.16} \cdot 10 \approx 2.81 \approx 3$
C	218	4	54.50	186.16	$\frac{54.50}{186.16} \cdot 10 \approx 2.93 \approx 3$

3.6 Inkrementelles Wachstum und minimale Strukturen in NEAT

Im Gegensatz zu anderen TWEANNs, die oft mit einer Anfangspopulation zufälliger Topologien beginnen, wie in Kapitel 2.4.3 gezeigt, beginnt NEAT mit einer einheitlichen Population von Netzen ohne versteckte Knoten, in denen alle Eingaben direkt zu Ausgaben führen. Neue Strukturen werden schrittweise durch strukturelle Mutationen eingeführt, und nur die Strukturen, die sich als nützlich erweisen, überleben die Fitnessbewertungen. Auf diese Weise werden alle strukturellen Erweiterungen in NEAT gerechtfertigt. Da die Population zu Beginn minimal ist, wird der Suchraum auf ein Minimum reduziert. Das bedeutet, dass NEAT in weniger Dimensionen sucht als andere TWEANNs oder Systeme mit fester Topologie, was NEAT einen Leistungsvorteil verschafft. Dies ist nur möglich dank der effizienten Mechanismen der Speziation und der gezielten Integration neuer Knoten.

Zur Validierung der in diesem Kapitel beschriebenen Komponenten und Mechanismen des NEAT-Algorithmus wurden zwei Praxisprojekte durchgeführt. Diese Projekte dienen nicht nur dazu, den Beitrag der einzelnen Komponenten zum Gesamtalgorithmus zu bewerten, sondern auch dazu, den NEAT-Algorithmus mit anderen modernen Ansätzen zu vergleichen.

4 Praktische Projekte

4.1 Projekt 1: Komponentenanalyse des NEAT Algorithmus

Das erste praktische Projekt soll zeigen, dass jede Komponente des NEAT-Algorithmus ihre Daseinsberechtigung hat und den Algorithmus unterstützt. Dazu wird die *Swimmer*-Umgebung der Python-Bibliothek Gymnasium[Mujoco] [TKT+24] untersucht, die auf dem gleichnamigen Experiment von Rémi Coulom [Cou02] basiert.

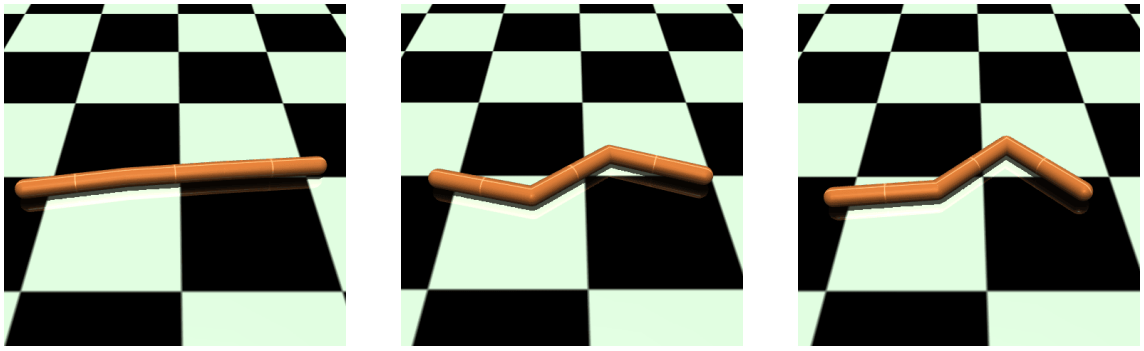


Abbildung 9: Mujoco Swimmer Umgebung

Die Swimmer-Umgebung ist für unsere Untersuchung relevant, da sie eine realistische Testbedingung darstellt, die komplex genug ist, um als angemessene Herausforderung zu dienen, aber gleichzeitig einfach genug, um gezielte Experimente durchzuführen. Sie simuliert einen wurmartigen Roboter, der sich innerhalb einer vorgegebenen Zeit so effizient wie möglich fortbewegen muss. Der Agent muss nicht nur grundlegende Bewegungsmuster erlernen, sondern diese auch bis an die Grenzen seiner physikalischen Leistungsfähigkeit optimieren.
MuJoCo ist eine Physik-Engine, die schnelle und präzise Simulationen für die Forschung und Entwicklung in den Bereichen Robotik, Biomechanik, Grafik und Animation ermöglicht [TET12]. Die Problemparameter sind $n = 3$ für die Anzahl der Körperteile des Swimmers, $l_i = 0,1$ für die Länge jedes Teils und $k = 0,1$ für den viskosen Reibungskoeffizienten.

4.1.1 Erklärung der Implementierung

Im ersten praktischen Projekt werden verschiedene Versionen des NEAT-Algorithmus vergliechen, bei denen jeweils bestimmte Komponenten aus der vollständigen Version entfernt wurden. Die untersuchten Versionen sind:

- **Normal**: Diese Version enthält alle Komponenten des NEAT-Algorithmus.
- **InitRandom**: Diese Version startet mit mehreren inneren versteckten Knoten anstatt nur mit einem minimalen Netz, um zu untersuchen, wie sich die anfängliche Netzstruktur auf die Lernfähigkeit und die Entwicklung der Netztopologie auswirkt. Ein Netz, das mit mehreren versteckten Knoten beginnt, könnte schneller eine effiziente Lösung finden, könnte jedoch auch riskieren, suboptimale Topologien zu

entwickeln oder unnötige Rechenzeit durch nicht optimierte Anfangsstrukturen zu verschwenden.

- **NonMating**: In dieser Version gibt es keine Kreuzungen zwischen den Netzen, so dass die Evolution ausschließlich durch Mutation erfolgt. Diese Variante untersucht, wie sich das Fehlen von Kreuzungen auf die Evolution des Netzes auswirkt.

- **NoGrowth**: In dieser Version kann das Netz seine Topologie nicht ändern und bleibt auf seine ursprüngliche Struktur und Gewichtsänderungen beschränkt. Zur Unterstützung wird das Netz mit einigen inneren versteckten Knoten initialisiert. Diese Version untersucht, wie Neuroevolution auf grundlegenden genetischen Algorithmen aufbaut.

- **NoSpecie**: In dieser Version wird auf die Spezifikation verzichtet, so dass einzelne Netze direkt mit der Grundgesamtheit konkurrieren müssen. Diese Variante analysiert, wie das Fehlen von Spezialisierung die Evolution und die Leistung von Netzen beeinflusst. Ohne Spezialisierung könnte der Algorithmus Schwierigkeiten haben, Innovationen langfristig zu schützen und damit lokale Minima zu überwinden.

Die Implementierung des NEAT-Algorithmus erfolgt mit Hilfe der auf PyTorch [PGC+17] basierenden NEAT-Python-Bibliothek [MKM+22]. Es ergibt sich eine Umgebung mit 8 verschiedenen Eingabeknoten und 2 kontinuierlichen Ausgabeknoten. Unser Algorithmus startet jeweils ohne initiale Verbindungen (mit Ausnahme der NoGrowth Version) und mit einer Population von 100. Weitere technische Details sind in der Konfigurationsdatei zu finden.

4.1.2 Analyse der Ergebnisse

Die oben vorgestellten Versionen werden im Rahmen des Experiments fünfmal ausgewertet, um genauere Ergebnisse zu erhalten. Ziel jeder Version ist es, innerhalb einer vorgegebenen Generationsperiode von 250 Generationen eine maximale Fitness von 360 zu erreichen. Wird die Zielfitness vor Ablauf der Generationsperiode erreicht, wird der Algorithmus abgebrochen, was in den Abbildungen immer durch einen Punkt gekennzeichnet ist. Die Zielfitness wurde durch umfangreiche Experimente ermittelt und gibt die Fähigkeit des Agenten an, sich während des Simulationszeitraums kontinuierlich in einer effizienten Bewegung vorwärts zu bewegen. Eine Fitness von 350 kann bereits als erfolgreiche Implementierung angesehen werden, wobei die letzten Fitnesspunkte sehr kleine Bewegungen optimieren, die dem Agenten helfen, absolut optimal zu arbeiten. Wenn der Algorithmus beim Erreichen der Zielfitness vorzeitig abbricht, bedeutet dies, dass er schneller eine Lösung gefunden hat. Die Ergebnisse sind in Abbildung 10 dargestellt.

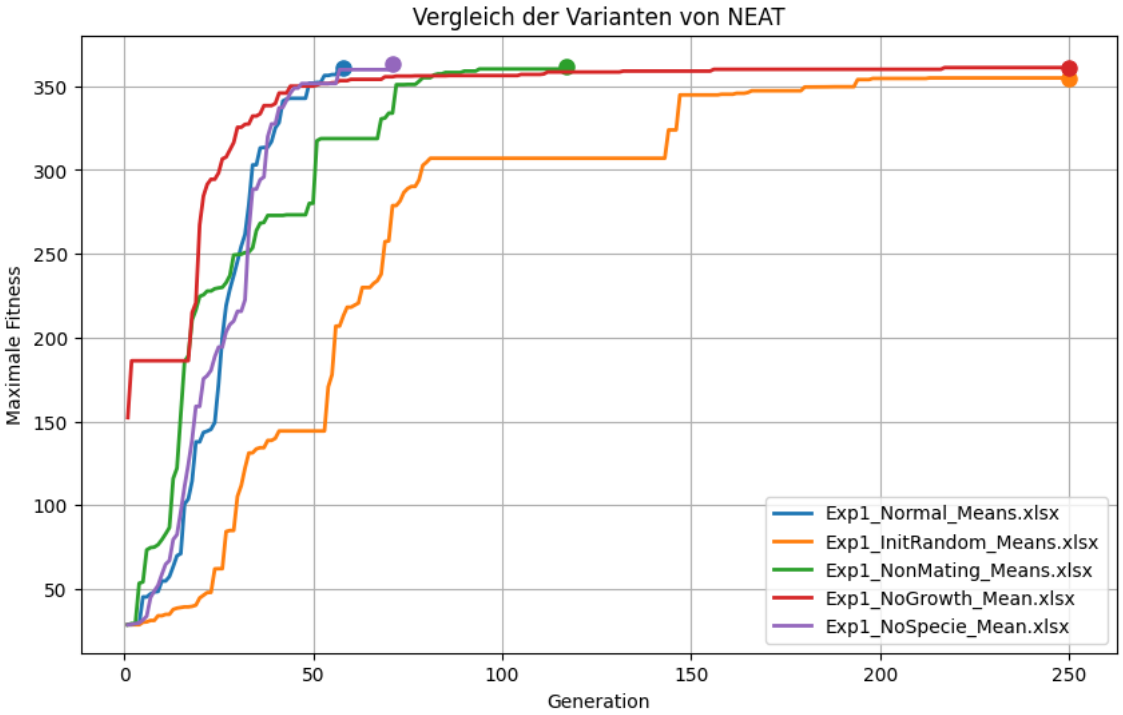


Abbildung 10: Ergebnisse des ersten Experiments

Um die Ergebnisse besser zu verstehen, werden die durchschnittlichen Zeiten für jede Version betrachtet. Diese sind in der Tabelle 8 dargestellt.

Version	Durchschnittliche Zeit [s]	Durchschnittliche Generationen
Normal	184.79	51.80
InitRandom	492.08	161.00
NonMating	190.60	69.00
NoGrowth	740.21	164.20
NoSpecie	164.00	48.80

Tabelle 8: Durchschnittliche Laufzeiten der einzelnen Version

Die ersten Ergebnisse zeigen, dass die vollständige NEAT-Version und die Version ohne Speziation fast gleich gut und deutlich besser als die anderen 3 Versionen abschneiden. Die NonMating-Version konnte zwar jedes Mal eine Lösung finden, benötigte aber deutlich mehr Zeit als die Versionen mit dem Kreuzungsoperator. In der Abbildung wird dies insbesondere durch die Größe der Fitnesssprünge der NonMating-Version deutlich. Dies lässt sich dadurch erklären, dass in den normalen Versionen mehrere Individuen, die unabhängig voneinander zu einem ähnlichen Zeitpunkt unterschiedliche Topologie- als Parameteroptimierungen gefunden haben, diese sehr schnell untereinander austauschen konnten. In der NoGrowth-Version hingegen musste jedes dieser Individuen unabhängig voneinander die Ergebnisse oder eine Alternative erneut selbst entdecken. Diese Ergebnisse bestätigen also in gewisser Weise die Diskussion, ob der Kreuzungsoperator einen sinnvollen Beitrag zur Neuroevolution leistet, wie sie im Kapitel 2.3.1 diskutiert wurde.

Die beiden Versionen InitRandom und NoGrowth hatten jedoch beide deutlich größere Probleme: Beide Versionen hatten Läufe, die einerseits nicht in der Lage waren, in der vollen Generationszeit eine Lösung zu finden, und andererseits, wenn sie eine Lösung fanden, diese deutlich länger brauchten, meist 3 mal so viele Generationen wie die normale Version. Die optimierten Lösungen, die mit dem vollständigen NEAT-Algorithmus gefunden wurden, waren meist in der Lage, mit weniger als 4 inneren Knoten eine Lösung zu finden, dass aber trotz teilweise 8 inneren Knoten wie in der NoGrowth-Version eine Lösung gefunden wurde, ist in Abbildung 12 zu sehen. Diese Abbildungen zeigen jeweils alle 5-Durchläufe der entsprechenden Version.

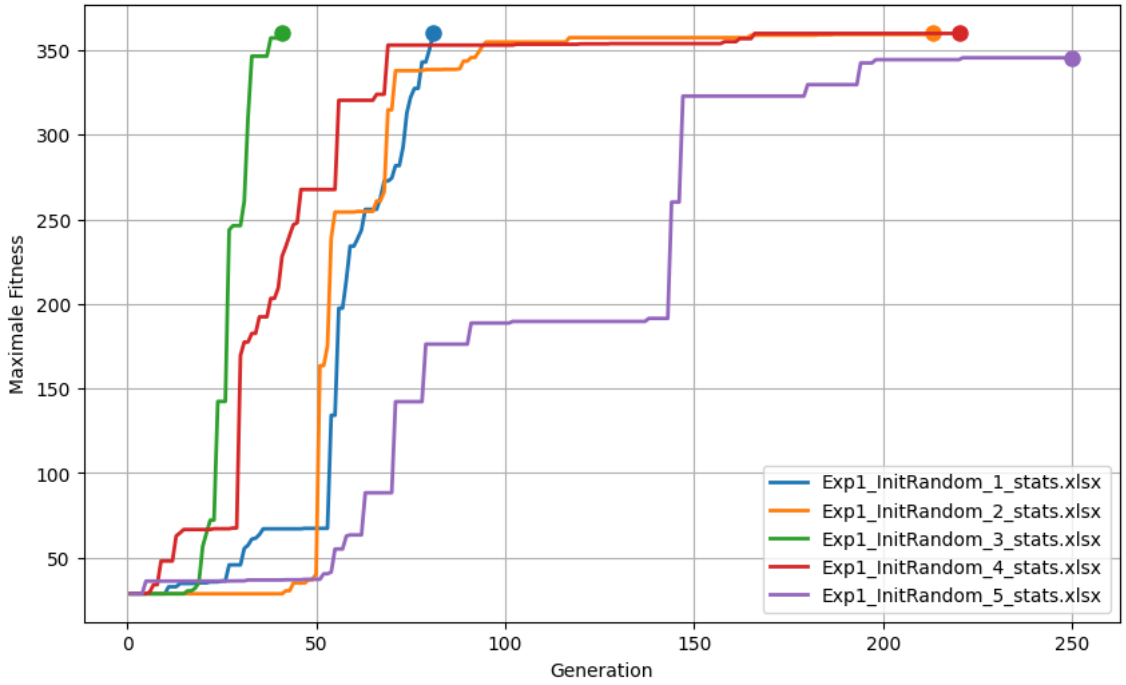


Abbildung 11: InitRandom

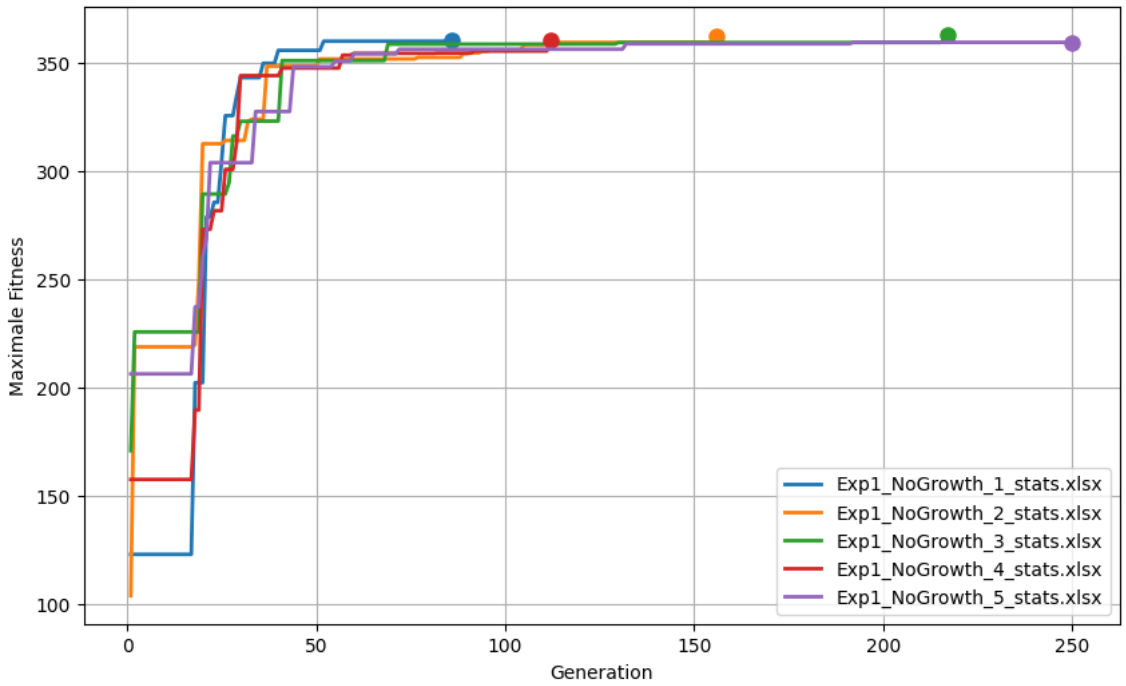


Abbildung 12: NoGrowth

Die InitRandom-Version wurde mit einer Anzahl von 4 inneren Knoten gestartet, was bereits zu einer merklichen Verlangsamung des Algorithmus geführt hat. Zudem ist die Mutationsrate für neue Verbindungen in den meisten Fällen deutlich höher als die Mutationsrate für Knoten, sodass davon auszugehen ist, dass dieser in einer akzeptablen Zeit angeschlossen werden kann. Die InitRandom-Version hat somit unsere Diskussionen bezüglich der Anfangspopulationen in Neuroevolution-Algorithmen aus den Kapiteln 2.3.2 , 2.4.3 und 3.6 bestätigt.

Die NoGrowth-Version war, wie bereits erwähnt, die einzige Version, die mit vollständigen Verbindungen startete, was man in den Abbildungen 10 und 12 sofort an der anfänglichen Fitness erkennen kann. Bereits nach wenigen Generationen wurde eine Fitness von über 200 erreicht, jedoch erwiesen sich diese Strukturen langfristig als eher schädlich als nützlich. Zum einen wurde der Algorithmus mit einem unnötig großen Suchraum konfrontiert, da die überflüssigen topologischen Strukturen ihre Daseinsberechtigung im Vorfeld nicht gerechtfertigt hatten. Zum anderen führten die überflüssigen Verbindungen zu einem deutlichen Anstieg der Rechenzeit, insbesondere im Vergleich zur Initrandom-Version (siehe Tabelle 8). Ganz zu schweigen davon, dass am Ende eine Lösung mit einer suboptimalen Topologie erhalten wurde. Die für diese Version gewählte Topologie sollte dennoch in der Lage sein, eine Lösung in ausreichender Zeit zu finden, wie der spätere Vergleich der Topologien mit den gefundenen Lösungen der anderen Versionen zeigt. Der bereits mehrfach kurz angesprochene menschliche Einfluss bei der Vorauswahl einer geeigneten Topologie ist bei neuronalen Netzen generell oft ein limitierender Faktor, der u.a. so groß ist, dass er zur Neuroevolution geführt hat.

Abschließend soll noch einmal kurz auf die Ergebnisse der besten Version eingegan-

gen werden. Die Ergebnisse der vollständigen und der NoSpeziation-Version sind nahezu identisch, so dass sich die Frage stellt, ob die Speziation wirklich so wichtig ist, wie in den Kapiteln 2.4.2 und 3.5 immer wieder diskutiert wurde. Dazu wird die mittlere Standardabweichung in jeder Generation (siehe Abbildung 13 betrachtet.

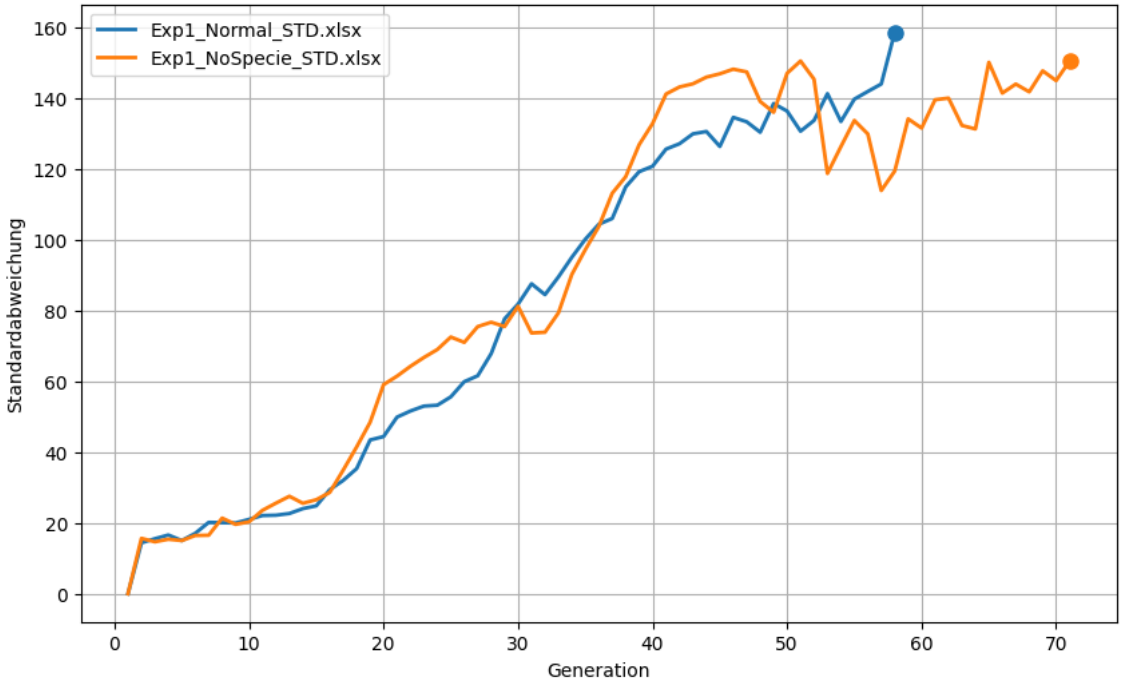


Abbildung 13: Standartabweichungen in den Version Normal und NoSpeziation

Auch diese sind nahezu identisch. Die Antwort liegt in der Topologie der gefundenen Lösungen. Diese sind, wie bereits erwähnt, jeweils sehr minimal, wobei sogar Lösungen gefunden wurden, die überhaupt keine inneren Knoten verwenden. Dies deutet darauf hin, dass die betrachtete Umgebung nicht komplex genug ist, um topologische Strukturen schützen zu müssen. Im Zusammenhang mit den theoretischen Ergebnissen wird die Existenz der Speziation im folgenden Kapitel näher begründet.

4.1.3 Bezug zu theoretischen Erkenntnissen

Stanley und Miikkulainen haben in ihrer Originalarbeit auch untersucht, inwieweit die einzelnen Komponenten zum gesamten NEAT-Algorithmus beitragen [SM02]. Dazu wurde der NEAT-Algorithmus mit den fünf bereits besprochenen Versionen getestet. Als Problem haben sie das Double Pole Balancing Problem betrachtet, bei dem zwei unterschiedlich lange Stangen auf einem beweglichen Wagen balanciert werden müssen. Der Wagen kann nur durch horizontale Kräfte nach links oder rechts bewegt werden, wobei die Positionen, Winkel und Geschwindigkeiten des Wagens und der Stäbe berücksichtigt werden. Ziel ist es, die Stangen in einer aufrechten Position zu halten, ohne dass der Wagen die vorgegebenen Grenzen verlässt. Dieses Problem dient als Benchmark für Algorithmen wie Reinforcement Learning und NeuroEvolution, da es komplexe Regelungs- und Stabilisierungsaufgaben beinhaltet [Sut18].

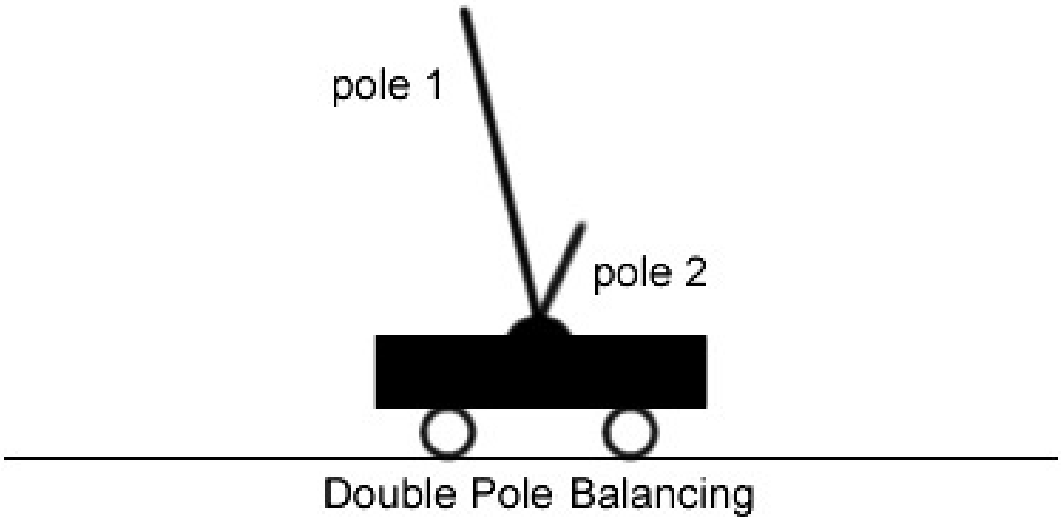


Abbildung 14: Double Pole Balancing Problem [Pro05]

Im Experiment versuchte jede Version 20 mal, das Problem zu lösen (mit Ausnahme der NonMating-Version, die 120 mal versuchte). Eine Lösung wurde akzeptiert falls diese in der Lage war über 100000 Zeitschritte (30 Minuten simulierter Zeit) die Stangen aufrecht zu erhalten. Dazu standen jeder Version maximal 1000 Generationen zur Verfügung. Aus diesen 20 Versuchen wurden die Erfolgsrate und die Anzahl der durchschnittlich benötigten Fitness-Evulationen berechnet. Die Ergebnisse sind in Tabelle 9 aufgeführt.

Method	**Evaluations**	**Failure Rate**
No-Growth NEAT (Fixed-Topologies)	30,239	80%
Non-speciated NEAT	25,600	25%
Initial Random NEAT	23,033	5%
Nonmating NEAT	5,557	0%
Full NEAT	3,600	0%

Tabelle 9: Ergebnisse aus [SM02]

Die Ergebnisse decken sich also weitgehend mit unseren bisherigen Beobachtungen. Unterschiede zu unseren Ergebnissen zeigen sich allerdings in der Version ohne Speziation. Dies ist auf die unterschiedliche Komplexität des betrachteten Problems zurückzuführen, wie vorher bereits diskutiert wurde. Es ist verständlich, dass der Schutz von Innovationen umso wichtiger ist, je komplizierter die Lösung des Problems ist.

Basierend auf dem gleichen Problem verglichen sie auch einige andere Vorläufer der Neuroevolution, die zu ihrer Zeit bekannt waren, mit dem vollständigen NEAT-Algorithmus. Die ersten beiden erwähnten Algorithmen sind grundlegende Version welche man sich unter Neuroevolution und Gentischen Algorithmen vorstellt. ESP (Enforced Subpopulations) teilt das Netzwerk in Subnetzwerke und optimiert jede Schicht separat (es handelt sich um einen Algorithmus mit fester Topologie), während SANE (Symbiotic, Adaptive Neuro-Evolution) Neuronen anstelle von kompletten Netzen evolviert und diese dy-

namisch zu kompletten Netzen kombiniert. Die Ergebnisse sind der Mittelwert von 50 Durchläufen bzw. 120 Durchläufen für NEAT.

Method	Evaluations	Generations	No. Nets
Ev. Programming	307,200	150	2048
Conventional NE	80,000	800	100
SANE	12,600	63	200
ESP	3,800	19	200
NEAT	3,600	24	150

Tabelle 10: Vergleich von Ev. Programming, Conventional NE, SANE, ESP und NEAT am Double Pole Balancing Problem with Velocities [SM02]

In einer etwas schwierigeren Version des Problems, bei der die Geschwindigkeit nicht übergeben wurde und diesmal eine Vielzahl von verschiedenen Startpositionen vorgegeben war, die zumeist vom Algorithmus für eine allerdings etwas kürzere Zeit gehalten werden mussten, wurde der NEAT-Algorithmus erneut mit ESP und dem Cellular Encoding-Algorithmus verglichen. Letzteres ist ein indirekter Kodierungsalgorithmus, der vor der Veröffentlichung von NEAT häufig verwendet wurde. Die Ergebnisse sind Mittelwerte aus über 20 Durchläufen.

Method	Evaluations	Generalization	No. Nets
CE	840,000	300	16,384
ESP	169,466	289	1,000
NEAT	33,184	286	1,000

Tabelle 11: Vergleich von CE, ESP und NEAT am Double Pole Balancing Problem without Velocities [SM02]

In beiden Experimenten zeigten Stantley und Miikkulainen, dass der NEAT-Algorithmus in der Lage ist, eine konsistente Lösung in weniger Zeit bzw. Durchläufen zu finden. Diese zeigten auch, dass NEAT immer in der Lage ist, das Problem mit einer geringeren Anzahl innerer Knoten zu lösen als die Fixed Topology Methoden. Insbesondere bei schwierigeren Problemen zeigt NEAT einen signifikanten Vorteil gegenüber ESP, da der NEAT-Algorithmus nie neu gestartet werden muss, während ESP sich bei Stagnation selbst neu startet. Dies zeigt, dass NEAT weniger von lokalen Maxima getäuscht wird.

Stanley demonstrierte zudem in [Sta04] eine Vielzahl von weiteren effektiven Anwendungen von NEAT. Zum Beispiel zeigte der NEAT-Algorithmus seine Fähigkeit, das Brettspiel Go auf kleinen Spielfeldern effizient zu spielen. Weiterhin wurde gezeigt, dass NEAT mit Hilfe der Roving-Eye-Technik und rekurrenter Verbindungen in der Lage ist, sein Vorwissen von kleinen Spielfeldern auf größere Spielfelder zu übertragen. Die Roving-Eye-Technik erlaubt es dem Agenten, zu jedem Zeitpunkt nur eine begrenzte Sicht auf das gesamte Spielfeld zu haben, während er sein Sichtfeld durch Bewegung dynamisch anpassen kann. Die wiederkehrenden Verbindungen fungieren als Gedächtnis des Netzes, so dass der Agent das Spielfeld schrittweise eingrenzen, relevante Informationen speichern und analysieren kann.

In einer anderen Anwendung zeigte Stanley, dass der NEAT-Algorithmus in der Rennsimulation RARS (= The Robot Auto Racing Simulator) in der Lage ist, Autos kollisionsfrei zu steuern und optimale Kurven in vergleichbarer Zeit wie andere vortrainierte Agenten zu fahren. Basierend auf diesen Experimenten wurde NEAT beigebracht, Kollisionen mit anderen Fahrzeugen und Objekten entlang der Strecke, wie zum Beispiel Schleudern, anhand der Fahrzeughistorie zu erkennen. Zusätzlich war NEAT in der Lage, sensorische Beeinträchtigungen durch rekurrente Verknüpfungen zu kompensieren.

Dies sind nur kleine Beispiele und Anwendungen welche die Effizienz des NEAT Algorithmus demonstrieren. Es gibt eine Vielzahl von weiteren Veröffentlichungen welche im Folgenden zum Teil noch erwähnt werden. Es wurden bereits viele Stärken von NEAT gezeigt, welche im Laufe der Arbeit noch deutlicher herausgestellt werden.

4.2 Projekt 2: Vergleich zu anderen Algorithmen

Im zweiten praktischen Projekt wird verglichen, wie der NEAT-Algorithmus ein Problem im Vergleich zu anderen klassischen, bewährten Algorithmen löst.
Die zu untersuchende Umgebung ist die Bipedalwalker-Umgebung der Python-Bibliothek Gymnasium [TKT+24]. In dieser Umgebung muss ein Agent (Walker) versuchen, sich in einer vorgegebenen Zeit auf einem leicht ungeraden Terrain fortzubewegen. Der Agent wird belohnt, wenn er sich auf dem Gelände vorwärts bewegt und kann bis zu 300 Punkte erreichen. Fällt der Agent hin, erhält er eine Strafe von 100 und die Umgebung wird vorzeitig beendet. Jede Bewegung kostet eine kleine Anzahl von Punkten, so dass der Agent einen Anreiz hat, sich so effizient wie möglich zu bewegen.

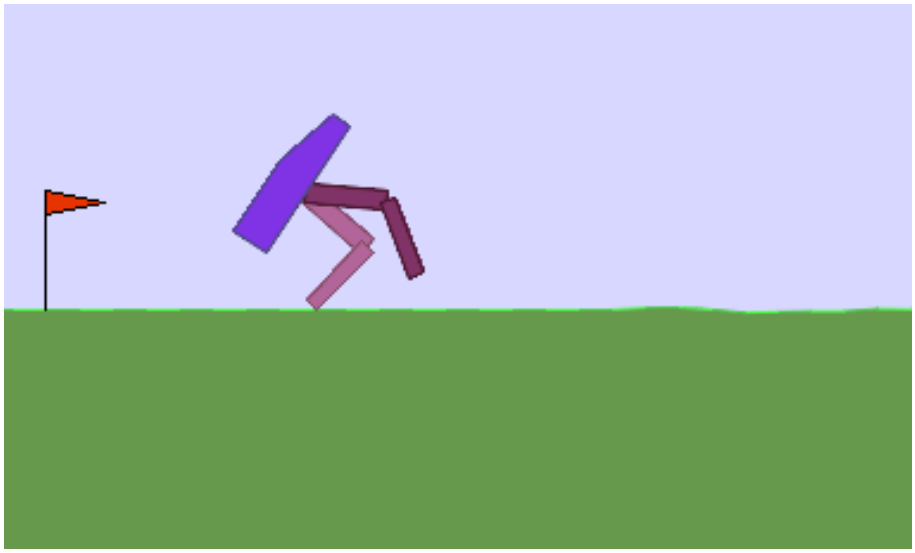


Abbildung 15: Die Bipedal Walker Umgebung

Der Status besteht aus der Rumpfwinkelgeschwindigkeit, der Winkelgeschwindigkeit, der horizontalen Geschwindigkeit, der vertikalen Geschwindigkeit, der Position der Gelenke und der Winkelgeschwindigkeit der Gelenke, dem Kontakt der Beine mit dem Boden und 10 Lidar-Entfernungsmesser-Messungen. Mithilfe dieser Beobachtungen muss der Agent

sinnvolle Bewegungsmuster erlernen um die vorher beschriebene Aufgabe zu lösen.

Im Folgenden wird die Leistungsfähigkeit der betrachteten Algorithmen innerhalb eines vorgegebenen Generations-/Zeitfensters verglichen. Das Problem gilt für jeden Algorithmus als gelöst, sobald er in der Lage ist, ein Individuum mit einer Fitness von mindestens 300 zu finden. Diese Fitness gibt an, dass das Individuum in der Lage ist, den Agenten innerhalb des Zeitfensters mit einer akzeptablen Geschwindigkeit und effizienten Bewegungsmustern zum Ende der Umgebung zu bewegen.

4.2.1 Erklärung der Implementierung

Im Gegensatz zu evolutionären Ansätzen wie NEAT, die auf der Optimierung von Netzwerkarchitekturen basieren, konzentrieren sich viele klassische Reinforcement-Learning-Algorithmen auf die Optimierung von Policies. Die Funktion $\pi(a \mid s, \theta)$ definiert die Aktionen a in Abhängigkeit der Zustände s sowie der Parameter der Policy-Funktion θ. Das Ziel dieser Algorithmen besteht in der Findung einer Policy, deren kumulierte erwartete Belohnung maximiert wird:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right],$$

Hier stellt $J(\theta)$ den erwarteten kumulierten Belohnungswert dar, wobei τ eine Trajektorie ist, die Zustände s_t, Aktionen a_t und Belohnungen $r(s_t, a_t)$ umfasst. Der Diskontfaktor γ gewichtet zukünftige Belohnungen relativ zu aktuellen Belohnungen. Wenn γ nahe bei 1 liegt, berücksichtigt der Agent auch langfristige Belohnungen stark. Wenn γ kleiner ist, fokussiert sich der Agent stärker auf kurzfristige Belohnungen.

Zur Optimierung der Policy π_θ verwenden diese Algorithmen *Policy-Gradient-Verfahren*, die den Gradienten von $J(\theta)$ berechnen um die Policy anpassen:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \hat{G}_t \right],$$

Dabei beschreibt $\nabla_\theta J(\theta)$ den Gradient der Policy-Parameter θ, der die Anpassung der Policy in Richtung besserer Belohnungen zeigt. Der Term $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ beschreibt, wie empfindlich die Wahl einer Aktion auf Änderungen der Parameter reagiert. $\hat{G}_t$ ist der geschätzte kumulierte Belohnungswert ab Zeitpunkt t, der verwendet wird, um den Wert der getroffenen Entscheidung zu quantifizieren. Diese Formel bildet die Grundlage vieler moderner Ansätze, die durch zusätzliche Mechanismen oder Stabilitätsregeln weiter verfeinert werden.

Zwei weit verbreitete Algorithmen, die auf diesen Prinzip basieren, sind der Proximal Policy Optimization (PPO) Algorithmus und Deep Deterministic Policy Gradient (DDPG) Algorithmus. Die beiden Algorithmen gehören zu den modernsten Techniken des klassischen Reinforcement Learning und eignen sich als Benchmark, um den NEAT-Algorithmus mit modernen Reinforcement-Learning-Ansätzen zu vergleichen. Eine detaillierte Erläuterung der beiden Algorithmen würde den Umfang dieser Arbeit überschreiten. Für weiterführende Informationen wird daher auf die Arbeiten von Schulman

et al. (2017) für PPO [SWD+17] sowie auf die von Lillicrap et al. (2019) für DDPG [LHP+19] verwiesen.

Beide Implementierungen wurden mit der Python-Bibliothek Stable Baseline [RHG+21] programmiert.

4.2.2 Analyse der Ergebnisse

Alle drei Algorithmen dauerten ca. 4 Stunden und 30 Minuten. Die geringen Laufzeitunterschiede ergeben sich aus der Anpassung an unterschiedliche Maßstäbe wie Generationen oder Schritte, um die Algorithmen vergleichbar zu machen.

Lauf	Generationen / Timesteps	Dauer (Minuten)
NEAT	1000 Generationen	270
DDPG	6400000 Timesteps	296
PPO	8500000 Timesteps	309

Tabelle 12: Dauer der Algorithmen in Minuten

Da der NEAT-Algorithmus der einzige ist, der das Problem während des Trainings kontinuierlich beendet und auswertet, müssen die anderen Algorithmen zeitweise unterbrochen werden, um geeignete Messungen für die Statistik zu entnehmen.[6]. Diese zusätzlichen Unterbrechungen verlängern die Gesamtlaufzeit entsprechend.

Die Ergebnisse sind dargestellt in Abbildung 16.

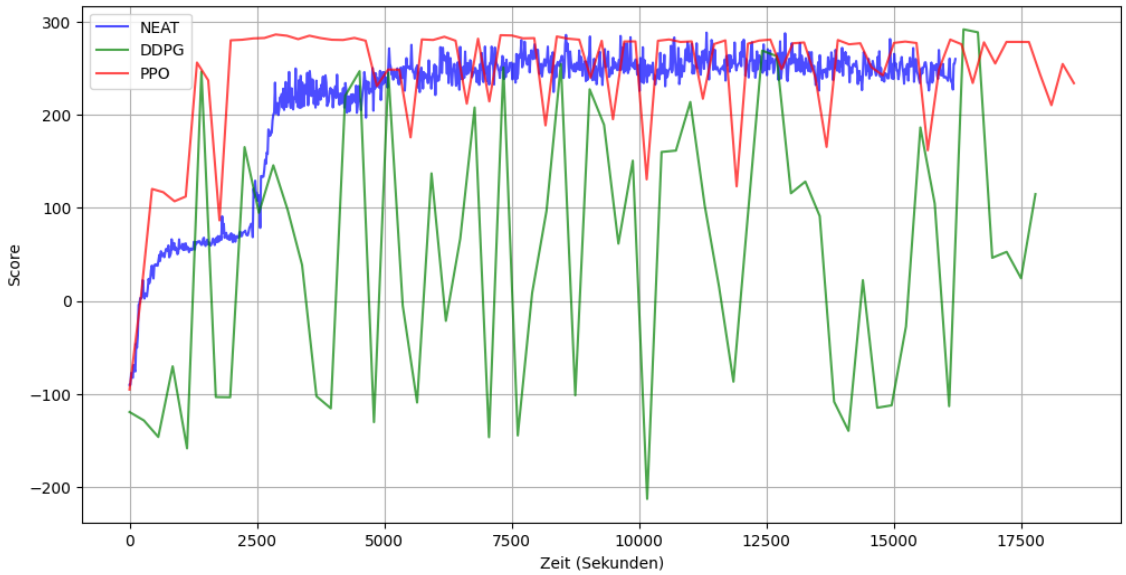


Abbildung 16: Ergebnisse des 2. Projektes

[6]Dies ist nur in meiner Implementierung der Fall, im Allgemeinen sind DDPG und PPO dazu in der Lage

Da in erster Linie das Erreichen der maximalen Fitness von Interesse ist, werden auch die Graphen für das Erreichen der maximalen Fitness geglättet, wie in Abbildung 17 dargestellt.

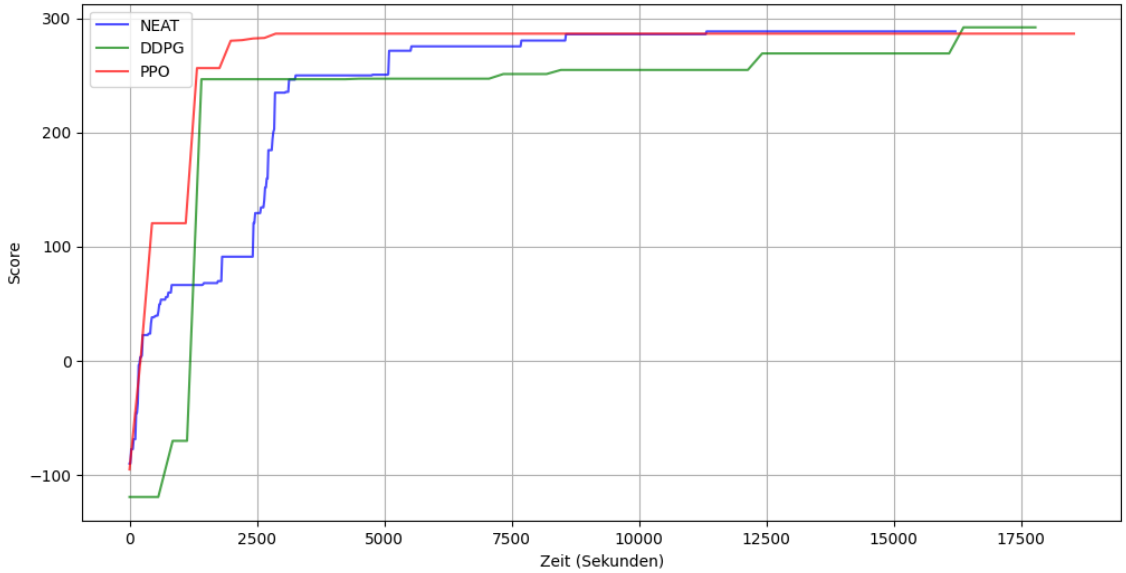


Abbildung 17: geglättete Ergebnisse des 2. Projektes

Die Ergebnisse zeigen, dass alle drei Algorithmen nicht in der Lage waren, die Zielfitness von 300 in der vorgegebenen Zeit zu erreichen, aber sie kamen ihr sehr nahe und hätten sie mit etwas mehr Rechenzeit durchaus erreichen können. Jeder Algorithmus war in der Lage, einen Agenten zu erzeugen, der in der Lage war, das Ende des Parcours zu erreichen, aber seine Bewegungen nicht vollständig optimiert hatte. Der NEAT-Algorithmus benötigt etwas mehr Zeit, um den ersten Sprung in seiner Fitness zu erreichen, ist aber wie DDPG in der Lage, kontinuierlich kleine Fortschritte in seiner Fitness zu machen. Insgesamt schneidet jedoch der PPO-Algorithmus am besten ab, da er in kürzester Zeit den ersten funktionsfähigen Agenten erzeugt, die anschließende längere Stagnation in der Fitness kann wahrscheinlich auf eine zu hoch gewählte Lernrate zurückgeführt werden.

Über die erzeugten topologischen Strukturen kann wenig gesagt werden, da die verschiedenen Algorithmen sehr unterschiedlich arbeiten. Der NEAT-Algorithmus war jedoch erneut in der Lage, seine Lösung sehr minimal zu halten und viele unnötige Verbindungen zu vermeiden.

Zusammenfassend kann man sagen, dass der NEAT-Algorithmus mit den modernen Giganten auf dem Gebiet des Reenforcement Learning mithalten kann und für viele Probleme seine eigene Nische gefunden hat. Darüber hinaus wird ein Problem betrachtet, das als allgemeiner Benchmark für verschiedene Algorithmen dient und daher nicht auf ein einzelnes Anwendungsgebiet beschränkt ist. In vielen Situationen, in denen eine optimale Topologie gefunden werden muss oder in denen die Belohnung der Umgebung (Fitness) schwer zu messen ist, kann NEAT seine Stärken noch besser ausspielen.

5 Untersuchung der Weiterentwicklungen

Über die Anzahl der 20 Jahre seitdem der ursprüngliche NEAT Algorithmus entwicklet wurde hat sich eine Vielzahl von Weiterentwicklung gebildet welche NEAT in verschiedenen Aspekten erweitern. In ihrer Arbeit „A Systematic Literature Review of the Successors of NeuroEvolution of Augmenting Topologies“ [PCJ21] fassen Papavasileiou, Cornelis und Jansen diese Erweiterungen zusammen und katalogisieren die meisten dieser (siehe Abbildung 18).

Im folgenden werden zwei dieser Erweiterungen genauer betrachtet.

5.1 HyperNEAT

Eine der interessantesten Weiterentwicklungen des NEAT-Algorithmus ist der bereits im Kapitel über Kodierungen angesprochene „Hypercube-based NeuroEvolution of Augmenting Topologies“ Algorithmus oder kurz HyperNEAT [SDG09] .

Künstliche Neuronale Netze sind zwar bemerkenswert leistungsfähig, doch ihre Komplexität kann nicht annähernd mit der astronomischen Komplexität des menschlichen Gehirns verglichen werden. Mit etwa 100 Billionen Verbindungen, deren kollektive Funktion von keinem künstlichen System erreicht wird, ist das menschliche Gehirn das komplexeste bekannte System [Gue00].

Die Komplexität des Gehirns zeigt sich in präzisen, sich wiederholenden Mustern, wie den kortikalen Säulen [Spo02], die im gesamten Gehirn vorkommen und oft Variationen eines gemeinsamen Themas darstellen. Im Gegensatz dazu erzeugen neuroevolutionäre Ansätze Netze, die erheblich weniger Neuronen und eine geringere Organisation aufweisen. Diese Unterschiede deuten darauf hin, dass aktuellen evolutionären Algorithmen etwas fehlt, um ähnliche Komplexität wie das natürliche Gehirn zu erreichen.

Die Forschung auf dem Gebiet der generativen und evolutiven Kodierung zeigt, dass große Strukturen in der DNA durch Wiederholung mittels genetischer Wiederverwendung kompakt dargestellt werden können [BK$^+$99]. Das bedeutet, dass wiederkehrende Strukturen durch ein einziges Gen beschrieben und in den Phänotyp übertragen werden. Andere wichtige Prinzipien wie die Geometrie des Gehirns und die lokale Konnektivität spielen ebenfalls eine wichtige Rolle, da Neuronen, die mit nahen Ereignissen verbunden sind, auch räumlich nah beieinander liegen.

Basierend auf diesen Ideen versucht der HyperNEAT Algorithmus eine höhere Komplexität und Struktur in künstlichen neuronalen Netzen durch indirekte Kodierung mithilfe von Compositional Pattern-Producing Networks (CPPNs) zu erreichen. CPPNs ermöglichen die Erzeugung komplexer Netzstrukturen durch die Wiederverwendung von Mustern. Anstatt jede Verbindung zwischen Neuronen direkt zu kodieren, beschreiben CPPNs die Funktionen, die die Verbindungen zwischen Neuronen erzeugen. Diese Funktionen können wiederverwendet werden, um große und komplexe Netze zu entwickeln, die auf den Prinzipien der biologischen Genetik basieren.

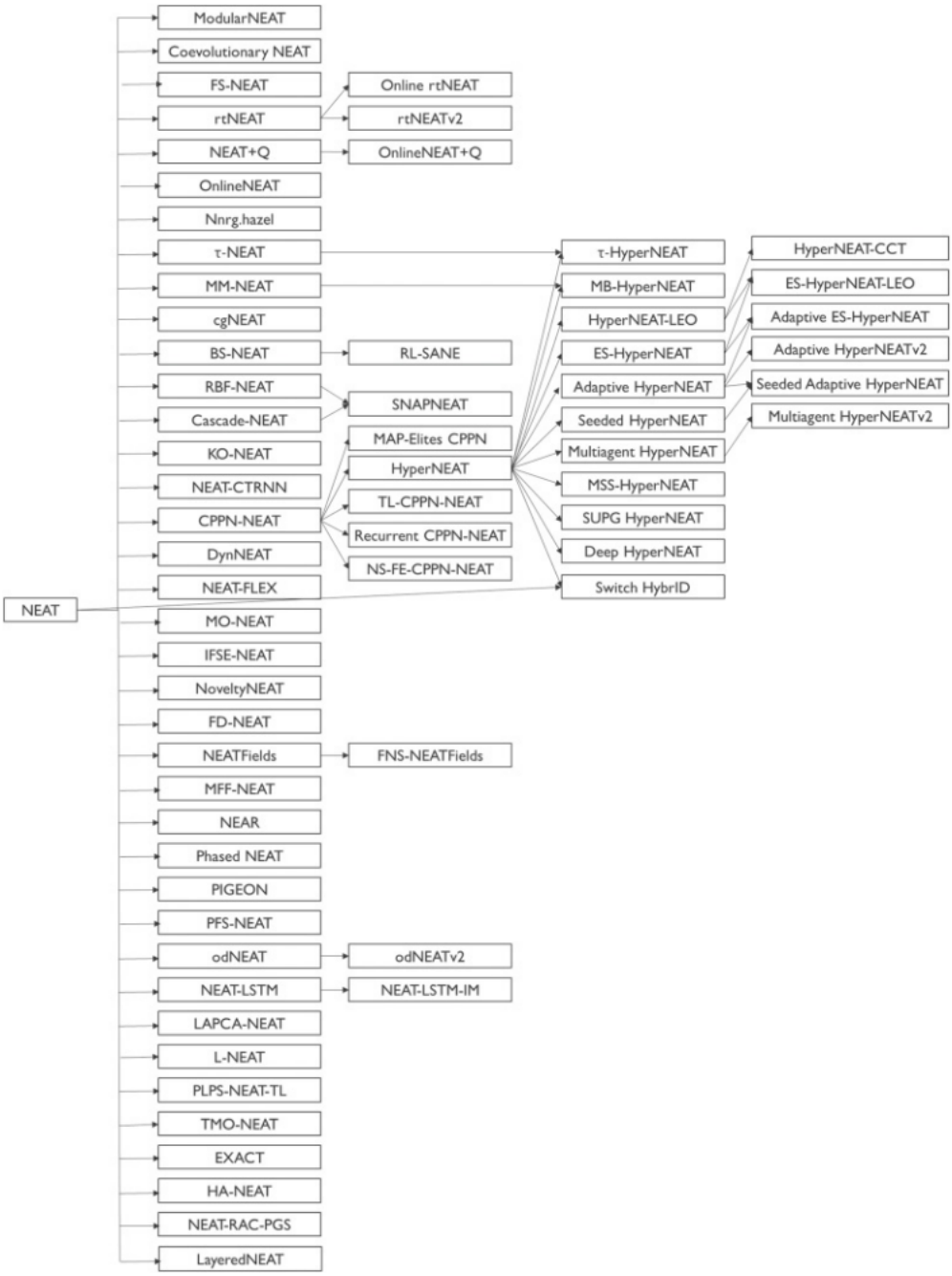


Abbildung 18: Der Entwicklungspfad von NEAT-Methoden. [PCJ21]

Der HyperNEAT-Algorithmus verwendet diese indirekte Kodierung, um die Geometrie und die räumlichen Beziehungen der physischen Welt nachzuahmen. Durch die Verwendung von CPPNs kann HyperNEAT Netze erzeugen, die nicht nur eine größere Anzahl von Neuronen und Verbindungen enthalten, sondern auch eine höhere Organisation und Regelmäßigkeit aufweisen. Dies ermöglicht die Generierung von Netzen, die in ihrer Struktur und Funktion komplexer sind, als dies durch direkte Kodierung allein möglich wäre.

HyperNEAT hat sich daher als besonders effektiv bei der Modellierung komplexer neuronaler Netze mit räumlicher Struktur und bei der Lösung von Aufgaben erwiesen, die eine hohe Netzkomplexität erfordern. Diese Fähigkeiten machen HyperNEAT besonders geeignet für Herausforderungen in der Robotik, wie z.B. von Clune et al. [LYG+13] in ihrer Studie „Evolving Gaits for Physical Robotswith the HyperNEAT Generative Encoding:The Benefits of Simulation“ gezeigt wurde. In dieser Studie waren Clune et al. in der Lage mit Hilfe von HyperNEAT koordinierte und symmetrische Bewegungsmuster für vierbeinige Roboter zu entwickeln. Insbesondere konnte HyperNEAT automatisch geometrische Regelmäßigkeiten wie Links-Rechts-Symmetrie und modulare Wiederholungen erkennen, die für eine natürliche und effiziente Fortbewegung entscheidend sind, ohne dass diese explizit vorgegeben werden mussten. Darüber hinaus zeigte HyperNEAT seine Stärken in der Studie von Hausknecht et al. [HKMS12], in der eine generische Spielsteuerung für Atari-2600-Spiele entwickelt wurde. HyperNEAT erkannte und nutzte effektiv Symmetrien und andere geometrische Regelmäßigkeiten der 2D-Spieloberfläche, was in den getesteten Spielen Freeway und Asterix zu besseren Ergebnissen führte als Reinforcement-Learning-Ansätze wie Sarsa(λ).

5.2 rtNEAT

Eine weitere Weiterentwicklung von NEAT ist die sogenannte rtNEAT (Real-Time NeuroEvolution of Augmenting Topologies), die dazu dient, neuronale Netze in Echtzeit zu evolvieren [SBM05]. Zu Beginn wird eine Population von neuronalen Netzen zufällig erzeugt. Jedes Netz hat eine einfache Struktur, oft bestehend aus Eingangs- und Ausgangsschichten, ähnlich wie im klassischen NEAT.

Nach jeweils n Ticks des Programms werden die folgenden Schritte ausgeführt:

1. Entferne den Agenten mit der niedrigsten angepassten Fitness, vorausgesetzt, er hatte genügend Zeit für eine zuverlässige Bewertung.

2. Berechne die neue durchschnittliche Fitness nach dynamischer Anpassung der Populationsgröße (wie in NEAT) für alle Spezies.

3. Wähle eine Spezies, die die Elternindividuen für die Fortpflanzung liefert, und generiere daraus ein neues Individuum unter Berücksichtigung der Mutation.

4. Passe die Kompatibilitätsschwelle dynamisch an und weise die Agenten den Spezies neu zu.

5. Platziere den neu generierten Agenten in der simulierten Welt.

Diese Schritte ermöglichen es, die Leistung der Agenten während des Programmablaufs kontinuierlich zu optimieren und sie reibungslos in Echtzeit durch bessere Agenten zu ersetzen. Dies geschieht ohne Unterbrechung des Systems, was besonders in dynamischen und interaktiven Umgebungen von Vorteil ist, da sich die Agenten während ihrer Laufzeit an neue Gegebenheiten anpassen können.

Die Effektivität von rtNEAT wurde in der Studie von Olesen et al. [OYH08] demonstriert, in der der Algorithmus zur dynamischen Anpassung einer künstlichen Intelligenz in einem Echtzeit-Strategiespiel (RTS) eingesetzt wurde. Die Anpassung der Stärke der KI ist gerade bei RTS von großer Bedeutung, da der Spieler in jedem Moment mit einer Vielzahl von Möglichkeiten und Entscheidungen konfrontiert ist und somit sehr schnell der KI unterlegen sein kann. Gerade in solchen Szenarien ist eine dynamische Anpassung der KI notwendig, um eine ausgewogene Herausforderung zu bieten und den Spieler nicht zu überfordern. Die Autoren weisen jedoch darauf hin, dass viele Entscheidungen in RTS-Spielen, wie z.B. das Ressourcenmanagement oder der Bau von Gebäuden, nicht direkt mit dem Gegner zusammenhängen, was die Anpassung der KI erschwert. Sie kommen jedoch zu dem Schluss, dass rtNEAT noch effektiver in anderen Spielen eingesetzt werden kann, die eine intensivere Interaktion zwischen Spieler und Gegner ermöglichen, wie z.B. Ego-Shooter, Kampfspiele oder 2D-Actionspiele.

6 Abschluss

6.1 Stärken und Schwächen von NEAT

Der NEAT-Algorithmus hat sich im Laufe der Zeit als einer der bedeutendsten und einflussreichsten Algorithmen in der Neuroevolution etabliert. Durch seine innovative Herangehensweise hat er viele zuvor ungeklärte Fragen in diesem Forschungsbereich beantwortet und bietet wertvolle Einblicke in die Optimierung und Entwicklung neuronaler Netze. Im folgenden werden sowohl die Stärken als auch die Schwächen von NEAT und der Neuroevolution im Allgemeinen betrachten, um ein umfassendes Verständnis ihrer Leistungsfähigkeit und Herausforderungen zu gewinnen.

In ihrer Abhandlung [GM21] heben Edgar Galv'an und Peter Mooney die Vorzüge der Neuroevolution hervor und betonen insbesondere die Stärken des NEAT-Algorithmus. Zu den wesentlichen Vorteilen der Neuroevolution zählt die automatische Optimierung, welche die Konfiguration und das Training von Deep Neural Networks (DNNs) erheblich erleichtert.

Ein weiteres herausragendes Merkmal der Neuroevolution ist ihre hohe Flexibilität, wodurch sich diese Methode von anderen, weniger anpassungsfähigen Verfahren abhebt. Die Fähigkeit evolutionärer Algorithmen (EAs), verschiedene Elemente und Varianten zu kombinieren, erlaubt die Identifikation optimaler Netzarchitekturen. Diese Flexibilität erlaubt es Forschern, unterschiedliche Repräsentationen von neuronalen Netzen zu nutzen, einschließlich Anpassungen an Netzarchitekturen, Aktivierungsfunktionen und Lern-

strategien. Des Weiteren besteht die Möglichkeit, evolutionäre Algorithmen mit anderen Algorithmen zu kombinieren, um hybride Ansätze zu entwickeln. Die Kombination der Stärken beider Methoden resultiert in einer Effizienzsteigerung hinsichtlich der Lösungssuche.

Ein wesentlicher Vorteil der Neuroevolution besteht in der Fähigkeit, mehrere Lösungen simultan zu explorieren. Dies resultiert in einer umfassenderen Untersuchung des Suchraums und einer erhöhten Wahrscheinlichkeit, optimale oder nahezu optimale Lösungen zu identifizieren. Die Anpassungsfähigkeit und Vielseitigkeit machen EAs zu einem wertvollen Werkzeug in der Neuroevolution, insbesondere in komplexen und dynamischen Anwendungsbereichen, in denen die Grenzen der Anwendbarkeit traditioneller Methoden erreicht werden.

Des Weiteren ermöglicht die Neuroevolution eine effiziente Parallelisierung. Die simultane Untersuchung mehrerer Lösungen, welche durch evolutionäre Algorithmen ermöglicht wird, resultiert in einer gesteigerten Effizienz und einer reduzierten Wahrscheinlichkeit, sich in lokalen Optima zu verharren. Diese Eigenschaft ist auf die populationsbasierte Methodik zurückzuführen, auf der die evolutionären Algorithmen basieren. In einer Vielzahl von realen Anwendungsszenarien hat sich die Neuroevolution als wettbewerbsfähig gegenüber anderen Methoden der künstlichen Intelligenz erwiesen und liefert in vielen Fällen beeindruckende Ergebnisse.

Allerdings sind auch erhebliche Herausforderungen zu bewältigen. Die Evaluierung der Lösungen ist mit einem signifikanten Rechenaufwand verbunden, insbesondere bei großen Populationen und umfangreichen Datensätzen, was zu langen Trainingszeiten führt. Des Weiteren wird die Neuroevolution mitunter als langsamer Lerner kritisiert, da die Konvergenzgeschwindigkeit im Vergleich zu anderen Optimierungsverfahren geringer sein kann. Ein weiterer Nachteil ist das Fehlen standardisierter Studien in diesem Bereich, was die eindeutige Schlussfolgerung über die Effizienz der verschiedenen Techniken erschwert.

6.2 Zusammenfassung

In dieser Bachelorarbeit wurde der NEAT-Algorithmus ausführlich vorgestellt und untersucht. Nach einer kurzen Einführung in die grundlegende Funktionsweise von genetischen Algorithmen und künstlichen neuronalen Netzen wurde das Gebiet der Neuroevolution vorgestellt, das genetische Algorithmen verwendet, um KNNs sowohl hinsichtlich ihrer Parameter als auch ihrer Topologie zu optimieren. Anschließend wurde eine Vielzahl von Problemen, die in der Neuroevolution gelöst werden müssen, beleuchtet. Dies war notwendig, um die Bedeutung des NEAT-Algorithmus zu seiner Zeit zu verstehen. Anschließend wurde der NEAT-Algorithmus mit seinen einzelnen Komponenten betrachtet und untersucht, wie er die oben genannten Probleme löst. Das bekannte Competing Convention Problem wurde durch die Wahl einer genetischen Kodierung mit Hilfe von Innovationsnummern gelöst. Diese Innovationsnummern halfen, den Ursprung jedes Gens im Genotyp zu bestimmen, um eine effiziente Kreuzung verschiedener Genotypen zu ermöglichen. Außerdem wurde untersucht, wie NEAT durch explizites Fitness Sharing sicherstellt, dass topologische Innovationen parallel zu den Parametern optimiert

werden können, ohne vorzeitig verloren zu gehen. Zusätzlich wurde ein Blick auf die Ausgangspopulation geworfen, um zu sehen, wie NEAT eine signifikante Leistungssteigerung durch eine minimale Starttopologie und inkrementelles Wachstum erreichen kann.

In den praktischen Projekten wurde die Effizienz der zuvor einzeln betrachteten Komponenten überprüft, um zu sehen, ob tatsächlich jede einzelne einen positiven Beitrag zum Algorithmus leistet. Dazu wurden unterschiedlich modifizierte Versionen von NEAT betrachtet, die jeweils eine signifikante Komponente entfernen und zeigen, wie diese Version schlechter abschneidet als die vollständige Version. Diese Ergebnisse wurden mit theoretischen Ergebnissen verglichen, um diese zu bestätigen. Zusätzlich wurde in einem praktischen Projekt der NEAT-Algorithmus mit den modernen gradientenbasierten Verfahren PPO und DDPG verglichen, um zu sehen, wie dieser eine vergleichbare Leistung erzielt.

Am Ende der Arbeit wurden die Erweiterungen HyperNEAT und rtNEAT betrachtet, um das Potential des NEAT-Algorithmus weiter zu beleuchten. Dieses Potential für die Zukunft wurde dann zusammen mit den Stärken und Schwächen von NEAT im Fazit zusammengefasst.

6.3 Ausblick

In seiner Dissertation „Efficient Evolution of Neural Networks through Complexification“ präsentiert Stanley den NEAT-Algorithmus als bedeutenden Fortschritt in der Neuroevolution. Die zentrale Innovation liegt in der Einführung historischer Markierungen, die eine präzise genetische Verfolgung ermöglichen und die Grundlage für eine kontrollierte Kreuzung genetisch unterschiedlicher Netze bilden. Dies führt zu einer vielfältigen Population, in der Innovationen durch Speziation geschützt werden.

NEAT beginnt mit minimalen Netzen und fügt schrittweise Struktur hinzu. Dadurch wird nicht nur die Suche im Lösungsraum effizienter, sondern auch die Komplexität der Netze dynamisch an die Problemstellung angepasst. Diese Effizienz zeigt sich sowohl in den theoretischen Analysen als auch in den praktischen Projekten dieser Arbeit, wie in Kapitel 4.1 und Kapitel 4.2 demonstriert.

NEAT legt den Grundstein für zukünftige Entwicklungen, die signifikante Fortschritte in einer Vielzahl von Anwendungen, von der Robotik bis hin zu realen Problemlösungen, versprechen. Ein minimaler Einblick in diese Entwicklungen und ihre vielversprechenden Möglichkeiten wurde bereits im Kapitel 5 gegeben. Darüber hinaus bietet NEAT aufgrund seiner Flexibilität ein unglaubliches Potential in Kombination mit Deep Learning Ansätzen (z.B. DDPG aus Kapitel 4.2), um die Vorteile beider Welten zu vereinen.

Die Frage, ob Neuroevolution und der NEAT-Algorithmus eine Brücke zwischen biologischer Evolution und künstlicher Intelligenz schlagen können, bleibt spannend. Mit zunehmender Rechenleistung und fortschreitender Algorithmenentwicklung bietet NEAT eine vielversprechende Grundlage, um die Grenzen der künstlichen Intelligenz weiter zu verschieben.

Literatur

[ASP93] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65, 1993.

[BK+99] P. J. Bentley, S. Kumar, et al. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *GECCO*, volume 99, pages 35–43, 1999.

[BLB+15] Y. Bengio, D.-H. Lee, J. Bornschein, T. Mesnard, and Z. Lin. Towards biologically plausible deep learning. *arXiv Preprint arXiv:1502.04156*, 2015.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[BW93] H. Braun and J. Weisbrod. Evolving neural feedforward networks. In Rudolf F. Albrecht, Colin R. Reeves, and Nigel C. Steele, editors, *Artificial Neural Nets and Genetic Algorithms*, pages 25–32, Vienna, 1993. Springer Vienna.

[Cou02] R. Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. Theses, Institut National Polytechnique de Grenoble - INPG, June 2002.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[DM92] D. Dasgupta and D. R. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96. IEEE, 1992.

[Fog06] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. John Wiley & Sons, 2006.

[FU00] D. Floreano and J. Urzelai. Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines. *IEEE Transactions on Evolutionary Computation*, 4(3):284–298, 2000.

[GM21] E. Galván and P. Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2(6):476–493, 2021.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[Gue00] J.-M. Guerit. Fundamental neuroscience. *Neurophysiologie Clinique/Clinical Neurophysiology*, 30(2):58, 2000.

[Hay98] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1998.

[HKMS12] M. Hausknecht, P. Khandelwal, R. Miikkulainen, and P. Stone. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO '12)*, pages 217–224, New York, NY, USA, 2012. ACM.

[Hol92] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[HP04] G. S. Hornby and J. B. Pollack. Functional scalability through generative representations: The evolution of table designs. *Environment and Planning B: Planning and Design*, 31(4):570–571, 2004.

[Kit90] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[Koo05] E. V. Koonin. Orthologs, paralog, and evolutionary genomics. *Annu. Rev. Genet.*, 39(1):311–312, 2005.

[KS05] Y. Kassahun and G. Sommer. Efficient reinforcement learning through evolutionary acquisition of neural topologies. In *ESANN*, pages 259–266, 2005.

[LHP+19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2019.

[LHPS22] Y. Li, Y. Huang, G. F. Pedersen, and M. Shen. Recurrent neat assisted 2d-doa estimation with reduced complexity for satellite communication systems. *IEEE Access*, 10:11551–11563, 2022.

[LYG+13] S. Lee, J. Yosinski, K. Glette, H. Lipson, and J. Clune. Evolving gaits for physical robots with the hyperneat generative encoding: The benefits of simulation. In A. I. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 540–549. Springer Berlin Heidelberg, 2013.

[Mah95] S. W. Mahfoud. *Niching Methods for Genetic Algorithms*. University of Illinois at Urbana-Champaign, 1995.

[MKM+22] A. McIntyre, M. Kallada, C. G. Miguel, C. Feher de Silva, and M. L. Netto. neat-python. `https://github.com/CodeReclaimers/neat-python`, 2022. Accessed: 2024-09-07.

[MLM+17] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat. Evolving deep neural networks, 2017.

[OYH08] J. K. Olesen, G. N. Yannakakis, and J. Hallam. Real-time challenge balance in an rts game using rtneat. In *2008 IEEE Symposium on Computational Intelligence and Games*, pages 87–94. IEEE, 2008.

[PCJ21] E. Papavasileiou, J. Cornelis, and B. Jansen. A systematic literature review of the successors of "neuroevolution of augmenting topologies". *Evolutionary Computation*, 29(1):1–73, 2021.

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. *Proceedings of the Neural Information Processing Systems (NeurIPS)*, 30:1–6, 2017. Available online: `https://arxiv.org/abs/1709.04906`.

[Pro05] ANJI Project. Double pole balancing problem with velocities, 2005. Accessed: 2024-09-06.

[Rad93] N. J. Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing and Applications*, 1(1):67–90, 1993.

[RHG+21] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[Rin94] M. B. Ring. *Continual Learning in Reinforcement Environments*. The University of Texas at Austin, 1994.

[RRR14] R. E. Ricklefs, R. Relyea, and C. Richter. *Ecology: The Economy of Nature*, volume 7. WH Freeman New York, 2014.

[SBM05] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. Evolving neural network agents in the nero video game. *Proceedings of the IEEE*, pages 182–189, 2005.

[SDG09] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

[SGH95] J. L. Spears, D. F. Goldberg, and D. L. Hull. Speciation in genetic algorithms. In *Proceedings of the 1995 IEEE International Conference on Evolutionary Computation*, pages 345–350, Piscataway, NJ, USA, 1995. IEEE.

[SM02] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[SMC+17] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv Preprint arXiv:1712.06567*, 2017.

[Spo02] O. Sporns. Network analysis, complexity, and brain function. *Complexity*, 8(1):56–60, 2002.

[Sta04] K. O. Stanley. *Efficient Evolution of Neural Networks Through Complexification*. The University of Texas at Austin, 2004.

[Sta13] K. Stanley. Kenneth stanley's neat code reorganized (tidier?), 2013. Accessed: 2024-09-27.

[Sut18] R. S. Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018.

[SWD+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[TET12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[Thi96] D. Thierens. Non-redundant genetic coding of neural networks. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 571–575, Piscataway, New Jersey, 1996. IEEE Press.

[TKT+24] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[VL+91] M. D. Vose, G. E. Liepins, et al. Punctuated equilibria in genetic search. *Complex Systems*, 5(1):40, 1991.

[Wri91] Alden H Wright. Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms*, volume 1, pages 205–218. Elsevier, 1991.

[WSB90] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14(3):347–361, 1990.

[Yao99] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[YL96] X. Yao and Y. Liu. Evolutionary programming made faster. *IEEE Transactions on Evolutionary Computation*, 1(1):82–102, 1996.

[ZM+93] B.-T. Zhang, H. Muhlenbein, et al. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex Systems*, 7(3):199–220, 1993.