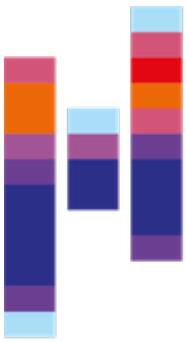

Hochschule
Flensburg
University of
Applied Sciences

# Neuroevolution mit MPI

## Analyse und Optimierung von NEAT für ein verteiltes System

**Masterthesis**
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang Angewandte Informatik
an der Hochschule Flensburg

**Simon Hauck**
Matrikelnummer: 660158

Erstprüfer: Prof. Dr. rer. nat. Tim Aschmoneit
Zweitprüfer: Prof. Dr. rer. nat. Torben Wallbaum

9. August 2020

Neuroevolutionäre Algorithmen sind ein mögliches Optimierungsverfahren für neuronale Netze. Abhängig von dem verwendeten Algorithmus können die Gewichte der Verbindungen im Netz und die Struktur entwickelt und optimiert werden.
Der Optimierungsprozess ist, unabhängig vom Verfahren, sehr aufwändig und dementsprechend zeit- und rechenintensiv. Für eine schnellere Durchführung des Trainingsprozesses bieten sich Algorithmen an, die gut parallelisierbar sind. Die benötigte Ausführungszeit dieser kann durch Hinzufügen weiterer Rechenknoten mit geringem Aufwand maßgeblich reduziert werden.
Neuroevolutionäre Algorithmen bieten sich aufgrund der Verfahrensweise und der vielen unabhängigen neuronalen Netzen für eine parallele Ausführung an.
In dieser Arbeit wird, stellvertretend für neuroevolutionäre Algorithmen, der *NeuroEvolution of Augmenting Topologies* (NEAT) Algorithmus betrachtet. Dieser wurde im Jahr 2002 veröffentlicht und ist im Vergleich zu den damals bekannten Algorithmen besonders effizient. Zudem dient der Algorithmus als Grundlage für viele Erweiterungen. Die erhaltenen Ergebnisse dieser Arbeit lassen sich somit gut auf ebendiese Erweiterungen übertragen.
Im ersten Schritt dieser Arbeit wird die Laufzeit des NEAT Algorithmus mit verschiedenen Optimierungsaufgaben analysiert. Mit den erhaltenen Ergebnissen wird eine parallelisierte Implementierung erstellt. Diese führt mit unterschiedlich vielen Rechenknoten dieselben Optimierungsaufgaben durch. Am Ende dieser Arbeit werden die Ergebnisse von beiden Implementierungen verglichen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Akronymverzeichnis

**NEAT** *NeuroEvolution of Augmenting Topologies*

**KNN** Künstliche neuronale Netze

**PNS** Periphere Nervensystem

**ZNS** Zentrale Nervensystem

**tanh** Tangens Hyperbolicus

**TWEANN** *Topology and Weight Evolving Artificial Neural Network*

# 1 Motivation

## 1.1 Problemstellung

## 1.2 Ziel der Arbeit

## 1.3 Struktur der Arbeit

# 2 Grundlagen

## 2.1 Neuronale Netze

Klassische Algorithmen in der Informatik beschreiben, mit welchen Schritten ein spezielles Problem gelöst werden kann. In vielen Anwendungsfällen, wie zum Beispiel beim Sortieren einer Liste, verwenden Computersysteme diese und lösen das gegebene Problem schneller und effizienter als es Menschen möglich ist.

Dennoch gibt es Aufgaben, die von Menschen ohne Aufwand gelöst werden, aber Computersysteme vor große Herausforderungen stellen. Hierzu zählt unter anderem die Klassifizierung von Bildern. Ein Mensch kann zum Beispiel Bilder von Hunden und Katzen unabhängig von Blickwinkel und Bildqualität unterscheiden beziehungsweise richtig zuordnen. Trotzdem lassen sich für solche Probleme keine klassischen Algorithmen finden, da die Lösung von vielen subtilen Faktoren abhängig ist [1].

In vielen dieser Aufgabenfelder werden Künstliche neuronale Netze (KNN) eingesetzt, welche von biologischen neuronalen Netzen inspiriert sind und zum Forschungsgebiet des maschinellen Lernens gehören. Auch wenn die KNN heute aktuell sind und viel Aufmerksamkeit erhalten, ist die Grundlage die Arbeit von McCulloch und Pitts, welche 1943 ein einfaches neuronales Netz mit Schwellwerten entwickelt haben. Dies ermöglicht die Berechnung von logischen und arithmetischen Funktionen [2]. In den folgenden Jahrzehnten wurde die Funktionsweise der neuronalen Netze weiterentwickelt und der Einsatz in verschieden Aufgabenfeldern ermöglicht. Hierzu zählen neben der Klassifizierung von Bildern [3] unter anderem das Erkennen und die Interpretation von Sprache [4], [5] sowie das selbständiges Lösen von Computer- und Gesellschaftsspielen [6], [7].

In diesem Kapitel wird zuerst ...

### 2.1.1 Biologische neuronale Netze

Wie bereits beschrieben, orientiert sich das Fachgebiet der KNN an den erfolgreichen biologischen neuronalen Netzen, wie zum Beispiel dem menschlichen Gehirn [1]. In diesem Abschnitt werden die Eigenschaften betrachtet, die das Vorbild erfolgreich machen und für die KNN übernommen werden sollen. Im Zuge dessen wird ein grober Überblick über die Struktur und Funktionsweise des menschlichen Gehirns gegeben.

Jede Sekunde erfassen die Rezeptoren des menschlichen Körpers unzählige Reize, wie zum Beispiel Licht, Druck, Temperatur und Töne. Die Reize werden anschließend elektrisch oder chemisch kodiert und über Nervenbahnen an das Gehirn geleitet, welches die Aufgabe hat, diese zu filtern, zu verarbeiten und entsprechend zu reagieren. Als Reaktion können

zum Beispiel Signale an entsprechende Muskeln oder Drüsen gesendet werden [8].

Bevor im nächsten Kapitel die Funktionsweise des Gehirns näher betrachtet wird, werden hier zunächst drei Eigenschaften beschrieben, die klassische Algorithmen entweder nicht besitzen oder nur schwer umsetzen können, aber für biologische neuronale Netze keine Herausforderung sind. Ziel ist es, diese mit den KNN umzusetzen [1].

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

2. **Fähigkeit zur Generalisierung**
   Allerdings kann nicht jedes mögliche Szenario für ein Aufgabenfeld durch Ausprobieren oder Beobachtung gelernt werden. Trotzdem trifft das Gehirn in den meisten Situationen plausible Lösungen, da es die Fähigkeit zur Generalisierung besitzt [1]. Das bedeutet, dass viele Situationen bereits bekannten Problemen zugeordnet werden können, mithilfe derer eine passende Verhaltensstrategie ausgewählt wird.

3. **Toleranz gegenüber Fehlern**
   Die Fähigkeit zu Generalisieren erlaubt auch eine hohe Fehlertoleranz gegenüber verrauschten Daten. Bei oben genanntem Beispiel der Klassifizierung von Bildern kann ein Teil des Bildes fehlen oder unscharf sein, trotzdem kann das abgebildete Motiv richtig zugeordnet werden.

### Struktur des menschlichen Gehirns

Das Forschungsgebiet der Neurowissenschaften befasst sich unter anderem mit dem menschlichen Gehirn, dessen Funktionsweise auch heute noch nicht vollständig erforscht ist. Dennoch ist schon seit 1861 durch die Arbeit von Paul Broca bekannt, dass es im menschlichen Gehirn verschiedene Regionen mit unterschiedlichen Aufgaben gibt [9]. Zum Beispiel wird das sogenannte Kleinhirn (Cerebellum) für einen Großteil der motorischen Koordination verwendet während an das Großhirn (Telencephalon) unter anderem visuelle Reize geleitet werden [1]. Trotz der unterschiedlichen Aufgaben haben alle Bereiche des Gehirns einen gemeinsamen Grundbaustein, die sogenannten Neuronen [9]. Im folgenden wird der Aufbau und die Funktionsweise von diesen oberflächlich im Bezug zu den später vorgestellten künstlichen Neuronen betrachtet. Für einen vollständigen Überblick und eine genaue Beschreibung der Vorgänge wird auf entsprechende Fachliteratur verwiesen.
Das menschliche Gehirn besitzt ungefähr $10^{11}$ einzelne Neuronen, deren schematischer Aufbau in Abbildung 2.1 dargestellt ist. Jedes Neuron besitzt einen Zellkern, der sich im Zellkörper (Soma) befindet. Von dem Zellkörper gehen mehrere Fasern aus, die Dendriten genannt werden [9]. An diesen befinden sich Synapsen, welche als Übertragungsstelle fungieren und elektrische oder chemische Signale von Rezeptoren oder anderen Neuronen

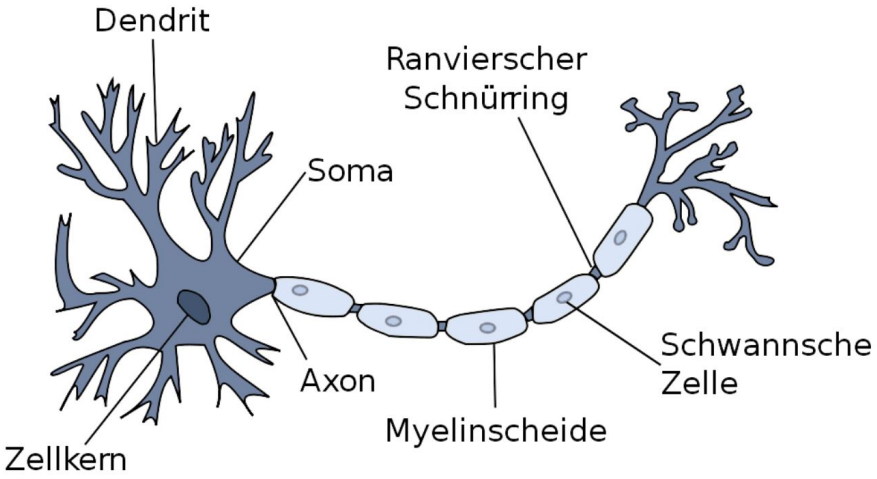


Abbildung 2.1: Schematische Abbildung einer Nervenzelle, Quelle [1].

empfangen [1]. Typischerweise empfängt ein Neuron Signale von 2000 und 10.000 anderen Nervenzellen [10].
Synapsen, die elektrische Signale empfangen, haben eine starke, direkte, nicht regulierbare Verbindung vom Sender zum Empfänger. Diese sind für hart kodierte Verhaltensmechanismen nützlich wie zum Beispiel den Fluchtreflex. Die chemische Synapse hingegen ist nicht direkt mit dem Sender verbunden, sondern durch den synaptischen Spalt getrennt [1]. Zur Übertragung eines elektrischen Signals wird dieses auf der präsynaptischen Seite in ein chemisches Signal kodiert, indem Neutransmitter freigesetzt werden. Diese können über den synaptischen Spalt übertragen und anschließend auf der postsynaptischen Seite wieder in ein elektrisches Signal kodiert werden. Ein großer Vorteil dieser Übertragungsart ist die Regulierbarkeit [1]. Verschiedene Neurotransmitter können unterschiedliche Effekte auf das Neuron haben, beispielsweise anregend (exzitatorisch) oder hemmend (inhibitorisch) [11]. Zusätzlich kann die Menge der freigesetzten Neurotransmitter die Stärke des Signals beeinflussen [1]. Auf lange Zeit gesehen können auch neue Verbindungen entstehen oder alte aufgelöst werden. Es wird angenommen, dass dies die Grundlage des Lernens im menschlichen Gehirn ist [9].
Sowohl die erregenden als auch hemmenden Signale werden über die Dendriten an den Axonhügel weitergeleitet, welcher sich zwischen dem Soma und dem Axon befindet. Dort werden die Signale akkumuliert. Wird bei diesem Vorgang ein gewisser Schwellwert überschritten, wird ein elektrischer Impuls erzeugt der über das Axon weitergeleitet wird [11]. Das Axon ist typischerweise 1cm in Ausnahmen sogar bis zu einem 1m lang und von der Myelinscheide umgeben, die unter anderem Schutz vor mechanischer Überanspruchung bietet [9]. Zusammen mit den Ranvierschen Schnürringen ermöglicht diese zudem eine schnellere Weiterleitung des Aktionspotenzials [11]. Das Axon endet mit dem sogenannten Endknopf oder auch Axonterminal genannt. Dieses ist mit den Synapsen von anderen Neuronen verbunden und kann beim Eintreffen eines Signals die Neurotransmitter freisetzten und somit das Signal übertragen [11]. Typischerweise gibt ein einzelnes Neuron

sein Signal an 1000 bis 10.000 anderen Neuronen weiter, in Ausnahmefällen sogar an bis zu 150.000 andere Neuronen [10], die alle parallel arbeiten. So entsteht ein sehr großes und leistungsfähiges neuronales Netz.

### 2.1.2 Künstliche neuronale Netze

KNN sind ein mathematisches Modell, dass im Vergleich zum biologischen Vorbild stark vereinfacht und idealisiert. Trotzdem können unterschiedlichste mathematische Funktionen abgebildet werden. In diesem Kapitel wird die grundsätzliche Funktionsweise sowie die einzelnen Komponenten der KNN vorgestellt.

Betrachtet man ein KNN als Blackbox (TODO REFERENZ BILD), gibt es eine Menge von Eingabewerten, die in einem Eingabevektor kodiert sind und eine Menge an Ausgaben, die in einem Ausgabevektor kodiert sind [12]. Die Eingaben werden im Falle der KNN nicht durch Rezeptoren erfasst sondern durch ein Optimierungsproblem gegeben. Der Ausgabevektor soll das gewünschte Ergebnis enthalten. Die Interpretation von diesem variiert je nach Optimierungsproblem und Netzarchitektur.

Betrachtet man die Struktur der KNN sind einige Ähnlichkeiten zum biologischen Vorbild erkennbar. Diese werden im folgenden genauer betrachtet [10]:

1. **Neuronen**
   Ähnlich zu den biologischen neuronalen Netzen, besteht auch das KNN aus vielen Neuronen [10]. Dies sind einfache Recheneinheiten, die primitive Funktionen bestimmen können [12] und deren genaue Funktionsweise in Kapitel 2.1.3 erläutert wird. Vorweggenommen sei, dass ein Neuron mehrere Eingabewerte besitzt, welche gewichtet sind und akkumuliert werden. Hierbei entsteht ein skalarer Ausgabewert, der den Aktivierungsgrad des Neurons repräsentiert und von anderen Neuronen als Eingabe verwendet werden kann [1].

2. **Gerichtete gewichtete Verbindungen**
   Wie im vorherigen Punkt angedeutet, sind Neuronen über gerichtete Verbindungen miteinander vernetzt. Der Aktivierungszustand eines Neurons wird entsprechend der Verbindungen an die Zielneuronen weitergegeben, welche diesen Wert als Eingabe verarbeiten. Wie bei den biologischen neuronalen Netzen auch, können Eingaben unterschiedlich stark anregend und hemmend wirken. Dies wird bei den KNN über Gewichte in den Verbindungen realisiert [10].

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

Trotz der vorgestellten Ähnlichkeiten, gibt es sehr viele Unterschiede zwischen den biologischen neuronalen Netzen und den KNN. Beispiel hierfür ist der Größenunterschied. Das menschlichge Gehirn mit seinen $10^{11}$ Neuronen besitzt pro Neuron ungefähr $10^4$ Verbindungen, während die meisten KNN nur $10^2$ bis $10^4$ Neuronen mit insgesamt $10^5$ Verbindungen besitzen. Auch werden keine chemischen Effekte die auf benachbarte Neuronen wirken sowie zeitliche und räumliche Lokalitätsprinzipien beachtet [10]. Aus diesen Gründen sind die KNN keine Nachbildung der biologischen neuronalen Netzen sondern verwenden diese nur als Inspiration.

### 2.1.3 Das Neuron

In diesem Kapitel wird die genaue Funktionsweise der einzelnen Neuronen betrachtet. Hierfür werden drei Phasen vorgestellt, in denen die Ausgabe eines einzelnen Neurons berechnet wird. Betrachtet man ein KNN führen typischerweise mehrere Verbindungen zu einem Neuron $j$, welche von den Neuronen $i_1, i_2, ..., i_n$ ausgehen [1]. Dieses ist schematisch in Abbildung (TODO ABBILDUNG EINFÜGEN) dargestellt.

#### Propagierungsfunktion

Die Ausgabewerte $o_{i_1}, o_{i_2}, ..., o_{i_n}$ der Neuronen $i_1, i_2, ..., i_n$ werden als Eingabewerte für das Neuron $j$ verwendet. Für jeden Eingabewert existiert ein entsprechendes Gewicht $w_1, w_2, ..., w_n$ [1]. Somit repräsentiert $w_{ij}$ das Gewicht für die Verbindung von Neuron $i$ zu Neuron $j$ [10]. Die Propagierungsfunktion $f_{prop}$ berechnet die Netzeingabe $net_j$, welche in der nächsten Phase weiterverwendet wird [1].

$$net_j = f_{prop}(o_1, o_2, ..., o_n, i_1, i_2, ..., i_n)$$

Die meist verwendete Propagierungsfunktion, welche auch in den späteren Beispielen genutzt wird, ist die gewichtete Summe. Hierbei werden, entsprechend der Formel, die Werte $o_i$ mit dem entsprechenden Gewicht $w_i$ multipliziert und aufsummiert [1]:

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

#### Aktivierungsfunktion

Der Aktivierungszustand $a_j(t)$ gibt den Grad der Aktivierung von Neuron $j$ zum Zeitpunkt $t$ an [10]. Ein neuer Aktivierungszustand zum Zeitpunkt $t + 1$ wird mit der Aktivierungsfunktion $f_{act}$ berechnet. Diese berücksichtigt nicht nur die Netzeingabe $net_j(t)$ sondern auch den vorherigen Aktivierungszustand $a_j(t)$ und den Schwellwert $\Theta$ der Aktivierungsfunktion [10]. Ein Schwellwert $\Theta_j$, auch Bias genannt, ist dem Neuron $j$ zugeordnet und gibt die Stelle an, an welcher die Aktivierungsfunktion die größte Steigung hat [1]. Somit kann die Berechnung der Aktivierung $a_j(t + 1)$ durch folgende Formel ausgedrückt werden [10]:

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

Bei der Berechnung ist der Schwellwert $\Theta$ besonders wichtig. Oftmals verwenden einige oder alle Neuronen eines KNN dieselbe Aktivierungsfunktion, die Schwellwerte hingegen unterscheiden sich je nach Neuron. Des weiteren sei angemerkt, dass die vorherige Aktivierung $a_j(t)$ je nach Netzstruktur oft nicht bei der Berechnung berücksichtigt wird [1]. Zudem wird in der Praxis bei Verwendung der gewichteten Summe als Propagierungsfunktion der Schwellwert eines Neurons oft schon in der ersten Phase miteinbezogen. Hierdurch ändert sich die Berechnung der Netzeingabe zu $net_j = \sum_i (o_i \cdot w_{i,j}) - \Theta_j$. Bei der Berechnung der Aktivierungsfunktion gilt dann $\Theta_j = 0$.

Je nach Anwendungsgebiet können verschiedene Aktivierungsfunktionen mit unterschiedlichen Eingenschaften eingesetzt werden, von denen vier in Abbildung (TODO ABBILDUNG) dargestellt sind. Im folgenden wird angenommen, dass gilt $\Theta_j = 0$.
Das einfachste Beispiel für eine Aktivierungsfunktion ist die sogenannte binäre Schwellwertfunktion, welche abhängig vom Schwellwert $\Theta$ nur die Werte 0 und 1 zurückgeben kann [1]. Die Formel hierfür ist:

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

Allerdings ist diese Funktion an ihrem Schwellwert nicht differenzierbar und ansonsten ist der Wert der Ableitung immer 0 [1]. Diese Eigenschaften machen sie ungeeignet für bestimmte Lernverfahren, wie zum Beispiel den Backpropagation Algorithmus auf welchen kurz in Kapitel 2.1.5 eingegangen wird [1]. Dieses Problem kann durch die Verwendung einer Sigmoidfunktion gelöst werden. Zwei bekannte Beispiele für Sigmoidfunktionen sind die logistische Funktion und der Tangens Hyperbolicus (tanh) [13]. Die logistische Funktion kann Werte von 0 bis 1 annehmen und durch einen entsprechenden Parameter $T$ bezüglich der x-Achse gestreckt und gestaucht werden [1]. Berechnet wird sie mit:

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

Allerdings können neuronale Netze je nach Verfahren schneller optimiert werden, wenn das durchschnittliche Gewicht aller Verbindungen nahe 0 ist. In diesem Fall ist die tanh Funktion besser geeignet, da sie Werte zwischen -1 und 1 annehmen kann [13]. Das letzte hier vorgestellte Beispiel ist die sogenannte *Rectifier* Funktion. Diese wird oft in Zusammenhand mit dem Backpropagation Algorithmus erfolgreich eingesetzt und erzielt mit diesem schneller bessere Optimierungsergebnisse [14]. Berechnet wird sie mit:

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

#### Ausgabefunktion

Die Ausgabefunktion $f_{out}$ berechnet die Ausgabe $o_j$ von Neuron $j$. Als Eingabewert wird die Aktivierung $a_j$ verwendet [10]. Somit ist die Funktion definiert mit:

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

In der Praxis ist die Ausgabefunktion, ähnlich wie die Aktivierungsfunktion, meistens global für alle Neuronen definiert. Zudem wird oft die Identitätsfunktion verwendet. In diesem Fall gilt $o_j = a_j$ [1]. Dies gilt auch für die später vorgestellten Beispiele. Ist die Ausgabe $o_j$ berechnet, kann sie als Eingabewert für andere verbundene Neuronen dienen.

### 2.1.4 Netzstrukturen

Aus dem vorherigen Kapitel ist ersichtlich, dass die Gewichte einen großen Einfluss auf das Ergebnis eines einzelnen Neuron haben. Der Ausgabevektor eines KNN wird neben den Gewichten auch von der Anzahl an Neuronen sowie deren Verbindungsstruktur beeinflusst. Je nach Optimierungsproblem können unterschiedliche Varianten eingesetzt werden, welche in diesem Kapitel genauer vorgestellt werden.

Typischerweise besitzt jedes KNN Eingabe- und Ausgabeneuronen. Optional kann ein KNN beliebig viele verdeckte Neuronen enthalten. Diese werden auch als *Input*-, *Output*- und *Hidden*-Neuronen bezeichnet [10]. Die Anzahl der Eingabe- und Ausgabeneuronen ist abhängig von der Größe des Eingabe- bzw. Ausgabevektors. Für jedes Element in den Vektoren gibt es ein entsprechendes Neuron (TODO ABBILDUNG). Bei vielen Netzstrukturen werden die Neuronen des KNN verschiedenen Schichten zugeordnet. In der ersten Schicht befinden sich die Eingabeneuronen und in der letzten die Ausgabeneuronen. Dazwischen befinden sich $n$ Schichten mit verdecken Neuronen [10].

Bei der Berechnung eines KNN werden zuerst die Werte des Eingabevektors in die entsprechenden Eingabeneuronen gesetzt. Anschließend werden alle Neuronen in einer bestimmen Reihenfolge aktiviert bzw. berechnet. Zuletzt bilden die Werte der Ausgabeneuronen den Ausgabevektor. Die verstecken Neuronen befinden sich zwischen den Eingabe- und Ausgabeneuronen und werden so genannt, da ihr Ausgabewert nur ein Zwischenergebnis ist und vor dem Anwender verborgen bleibt. Trotzdem sind sie ein elementarer Bestandteil der KNN und bestimmen maßgeblich dessen Leistungsfähigkeit. Beispielweise kann ein KNN, welches nur aus *Input*- und *Output*-Neuronen besteht nur eine lineare Funktion nachbilden. Ein KNN mit einer ausreichend großen verdeckten Schicht kann jede beliebige kontinuierliche Funktion darstellen. Mit zwei Schichten kann ein KNN sogar jede unstetige mathematische Funktion mit beliebiger Genauigkeit abbilden [9].

Je nach Art des Verbindungsmuster zwischen den Neuronen werden KNN einer von zwei Gruppen zugeordnet. Die erste Gruppe enthält Netze ohne Rückkopplung, wel-

che auch *feedforward*-Netze genannt werden. Die zweite Gruppe sind die sogenannten *recurrent*-Netze, zu welchen KNN mit Rückkopplungen gehören [10].

### Netze ohne Rückkopplung

Die Definition der *feedforward*-Netze ist einfach: Es darf keine Verbindung von einem Neuron $j$ ausgehen, welche wieder zu sich selbst führt. Dabei ist es irrelevant ob eine direkte oder indirekte Verbindungen über Zwischenneuronen besteht. Somit entsteht ein azyklischer Graph [10] und das KNN kann infolgedessen keinen internen Zustand besitzen. Für die gleiche Eingabe wird immer dasselbe Ergebnis berechnet. Innerhalb dieser Kategorie gibt es zwei Untergruppen, die ebenenweise verbundenen KNN und die KNN welche über sogenannte *shortcut* Verbindungen verfügen.
Bei den rein ebenenweise verbundenen KNN sind die Eingabewerte eines Neurons immer aus der vorherigen Schicht. Der berechnete Ausgabewert eines Neurons wird nur an die Neuronen der nächsten Schicht weitergeleitet [10]. Ein Beispiel hierfür ist in Abbildung (TODO ABBILDUNG) dargestellt.
Im Gegensatz dazu stehen die KNN mit *shortcut* Verbindungen können. Eine *shortcut* Verbindung kann eine oder mehrere Schichten überspringen. Für gewisse Optimierungsprobleme, unter anderem für das Beispiel in Kapitel (TODO CHAPTER), können so kleiner KNN erzeugt werden [10].

### Netze mit Rückkopplung

Netze mit Rückkopplung werden oft auch in Schichten dargestellt, allerdings kann ein KNN sich je nach Art selbst beeinflussen indem Zyklen in der Berechnung entstehen, wodurch das Zwischenspeichern von Werten ermöglicht wird [9]. Somit wird das Ergebnis sowohl durch die Eingabewerte des KNN als auch durch die vorherigen Berechnungen beeinflusst [15]. Wie auch bei den *feedforward*-Netzen, können auch die Netze mit Rückkopplung je nach Verbindungsart in verschiedenen Untergruppen zugeordnet werden [10].

1. Bei KNN mit direkter Rückkopplung können Neuronen Verbindungen zu sich selbst haben (TODO ABBILDUNG). Dadurch können sie ihre Aktivierung verstärken oder abschwächen [10].

2. Netze mit einer indirekten Rückkopplung erlauben im Gegensatz zu den *feedforward*-Netzen auch Verbindungen in die vorherige Schicht (TODO ABBILDUNG) [10]. Wie bei der direkten Rückkopplung kann sich ein Neuron $j$ selbst beeinflussen, wenn es seinen Ausgabewert an ein Neuron $i$ der nächsten Schicht weiterleitet, welches eine Rückkopplung zu $j$ hat [1].

3. KNN mit lateralen Rückkopplungen erlauben Verbindungen von Neuronen innerhalb einer Schicht (TODO ABBILDUNG), welche hemmend oder aktivierend wirken können. Oft entsteht dabei ein *Winner-Takes-All*-Schema, da das beste Neuron alle anderen hemmt und sich selbst aktiviert [1].

4. Bei den vollständig verbundenen Netze darf ein Neuron zu jedem anderen verbunden sein, außer sich selbst (direkte Rückkopplung). Ein KNN, in welchem jedes Neuron zu jedem anderen eine Verbindung hat, wird auch Hopfield-Netz genannt. Ein Beispiel hierfür ist in Abbildung (TODO ABBILDUNG) dargestellt [1].

### 2.1.5 Optimierungsmöglichkeiten

Durch die vorherigen Kapitel ist erkennbar, dass das erfolgreiche lösen eines Optimierungsproblems mit einem KNN von vielen Faktoren abhängt. In der Praxis ist es für komplexe Aufgaben nicht möglich, diese manuell zu bestimmen. Aus diesem Grund muss ein Optimierungsverfahren, welches auch als Lernverfahren bezeichnet wird, angewendet werden. Ziel von diesem ist, einen Teil oder alle Parameter des KNN durch einen Algorithmus automatisch zu bestimmen. Typischerweise ist das Lernverfahren unabhängig von dem eigentlichen Optimierungsproblem und kann daher in verschiedenen Bereichen ohne großen zusätzlichen Aufwand eingesetzt werden.

Ein Lernverfahren kann theoretisch über vier verschiedene Arten die Eigenschaften eines KNN optimieren [10]. Diese sind im Folgenden kurz zusammenfasst.

1. **Modifizieren der Verbindungsgewichte**:
   Die Gewichte der einzelnen Verbindungen werden in der Praxis von allen Lernverfahren optimiert [10]. Gründe hierfür sind, dass ein große Netzwerke mehrere Millionen Verbindungen besitzen, welche unmöglich manuell optimiert werden können und dass die Gewichte entscheidend für die erfolgreiche Optimierung sind.

2. **Modifizieren der Schwellwerte**:
   Die Schwellwerte der Neuronen werden wie die Gewichte von den meisten Lernverfahren optimiert. In der Praxis ist der hierbei verwendete Vorgang oft identisch zur Gewichtsoptimierung. Dies ist möglich, wenn, wie in einigen Implementierungen umgesetzt, die Schwellwerte durch Gewichte repräsentiert werden. Hierzu wird einem KNN ein sogenanntes *Bias*-Neuron hinzugefügt, welches immer den Wert 1 hat. Von diesem gehen dann Verbindungen zu allen Neuronen aus. Der Schwellwert $\Theta_j$ von einem Neuron $j$ wird durch das Gewicht $w_{\Theta j}$ repräsentiert. Dieses ist der eingehenden Verbindung vom *Bias*-Neuron zugeordnet, dass gilt $1 \cdot w_{\Theta j} = \Theta_j$. Somit muss bei der Berechnung eines Neurons der Schwellwert nicht mehr explizit miteinbezogen werden sondern wird im Rahmen der Propagierungsfunktion indirekt mit den anderen gewichteten Eingaben verarbeitet. Bezüglich der Optimierung wird die Verbindung zum *Bias*-Neuron, wie andere gewichtete Verbindungen behandelt [10].

3. **Hinzufügen und Entfernen von Verbindungen oder Neuronen**:
   Das Hinzufügen beziehungsweise Entfernen von Verbindungen und Neuronen ist im Vergleich zu den bereits vorgestellten Möglichkeiten aufwändig und schwierig sinnvoll umzusetzen. Daher wird es von vielen bekannten Algorithmen nicht implementiert. Bei diesen muss die Struktur mithilfe von Expertenwissen oder

Erfahrung festgelegt werden [16], andernfalls muss eine geeignete Struktur experimentell ermittelt werden. Da dieses Vorgehen nicht effizient ist, gibt es dennoch einige Algorithmen, welche diese Art der Optimierung umsetzen. Diese gehöhren häufig zu der Klasse der Evolutionären Algorithmen, auf welche in Kapitel (TODO REFF!!) genauer eingegangen wird [1].

4. **Ändern der Propagierungs, Aktivierungs-, und Ausgabefunktion**:
   Die Optimierung der verwendeten Propagierungs-, Aktivierungs- und Ausgabefunktion ist theoretisch möglich, dennoch ist die Umsetzung in der Praxis nicht sehr verbreitet [10]. Auch in dieser Arbeit werden diese Funktionen nicht durch einen Algorithmus angepasst und sind werden daher nicht weiter betrachtet.

### 2.1.6 Lernen in neuronalen Netzen TODO CHANGE TITLE

In Kapitel 2.1.5 sind Optimierungsmöglichkeiten aufgelistet, welche von einem Lernverfahren, in der sogennanten Trainingsphase des KNN, angepasst werden können. Ziel ist, dass am Ende dieser Phase der Ausgabevektor des KNN dem gewünschten Ergebnis entspricht. Voraussetzung hierfür ist, dass das gewünschte Ergebnis erkennbar ist [10]. Bei den Lernverfahren wird grundsätzlich zwischen dem überwachtem, unüberwachtem und bestärkendem Lernen unterschieden, welche unterschiedliche Arten des Lernens für verschiedene Aufgabenstellungen repräsentieren. Im folgenden wird ein Überblick über diese gegeben. Für eine genaue Beschreibung und die dazugehörigen Algorithmen wird auf entsprechende Fachliteratur verwiesen.

#### Überwachtes Lernen

Das überwachte Lernen, auch *supervised learning* genannt, wird häufig mit dem Backpropagation Algorithmus und seinen Derivaten umgesetzt und beruht auf bekannten Beispielen, welche durch einen externen "Lehrer"gegeben sind [10]. Dabei müssen die Beispieldaten in großer Anzahl schon vor dem Lernvorgang vorhanden sein und den Eingabevektor sowie den gewünschten Ausgabevektor des KNN enthalten [10]. Beispiel hierfür ist die Klassifizierung von Hunde- und Katzenbildern. Für jedes Bild muss der Eingabevektor bekannt sein, welcher aus den einzelnen Pixeln besteht sowie der Ausgabevektor, der in diesem Fall angibt ob ein Hund oder eine Katze abgebildet ist. In der sogenannten Trainingsphase, in welcher unter anderem die Gewichte optimiert werden, berechnet das KNN die Ausgabewerte für die in den Beispielen enthaltenen Eingabevektoren. Das erhaltene Ergebnis wird direkt mit dem gewünschtem Wert verglichen. Je nachdem wie groß die Differenz ist, werden die Parameter des KNN entsprechend angepasst [1]. Ziel dieses Vorgangs ist, dass Muster aus den Beispieldaten extrahiert werden und somit nicht nur für bekannten Beispiele die korrekte Lösung angegeben werden kann kann sondern auch für ähnlichen, unbekannten Eingabedaten, sodass die Eigenschaft der Generalisierung gegeben ist [10]. Dies wird überprüft, indem die Beispieldaten in Trainings- und Testdaten unterteilt werden. Die Trainingsphase wird nur mit den Trainingsdaten durchgeführt, sodass es die Testdaten dem KNN unbekannt sind. Ist

diese Phase abgeschlossen, weil das KNN zum Beispiel eine gute Genauigkeit erreicht hat, werden die Testdaten zur Validierung eingesetzt. Hierbei wird überprüft, ob das KNN auch für unbekannte Eingabevektoren die richtigen Ergebnisse berechnet [1]. Diese Art des Lernens ist im Vergleich zu den anderen Varianten sehr schnell, da zum Beispiel die Gewichte direkt so angepasst werden können, dass sie das gewünschte Ergebnis erzeugen [10]. Allerdings kann das Verfahren nicht in jeder Situation angewendet werden. Liegen keine Beispiele vor kann das KNN nicht trainiert werden. Sind die Beispieldaten fehlerhaft, verrauscht oder bieten nicht die beste Lösung, kann das Training langsam, nicht zufriedenstellend oder unmöglich sein.

### Unüberwachtes Lernen

Beim unüberwachtem Lernen, im Englischen *unsupervised learning* genannt, gibt es auch Beispieldaten, allerdings enthalten diese nur den Eingabevektor und keine gewünschten Ausgabewerte. Ziel von solchen Lernverfahren ist, die Eingabedaten verschiedenen Gruppen zuzuordnen, wobei sich ähnliche Eingabevektoren in derselben Gruppe befinden sollen [10]. An dieser wird die Funktionsweise wieder mit dem Beispiel von den Hunden- und Katzenbildern aus dem vorherigen Kapitel verdeutlicht. Durch das Lernverfahren werden dem KNN die Bilder aus den Beispieldaten gegeben. In diesem Fall ist aber nicht bekannt, welches Tier sich auf einem Bild befindet. Das KNN soll selbständig erkenne, dass es sich um 2 Arten von Tieren handelt und diese richtig zuordnen. Ein solches Verfahren kann einige Vorteile gegenüber überwachtem Lernen bieten [17]. Zum Beispiel müssen vor dem Training keine Beispieldaten mit Ausgabevektoren vorliegen, welche teilweise sehr teuer und aufwändig zu erstellen sind. Des weiteren kann je nach Algorithmus die Anzahl an Gruppen automatisch zugewiesen werden. So können auch unterschwellige Muster die Zuweisung beeinflussen, die nicht von einem Menschen erkannt werden würden [17].

### Bestärkendes Lernen

Die letzte Klasse ist das bestärkende Lernen, auch *reinforcment learning* genannt. Während der Optimierung erhält das KNN Beispieldaten, für welche der Ausgabevektor berechnet wird. Für das berechnete Ergebnis wird ein Feedback gegeben, welches auch als *reward* bezeichnet wird. Dieses gibt an, ob der Ausgabewert korrekt ist beziehungsweise wie richtig oder falsch. Das Lernverfahren muss mit diesen Angaben das KNN optimieren, kann aber nicht wissen wie die Gewichte und je nach Algorithmus die Struktur verändert werden müssen. Dies ist auch der Grund, warum diese Art des Lernens im Vergleich zum überwachtem Lernen sehr langsam ist, da die Gewichte nicht gezielt angepasst werden [10].

## 2.2 Evolutionäre Algorithmen

### 2.2.1 Genome und Phänotyp

### 2.2.2 Phasen des Algos

### 2.2.3 Kodierung

### 2.2.4 TWEANN?

### 2.2.5 Competing Convention Problem

## 2.3 NeuroEvolution of Augmenting Topologies

Der in dieser Arbeit verwendete Algorithmus heißt *NeuroEvolution of Augmenting Topologies* (NEAT), welcher im Jahr 2002 von Stanley und Miikkulainen vorgestellt wurde. Bei der Veröffentlichung hat NEAT für die meisten Optimierungsprobleme im Vergleich zu anderen Verfahren schneller Lösungen gefunden obwohl es neben den Gewichten des KNN auch die Struktur optimiert [18]. Somit gehört der Algorithmus zur Gruppe der *Topology and Weight Evolving Artificial Neural Network* (TWEANN) Algorithmen. Heute gilt NEAT immer noch als einer der bekanntesten Vertretern der neuroevolutionären Algorithmen und dient als Basis für viele Erweiterungen wie zum Beispiel HyperNEAT, cgNEAT, ...
Für den Erfolg nennen die Autoren drei besonders relevante Faktoren [18]:

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

In diesem Kapitel wird die grundsätzliche Funktionsweise von NEAT erläutert, wie sie in der originalen Publikation vorgestellt ist. Wenn nicht anderweitig gekennzeichnet, beziehen sich alle Informationen aus diesem Kapitel auf Quelle [18]. Für eine bessere Lesbarkeit wird in diesem Kapitel auf weitere Zitierungen verzichtet.

### 2.3.1 Kodierung

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

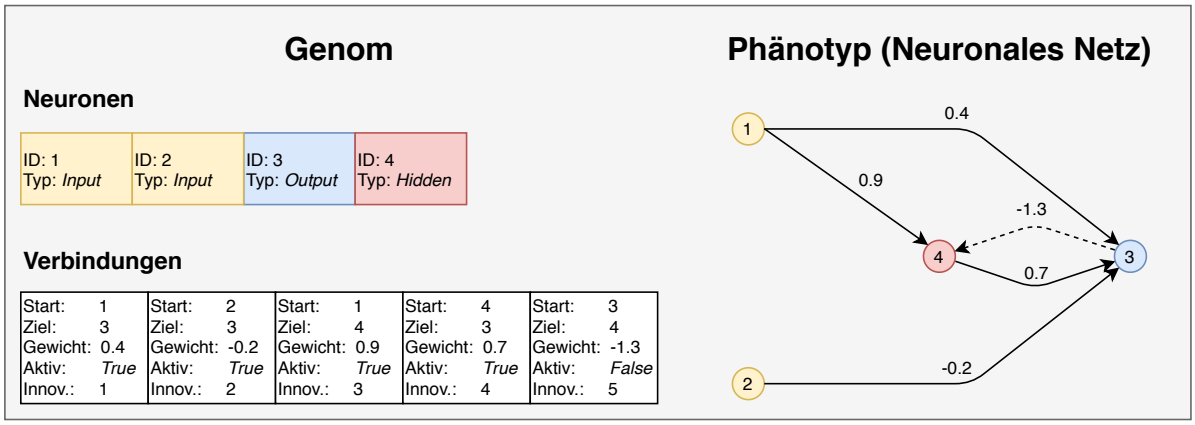


Abbildung 2.2: Schematische Darstellung von einem Genom mit dazugehörigem Phänotyp

### 2.3.2 Mutation

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

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

### 2.3.3 Reproduktion

Das Ergebnis der in Kapitel 2.3.2 vorgestellten Mutationen ist eine Population mit verschiedensten Genomen, welche unterschiedliche Gewichte und Strukturen haben können. Dies ist die schwierigste Form des in Kapitel 2.2.5 vorgestellten *competing convention* Problems und macht das Erstellen von Nachkommen besonders schwierig.

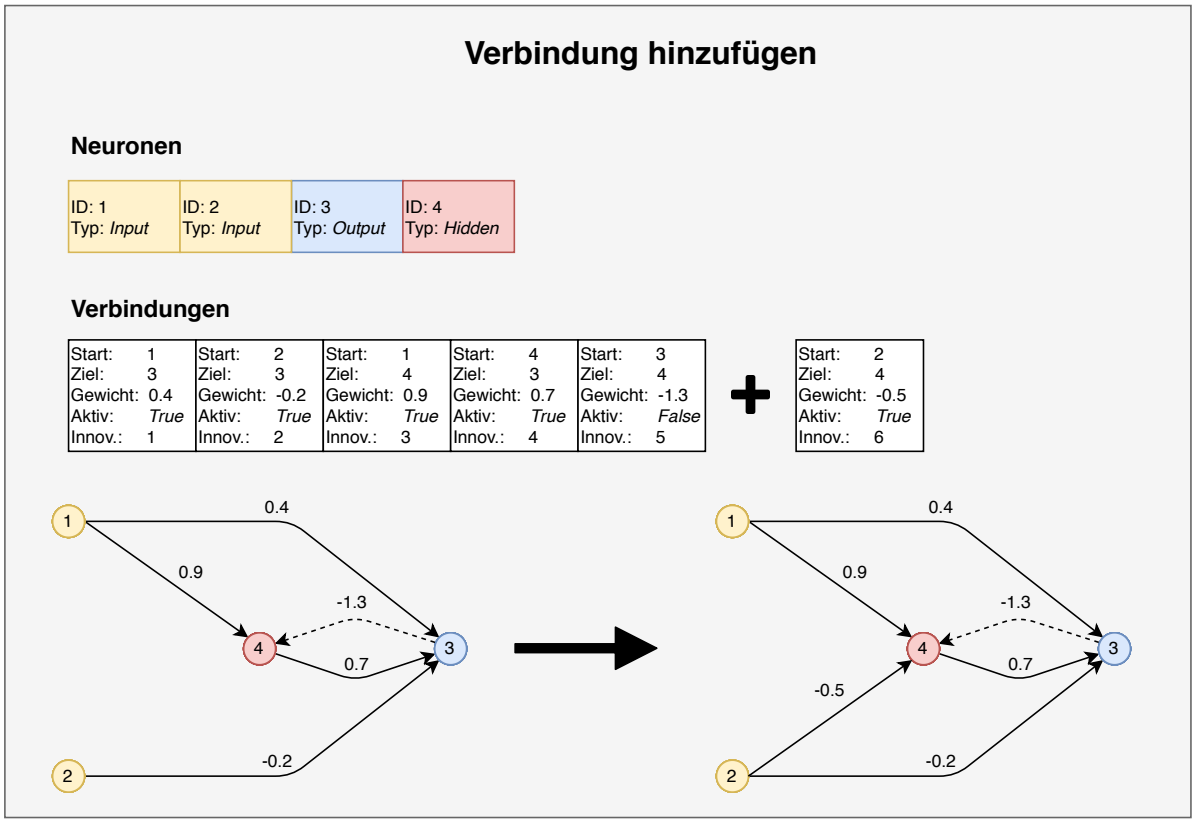


Abbildung 2.3: Schematische Darstellung von einem Genom mit dazugehörigem Phänotyp

NEAT löst dieses Problem, indem es den historischen Ursprung von jeder strukturellen Mutation überwacht. Haben zwei Verbindungen denselben Ursprung, haben sie in der Vergangenheit dieselbe Struktur repräsentiert, auch wenn sie inzwischen unterschiedliche Gewichte haben. Zu diesem Zweck besitzt jede Verbindungen die im Kapitel 2.3.1 erwähnte Innovationsnummer. Jedes mal, wenn eine neue Verbindung entsteht wird ein globaler Zähler inkrementiert und der Wert als Innovationsnummer der Verbindung verwendet. Abbildung (TODO ABBOLDUNG) zeigt die Zuweisung beispielhaft. Die erste Mutation, welche nur eine neue Verbindung herstellt hat die Innovatiosnummer X zugewiesen bekommen. Wenn im folgenden ein neues Neuron mit zwei weiteren Verbindungen hinzugefügt wird, erhalten diese die Nummern Y und Z. Werden Verbindungen von einem Genom in der Reproduktiosnphase für die Nachkommen ausgewählt, wird auch die Innovationsnummer übertragen. Somit ist auch bei den nachfolgenden Generationen ersichtlich, was der historische Ursprung einer Verbindung ist. Tritt durch Zufall dieselbe Mutation in einer Generation mehrfach auf, erhalten die neuen Verbindungen dieselben Innovationsnummern. Hierfür müssen alle aufgetretenen Mutation in einer Generation zwischengespeichert werden.
Die Innovationsnummern können nicht nur ressourcensparend implementiert werden, sie machen das Erzeugen von Nachkommen in der Reproduktionsphase bedeutend einfacher da beim kreuzen von zwei Elternteilen keine aufwendige Strukturanalyse benötigt wird.

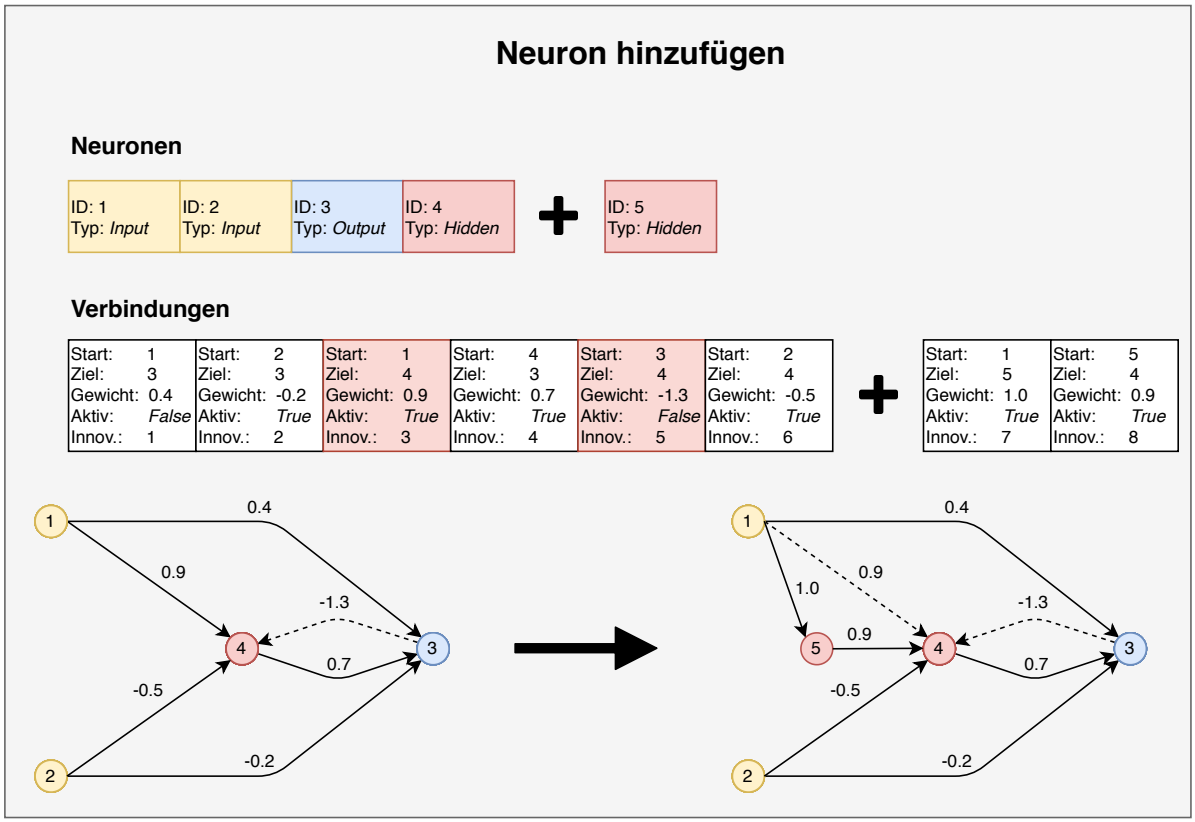


Abbildung 2.4: Schematische Darstellung von einem Genom mit dazugehörigem Phänotyp

Abbildung (TODO ABBILDUNG) zeigt beispielhaft wie ein Nachkommen aus zwei Elterngenomen $X$ und $Y$ entsteht. Die sogenannten *maching genes* sind Verbindungen, deren Innovationsnummern in beiden Elterngenomen vorkommen. Beim Erstellen der Nachkommen wir für jede Verbindung in den *matching genes* zufällig entschieden, aus welchem Elternteil diese übernommen wird. Die sogenannten *disjoint genes* und *excess genes* sind Verbindungen, die nur in einem Elternteil vorkommen. Zu den *disjoint genes* gehören die Verbindungen, deren Innovationsnummer kleiner als die größte Innovationsnummer des zweiten Elterngenoms ist. Die *excess genes* sind Verbindungen, deren Innovationsnummer größer als die höchste Innovationsnummer im anderen Elternteil ist. Beim Erzeugen von Nachkommen werden nur die *excess genes* und *disjoint genes* von dem Elternteil übernommen, welches den höheren Fitnesswert erzielt hat. Haben beide Elternteile denselben Wert, werden die Verbindungen von beiden übernommen. Bei dieser Implementierung wird angenommen, dass der Schwellwert der Neuronen wie in Kapitel (TODO KAPITE) erläutert ist, durch eine Verbindung zu einem Bias-Neuron ausgedrückt wird. Dadurch enthalten die Neuronen kein spezifischen Informationen, die sich zwischen den Elterngenomen unterscheiden. Die Nachkommen übernehmen deshalb immer die Neuronen des Elternteils mit dem größeren Fitnesswert.

### 2.3.4 Spezies

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

Das Problem wird von NEAT durch das Einführen von verschiedenen Spezies gelöst. Das Ziel ist, Genome die sich strukturell ähneln in einer Spezies zu gruppieren. Bei der Auswahl der Elterngenome für die Nachkommen muss ein Genom nicht mehr mit der ganzen Population konkurrieren, sondern nur noch mit den anderen Genomen in der eigenen Spezies. Somit sind neue Innovationen erst einmal in ihrer Spezies vor dem Aussterben geschützt und können mit der Zeit optimiert werden. Für die Implementierung eines solchen Verfahrens wird eine Funktion benötigt, die messen kann wie ähnlich oder unterschiedlich zwei Genome sind. Auch hier kann wie bei der Rekombination auf eine aufwendige Strukturanalyse verzichtet werden, da dies mit den bereits bekannten Innovationsnummern umsetzbar ist. Je mehr *excess genes* und *disjoint genes* zwei Genome besitzen, desto weniger evolutionäre Geschichte teilen sie und sind somit unterschiedlicher. Auch der Gewichtsunterschied ist ein wichtiger Faktor, wie in Kapitel 2.2.5 dargestellt. Die von NEAT verwendete Formel um die Kompatibilität $\delta$ zwischen zwei Genomen zu berechnen ist im Folgenden abgebildet.

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

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

Die Zuordnung von neu erstellten Genomen zu einer Spezies erfolgt nach der Reproduktions- und Mutationsphase. Hierfür wird eine geordnete Liste mit allen verfügbaren Spezies benötigt. Jede Spezies wird durch ein Genom repräsentiert, welches in der vorherigen Generation ein Mitglied von dieser war. Bei der Zuordnung von einem Genom wird über die Liste der Spezies iteriert und zu jedem Repräsentanten der Kompatibilitätswert

$\delta$ gebildet. Ist $\delta \leq \delta_t$, wobei $\delta_t$ ein konfigurierbarer Schwellwert ist, wird das Genom der Spezies zugeordnet und die Suche abgebrochen. Ist das Genom zu keiner Spezies kompatibel wird eine Neue erstellt und das Genom als Repräsentant gesetzt.

Zum erhalten von verschiedenen Strukturen muss verhindert werden, dass eine Spezies zu groß wird und die restlichen verdrängt auch wenn viele der Mitglieder gute Fitnesswerte erzielen. Zusätzlich müssen vorallem neue Spezies geschützt werden. Diese haben initial wenige Mitglieder und somit eine geringere Chance als Elterngenome ausgewählt zu werden. Zum Lösen dieses Problems verwendet NEAT sogenanntes *explicit fitness sharing*, welches 1987 von Goldberg, Richardson u. a. in ihrer Arbeit [19] vorgestellt wurde. Jede Spezies bekommt bei der Reproduktion eine Anzahl an Nachkommen zugewiesen, welche proportional zu der Fitness $f_s$ der Spezies ist. Diese ergibt sich aus der Summe aller angepassten Fitnesswerte $f'$ der Mitglieder. Der angepasste Fitnesswert $f'$ eines Genoms wird berechnet indem die erreichte Fitness $f$ durch die Anzahl an Mitgliedern in der Spezies geteilt wird. Das Ziel dieser Maßnahme ist, dass große Spezies im Vergleich zu kleinen benachteiligt werden und kleine erfolgreiche Spezies entsprechend viele Nachkommen zugewiesen bekommen. Ein Beispiel hierfür ist in Abbildung xy (TODO ABBILDUNG) dargestellt. Obwohl die zweite Spezies bedeutend weniger, aber dafür gute Genome besitzt, werden ihr mehr Nachkommen zugewiesen. Würden die Anzahl von Nachkommen einer Spezies proportional zu der Summe der erreichten Fitnesswerte vergeben, hätte die kleinere Spezies weniger zugewiesen bekommen.

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

### 2.3.5 Starten mit einer minimalen Struktur

Ein Ziel von NEAT ist, wie bei vielen anderen Optimierungsalgorithmen auch, eine Lösung so schnell wie möglich zu finden. Ein wichtiger Faktor hierbei ist die Größe des KNN. Ein zu großes KNN hat viele modifizierbare Parameter, welche nicht zur erfolgreichen Lösung benötigt werden. Trotzdem wird die Laufzeit des Algorithmus erhöht, da auch diese optimiert werden müssen. Ein zu kleines KNN kann, wie in Kapitel (TODO REF XOR) veranschaulicht, unter Umständen nicht in der Lage sein eine Lösung zu finden. Somit ist die richtige Größe des KNN entscheidend für die schnelle Optimierung. Für Algorithmen welche nur die Gewichte eines KNN optimieren, muss diese Struktur von einem Menschen festgelegt werden. Meistens basiert dies auf Basis von Expertenwissen oder Erfahrung [16]. Im Gegensatz hierzu stehen die TWEANN Algorithmen, welche selbstständig eine gute Struktur bilden sollen. Diese starten oft mit einer initialen Population mit vielen verschiedenen zufällig erstellten Topologien, mit dem Ziel genetische Diversität zu bieten. Wie in Kapitel 2.2.4 erläutert, ist dies oft nicht effizi-

ent, da viele Strukturen nicht gebraucht werden und Zeit benötigt wird diese zu entfernen.

NEAT hingegen startet mit einer Population, bei welcher alle Genome dieselbe minimale Struktur besitzen. Die entstehenden KNN haben nur aus *Input*- und *Output*-Neuronen und keine *Hidden*-Neuronen. Jedes *Input*-Neuron besitzt eine Verbindungen zu jedem *Output*-Neuron mit einem zufällig gewählten Gewichten. Neue Strukturen werden durch die vorgestellten Arten der Mutation hinzugefügt, von denen nur die langfristig integriert werden, welche den Fitnesswert erhöhen. Somit ist die Existenz von jeder Struktur in einem Genom gerechtfertigt. Insgesamt gibt dies NEAT einen Vorteil bezüglich der Evaluationszeit gegenüber anderen TWEANN Algorithmen, da die Anzahl der zu optimierenden Parameter und somit die Dimensionen des Suchraums minimiert sind.

## 2.4 MPI

# 3 Analyse

## 3.1 Anforderungen

## 3.2 Softwarearchitektur und Implementierung

## 3.3 Testsetup

## 3.4 Evaluation

# 4 Software Architektur und Implementierung

# 5 Evaluation

## 5.1 Testsetup

## 5.2 Ergebnisse

# 6 Zusammenfassung und Ausblick

# Quellenverzeichnis

[1] David Kriesel. 2008. Ein kleiner überblick über neuronale netze. *Download unter http://www. dkriesel. com/index. php*.

[2] Warren S McCulloch und Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5, 4, 115–133.

[3] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

[4] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath u. a. 2012. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. *IEEE Signal processing magazine*, 29, 6, 82–97.

[5] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov und Michael Collins. 2016. Globally normalized transition-based neural networks. *arXiv preprint arXiv:1603.06042*.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra und Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[7] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot u. a. 2016. Mastering the game of go with deep neural networks and tree search. *nature*, 529, 7587, 484.

[8] Werner Kinnebrock. 2018. *Neuronale Netze: Grundlagen, Anwendungen, Beispiele*. Walter de Gruyter GmbH & Co KG.

[9] Stuart Russell und Peter Norvig. 2013. Künstliche intelligenz. ein moderner ansatz, 3. ak. aufl. (2013).

[10] Andreas Zell. 2003. Simulation neuronaler netze. 4., unveränderte auflage. (2003).

[11] Clemens Kirschbaum. 2008. *Biopsychologie von A bis Z*. Springer-Verlag.

[12] Andreas Scherer. 2013. *Neuronale Netze: Grundlagen und Anwendungen*. Springer-Verlag.

[13] Yann A LeCun, Léon Bottou, Genevieve B Orr und Klaus-Robert Müller. 2012. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 9–48.

[14] Xavier Glorot, Antoine Bordes und Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 315–323.

[15] Tsungnan Lin, Bill G Horne und C Lee Giles. 1998. How embedded memory in recurrent neural network architectures helps learning long-term temporal dependencies. *Neural Networks*, 11, 5, 861–868.

[16] Stanley, Kenneth O. 2017. Neuroevolution: a different kind of deep learning. `www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/`. [Online; Abgerufen am 27. Juli 2020]. (2017).

[17] Mahamed Omran, Andries Engelbrecht und Ayed Salman. 2005. Differential evolution methods for unsupervised image classification. In Band 2. (Januar 2005), 966–973. DOI: `10.1109/CEC.2005.1554795`.

[18] Kenneth O. Stanley und Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10, 2, 99–127.

[19] David E Goldberg, Jon Richardson u. a. 1987. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum, 41–49.

# Eidesstattliche Erklärung

This is the beginning