

# **Neuroevolution mit NEAT**

## **Bachelorthesis**

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang Angewandte Informatik (AIB)

an der Hochschule Heilbronn

**Simon Hauck**

Matrikelnummer: 192956

Erstprüfer:

Prof. Dr. Tim Reichert

Zweitprüfer:

Dipl.-Inform. Ruben Nuredini

Heilbronn, Januar 2019

## Kurzfassung

In dieser Arbeit wird eine Library für die Spieleentwicklungsumgebung Unity implementiert. Die Library stellt den NEAT Algorithmus zur Verfügung, der zur Optimierung von neuronalen Netzen verwendet werden kann. Diese können unter anderem zur Steuerung von Charakteren in Spielen verwendet werden. In dieser Arbeit wird die Library konzipiert und implementiert. Zudem wird die Anwendung der Library in drei Beispielprojekten demonstriert. In allen Beispielen schafft es die Library, die neuronalen Netze so zu optimieren, dass diese die gestellte Aufgabe erfolgreich lösen können.

**Schlagwörter:** Neuroevolution, NEAT, Spieleentwicklung

# Inhaltsverzeichnis

Kurzfassung .....	2
Inhaltsverzeichnis .....	3
Abbildungsverzeichnis .....	5
1 Motivation.....	7
1.1 Problemstellung .....	8
1.2 Ziel der Arbeit.....	8
1.3 Struktur der Arbeit .....	9
2 Grundlagen.....	10
2.1 Neuronale Netze .....	10
2.1.1 Aufbau und Funktionsweise neuronaler Netze .....	11
2.1.2 Lernprozess von neuronalen Netzen .....	14
2.2 Neuroevolutionäre Algorithmen .....	15
2.2.1 Natürliche Evolution .....	15
2.2.2 Evolutionäre Algorithmen .....	16
2.3 NeuroEvolution of Augmenting Topologies .....	17
2.3.1 Genetische Kodierung .....	18
2.3.2 Mutation .....	18
2.3.3 Rekombination von Genomen mit verschiedenen Strukturen.....	20
2.3.4 Beschützen von strukturellen Innovationen durch verschiedene Spezies.....	23
2.3.5 Inkrementelles Wachsen einer am Anfang minimalen Struktur ....	25
2.3.6 Einschränkungen des Zufalls in der Reproduktion .....	26
3 Softwarearchitektur .....	27
3.1 Architektur der Schnittstelle .....	27
3.1.1 <i>Genome</i> .....	28
3.1.2 <i>AgentObject</i> .....	28
3.1.3 <i>Species</i> .....	28
3.1.4 <i>PopulationManager</i> .....	29
3.1.5 <i>IPopulationManagerCallback</i> .....	29
3.2 Architektur NEAT .....	30
3.2.1 Klassendiagramm .....	31
3.2.2 Aktivitätsdiagramm .....	32
3.3 Architektur Neuronales Netz.....	36
3.3.1 Berechnung der Ausgabeneuronen.....	37
4 Evaluation.....	39
4.1 XOR-Beispiel.....	39
4.1.1 Das XOR-Problem .....	39
4.1.2 Aufbau des Tests.....	40

4.1.3	Implementierung .....	41
4.1.4	Parametrisierung .....	45
4.1.5	Ergebnis .....	45
4.2	Autonomes Fahren .....	47
4.2.1	Aufbau des Tests .....	47
4.2.2	Implementierung .....	49
4.2.3	Parametrisierung .....	53
4.2.4	Steuerung .....	53
4.2.5	Versionen .....	53
4.2.6	Sequenzielle vs. Parallele Evaluation .....	55
4.2.7	Ergebnis .....	56
4.3	Mario Klon .....	59
4.3.1	Spielprinzip .....	60
4.3.2	Aufbau des Tests in der vereinfachten Version .....	61
4.3.3	Implementierung .....	63
4.3.4	Parametrisierung .....	68
4.3.5	Ergebnis .....	68
5	Zusammenfassung und Ausblick .....	72
5.1	Ergebnisse .....	72
5.1.1	Integration .....	72
5.1.2	Performance .....	73
5.2	Weiterentwicklung .....	74
5.3	Einsatz in kommerziellen Spielen .....	74
	Quellenverzeichnis .....	76
	Eidesstattliche Erklärung .....	78

## Abbildungsverzeichnis

Abbildung 1 Neuronales Netz als Abbildungsvorschrift (Scherer, 1997) .....	11
Abbildung 2 Mathematisches Modell eines Neurons (Frochte, 2018) .....	11
Abbildung 3 Darstellung der Sigmoid-Funktion.....	12
Abbildung 4 Neuronales <i>feed-forward</i> Netz in Schichtenarchitektur .....	13
Abbildung 5 Neuronales <i>recurrent</i> Netz in Schichtenarchitektur .....	13
Abbildung 6 Kodierung eines Genoms mit NEAT (Stanley & Miikkulainen, 2002) ..	18
Abbildung 7 Strukturelle Mutation, die eine Verbindung hinzufügt (Stanley & Miikkulainen, 2002).....	19
Abbildung 8 Strukturelle Mutation, die ein Neuron hinzufügt (Stanley & Miikkulainen, 2002).....	19
Abbildung 9 Rekombination mit dem <i>Competing Conventions</i> Problem (Stanley & Miikkulainen, 2002).....	20
Abbildung 10 Rekombination zweier Genome mit NEAT (Stanley & Miikkulainen, 2002) .....	22
Abbildung 11 Berechnung der Nachkommen mit dem Fitnesswert $f$ und dem angepassten Fitnesswert $f'$ .....	24
Abbildung 12 Klassendiagramm der Schnittstelle .....	27
Abbildung 13 Aktivitätsdiagramm für die Schnittstelle.....	29
Abbildung 14 Ausschnitt aus dem Klassendiagramm mit weiteren genombezogenen Klassen.....	31
Abbildung 15 Ausschnitt aus dem Klassendiagramm mit weiteren Klassen des <i>PopulationManager</i> .....	31
Abbildung 16 Aktivitätsdiagramm zum Erstellen der initialen Population.....	32
Abbildung 17 Aktivitätsdiagramm zum Beenden der Evaluation eines Agenten.....	33
Abbildung 18 Ausschnitt 1 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation .....	33
Abbildung 19 Ausschnitt 2 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation .....	34
Abbildung 20 Ausschnitt 3 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation .....	34
Abbildung 21 Ausschnitt 4 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation .....	35
Abbildung 22 Ausschnitt 5 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation .....	35
Abbildung 23 Ablauf der Evaluation eines Agenten.....	36
Abbildung 24 Darstellung eines neuronalen <i>feed-forward</i> Netzes mit drei verschiedenen Schichten .....	37
Abbildung 25 Darstellung eines neuronalen <i>feed-forward</i> Netzes ohne Schichtenarchitektur (links) und einem neuronalen <i>recurrent</i> Netz ohne Schichtenarchitektur (rechts).....	37
Abbildung 26 Wahrheitstabelle für die XOR-Funktion.....	39
Abbildung 27 Nichtlineare Separation der XOR-Funktion (Burkill, 2016) .....	40
Abbildung 28 Das Startgenom für das XOR-Problem .....	40
Abbildung 29 Implementierung der Klasse <i>XOR</i> .....	41

Abbildung 30 Globale Variablen der Klasse <i>XOR_Agent</i> .....	42
Abbildung 31 <i>Start()</i> Methode der Klasse <i>XOR_Agent</i> .....	42
Abbildung 33 <i>Start()</i> Methode der Klasse <i>XORCallback</i> .....	43
Abbildung 32 <i>Update()</i> Methode der Klasse <i>XOR_Agent</i> .....	43
Abbildung 34 <i>InitNewAgent()</i> Methode der Klasse <i>XORCallback</i> .....	44
Abbildung 35 <i>AllAgentsKilledCallback()</i> Methode der Klasse <i>XORCallback</i> .....	44
Abbildung 36 Benötigte Generationen zur Lösung des XOR-Problems .....	46
Abbildung 37 Minimal benötigte Struktur zur Lösung des XOR-Problems .....	46
Abbildung 38 Kapsel, die einen Agenten repräsentiert .....	47
Abbildung 39 Die verwendete Strecke im Projekt Autonomes Fahren .....	47
Abbildung 40 Ausrichtung der Sensoren eines Agenten .....	48
Abbildung 41 Implementierung des Agenten .....	49
Abbildung 42 <i>Start()</i> Methode der Kapsel .....	49
Abbildung 43 Ausschnitt 1 aus der <i>Update()</i> Methode der Kapsel .....	50
Abbildung 44 Ausschnitt 2 aus der <i>Update()</i> Methode der Kapsel .....	51
Abbildung 45 <i>OnTriggerEnter()</i> Methode der Kapsel.....	52
Abbildung 46 Berechnung des Fitnesswertes einer Kapsel.....	52
Abbildung 47 Einfache Version der verwendeten Strecke .....	54
Abbildung 48 Höchster und durchschnittlicher Fitnesswert der Agenten in jeder Generation im Projekt Autonomes Fahren .....	56
Abbildung 49 Höchster und durchschnittlicher Fitnesswert der Agenten in jeder Generation im Projekt Autonomes Fahren mit einem festen Zeitintervall .....	58
Abbildung 50 Anzahl verschiedener Spezies pro Generation im Projekt Autonomes Fahren.....	59
Abbildung 51 Titelbild von <i>Super Mario Bros.</i> (Nintendo). .....	60
Abbildung 52 Ausschnitt aus dem implementierten Mario Level .....	63
Abbildung 53 Flagge die das Ende des implementierten Levels markiert .....	63
Abbildung 54 <i>Update()</i> Methode eines Mario .....	65
Abbildung 55 Unterteilung des sichtbaren Bereiches in einzelne Blöcke.....	66
Abbildung 56 <i>OnTriggerEnter()</i> Methode eines Mario .....	67
Abbildung 57 Berechnung des Fitnesswertes eines Mario.....	67
Abbildung 58 Höchster und durchschnittlicher Fitnesswert der Marios in jeder Generation .....	68
Abbildung 59 Beispiel für einen herausfordernden Abschnitt im implementierten Mario Level .....	69
Abbildung 60 Anzahl verschiedener Spezies pro Generation im Mario Projekt .....	70

# 1 Motivation

Im Jahr 2015 hat der Nutzer Seth Bling ein Video mit dem Namen *MarI/O - Machine Learning for Video Games* auf der Plattform YouTube hochgeladen. In diesem Video zeigt er, wie er ein neuronales Netz trainiert, welches die Figur Mario im Spiel *Super Mario World* steuert. Ziel von Mario ist, das erste Level des Spiels erfolgreich zu absolvieren.

Zu Beginn des Trainings werden kleine neuronale Netze verwendet, die nicht sehr erfolgreich sind. Mario schafft es in vielen Fällen nicht einmal, sich von der Stelle zu bewegen. Je länger die neuronalen Netze trainiert werden, desto besser wird Mario. Schlussendlich schafft es Mario, das Level mit einer beeindruckenden Präzision zu lösen. Der Algorithmus, der in diesem Video verwendet wird, ist der NEAT Algorithmus. NEAT steht für *NeuroEvolution of Augmenting Topologies* und ist ein neuroevolutionärer Algorithmus (Bling, 2015).

Neuroevolutionäre Algorithmen stellen ein alternatives Trainingsverfahren für neuronale Netze zu dem sehr bekannten *Backpropagation* Algorithmus dar. Algorithmen, die zu dieser Art gehören, sind von der natürlichen Evolution inspiriert und können auf verschiedene Optimierungsprobleme angewendet werden.

Auch in der Spieleentwicklung können neuroevolutionäre Algorithmen in verschiedenen Bereichen eingesetzt werden. Eine der historisch ersten Nutzungsmöglichkeiten besteht in der *state/action evaluation*. In diesem Bereich werden neuroevolutionäre Algorithmen zur Einschätzung des aktuellen Spielzustandes eingesetzt. Es erfolgt eine Auswertung aller möglichen Spielzüge und für jeden Spielzug wird eine Gewinnprognose erstellt. Sind alle Spielzüge ausgewertet, wird der Zug mit der höchsten Gewinnprognose ausgewählt und ausgeführt (Risi & Togelius, 2015).

Diese Vorgehensweise ist nicht bei allen Spielen möglich. Dies betrifft Spiele, die so viele verschiedene Aktionen ermöglichen, dass die Bewertung aller Spielzüge zu viel Rechenzeit benötigt. Die Methode kann auch nicht angewendet werden, wenn es keine Möglichkeit gibt, den durch eine Aktion entstehenden Spielzustand verlässlich zu berechnen. Somit ist es nicht möglich, die Gewinnprognose eines Spielzugs zu ermitteln. Dies ist zum Beispiel der Fall, wenn kein Zugriff auf den Programmcode möglich ist (Risi & Togelius, 2015).

Trotzdem können auch in solchen Spielen Spielzüge oder Charaktere mittels eines neuronalen Netzes gesteuert werden. Erhält das neuronale Netz eine Beschreibung des aktuellen Spielzustandes, kann es eine oder mehrere Aktionen auswählen, die dann ausgeführt werden. Dies wird auch *direct action selection* genannt. Mit entsprechenden Trainingsmethoden wie dem NEAT Algorithmus ist es möglich, neuronale Netze zu erzeugen, die besonders erfolgreich in diesen Spielen sind. Mit NEAT können sogar unterschiedliche Lösungsstrategien für ein Spiel entwickelt werden (Risi & Togelius,

2015). Die *direct action selection* wird sowohl in dem auf YouTube veröffentlichten Video als auch in den später vorgestellten Beispielprojekten verwendet.

Es gibt noch weitere Anwendungsmöglichkeiten im Bereich der Spieleentwicklung. Neuroevolution kann eingesetzt werden, um die Strategie eines Gegners zu prognostizieren. Dies wurde zum Beispiel im Spiel *Texas Hold'em Poker* eingesetzt und führte zu einer Steigerung der Gewinnrate. Ebenfalls können neuroevolutionäre Algorithmen für die Generierung von Spieleinhalten verwendet werden. Es können unter anderem neue Level, Gegenstände im Spiel und Spielregeln generiert werden (Risi & Togelius, 2015).

## 1.1 Problemstellung

Im vorherigen Kapitel wurde aufgezeigt, wie neuroevolutionäre Algorithmen in der Spieleentwicklung verwendet werden können. Einer der bekanntesten Vertreter ist der NEAT Algorithmus. NEAT ermöglicht es, neuronale Netze für verschiedene Optimierungsprobleme zu trainieren. Die neuronalen Netze können unter anderem für die Steuerung von Charakteren in Spielen eingesetzt werden.

Der NEAT Algorithmus ist zudem die Basis für verschiedene Spezialisierungen. Diese ermöglichen den Einsatz von NEAT auch in anderen Bereichen, zum Beispiel der Prognose von Strategien, Generierung von Spieleinhalten und der *State/Action evaluation* (Risi & Togelius, 2015).

Der Einstieg in die Thematik neuroevolutionäre Algorithmen ist für fachfremde Personen nicht einfach. Voraussetzung sind Kenntnisse im Bereich der neuronalen Netze und evolutionären Algorithmen. Zudem kann die Implementierung der verschiedenen Algorithmen sehr komplex und schwierig sein. Informationen über deren Funktionsweise gibt es oft nur in entsprechenden wissenschaftlichen Publikationen. Für Spieleentwickler, die solche Algorithmen in ihren Projekten einsetzen, ist dies mit einem sehr großen Aufwand verbunden.

Auch der NEAT Algorithmus ist diesbezüglich keine Ausnahme. Die Implementierung des Algorithmus ist sehr aufwändig. Es sind viele Feinheiten zu beachten, die für eine gute Performance nötig sind. Dies steigert die Komplexität beträchtlich.

## 1.2 Ziel der Arbeit

Zwar ist der NEAT Algorithmus aufwändig zu implementieren, er hat jedoch den großen Vorteil, dass er mit entsprechender Parametrisierung auf verschiedene Probleme angewendet werden kann. Durch diese Eigenschaften bietet es sich an, den Algorithmus in einer Library zu implementieren. Somit muss der Algorithmus nur einmalig implementiert werden. Für die Nutzung in verschiedenen Projekten ist nur die Anpassung aufgabenspezifischer Parameter erforderlich.



Der Algorithmus ist bereits in vielen Sprachen verfügbar. Die originale Library ist in C++ implementiert. Auf der offiziellen Nutzerseite von NEAT verweisen die Entwickler auch auf Implementierungen in Java, Matlab, Delphi und C# (Stanley, 2015).

Für Spieleentwickler, welche die Entwicklungsumgebung Unity verwenden, gibt es keine offizielle Implementierung. Ziel dieser Arbeit ist, den NEAT Algorithmus als Library für die Entwicklungsumgebung Unity zur Verfügung zu stellen.

Spieleentwickler sollen mit dem Algorithmus über eine einfach zu nutzende Schnittstelle interagieren können. Somit ist es nicht mehr nötig, den Algorithmus selbst zu implementieren. Die durch den Algorithmus erstellten neuronale Netze können die Steuerung unterschiedlicher Komponenten in einem Spiel übernehmen.

Der Algorithmus selbst wird auch in einem Unity Projekt implementiert. Dies bietet die Möglichkeit, die fertige Implementierung als *AssetBundle* zu exportieren, welches Spieleentwicklern zur Verfügung gestellt werden kann. Hierfür eignet sich besonders der Unity eigene *Asset Store*. In diesem können *AssetBundles* hochgeladen werden, die dann mit wenig Aufwand in andere Projekte integriert werden können.

### 1.3 Struktur der Arbeit

Zu Beginn dieser Arbeit wird auf die benötigten Grundlagen eingegangen. Es wird sowohl das Prinzip der neuronalen Netze als auch der neuroevolutionären Algorithmen erläutert. Der Fokus dieser Arbeit liegt auf dem NEAT Algorithmus. Die Funktionalität von diesem wird in Kapitel 2.3 ausführlich erläutert.

In Kapitel 3 wird die Software Architektur der Library vorgestellt. Es wird die Schnittstelle, die verwendeten Klassen und der vollständige Ablauf der Methoden aufgezeigt. Die genaue Implementierung wird nicht vorgestellt, da die Funktionsweise durch die Diagramme ersichtlich und somit redundant ist.

In Kapitel 4 werden drei Beispielprojekte mit der fertigen Library implementiert. Das erste Projekt wird verwendet, um die Funktionalität der Library zu beweisen und einen Performancevergleich mit der originalen Implementierung zu ermöglichen. Das zweite und dritte Projekt veranschaulichen die Verwendung der Library in einem Projekt und den benötigten Aufwand zur Integration der Library.

Kapitel 5 beinhaltet eine Zusammenfassung der Ergebnisse und einen Ausblick auf die noch ausstehenden Aufgaben.

Der Programmcode, die Diagramme und die Beispielprojekte sind auf GitHub unter folgendem Link verfügbar:

<https://github.com/simonhauck/>

## 2 Grundlagen

In diesem Kapitel werden die benötigten Grundlagen für diese Arbeit vorgestellt.

In Kapitel 2.1 wird die Funktionsweise neuronaler Netze dargestellt. Danach werden in Kapitel 2.2 die Prinzipien von neuroevolutionären Algorithmen vorgestellt, die evolutionäre Algorithmen zur Optimierung von neuronalen Netzen nutzen. Zuletzt wird in Kapitel 2.3 auf den in dieser Arbeit verwendeten Algorithmus eingegangen und dessen Umsetzung erläutert.

### 2.1 Neuronale Netze

Die Idee eines künstlichen neuronalen Netzes, im englischen *Artificial neural network*, wird bereits seit vielen Jahren erforscht. Die Anfänge liegen in den 1940er Jahren mit den Arbeiten von McCulloch und Walter Pitts (Frochte, 2018).

Künstliche neuronale Netze sind von biologischen neuronalen Netzen wie dem menschlichen Gehirn inspiriert. Langfristiges Ziel ist, menschenähnliche Performance, zum Beispiel im Bereich der Bild- und Spracherkennung, zu erreichen (Lippmann, 1987). Als Beispiel hierfür kann die Gesichtserkennung genannt werden. Klassische Algorithmen stoßen in diesem Bereich an ihre Grenzen, sobald gewisse Voraussetzungen an die Bildqualität nicht gegeben sind. Der Mensch kann hingegen Gesichter auch unter schlechten Bedingungen erfolgreich zuordnen. Der Fachbereich der künstlichen neuronalen Netze beschäftigt sich mit Methoden und Verfahren, um die positiven Eigenschaften der biologischen neuronalen Netze auf Computerprogramme zu übertragen.

Weitere Einsatzgebiete für künstliche neuronale Netze finden sich im Bereich der Mustererkennung, Kategorisierung, Funktionsapproximation, Optimierung und Prognose (Scherer, 1997). Im weiteren Verlauf dieser Arbeit bezieht sich der Begriff neuronale Netze auf die künstlichen neuronalen Netze.

### 2.1.1 Aufbau und Funktionsweise neuronaler Netze

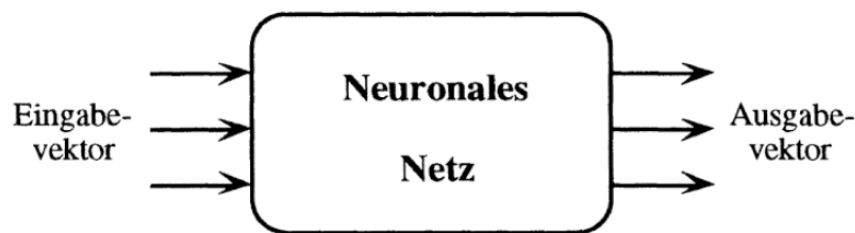


Abbildung 1 Neuronales Netz als Abbildungsvorschrift (Scherer, 1997)

Trotz verschiedener Anwendungsbereiche werden neuronale Netze immer in ähnlicher Weise eingesetzt. Ein neuronales Netz kann als eine mathematische Abbildungsvorschrift verstanden werden, die unterschiedlichste Funktionen abbilden kann. Für diese gibt es eine bestimmte Menge an Eingaben, die in einem Eingabevektor kodiert sind und zu einer Menge an Ausgaben führen, die in einem Ausgabevektor kodiert sind (Scherer, 1997). Die Funktionsweise ist in Abbildung 1 dargestellt und wird mit folgendem Beispiel genauer erläutert. Das neuronale Netz soll die Farbe einer auf einem Bild dargestellten Ampel auswerten. Der benötigte Eingabevektor für dieses Problem kann beispielsweise aus den einzelnen Pixeln des Bildes bestehen. Der Ausgabevektor kann in diesem Beispiel aus drei Elementen bestehen, die je eine Farbe repräsentieren. Die Ausgabewerte können den Wert  $0$  und  $1$  annehmen. Ist der Ausgabewert eines Ele-

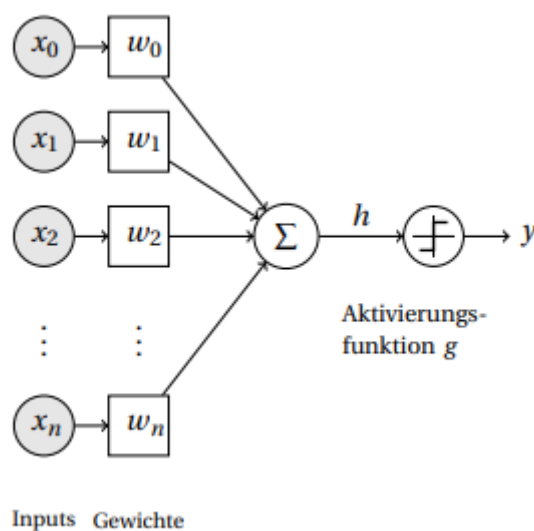


Abbildung 2 Mathematisches Modell eines Neurons (Frochte, 2018)

ments  $1$ , leuchtet die entsprechende Farbe auf der Ampel, ist der Wert  $0$  so leuchtet die Farbe nicht.

Um solche Probleme zu lösen, besteht ein neuronales Netz aus mehreren künstlichen Neuronen. Neuronen sind einfache Recheneinheiten, die über Verbindungen, die gerichtet und gewichtet sind, miteinander verbunden sind. Die Neuronen lassen sich in

drei verschiedene Kategorien einteilen. Jedes neuronale Netz besitzt Input- und Output-Neuronen. Zudem kann ein neuronales Netz beliebig viele Hidden-Neuronen besitzen. Soll das neuronale Netz eine Berechnung durchführen, wird der Eingabevektor in die Input-Neuronen gesetzt. Für jedes Element im Eingabevektor gibt es ein Input-Neuron, für jedes Element im Ausgabevektor ein Output-Neuron.

Nach Setzen der Input-Werte erfolgt die Berechnung der einzelnen Neuronen. Die Berechnung in einem einzelnen Neuron ist in Abbildung 2 dargestellt. Jedes Neuron, mit Ausnahme der Input-Neuronen, besitzt ein oder mehrere Eingabewerte. Diese sind in Abbildung 2 mit  $x_0$  bis  $x_n$  beschriftet. Die Eingabewerte sind das Ergebnis anderer Neuronen, zu denen eine Verbindung besteht. Jede Verbindung hat ein entsprechendes Gewicht  $w_i$ . Im ersten Schritt der Berechnung werden die Eingabewerte mit den entsprechenden Gewichten multipliziert und aufsummiert. Das Ergebnis ist die Funktion  $h$  mit

$$h(x) = \sum_{i=0}^n w_i * x_i$$

Das Ergebnis dieser Funktion ist wiederum der Eingabewert für die Aktivierungsfunktion  $g$  (Frochte, 2018). Es gibt verschiedene Aktivierungsfunktionen. In den später vorgestellten Beispielprojekten wird eine angepasste Sigmoid-Funktion verwendet. Die Sigmoid-Funktion ist in Abbildung 3 dargestellt.

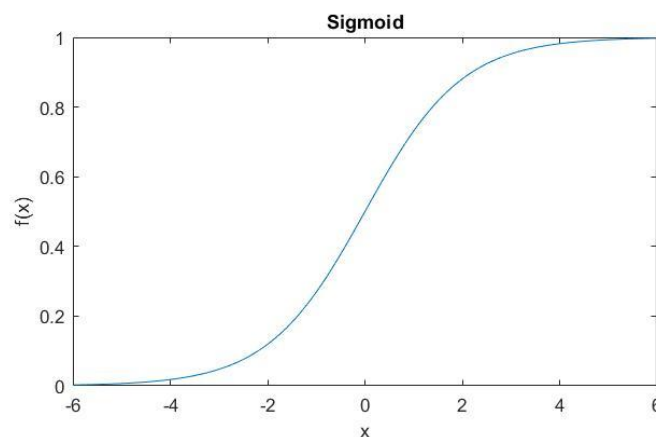
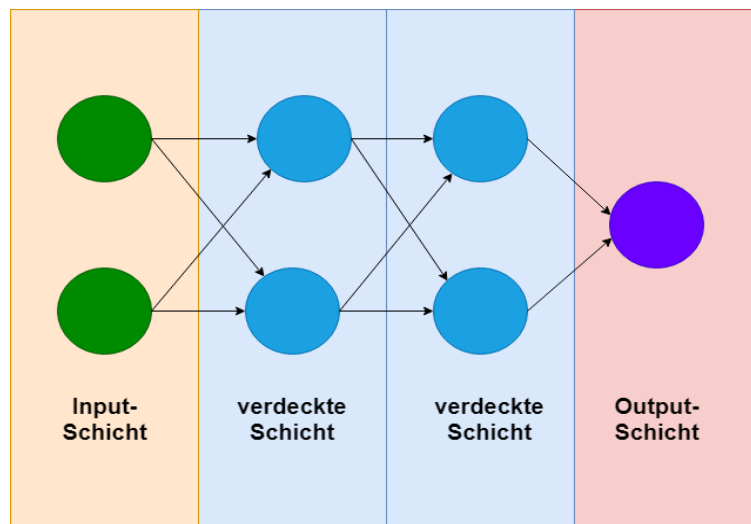


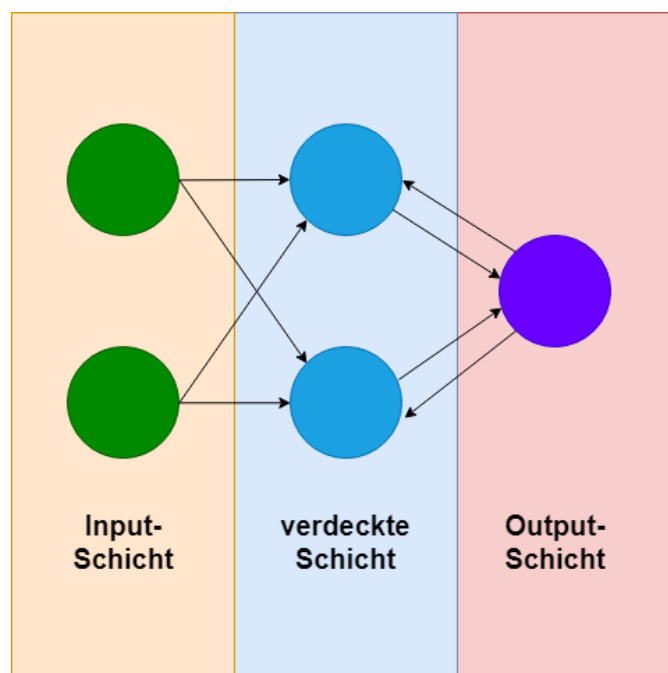
Abbildung 3 Darstellung der Sigmoid-Funktion

Das Ergebnis der Aktivierungsfunktion wird im Neuron gespeichert. Ist das Neuron ein Hidden-Neuron, kann das Ergebnis anderen Neuronen als Eingabewert dienen. Ist das Neuron ein Output-Neuron, wird das Ergebnis in den Ausgabevektor übertragen.

Die einzelnen Neuronen werden durch Verbindungen zu einem Netz zusammengefügt. Hierfür gibt es zwei grundlegend verschiedene Strukturen. In der ersten Struktur werden die Neuronen zu einem *feed-forward* Netzwerk zusammengefügt. In diesem haben alle Verbindungen dieselbe Richtung und es entsteht so ein azyklischer Graph. Ein Beispiel hierfür ist in Abbildung 4 dargestellt. In diesem Beispiel sind alle Verbindungen nach rechts gerichtet. Ein *feed-forward* Netzwerk repräsentiert eine Funktion mit den

Abbildung 4 Neuronales *feed-forward* Netz in Schichtenarchitektur

gegebenen Eingabewerten und besitzt, außer den Gewichten der Verbindung, keinen eigenen internen Zustand. Das bedeutet, dass bei Verwendung der gleichen Eingabe, immer dasselbe Ergebnis berechnet wird.

Abbildung 5 Neuronales *recurrent* Netz in Schichtenarchitektur

Die zweite Struktur stellen die *recurrent* Netze dar. Ein mögliches *recurrent* Netz ist in Abbildung 5 dargestellt. *Recurrent* Netze haben zyklische Verbindungen und können Output-Werte als Input-Werte in der nächsten Berechnung verwenden. In dieser Abbildung ist das in der verdeckten Schicht sichtbar. Die Neuronen verwenden nicht nur die Werte der Input-Schicht, sondern auch den Wert des Neurons in der Output-Schicht. Diese gibt den Wert der letzten durchgeführten Berechnung zurück. Das bedeutet, dass bei einer Berechnung das Ergebnis nicht wie in einem *feed-forward* Netz nur von

den Input-Werten abhängt, sondern auch von dem internen Zustand des Netzes. Dieser kann durch vorherige Berechnungen beeinflusst sein.

Die *feed-forward* Netze sind meistens in Schichten angeordnet. Beispiel hierfür ist das neuronale Netz in Abbildung 4. Dieses ist in vier Schichten angeordnet. Bei einer solchen Struktur sind die Eingabewerte einer Schicht die berechneten Ergebnisse der Neuronen in der vorherigen Schicht. Solche Netzwerke haben immer eine Input-Schicht und Output-Schicht, sowie beliebig viele verdeckte Schichten. In der Input-Schicht befinden sich die Input-Neuronen, in der Output-Schicht die Output-Neuronen und in den verdeckten Schichten die Hidden-Neuronen. Eine verdeckte Schicht mit genügend Hidden-Neuronen ist ausreichend, um jede stetige Funktion mit beliebiger Genauigkeit abzubilden. Besitzt ein neuronales Netz zwei verdeckte Schichten, kann auch jede unstetige Funktion abgebildet werden (Russell & Peter, 1994).

### 2.1.2 Lernprozess von neuronalen Netzen

Im vorherigen Kapitel wurden der Aufbau und die Funktionsweise von neuronalen Netzen erklärt. Für die erfolgreiche Lösung eines Problems, zum Beispiel dem Erkennen einer leuchtenden Ampelfarbe, benötigt das neuronale Netz eine entsprechende Struktur und speziell an dieses Problem angepasste Gewichte in den Verbindungen.

Da dies problemspezifische und zu Beginn unbekannte Parameter sind, wird eine Trainingsphase benötigt. In dieser wird ein Lernverfahren durchgeführt, welches die zu Beginn zufällig initialisierten Parameter optimieren soll.

Eines der bekanntesten Lernverfahren ist der *Backpropagation* Algorithmus. Dieser Algorithmus ist in der Lage, in einem iterativen Prozess die Gewichte eines mehrschichtigen neuronalen Netzes zu optimieren (Rojas, 1996). Die genaue Funktionalität des Algorithmus wird in dieser Arbeit nicht betrachtet. Aber dieser Algorithmus kann, im Gegensatz zu dem in dieser Arbeit verwendeten Verfahren, nicht die Struktur des neuronalen Netzes anpassen. Wenn der *Backpropagation* Algorithmus verwendet wird, muss die Struktur des neuronalen Netzes zuvor festgelegt werden. Um eine passende Struktur des neuronalen Netzes zu wählen, muss entweder Expertenwissen in der entsprechenden Domäne vorhanden sein oder der Parameter muss experimentell ermittelt werden (Stanley, 2017). Die Optimierung der Gewichte erfolgt in diesem Verfahren iterativ. Es gibt ein neuronales Netz, welches in der Trainingsphase Berechnungen durchführt. Danach wird der Fehler zwischen dem berechneten und dem erwarteten Ergebnis gebildet. In Abhängigkeit von diesem Fehler werden die Gewichte so angepasst, dass der Fehler geringer wird (Rojas, 1996). Dies wird für viele Iterationen wiederholt, bis das gewünschte Ergebnis vorhanden ist.

In dieser Arbeit wird ein neuroevolutionärer Algorithmus als Lernverfahren verwendet. Die Funktionsweise des Verfahrens wird in den folgenden Kapiteln erläutert.

## 2.2 Neuroevolutionäre Algorithmen

In der Informatik sind nicht nur neuronale Netze von der Natur inspiriert. In dieser Arbeit wird der Algorithmus *NeuroEvolution of Augmenting Topologies* (NEAT) verwendet, der zur Gruppe der neuroevolutionären Algorithmen gehört. Diese verwenden evolutionäre Algorithmen, um neuronale Netze zu optimieren. Evolutionäre Algorithmen sind von der natürlichen Evolution inspirierte Optimierungsverfahren. Es gibt eine Vielzahl an evolutionären Algorithmen, die unterschiedliche Aspekte der Evolution umsetzen. Gemeinsam ist allen, dass eine künstliche Evolution simuliert wird, um möglichst gute Näherungslösungen für ein Optimierungsproblem zu erhalten (Wicker, 2015). In diesem Kapitel wird der übergeordnete Ablauf eines evolutionären Algorithmus beschrieben.

### 2.2.1 Natürliche Evolution

Die natürliche Evolution dient als Inspiration für evolutionäre Algorithmen. Aus diesem Grund werden in diesem Kapitel die Prozesse der natürlichen Evolution vorgestellt, die auf die evolutionären Algorithmen übertragen werden. Die vorgestellten biologischen Prozesse sind stark vereinfacht erklärt, da ein genaues Verständnis der technischen und biologischen Mechanismen für diese Arbeit nicht notwendig ist.

Die natürliche Evolution wird am Beispiel einer Population von Giraffen erläutert. Diese besteht aus vielen einzelnen Individuen. Jede Giraffe hat ein eigenes Genom. Das Genom besteht aus einzelnen Genen, die das Erscheinungsbild der Giraffe prägen. Ein Teil der Gene kann zum Beispiel für die Fellfarbe, ein anderer Teil für die Länge des Halses verantwortlich sein (Wicker, 2015).

Die Umwelt stellt die einzelnen Individuen vor zahlreiche Aufgaben. Individuen, die besser an die Umwelt angepasst sind, haben eine größere Chance zu überleben und mehr Nachkommen zu erzeugen. Dies wird auch Selektion genannt (Wicker, 2015). Im vorgestellten Beispiel ist eine Giraffe mit einem längeren Hals besser an die Umwelt angepasst, da sie mehr Futter erreichen kann.

Die Nachkommen einer Population werden durch sexuelle Paarung erzeugt. Das Genom eines Nachkommen entsteht durch Rekombination der Eltern-Genome. Dabei entstehen keine neuen Gene, sondern die bestehenden Gene werden neu kombiniert (Wicker, 2015). Es besteht die Chance, dass das neu erzeugte Genom und das daraus entstehende Individuum besser an die Umwelt angepasst ist als seine Eltern. Es kann aber auch der Fall eintreten, dass das neue Individuum schlechter angepasst ist. Langfristig werden sich nur die am besten an die Umwelt angepassten Individuen durchsetzen, da diese eine größere Chance zum Überleben und damit zur Fortpflanzung haben.

Während der Reproduktion entstehen Fehler, die zu Mutationen des Genoms führen können. Diese Mutationen sind die Grundlage für Veränderungen in der Evolution. Veränderungen entstehen meist über einen langen Zeitraum und viele kleine Mutationen

(Wicker, 2015). Eine einzelne Mutation kann sich sowohl positiv als auch negativ auf das Individuum auswirken. Durch die Selektion besteht eine höhere Chance, dass sich die Individuen fortpflanzen, bei denen sich die Mutationen positiv ausgewirkt haben.

Durch die Selektion, Reproduktion und Mutation ist eine Population in der Lage, sich an Veränderungen in der Umwelt anzupassen und zu behaupten.

### 2.2.2 Evolutionäre Algorithmen

Evolutionäre Algorithmen verwenden die vorgestellten Prinzipien der Evolution zur Optimierung eines Problems. Im folgenden Abschnitt wird erläutert, wie diese Prinzipien in einem evolutionären Algorithmus angewendet werden.

Auch in einem evolutionären Algorithmus gibt es eine Population. Diese besteht aus Individuen, die in dieser Arbeit als Agent bezeichnet werden. Wie ein natürliches Individuum besitzt ein Agent ein eigenes Genom (Wicker, 2015). Bei neuroevolutionären Algorithmen sind in diesem die Struktur des neuronalen Netzes und die Gewichte der Verbindungen kodiert.

Evolutionäre Algorithmen durchlaufen vier Phasen, diese heißen Evaluation, Selektion, Reproduktion und Mutation. Ein Durchlauf dieser Phasen wird als eine Generation bezeichnet (Wicker, 2015). In den folgenden Abschnitten wird der Ablauf einer Generation erläutert.

Zu Beginn des Algorithmus wird eine initiale Population erzeugt. Hierfür gibt es je nach Algorithmus verschiedene Verfahren. Die Population besteht aus vielen unterschiedlichen Agenten mit unterschiedlichen Genomen.

Die erste Phase ist die Evaluationsphase. In dieser versuchen die einzelnen Agenten, das gegebene Optimierungsproblem zu lösen. Klassische Optimierungsprobleme haben meist ein klar definiertes Bewertungskriterium, mit dem die Qualität der Lösung und damit des Agenten gemessen werden kann. Die Qualität der Lösung wird im Folgenden als Fitnesswert bezeichnet. Die Funktion, die den Fitnesswert berechnet, wird als Fitnessfunktion bezeichnet. Am Ende der Evaluationsphase wird der Fitnesswert für jeden Agenten berechnet. Da Agenten zu Beginn nicht optimiert sind, werden die Fitnesswerte sehr gering sein. Trotzdem wird es Agenten geben, die besser als andere sind (Wicker, 2015).

Der nächste Schritt im Algorithmus ist die Selektion. In dieser Phase wird basierend auf dem Fitnesswert jedem Agenten der Population eine Anzahl an Nachkommen zugewiesen. Je höher der Fitnesswert, desto mehr Nachkommen werden durch den Agenten erzeugt (Wicker, 2015).

Die Nachkommen werden im dritten Schritt, der Reproduktionsphase, erzeugt. Das Genom eines Nachkommen entsteht durch Rekombination mehrerer Eltern-Genome. Die Rekombination dient wie in der biologischen Evolution der Durchmischung der Gene (Wicker, 2015).



Im letzten Schritt erfolgt die Mutation der Gene. Diese verändert das Genom des neuen Agenten. Meist sind dies nur kleine Änderungen, um die Vererbung der Elterneigenschaften nicht zu stark zu stören (Wicker, 2015). In Bezug auf neuronale Netze kann zum Beispiel bei einer Mutation das Gewicht einer Verbindung minimal geändert werden.

Die neuen Agenten werden anschließend in die Population integriert. Da diese eine feste Größe hat, gibt es hierfür verschiedene Strategien. Entweder werden einzelne Agenten aus der Population entfernt und entsprechend neu erzeugte Agenten hinzugefügt oder die gesamte Population wird durch neu erzeugte Agenten ersetzt (Wicker, 2015).

Der Durchlauf dieser Phasen kann beliebig oft wiederholt werden. Wie in der biologischen Evolution versuchen die Agenten, sich immer besser der Umgebung anzupassen. Dies ist durch eine Steigerung des Fitnesswertes sichtbar. Je höher der Fitnesswert, desto besser ist die gefundene Lösung.

Am Ende jeder Evaluationsphase wird überprüft, ob eine Abbruchbedingung eingetreten ist. Dies kann zum Beispiel das Erreichen eines bestimmten Fitnesswertes sein, einer bestimmten Anzahl von Generationen ohne Steigerung des Fitnesswertes oder bei Erreichen der maximalen Anzahl an Iterationen (Wicker, 2015).

### 2.3 NeuroEvolution of Augmenting Topologies

Der Algorithmus *NeuroEvolution of Augmenting Topologies*, kurz NEAT, wurde an der University of Texas at Austin von Kenneth Stanley und Risto Miikkulainen entwickelt. Die Ergebnisse wurden im Jahr 2002 in einer Publikation mit dem Namen *Evolving Neural Networks through Augmenting Topologies* veröffentlicht. Diese Publikation dient, sofern nicht anderweitig gekennzeichnet, als Quelle für dieses Kapitel (Stanley & Miikkulainen, 2002).

Ziel des Algorithmus ist die Entwicklung künstlicher neuronaler Netze durch einen genetischen Algorithmus. Somit gehört NEAT zur Gruppe der neuroevolutionären Algorithmen.

NEAT entwickelt bei diesem Verfahren eine geeignete Topologie und optimiert die Gewichte für das neuronale Netz. Im Vergleich zu anderen neuroevolutionären Algorithmen, welche die Topologie nicht optimieren, findet NEAT bei verschiedenen Benchmark Tests schneller eine geeignete Lösung.

Die gute Performance von NEAT beruht auf drei wichtigen Eigenschaften:

1. Der einfachen Rekombination von Genomen mit verschiedenen Strukturen,
2. dem Beschützen von strukturellen Innovationen durch verschiedene Spezies
3. durch ein inkrementelles Wachsen einer am Anfang minimalen Struktur.

In den folgenden Kapiteln wird erklärt, wie der NEAT Algorithmus funktioniert und warum diese Prinzipien wichtig für dessen Performance sind.

### 2.3.1 Genetische Kodierung

Das genetische Kodierungsschema von NEAT wurde entworfen, um eine einfache Rekombination zwischen Genomen mit verschiedenen Strukturen zu ermöglichen. Ein einzelnes Genom ist eine lineare Repräsentation der Verbindungen in einem neuronalen Netz (Abbildung 6).

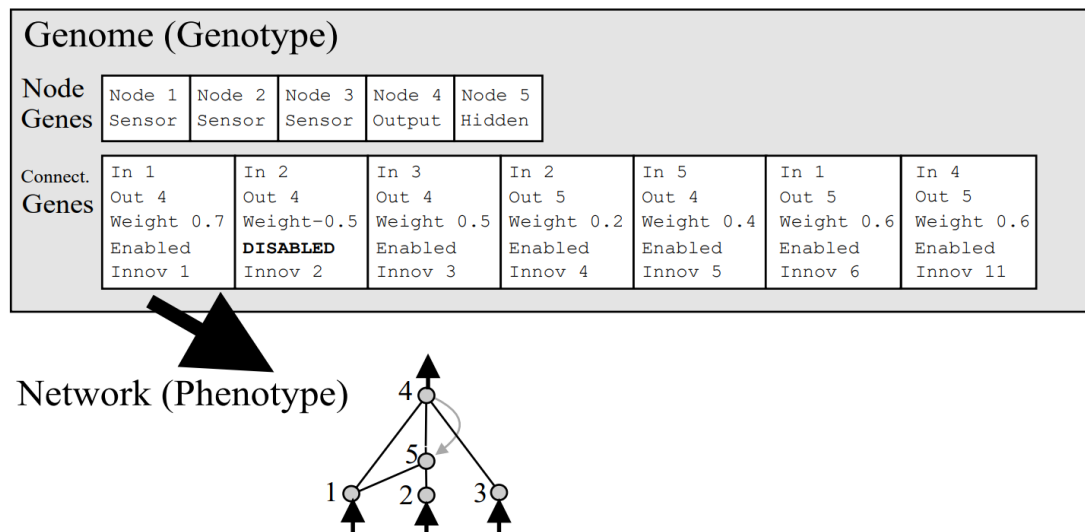


Abbildung 6 Kodierung eines Genoms mit NEAT (Stanley & Miikkulainen, 2002)

Jedes Genom besteht aus einer Liste *Node Genes* und einer Liste *Connection Genes*.

Die Liste *Node Genes* enthält eine Liste aller Input-, Output- und Hidden-Neuronen im neuronalen Netz. Jedes Neuron speichert eine ID (z.B.: Node 1, Node 2, ...) und den Typ des Neurons (Sensor/Input, Hidden, Output).

Die Liste *Connection Genes* enthält alle Verbindungen zwischen den Neuronen. Eine Verbindung enthält je eine ID für ein In- und Out-Neuron, zwischen denen die Verbindung besteht. Des Weiteren wird das Gewicht der Verbindung gespeichert und ein Aktivierungsbit, welches angibt, ob die Verbindung im neuronalen Netz verwendet werden soll. Das letzte Datenfeld einer Verbindung ist die Innovationsnummer.

Diese wird in Kapitel 2.3.3.2 noch genauer erläutert.

### 2.3.2 Mutation

Mutationen in NEAT können sowohl die Gewichte der Verbindungen als auch die Topologie des neuronalen Netzes verändern.

Die Gewichte werden wie in anderen neuroevolutionären Algorithmen mutiert. Das bedeutet, für jede Verbindung besteht eine Wahrscheinlichkeit, dass das Gewicht in dieser Generation mutiert wird oder nicht. In den meisten Fällen wird das Gewicht zufällig

minimal nach oben oder unten abgeändert. Aber es besteht auch eine kleine Wahrscheinlichkeit, dass das Gewicht zufällig neu gewürfelt wird.

Neben der Mutation der Gewichte gibt es noch zwei Arten der strukturellen Mutation. Diese ermöglichen es NEAT, die Topologie des neuronalen Netzes anzupassen.

Bei der ersten Art wird eine einzelne neue Verbindung dem Genom hinzugefügt, indem zwei bisher nicht verbundene Neuronen verbunden werden.

Das Gewicht für diese Verbindung wird zufällig gewählt und das Aktivierungsbit wird auf aktiv gesetzt (Abbildung 7).

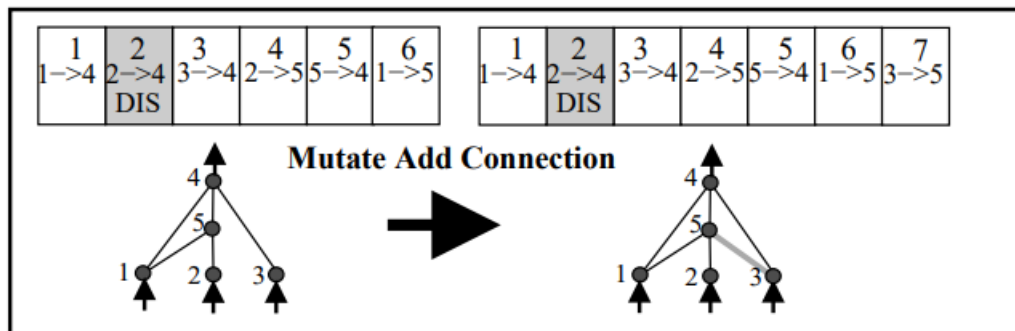


Abbildung 7 Strukturelle Mutation, die eine Verbindung hinzufügt (Stanley & Miikkulainen, 2002)

Bei der zweiten Art wird ein neues Neuron hinzugefügt. Zu diesem Zweck wird eine bestehende aktive Verbindung in der Mitte geteilt und ein neues Neuron an dieser Stelle platziert. Die alte Verbindung wird deaktiviert und zwei neue Verbindungen werden erstellt. Die erste, die in das neu erstellte Neuron geht, erhält das Gewicht 1. Die Verbindung, die von dem neuen Neuron herausgeht, erhält das Gewicht der deaktivierten Verbindung (Abbildung 8). Durch diese Art der Mutation wird der initiale Effekt des neuen Neurons minimiert. Dies hat den Vorteil, dass neue Neuronen direkt in die Topologie mit eingebunden werden und dass das Genom die neue Struktur direkt verwenden und mit der Zeit optimieren kann.

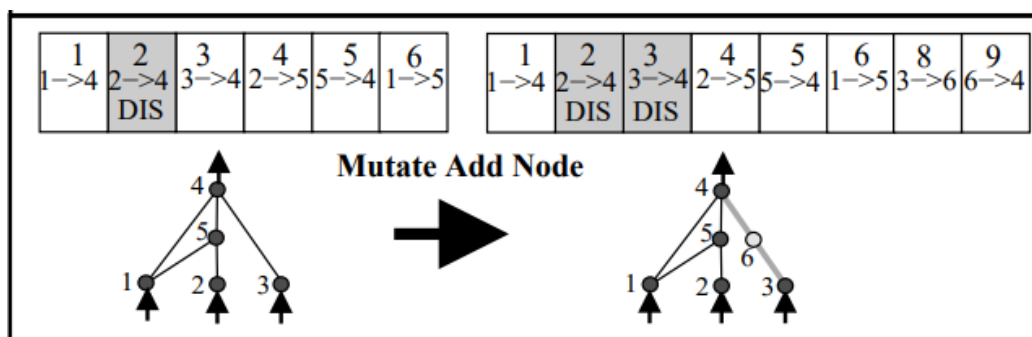


Abbildung 8 Strukturelle Mutation, die ein Neuron hinzufügt (Stanley & Miikkulainen, 2002)

### 2.3.3 Rekombination von Genomen mit verschiedenen Strukturen

Die im vorherigen Kapitel vorgestellten Arten der strukturellen Mutation führen dazu, dass die Genome in NEAT mit der Zeit immer größer werden und die Unterschiede in der Topologie immer mehr zunehmen. Es wird Genome in verschiedenen Größen, mit verschiedenen Verbindungen und unterschiedlichen Gewichten geben. Dies ist bei uneingeschränktem Wachstum wie in NEAT nicht zu vermeiden und macht die Rekombination durch das *Competing Conventions* Problem sehr komplex.

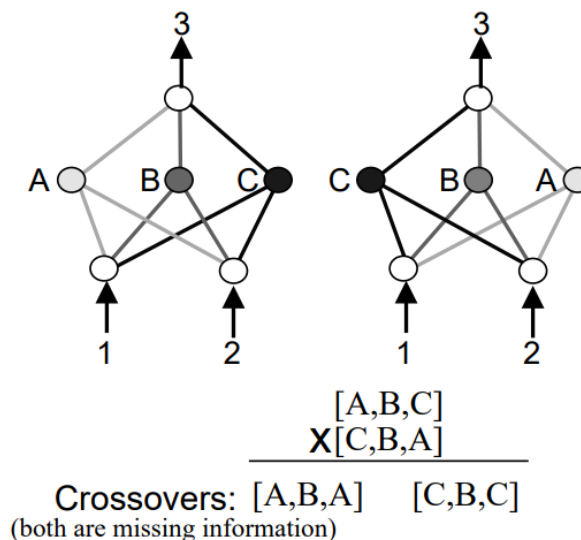


Abbildung 9 Rekombination mit dem *Competing Conventions* Problem (Stanley & Miikkulainen, 2002)

#### 2.3.3.1 Das *Competing Conventions* Problem

*Competing Conventions* ist eines der größten Probleme für neuroevolutionäre Algorithmen. Dies bedeutet, dass es mehrere Möglichkeiten gibt, dieselbe Lösung zu kodieren. Werden zwei Genome mit unterschiedlichen Kodierungen bei der Reproduktion ausgewählt, kann es zum Verlust von Informationen kommen. Dies kann dazu führen, dass die Nachkommen unbrauchbar werden.

In Abbildung 9 sind zwei einfache neuronale Netze mit je drei Hidden-Neuronen *A*, *B* und *C* dargestellt. Diese repräsentieren dieselbe Lösung, haben jedoch eine unterschiedliche Kodierung. Bei der Reproduktion wird das linke Genom [A, B, C] mit dem rechten Genom [C, B, A] gekreuzt. Das Ergebnis dieser Reproduktion kann unter anderem [A, B, A] oder [C, B, C] sein. Diese Varianten haben je ein Drittel der Informationen der Eltern verloren und sind mit hoher Wahrscheinlichkeit in Zukunft unbrauchbar.

Noch schwieriger wird es, wenn Genome mit verschiedenen Topologien oder unterschiedlichen Größen dieselbe Lösung repräsentieren.

### 2.3.3.2 Verfolgen der Gene mit historischen Markern

Das *Competing Conventions* Problem kann gelöst werden, wenn der historische Ursprung eines Genes bekannt ist. Zwei Gene, welche denselben historischen Ursprung haben, müssen auch dieselbe Struktur repräsentieren, auch wenn sie unterschiedliche Gewichte haben. Um das *Competing Conventions* Problem zu lösen, muss NEAT den historischen Ursprung von jedem Gen verfolgen.

Dies kann sehr ressourcensparend umgesetzt werden. Sobald ein neues Gen durch eine strukturelle Mutation entsteht, wird eine globale Innovationsnummer inkrementiert und diesem Gen zugewiesen. Die Innovationsnummer repräsentiert somit den chronologischen Verlauf der Mutationen.

Dies kann am Beispiel der Mutationen, die in Abbildung 7 und Abbildung 8 aufgetreten sind, dargestellt werden. Die Mutationen sind nacheinander im selben System aufgetreten. Die neue Verbindung in Abbildung 7 bekommt die Innovationsnummer 7 zugewiesen. Die zweite Mutation in Abbildung 8 fügt ein neues Neuron hinzu und im Anschluss zwei neue Verbindungen. Diese bekommen dann die Innovationsnummer 8 und 9 zugewiesen. Die Innovationsnummern werden nicht geändert und auch bei der Rekombination in die Nachkommen übernommen. Somit ist der historische Ursprung von jedem Gen bekannt.

Sollte dieselbe Mutation in einer Generation mehrmals unabhängig voneinander auftreten, würden die Gene mit der oben beschriebenen Methode unterschiedliche Innovationsnummer zugewiesen bekommen, obwohl die Mutationen dieselbe Struktur repräsentiert. Um dies zu verhindern, werden alle Mutationen einer Generation gespeichert. Sollte dieselbe Mutation in einer Generation mehrmals auftreten, wird dem neuen Gen die gespeicherte Innovationsnummer zugewiesen. Somit ist sichergestellt, dass mehrere Mutationen, welche trotzdem dieselbe Struktur repräsentieren, den gleichen historischen Ursprung haben.

### 2.3.3.3 Rekombination

Die historischen Marker machen NEAT die Rekombination verschiedener Genome, auch mit unterschiedlichen Größen und Topologien, sehr einfach, da passende Gene immer gleiche Innovationsnummern haben.

Bei der Rekombination werden Verbindungen, deren Innovationsnummern in beiden Eltern-Genomen vorkommen, *matching Genes* genannt. Verbindungen, die nur in einem der beiden Eltern-Genomen vorhanden sind, sind entweder *disjoint* oder *excess Genes*. Dies ist abhängig von der Position der Verbindung im Genom. Ist die Innovationsnummer der Verbindung kleiner als die größte Innovationsnummer des anderen Eltern-Genoms, ist das Gen ein *disjoint Gene*, andernfalls ist es ein *excess Gene*.

Beim Erstellen des Kinder-Genoms werden alle Neuronen des Eltern-Genoms mit dem höheren Fitnesswert übernommen. Bei den Verbindungen wird zwischen den *matching*

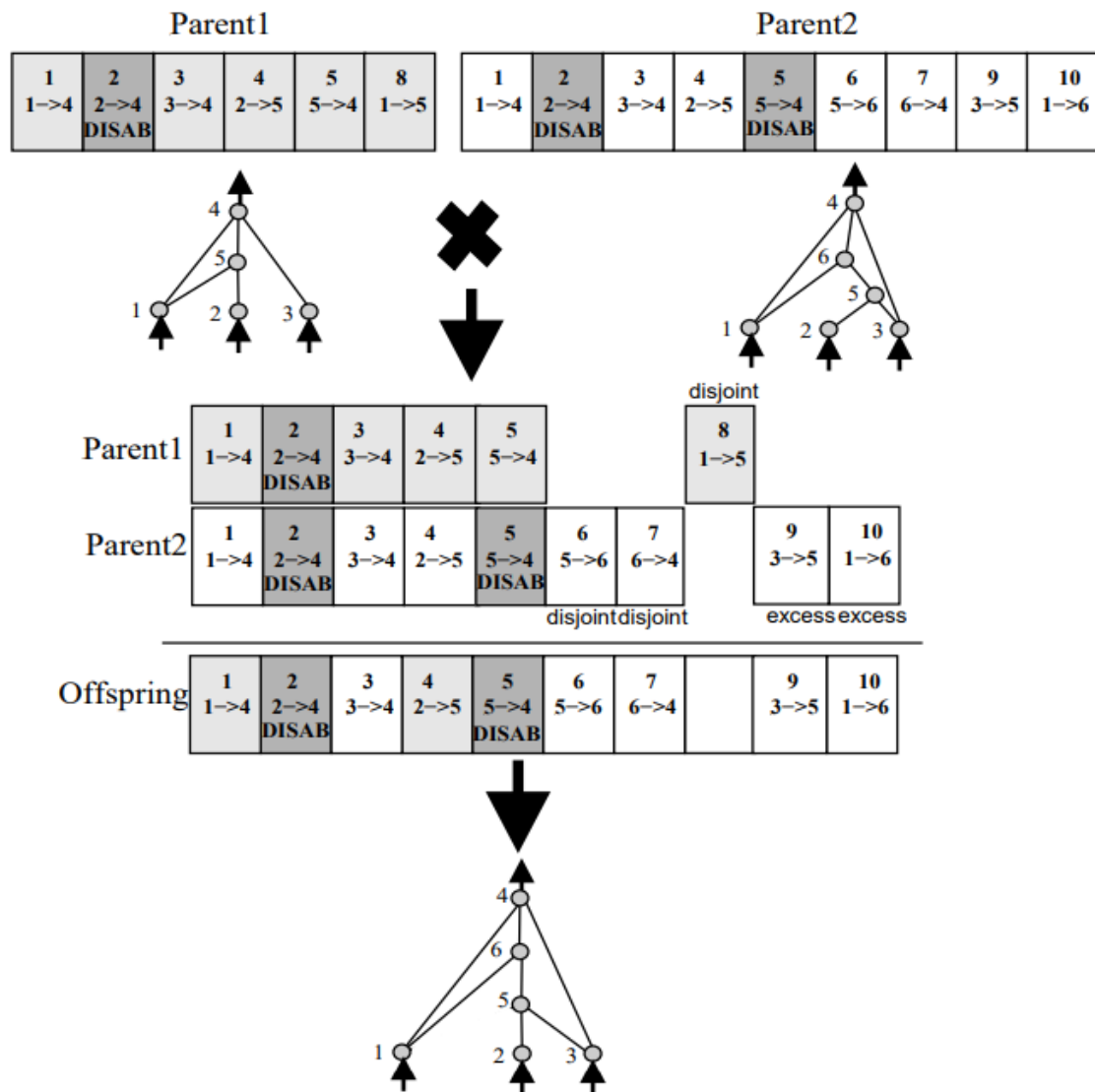


Abbildung 10 Rekombination zweier Genome mit NEAT (Stanley &amp; Miikkulainen, 2002)

Genes und den *disjoint* und *excess* Genes unterschieden. *Disjoint* und *excess* Genes, also die Verbindungen, die nur in einem Eltern-Genom vorkommen, werden nur von dem Eltern-Genom übernommen, das den höheren Fitnesswert besitzt. Bei den *matching* Genes wird zufällig entschieden, von welchem Eltern-Genom die Verbindung übernommen wird. Ein Beispiel für eine solche Rekombination ist in Abbildung 10 zu sehen. Wenn alle Kinder-Genome erstellt wurden, wird die alte Population durch die neu erstellten Genome ersetzt.

Mit diesem Verfahren kann NEAT die Rekombination zweier linear dargestellter Genome durchführen, die unterschiedliche Topologien haben können und muss bei diesem Verfahren keine aufwändige Analyse der Topologie machen.

### 2.3.4 Beschützen von strukturellen Innovationen durch verschiedene Spezies

Durch die bereits vorgestellten Möglichkeiten der Mutation und der verwendeten Rekombination kann NEAT eine Population mit unterschiedlichen Topologien und strukturellen Innovationen bilden.

Aber eine Population kann solche strukturellen Innovationen und unterschiedliche Topologien nicht allein erhalten. Kleine Strukturen werden, bedingt durch weniger Kombinationsmöglichkeiten, schneller optimiert als große Strukturen. Zudem senken strukturelle Mutationen, zum Beispiel das Hinzufügen eines Neurons oder einer neuen Verbindung, initial den Fitnesswert des Genoms.

Dies hat zur Folge, dass Strukturen, die kürzlich gewachsen sind, eine geringe Chance haben, sich durchzusetzen, auch wenn die neue Struktur unverzichtbar für die Lösung des gegebenen Problems ist.

Dieses Problem wird gelöst, indem die Population in verschiedene Spezies unterteilt wird. Jedes Genom wird einer Spezies zugewiesen, deren Topologie der des Genoms ähnlich ist. Ziel dieser Maßnahme ist, dass die Genome nicht mehr mit der ganzen Population konkurrieren müssen, sondern nur mit den Genomen der eigenen Spezies. So können die Genome die neuen Strukturen entwickeln und optimieren.

#### 2.3.4.1 Zuweisen der Genome in passende Spezies

Um zu entscheiden, wie sehr sich die Topologien zweier Genome unterscheiden, werden wieder die historischen Marker verwendet. Je mehr *disjoint* und *excess Genes* die Genome haben, desto unterschiedlicher sind die Topologien und dementsprechend sind die Genome weniger kompatibel.

Die Kompatibilität  $\delta$  zweier Genome wird berechnet, indem die Anzahl der *disjoint Genes*  $D$ , *excess Genes*  $E$  sowie der durchschnittlichen Gewichtsdivergenz der *matching Genes*  $\bar{W}$  addiert wird.

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

Die Koeffizienten  $c_1$ ,  $c_2$  und  $c_3$  ermöglichen das Anpassen der einzelnen Faktoren. Der Faktor  $N$  ist die Anzahl der Verbindungen in dem größeren Genom. Für kleine Genome wird dieser Wert auf 1 gesetzt.

Der Kompatibilitätswert  $\delta$  ermöglicht das Einteilen der Genome in verschiedene Spezies mit einem Kompatibilitätsschwellwert  $\delta_t$ . NEAT hat eine geordnete Liste mit allen Spezies, die in einer Population aufgetreten sind. Jede Spezies hat einen Repräsentanten  $r$ . Der Repräsentant ist ein zufällig ausgewähltes Genom der vorherigen Generation, das zu dieser Spezies gehört.

Ein Genom  $g$  der aktuellen Generation wird einer Spezies zugeteilt, indem sequenziell über alle Spezies iteriert wird, der Kompatibilitätswert  $\delta$  des Genoms  $g$  und des Repräsentanten  $r$  gebildet wird und das Ergebnis mit dem Schwellwert  $\delta_t$  verglichen wird. Ist  $\delta \leq \delta_t$  wird das Genom der Spezies zugewiesen und die Suche nach einer Spezies beendet. Ist keine passende Spezies vorhanden, wird eine neue Spezies erstellt. Das Genom  $g$  wird dieser Spezies zugewiesen und als Repräsentant gesetzt.

### 2.3.4.2 Explicit fitness sharing

Genome, die in einer Spezies sind, werden nur noch mit Genomen rekombiniert, die sich in derselben Spezies befinden. Allerdings reicht das Unterteilen der Genome in verschiedene Spezies nicht aus, um strukturelle Innovationen zu beschützen.

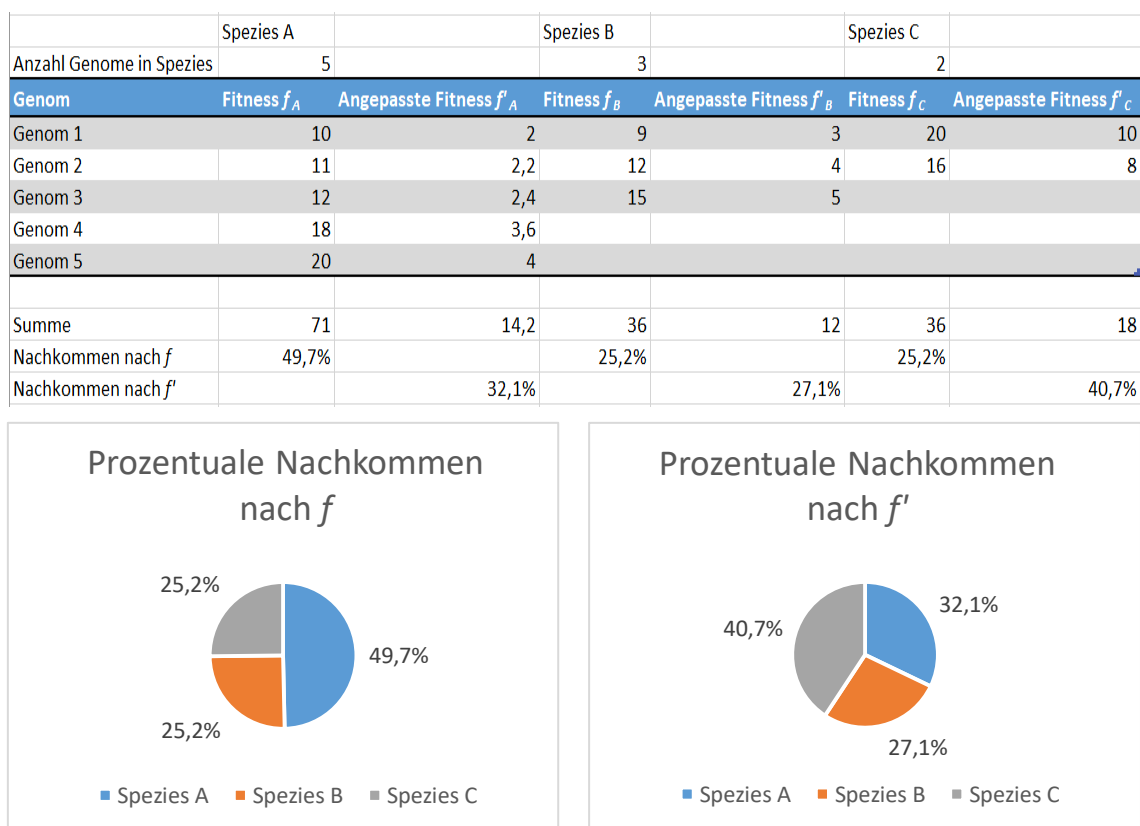


Abbildung 11 Berechnung der Nachkommen mit dem Fitnesswert  $f$  und dem angepassten Fitnesswert  $f'$

In einer neuen Spezies befinden sich nicht so viele Genome wie in älteren Spezies und die wenigen Genome, die sich zu Beginn in dieser Spezies befinden, sind noch nicht optimiert. Infolgedessen haben diese einen geringeren Fitnesswert und eine geringere Chance bei der Rekombination. Dies führt dazu, dass die Spezies aussterben wird und so die strukturellen Innovationen wieder verloren gehen.

Um das zu verhindern, verwendet NEAT *explicit fitness sharing*. Das bedeutet, dass jedes Genom neben dem erzielten Fitnesswert  $f$  noch einen angepassten Fitnesswert  $f'$  erhält, der von der restlichen Spezies beeinflusst wird. Ziel des Verfahrens ist, den Genomen einer kleinen Spezies eine größere Chance in der Reproduktion zuzuweisen.



Um den angepassten Fitnesswert  $f'$  zu erhalten, wird die erzielte Fitness  $f$  durch die Größe der Spezies geteilt, zu der das Genom gehört (Buckland, 2002). In der Reproduktionsphase bekommt jede Spezies eine gewisse Anzahl Nachkommen zugewiesen, die von der angepassten Fitness abhängt. Die Anzahl der Nachkommen einer Spezies ist proportional zur Summe der angepassten Fitness  $f'$  der Genome in dieser Spezies.

In Abbildung 11 ist die Berechnung der Nachkommen beispielhaft dargestellt. Es gibt drei Spezies [A, B, C]. In der ersten Spezies befinden sich fünf Genome, in der zweiten Spezies drei Genome und in der letzten Spezies zwei Genome. Zu jedem Genom gibt es einen Fitnesswert  $f$  und einen angepassten Fitnesswert  $f'$ . Würden die Nachkommen nur nach dem Fitnesswert  $f$  zugeteilt werden, wären fast 50% der Nachkommen von Spezies A erzeugt worden. Dies liegt vor allem daran, dass A mehr Genome als seine Konkurrenten hat und nicht daran, dass A bessere Fitnesswerte erzielt. Durch den angepassten Fitnesswert haben größere Spezies einen Nachteil. In diesem Fall erhält A nur noch  $\approx 32\%$  der Nachkommen. Kleinere erfolgreiche Spezies, wie zum Beispiel C, bekommen in diesem Fall mehr Nachkommen zugewiesen (40,7%). Die genauen Ergebnisse sind in den Diagrammen in Abbildung 11 dargestellt.

Durch den angepassten Fitnesswert  $f'$  kann es sich keine Spezies leisten, zu groß zu werden, auch wenn ein Großteil der Genome sehr hohe Fitnesswerte hat. Infolgedessen ist es unwahrscheinlich, dass eine Spezies die ganze Population übernimmt und ermöglicht so das Bilden von kleinen Nischen, in denen strukturelle Innovationen und neue Topologien gebildet werden können.

### 2.3.5 Inkrementelles Wachsen einer am Anfang minimalen Struktur

Andere neuroevolutionäre Algorithmen erzeugen die initiale Population mit zufällig erstellten Topologien, mit dem Ziel, strukturelle Vielfalt zu erzeugen. Dieses Vorgehen hat zwei große Nachteile.

Viele dieser Topologien haben Input-Neuronen, die nicht zu allen Output-Neuronen verknüpft sind. Da dies nicht zielführend ist, müssen die Netze aussortiert werden. Dafür benötigt der Algorithmus Zeit, was zu einer schlechteren Performance führt.

Der zweite Nachteil ist, dass diese Strukturen oftmals Neuronen und Verbindungen enthalten, die nicht benötigt werden. Das neuronale Netz könnte dieselben Ergebnisse mit einer kleineren Struktur erzielen. Diese können schneller optimiert werden als große Strukturen. Netze mit unnötigen Strukturen benötigen somit eine längere Optimierungszeit für dasselbe Ergebnis.

Um diese Probleme zu umgehen, haben die Genome der initialen Population in NEAT eine einheitliche minimale Struktur. Dies ist typischerweise ein Genom, das nur aus Input- und Output-Neuronen besteht. In diesem Genom sind alle Input-Neuronen mit allen Output-Neuronen verbunden. Neue Strukturen werden inkrementell durch Mutationen hinzugefügt. Nur jene können sich durchsetzen, die eine Steigerung des Fitnesswertes bewirken. So ist jede Struktur in dem Genom gerechtfertigt. Dies löst auch den

zweiten Nachteil, der bei zufälligen Strukturen entsteht. NEAT verschwendet keine Optimierungszeit für unnötige Strukturen und ist so bedeutend schneller als andere Algorithmen.

### **2.3.6 Einschränkungen des Zufalls in der Reproduktion**

Die Auswahl der Elterngenome ist nicht komplett zufällig. Die folgenden Regeln schränken die zufällige Auswahl der Genome ein.

#### **2.3.6.1 Speichern der besten Genome**

Nicht alle Genome der nächsten Generation werden durch Rekombination und Mutation erzeugt. Das beste Genom, also das mit dem höchsten Fitnesswert, wird immer unverändert in die nächste Generation kopiert. Mit dieser Maßnahme wird verhindert, dass das beste Genom bei der Rekombination verloren gehen kann. Somit kann der Fitnesswert in der nächsten Generation nicht sinken. Neben dem besten Genom der ganzen Generation wird auch das beste Genom jeder Spezies kopiert, die mehr als fünf Mitglieder hat.

#### **2.3.6.2 Aussterben der Spezies**

Schafft es eine Spezies nicht, ihren Fitnesswert in 15 Generationen zu steigern, werden ihr bei der Reproduktion keine Nachkommen zugewiesen und sie wird aussterben. Somit bekommen die Spezies, die noch optimiert werden können, mehr Nachkommen zugewiesen.

#### **2.3.6.3 Auswahl der Elterngenome**

Nachdem einer Spezies eine Anzahl an Nachkommen zugewiesen worden ist, werden diese durch Genome, die sich in dieser Spezies befinden, erzeugt. Bei der Rekombination werden zwei Elterngenome zufällig ausgewählt. Die Chance, dass ein Genom ausgewählt wird, ist proportional zu dem angepassten Fitnesswert. Aber dies gilt nur für die Genome, die basierend auf dem Fitnesswert zu den besten 50% der Spezies gehören. Die Genome, die nicht dazu gehören, werden nicht zur Reproduktion verwendet.

### 3 Softwarearchitektur

Ziel der Softwarearchitektur ist, den NEAT Algorithmus als Library für die Entwicklungsumgebung Unity zu Verfügung zu stellen. Die Zielgruppe dieses Projektes sind Spieleentwickler, die mit NEAT ein neuronales Netz trainieren möchten, um dieses später als künstliche Intelligenz einzusetzen.

Da NEAT anwendungsunabhängig eingesetzt werden kann, sollen die im vorherigen Kapitel vorgestellten Funktionen durch die Library implementiert und durch eine einfache Schnittstelle zur Verfügung gestellt werden. So soll NEAT in bestehende Projekte ohne viel Aufwand integriert werden können.

Im folgenden Kapitel werden der Aufbau und die Funktionsweise der Schnittstelle erläutert.

#### 3.1 Architektur der Schnittstelle

Die Schnittstelle für den Programmierer besteht aus vier Klassen und einem Interface wie in Abbildung 12 dargestellt. Im Folgenden werden die Funktionen der einzelnen Komponenten erklärt.

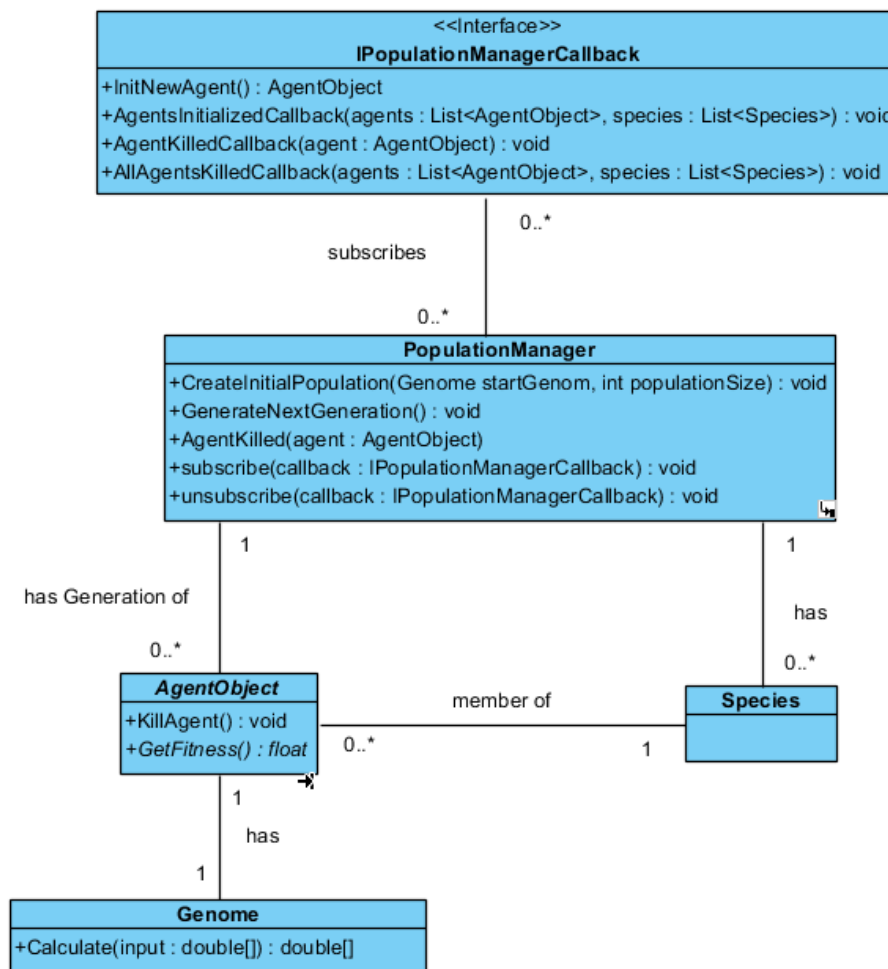


Abbildung 12 Klassendiagramm der Schnittstelle

### **3.1.1 Genome**

Die Klasse *Genome* repräsentiert ein Genom aus dem NEAT Algorithmus. In dem Genom sind der Aufbau und die Struktur des neuronalen Netzes sowie die Gewichte der Verbindungen gespeichert.

Um auf das neuronale Netz zuzugreifen, wird die Methode *Calculate()* verwendet. Diese nimmt als Parameter ein Array mit *double* Werten an und gibt diese in das gespeicherte neuronale Netz. Das neuronale Netz berechnet aus den Werten ein Ergebnis und gibt dieses ebenfalls durch ein Array mit *double* Werten zurück. Die Größe des Eingabe- beziehungsweise Ausgabe-Arrays ist abhängig von der Anzahl an Input- und Output-Neuronen.

Auf das Genom kann über ein Objekt der Klasse *AgentObject* zugegriffen werden.

### **3.1.2 AgentObject**

Die Klasse *AgentObject* ist eine abstrakte Klasse, die vom Nutzer implementiert werden muss. Sie repräsentiert einen Agenten im System, welcher von einem neuronalen Netz gesteuert werden soll. Zu diesem Zweck besitzt jeder Agent Zugriff auf eine eigene Instanz der Klasse *Genome*.

Der Agent muss das Eingabe-Array erstellen und die Ergebnisse des neuronalen Netzes interpretieren.

Zudem muss der Agent die Methode *GetFitness()* implementieren. Der Rückgabewert dieser Methode ist ein *float*. NEAT ruft diese Methode am Ende der Evaluation auf, um den erreichten Fitnesswert eines Genoms zu erhalten. Da diese Funktion anwendungsspezifisch ist, gibt es keine Standardimplementierung.

Die Methode *KillAgent()* ist vorimplementiert und muss aufgerufen werden, wenn die Evaluation des Agenten endet. Die Evaluation kann durch mehrere Bedingungen enden. Unter anderem kann ein festgelegtes Zeitlimit erreicht sein, der Agent hat die Aufgabe erfolgreich beendet oder hat frühzeitig einen Fehlerzustand erreicht.

### **3.1.3 Species**

Die Klasse *Species* repräsentiert eine Spezies in NEAT. Eine Spezies hält eine Referenz auf alle Agenten, die zu ihr gehören. Die Spezies wird der Schnittstelle zur Verfügung gestellt, da dies weitere Möglichkeiten der Auswertung bietet. Es wird ersichtlich, welche Genome zusammen gruppiert wurden, welche Spezies wann aufgetreten ist, wie viele Mitglieder diese hat und wie die durchschnittliche Leistung ist.

### 3.1.4 PopulationManager

Der *PopulationManager* bietet die Schnittstelle für den NEAT Algorithmus und implementiert dessen Logik. Die Klasse hat eine Referenz auf alle Agenten, deren Genome und auf alle Spezies.

Um mit der Schnittstelle zu interagieren, muss das Interface *IPopulationManagerCallback* implementiert werden. Dies ist eine Callback Klasse, die nach Aufrufen der Methode *subscribe()* Nachrichten über den Zustand der Population erhält. Wenn keine weiteren Nachrichten benötigt werden, können diese mit der *unsubscribe()* Methode beendet werden.

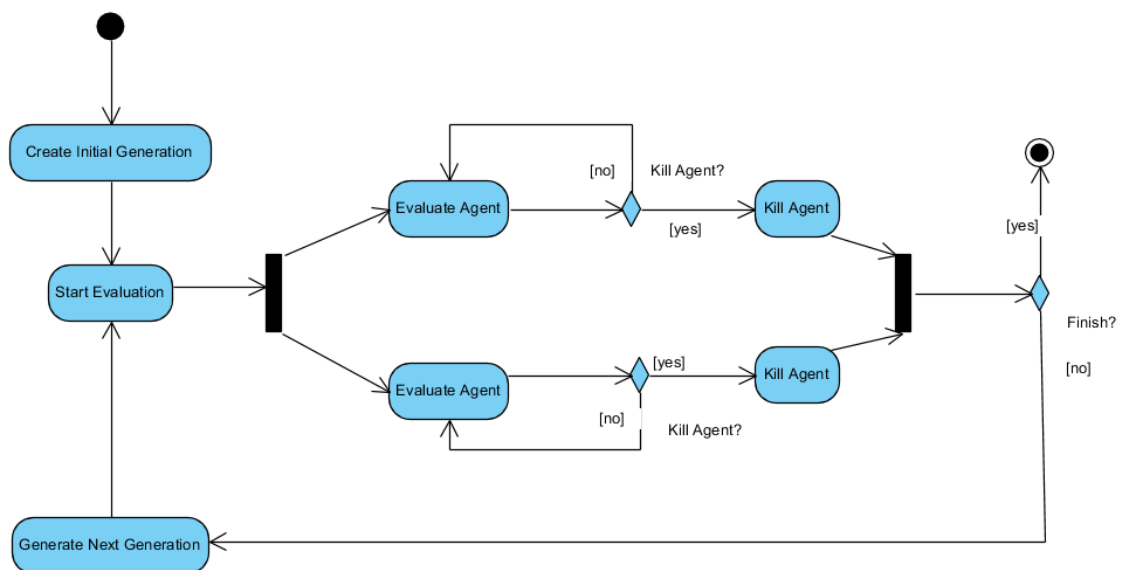


Abbildung 13 Aktivitätsdiagramm für die Schnittstelle

Die Klasse bietet zwei Methoden. Die erste heißt *CreateInitialPopulation()* und wird aufgerufen, um die erste Generation der Genome zu erzeugen. Als Parameter müssen das Startgenom sowie die Populationsgröße übergeben werden. Das Startgenom enthält die gewünschten Neuronen und Verbindungen der ersten Generation. Die Populationsgröße gibt an, wie viele Genome in jeder Generation vorhanden sein sollen.

Die zweite heißt *GenerateNextGeneration()* und kann aufgerufen werden, wenn die Evaluation der Agenten in der vorherigen Generation beendet ist. In ihr wird die nächste Generation mit den Genomen der vorherigen Generation erzeugt. Abbildung 13 stellt den Ablauf beispielhaft dar.

### 3.1.5 IPopulationManagerCallback

Über das Interface *IPopulationManagerCallback* können Nachrichten bezüglich der Population erhalten werden. Für diesen Zweck gibt es drei Methoden.

Die Methode *AgentsInitializedCallback()* wird vom *PopulationManager* aufgerufen, wenn alle Genome, die dazugehörigen Agenten und Spezies erstellt wurden. Mit dem Aufruf dieser Methode kann die Evaluation starten. Die Evaluation der Genome kann

entweder parallel oder seriell stattfinden. Dies ist abhängig von der erstellten Testumgebung und der Leistung des Systems. Dies wird in Kapitel 4.2.6 näher erläutert.

Wenn die Evaluation eines Agenten beendet ist, wird der Callback über die Methode *AgentKilledCallback()* informiert. Der Parameter für diese Methode ist der Agent, dessen Evaluation beendet wurde.

Wenn die Evaluation aller Agenten beendet ist, wird zu der vorherigen Methode zusätzlich die Methode *AllAgentsKilledCallback()* aufgerufen. Diese enthält eine Liste aller Agenten und Spezies der Generation. Diese Informationen können für Auswertungszwecke verwendet werden. Mit dem Aufruf dieser Methode ist der *PopulationManager* bereit, die nächste Generation zu erstellen.

Da die Klasse *AgentObject* abstrakt ist, kann der *PopulationManager* von dieser keine Instanzen erzeugen. Um Zugriff auf die implementierte Klasse zu erhalten, ruft der *PopulationManager* die Methode *InitNewAgent()* auf. In dieser muss ein neuer Agent erstellt und zurückgegeben werden.

## 3.2 Architektur NEAT

Das im vorherigen Kapitel vorgestellte Klassendiagramm ist eine vereinfachte Version. Diese reicht, um die Schnittstelle zu beschreiben, aber nicht, um die Funktionen von NEAT zu implementieren. Im folgenden Kapitel wird das Klassendiagramm vervollständigt und die Funktionsweise der Methoden im *PopulationManager* dargestellt.

### 3.2.1 Klassendiagramm

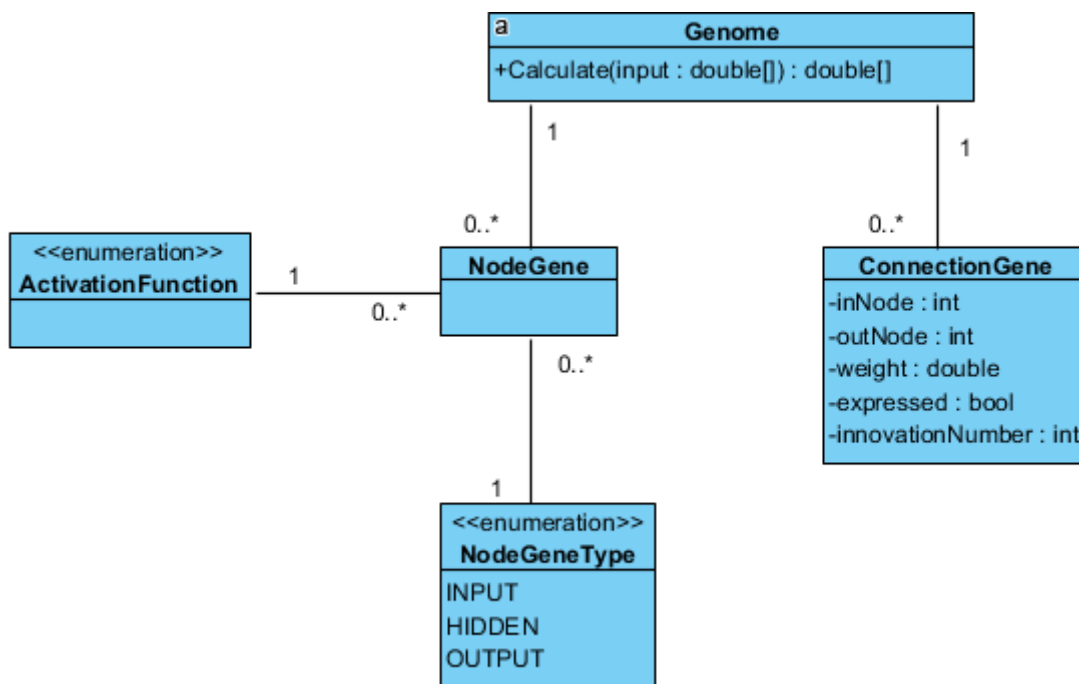


Abbildung 14 Ausschnitt aus dem Klassendiagramm mit weiteren genombezogenen Klassen

Das NEAT Genom besitzt, wie in Abbildung 14 dargestellt und in Kapitel 2.3.1 erklärt, eine Liste mit *NodeGenes* und *ConnectionGenes*. Die Klasse *ConnectionGene* enthält alle Datenfelder, die in NEAT beschrieben sind. Die Klasse *NodeGene* enthält den Typ des Neurons und eine Aktivierungsfunktion. Diese wird später im neuronalen Netz verwendet. Es ist möglich, im selben Genom verschiedene Aktivierungsfunktionen zu verwenden.

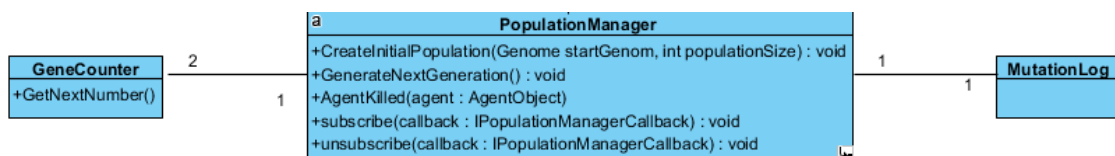


Abbildung 15 Ausschnitt aus dem Klassendiagramm mit weiteren Klassen des *PopulationManager*

Für die Klasse *PopulationManager* werden zwei neue Klassen erstellt (Abbildung 15). Der *PopulationManager* hat Referenzen auf zwei Objekte der Klasse *GeneCounter*. Der *GeneCounter* ermöglicht es, eindeutige IDs zu generieren. Eine Instanz generiert die IDs der Neuronen, die zweite Instanz generiert die Innovationsnummern der Verbindungen.

Alle auftretenden Mutationen werden in der Klasse *MutationLog* gespeichert. Sollte eine strukturelle Mutation mehrmals auftreten, können die entsprechenden IDs oder Innovationsnummern aus diesem gelesen werden.

### 3.2.2 Aktivitätsdiagramm

Der *PopulationManager* hat drei Methoden, die sich direkt auf den NEAT Algorithmus beziehen. In diesem Kapitel wird anhand drei verschiedener Aktivitätsdiagramme der genaue Ablauf dieser Methoden betrachtet.

#### 3.2.2.1 Initiale Population

Wie in Abbildung 13 gezeigt, beginnt NEAT immer mit dem Erstellen der initialen Population. Dies geschieht durch das Aufrufen der Methode *CreateInitialPopulation()*. Der Ablauf der Methode ist in Abbildung 16 dargestellt.

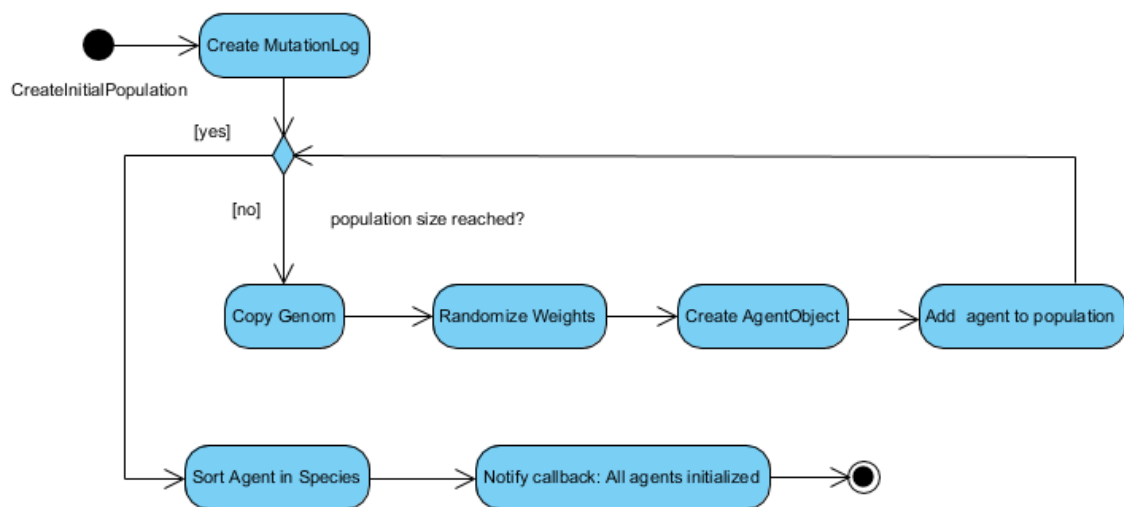


Abbildung 16 Aktivitätsdiagramm zum Erstellen der initialen Population

Zu Beginn wird eine Instanz der Klasse *MutationLog* erstellt. Diese dient dazu, die im späteren Verlauf auftretenden Mutationen zu speichern. Danach wird das Startgenom, das an die Methode übergeben wird, so oft kopiert, bis die gewünschte Populationsgröße erreicht ist. Bei dem Kopiervorgang werden die Gewichte des ursprünglichen Genoms übernommen. Dies führt dazu, dass alle Genome gleich sind und in der später stattfindenden Evaluation gleiche Fitnesswerte erhalten. Ziel ist es aber, verschiedene Genome zu haben. Zu diesem Zweck werden die Gewichte jeder Verbindung in jedem Genom zufällig neu gesetzt.

Danach wird zu jedem Genom ein entsprechender Agent erstellt. Dies passiert über die Methode *InitNewAgent()* in der Callback Klasse. Der Agent wird dann der neuen Population hinzugefügt. Als letzten Schritt werden die Agenten anhand der Genome und deren Kompatibilitätswerte in verschiedene Spezies eingeteilt. Mit diesem Schritt ist die Initialisierung der ersten Generation abgeschlossen. Der Callback wird davon über den Aufruf von *AllAgentsInitialized()* informiert und kann die Evaluation starten.



### 3.2.2.2 Ende der Evaluation eines Agenten

Immer wenn die Evaluation eines Agenten endet, wird bei diesem die Methode *KillAgent()* aufgerufen. Der Agent wird auf inaktiv gesetzt und der *PopulationManager* wird entsprechend benachrichtigt, indem die Methode *AgentKilled()* aufgerufen wird. Abbildung 17 zeigt wie der *PopulationManger* diese Nachricht verarbeitet.

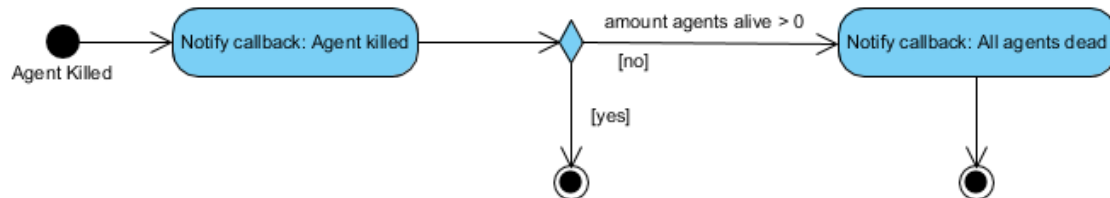


Abbildung 17 Aktivitätsdiagramm zum Beenden der Evaluation eines Agenten

Zu Beginn wird der Callback über das Ende der Evaluation des Agenten benachrichtigt. Danach wird geprüft, ob es noch mindestens einen Agenten gibt, der sich in der Evaluationsphase befindet. Ist dies der Fall, wird die Methode beendet. Haben alle Agenten die Evaluationsphase abgeschlossen, wird der Callback darüber informiert.

### 3.2.2.3 Neue Generation

Nachdem die Evaluation aller Agenten beendet wurde, kann durch den Aufruf der Methode *GenerateNextGeneration()* eine neue Generation erstellt werden.

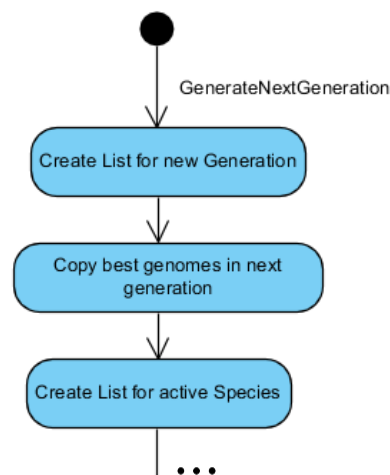


Abbildung 18 Ausschnitt 1 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation

Abbildung 18 zeigt den Beginn der Methode. Es wird eine Liste erstellt, in der alle neuen Agenten zwischengespeichert werden. Die besten Agenten jeder Spezies mit mehr als fünf Mitgliedern werden in die neue Generation unverändert kopiert. Der beste Agent der ganzen Generation wird, falls noch nicht kopiert, ebenfalls in die neue Generation kopiert.

Im nächsten Schritt wird jede Spezies auf die Rekombination vorbereitet. Dies umfasst mehrere Schritte (Abbildung 19). Zuerst wird eine temporäre Liste angelegt, in der später alle ausgewählten Spezies für die Rekombination gespeichert werden. Danach wird über alle Spezies iteriert und es werden folgende Schritte ausgeführt:

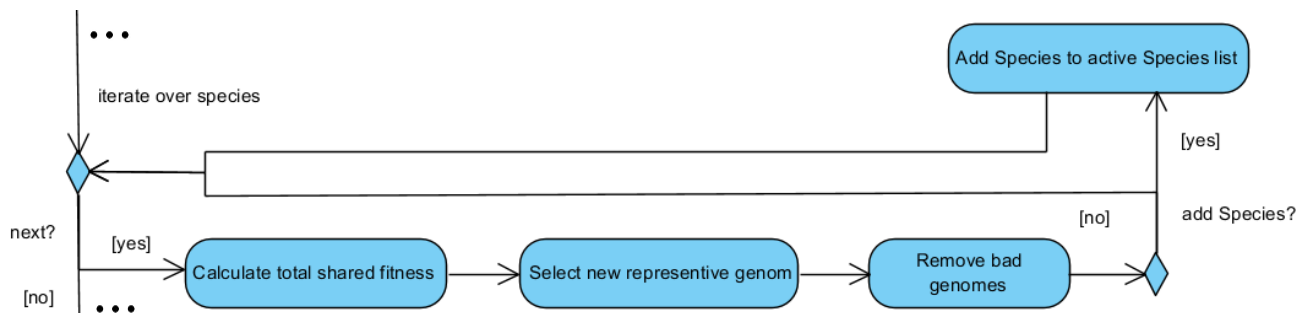


Abbildung 19 Ausschnitt 2 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation

1. Bevor eine Spezies modifiziert wird, wird der angepasste Fitnesswert aller Genome in der Spezies aufsummiert und gespeichert. Der Wert wird später bei der Zuweisung der Nachkommen benötigt.
2. Danach wird ein zufällig ausgewähltes Genom als nächster Repräsentant der Spezies gesetzt.
3. Für die Rekombination werden nur die besten 50% der Genome verwendet. Die schlechten Genome werden aus der Spezies gelöscht.
4. Im letzten Schritt wird entschieden, ob die Spezies für die Reproduktion verwendet wird. Hat es kein Genom geschafft, in 15 Generationen die maximal erreichte Fitness der Spezies zu steigern, wird die Spezies aussortiert. Ist dies nicht der Fall, wird die Spezies der temporären Liste hinzugefügt.

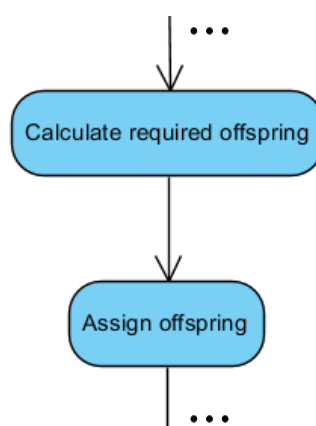


Abbildung 20 Ausschnitt 3 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation

Nachdem ermittelt ist, welche Spezies für die Rekombination verwendet werden, wird diesen eine entsprechende Anzahl an Nachkommen zugewiesen (Abbildung 20). Da be-

reits einige Genome kopiert wurden, wird zuerst die Anzahl der noch benötigten Genome berechnet. Jede für die Rekombination ausgewählte Spezies erzeugt einen Teil der Nachkommen. Die Anzahl ist proportional zur Summe des angepassten Fitnesswertes. Nach diesem Schritt beginnt die eigentliche Rekombination (Abbildung 21).

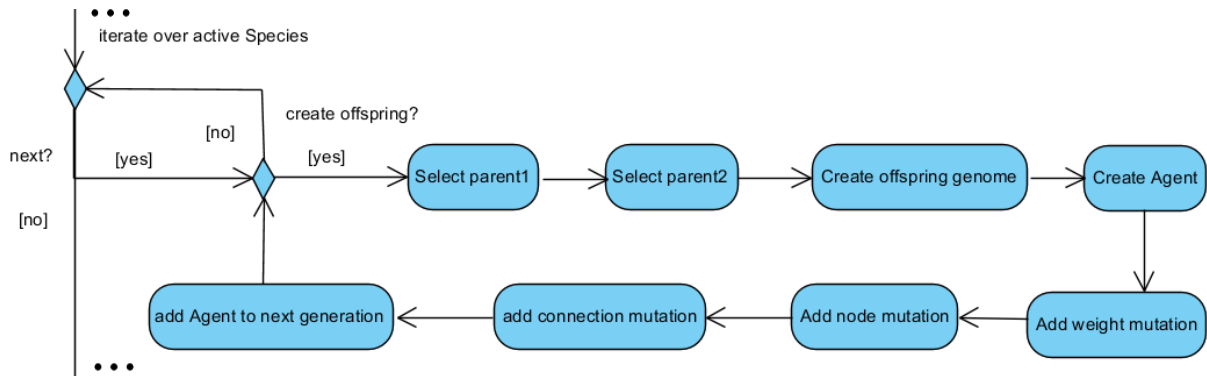


Abbildung 21 Ausschnitt 4 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation

Es wird erneut über jede Spezies iteriert. Für jede Spezies werden die entsprechenden Nachkommen erstellt. Im ersten Schritt werden zwei Elterngenome, die sich in der Spezies befinden, ausgewählt. Die Wahrscheinlichkeit, dass ein Genom ausgewählt wird, ist proportional zum erreichten Fitnesswert. Die angepasste Fitness wird nicht benötigt, da nur das relative Verhältnis der Werte relevant ist. Mit diesen Genomen wird, wie in Kapitel 2.3.3 erläutert, die Rekombination durchgeführt. Für das neu erstellte Genom wird im folgenden Schritt ein Agent erzeugt. Dieser bekommt das neu erstellte Genom zugewiesen.

Nach der Rekombination wird das Genom noch mutiert. Jede Mutation kann unterschiedliche Wahrscheinlichkeiten haben und unabhängig von den anderen Mutationen auftreten. Zuerst wird entschieden, ob die Gewichte der Verbindungen mutiert werden. Danach, ob ein neues Neuron hinzugefügt werden soll und anschließend, ob eine weitere Verbindung hinzukommt. Als Letztes wird der neu erstellte Agent der am Anfang erstellten Liste hinzugefügt. Diese Schritte werden so oft wiederholt, bis alle Nachkommen jeder Spezies erzeugt wurden.

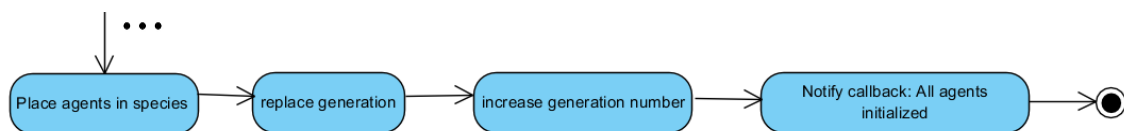


Abbildung 22 Ausschnitt 5 aus dem Aktivitätsdiagramm zum Erstellen der nächsten Generation

Die am Anfang erzeugte Liste enthält alle Agenten für die nächste Generation. Diese werden im nächsten Schritt einer passenden Spezies zugeteilt (Abbildung 22). Danach wird die alte Generation gelöscht und durch die neuen Agenten ersetzt. Zuletzt wird noch die Generationsnummer um eins erhöht und der Callback über die neuen Agenten informiert.

Dieser Ablauf kann so oft wiederholt werden, bis das gewünschte Ziel erreicht ist.

### 3.3 Architektur Neuronales Netz

Im vorherigen Kapitel wurde vorgestellt, wie Genome und deren neuronale Netze im *PopulationManager* ausgewertet und neue Nachkommen erstellt werden. Die eigentliche Evaluation findet in der Klasse *AgentObject* statt (Abbildung 23).

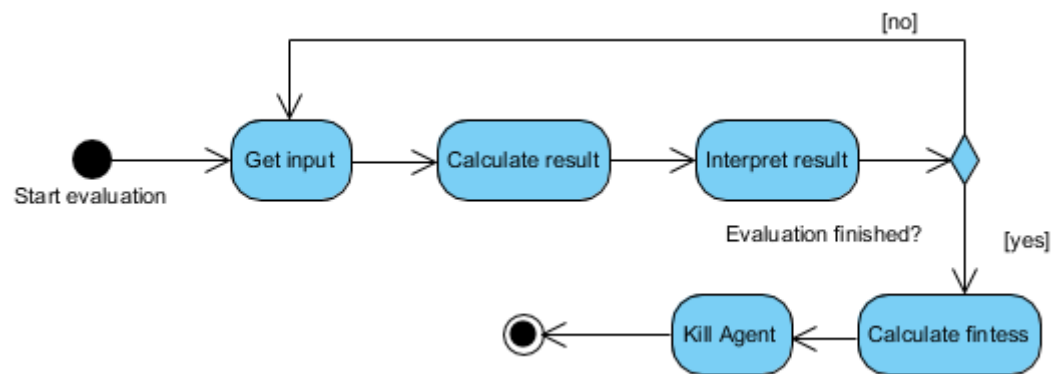


Abbildung 23 Ablauf der Evaluation eines Agenten

Nachdem die Evaluation des Agenten gestartet wurde, startet eine Schleife. Zuerst wird der Input für das neuronale Netz erstellt. Dieser ist anwendungsspezifisch und muss je nach Programm implementiert werden. Anschließend wird der Input in das neuronale Netz gegeben und ein Ergebnis berechnet. Dieses muss von dem Agenten interpretiert werden und führt zu einer Aktion im System. Ein Beispiel hierfür ist ein Charakter in einem Spiel. Das neuronale Netz besitzt nur ein Output-Neuron. Ist in diesem der Wert  $x \geq 0.5$ , geht der Charakter ein Feld nach vorne, andernfalls nicht. Dieses Beispiel kann mit mehr Output-Neuronen und mehr Aktionen erweitert werden, so dass ein komplexes Verhalten entsteht.

Nach der Aktion des Agenten wird entschieden, ob die Evaluation für diesen endet. Dafür werden in diesem Schritt alle Fehlerbedingungen abgefragt. Beispiele hierfür sind, dass der Charakter einen Gegner berührt hat, seit längerer Zeit keinen Fortschritt erzielt hat oder die maximale Zeit der Evaluation überschritten wurde. Trifft keine der Fehlerbedingungen zu, startet ein neuer Zyklus in der Schleife, andernfalls wird die Schleife abgebrochen.

Wird die Schleife abgebrochen, erfolgt die Berechnung des Fitnesswertes mit dem aktuellen Zustand. Dieser ist abhängig von der Fitnessfunktion, die im Agenten implementiert ist. Schlussendlich wird noch die Methode *KillAgent()* aufgerufen. Diese benachrichtigt den *PopulationManager*, dass die Evaluation des Agenten geendet hat.

### 3.3.1 Berechnung der Ausgabeneuronen

Die Berechnung der Werte in den Output-Neuronen ist mit einem *feed-forward* Netz-

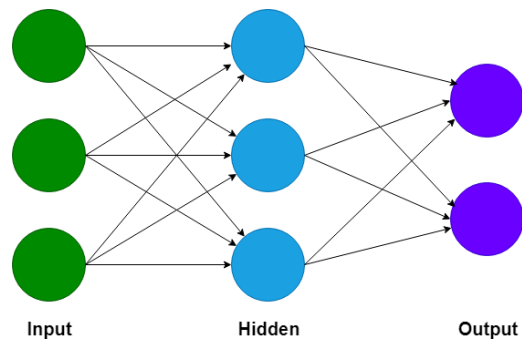


Abbildung 24 Darstellung eines neuronalen *feed-forward* Netzes mit drei verschiedenen Schichten

werk, wie in Abbildung 24 dargestellt, sehr einfach. Das Netzwerk hat drei Input-, drei Hidden- und zwei Output-Neuronen. Die Neuronen sind in feste Schichten eingeteilt und jede Schicht, mit Ausnahme der Input-Schicht, ist mit allen Neuronen der vorherigen Schicht verbunden. Um den Wert der Output-Neuronen zu berechnen, wird von links nach rechts vorgegangen. Zuerst werden die Werte der Input-Neuronen gesetzt. Dann werden alle Werte der aktuellen Schicht an die nächste Schicht weitergegeben. In dieser wird das Ergebnis aller Neuronen berechnet. Dieser Prozess wird wiederholt, bis die Output-Neuronen berechnet sind.

In NEAT werden die Netze durch strukturelle Mutation verändert. Abbildung 25 zeigt zwei neuronale Netze, die durch NEAT entstehen können.

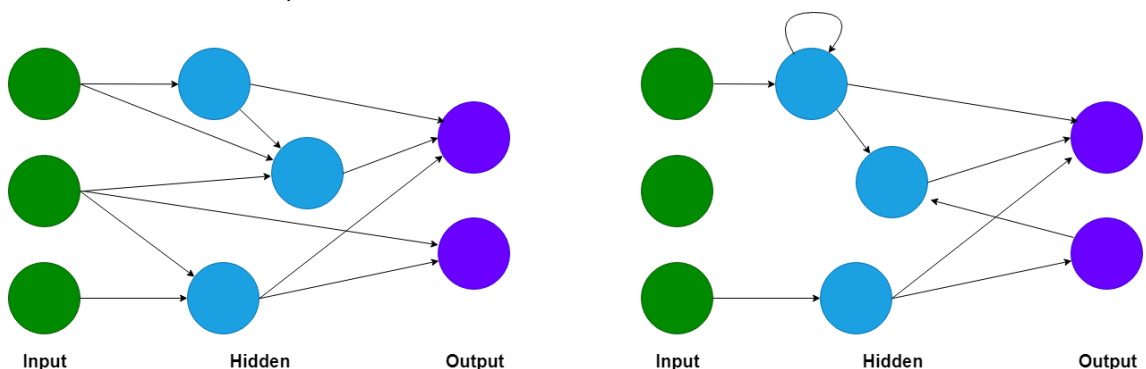


Abbildung 25 Darstellung eines neuronalen *feed-forward* Netzes ohne Schichtenarchitektur (links) und einem neuronalen *recurrent* Netz ohne Schichtenarchitektur (rechts)

Das linke neuronale Netz zeigt, dass die Neuronen nicht in Schichten eingeteilt werden können. Verbindungen können zwischen allen Neuronen entstehen, zum Beispiel kann ein Input-Neuron gleichzeitig zu Hidden-Neuronen und Output-Neuronen verbunden sein. Das rechte neuronale Netz zeigt, dass NEAT zudem Rückkopplungen in einem neuronalen Netz entwickeln kann. Durch diese Eigenschaften ist es nicht möglich, die Output-Neuronen wie in einem einfachen *feed-forward* Netz zu berechnen.

Um aufwändige Analysen der Topologie zu vermeiden, startet die Berechnung in dieser Implementierung bei den Output-Neuronen.

Zu Beginn werden die Eingabewerte in die Input-Neuronen gesetzt und diese markiert. Ein markiertes Neuron bedeutet, dass es nicht mehr berechnet werden muss. Im nächsten Schritt wird über alle Output-Neuronen iteriert und jedes wird nacheinander berechnet.

Das Output-Neuron fragt bei allen Neuronen, die es als Input verwendet, an, welchen Wert diese haben. Ist das Neuron markiert, gibt es den gespeicherten Wert zurück. Ist das Neuron nicht markiert, fragt dieses ebenfalls wie das Output-Neuron die Werte der Neuronen an, die es als Input verwendet. Dieser Aufruf wird rekursiv durchgeführt bis die Input-Neuronen erreicht werden. Hat ein Neuron seinen Wert berechnet, speichert es diesen und markiert sich. So wird sichergestellt, dass keine Neuronen doppelt berechnet werden müssen.

Dieses Verfahren kann auch für die Berechnung von Neuronen mit Rückkopplung verwendet werden. Bei der Berechnung eines Neurons wird gespeichert, welche Neuronen bereits angefragt wurden. Wird ein Neuron während der Berechnung eines Output-Neurons mehrfach angefragt, gibt es eine Rückkopplung und der letzte gespeicherte Wert wird zurückgegeben.

## 4 Evaluation

Die Funktionsweise der in Kapitel 3 vorgestellten Library wird mit drei verschiedenen Beispielprojekten demonstriert und validiert. Das erste Projekt ist das XOR-Beispiel. Das XOR-Problem ist ein klassisches Problem für neuronale Netze und wird verwendet, um die korrekte Funktionalität der Library zu bestätigen. Das zweite Projekt beschäftigt sich mit einer Simulation eines autonom fahrenden Objektes. Dieses soll eine gegebene Strecke fahren, ohne an den Rand dieser zu stoßen. Im letzten Beispiel wird eine einfache Version des NES-Spiels *Super Mario Bros.* implementiert. In diesem Projekt soll NEAT die Mario Figur durch ein nachgebildetes Level steuern.

### 4.1 XOR-Beispiel

Die Implementierung des NEAT Algorithmus soll mit einem Beispiel validiert werden. Hierfür gibt es verschiedene Tests, mit der die Effizienz verschiedener Optimierungsstrategien gemessen und mit anderen Algorithmen verglichen werden können. Eine der bekanntesten Aufgaben ist das XOR-Problem. Dieses bietet sich für NEAT besonders an, da dieser Test auch mit der originalen Implementierung durchgeführt wurde und die Ergebnisse in der Publikation enthalten sind. Ein weiterer Vorteil dieses Tests ist, dass das XOR-Problem strukturelle Anforderungen an das neuronale Netz stellt. Mit diesen kann NEATs Fähigkeit zu strukturellen Innovationen getestet werden. Das XOR-Problem bietet sich zudem an, da die Implementierung sehr einfach ist und das Lösen des Problems bei korrekter Implementierung in einer kurzen Trainingszeit möglich ist.

#### 4.1.1 Das XOR-Problem

Das XOR-Problem, auch *Exklusives-Oder-Gatter* genannt, ist ein bekanntes Klassifizierungsproblem für neuronale Netze. Das neuronale Netz soll das Ergebnis der XOR-Funktion mit zwei binären Eingaben vorhersagen. Das Ergebnis soll *wahr* bzw. *1* sein, wenn exakt ein Eingabewert *1* ist. Für den Fall, dass beide Eingabewerte *0* oder *1* sind, soll das Ergebnis *falsch* bzw. *0* sein (Burkill, 2016). Alle möglichen Kombinationen sind in Abbildung 26 dargestellt.

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Abbildung 26 Wahrheitstabelle für die XOR-Funktion

Die XOR-Funktion ist nicht linear separierbar (Abbildung 27). Das bedeutet, dass ein einfaches neuronales Netz, welches nur Input- und Output-Neuronen besitzt, dieses Problem nicht lösen kann. Das neuronale Netz benötigt mindestens ein Hidden-Neuron, um die XOR-Funktion abzubilden.

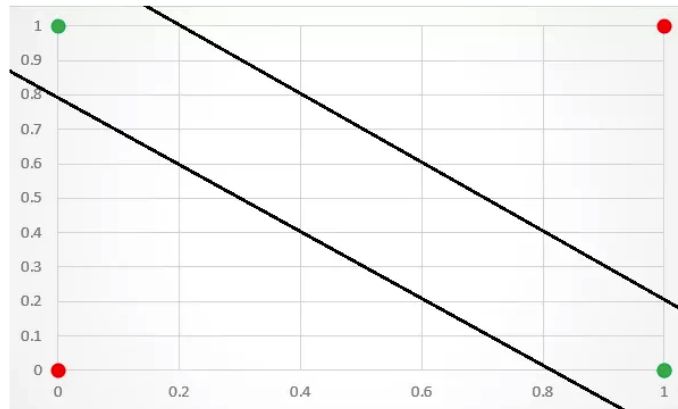


Abbildung 27 Nichtlineare Separation der XOR-Funktion (Burkill, 2016)

#### 4.1.2 Aufbau des Tests

Die vorgestellten Anforderungen an die Struktur eignen sich zum Test, ob die Library zur Entwicklung der benötigten Strukturen fähig ist. Der Algorithmus startet mit dem in Abbildung 28 dargestellten neuronalen Netz. Dieses besitzt drei Input-Neuronen und ein Output-Neuron und ist demzufolge nicht in der Lage, das XOR-Problem zu lösen. Der Algorithmus muss mindestens ein Hidden-Neuron in die Struktur einbauen, um das Problem vollständig zu lösen. In diesem Test werden die Generationen gezählt, bis NEAT das Problem gelöst hat.

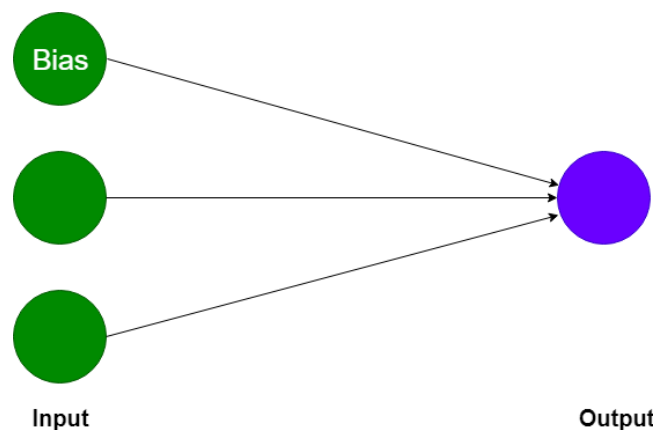


Abbildung 28 Das Startgenom für das XOR-Problem

In der Evaluationsphase muss jeder Agent vier Berechnungen durchführen, eine mit jeder vorhandenen Kombination an Eingabewerten. Die zwei Eingabewerte der XOR-Funktion werden in je ein Input-Neuron gesetzt. Das dritte Input-Neuron ist ein Bias-Neuron. Dieses hat bei jeder Berechnung den Wert 1.



Das Output-Neuron gibt das Ergebnis der Funktion an. Dieses kann wegen der verwendeten Aktivierungsfunktion zwischen 0 und 1 liegen. Das Problem gilt als gelöst, wenn ein Agent gefunden ist, der für jede der vier Berechnung das richtige Ergebnis liefert. Das Ergebnis einer Berechnung gilt als richtig, wenn der Wert des Output-Neurons für Berechnungen, die das Ergebnis 1 haben,  $x \geq 0.5$  ist. Für die Berechnungen, in denen das Ergebnis 0 ist, muss das Output-Neuron  $x < 0.5$  sein.

Die verwendete Fitnessfunktion soll umso höher sein, je näher die berechneten Ergebnisse des Agenten an den erwarteten Werten sind. Der Fitnesswert wird aufgrund der vier Berechnungen zu Beginn auf 4 initialisiert. Bei jeder Berechnung wird die Differenz des berechneten zum erwarteten Wert gebildet. Diese Differenz wird von dem Fitnesswert abgezogen. So repräsentiert ein höherer Fitnesswert eine bessere Leistung des Agenten.

Nachdem alle vier Berechnung durchgeführt wurden, wird der verbleibende Fitnesswert quadriert. So bekommen bessere Agenten proportional mehr Fitness zugeteilt. Der minimale Wert der Fitness liegt mit dieser Berechnung bei 0, der maximale Wert liegt bei 16.


### 4.1.3 Implementierung



```
1 public class XOR : AgentObject
2 {
3     public float _fitness;
4
5     public XOR(Genome genome, PopulationManager manager)
6     {
7         InitGenome(genome, manager);
8     }
9
10    public override float GetFitness()
11    {
12        return _fitness;
13    }
14 }
```

Abbildung 29 Implementierung der Klasse *XOR*

Der vorgestellte Test wird unter Verwendung der erstellten Library in einem Unity-Projekt implementiert. Im ersten Schritt wird die abstrakte Klasse *AgentObject* implementiert.

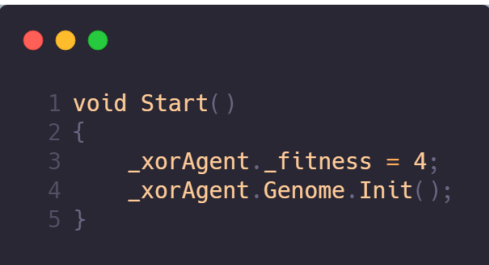


```
1 public class XOR_Agent : MonoBehaviour
2 {
3     public XOR _xorAgent;
4
5     public static double[,] _xorValues = {
6         { 0, 0, 0 },
7         { 0, 1, 1 },
8         { 1, 0, 1 },
9         { 1, 1, 0 }
10    };
11
12    //...
13 }
```

Abbildung 30 Globale Variablen der Klasse *XOR\_Agent*

Im Konstruktor wird die Methode *InitGenome()* aufgerufen. In dieser wird das *Genome* und der dazugehörige *PopulationManager* gesetzt. Die *GetFitness()* Methode gibt den Wert der Fitnessvariable zurück (Abbildung 29).

Diese Klasse wird in einem Skript mit dem Namen *XOR\_Agent* verwendet. Dieses Skript erbt von *MonoBehaviour* und kann somit an ein *GameObject* angeheftet werden. In diesem Skript wird ein statisches mehrdimensionales Array gespeichert, in dem die Werte und das dazugehörige Ergebnis der XOR-Funktion enthalten sind (Abbildung 30).



```
1 void Start()
2 {
3     _xorAgent._fitness = 4;
4     _xorAgent.Genome.Init();
5 }
```

Abbildung 31 *Start()* Methode der Klasse *XOR\_Agent*

Durch die Klasse *MonoBehaviour* müssen zwei Methoden implementiert werden. Dies sind die *Start()* und *Update()* Methode. Die *Start()* Methode (Abbildung 31) wird einmal nach der Initialisierung aufgerufen. In dieser wird die Fitness des Agenten auf 4 gesetzt und das Genom initialisiert. Danach ist das neuronale Netz bereit.

```
1 void Update()
2 {
3     if (_loopCounter <= 3)
4     {
5         double input1 = _xorValues[_loopCounter, 0];
6         double input2 = _xorValues[_loopCounter, 1];
7         double output = _xorValues[_loopCounter, 2];
8
9         //One bias node
10        double[] result = _xorAgent.Genome.Calculate(new double[] { input1, input2, 1 });
11
12        float difference = Mathf.Abs((float)(output - result[0]));
13        _xorAgent._fitness -= difference;
14
15        _loopCounter++;
16    }
17    else
18    {
19        _xorAgent._fitness = Mathf.Pow(_xorAgent._fitness, 2);
20        _xorAgent.KillAgent();
21    }
22 }
```

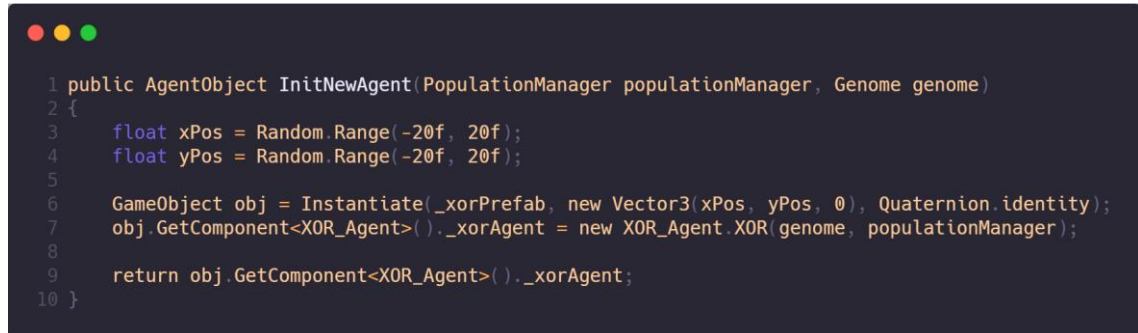
Abbildung 33 *Update()* Methode der Klasse *XOR\_Agent*

Die *Update()* Methode wird zu jedem gerenderten Bild aufgerufen. Pro Bild wird eine der vier Berechnungen durchgeführt (Abbildung 33). Zu Beginn wird ein Paar aus Input- und Output-Werten geladen. Im nächsten Schritt werden die Input-Werte in das neuronale Netz gegeben und berechnet. Das Ergebnis ist ein Array mit einem Element, dem berechneten Rückgabewert. Im nächsten Schritt wird die Differenz zwischen diesem und dem erwarteten Wert gebildet. Diese wird von dem Fitnesswert des Agenten abgezogen. Schlussendlich wird noch der Schleifenzähler erhöht. Überschreitet dieser den Wert 3, sind alle Berechnungen ausgeführt. Die verbleibende Fitness wird quadriert und die Evaluation des Agenten mit der Methode *KillAgent()* beendet.

```
1 void Start()
2 {
3     startingGenome = new Genome();
4
5     //...
6
7     _manager = new PopulationManager();
8     _manager.Callback = this;
9     _manager.CreateInitialPopulation(startingGenome, 150);
10 }
```

Abbildung 32 *Start()* Methode der Klasse *XORCallback*

Die zweite Klasse, welche die Library benötigt, ist der Callback. Dieser ist im Skript *XORCallback* implementiert und erbt von *MonoBehaviour* und dem *IPopulationManagerCallback*. In der *Start()* Methode wird das im vorherigen Kapitel vorgestellte Startgenom erstellt, sowie eine Instanz der Klasse *PopulationManager*. Bei dieser wird der Callback registriert, im Anschluss wird die Evaluation mit 150 Agenten gestartet (Abbildung 32).



```

1 public AgentObject InitNewAgent(PopulationManager populationManager, Genome genome)
2 {
3     float xPos = Random.Range(-20f, 20f);
4     float yPos = Random.Range(-20f, 20f);
5
6     GameObject obj = Instantiate(_xorPrefab, new Vector3(xPos, yPos, 0), Quaternion.identity);
7     obj.GetComponent<XOR_Agent>()._xorAgent = new XOR_Agent.XOR(genome, populationManager);
8
9     return obj.GetComponent<XOR_Agent>()._xorAgent;
10 }

```

Abbildung 34 *InitNewAgent()* Methode der Klasse *XORCallback*

Der Callback verwendet zwei Methoden vom Interface *IPopulationManagerCallback*. Die *InitNewAgent()* Methode (Abbildung 34) erzeugt eine neue Instanz der Klasse *XOR\_Agent* und setzt bei diesem das erzeugte Genom.



```

1 public void AllAgentsKilledCallback(List<AgentObject> agents, List<Species> species, int generation)
2 {
3
4     //...
5
6     if (CanAgentSolveXOR(bestAgent))
7     {
8         // Solution found
9     }
10    else
11    {
12        _manager.GenerateNextGeneration();
13    }
14 }

```

Abbildung 35 *AllAgentsKilledCallback()* Methode der Klasse *XORCallback*

Ist die Evaluation aller Agenten beendet, wird die Methode *AllAgentsKilledCallback()* aufgerufen. In dieser werden alle alten Instanzen des *XOR\_Agenten* gelöscht und das Ergebnis der Generation ausgewertet. Um zu überprüfen, ob das XOR-Problem gelöst wurde, wird der beste Agent ausgewählt und muss erneut die Berechnungen ausführen. Sind die Ergebnisse korrekt, ist das XOR-Problem erfolgreich gelöst. Das Genom kann unter anderem ausgegeben und gespeichert werden (Abbildung 35).

#### 4.1.4 Parametrisierung

In diesem Kapitel werden die Werte verschiedener Parameter vorgestellt. Diese sind, sofern nicht gekennzeichnet, direkt aus der originalen Publikation entnommen. So kann ein objektiver Vergleich stattfinden.

Jede Generation besteht aus 150 Agenten. Die Koeffizienten  $c_1$ ,  $c_2$  und  $c_3$ , die verwendet werden, um die Kompatibilität zweier Genome zu messen, werden auf  $c_1 = 1,0$ ,  $c_2 = 1,0$  und  $c_3 = 0,4$  gesetzt. Der dazugehörige Schwellwert beträgt  $\delta_t = 3,0$ .

Jedes Genom hat eine Chance von 80%, dass die Gewichte aller Verbindungen mutiert werden. In diesem Fall gibt es eine Chance von 10%, dass das Gewicht neu gewürfelt wird. Andernfalls wird das Gewicht leicht nach oben oder unten korrigiert. Hierfür wird eine Normalverteilung verwendet. Diese kann über die Standardabweichung parametrisiert werden. In diesem Projekt wird die Standardabweichung auf  $\sigma = 1,3$  gesetzt. Dieser Wert wurde experimentell ermittelt.

Die Chance, dass eine neue Verbindung hinzugefügt wird, liegt bei 5% und die Chance, dass ein neues Neuron hinzukommt, bei 3%.

Die verwendete Aktivierungsfunktion ist in allen Neuronen eine angepasste Sigmoid-Funktion,  $\varphi(x) = \frac{1}{1+e^{-4.9x}}$  (Stanley & Miikkulainen, 2002).

#### 4.1.5 Ergebnis

Das XOR-Problem wurde wie in der Publikation 100-mal durchgeführt und die Ergebnisse dokumentiert. Diese sind in Abbildung 36 dargestellt.

Durchschnittlich braucht der Algorithmus 34 Generationen, um das XOR-Problem zu lösen. Der beste Durchlauf benötigte 11 Generationen, der schlechteste Durchlauf benötigte 95 Generationen. Eine Lösung hat im Schnitt 2,5 Hidden-Neuronen. Die niedrigste Anzahl ist ein Hidden-Neuron. Dies entspricht der idealen Lösung, die in Abbildung 37 dargestellt ist.

Die in der Publikation aufgeführten Ergebnisse sind diesen sehr ähnlich. Die originale Implementierung benötigte durchschnittlich 32 Generationen. Die schlechteste Performance benötigte 90 Generationen und die Lösungen hatten durchschnittlich 2,35 Hidden-Neuronen (Stanley & Miikkulainen, 2002).

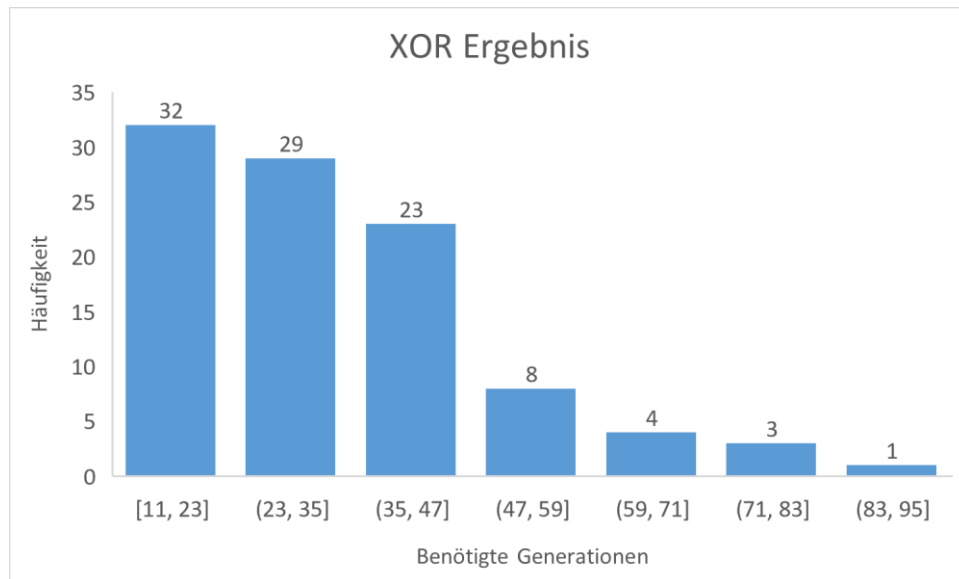


Abbildung 36 Benötigte Generationen zur Lösung des XOR-Problems

Die Library liefert nicht exakt die gleichen Ergebnisse, die in der Publikation aufgeführt sind. Dies liegt daran, dass NEAT nicht deterministisch ist. Die Ergebnisse können bei verschiedenen Durchläufen schwanken. Da die Unterschiede nur minimal sind, können diese vernachlässigt werden.

Durch diesen Test hat die Library bewiesen, dass sie in der Lage ist, die benötigten Strukturen zu entwickeln und zu optimieren. Daher kann die Library auch für andere Probleme eingesetzt werden.

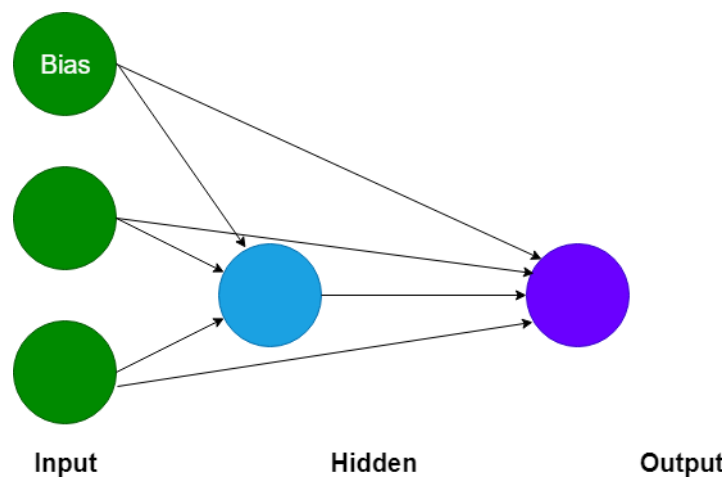


Abbildung 37 Minimal benötigte Struktur zur Lösung des XOR-Problems

## 4.2 Autonomes Fahren

Nachdem NEAT das XOR-Problem erfolgreich gelöst hat, soll der Algorithmus in einem zweiten Schritt ein neuronales Netz trainieren, welches in der Lage ist, ein *GameObject* zu steuern. Ziel ist, dieses auf einer Strecke so schnell wie möglich in das gegebene Ziel zu navigieren.

### 4.2.1 Aufbau des Tests

Im ersten Schritt wird eine *GameObject* erstellt, welches das Fahrzeug repräsentiert. Da der Fokus in dieser Ausarbeitung auf dem NEAT Algorithmus liegt, wird dieses durch eine einfache Kapsel, wie in Abbildung 38 dargestellt, repräsentiert. Die Kapsel

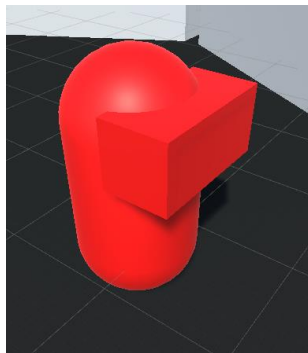


Abbildung 38 Kapsel, die einen Agenten repräsentiert

besitzt einen Quader, der die Ausrichtung anzeigt. Ziel ist, diese Kapsel auf der in Abbildung 39 dargestellten Strecke mit einem neuronalen Netz zu navigieren. Die Strecke wird durch Wände in Form von Hexagonen eingegrenzt. Diese Form bietet sich besonders an, da die Strecke so einfach abgeändert werden kann.

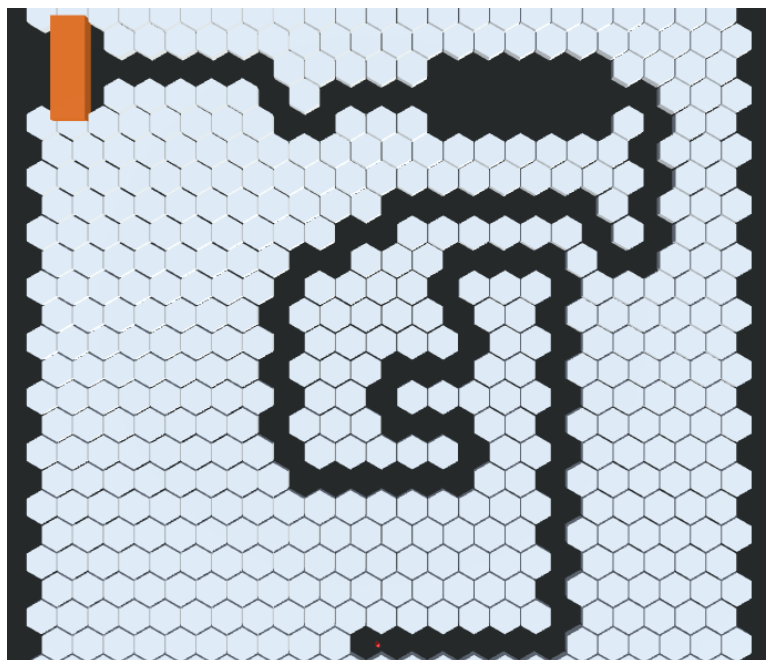


Abbildung 39 Die verwendete Strecke im Projekt Autonomes Fahren

Die Kapsel startet unten mittig und muss das in der oberen linken Ecke befindliche orange gefärbte Ziel in einer möglichst kurzen Zeit erreichen. Die Kapsel kann sich nur nach vorne bewegen, sowie sich nach links oder rechts drehen. Sobald die Kapsel durch ihre Bewegung den weißen Rand berührt, ist der Versuch gescheitert.

Um dieses Ziel zu erreichen, ist die Kapsel mit fünf Sensoren ausgestattet. Jeder Sensor misst den Abstand zur Wand. Um ein möglichst großes Sichtfeld zu erhalten, sind die Sensoren verschieden ausgerichtet, und zwar in  $-60^\circ$ ,  $-30^\circ$ ,  $0^\circ$ ,  $30^\circ$  und  $60^\circ$  der Fahrtrichtung. Abbildung 40 zeigt die Ausrichtung der Sensoren.

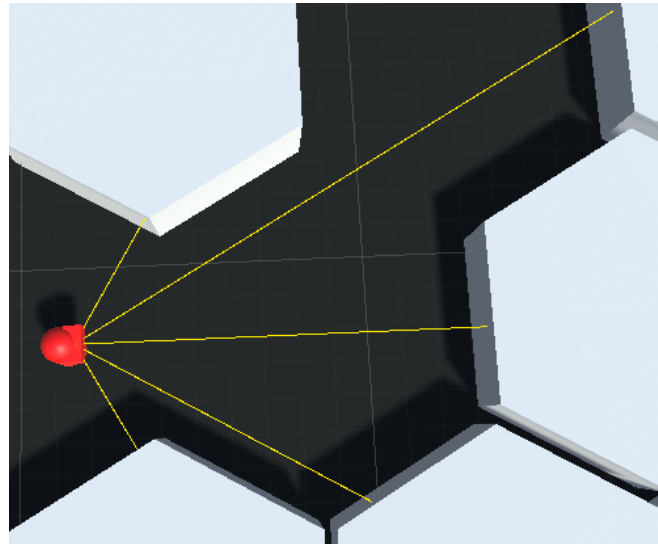


Abbildung 40 Ausrichtung der Sensoren eines Agenten

Jede Kapsel wird durch einen Agenten und dessen neuronales Netz gesteuert. Dieses startet mit sechs Input-Neuronen und zwei Output-Neuronen. Der Input setzt sich aus den Sensorwerten und einem Bias-Neuron zusammen. Der Output besteht aus zwei Werten. Der erste gibt an, wie weit die Kapsel nach vorne gehen soll, der zweite, wie weit sich die Kapsel nach links oder rechts drehen soll. Alle Input-Neuronen sind zu Beginn mit allen Output-Neuronen verbunden.

Die Fitness des Agenten setzt sich aus zwei Komponenten zusammen.

Die erste Komponente betrifft die Länge der Strecke, die der Agent zurückgelegt hat. Zu diesem Zweck wird gezählt, wie oft der Agent ein neues leeres Feld betritt. Für jedes neue Feld bekommt er einen Punkt. Betritt er dasselbe Feld mehrmals, wird es nur das erste Mal gezählt. So wird verhindert, dass diejenigen Agenten einen höheren Fitnesswert erhalten, die auf der Strecke drehen und dieselben Felder mehrmals abfahren. Die Anzahl der erreichten Felder wird quadriert, um den besseren Agenten proportional mehr Fitness zuzuteilen.

Die zweite Komponente der Fitness ist ein Zeitbonus, der gegeben wird, wenn der Agent das Ziel erreicht. Für das Level gibt es eine maximale Zeit, in der die Evaluation abgeschlossen werden muss. Wenn der Agent das Ziel erreicht, wird die verbleibende Zeit quadriert und auf seinen Fitnesswert addiert. Der Zeitbonus wird nur gegeben,



wenn der Agent das Ziel erreicht. Dies soll verhindern, dass Agenten, die zu Beginn der Evaluation an die Wand fahren, aufgrund der hohen verbleibenden Zeit einen entsprechend hohen Zeitbonus und damit einen hohen Fitnesswert erhalten, obwohl dieses Verhalten unerwünscht ist. So wird verhindert, dass Agenten falsches Verhalten erlernen.

### 4.2.2 Implementierung

Die Implementierung des *IPopulationManagerCallback* ist in diesem Projekt nahezu identisch wie in dem XOR-Beispiel aufgebaut. Aus diesem Grund erübrigt sich eine nähere Betrachtung. In diesem Kapitel wird die Implementierung des Agenten erläutert.



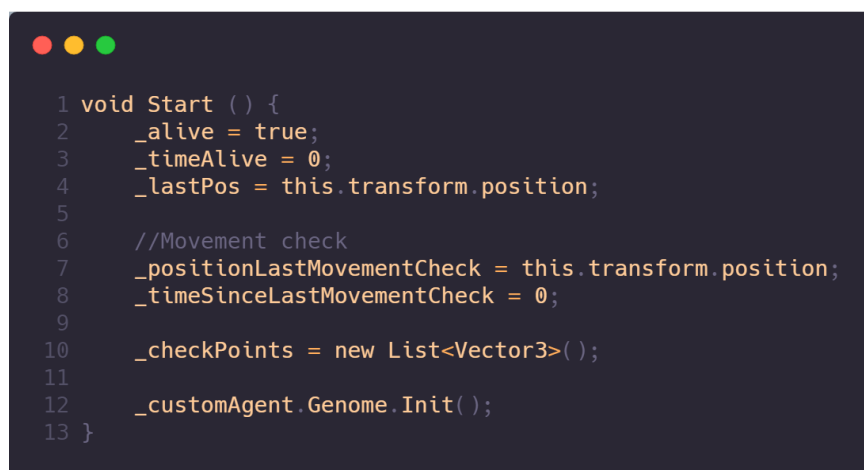
```

1 public class CustomAgent : AgentObject
2 {
3     public float _fitness;
4
5     public CustomAgent(Genome genome, PopulationManager populationManager)
6     {
7         InitGenome(genome, populationManager);
8     }
9
10    public override float GetFitness()
11    {
12        return _fitness;
13    }
14 }

```

Abbildung 41 Implementierung des Agenten

Zuerst wird, wie schon im XOR-Beispiel, eine Klasse erstellt, die vom *AgentObject* erbt. Die Implementierung dieser entspricht ebenfalls dem XOR-Beispiel (Abbildung 41).



```

1 void Start () {
2     _alive = true;
3     _timeAlive = 0;
4     _lastPos = this.transform.position;
5
6     //Movement check
7     _positionLastMovementCheck = this.transform.position;
8     _timeSinceLastMovementCheck = 0;
9
10    _checkPoints = new List<Vector3>();
11
12    _customAgent.Genome.Init();
13 }

```

Abbildung 42 *Start()* Methode der Kapsel

Diese Klasse wird im *Player* Skript verwendet. Dieses erbt von *MonoBehaviour* und ist für die Steuerung einer Kapsel zuständig. Zu Beginn wird die *Start()* Methode aufgerufen. Diese ist in Abbildung 42 dargestellt. Zunächst wird ein *flag* gesetzt, dass die Kapsel aktiv ist. Dies ist wichtig, da die Kapsel ansonsten keine Bewegungen ausführen kann. Im nächsten Schritt werden weitere Anfangswerte initialisiert. Unter anderem

wird die Zeit, die der Agent aktiv ist, auf 0 gesetzt, die letzte Position mit der Startposition initialisiert und zwei Variablen zur Überprüfung der Bewegung des Agenten erstellt. Auf letztere wird in der *Update()* Methode genauer eingegangen. Zudem wird eine Liste mit den Checkpoints erstellt. In ihr sind die Positionen der Felder gespeichert, die der Agent betreten hat. Zuletzt wird noch das Genom initialisiert, sodass das neuronale Netz verwendet werden kann.

Der Agent wird in der Methode *Update()* bewegt, die regelmäßig aufgerufen wird. Diese überprüft, ob die Evaluation vorzeitig abgebrochen werden soll und steuert die Bewegungen der Kapsel.



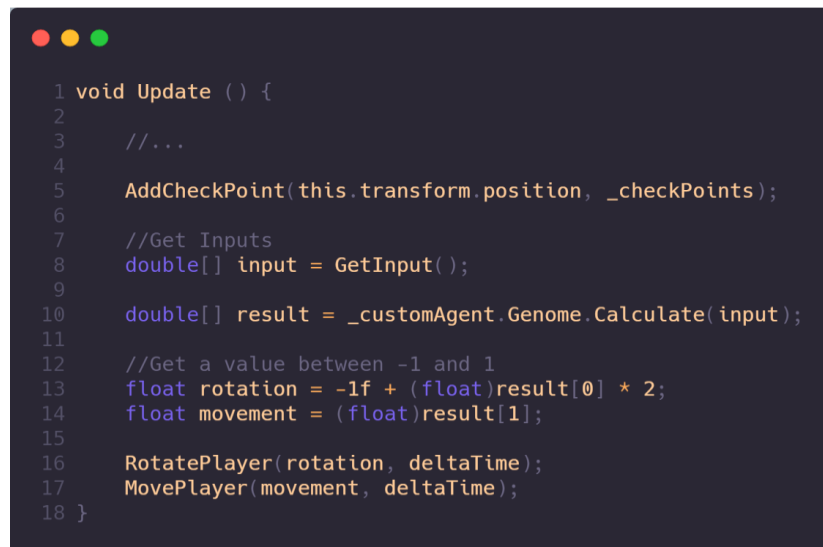
```
1
2 void Update () {
3
4     if (!_alive) return;
5
6     float deltaTime = _useFixTimeIntervalls ? 0.02f : Time.deltaTime;
7
8     //Update values
9     _timeAlive += deltaTime;
10    _lastPos = this.transform.position;
11
12    //Check if evaluation time is expired
13    if (_timeAlive >= _maxTimeForLevel)
14    {
15        KillPlayer(false);
16        return;
17    }
18
19    //Check if character has moved
20    if(CheckMovement(deltaTime))
21    {
22        KillPlayer(false);
23        return;
24    }
25
26    //...
27 }
```

Abbildung 43 Ausschnitt 1 aus der *Update()* Methode der Kapsel

Abbildung 43 zeigt den Beginn der *Update()* Methode. Ist der Agent nicht aktiv, wird die Methode sofort abgebrochen. Ist dies nicht der Fall, wird entschieden, welches Zeitintervall verwendet werden soll. Hierfür gibt es zwei Auswahlmöglichkeiten. Die eine ist ein festes Zeitintervall, die andere ist abhängig von der Framerate. Auf die Auswirkung wird in Kapitel 4.2.7 genauer eingegangen. Danach wird die Zeit, die der Agent aktiv ist, sowie die letzte Position aktualisiert. Nach Aktualisierung der Werte wird überprüft, ob der Agent die maximale Evaluationszeit für diese Aufgabe überschritten hat. Ist dies der Fall, wird die Evaluation beendet und die Ausführung der restlichen Methode abgebrochen.

Dies ist nicht einzige Möglichkeit, wie die Evaluation des Agenten beendet werden kann. In der *Update()* Methode gibt es eine weitere Überprüfung. In dieser wird getestet, ob der Agent sich in den letzten Sekunden bewegt hat. Ist dies nicht der Fall, wird

die Evaluation des Agenten ebenfalls abgebrochen. Dies erhöht die Effizienz der Trainingszeit. Sollte ein Agent sich nicht mehr bewegen, würde er bis zum Ende der Evaluation an derselben Stelle stehen. Sollten alle anderen Agenten die Evaluation bereits abgeschlossen haben, würde NEAT auf diesen Agenten warten, obwohl es keine weitere Veränderung des Fitnesswertes geben wird. Je länger die Evaluationszeit ist, desto länger müsste der Algorithmus warten. Durch diese zusätzliche Überprüfung wird dies verhindert.



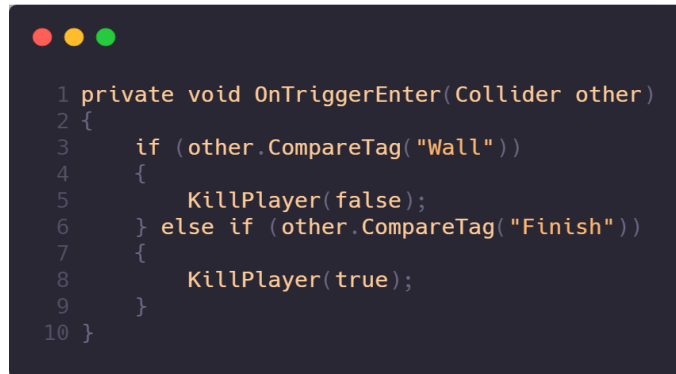
```
1 void Update () {
2
3     //...
4
5     AddCheckPoint(this.transform.position, _checkPoints);
6
7     //Get Inputs
8     double[] input = GetInput();
9
10    double[] result = _customAgent.Genome.Calculate(input);
11
12    //Get a value between -1 and 1
13    float rotation = -1f + (float)result[0] * 2;
14    float movement = (float)result[1];
15
16    RotatePlayer(rotation, deltaTime);
17    MovePlayer(movement, deltaTime);
18 }
```

Abbildung 44 Ausschnitt 2 aus der *Update()* Methode der Kapsel

Falls noch nicht geschehen, wird jetzt das aktuelle Feld, auf dem sich der Agent befindet, der Liste mit den Checkpoints hinzugefügt (Abbildung 44). Im nächsten Schritt wird der Input für die nächste Bewegung mit dem Aufruf der Methode *GetInput()* vorbereitet. In dieser wird ein Array mit den Sensorwerten erstellt. Um den Abstand zu einer Wand zu messen, wird ein *Raycast* verwendet. Es werden fünf *Raycasts* in verschiedene Richtungen ausgestrahlt. Die Ergebnisse der Messungen werden dem erstellten Array hinzugefügt. Der Raycast kann nur Wandobjekte erkennen. Dies ist wichtig, da später mehrere Kapseln gleichzeitig aktiv sind und diese sich nicht gegenseitig behindern sollen. Der *Raycast* hat eine begrenzte Reichweite. Sollte er auf keine Wand treffen, wird die maximale Reichweite als Ergebnis eingetragen. Dem Array wird im letzten Schritt noch die Eingabe für das Bias-Neuron hinzugefügt. Insgesamt hat das Array sechs Elemente, die der Anzahl an Input-Neuronen entsprechen.

Das Input-Array wird an das neuronale Netz gegeben und das Ergebnis berechnet und gespeichert (Abbildung 44). Dieses enthält die Werte für die Vorwärtsbewegung und der Rotation. Die Kapsel wird mit der Methode *RotatePlayer()* gedreht. Diese benötigt neben dem Zeitintervall noch einen Rotationswert. Dieser muss zwischen  $-1$  und  $1$  sein. Je negativer der Parameter ist, desto mehr dreht sich die Kapsel nach links. Je positiver er ist, desto weiter dreht sich die Kapsel nach rechts. Ist der Parameter  $0$ , dreht sich die Kapsel nicht. Da das Ergebnis im Output-Neuron wegen der verwendeten Aktivierungsfunktion nur ein Wert zwischen  $0$  und  $1$  sein kann, wird das Ergebnis vor der

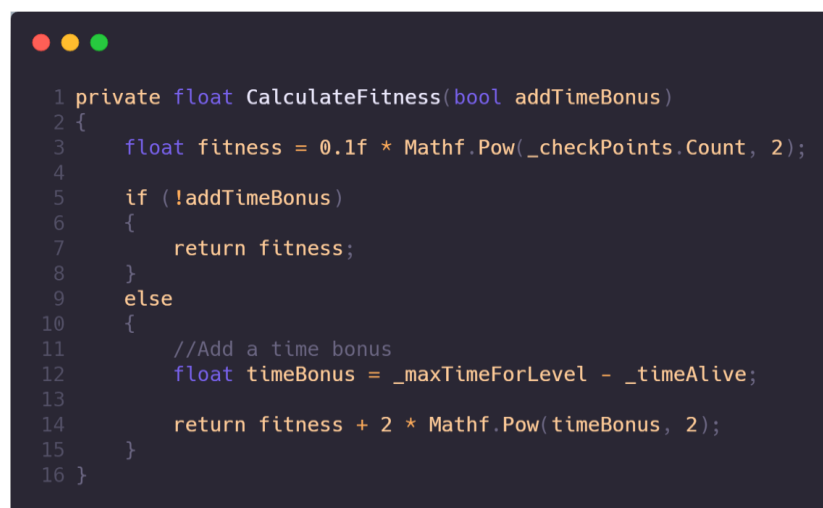
Verwendung auf die gewünschte Reichweite angepasst. Die Methode *MovePlayer()* bewegt die Kapsel nach vorne. Diese benötigt neben dem Zeitintervall einen Parameter für die Vorwärtsbewegung. Dieser muss zwischen 0 und 1 sein. Der Wert des zweiten Output-Neurons kann somit direkt verwendet werden. Je größer der Wert des Parameters, desto weiter bewegt sich die Kapsel nach vorne.

A screenshot of a code editor showing the `OnTriggerEnter` method in C#. The code is as follows:

```
1 private void OnTriggerEnter(Collider other)
2 {
3     if (other.CompareTag("Wall"))
4     {
5         KillPlayer(false);
6     } else if (other.CompareTag("Finish"))
7     {
8         KillPlayer(true);
9     }
10 }
```

Abbildung 45 *OnTriggerEnter()* Methode der Kapsel

Kollidiert die Kapsel mit einem Gegenstand, wird die Methode *OnTriggerEnter()* aufgerufen (Abbildung 45). Bei der Kollision wird das *Tag* des anderen Objektes angefragt und verglichen. Ist das Objekt eine Wand, wird die Evaluation des Agenten beendet. Ist das Objekt das Ziel, wird die Evaluation ebenfalls beendet und der Agent bekommt den Zeitbonus zugeschrieben.

A screenshot of a code editor showing the `CalculateFitness` method in C#. The code is as follows:

```
1 private float CalculateFitness(bool addTimeBonus)
2 {
3     float fitness = 0.1f * Mathf.Pow(_checkPoints.Count, 2);
4
5     if (!addTimeBonus)
6     {
7         return fitness;
8     }
9     else
10    {
11        //Add a time bonus
12        float timeBonus = _maxTimeForLevel - _timeAlive;
13
14        return fitness + 2 * Mathf.Pow(timeBonus, 2);
15    }
16 }
```

Abbildung 46 Berechnung des Fitnesswertes einer Kapsel

Die Berechnung des Fitnesswertes ist in Abbildung 46 dargestellt. Diese hat einen Parameter, der angibt, ob der Zeitbonus gegeben wird. Der Fitnesswert, der für die zurückgelegte Strecke gegeben wird, wird berechnet, indem die besuchten Felder gezählt und quadriert werden. Das Ergebnis wird mit 0,1 multipliziert. Grund hierfür ist, ein besseres Verhältnis zwischen den Komponenten des Fitnesswertes zu erhalten. Erhält der Agent keinen Zeitbonus, wird der Fitnesswert der Strecke zurückgegeben, andernfalls wird die verbleibende Zeit für die Evaluation berechnet. Das Ergebnis wird eben-

falls quadriert, um den schnelleren Agenten proportional mehr Fitness zuzuteilen. Danach wird es für ein besseres Verhältnis mit dem Wert 2 multipliziert. Das Ergebnis wird auf die bisher erreichte Fitness addiert und zurückgegeben.

### 4.2.3 Parametrisierung

Die Parameter für dieses Projekt werden, wenn nicht anderweitig beschrieben, von dem XOR-Beispiel übernommen. Allerdings wird für dieses Projekt die Populationsgröße auf 300 Agenten pro Generation erhöht. Durch mehr Agenten besteht eine größere Chance, dass nützliche Mutationen auftreten.

Zudem wurde der Faktor  $c_3$ , der in der Kompatibilitätsfunktion verwendet wird und die durchschnittliche Gewichtsdivergenz zweier Genome gewichtet, von 0,4 auf 0,8 erhöht. Die Genome haben zu Beginn bereits zwölf Verbindungen. Ziel der Änderung ist, die Genome stärker anhand unterschiedlicher Gewichte in verschiedene Spezies zuzuteilen.

### 4.2.4 Steuerung

Mit diesem Spiel kann interagiert werden. Die verfügbaren Steuerelemente werden in diesem Kapitel vorgestellt.

Der Nutzer kann die Steuerung eines Agenten selbst übernehmen, um ein Gefühl für den Schwierigkeitsgrad des Levels zu bekommen. Zu diesem Zweck ist zu Beginn ein Agent im Level platziert. Der Agent kann mit den Tasten *W*, *A*, *D* gesteuert werden. Die Taste *W* bewegt den Agenten nach vorne, die Tasten *A* und *D* drehen den Agenten nach links bzw. nach rechts.

Das Training der Agenten kann durch Drücken der *Leertaste* gestartet werden. Ist das Training gestartet, kann es durch Drücken der *Leertaste* beendet werden. Die Evaluation einer Generation kann durch Drücken der Taste *K* beendet werden. In diesem Fall wird die Evaluation aller aktiven Agenten beendet und die nächste Generation erstellt. Diese Funktion kann eingesetzt werden, wenn das Handeln der Agenten nicht effektiv ist und die Evaluation vorzeitig beendet werden soll.

Das Training kann auf verschiedenen Strecken durchgeführt werden. Zu diesem Zweck ist ein Leveleditor verfügbar. Mit der Taste *P* kann ein leeres Level geladen werden. Die gewünschte Strecke kann danach mit der *rechten Maustaste* gezeichnet werden. Das Standardlevel kann mit der Taste *R* geladen werden.

### 4.2.5 Versionen

Die vorgestellte Version des Spiels ist die finale Fassung. In diesem Kapitel werden die vorherigen Versionen beschrieben und wie diese zur finalen Version optimiert wurden.

Version 1 war die erste Version, in der die Agenten mit NEAT trainiert wurden. Da nicht bekannt war, wie gut NEAT auf das Problem reagiert, wurde mit einer sehr leichten Version des Problems angefangen. Die Strecke ist immer mindestens zwei Felder breit. So bietet sie bedeutend mehr Toleranz gegenüber Fehlentscheidungen und kann keine

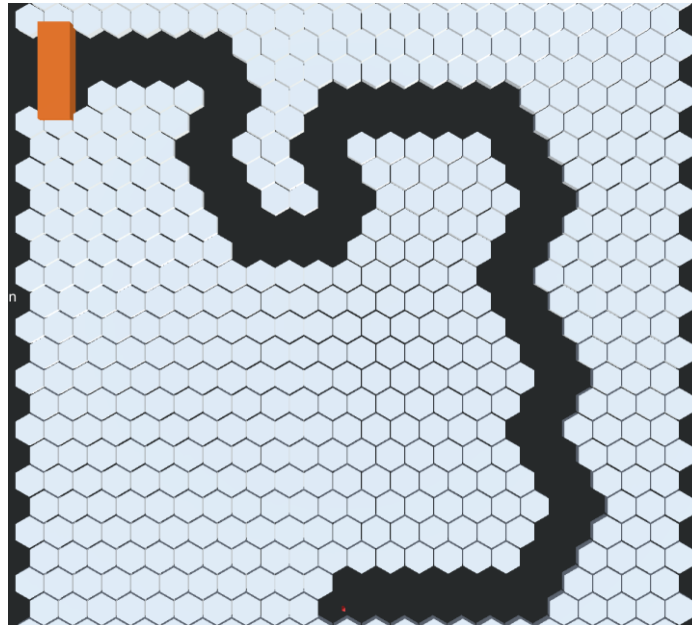


Abbildung 47 Einfache Version der verwendeten Strecke

scharfen Kurven enthalten. Die Schwierigkeit des Levels ist somit sehr gering. Abbildung 47 zeigt die verwendete Version des Levels. Neben dem einfachen Schwierigkeitsgrad des Levels besitzt der Agent weniger Steuermöglichkeiten. Das neuronale Netz besitzt nur ein Output-Neuron, welches zur Steuerung der Rotation verwendet wird. Der Agent wird mit jedem Aufruf der *Update()* Methode ein festes Stück vorwärts bewegt. Er hat in dieser Version keinen Einfluss auf die Vorwärtsbewegung und muss durch die Rotation verhindern, dass die Kapsel die Wand berührt. Es zeigte sich, dass dieses Problem zu einfach ist. Einige Agenten hatten bereits in der ersten Generation das Ziel erreicht.

In Version 2 wurde der Schwierigkeitsgrad der Strecke angehoben. Die erstellte Strecke wird auch in der finalen Version verwendet. Die Strecke besitzt scharfe Kurven sowie breitere und schmalere Abschnitte. Auch in dieser Version schafften es Agenten der ersten Generation, das Ziel zu erreichen.

Um die Agenten mehr zu fordern, müssen diese in Version 3 selbst entscheiden, ob und wie weit sie sich vorwärtsbewegen. Zu diesem Zweck bekommt das neuronale Netz das zweite Output-Neuron, mit dem die Vorwärtsbewegung gesteuert wird. Auch in dieser Version schafften es Agenten der ersten Generation, das Ziel zu erreichen. Bei genauerer Betrachtung stellte sich heraus, dass die Vorwärtsbewegung in der aktuellen Implementierung keinen großen Einfluss auf den Schwierigkeitsgrad hat, da die Optimierung des Output-Neurons sehr einfach ist. Die Mutationen laufen darauf hinaus, dass das neue Output-Neuron immer den Wert 1 hat. In diesem Fall kann der Agent

sich mit der maximalen Geschwindigkeit vorwärtsbewegen und erzielt den höchst möglichen Zeitbonus. Ist dies der Fall, liegen die gleichen Voraussetzungen wie in Version 2 vor. Der Agent bewegt sich in der *Update()* Methode ein gleichbleibendes Stück vorwärts und muss nur die Rotation bestimmen.

Ziel der finalen Version 4 ist, dass NEAT auch das Output-Neuron, das die Vorwärtsbewegung bestimmt, richtig optimieren muss. Zu diesem Zweck wird die maximale Geschwindigkeit des Agenten stark erhöht. Sollte ein Agent jetzt die Strecke mit der maximalen Geschwindigkeit abfahren, ist es durch den größeren Wendekreis unmöglich, die scharfen Kurven zu fahren. In dieser Implementierung muss ein Agent, um den Fitnesswert zu maximieren, in den Kurven abbremsen und auf gerader Strecke auf die maximale Geschwindigkeit beschleunigen.

#### 4.2.6 Sequenzielle vs. Parallele Evaluation

Bevor die Evaluation startet, wird entschieden, ob die Agenten sequenziell oder parallel evaluiert werden sollen. Beide Alternativen bieten Vor- und Nachteile, die gegeneinander abgewogen werden müssen.

Grundsätzlich ist das Ziel, dass jeder Agent dieselbe Aufgabe mit denselben Hindernissen absolvieren muss.

Bei der sequenziellen Evaluation ist dies sehr einfach umzusetzen. Die Evaluation eines Agenten wird gestartet und dieser muss das Level absolvieren. Am Ende der Evaluation werden die Änderungen im Level, zum Beispiel getötete Gegner und Positionen von Gegenständen, zurückgesetzt. Danach beginnt die Evaluation des nächsten Agenten, dieser hat dieselben Voraussetzung wie der vorherige Agent.

Bei der parallelen Evaluation bedeutet dies Mehraufwand bei der Programmierung. Das Training muss so erstellt werden, dass die Agenten sich nicht gegenseitig behindern und dass kein Agent von den Leistungen eines anderen profitiert. In diesem Beispielprojekt ist der Aufwand für die benötigten Maßnahmen gering. Die Kapseln können sich nicht gegenseitig berühren und die *Raycasts*, die den Abstand messen, werden nicht von anderen Kapseln abgebrochen. Großer Vorteil der parallelen Evaluation ist, dass die benötigte Trainingszeit stark verringert wird. Dies wird mit folgendem Beispiel belegt. Die Populationsgröße beträgt 300 Agenten und die durchschnittliche Evaluationszeit beträgt 15 Sekunden. Die maximale Evaluationszeit eines Agenten beträgt 30 Sekunden. Die benötigte Zeit für die Evaluation einer Generation mit dem sequentiellen Verfahren wird wie folgt berechnet:  $300 \frac{\text{Agenten}}{\text{Generation}} * 15 \frac{s}{\text{Agent}} = 4500 \frac{s}{\text{Generation}}$

Für die Evaluation einer Generation werden 4500 Sekunden benötigt. Dies entspricht 75 Minuten. Das bedeutet, dass die Evaluation von 100 Generationen 125 Stunden benötigt. Bei der parallelen Evaluation benötigt eine Generation nur solange wie die längste Evaluation dauert. In diesem Beispiel beträgt der Wert 30 Sekunden. Die Evaluation von 100 Generationen benötigt dann nur noch 50 Minuten.

Die parallele Evaluation benötigt mehr Rechenleistung als die sequenzielle Evaluation, da alle neuronalen Netze parallel rechnen. Dies kann problematisch sein, wenn größere neuronale Netze benötigt werden, aber in diesem Beispielprojekt stellt dies kein Problem dar. Aus diesem Grund und der bedeutend schnelleren Trainingszeit wird in diesem Projekt die parallele Evaluation verwendet.

#### 4.2.7 Ergebnis

In diesem Kapitel werden die Ergebnisse der finalen Projektversion untersucht. Ziel der Agenten ist, das Level so schnell wie möglich zu lösen. In jeder Generation wird der beste Fitnesswert, der durchschnittliche Fitnesswert und die Anzahl an Spezies dokumentiert. Die Agenten werden insgesamt 100 Generationen lang trainiert. Die Ergebnisse der höchsten und durchschnittlichen Fitness sind in Abbildung 48 zu sehen.

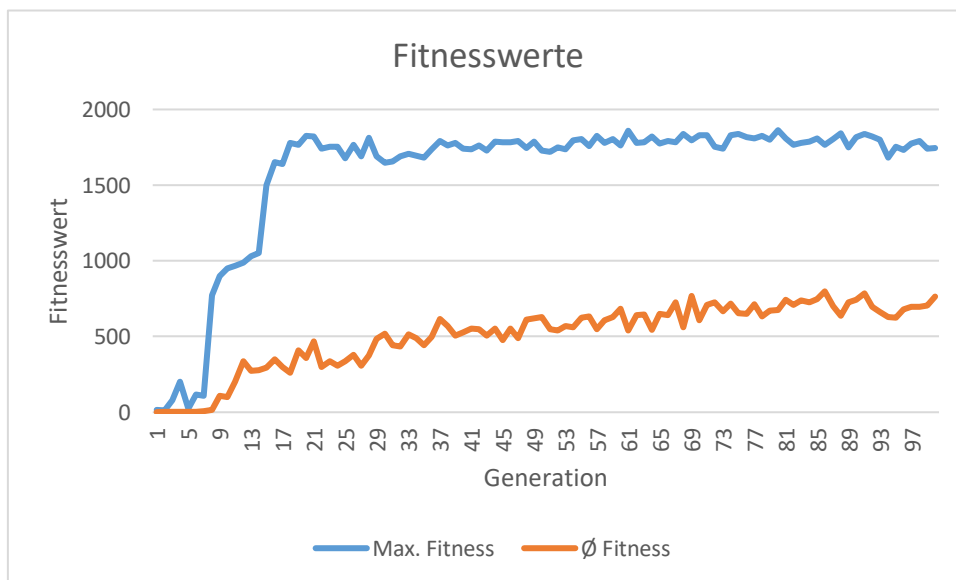


Abbildung 48 Höchster und durchschnittlicher Fitnesswert der Agenten in jeder Generation im Projekt Autonomes Fahren

Die blaue Linie zeigt den höchsten Fitnesswert der Generationen an. In Generation 15 hat der erste Agent das Ziel erreicht. Dieser hat den Zeitbonus erhalten und dadurch im Vergleich zu den vorherigen Generationen einen bedeutend höheren Fitnesswert von  $\approx 1500$  Punkten erhalten. In den folgenden Generationen wird versucht, die benötigte Zeit zu minimieren, um einen höheren Zeitbonus zu erhalten. Der höchste Fitnesswert wurde in Generation 80 erreicht, mit einem Wert von  $\approx 1863$  Punkten. Aus dem Diagramm ist ersichtlich, dass der Fitnesswert ab Generation 21 nicht mehr stark zunimmt. Das zeigt, dass das neuronale Netz sich der maximal erreichbaren Fitness angenähert hat. Kleine Steigerungen wie in Generation 80 sind noch möglich, treten aber selten auf.

Die orange Linie zeigt die durchschnittliche Fitness in jeder Generation. Diese nimmt im Verlauf der Trainingszeit zu, schwankt aber zwischen den Generationen. Dies kann



dadurch erklärt werden, dass nicht garantiert ist, dass Mutationen den Fitnesswert steigern. Zudem werden Spezies, auch wenn sie gute Fitnesswerte haben, nach 15 Generationen aussterben, wenn die maximale Fitness nicht gesteigert wird. In diesem Fall bekommen auch Spezies, die nicht so gute Fitnesswerte haben, mehr Agenten zugeteilt. Infolgedessen sinkt der durchschnittliche Fitnesswert.

Im Gegensatz zum durchschnittlichen Fitnesswert sollte der maximale Fitnesswert nicht sinken. Das Genom des besten Agenten in jeder Generation wird in die nächste Generation kopiert. Der Agent der nächsten Generation trifft dieselbe Entscheidung, wenn das neuronale Netz und die Inputwerte gleich zur vorherigen Generation sind. Aber, wie in Abbildung 48 zu sehen ist, schwankt der maximale Fitnesswert zwischen den Generationen. Das bedeutet, dass entweder das neuronale Netz nicht richtig kopiert wird oder dass die Input-Werte unterschiedlich sind. Da durch Tests und das XOR-Beispiel die richtige Funktionalität der Library bestätigt ist, müssen die Input-Werte schwanken.

Der Parameter, der sich zwischen den Generationen ändern kann, ist die Bildrate, beziehungsweise die Zeit zwischen zwei gerenderten Bildern. Diese Zeit ist maßgeblich an der Bewegung von Charakteren und Objekten in einem Spiel beteiligt. Dies wird an folgendem Beispiel demonstriert: In einem Spiel soll sich ein Charakter maximal  $10 \frac{m}{s}$  vorwärtsbewegen können. Die Bewegung soll flüssig dargestellt werden. Aus diesem Grund wird der Charakter bei jedem gerenderten Bild ein Stück in die gewünschte Richtung bewegt. Bei einem System mit 50 Bildern pro Sekunde, beträgt die Zeit zwischen den Bildern  $\frac{1}{50} s$ . Die benötigte Bewegung des Agenten wird berechnet, indem die verstrichene Zeit seit dem letzten Bild mit der maximalen Bewegungsgeschwindigkeit multipliziert wird:  $\frac{1}{50} s * 10 \frac{m}{s} = 0,2m$ . Der Agent wird in einer Sekunde 50-mal vorwärtsbewegt und in jedem Bild wird die Position um 0,2 Meter verschoben. So bewegt sich der Agent in einer Sekunde maximal 10 Meter vorwärts. Bei einem System mit einer geringeren Bildrate wird der Agent weniger häufig vorwärtsbewegt, aber dafür mit proportional größeren Schritten.

Wenn die Bildrate während des ganzen Trainings konstant wäre, würde ein Agent bei mehrfacher Evaluation immer dieselben Input-Werte haben, dasselbe Ergebnis berechnen und durch die konstante Bildrate die gleiche Bewegung ausführen. Der Fitnesswert wäre somit immer gleich und würde nicht sinken. Die Bildrate wird durch verschiedene interne und externe Parameter beeinflusst und ist nicht konstant. Zum Beispiel wird die Bildrate von der Anzahl an aktiven Agenten in der Evaluation und der restlichen Systemlast beeinflusst. Je höher die Last im System, desto geringer wird die Bildrate.

Bei der Evaluation desselben Agenten zu zwei unterschiedlichen Zeitpunkten können unterschiedliche Bildwiederholungsraten vorliegen. Infolgedessen wird der Agent unterschiedlich bewegt. Da die Position des Agenten die Input-Werte bestimmt, werden

bei der nächsten Berechnung andere Input-Werte vorliegen. Somit wird auch das Ergebnis ein anderes sein. Je mehr Berechnungen durchgeführt werden, desto weiter können die Positionen auseinanderdriften. Dies kann dazu führen, dass ein Agent aufgrund einer günstigen Bildrate in einer Evaluation auf der Strecke besonders weit kommt und einen sehr hohen Fitnesswert erhält. Der Agent wird in die nächste Generation kopiert, schafft aber in dieser Evaluation nicht, dieselbe Strecke erneut zu fahren, da sich die Bildrate geändert hat.

Um den unerwünschten Effekt zu verhindern, dass die maximale Fitness sinkt, gibt es in diesem Beispielprojekt eine Option, mit der die Bewegung nicht mit der Zeit zwischen den gerenderten Bildern, sondern mit einem festen Wert multipliziert wird. Dies führt dazu, dass bei einer hohen Last im System die Berechnung einer Sekunde im Spiel mehrere Sekunden in Echtzeit benötigt. Umgekehrt kann bei niedriger Last eine Sekunde im Spiel mit weniger Zeit berechnet werden.

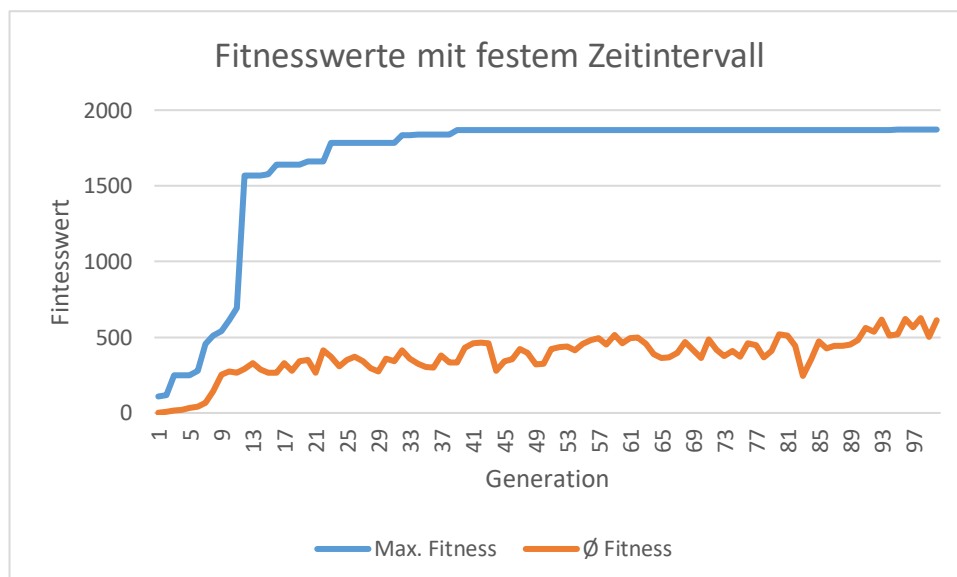


Abbildung 49 Höchster und durchschnittlicher Fitnesswert der Agenten in jeder Generation im Projekt Autonomes Fahren mit einem festen Zeitintervall

Die Agenten werden erneut 100 Generationen mit einem festen Zeitintervall trainiert und die Ergebnisse protokolliert. Diese sind Abbildung 49 dargestellt. Es ist zu sehen, dass die maximale Fitness in jeder Generation steigt oder gleichbleibt. Die im vorherigen Durchgang aufgetretenen Schwankungen des Fitnesswertes sind demnach der dynamischen Bildrate geschuldet. In diesem Durchlauf hat der erste Agent in Generation 12 das Ziel erreicht. Die letzte Steigerung der Fitness ist in Generation 95 aufgetreten. In dieser hat der Fitnesswert  $\approx 4$  Punkte zugenommen und einen Gesamtwert von  $\approx 1872$  Punkten erreicht. Vor dieser Veränderung lag der maximale Fitnesswert bei  $\approx 1868$  Punkten, welcher erstmalig in Generation 39 erzielt wurde. In den beiden vorgestellten Durchläufen war der höchste Fitnesswert bis auf wenige Punkte sehr ähnlich.

Auch bei weiteren Durchläufen konnten keine stark höheren Fitnesswerte erreicht werden. Das zeigt, dass die Library auch bei mehreren Durchläufen ähnliche Fitnesswerte erreicht.

Neben den Fitnesswerten wird auch die Anzahl an Spezies dokumentiert. Es werden zwei Abschnitte genauer betrachtet. Zum einen wird betrachtet, wann die ersten neuen Spezies erscheinen und zum anderen, wie viele Spezies existieren, wenn die Fitness stagniert. Bei einer falschen Parametrisierung kann es dazu kommen, dass so viele Spezies entstehen, dass sich in jeder nur noch ein einzelner Agent befindet. Die Anzahl an Spezies in jeder Generation sind in Abbildung 50 dargestellt.

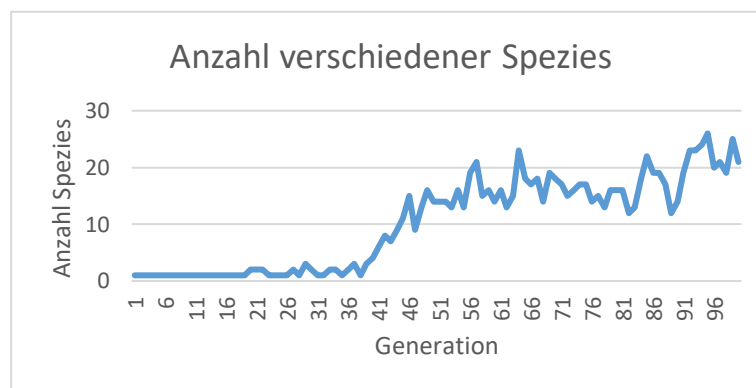


Abbildung 50 Anzahl verschiedener Spezies pro Generation im Projekt Autonomes Fahren

Zu Beginn der Evaluation gibt es bis Generation 19 nur eine Spezies. Nach Generation 38 steigt die Anzahl rasch an und schwankt danach zwischen 12 und 26 verschiedenen Spezies. Während des starken Anstiegs des Fitnesswertes gab es nur wenige Spezies. Als der Fitnesswert anfängt zu stagnieren, entstehen mehr Spezies, die verschiedene Ansätze zur Steigerung der Fitness verfolgen. Der Anzahl an Spezies ist auch bei langer Evaluation ohne Steigerung der Fitness konstant. Es ist nicht der Fall, dass zu viele Spezies entstehen.

Als Letztes wird das entstandene neuronale Netz untersucht. Es wird betrachtet, wie groß das neuronale Netz gewachsen ist. Das neuronale Netz des besten Agenten hat 4 Hidden-Neuronen entwickelt und insgesamt neun Verbindungen hinzugefügt. Im Vergleich zur Trainingsdauer sind dies nur sehr wenige strukturelle Änderungen. Das zeigt, dass NEAT versucht, eine minimale Struktur zu entwickeln und nicht unkontrolliert neue Verbindungen oder Neuronen hinzufügt.

### 4.3 Mario Klon

Im Projekt Autonomes Fahren hat NEAT erfolgreich die Agenten trainiert. Aber diese hatten nur fünf Eingabewerte. Im letzten Beispielprojekt wird eine vereinfachte Version des ersten Levels vom Spiel *Super Mario Bros.* implementiert. Für dieses wird ein bedeutend größeres neuronales Netz benötigt.

### 4.3.1 Spielprinzip

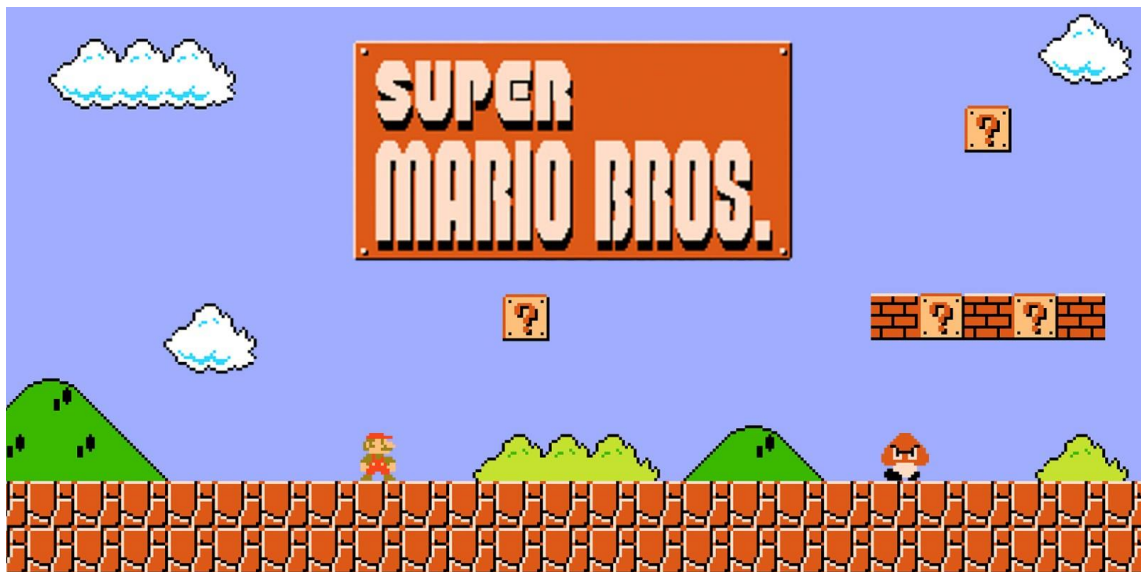


Abbildung 51 Titelbild von *Super Mario Bros.* (Nintendo).

Das originale Spiel (Abbildung 51) wurde 1985 von Nintendo für die NES-Konsole veröffentlicht. Ziel dieses zweidimensionalen Spiels ist, die Spielfigur Mario durch das Level von links nach rechts zu steuern. Das Ende des Levels wird durch eine Flagge markiert. Wenn Mario diese berührt, hat er das Level erfolgreich absolviert. Um sein Ziel zu erreichen, stehen Mario verschiedene Aktionen zur Verfügung. Er kann sich nach rechts und links bewegen sowie springen.

Neben der Bewegung gibt es weitere Aktionen, die abhängig von Zustand sind, in dem Mario sich befindet. Mario kann sich im Zustand *Klein*, *Groß* und *Feuer* befinden. Im Zustand *Klein* hat Mario keine weiteren Fähigkeiten. Wenn er sich im Zustand *Groß* befindet, kann er manche Blöcke zerstören. Im Zustand *Feuer* kann er zudem Feuerbälle schießen, die Gegner töten können.

Von diesen gibt es im ganzen Spiel verschiedene Arten, im ersten Level erscheinen jedoch nur die Gumbas. Diese ähneln vom Aussehen her kleinen braunen Pilzen und sind Standardgegner, die mehrfach im Level vorkommen. Die Gumbas sind nicht sehr intelligent. Sie bewegen sich unabhängig vom Spieler nach rechts und links. Wenn Mario die Gumbas berührt und im Zustand *Klein* ist, stirbt er. Befindet er sich im Zustand *Groß*, wird er *Klein*. Falls er einen Gegner im Zustand *Feuer* berührt, kommt er in den Zustand *Groß*. Die Gumbas können besiegt werden, indem Mario ihnen auf den Kopf springt. In diesem Fall sterben diese, verschwinden aus dem Level und Mario erleidet keinen Schaden.

Neben den Gegnern gibt es noch weitere Gefahren im Level. Wenn Mario das Level nicht in der vorgegeben Zeit schafft, stirbt er. Zudem gibt es Löcher im Level. Wenn Mario in eines dieser fällt, stirbt er ebenfalls unabhängig von seinem Zustand.

Neben dem Hauptziel, das Ende des Levels zu erreichen, gibt es noch Zusatzziele. Eines der Ziele ist, alle im Level verteilten Münzen einzusammeln. Ein anderes ist, das Level so schnell wie möglich abzuschließen oder die höchst mögliche Anzahl an Punkten zu erreichen.

Neben den Münzen gibt es noch weitere Gegenstände, die sich im Level befinden können. Diese sind teilweise in Blöcken versteckt und erscheinen nur, wenn Mario sie zerstört. Ein Gegenstand im Level sind Pilze, die von Mario eingesammelt werden können. Diese versetzen Mario in den Zustand *Groß*, wenn er sich im Zustand *Klein* befindet. Ist Mario bereits im Zustand *Groß*, können Blumen erscheinen. Wenn diese von Mario eingesammelt werden, wird er in den Zustand *Feuer* versetzt (Togelius, Karakovskiy, Koutnik, & Schmidhuber, 2006).

### 4.3.2 Aufbau des Tests in der vereinfachten Version

Ziel dieses Beispielprojektes ist, die Funktionalität der Library in einem komplexeren Umfeld zu testen. Zu diesem Zweck wird eine eigene Version des ersten Levels vom Spiel *Super Mario Bros.* erstellt. Die selbstimplementierte Version hat gegenüber einer fertigen Implementierung einige Vorteile. Einer der großen Vorteile ist, dass in diesem Beispielprojekt nochmals veranschaulicht wird, wie die Library in einem Unity-Projekt verwendet werden kann. Zudem kann auf alle Daten im Level zugegriffen werden. Dies ist besonders hilfreich, wenn die Inputdaten für das neuronale Netz zusammengestellt werden. Zudem kann das Projekt so erstellt werden, dass eine parallele Evaluation der Agenten möglich ist. Dies ist für Demonstrationszwecke besonders wichtig, da wie in Kapitel 4.2.6 erläutert, ein bedeutend schnelleres Training möglich und ein direkter Vergleich zwischen verschiedenen Marios sichtbar ist.

Ein vollständiger Klon des Spiels zu implementieren, ist im zeitlichen Rahmen dieser Arbeit nicht möglich. Aus diesem Grund werden nur die Grundfunktionalitäten des Spiels implementiert. Zudem wird das Level so abgeändert, dass es für die parallele Evaluation geeignet ist.

Eines der größten Probleme bei der parallelen Evaluation ist sicherzustellen, dass ein Mario nicht von den Aktionen eines anderen Mario profitieren kann. Jeder Mario soll dieselben Voraussetzungen in dem Level haben. Die entstehende Problematik wird an folgendem Beispiel veranschaulicht. In dem Level befinden sich zwei Marios, die evaluiert werden. Der erste Mario ist es etwas weiter vorne und steuert auf einen Gumba zu. Der Mario springt auf dessen Kopf und tötet diesen. Der zweite Mario wäre in dieser Situation nicht gesprungen. Wenn der Gumba nach seinem Tod aus dem Level entfernt wird, hat der zweite Mario einen Vorteil, da er ohne den benötigten Sprung im Level weiterkommt und einen höheren Fitnesswert erhält.

Dieses Problem kann auf zwei Arten gelöst werden. Die erste ist, dass das Spiel überwacht, welche Gumbas für welchen Mario existieren. So wird sichergestellt, dass jeder

Mario dieselben Voraussetzungen hat. Die zweite ist, dass das Spiel dahingehend abgeändert wird, dass die Gumbas nicht mehr durch einen Sprung auf den Kopf getötet werden können. Daher stirbt Mario bei jeder Berührung mit einem Gumba. Somit ist es nicht mehr möglich, dass ein Mario das Level in Bezug auf die Gumbas verändert und andere Marios davon profitieren. Die zweite Möglichkeit wird verwendet, da sie im Rahmen dieser Arbeit einfacher zu implementieren ist.

Eine weitere Vereinfachung in dieser Implementierung ist, dass es keine Gegenstände wie Münzen, Pilze und Blumen im Level gibt. Ebenfalls können keine Blöcke des Levels von Mario zerstört werden. Bei diesen Aktionen entsteht das gleiche Problem wie bei den Gumbas. Das Spiel müsste überwachen, welche Blöcke von welchem Mario zerstört wurden und welche Items für welchen Mario verfügbar sind, um die Bewegungen korrekt zu berechnen. Da es keine Pilze und Blumen gibt, kann sich der Zustand des Marios nicht ändern, er bleibt demnach immer im Startzustand *Klein*.

Mit diesen Einschränkungen wird das Spiel in Unity implementiert. Durch die Einschränkungen im Level bleibt neben dem Hauptziel nur noch ein Zusatzziel übrig, nämlich das Level so schnell wie möglich zu beenden. Um dieses Ziel zu erreichen, wird wieder eine Fitnessfunktion benötigt, die das gewünschte Verhalten bewertet. Die Fitnessfunktion besteht in diesem Beispiel wieder aus zwei Teilen. Der erste Teil wird davon beeinflusst, wie weit ein Mario in dem Level nach rechts kommt. Je weiter er nach rechts kommt, desto näher ist er am Ende des Levels und somit an der Erfüllung des Hauptziels. Das Zusatzziel bildet den zweiten Teil der Fitnessfunktion. Je schneller ein Mario das Ziel erreicht, desto besser ist er. Daher bekommt jeder Mario, der das Ziel erreicht, die verbleibende Zeit bis zum Ende der Evaluation als Zeitbonus auf den Fitnesswert addiert.

Die Steuerung des Marios beschränkt sich auf die Bewegung nach links bzw. rechts und auf das Springen. Für diesen Zweck erhält das neuronale Netz zwei Output-Neuronen. Das erste gibt an, wie weit Mario nach links bzw. nach rechts gehen soll, das andere, wie hoch Mario springen soll. Sowohl in der originalen Implementierung als auch in der vereinfachten Version gibt es die Möglichkeit, unterschiedlich hoch zu springen.

Die Input-Werte werden in diesem Projekt durch die umliegenden Blöcke und Gegner bestimmt. Mario kann die umliegenden Bereiche unterteilen und jeden einzelnen Bereich scannen und klassifizieren. Der Bereich bekommt den Wert  $-1$  zugewiesen, wenn der Bereich tödlich ist, sich also ein Gegner in diesem befindet. Ist weder ein Block noch ein Gegner in dem Bereich, bekommt der Bereich den Wert  $0$  zugewiesen. Befinden sich nur Blöcke in dem Bereich, wird diesem der Wert  $1$  zugeteilt. Das Ergebnis aller Klassifizierungen bildet den Input für das neuronale Netz. Somit ist die Anzahl an Input-Neuronen von dem Radius abhängig. Mit diesen Input-Werten soll NEAT eine geeignete Strategie zur Lösung des Problems finden.

### 4.3.3 Implementierung

Die Implementierung der für die Library benötigten Klassen ist sehr stark am Projekt Autonomes Fahren orientiert. Aus diesem Grund werden in diesem Kapitel nur die Funktionsweise des Agenten und die Bauweise des Levels vorgestellt.

Im ersten Schritt wird das Level erstellt (Abbildung 52). Dieses ist aus einzelnen Blöcken zusammengesetzt. Das Leveldesign und die Grafiken werden von dem originalen Spiel übernommen. Lediglich die Positionen und die Anzahl der Gegner werden im Vergleich zum originalen Level verändert.

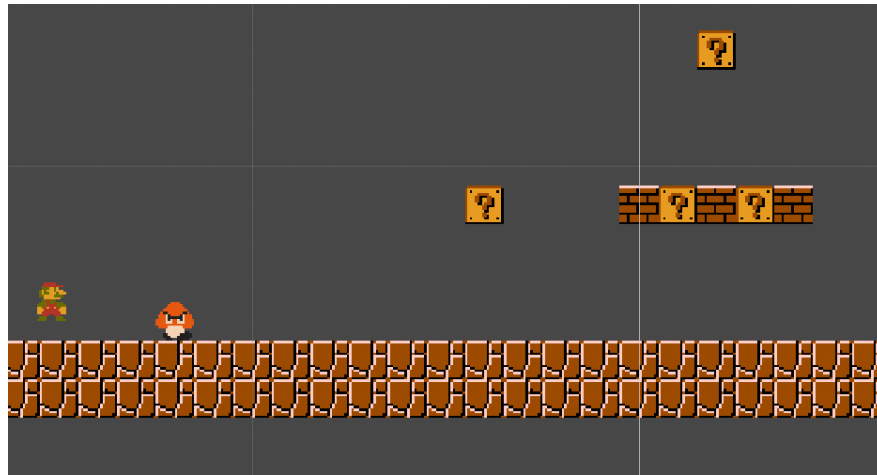


Abbildung 52 Ausschnitt aus dem implementieren Mario Level

Alle Blöcke, Röhren und Gegner haben einen *BoxCollider2D* und ein *Tag*. Der *BoxCollider2D* wird später für die Kollisionsabfrage verwendet, während das *Tag* bei der Klassifizierung von Blöcken Verwendung findet. Das Ende des Levels wird durch die Flagge markiert (Abbildung 53). Berührt ein Mario diese, hat er das Level vollständig absolviert.

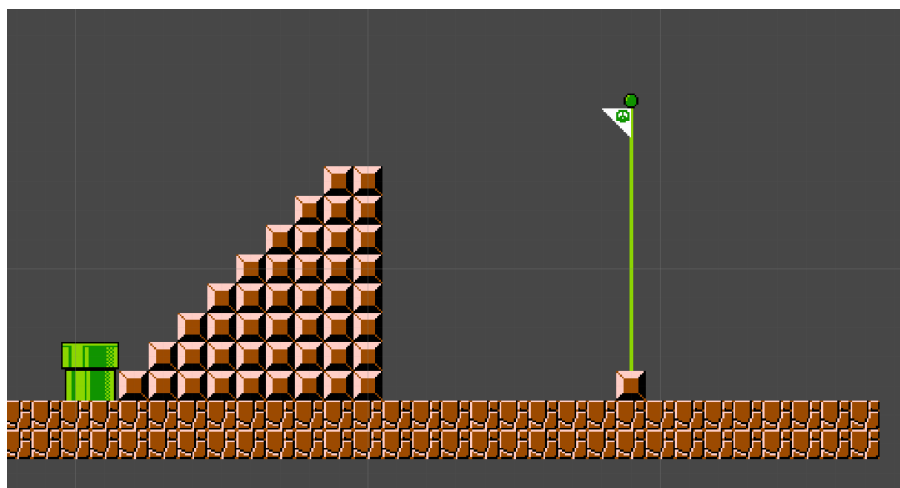


Abbildung 53 Flagge die das Ende des implementierten Levels markiert

Das *GameObject*, das einen Mario repräsentiert, wird durch zwei Skripte gesteuert. Das erste heißt *PlayerMovement* und ist für die Bewegung und Kollisionsabfrage des Marios

zuständig, das zweite ist das *PlayerController* Skript und erhebt die Input-Werte für das neuronale Netz und interpretiert das Ergebnis.

Das *PlayerMovement* Skript hat eine öffentliche Methode *Move()*, die vom *PlayerController* aufgerufen werden kann. Diese nimmt zwei Parameter entgegen. Der erste gibt an, in welche Richtung und wie weit der Charakter laufen soll. Der Parameter muss zwischen  $-1$  und  $1$  sein. Ist der Wert negativ, läuft Mario nach links, ist der Wert positiv, läuft er nach rechts. Der zweite Parameter gibt an, ob und wie stark der Charakter springen soll. Dieser Wert muss zwischen  $0$  und  $1$  sein. Je höher der Wert, desto höher springt Mario. Mario kann wie im originalen Spiel in der Luft nach links und rechts gesteuert werden. Im Gegensatz dazu ist das Springen nur möglich, wenn Mario sich auf dem Boden befindet. Ist dies nicht der Fall und ein Sprungbefehl wird gegeben, wird dieser ignoriert.

Um zu verhindern, dass Mario an eine ungültige Position bewegt wird, gibt es eine Kollisionsabfrage. Diese wird mit *Raycasts* vor jeder Bewegung durchgeführt. Mit den *Raycasts* wird nicht nur überprüft, ob die Zielposition frei ist, sondern auch, ob die Bewegung zu dieser ohne Kollision durchgeführt werden kann. Kommt es bei einer Bewegung zu einer Kollision, wird die Bewegung an der entsprechenden Stelle abgebrochen. So wird verhindert, dass Mario an eine ungültige Position bewegt wird.

Das Skript *PlayerController* hat Zugriff auf das Skript *PlayerMovement* und das neuronale Netz. In der *Update()* Methode erfasst der Controller die Input-Werte, gibt diese an das neuronale Netz, interpretiert das Ergebnis und bewegt Mario mit der Methode *Move()* (Abbildung 54). Bevor die Input-Werte erfasst werden, wird wie im Projekt Autonomes Fahren überprüft, ob die Evaluation des Agenten beendet werden muss.



```
1 void Update()
2 {
3     if (!_alive) return;
4
5     float deltaTime = _useFixDeltaTime ? 0.02f : Time.deltaTime;
6
7     //Update value
8     _timeAlive += deltaTime;
9
10    //Check if evaluation time is expired
11    if (_timeAlive >= _maxTimeForLevel)
12    {
13        KillPlayer(false);
14        return;
15    }
16
17    //Check if character has moved
18    if (!CheckMovement())
19    {
20        KillPlayer(false);
21        return;
22    }
23
24    //Check if a neuronal network is set
25    if (_customAgent != null)
26    {
27        double[] input = GetInput(_levelViewWidht, _levelViewHeight);
28        double[] output = _customAgent.Genome.Calculate(input);
29
30        float move = -1f + 2 * (float)output[0];
31        float jump = (float)output[1];
32        _playerMovement.Move(move, jump);
33    }
34    else
35    {
36        _playerMovement.Move(Input.GetAxis("Horizontal"), Input.GetAxis("Jump"));
37    }
38 }
39 }
```

Abbildung 54 *Update()* Methode eines Mario

Zu diesem Zweck wird zuerst das verwendete Zeitintervall ausgewählt. Auch in diesem Beispiel gibt es die Option, ob die Zeit zwischen zwei gerenderten Bildern oder ein festes Zeitintervall verwendet werden soll. Mit dem verwendeten Zeitintervall wird die Evaluationszeit inkrementiert. Danach wird überprüft, ob die maximale Evaluationszeit überschritten wurde. Ist dies der Fall, wird die Evaluation mit dem Aufruf der Methode *KillAgent()* beendet. Zudem wird überprüft, ob sich der Agent in den letzten Sekunden bewegt hat. Ist dies nicht der Fall, wird die Evaluation ebenfalls beendet.

Danach wird Mario bewegt. Für diesen Fall gibt es zwei Möglichkeiten. Ist ein Agent mit einem neuronalen Netz vorhanden, wird dieses zur Bewegung verwendet. Ist dies nicht der Fall, kann Mario durch Tastatureingaben gesteuert werden.

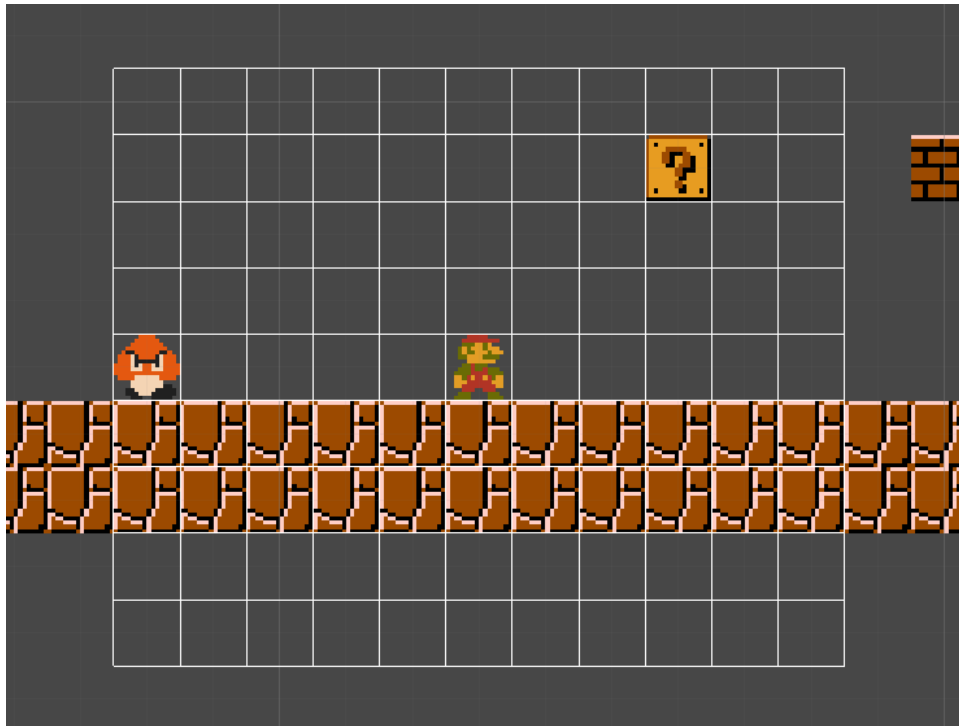
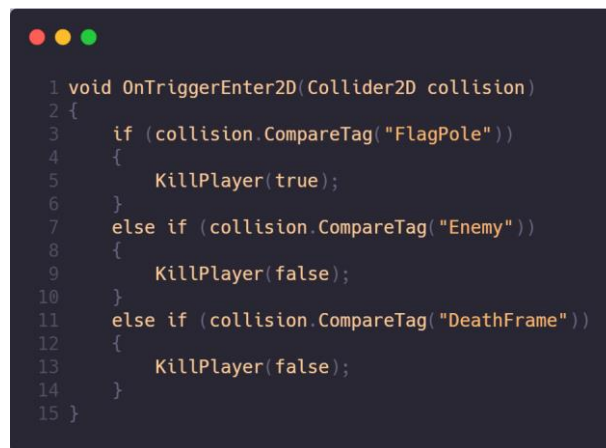


Abbildung 55 Unterteilung des sichtbaren Bereiches in einzelne Blöcke

Um die Eingaben für das neuronale Netz zu erhalten, wird die Methode *GetInput()* aufgerufen. Diese hat zwei Parameter, die angeben, wie breit und wie hoch der gescannte Bereich um Mario sein soll. In der Methode wird der Bereich in einzelne Blöcke unterteilt (Abbildung 55), die dann mit dem Aufruf der Methode *Physics2D.OverlapAreaAll()* gescannt werden. Die Methode gibt alle *Collider2D* zurück, die entweder zu Gegnern oder zu Blöcken gehören. Diese werden dann überprüft und das Ergebnis in ein Array eingetragen. Gehört mindestens ein *Collider2D* zu einem Gegner, wird für den Bereich der Wert *-1* eingetragen. Gibt es nur *Collider2D*, die zu Blöcken gehören, wird der Wert *1* eingetragen. Wurde kein *Collider2D* gefunden, wird der Wert *0* eingetragen.

Das erstellte Input-Array wird an das neuronale Netz gegeben. Das Ergebnis ist ein Array mit zwei Werten, die zwischen *0* und *1* liegen können. Da der Wert für die Bewegung nach links bzw. rechts zwischen *-1* und *1* liegen muss, wird der erste Ergebniswert noch auf die gewünschte Reichweite angepasst. Die Sprunghöhe wird direkt von dem zweiten Output-Neuron übernommen. Die Werte werden dann an die *Move()* Methode übergeben.

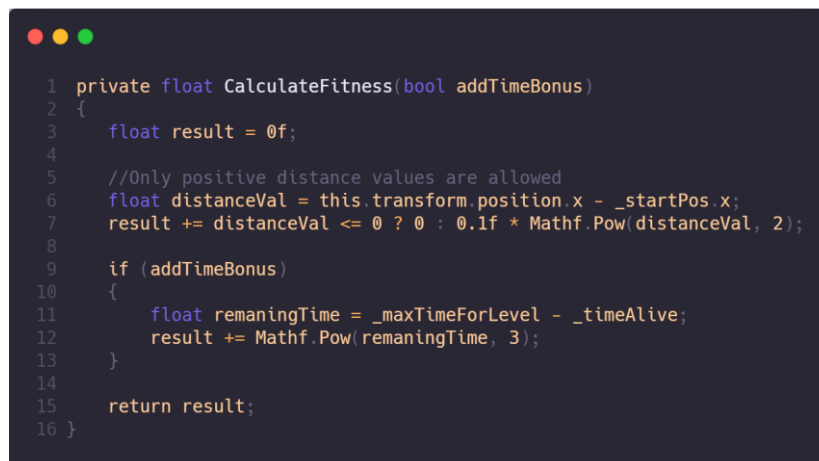
Sollte es zu einer Kollision kommen, wird die Methode *OnTriggerEnter2D()* aufgerufen. Diese ist in Abbildung 56 dargestellt. Bei der Kollision wird das *Tag* des anderen Objektes angefragt und verglichen. Hat Mario die Flagge berührt, wird die Evaluation beendet und der Zeitbonus gegeben. Berührt Mario einen Gegner oder den Rahmen des Levels, wird die Evaluation ebenfalls beendet, in diesem Fall bekommt er aber keinen



```
1 void OnTriggerEnter2D(Collider2D collision)
2 {
3     if (collision.CompareTag("FlagPole"))
4     {
5         KillPlayer(true);
6     }
7     else if (collision.CompareTag("Enemy"))
8     {
9         KillPlayer(false);
10    }
11    else if (collision.CompareTag("DeathFrame"))
12    {
13        KillPlayer(false);
14    }
15 }
```

Abbildung 56 *OnTriggerEnter()* Methode eines Mario

Zeitbonus. Der Rahmen wird für den Fall benötigt, wenn Mario in ein Loch fällt. Durch die Berührung wird die Evaluation direkt beendet. Andernfalls würde Mario bis zum Ende der Evaluation weiterhin fallen.



```
1 private float CalculateFitness(bool addTimeBonus)
2 {
3     float result = 0f;
4
5     //Only positive distance values are allowed
6     float distanceVal = this.transform.position.x - _startPos.x;
7     result += distanceVal <= 0 ? 0 : 0.1f * Mathf.Pow(distanceVal, 2);
8
9     if (addTimeBonus)
10    {
11        float remaningTime = _maxTimeForLevel - _timeAlive;
12        result += Mathf.Pow(remaningTime, 3);
13    }
14
15    return result;
16 }
```

Abbildung 57 Berechnung des Fitnesswertes eines Mario

Nach Beendigung der Evaluation eines Marios, wird der erreichte Fitnesswert berechnet. Die Berechnung ist in Abbildung 57 dargestellt. Zuerst wird die Differenz auf der x-Achse zwischen der aktuellen Position von Mario und seiner Startposition gebildet. Ist Mario nur nach links gelaufen, ist dieser Wert negativ. In diesem Fall wird die Fitness auf 0 zurückgesetzt. Ist der Wert positiv, wird er quadriert und das Ergebnis mit  $0,1$  multipliziert. Die maximal erreichbare Fitness durch die Strecke beträgt  $\approx 3850$  Punkte. Hat Mario die Flagge berührt, wird der Zeitbonus gegeben. Dieser wird berechnet, indem die verbleibende Zeit bis zum Ende der Evaluation berechnet wird. In diesem Beispiel wird der Wert kubiert. Dies wurde experimentell ermittelt. Jeder Mario hat für das Level maximal 30 Sekunden Zeit. Um das Ziel bei gerader Strecke ohne Hindernisse zu erreichen, benötigt Mario bei maximaler Geschwindigkeit mindestens 13 Sekunden. Der maximale Zeitbonus, der gegeben werden kann, beträgt somit  $\approx 4900$  Punkte. Somit stehen beide Werte in einem ähnlichen Verhältnis.

#### 4.3.4 Parametrisierung

In diesem Projekt wird eine Populationsgröße von 300 Agenten verwendet. Auch die restlichen NEAT spezifischen Parameter werden von dem Projekt Autonomes Fahren übernommen.

Um die Inputs für das neuronale Netz zu erstellen, kann Mario die Blöcke in einem gegebenen Radius klassifizieren. Die Sichtweite beträgt in diesem Projekt 5 Felder nach vorne und hinten, 4 Felder nach oben und unten, sowie die Reihe und Spalte der aktuellen Position. Diese Werte wurden gewählt, da die Sichtweite so etwas größer ist als die Sprungreichweite. Sollte ein Mario sich dazu entscheiden, einen Sprung zu auszuführen, stehen ihm alle wichtigen Informationen zur Verfügung.

Insgesamt hat Mario 99 verschiedene Inputs. Das neuronale Netz hat 100 Input-Neuronen, von denen eins ein Bias-Neuron ist.

Die Evaluation der Marios erfolgt parallel, sodass die Trainingszeit möglichst gering und ein guter visueller Vergleich zwischen den verschiedenen Agenten möglich ist.

Im Projekt Autonomes Fahren wurde die Evaluation sowohl mit einem festen Zeitintervall als auch mit der dynamischen Bildrate durchgeführt. Da sich in diesem Projekt gezeigt hat, dass die Evaluation mit einem festen Zeitintervall konstantere Fitnesswerte erzielt, wird in diesem Projekt die Evaluation nur mit einem festen Zeitintervall von 0,02 Sekunden durchgeführt. Dies entspricht 50 Bildern pro Sekunde.

#### 4.3.5 Ergebnis

In diesem Kapitel werden die Ergebnisse des letzten Projektes vorgestellt. Das verwendete neuronale Netz ist um ein Vielfaches größer als die bisher verwendeten neuronalen Netze und hat schon zu Beginn 200 Verbindungen. Es wird sich zeigen, ob NEAT es schafft, dieses so zu optimieren, dass eine geeignete Strategie zur Lösung des Levels entsteht.

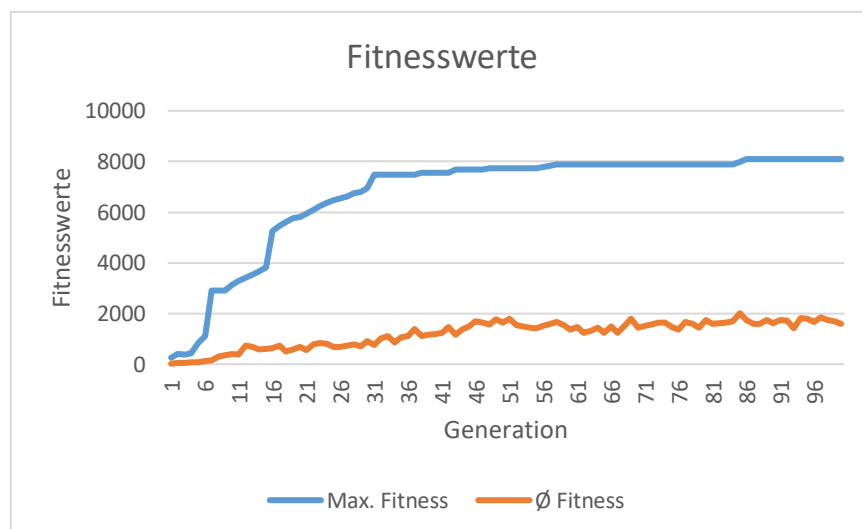


Abbildung 58 Höchster und durchschnittlicher Fitnesswert der Marios in jeder Generation

Die Agenten werden 100 Generationen lang trainiert und die Ergebnisse dokumentiert. Abbildung 58 zeigt den maximal erreichten Fitnesswert und den durchschnittlichen Fitnesswert in jeder Generation. In Generation 16 hat der erste Mario das Ziel erreicht und einen Fitnesswert von  $\approx 5265$  Punkten erhalten. In den folgenden Generationen wurde die benötigte Zeit für das Level drastisch verringert, was zu einer Steigerung des Fitnesswertes auf  $\approx 8094$  Punkte führte. Aus den gegebenen Daten ist ersichtlich, dass zu Beginn der Fitnesswert sehr schnell und gleichzeitig sehr stark gestiegen ist. Das ist vor allem darauf zurückzuführen, dass durch eine kleine Verbesserung große Änderung möglich sind. Schafft es zum Beispiel ein Mario, einen Gegner mehr als seine Konkurrenten zu überwinden, hat er durch die weiter zurückgelegte Strecke eine große Steigerung des Fitnesswertes. Zum Ende werden die Möglichkeiten zur Optimierung bedeutend schwieriger, da die benötigte Zeit nur noch um einige Zehntelsekunden verbessert werden kann.

Wenn Mario nur die horizontale Strecke des Levels ohne Gegner und behindernde Blöcke überwinden müsste, würde dies mindestens 13 Sekunden dauern. Dies ergibt einen maximal erreichbaren Fitnesswert von  $\approx 8754$  Punkten. Würde Mario nur eine Sekunde länger benötigen, hätte er nur noch einen Fitnesswert von  $\approx 7937$  Punkten. Das zeigt, dass die Fitnessfunktion die letzten kleinen Optimierungen besonders stark gewichtet.

Wenn in dem Level alle Gegner und Hindernisse vorhanden sind, ist der maximale Fitnesswert von  $\approx 8754$  Punkten nicht erreichbar. Die Gegner sind in dem Level an Schlüsselstellen platziert. Wenn ein Mario an diesen vorbeikommen möchte, muss er seine Geschwindigkeit reduzieren. Ein Beispiel hierfür ist in Abbildung 59 dargestellt.

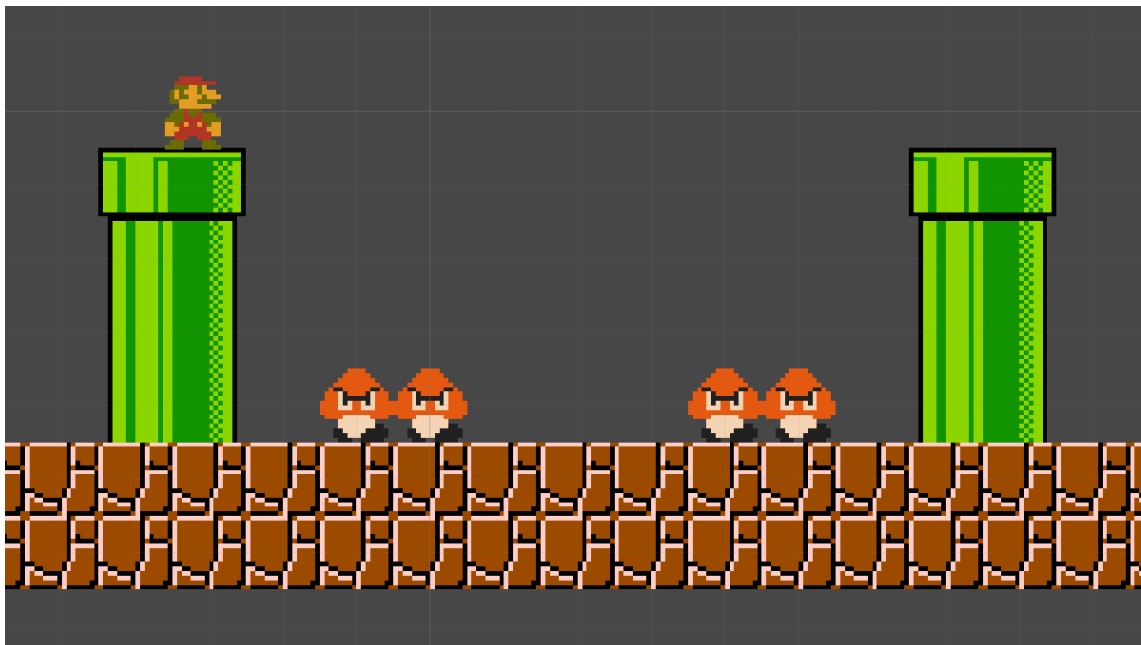


Abbildung 59 Beispiel für einen herausfordernden Abschnitt im implementierten Mario Level

Wenn der Mario mit der maximalen Geschwindigkeit von der Röhre springt, berührt er einen der beiden Gumbas, die sich auf der rechten Seite befinden. Um diese Stelle zu

überwinden, muss er seine Geschwindigkeit reduzieren und in der Mitte der beiden Röhren landen.

Trotz dieser Hindernisse hat es der beste Mario geschafft, das Level mit einem Fitnesswert von  $\approx 8094$  Punkten zu absolvieren. Dies entspricht einer benötigten Zeit von ungefähr 13,8 Sekunden.

Die durchschnittliche Fitness ist im Vergleich zur maximalen Fitness sehr gering. Betrachtet man die zweite Hälfte der Evaluation, liegt der Durchschnitt bei  $\approx 1613$  Punkten. In der Evaluation ist zu sehen, dass viele Marios vor allem am Anfang an den schwierigen Sprüngen scheitern, einen Gegner berühren und sterben. Diese Marios bekommen nur einen sehr geringen Fitnesswert, wodurch die durchschnittliche Leistung stark sinkt. Eventuell wäre dies anders, wenn es zu Beginn weniger Gegner geben würde und die Sprünge somit einfacher wären.

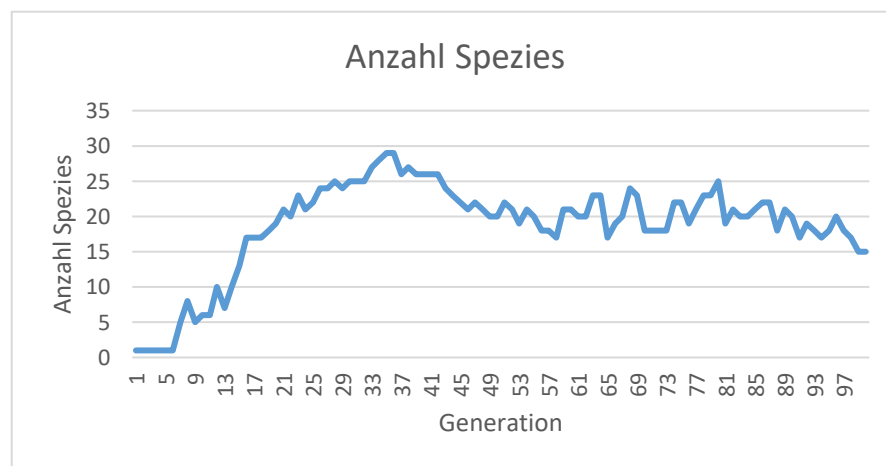


Abbildung 60 Anzahl verschiedener Spezies pro Generation im Mario Projekt

Auch in diesem Projekt wurde die Anzahl an Spezies, die während der Evaluation aufgetreten sind, dokumentiert (Abbildung 60). Wie im Projekt Autonomes Fahren ist auch hier zu sehen, dass in den ersten Generationen nur eine Spezies vorhanden ist. Es dauert einige Generationen, bis die Unterschiede im neuronalen Netz größer werden, so dass mehrere Spezies benötigt werden. Bei der Betrachtung aller 100 Generationen gibt es im Schnitt  $\approx 19$  Spezies pro Generation. Aus dem Diagramm ist ersichtlich, dass dieser Wert auch bei längerer Evaluation konstant ist.

Im letzten Abschnitt dieses Kapitels wird das neuronale Netz betrachtet. Der beste Mario hatte ein neuronales Netz mit 108 Neuronen und 220 Verbindungen. Werden die zu Beginn vorhandenen Neuronen und Verbindungen abgezogen, wurden 6 Neuronen und 20 Verbindungen dem Netz hinzugefügt. Das zeigt, dass für eine Aufgabe wie die Steuerung eines Marios kein großes neuronales Netz mit vielen Hidden-Neuronen benötigt wird.

NEAT beweist mit diesem Ergebnis, dass es nicht eine unnötig große Struktur entwickelt, sondern nur sinnvolle Neuronen beziehungsweise Verbindungen hinzufügt. Trotz

dieser kleinen Struktur schafft es NEAT, sehr gute Fitnesswerte zu erzielen und ein gut optimiertes neuronales Netz zu erstellen.

## 5 Zusammenfassung und Ausblick

Ziel dieser Arbeit ist, den neuroevolutionären Algorithmus NEAT für die Spieleentwicklung zugänglich zu machen. Hierfür wurde der Algorithmus als Library implementiert. In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf weiterführende Implementierungen gegeben.

### 5.1 Ergebnisse

Die Betrachtung der Ergebnisse erfolgt hinsichtlich der Integration der Library als auch auf deren Performance.

#### 5.1.1 Integration

Der erste Schritt zur Verwendung der Library ist die Integration in ein Projekt. Dies ist mit dem erstellten *AssetBundle* ohne großen Aufwand möglich und hat bei allen drei Projekten problemlos funktioniert.

Auch das Design der Schnittstelle ist sehr effizient. Die Library ermöglicht sowohl die parallele als auch die serielle Evaluation. Die parallele Evaluation bietet sich insbesondere an, wenn wie in diesen Projekten die Funktionsweise der neuroevolutionären Algorithmen veranschaulicht werden soll. Die serielle Evaluation kann verwendet werden, wenn die benötigte Trainingszeit nicht maßgeblich ist und bei der Gestaltung des Levels kein zusätzlicher Aufwand gewünscht ist.

Beim Training der neuronalen Netze ermöglicht die Library Spielentwicklern, selbst Abbruchbedingungen zu definieren. So konnte im XOR-Beispielprojekt das Training solange durchgeführt werden, bis eine erfolgreiche Lösung gefunden wurde, während im Projekt Autonomes Fahren und beim Mario-Klon eine feste Anzahl an Generationen durchlaufen wurde.

Auch die Steuerung der einzelnen Agenten ist ohne großen Mehraufwand mit einem neuronalen Netz möglich. Es muss eine Fitnessfunktion zur Bewertung der einzelnen Agenten implementiert werden. Der größte Vorteil der Fitnessfunktion ist, dass diese keine Lösungsstrategie enthalten muss, sondern nur die Lösung bewertet. Mit den Beispielprojekten wurde veranschaulicht, wie einfach diese aufgestellt werden kann.

Des Weiteren muss implementiert werden, wie die Erfassung der Eingabedaten erfolgt. Diese können das Training maßgeblich beeinflussen. Hierbei hat sich gezeigt, dass sowohl die Verwendung kontinuierlicher als auch diskreter Eingabewerte möglich ist. Im Projekt Autonomes Fahren wurden kontinuierliche Werte verwendet, dies waren die Abstandswerte zur Wand. Diskrete Werte wurden unter anderem beim Mario-Klon verwendet. Hier gab es drei verschiedene Werte zur Klassifizierung der Levelabschnitte.

Der letzte Teil, der für die Steuerung implementiert werden muss, ist die Interpretation der Daten. Der Wertebereich der Output-Neuronen ist abhängig von der verwendeten



Aktivierungsfunktion. Die in diesen Projekten verwendete Sigmoid-Funktion liefert Werte im Bereich zwischen 0 und 1. Bei gutem Softwaredesign können diese Werte entweder direkt oder mit wenig Modifizierung zur Steuerung der Agenten verwendet werden. Dies wurde mit den Beispielprojekten demonstriert.

Insgesamt ist die Integration der Library in ein bestehendes Unity-Projekt sehr einfach und mit wenig Aufwand zu realisieren.

### 5.1.2 Performance

Neben der Integration sind vor allem die Ergebnisse der Library wichtig. Um einen Vergleich mit der originalen Implementierung zu ermöglichen, wurde das XOR-Beispielprojekt implementiert.

Die Library hat bewiesen, dass sie das XOR-Problem erfolgreich lösen kann. In jedem Durchlauf wurden die benötigten Strukturen entwickelt und das neuronale Netz richtig optimiert. Einzig die Anzahl der Generationen schwankt, da NEAT ein nicht deterministischer Algorithmus ist. Somit kann auch die durchschnittliche Leistung um einige Generationen abweichen. In dem in dieser Arbeit vorgestellten Durchlauf hat die Library durchschnittlich 34 Generationen zur Lösung des XOR-Problems benötigt. Die originale Implementierung benötigte durchschnittlich 32 Generationen. Die Abweichungen sind somit zu vernachlässigen. Durch Lösen des XOR-Problems beweist die Library, dass sie die benötigten Strukturen entwickeln kann.

Im Projekt Autonomes Fahren und beim Mario Klon wurde die Funktionsweise der Library in einem Spiel veranschaulicht. Die Agenten in beiden Spielen werden mit einem durch die Library erzeugten neuronalen Netz gesteuert. Im Projekt Autonomes Fahren hat die Library bereits nach 12 Generationen, beim Mario-Klon nach 16 Generationen den ersten Agenten ins Ziel gesteuert. Die hierfür benötigte Zeit wurde in den folgenden Generationen noch stark verringert. In beiden Projekten hat die Library sehr gut optimierte neuronale Netze erzeugt.

Während das neuronale Netz im Projekt Autonomes Fahren nur sechs Input-Neuronen hat, hat es im Mario-Klon 100 Input-Neuronen. Trotz der vielen Input-Neuronen schafft es die Library, die wichtigen Informationen zu extrahieren und eine geeignete Strategie zu entwickeln. Das beweist, dass die Library auch größere neuronale Netze optimieren kann.

Das Verhältnis der Anzahl an Hidden-Neuronen zu der Anzahl an Input-Neuronen ist vor allem beim Mario-Klon gering. Die Library hat nur sechs Hidden-Neuronen hinzugefügt und dennoch eine sehr gut optimierte Lösung geschaffen. Das zeigt, dass die Library nur die benötigten Neuronen entwickelt. Diese Eigenschaft macht den Einsatz der Library auch für fachfremde Personen sehr einfach, die die benötigte Anzahl an Hidden-Neuronen nicht einschätzen können.

## 5.2 Weiterentwicklung

Die Integration der Library ist sehr einfach und auch die Performance ist sehr gut. Trotzdem gibt es noch weitere Funktionen, die für eine einfachere Nutzung implementiert werden sollten.

Die optimierten neuronalen Netze müssen für die spätere Nutzung gespeichert werden. Je nach Programm kann es hierfür unterschiedliche Ansätze geben. Durch die lineare Repräsentation der Genome, ist unter anderem eine Speicherung in Textdateien sowie in Datenbanken möglich. Um Spieleentwicklern alle Möglichkeiten offen zu lassen, ist die Erstellung einer Schnittstelle erforderlich, die das Speichern und Laden von Genomen in verschiedene Systeme ermöglicht.

Ein weitere Optimierungsmöglichkeit ist die Verwendung einer Konfigurationsdatei. Die Library funktioniert mit den im XOR-Beispiel verwendeten Parametern in vielen Projekten sehr gut. Dennoch müssen manche Parameter für eine optimale Performance angepasst werden. Durch eine Konfigurationsdatei sind alle Parameter übersichtlich in einer Datei angeordnet und können schnell und einfach geändert werden.

Wie in Kapitel 1 beschrieben, gibt es Spezialisierungen, die auf NEAT beruhen. Diese können noch implementiert und der Library hinzugefügt werden. Besonders interessant für die Spieleentwicklung sind unter anderem *rtNEAT*, *HyperNEAT* und *cgNEAT*.

Der Algorithmus *rtNEAT* ermöglicht die Evolution von Agenten in Echtzeit anstatt in Generationen wie bei NEAT (Stanley, Bryant, & Miikkulainen, 2005). *HyperNEAT* kann bedeutend größere neuronale Netze entwickeln, die mehrere Millionen Verbindungen haben können (Stanley, Ambrosio, & Gauci, 2009). Der Algorithmus *cgNEAT* kann für die Generierung von Spieleinhalten verwendet werden (Hastings, Guha, & Stanley, 2009).

## 5.3 Einsatz in kommerziellen Spielen

Obwohl neuroevolutionäre Algorithmen eine Vielzahl an Anwendungsmöglichkeiten für die Spieleentwicklung bieten, werden die Algorithmen in kommerziellen Spielen selten eingesetzt (Risi & Togelius, 2015).

Grund hierfür ist vor allem, dass größere neuronale Netze eine Black Box sind. Für Menschen ist es oft schwer nachzuvollziehen, warum ein neuronales Netz eine Entscheidung getroffen hat. Das macht Spieleentwicklern die Kontrolle über ihre Charaktere bedeutend schwieriger. Zudem kann es passieren, dass neuronale Netze nicht vorhergesehene Entscheidungen treffen. Dies ist zum Beispiel ein großes Problem, wenn ein neuronales Netz einen Charakter in einem Spiel tötet, der für die Geschichte elementar ist (Risi & Togelius, 2015).

Ein weiteres Problem kann auftreten, wenn die neuronalen Netze trainiert werden, während das Spiel gespielt wird. Steuert das neuronale Netz zum Beispiel einen Geg-

ner, kann das Netz während des Spiels neue Strategien lernen und sein Verhalten anpassen. Dies macht es Entwicklern bedeutend schwieriger sicherzustellen, dass Gegner nicht zu stark oder zu schwach sind (Risi & Togelius, 2015).

Trotz dieser Nachteile haben neuroevolutionäre Algorithmen Anwendung einigen Spielen gefunden. Zu diesen zählen unter anderem *Creatures*, *GAR*, *Petalz*, *Black&White* und *Colin McRae Rally 2*. Um in Zukunft neuroevolutionäre Algorithmen attraktiver für die Spieleentwicklung zu machen, ist es nötig, die Nachteile zu analysieren und gegebenenfalls die Spieleentwickler mit neuen Tools zu unterstützen (Risi & Togelius, 2015).

Auch die in dieser Arbeit entstandene Library kann einen Teil hierfür leisten. Die Library bietet Spieleentwicklern einen einfachen Zugriff auf den NEAT Algorithmus und ermöglicht so eine schnelle Evaluation hinsichtlich der Eignung des NEAT Algorithmus.

## Quellenverzeichnis

- Bling, S. (13. Juni 2015). MarI/O - Machine Learning for Video Games. Abgerufen am 25. Oktober 2018 von <https://www.youtube.com/watch?v=qv6UVOQ0F44>
- Buckland, M. (2002). *AI Techniques for Game Programming*. Cincinnati, Ohio: Premier Press.
- Burkill, J. (9. Juli 2016). *Machine Learning and Optimisation*. Abgerufen am 5. Januar 2019 von <http://www.mlopt.com/?p=160>
- Frochte, J. (2018). *Maschinelles Lernen: Grundlagen und Algorithmen in Python*. Carl Hanser Verlag GmbH & Co. KG.
- Hastings, E., Guha, R., & Stanley, K. (2009). Automatic Content Generation in the Galactic Arms Race Video Game. *IEEE Transactions on Computational Intelligence and AI in Games* (S. 245-263). IEEE.
- Lippmann, R. P. (1987). An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine* (S. 4-22). IEEE.
- Nintendo. (kein Datum). Super Mario Bros. Startbildschirm. Abgerufen am 15. Januar 2019 von <https://www.nintendo.de/Spiele/NES/Super-Mario-Bros--803853.html>
- Risi, S., & Togelius, J. (27. October 2015). Neuroevolution in Games: State of the Art and Open Challenges. *IEEE Transactions on Computational Intelligence and AI in Games* (S. 25-41). IEEE.
- Rojas, R. (1996). *Neural Networks - A Systematic Introduction*. Berlin: Springer.
- Russell, S. J., & Peter, N. (1994). *Artificial Intelligence - A Modern Approach*. Prentice Hall.
- Scherer, A. (1997). *Neuronale Netze: Grundlagen und Anwendung*. Braunschweig/Wiesbaden: Friedr. Vieweg & Sohn Verlagsgesellschaft.
- Stanley, K. (5. Mai 2015). *The NeuroEvolution of Augmenting Topologies (NEAT) Users Page*. Abgerufen am 05. Dezember 2018 von <https://www.cs.ucf.edu/~kstanley/neat.html#updates>
- Stanley, K. (13. Juli 2017). *O'Reilly*. Abgerufen am 7. Januar 2019 von <https://www.oreilly.com/ideas/neuroevolution-a-different-kind-of-deep-learning>
- Stanley, K., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* (S. 99-127). MIT Press.
- Stanley, K., Ambrosio, D., & Gauci, J. (2009). A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life 2009 Vol. 15* (S. 185-212). MIT Press.

- Stanley, K., Bryant, B., & Miikkulainen, R. (2005). Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation* (S. 653-668). IEEE.
- Togelius, J., Karakovskiy, S., Koutnik, J., & Schmidhuber, J. (2006). Super Mario Evolution. *IEEE Symposium on Computational Intelligence and Games* (S. 6). Italy: IEEE. Abgerufen am 15. Januar 2019 von <https://ieeexplore.ieee.org/abstract/document/5286481>
- Wicker, K. (2015). *Evolutionäre Algorithmen*. Leipzig: Springer.

## Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt wurde, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen und unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

---

Ort, Datum

---

Unterschrift