Technische Hochschule
Brandenburg
University of
Applied Sciences

Konzeption und Umsetzung eines Neuroevolutionären Algorithmus zur Steuerung eines Fahrzeugs in Unity

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science
des Fachbereichs Informatik der
Technischen Hochschule Brandenburg

vorgelegt von:
Mulham Alesali

Betreuer: Dipl.-Inform. Ingo Boersch
Zweitgutachter: Prof. Dr.-Ing. Jochen Heinsohn

Alesali, Mulham Bachelor Informatik 7. Fachsemester	alesali@th-brandenburg.de Matr.-Nr. 2020-3753

Brandenburg/H., den 22. März 2021

Danksagung

Mein Dank gilt zuerst meinem Betreuer, Herrn Dipl.-Inform. Ingo Boersch, der mich bei der Wahl des vorliegenden Themas bestärkt hat und mir bei der Planung und praktischen Umsetzung der Arbeit eine große Unterstützung war. Seine zahlreichen Vorschläge, Hinweise und Anmerkungen haben mir sehr geholfen und damit die Arbeit bereichert.

Weiterhin bedanke ich mich bei Herrn Prof. Dr. -Ing. Jochen Heinsohn für seine Bereitschaft als Zweitgutachter dieser Arbeit tätig zu sein.

Darüber hinaus danke ich Frau Dr. Petra Hoffmann für die sprachliche Begleitung des Arbeitsprozesses und ihre vielfältigen Anregungen und Korrekturvorschläge beim Schreiben der Arbeit. Insbesondere zu erwähnen sind hierbei ihre wertvollen Hinweise für das Verfassen einer schriftlichen Arbeit im Allgemeinen sowie ihre Unterstützung bei der sprachlichen Gestaltung.

Ebenfalls danke ich meinen Eltern, die trotz der Entfernung immer für mich da sind.

Inhaltsverzeichnis

1 Zielsetzung der Arbeit, Themaeingrenzung und methodisches Vorgehen

Neuroevolutionäre Algorithmen sind eine Kombination aus Künstlichen Neuronalen Netzen (KNN) und Evolutionären Algorithmen (EA). Diese sind zwei leistungsstarke Methoden der Künstlichen Intelligenz (KI), die von biologischen Systemen inspiriert sind. Dabei können komplexe KI-Probleme mit einfach zu verstehenden Verfahren effizient gelöst werden. Im Vergleich zu anderen Lernmethoden für Neuronale Netze ist die Neuroevolution sehr allgemein und erlaubt das Lernen ohne explizite Ziele zu definieren.

Das Ziel der Arbeit ist die Konzeption und Implementierung eines Neuroevolutionären Algorithmus zur Lösung einer Fahraufgabe in einem einfachen simulierten Szenario eines selbstfahrenden Fahrzeugs. Das Fahrzeug soll autonom auf einer Strecke fahren, ohne jemals vom Weg abzukommen.

Zur Erfüllung dieser Zielstellung werden zunächst die relevanten theoretischen Grundlagen zu den Künstlichen Neuronalen Netzen, den evolutionären Algorithmen und Neuroevolutionären Algorithmen dargestellt. Dabei wird auf die Elemente eines Neuroevolutionären Algorithmus besondere Aufmerksamkeit gerichtet. Ausgehend davon wird im praktischen Teil der Arbeit ein Algorithmus programmiert, der die Anforderungen eines Neuroevolutionären Algorithmus erfüllt. Wesentlich sind dabei die Funktionsweise und die Leistungsfähigkeit dieses Algorithmus. Dieser Algorithmus soll ein Neuronales Netz optimieren, das ein Fahrzeug kontrolliert. Damit werden Kollisionen mit Hindernissen vermieden und gleichzeitig wird die Fahrzeit optimiert. Indem das Fahrzeug entsprechend den jeweiligen Bedingungen die Geschwindigkeit variiert und sie den Besonderheiten der Route anpasst, folgt es einem optimalen Pfad. Das Fahrzeug „sieht“ die Umgebung mit Hilfe von Sensoren, die dem Algorithmus Informationen zum Abstand zwischen dem Fahrzeug und den Hindernissen liefern. Die vom Neuroevolutionären Algorithmus gefundenen Lösungen sollen generalisierbar sein. Das erlaubt dem Fahrzeug, Strecken mit demselben Prinzip zu fahren. Diese Anforderungen sind Aspekte, die im Rahmen des zu entwickelnden Algorithmus realisiert werden sollen.

Die autonomen Fahrzeuge, die mit dem zu entwickelnden Algorithmus bewegt werden, werden nur über eine Lenk-, Brems- und Beschleunigungssteuerung verfügen. Viele Faktoren wie Fußgänger, Verkehrsregeln und Wetter werden im Rahmen dieser Arbeit nicht berücksichtigt. Daher können die Testergebnisse von den realen Bedingungen im

Straßenverkehr abweichen. Um die Leistungsfähigkeit des Algorithmus zu demonstrieren und nachzuweisen, werden digitale Simulierungen der Strecken und der Bewegung eines Fahrzeuges mit Hilfe von Unity erstellt.

Die Arbeit soll schließlich am praktischen Beispiel zeigen, dass sich ein Neuroevolutionärer Algorithmus zur Lösung einer Fahraufgabe in einem einfachen simulierten Szenario eignet.

2 Theoretische Grundlagen

Die vorliegende Bachelorarbeit konzentriert sich auf Neuroevolutionsalgorithmen für den Entwurf eines Fahrzeugsteuerungssystems. Die Handlungsvorhersage ist ein Entscheidungsproblem, das auf Beobachtung basiert. Eine optimierte Lösung eines solchen Problems erhält man durch den Optimierungsalgorithmus.
In diesem Abschnitt werden einige grundlegende theoretische Konzepte und Forschungserkenntnisse für Neuroevolution dargestellt.

2.1 Grundlagen des maschinellen Lernens

In der klassischen Programmierung gibt der Mensch Daten ein, die nach von Menschen bestimmten Regeln (einem Programm) verarbeitet werden, Im Ergebnis erscheinen die Antworten. Beim maschinellen Lernen gibt der Mensch sowohl Daten als auch die von den Daten erwarteten Antworten ein, und im Ergebnis erscheinen die Regeln. Diese Regeln können dann auf neue Daten angewandt werden, um neue Antworten zu erzeugen, wie Abbildung 1 zeigt. (1)

Abbildung 1 Der Unterschied zwischen Machine Learning und „klassischer“ Software (1)

Ein maschinelles Lernsystem wird mit vielen "Beispielen" konfrontiert, die für eine Aufgabe relevant sind, und findet eine statistische Struktur in diesen Beispielen, die es dem System schließlich erlaubt, Regeln für die Automatisierung der Aufgabe zu finden. (1)

Techniken des maschinellen Lernens können in drei verschiedene Kategorien eingeteilt werden (2):

- Überwachtes Lernen (engl. Supervised learning)
  Beim überwachten Lernen müssen die richtigen Antworten als »Labels« mitgeliefert werden.
- Bestärkendes Lernen (engl. Reinforcement learning)
  Verstärkungslernen ist eine Methode des Machineles Lernen, indem es Interaktionen mit der Umgebung stattfindet, d. h., der Algorithmus führt einige Aktionen in der Umgebung aus und erhält Feedback (z.B. Spiele, Verhandlungen, der wissenschaftliche Forschungsprozess usw.) (3)
- Unüberwachtes Lernen (engl. Unsupervised learning)
  Beim unüberwachten Lernen gibt es nur die rohen Beispieldaten, ohne Labels oder Feedback. Damit können vereinfachende Beschreibungen der gesamten Beispielmenge gefunden werden. So können Daten in verschiedene Gruppen oder Cluster unterteilt oder die Dimensionen, also die Anzahl der Merkmale reduziert werden. (4)

2.2 Künstliches Neuronales Netz

Ein Künstliches Neuronales Netz ist ein Berechnungsmodell, das einige Eigenschaften mit dem tierischen Gehirn teilt, in dem viele einfache Einheiten (Künstliche Neuronen) parallel arbeiten, ohne dass eine zentrale Steuereinheit vorhanden ist. Die Gewichte

zwischen den Einheiten sind das primäre Mittel zur langfristigen Informationsspeicherung in Neuronalen Netzen. Das Aktualisieren der Gewichte ist die wesentliche Art und Weise, wie das Neuronale Netz neue Informationen lernt. (5)

2.2.1 Modell eines Neurons

Ein Künstliches Neuron hat als Eingabe eine Reihe von Werten, aus denen das Neuron einen einzelnen neuen Ausgabewert bildet. Alle Eingaben werden dazu mit Gewichten multipliziert und aufsummiert. (6) Das berechnete Ergebnis wird dann durch eine Aktivierungsfunktion an andere Neuronen weitergeleitet oder als Ergebnis ausgegeben (Sehe Abbildung 2). Die Aktivierungsfunktion regelt das Verhalten des Künstlichen Neurons. (5)

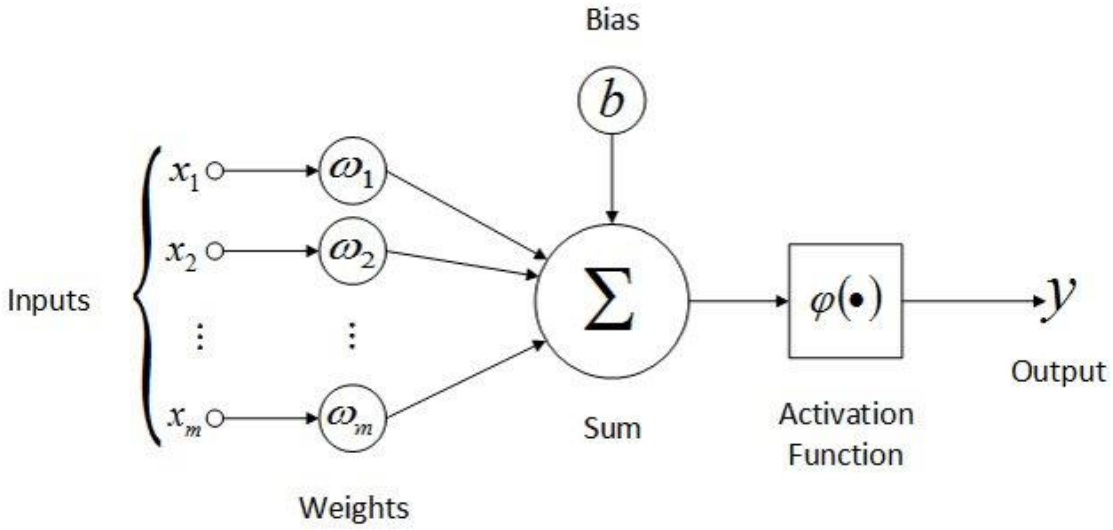

Abbildung 2 mathematisches Modell eines künstlichen Neurons (7)

2.2.2 Deep-Feedforward-Netze

Das bekannteste und am leichtesten zu verstehendem Neuronalem Netz ist das Deep-Feedforward-Netz. Es bestehen aus Knotenschichten, diese sind eine Eingabeschicht, eine oder mehrere versteckte Schichten und eine Ausgabeschicht. Es besteht nur aufsteigende Verbindungen von einer Schicht zur nächsten, aber keine Verbindungen zwischen Neuronen einer Schicht. Das Wort "Deep" weist darauf hin, dass die Netze mehrere versteckte Schichten haben. (2)

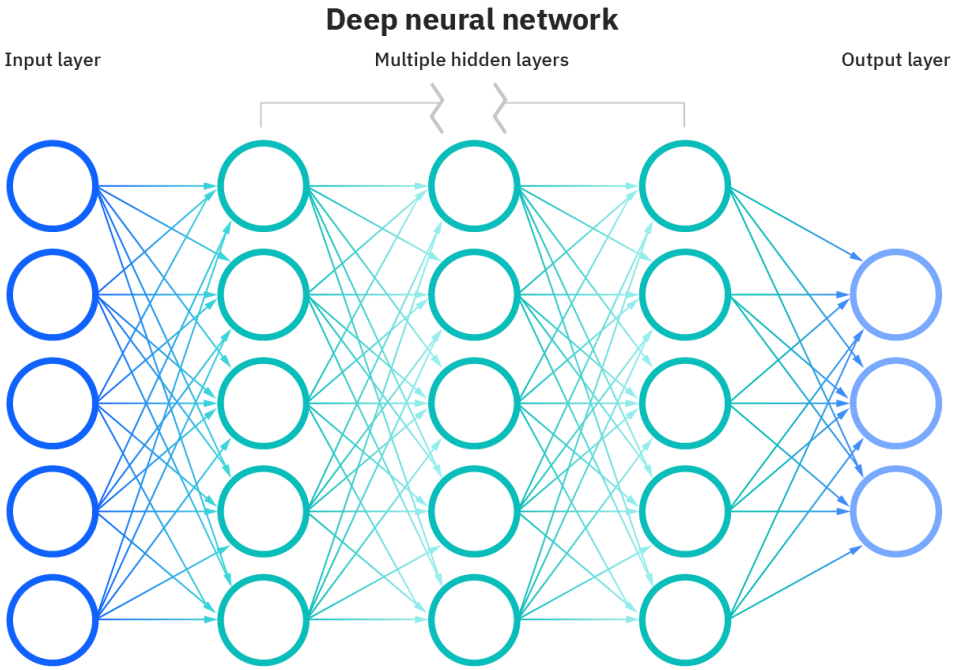

Abbildung 3: Aufbau eines Neuronalen Netzes (8)

2.3 Elemente eines evolutionären Algorithmus

Evolutionäre Algorithmen (EA) gehören zu den stochastischen[1] Suchmethoden, die von den Prinzipien der natürlichen biologischen Evolution inspiriert sind. Sie arbeiten gleichzeitig an einer Anzahl von möglichen Problemlösungen, die als Individuen repräsentiert werden. Das Prinzip „Überleben des Stärksten!“ wird auf diese Individuen angewendet, um immer bessere Individuen zu produzieren, die schließlich zu einer guten Problemlösung führen. (9)

[1] vom Zufall abhängig

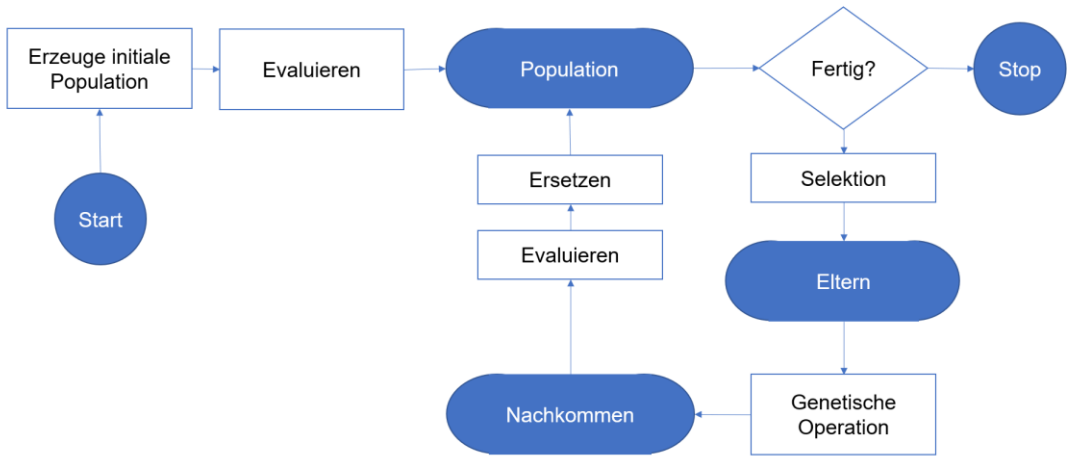

Abbildung 4: Ablauf eines Evolutionären Algorithmus (2)

In EA-Methoden werden aus einer zufällig oder mit Vorwissen erzeugten Anfangspopulation von Lösungen durch Fitnessevaluation, Selektion und Reproduktion über mehrere Generationen neue modifizierte Lösungen erstellt. (2)
In EA werden ständig neue Lösungen generiert. Diese werden gegen die anwendungsspezifische Fitnessfunktion evaluiert. Bestandteile guter Problemlösungen werden im Rahmen der Selektion und Vererbung weitergegeben, neu gemischt, durch zufällige Beeinflussungen modifiziert und dann werden sie neu evaluiert. Unter dem Einfluss der Selektion tendiert die Population also dazu, sich mit immer besseren Lösungen zu füllen, wobei sich die Suche immer mehr auf vielversprechende Regionen des Suchbereichs fokussieren wird. (10)

Es wird in EA zwischen zwei Verhalten unterschieden (10):

- *Exploitation*: Die Verwendung der in besten Lösungen beinhalteten Informationen über den Suchbereich.
- *Exploration*: Dabei wird durch die stochastisch geprägten Suchoperatoren sichergestellt, dass neue Bereiche des Suchraums weiter durchgesucht werden.

Für den erfolgreichen Einsatz von EA ist es erforderlich, eine gute Balance zwischen *Exploitation* und *Exploration* zu finden. Zu diesem Zweck gibt es vielfältige Freiheitsgrade bei der Prozessgestaltung und bei der Festlegung von Strategieparametern (z.B. Populationsgröße, Mutationswahrscheinlichkeit) (10)

2.3.1 Fitnessbestimmung

Unter dem Begriff ‚Fitness' wird das Bewertungsergebnis der Eigenschaften einer Problemlösung verstanden. Um dieses Ergebnis zu erhalten, wird eine Liste von Lösungen in

Bezug auf das Optimierungsziel bewertet. Anhand dieser Bewertung werden die gesuchten Lösungsmerkmale bestimmt. (2)
In EA wird nach Boersch u.a. zwischen drei Arten von Fitness unterschieden

- *Berechenbare Fitness*: Fitness wird als skalare Größe berechnet z.B. die Geschwindigkeit eines Autos
- *Interaktive Fitness*: Die Qualität eines Individuums wird von Menschen subjektiv bewertet z.B. durch Anschauung
- *Implizite Fitness*: Die Fitness wird als Überlebens- und Reproduktionswahrscheinlichkeit dargestellt z.B. Artificial Life (2)

2.3.2 Genetische Operatoren

Die Genetische Operatoren sind der Mechanismus, um eine Evolution von einer Generation zu einer anderen Generation zu erzeugen und mit denen einzelne Lösungskandidaten im Verlauf umgestellt werden. Diese Operatoren sind anwendungsspezifisch. (2)

2.3.3 Selektionsverfahren

Die Selektionsoperation in einem Genetischen Algorithmus (GA) dient dazu, Individuen (Eltern, vgl. Abbildung 4) für die Reproduktion auszuwählen. Die Elternauswahl ist sehr wichtig für die Konvergenzrate des GA, da gute Eltern bessere und passendere Lösungen erzeugen und liefern. (11)
Zu den bekanntesten Selektionsmethoden zählen folgenden:

- Rouletteselektion
- Rangbasierte Selektion
- Turnierselektion
- Truncation-Selektion

Zu den evolutionären Algorithmen gehören die folgenden Methoden nach Boersch u.a.:

- Evolutionäre Programmierung (EP)
- Evolutionsstrategien (ES)
- Genetische Algorithmen (GA)
- Genetische Programmieren (GP)

(2)

2.3.4 Genetische Algorithmen

Genetische Algorithmen wurden erstmals 1975 von Hohn Holland (Holland 75) als eine Variante von EA vorgestellt. Mögliche Problemlösungen werden als Bitvektoren fester Länge kodiert und einer Selektion, Mutation und einem Crossover unterzogen. (2) Die Problemlösungen werden in einer Art und Weise dargestellt, die leicht mit einem Rechensystem manipuliert werden kann. Diese Darstellung wird als Genotyp bezeichnet. Die Darstellung der Lösung wie in realen Situationen wird als Phänotyp bezeichnet. Bei einfachen Problemen sind der Phänotyp und der Genotyp identisch. In den meisten Fällen sind sie jedoch unterschiedlich.

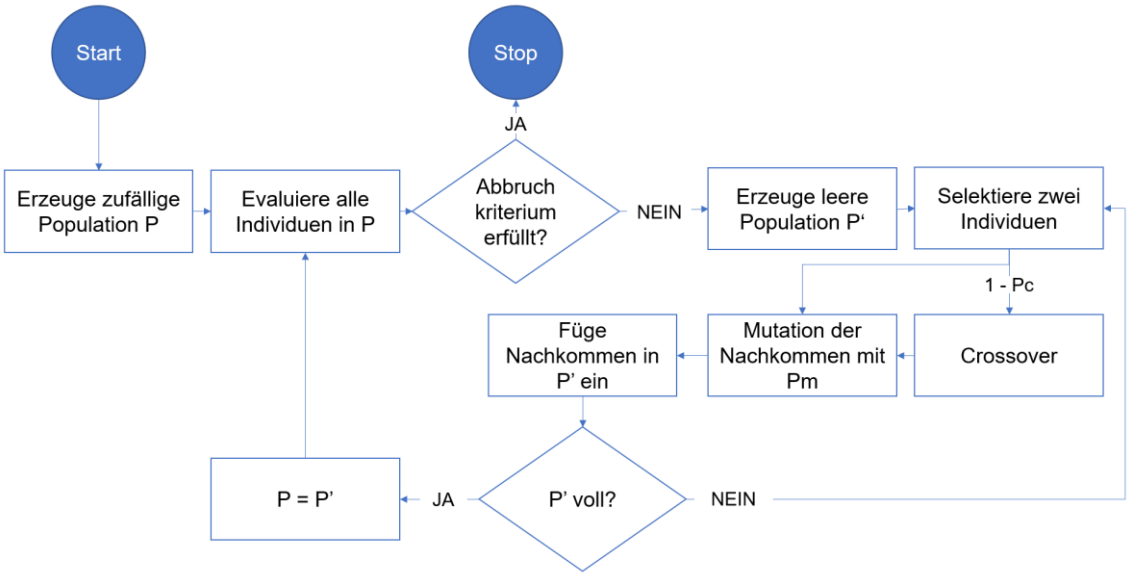

Abbildung 5: Simpler Genetischer Algorithmus (2)

aufgrund der Mutation entstehen zufallsbedingte Modifikationen an den Individuen. Diese Modifikationen (Mutationsschritte) sind in der Regel nur klein und werden mit einer geringen Häufigkeit (Mutationswahrscheinlichkeit) auf die Variablen der Individuen ausgeübt. (9) Die Mutation wird verwendet, um Vielfalt in die genetische Population einzuführen und aus dem lokalen Minimum auszulösen. (11)
Als Crossover ist ein Rekombinationsoperator zu verstehen, über dessen Mechanismus zwei oder mehrere Lösungen miteinander kombiniert werden. So entstehen beispielsweise die Nachkommen durch Neukombinationen aus zwei oder mehreren Elternindividuen. Dies erfolgt durch das Kombinieren der variablen Werte der Eltern. (9)

Die o.g. Elemente eines evolutionären Algorithmus bilden im Zusammenspiel mit Künstlichen Neuronalen Netzen eine wesentliche Grundlage für das Erstellen eines Neuroevolutionären Algorithmus, wie in Abschnitt 2.4 näher ausgeführt wird.

2.4 Neuroevolutionäre Algorithmen

In der letzten Zeit hat sich ein Großteil des maschinellen Lernens auf Deep Learning konzentriert. Wobei die Gewichte Neuronaler Netzwerke durch Varianten des stochastischen Gradientenverfahren trainiert werden. Ein alternativer Ansatz kommt aus dem Bereich der Neuroevolution. (12) Diese ist ein Teilgebiet innerhalb der Künstlichen Intelligenz (KI) und des maschinellen Lernens (ML), das sich mit der Entwicklung von Mitteln zur Evolution Neuronaler Netze durch evolutionäre Algorithmen beschäftigt. (13) Dieses Verfahren hat sich bei komplexen Reinforcement-Learning-Aufgaben als sehr vielversprechend erwiesen (14)

Es wird nach Stanley u.a. zwischen drei Verfahren des Künstliche Neuroevolution unterschieden:

1. Anpassung der Gewichte einer festen Netztopologie
2. Optimierung der Topologie des Künstlichen Neuronalen Netzes
3. Gleichzeitige Optimierung von Topologien und Gewichten von Neuronalen Netzen (z.B. NEAT) (14)

2.4.1 Klassische Neuroevolution

In traditionellen NE-Ansätzen wird eine EA verwendet für die Optimierung der Gewichte eines KNNs mit einer festen Topologie. Am Anfang wird eine feste Topologie für die zu entwickelnden Netze gewählt. Danach wird eine Population von Neuronalen Netzen initialisiert. Diese sind meistens Fully-Connected-Feed-Forward-Neuronale-Netze, wobei die Gewichte und Biases dieser Neuronalen Netze meistens mit zufälligen Dezimalzahlen initialisiert werden. Jedes KNN wird anhand einer Aufgabe evaluiert. Durch ein Selektionsverfahren werden die leistungsstärksten Netze ausgewählt. Diese werden vermehrt und reproduziert, um eine neue Population zu bilden. Dabei werden der Gewichtsraum (auch Genotyp genannt) der Neuronalen Netze durch das Crossover von Netzgewichtsvektoren und durch Mutation der gewichte einzelner Netze erforscht. Der Prozess der Bewertung und der Erzeugung einer neuen Population wird so lange wiederholt, bis sich die neue Population nicht mehr wesentlich verbessert. Das Ziel der Feste-Topologie-NE ist also die Optimierung der Verbindungsgewichte, die die Funktionalität eines Netzes bestimmen.

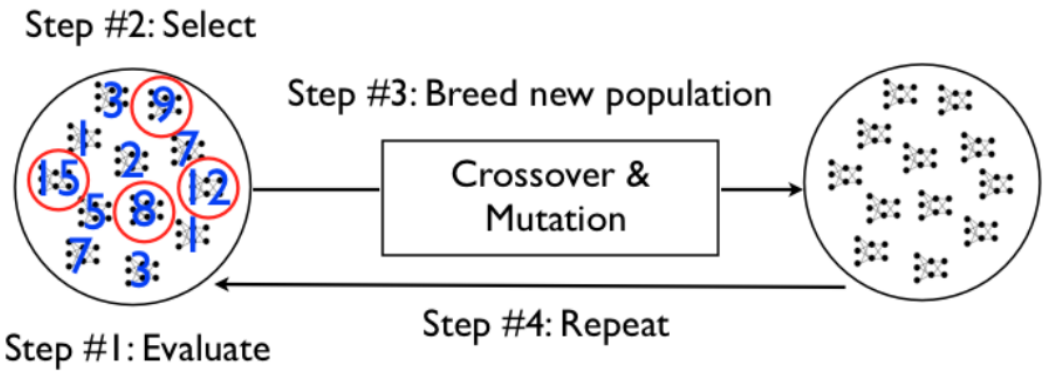

Abbildung 6: Die grundlegenden Schritte der Neuroevolution. (15)

2.4.2 Probleme der traditionellen Neuroevolution

Eine besondere Problematik der evolutionären Optimierung von KNN, ist das Competing-Conventions-Problem auch bekannt als das Permutationsproblem.
"Competing conventions" bedeutet, dass es mehr als eine Art und Weise gibt, eine Lösung für ein Gewichtsoptimierungsproblem für ein Neuronales Netz darzustellen. Wenn die Genotypen, die die gleiche Lösung darstellen, nicht die gleiche Kodierung haben, führt das Crossover häufig zu beschädigten Nachkommen. Abbildung 7 stellt das Problem für ein einfaches 3 Hidden-Unit-Netz dar. Die drei versteckten Neuronen A, B und C, können die gleiche allgemeine Lösung in 6 verschiedenen Varianten darstellen. Wenn sich eine dieser Varianten mit einer anderen kombiniert, gehen möglicherweise wichtige Informationen verloren. Zum Beispiel kann das Kombinieren von [A, B, C] und [C, B, A] zu [C, B, C] führen. Somit geht ein Drittel der Informationen, die beide Elternteile hatten, verloren. (14)

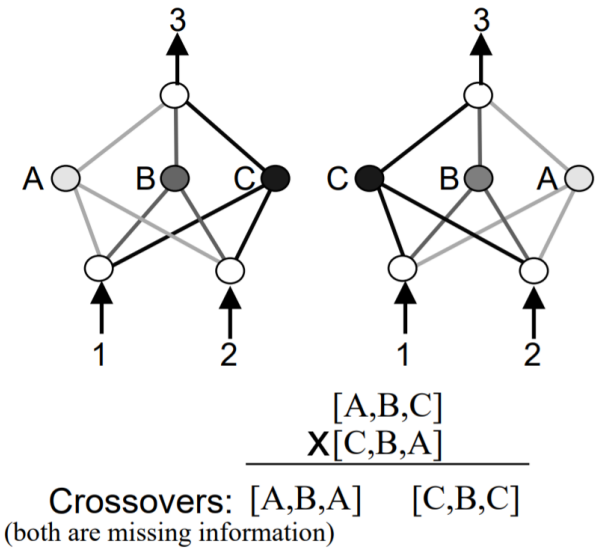

Abbildung 7: Beispiel für das Competing-Conventions-Problem (14)

Die Verbindungsgewichte sind jedoch nicht der einzige Aspekt von Neuronalen Netzen, der zu ihrem Verhalten beiträgt. Die Topologie oder Struktur von Neuronalen Neten hat auch einen Einfluss auf ihre Funktionalität. Obwohl viele NE-Systeme mit fester Topologie eine vollständig verbundene versteckte Schicht verwenden, ist die Entscheidung, wie viele versteckte Knoten benötigt werden, ein Versuch-und-Irrtum-Prozess. (14)
Die NEAT-Methode bietet Lösungen für die o.g. Problematik eines evolutionären Algorithmus, wie in Abschnitt 2.4.3 näher ausgeführt wird.

2.4.3 Gleichzeitige Optimierung von Topologien und Gewichten von Neuronalen Netzen

Eins der bekanntesten Neuroevolutionären Algorithmen ist NeuroEvolution of Augmenting Topologies (NEAT). Dieser ist in der Lage, sowohl die Struktur als auch die Hyper-Parameter eines KNNs zu generieren. NEAT kombiniert die übliche Suche nach den passenden Netzgewichten mit einer Komplexifizierung der Netzstruktur. Er beginnt mit einfachen Netzen und erweitert den Suchraum nur dann, wenn es erforderlich ist.

Nach Stanley u.a. hat der NEAT Algorithmus folgende Eigenschaften:

- Es bietet eine genetische Repräsentation, die es ermöglicht, unterschiedliche Topologien auf eine sinnvolle Art und Weise zu kombinieren (vgl. Abbildung 8). Damit wird auch das Competing-Conventions-Problem gelöst.
- Topologien werden im Laufe der Evolution minimiert, ohne dass eine speziell konstruierte Fitnessfunktion benötigt wird, die die Komplexität misst.
- Topologische Innovationen, die einige Generationen zur Optimierung benötigen, werden geschützt, damit sie nicht vorzeitig aus der Population verschwinden. (14)

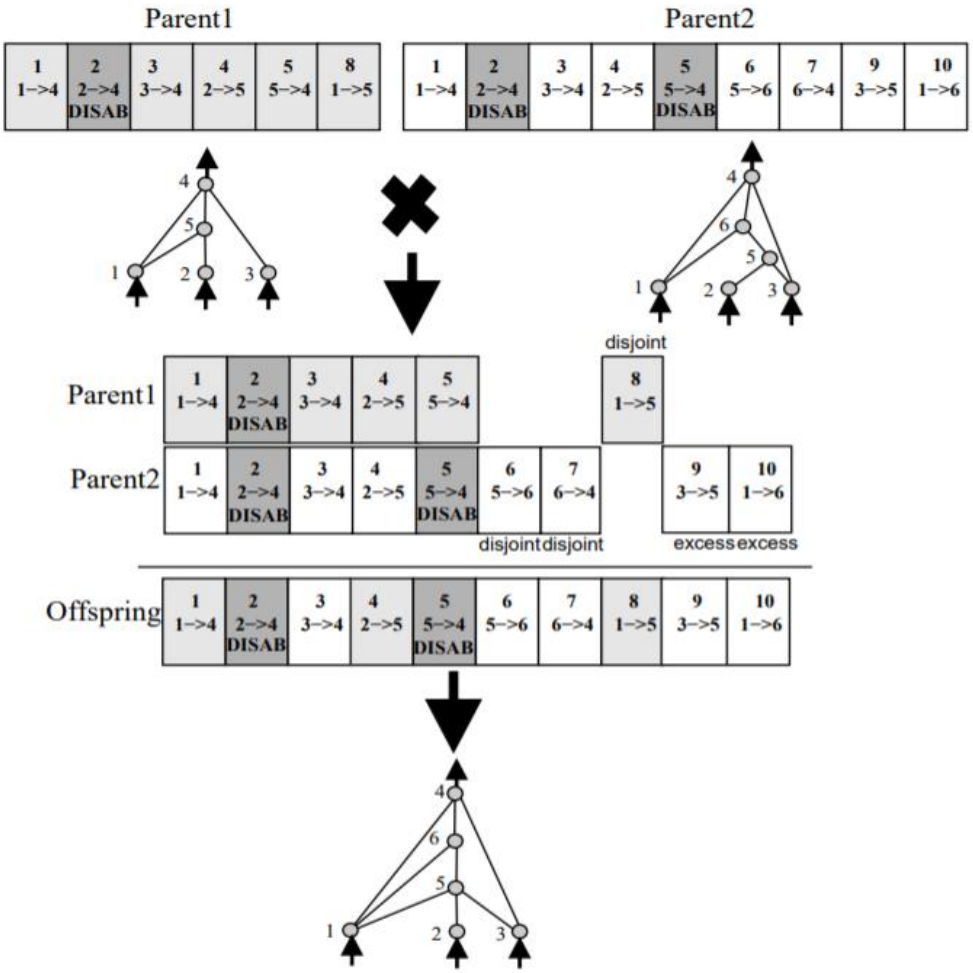

Abbildung 8: Matching up genomes for different network topologies using innovation numbers. (14)

Anhand dieser Eigenschaften übertrifft NEAT die Fixed-Topology-Methoden bei anspruchsvollen Reinforcement-Learning-Aufgaben, er ist aber komplizierter zu implementieren als traditionellen NE-Algorithmen. (15)

3 Methodik

In diesem Kapitel werden die die benötigten Tools und Programme, die in dieser Arbeit verwendet werden, dargestellt.

3.1 Unity

Die Unity Game Engine (kurz: „Unity) wurde von Unity Technologies entwickelt und sie ist ein leistungsfähiges und vielseitiges Werkzeug für die Entwicklung von 2D and 3D Spiele und anderer interaktiver 3D-Grafik-Anwendungen. Sie unterstützt die Sprachen C# und JavaScript und ist eine der meistgenutzten Spiele-Engines der Welt mit einem Marktanteil von 45 %.
Die Engine wurde auch von Branchen außerhalb der Videospielindustrie übernommen, z. B. von der Filmbranche, der Automobilindustrie, der Architektur, dem Maschinenbau und Bauwesen.

3.1.1 Wesentliche Vorteile des Programms Unity

Zu den allgemeinbekannten Vorteilen des Programms Unity gehören die folgenden:

- große Community
- der unterstützte Asset Store, in dem Grafiken und Ressourcen gekauft oder kostenlos heruntergeladen werden können.
- die Unterstützung von über 20 verschiedenen Zielplattformen für die Bereitstellung. (vgl. 3.1.2)
- der visuelle Editor
- Funktionen wie das vollständige und robuste Mecanim-Animationssystem, das aus Unity eines der besten Game Engines macht
- eine umfangreiche Dokumentation
- Built-in physics engines, um sicherzustellen, dass die Objekte korrekt beschleunigt werden und auf Kollisionen, Schwerkraft und verschiedene andere Kräfte reagieren. (16)

3.1.2 Unity Asset Store

Der Unity Asset Store bietet eine kontinuierlich wachsende Bibliothek mit kostenlosen und kommerziellen Assets, die sowohl von Unity Technologies als auch von Mitgliedern der Community erstellt wurden. Es steht eine große Auswahl an Assets zur Verfügung, die von Texturen, Modellen und Animationen bis hin zu ganzen Projektbeispielen, Tutorials und Extension Assets reicht. Der Zugriff auf die Assets erfolgt über eine einfache, in den Unity-Editor integrierte Schnittstelle, die das Herunterladen und den Import direkt in das eigene Projekt ermöglicht. Die Entwickler nutzen den Asset Store, da die Inhalte ihnen helfen, ihr eigenes Spiel oder ihre Anwendung zu verbessern und den Arbeitsaufwand für die Erstellung von Modellen oder Tools zu reduzieren. (17)

3.1.3 Wichtige Begriffe in Unity

In diesem Abschnitt sollen die grundlegenden Begriffe vom Unity geklärt werden. Diese sind wichtig für das Verstehen der Implementierung.

Szene Szenen enthalten die Objekte des Projekts. Sie können verwendet werden, um ein Hauptmenü, einzelne Ebenen und alles andere zu erstellen. (18)

GameObject Die Game-Objekten sind die grundlegenden Objekte in Unity. Sie machen an sich nicht viel, aber sie dienen als Container für Komponenten, die die Funktionalität implementieren. (19)

Komponente Komponente steuern das Verhalten der Game-Objekten, die mit ihnen verbunden sind.
Beispiele für Komponenten sind Materialien, physikalische Eigenschaften und Skripte.

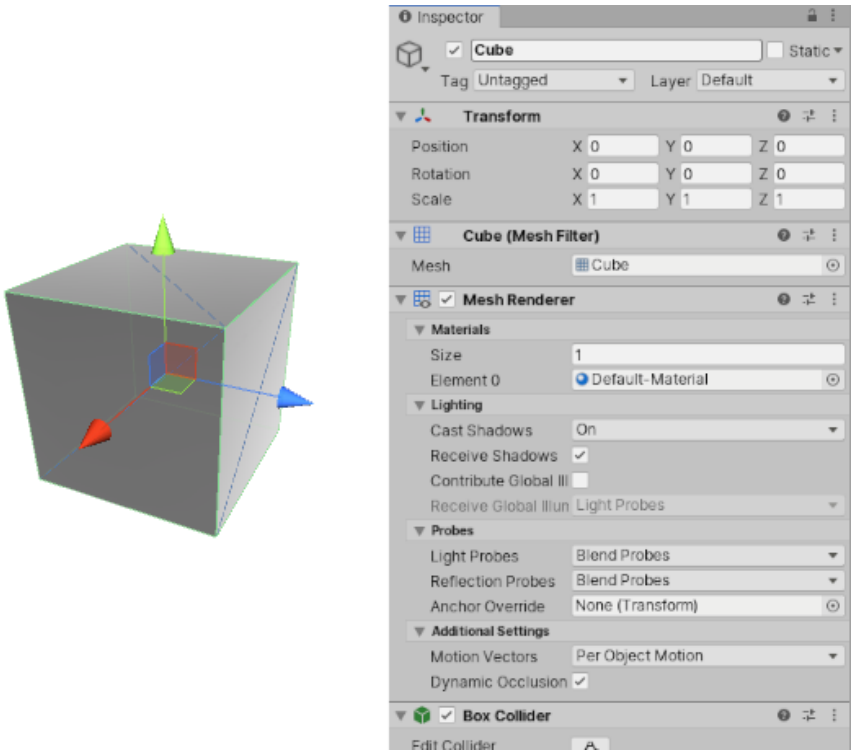

Abbildung 9: Ein grundlegendes Würfel-GameObject mit mehreren Komponenten (19)

Skripten Obwohl die in Unity eingebauten Komponenten sehr vielseitig sind, müssen die Entwickler häufig über das hinausgehen, um ihre eigenen Funktionen zu implementieren.
In Unity können mit Hilfe von Skripten eigene Komponenten erstellt werden. Diese ermöglichen es, Spielereignisse auszulösen, Komponenteneigenschaften im Laufe der Zeit zu ändern und auf Benutzereingaben zu reagieren. (20)

Rigidbodies sind Komponente, die GameObjects ermöglichen, unter der Kontrolle der Physik zu agieren. Der Rigidbody kann Kräfte und Drehmomente aufnehmen, um die Objekte auf realistische Weise zu bewegen. Jedes Spielobjekt muss einen Rigidbody enthalten, um von der Schwerkraft beeinflusst zu werden, unter zusätzlichen Kräften über Skripting zu agieren oder mit anderen Objekten über die NVIDIA PhysX Physik-Engine zu interagieren. (21)

Tabelle 1: Wichtige Eigenschaften von Rigidbodies und deren Funktionen (21)

Eigenschaft	***Funktion***
Mass	Die Masse des Objekts (standardmäßig in Kilogramm).
Drag	Wie stark der Luftwiderstand das Objekt bei

	der Bewegung von Kräften beeinflusst. 0 bedeutet keinen Luftwiderstand, und bei unendlich hört das Objekt sofort auf, sich zu bewegen.
Angular Drag	Wie stark der Luftwiderstand auf das Objekt wirkt, wenn es sich aus dem Drehmoment herausdreht. 0 bedeutet kein Luftwiderstand.
Use Gravity	Wenn aktiviert, wird das Objekt von der Schwerkraft beeinflusst.
Is Kinematic	Wenn diese Option aktiviert ist, wird das Objekt nicht von der Physik-Engine angesteuert und kann nur durch seine Transformation manipuliert werden.

Eine Kollision (engl. Collision) tritt auf, wenn die Physik-Engine erkennt, dass sich die Collider zweier GameObjects berühren oder überlappen, wenn mindestens eines eine Rigidbody-Komponente hat und in Bewegung ist.

Der Box Collider ist ein grundlegendes quaderförmiges Kollisionsprimitiv.

Der Wheel Collider ist ein spezieller Collider für Fahrzeuge mit Bodenkontakt. Er verfügt über eine integrierte Kollisionserkennung, Rad-Physik und ein auf Rutschen basierendes Reifenreibungsmodell. Er ist aber speziell für Fahrzeuge mit Rädern konzipiert. (22)

Tabelle 2: Eigenschaften der Wheel Collider und deren Funktionen (22)

Eigenschaft	Funktion
Mass	Die Masse des Rades.
Radius	Radius des Rades.
Wheel Damping Rate	This is a value of damping applied to a wheel.

3.1.4 Entwicklungsumgebung

Die Entwicklungsumgebung ist bekannten 3D-Animationsprogrammen sehr ähnlich. Die 3D-Szene wird in dem Hauptfenster visualisiert. Kameras und Szenen können mit Hilfe verschiedener Menüs angepasst werden. Teile der Szene lassen sich mit der Maus auswählen, skalieren, verschieben und rotieren. Einfache Objekte wie Lichtquellen oder grafische Primitive (Ebenen, Würfel, Kugeln) können direkt im Editor erzeugt werden. Komplexe Komponenten (sogenannte „Assets“) werden per Drag und Drop importiert, z.B. 3D-Modelle, Animationen, Texturen und Sounds, die in anderen Programmen erstellt wurden. Werden sie im Laufe der Produktion verändert, aktualisiert der Unity-Editor sie automatisch. Im „Game-View“ werden grafische Darstellung und Verhalten des Spiels simuliert. Unity ermöglicht anhand einer Exportfunktion die Erzeugung von ausführbaren Anwendungen. (vgl. Abbildung 10)

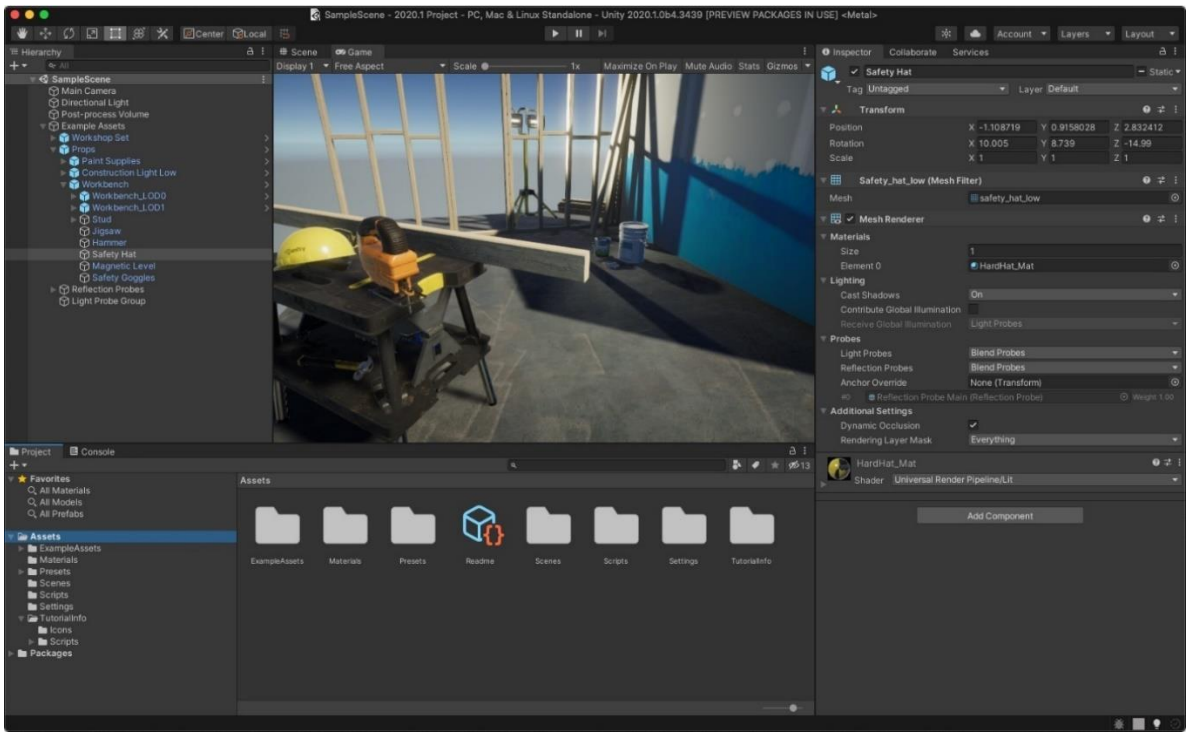

Abbildung 10: Die Unity-Entwicklungsumgebung (23)

3.2 CSharp

„C# (Aussprache „C Sharp“) ist eine moderne, objektorientierte und typsichere Programmiersprache.“ (24)

Mit C# können Entwickler sichere und robuste Applikationen erstellen, die im .NET-Framework laufen. C# stammt aus der C-Sprachfamilie und ist für Entwickler, die mit C, C++, Java und JavaScript Erfahrung haben, sofort vertraut. (24) Als integrierte Entwicklungsumgebung für C# wird meistens Visual Studio von Microsoft benutzt.

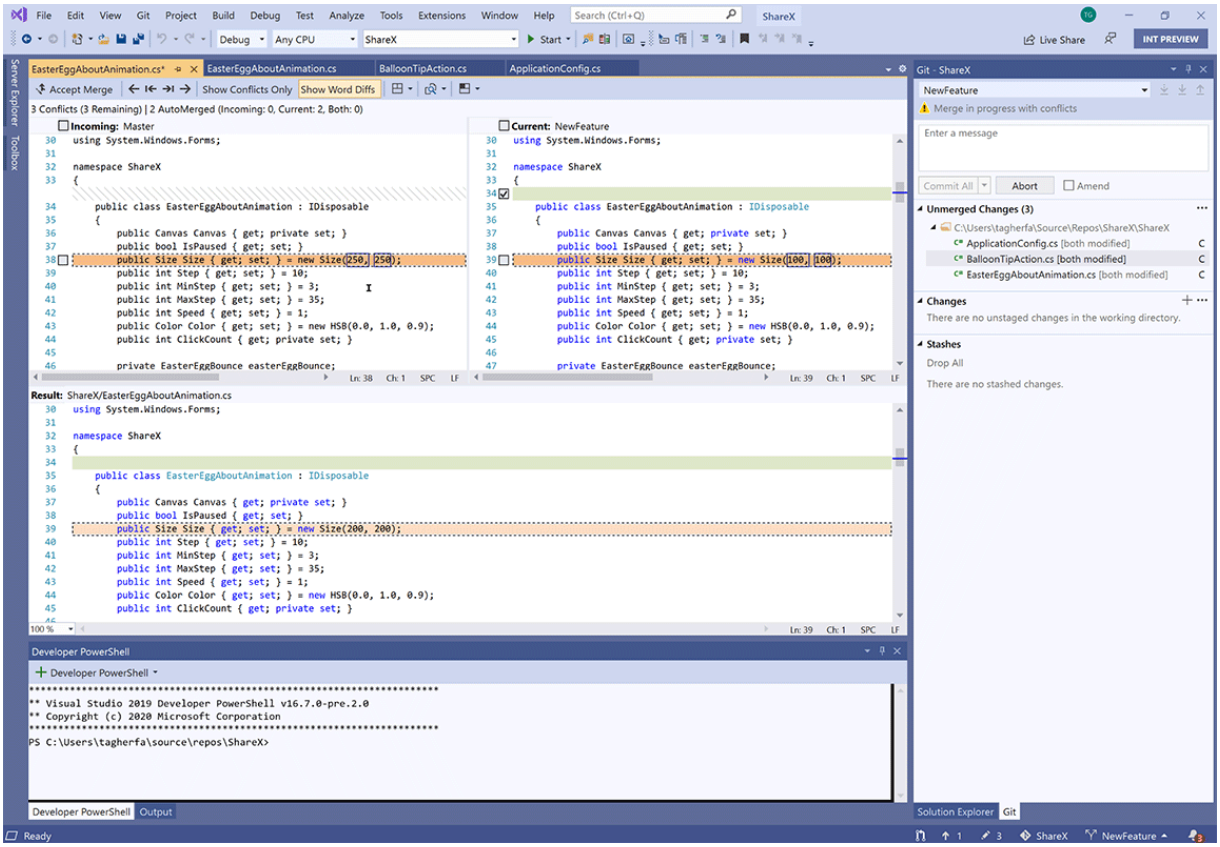

Abbildung 11: Die integrierte Entwicklungsumgebung Visual Studio (25)

4 Konzeption und Praktische Umsetzung

Dieses Kapitel ist vor allem für Fachleute relevant, die auf diesem Projekt aufbauen oder spezifische Design-Entscheidungen nachvollziehen möchten.

4.1 Konzeption für die zu entwickelnde Simulation

Das zu entwickelnde Programm soll als Windows-Executable nutzbar sein. Es soll den Benutzern ermöglichen, eine simulierte Umgebung von selbst fahrenden Fahrzeugen zu steuern. Um das zu erreichen, soll das Programm die folgenden Elemente enthalten:

- Neuronales Netz
- Ein Lernalgorithmus
- Visuelle Simulation in 3D mit Unity
- Sensoren: Eingabedaten des neuronalen Netzes
- Steuerung des Fahrzeugs: Ausgabedaten des Netzes
- Kameras: für die Betrachtung der Simulation
- SimulationController: steuert alle Abläufe und Algorithmen
- grafische Benutzeroberfläche

Jedes Fahrzeug soll ein Feed-Forward Neuronales Netz enthalten. Eine Klasse in C# mit dem Namen "NeuralNetwork" soll erstellt werden. Diese soll alle benötigten Attribute des in Schichten organisierten Neuronalen Netzes enthalten. Sie wird auch einen Konstruktor und die benötigten Methoden für die Durchführung genetischer Operationen haben.
Anstatt Backpropagation für das Lernen zu verwenden, soll Bestärkendes Lernen verwendet. Ein Neuroevolutionärer Algorithmus soll implementiert werden, der auf alle Autos angewendet werden soll. Dieser ist wesentlich einfacher zu programmieren und erfordert im Vergleich zu anderen Machine Learning Algorithmen nur geringe mathematische Kenntnisse.

Der zu entwickelnde Neuroevolutionärer Algorithmus kann in verschiedene Abschnitte unterteilt werden:

- Zufällige Initialisierung der Neuronalen Netze

- Auswahl der besten Fahrzeuge in Abhängigkeit von einer Fitnessfunktion
- Kombination des Genotyps der Eltern
- Mutation des neuen Genotyps

Zusätzlich zu den Algorithmen sollten alle benötigten GameObjekte wie die Strecken und das Fahrzeug erstellt werden. Es sollen auch die benötigten Komponenten für diese GameObjekte hinzugefügt, um deren Verhalten zu steuern.
Jedes Fahrzeugobjekt soll Sensoren habe. Die Werte dieser Sensoren sollen die Eingangsdaten des neuronalen Netzes für jedes Fahrzeug sein. Ein Feed-Forward-Algorithmus soll angewendet werden, um die Ausgangsdaten des neuronalen Netzes zu erhalten. Diese Ausgänge dienen zur Steuerung des Fahrzeugs. Schließlich sollen alle zuvor erstellten Elemente miteinander verbunden werden und miteinander kommunizieren. Es soll ein GameObject mit dem Namen SimulationController erstellt werden. Dieses soll für die Steuerung aller Objekte und Algorithmen in der Simulation zuständig sein. Die praktische Umsetzung dieses Konzeptes erfolgt im nächsten Abschnitt.

4.2 Entwicklung und Aufbau der Simulation

Unity-Spiele sind vor allem auf Szenen aufgebaut. Das Projekt enthält nur eine Main-Szene, die alle GameObjects in der Simulation enthält. Im Folgenden wird die Implementierung der wesentlichen Teile der Main-Szene näher betrachtet.

4.2.1 Simulation der Strecken

Für die Erstellung von Strecken wird das Asset Modular Track (26) aus dem Asset Store heruntergeladen und verwendet. Dabei wird eine Strecke als die vom autonomen Fahrzeug zurückzulegende Route definiert. Jede Strecke wird erstellt durch das Klonen, Drehen, Skalieren und Kombinieren der modularen Teile. Diese haben ein *Mesh-collider-Component* zur Erkennung von Kollisionen mit den Agenten. Durch diese Methode werden die folgenden drei Strecken erstellt. (vgl. Abbildung 12, Abbildung 13 und Abbildung 14)

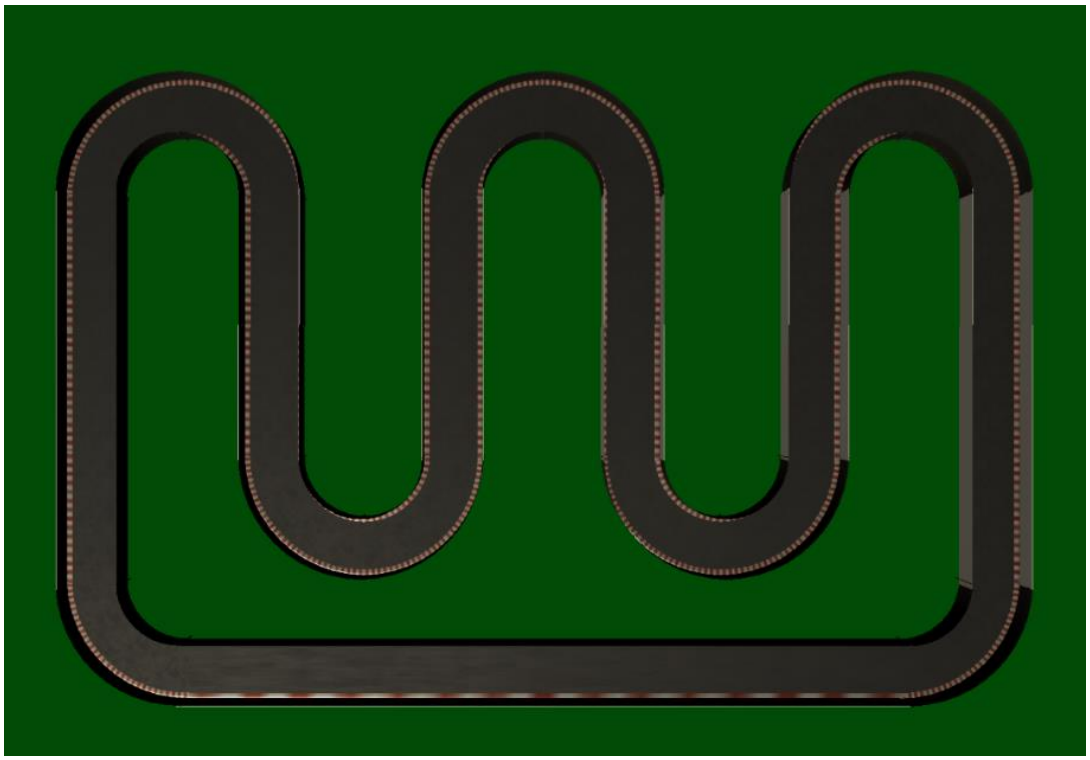

Abbildung 12: Strecke 1

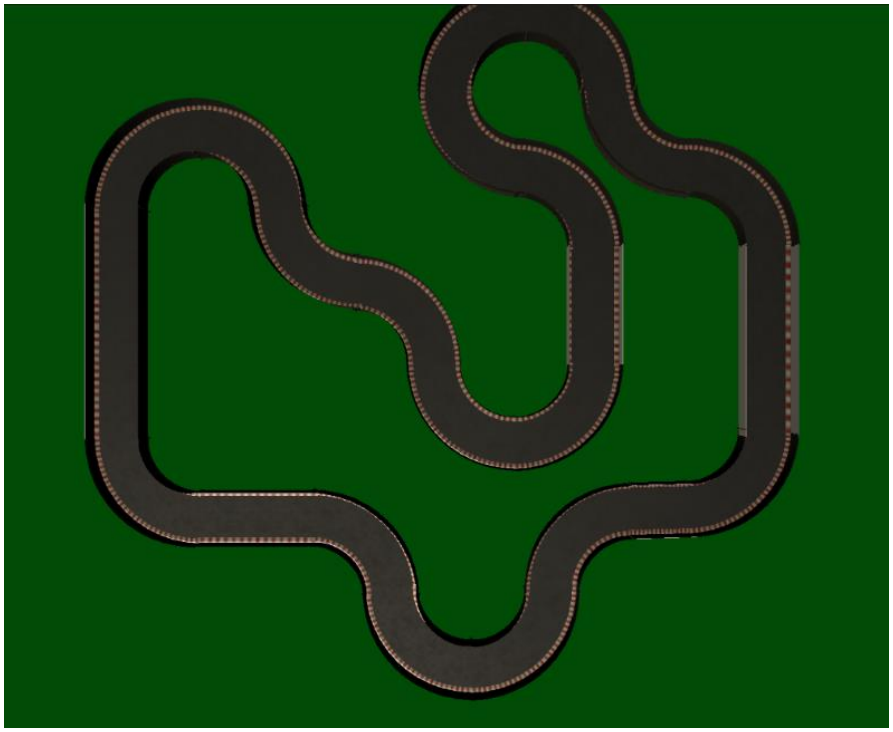

Abbildung 13: Strecke 2

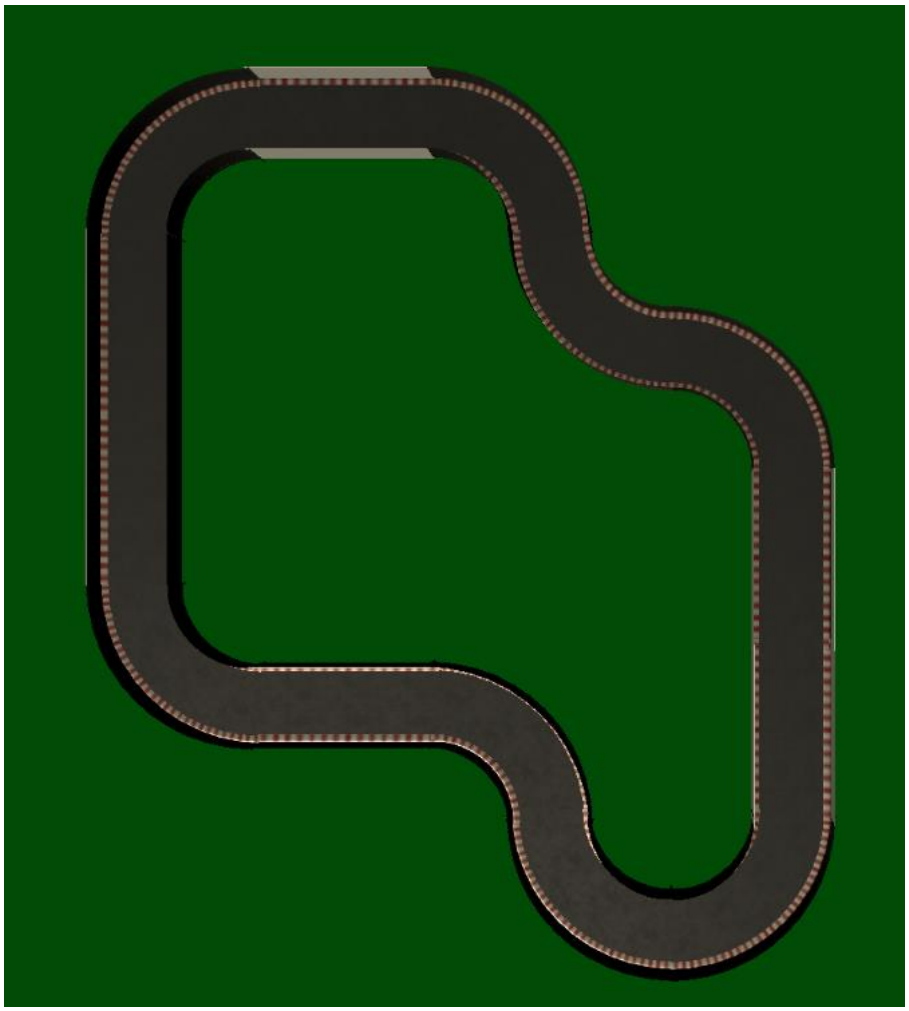

Abbildung 14: Strecke 3

4.2.2 Erstellen des Fahrzeugagenten

Für die Erstellung des Fahrzeugagenten wird das Asset ARCADE: FREE Racing Car (27) aus dem Asset Store importiert und verwendet. Dies ist ein von amerikanischen Designs inspirierter Rennwagen Asset (vgl. Abbildung 15). Räder sind in diesem Asset getrennte Meshes[2], so dass sie animiert werden können.

[2] Meshes enthalten Vertex-Daten (Positionen, TexCoords usw.) und "Flächen"-Daten, wobei Flächen meist in der Form von Dreiecken vorliegen.

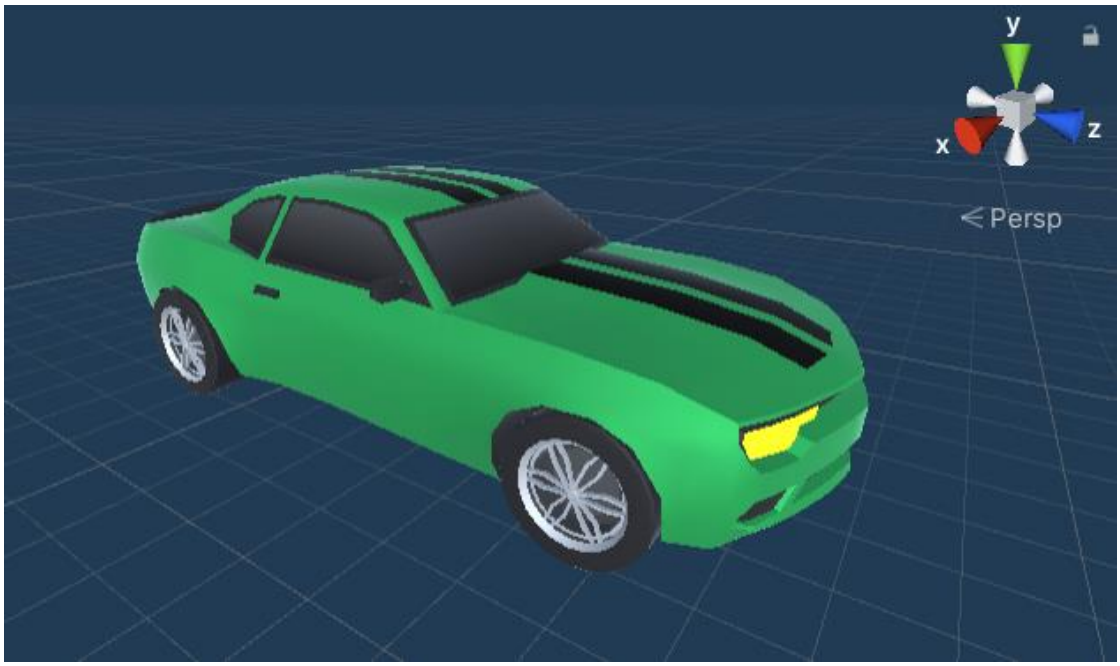

Abbildung 15: Fahrzeug 3D-Modell

Dem Fahrzeug-game-Object werden verschiedene Komponenten hinzugefügt (siehe Abbildung 16). Diese dienen der Steuerung des Fahrzeugs. Im Folgenden werden diese Komponenten näher betrachtet.

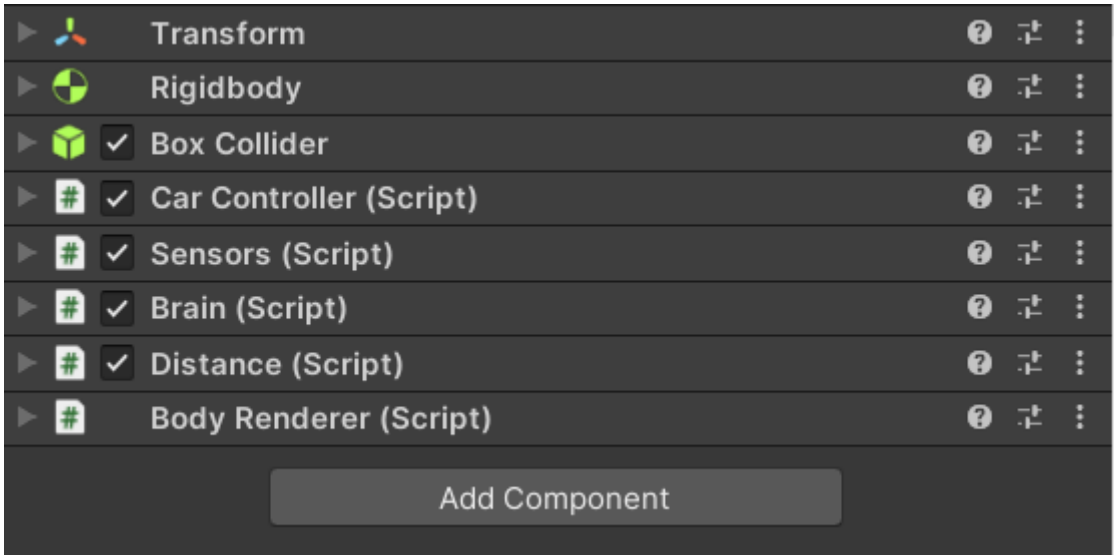

Abbildung 16: Komponente des Fahrzeug-game-Objectes

Rigidbody

Diese Komponente erlaubt das Fahrzeug unter der Kontrolle der Physik zu agieren. Die Eingabewerte für die Komponente werden in der Abbildung 17 dargestellt.

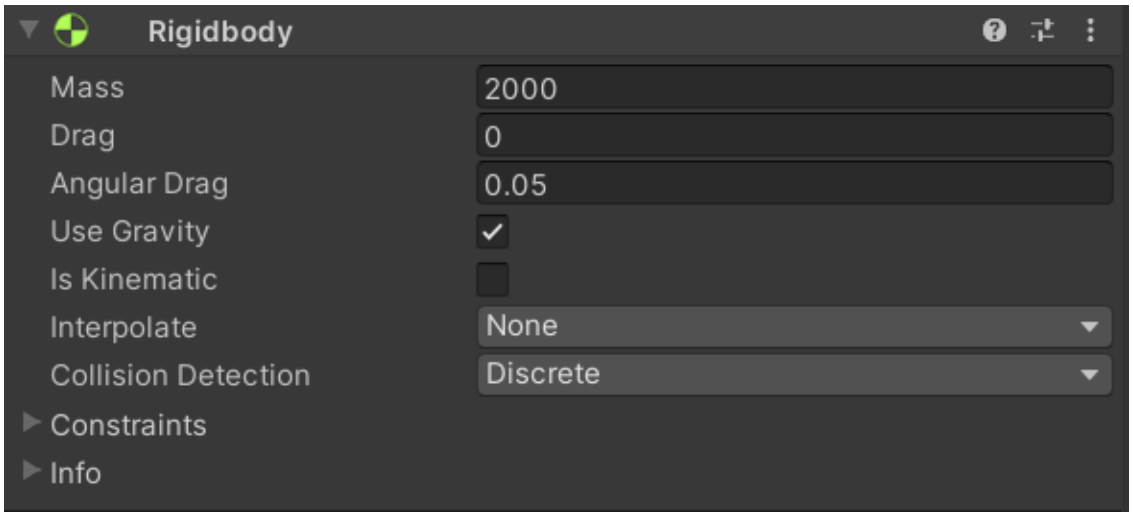

Abbildung 17: Die Eingabewerte der Rigidbody-Komponente

Box Collider

Dieser wird benutzt, um Kollisionen mit den anderen GameObjects zu erkennen.

Car Controller (Skript)

Die Aufgabe dieses Scripts ist die Steuerung der Lenkung des Fahrzeugs und der auf das Fahrzeug ausgeübten Kräfte, z. B. Motorleistung (siehe Listing 1).

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// this class controls the physical forces on the car
/// </summary>
public class CarController : MonoBehaviour
{
    private const string HORIZONTAL = "Horizontal";
    private const string VERTICAL = "Vertical";

    public float HorizontalInput;
    public float VerticalInput;
    private float currentSteerAngle;
    private float Currentbreakforce { get; set; } = 1000;

    [SerializeField] private bool isBreaking;
    [SerializeField] private readonly float motorForce = 1000;
    [SerializeField] private float breakForce;
    [SerializeField] private readonly float  maxSteeringAngle = 30;

    [SerializeField] private WheelCollider frontLeftWheelCollider;
    [SerializeField] private WheelCollider frontRightWheelCollider;
    [SerializeField] private WheelCollider rearLeftWheelCollider;
    [SerializeField] private WheelCollider rearRightWheelCollider;

    [SerializeField] private Transform frontLeftWheelTransform;
```

```
    [SerializeField] private Transform frontRightWheelTransform;
    [SerializeField] private Transform rearLeftWheelTransform;
    [SerializeField] private Transform rearRightWheelTransform;

    private void FixedUpdate()
    {
        HandleMotor();
        HandleSteering();
        UpdateWheels();
    }

    private void UpdateWheels()
    {
        UpdateSingleWheel(frontLeftWheelCollider, frontLeftWheelTransform);
        UpdateSingleWheel(frontRightWheelCollider, frontRightWheelTransform);
        UpdateSingleWheel(rearLeftWheelCollider, rearLeftWheelTransform);
        UpdateSingleWheel(rearRightWheelCollider, rearRightWheelTransform);
    }

    private void UpdateSingleWheel(WheelCollider wheelCollider
    , Transform wheelTransform)
    {
        Vector3 pos;
        Quaternion rot;

        wheelCollider.GetWorldPose(out pos, out rot);
        wheelTransform.rotation = rot;
        wheelTransform.position = pos;
    }

    private void HandleSteering()
    {
        currentSteerAngle = maxSteeringAngle * HorizontalInput;
        frontLeftWheelCollider.steerAngle = currentSteerAngle;
        frontRightWheelCollider.steerAngle = currentSteerAngle;
    }

    public float GetMotorForce()
    {
        return frontLeftWheelCollider.motorTorque;
    }

    private void HandleMotor()
    {
        if (!gameObject.GetComponent<Brain>().Surviving) return;

        frontLeftWheelCollider.motorTorque = VerticalInput * motorForce;
        frontRightWheelCollider.motorTorque = VerticalInput * motorForce;
        breakForce = isBreaking ? breakForce : 0f;

        if (isBreaking)
        {
            ApplyBreaking();
        }
        else
        {
            frontRightWheelCollider.brakeTorque = 0;
            frontLeftWheelCollider.brakeTorque = 0;
            rearLeftWheelCollider.brakeTorque = 0;
            rearRightWheelCollider.brakeTorque = 0;
        }
    }
```

```
    private void ApplyBreaking()
    {
        frontRightWheelCollider.brakeTorque = Currentbreakforce;
        frontLeftWheelCollider.brakeTorque = Currentbreakforce;
        rearLeftWheelCollider.brakeTorque = Currentbreakforce;
        rearRightWheelCollider.brakeTorque = Currentbreakforce;
    }

    /// <summary>
    /// control the movement of the car
    /// </summary>
    /// <param name="input">input[0] control the steering
    /// , input[1] the force applied on the motor</param>
    public void Control(float[] input)
    {

        HorizontalInput = input[0];
        VerticalInput = input[1] > 0 ? input[1] : 0 ;
    }

    /// <summary>
    /// Cancel all the forces applied to the vehicle
    /// </summary>
    public void RemoveAllForces()
    {
        GetComponent<Rigidbody>().velocity = Vector3.zero;
        GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
    }


}
```

Listing 1: Die Steuerung der Lenkung des Fahrzeugs und der auf das Fahrzeug ausgeübten Kräfte (28)

Sensors (Skript)

Dieses Skript ist für das Lesen der Sensoren-Werte des Fahrzeugs zuständig. Jedes Fahrzeug hat drei Sensoren, die den Abstand zum Zaun erkennen. Jeder Sensor gibt den Abstand zum ersten Objekt aus. Die Ausgabe ist ein Wert von [0-1]. Wenn der Sensor kein Objekt erkannt hat, wird der Wert 1 zurückgegeben.

```
using UnityEngine;

/// <summary>
/// Each car will have three sensors.
/// Each sensor will raycast the distance to the first object that could be track.
/// </summary>
public class Sensors : MonoBehaviour
{
    //starting Point of the sensor
    [SerializeField] private float frontSensorStartPoint = 2.4f;
    //the Length of the sensor
    [SerializeField] private float sensorLength = 10;
    //the horizontal distance between the sensor and the road
```

```
    [SerializeField] private float sensorHeight = 0.5f;
    //the distance between the left or right sensor and the middle sensor
    [SerializeField] private float sensorsDistance = 1;
    //the angle between the sensors in grad
    [SerializeField] private float sensorsAngle = 30;

    // Update is called once per frame
    void Update()
    {
       ReadFrontSensor();
       ReadRightSensor();
       ReadLeftSensor();
    }

    /// <summary>
    /// The output of the sensor will be
    /// the max distance of it divided by the collision point distance from the car.
    /// </summary>
    /// <param name="angle">the degree of the sensor</param>
    /// <returns>a value from [0-1].
    /// If the sensor haven't detected any object it will return 1.</returns>
    public float ReadSensor(float angle)
    {
        Vector3 pos;
        RaycastHit hit;

        pos = transform.position;
        pos += Quaternion
            .Euler(0f, angle, 0f) * transform.forward * frontSensorStartPoint;
        pos += (Vector3.up * sensorHeight);

        // Does the ray intersect any objects excluding the player layer
        if (Physics.Raycast(transform.position + (Vector3.up * sensorHeight)
            , Quaternion.Euler(0f, angle, 0f) *  transform.forward
            , out hit
            , sensorLength
            , Physics.IgnoreRaycastLayer)
            )
        {
            Debug.DrawRay(pos
                , Quaternion.Euler(0f, angle, 0f)
                * transform.forward * sensorLength
                , Color.red);
            return (Vector3.Distance(pos, hit.point)
                + frontSensorStartPoint) / sensorLength;
        }
            Debug.DrawRay(pos
                , Quaternion.Euler(0f, angle, 0f)
                *  transform.forward * sensorLength
                , Color.blue);
            return 1;

    }

    /// <returns>a value from [0-1].
    /// If the sensor haven't detected any object it will return 1.</returns>
    public float ReadFrontSensor(){
        return ReadSensor(1f);
    }

    /// <returns>a value from [0-1].
    /// If the sensor haven't detected any object it will return 1.</returns>
```

```
    public float ReadRightSensor()
    {
        return ReadSensor(sensorsAngle);
    }

    /// <returns>a value from [0-1].
    /// If the sensor haven't detected any object it will return 1.</returns>
    public float ReadLeftSensor()
    {
        return ReadSensor(sensorsAngle * -1);
    }

    /// <returns>the value of all the front Sensors</returns>
    public float[] GetValues()
    {
        return new float[] {
            ReadFrontSensor()
            , ReadRightSensor()
            , ReadLeftSensor() };
    }

}
```

Listing 2: Das Lesen der Sensoren-Werte des Fahrzeugs

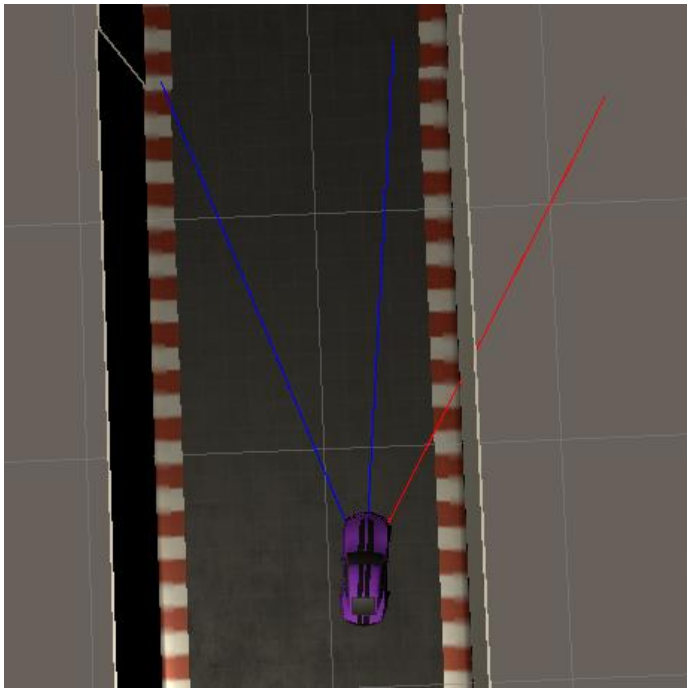

Abbildung 18: die Sensoren des Fahrzeuges

Brain (Skript)

Für das Fahrzeug wird auch ein Brain-script-Component hinzugefügt. Das Skript repräsentiert das KI-Component des Fahrzeugs. Es enthält eine Deklaration des Neuronalen Netzes, das Fitnessfunktion und eine Operation für die Kollisionserkennung.

Distance (Skript)

Das Skript dient zur Berechnung der Gesamtstrecke, die das Auto gefahren ist. Der berechnete Wert wird in der Fitnessfunktion benötigt.

```
using UnityEngine;

public class Distance : MonoBehaviour
{
    //the total distance the vehicle moved
    private float totalDistance;
    //last position of the vehicle
    private Vector3 lastPosition;
    private float speed;
    private float speedPerSec;

    public float SpeedPerSec {
        get { return speedPerSec; }
    }

    // Start is called before the first frame update
    void Start()
    {
        lastPosition = transform.position;
        totalDistance = 0;
    }

    void Update()
    {
        totalDistance += Vector3.Distance(lastPosition, transform.position);
        speedPerSec = Vector3.Distance(lastPosition, transform.position)
         / Time.deltaTime;
        speed = Vector3.Distance(lastPosition, transform.position);
        lastPosition = transform.position;

        if (this.speedPerSec == 0)
        {
            GetComponent<Brain>().Surviving = false;
            GetComponent<BodyRenderer>().Hide();
        }
    }

    public void Reset()
    {
        totalDistance = 0;
    }

    public float GetDistance()
    {
        return this.totalDistance;
    }

}
```

Listing 3: Berechnung der Gesamtstrecke

Body Renderer (Skript)

Dieses Skript ermöglicht es die Farbe des Fahrzeugs während der Evolution zu ändern.

```
using System.Linq;
using UnityEngine;

/// <summary>
/// change the color of the car
/// </summary>
public class BodyRenderer : MonoBehaviour
{
    public GameObject body;
    public Color color;
    const int x = 0;
    const int y = 33;
    const int width = 129;
    const int height = 192 - 33;

 /// <summary>
 /// change the color of the vehicle
 /// </summary>
 /// <param name="color">the new color</param>
    public void SetColor(Color color)
    {
        this.color = color;
        var oldTexture =
        (Texture2D)body.GetComponent<Renderer>().material.mainTexture;
        var newTexture =
        new Texture2D(oldTexture.width, oldTexture.height);
        var colors = Enumerable.Repeat(color, width * height).ToArray();
        newTexture.SetPixels(0, 0
            , oldTexture.width, oldTexture.height
            , oldTexture.GetPixels(0, 0
            , oldTexture.width
            , oldTexture.height));
        newTexture.SetPixels(x, y, width, height, colors);
        newTexture.Apply();
        body.GetComponent<Renderer>().material.mainTexture = newTexture;
    }

    /// <summary>
    /// change the color of the vehicle
    /// </summary>
    /// <param name="color">
    /// the new color value as integer
    /// </param>
    public void SetColor(int color)
    {
        Color c;
        float R, G, B;
        R = color & 255;
        G = (color >> 8) & 255;
        B = (color >> 16) & 255;
        R = R / 256;
        G = G / 256;
        B = B / 256;
        c = new Color(R, G, B);
        SetColor(c);
    }

    public Color GetColor()
```

```
    {
        return this.color;
    }

    public void Hide()
    {
        foreach (MeshRenderer mr
            in gameObject.GetComponentsInChildren<MeshRenderer>())
        {
            mr.enabled = false;
        }
    }

    public void Show()
        {
            foreach (MeshRenderer mr
            in gameObject.GetComponentsInChildren<MeshRenderer>())
            {
                mr.enabled = true;
            }
        }
}
```

Listing 4: Bearbeitung der Farbe des Fahrzeugs

4.2.3 Steuerung der Simulation

SimulationController

Das GameObject enthält Skripten für die Steuerung der Simulation und der EA.

AIController (Skript)

Dieses Skript dient zur Ausführung des Neuroevolutionären Algorithmus. Es dient auch zur Speicherung der dafür benötigten Werte.

SimulationController (Skript)

Dieses Skript hat folgende Aufgaben:

- Enthält Operationen für das Starten, das Pausieren und das Stoppen der Simulation.
- Der Auswahl der Strecke
- Änderung der Geschwindigkeit der Simulation
- Versteckung der zerstörten Fahrzeuge

4.2.4 Beobachtung der Simulation anhand von Kameras

Die Simulation soll anhand von Kameras beobachtet werden. Deswegen werden zwei Kamera-Objekte aufgestellt.

Folgende Kameras werden aufgestellt:

- Eine MainCamera, die eine statische Position hat und dem Benutzer eine Sicht auf der simulierten Welt ermöglicht.
- Eine FollowCamera, die ein Follow-Kamera-Skript-Component hat. Das Skript ermöglicht der Kamera, das Fahrzeug mit dem besten Fitnesswert zu verfolgen.

```
using UnityEngine;
/// <summary>
/// this script control the camera to chase
/// the fittest not destroyed vehicle
/// </summary>
public class CameraFollow : MonoBehaviour
{

    [SerializeField] private Vector3 offset = new Vector3(0f,3f,-6f);
    [SerializeField] private Transform target; //the target vehicle to follow
    [SerializeField] private float translateSpeed = 10;
    [SerializeField] private float rotationSpeed = 10;

    /// <summary>
    /// change the position of the followcamera
    /// object each time the fittest vehicle move
    /// </summary>
    private void FixedUpdate()
    {

        if (AIController.cars == null
            | AIController.cars.Count == 0) return;
        Transform newTarget = AIController.cars[0].transform;
        Brain bestBrain = newTarget.gameObject
            .GetComponent<Brain>();
        for(int i = 0; i < AIController.cars.Count; i++)
        {
            Brain b = AIController.cars[i].GetComponent<Brain>();
            if ((b.GetFitness() > bestBrain.GetFitness() & b.Surviving)
                | !bestBrain.Surviving)
            {
                newTarget = AIController.cars[i].transform;
                bestBrain = AIController.cars[i].GetComponent<Brain>();
            }
        }
        this.target = newTarget;

        HandleTranslation();
        HandleRotation();
    }

    /// <summary>
    /// handling the rotation of the camera
    /// </summary>
    private void HandleRotation()
    {
        var targetPosition = target.TransformPoint(offset);
        transform.position = Vector3
        .Lerp(transform.position
        , targetPosition, translateSpeed * Time.deltaTime);
    }

    /// <summary>
```

```
    /// handling the translation of the camera
    /// </summary>
    private void HandleTranslation()
    {
        var direction =
        target.position - transform.position;
        var rotation = Quaternion
            .LookRotation(direction, Vector3.up);
        transform.rotation =
        Quaternion.Lerp(transform.rotation, rotation
        , rotationSpeed * Time.deltaTime);
    }
}
```

Listing 5: Die Follow-Kamera-Skript-Component (28)

4.2.5 Erstellung der Benutzeroberfläche

Für die Erstellung der Benutzeroberfläche wird das Asset UI Widgets (28) aus dem AssetStore importiert und verwendet. Das Asset ist eine Sammlung von nützlichen Basis-UI-Skripten mit Beispielen für ihre Verwendung. Anhand dieses Assets werden vier Fenster erstellt. Diese liefern Informationen über den aktuellen Zustand der Simulation und ermöglichen der Benutzer, den Algorithmus zu steuern. Diese Fenster werden in diesem Abschnitt näher betrachtet.

Informationsfenster

Dieses Fenster zeigt Informationen über den aktuellen Status der Evolution, wie z.B. die Populationsgröße, die durchschnittliche Fitness (siehe Abbildung 19). Die gezeigten Informationen werden ständig anhand einer Update-Methode aktualisiert (Listing 6).

Abbildung 19: Informationsfenster

```
/// <summary>
/// Update the information in the Information window
/// </summary>
public void UpdateInformation()
{
     generation.GetComponent<Text>().text = "Generation: " +
     aIController.Generation;
     populationSize.GetComponent<Text>().text = "Population Size: " +
     aIController.PopulationSize;
     currentAlive.GetComponent<Text>().text = "Current Alive: " +
     aIController.Alive;
     bestFitnessEver.GetComponent<Text>().text = "Best Fitness Ever: " +
     System.Math.Round(aIController.BestFitnessEver,2);
     meanFitnessLastGen.GetComponent<Text>().text = "Mean Fitness: " +
     System.Math.Round(aIController.MeanFitness,2);
     avgFitnessLastGen.GetComponent<Text>().text = "Average Fitness: " +
     System.Math.Round(aIController.AverageFitness,2);
     roundTime.GetComponent<Text>().text = "Round Time: " +
     System.Math.Round(Timer.Time);
}
```

Listing 6: Die Methode UpdateInformation

Steuerungsfenster

Das Fenster erlaubt dem Benutzer anhand des ApplicationController-Scripts die Simulation zu steuern. Anhand dieses Fensters kann beispielsweise die Simulation durchgeführt, pausiert und gestoppt werden (vgl. Abbildung 20).

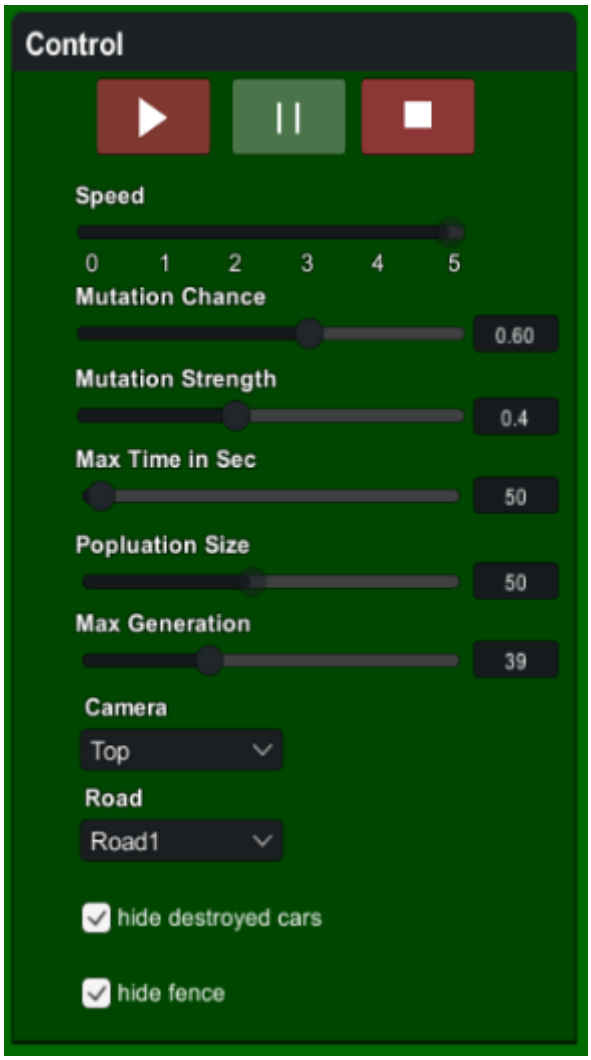

Abbildung 20: Steuerungsfenster

Fenster mit Verteilungsdiagramm

In diesem Fenster wird ein Säulendiagramm gezeigt. Das Diagramm liefert Informationen über die Verteilung der Fitness in der vorherigen Generation. Die x-Achse zeigt die nach Fitness sortierten Individuen. Auf der y-Achse wird der Fitnesswert dargestellt. Die Farben der Säule haben identische Farben, wie die Fahrzeuge in der vorherigen Generation. Diese zeigen die Verwandtschaft zwischen die Individuen (vgl. Abbildung 21).

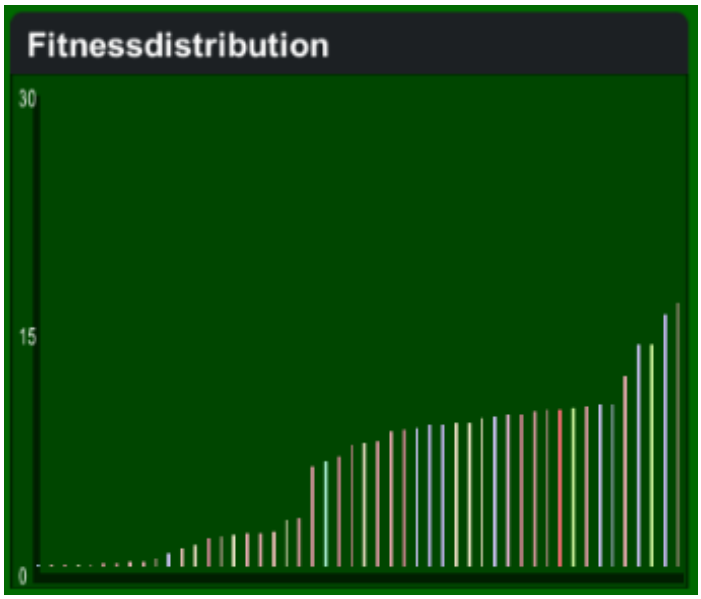

Abbildung 21: Fenster mit Verteilungsdiagramm

Fenster mit Fitnessdiagramm

In diesem Fenster wird ein Liniendiagramm veranschaulicht, das Auskunft über den Anstieg der besten Fitness und der durchschnittlichen Fitness während der Evolution gibt. Die x-Achse zeigt die Generationsnummer. Den Fitnesswert veranschaulicht die y-Achse. (siehe Abbildung 22)

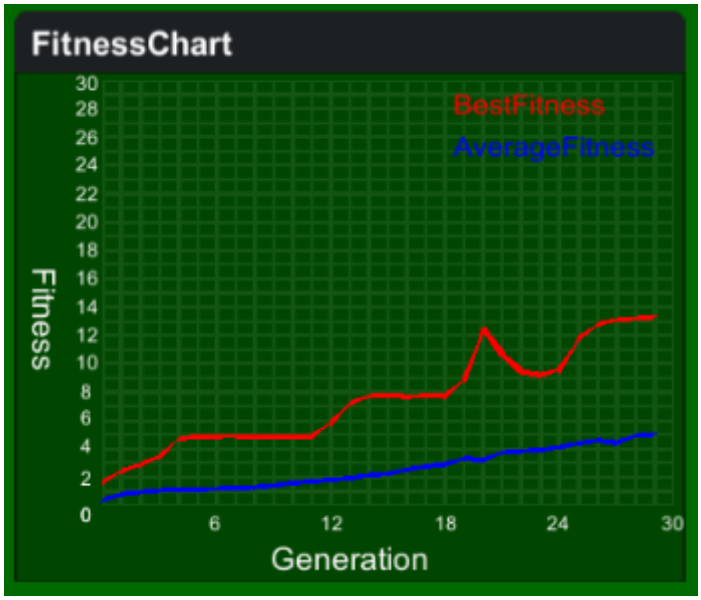

Abbildung 22: Fenster mit Fitnessdiagramm

4.3 Implementierung des Neuroevolutionären Algorithmus

4.3.1 Aufbau des Neuronalen Netzes

In diesem Abschnitt werden einige grundlegende Komponente der NeuralNetwork-Klasse dargestellt. (vgl. 2.2)
Die Klasse des Neuronales Netzes enthält Verzweigte Arrays zur Speicherung der Werte der Gewichte, Verzerrungen und Neuronen. Der Fitnesswert und die Farbe eines Individuums werden auch hier gespeichert (vgl. Listing 7).

```
private int[] layers;//layers
//this will be used to
//hold the values generated during the feedforward algorithm
private float[][] neurons;
private float[][] biases;//biasses
private float[][][] weights;//weights
public float Fitness { get; set; } = 0;//fitness
private int hashCode; //the hashCode hold the color of the vehicle.
```

Listing 7: Eigenschaften und Attribute der NeuralNetwork-Klasse (29)

Als Aktivierungsfunktion wird die Tanh-Funktion (siehe Listing 8) benutz. Im Gegensatz zur Sigmoid-Funktion reicht der Bereich der Tanh-Funktion von (-1 bis 1) (siehe Abbildung 23). Der Vorteil von Tanh-Funktion ist, dass es leichter mit negativen Zahlen umgehen kann. So kann die Ausgabewert der Funktion direkt für die Lenk- und Beschleunigungssteuerung des Fahrzeugs verwendet werden.

```
public static float Activate(float value)
{
    return (float)Math.Tanh(value);
}
```

Listing 8: Implementierung der Tanh-Funktion

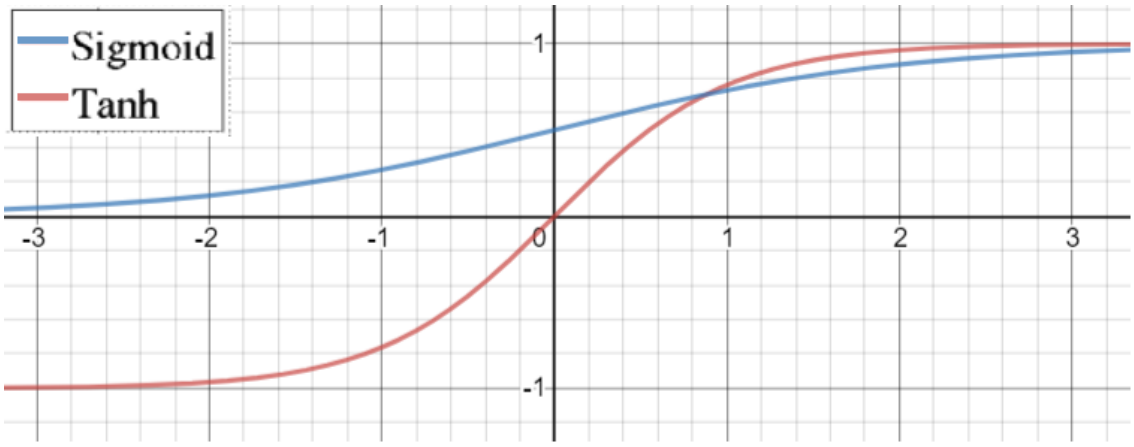

Abbildung 23: Vergleich zwischen der Tanh-Funktion und der Sigmoid-Funktion

Die NeuralNetwork-Klasse bietet auch Methoden an, um Genetische Operationen (vgl. 2.3.2) wie Crossover (siehe Listing 9) und Mutation (siehe Listing 10) durchzuführen.

```
/// <summary>
/// crossover two neural networks
/// </summary>
/// <param name="other">the other neural network to
/// crossover with the current one</param>
/// <returns>
/// The result is a combination between the two neural networks.
/// </returns>
public NeuralNetwork Crossover(NeuralNetwork other)
{
    NeuralNetwork result = DeepCopy(this);
    for (int i = 0; i < biases.Length; i++)
    {
        for (int j = 0; j < biases[i].Length; j++)
        {
            if(UnityEngine.Random.Range(0f, 1f) > 0.5)
                result.biases[i][j] = other.biases[i][j];
        }
    }

    for (int i = 0; i < weights.Length; i++)
    {
        for (int j = 0; j < weights[i].Length; j++)
        {
            for (int k = 0; k < weights[i][j].Length; k++)
            {
                if (UnityEngine.Random.Range(0f, 1f) > 0.5)
                    result.weights[i][j][k] = other.weights[i][j][k];
            }
        }
    }
    result.hashCode = Convert.ToInt32(0x000000FF);
    return result;
}
```

Listing 9: Crossover-Methode zum Kombinieren von Neuronalen Netzen

```
/// <summary>
/// mutate the values of the neural network
/// </summary>
/// <param name="chance">mutation chance</param>
/// <param name="strength">max mutation Strength</param>
public void Mutate(float chance, float strength)
{
    for (int i = 0; i < biases.Length; i++)
    {
    for (int j = 0; j < biases[i].Length; j++)
    {
        biases[i][j] =
            (UnityEngine.Random.Range(0f, 1f) < chance) ?
            biases[i][j] += UnityEngine.Random.Range(-strength, strength)
            : biases[i][j];
    }
    }

    for (int i = 0; i < weights.Length; i++)
    {
    for (int j = 0; j < weights[i].Length; j++)
    {
    for (int k = 0; k < weights[i][j].Length; k++)
    {
        weights[i][j][k]
        = (UnityEngine.Random.Range(0f, 1f) < chance) ?
        weights[i][j][k] += UnityEngine.Random.Range(-strength, strength)
        : weights[i][j][k];
    }
    }
    }

    hashCode = MutateHash(hashCode);
}

/// <summary>
/// mutate the hashcode
/// the hashcode represent the color of the car in the simulation
/// </summary>
/// <param name="color">the color to be mutated</param>
/// <returns>the mutated color</returns>
private static int MutateHash(int color)
{

    byte[] bytes = System.BitConverter.GetBytes(color);
    byte colorMutationValue = 20;

for (int i = 0; i < bytes.Length; i++)
{

    if (bytes[i] + (int)colorMutationValue > 255)
    {
        bytes[i] -= (byte)(colorMutationValue * 3);
    }
    else if (bytes[i] - (int)colorMutationValue < 0)
    {
        bytes[i] += (byte)(colorMutationValue * 3);
    }
    else
```

```
        {
            int addOrSubtract = rndm.Next(0, 1);
            if (addOrSubtract == 0)
            {
                bytes[i] += colorMutationValue;
            }
            else
            {
                bytes[i] -= colorMutationValue;
            }
        }
        }

        int result;
        result = System.BitConverter.ToInt32(bytes, 0);

        return result;
    }
```

Listing 10: Implementierung der Mutation-Methode (29)

4.3.2 Ablauf des entwickelten Neuroevolutionären Algorithmus

Folgende Schritte werden im Ablauf des Algorithmus durchgeführt (vgl. 2.4):

1. Als Anfangspopulation werden eine Menge von zufälligen Neuronalen Netzen (Genotyp) und eine Menge derselben Größe von Fahrzeug Game-Objekten (Phänotyp) initialisiert.
2. Die Fahrzeuge werden in der von Benutzer gewählten Strecke fahren. Die maximale Fahrzeit ist durch die MaxTimeInSeconds-Variable begrenzt.
3. Danach werden die Individuen anhand der Fitness-Funktion evaluiert. Diese werden danach nach deren Fitnesswert sortiert. (siehe Listing 11)
4. Die Population wird danach in zwei Hälften geteilt und dann wird die obere Hälfte in die untere geklont und mutiert. Es werden mit einer niedrigen Wahrscheinlichkeit Neuronale Netze vor dem Klonen mit anderen Neuronalen Netzen kombiniert. (vgl. Listing 12)
5. Die Schritte 2-4 werden für mehrere Generationen wiederholt, bis die maximale Anzahl von Genrationen erreicht wird.

```
    //list of all the neural networks of the population
    public List<NeuralNetwork> networks;

   /// <summary>
   /// this methode initializes the neural networks of the population
   /// </summary>
    public void InitNetworks()
    {
        networks = new List<NeuralNetwork>();
        for (int i = 0; i < PopulationSize; i++)
        {
```

```
            NeuralNetwork net = new NeuralNetwork(layers);
            networks.Add(net);
        }

    }
    private int alive = 0;//current number of not destroyed Vehicles
    public int Alive { get { return alive; } set { alive = value; } }

    //list of all vehicle in the population
    public static List<GameObject> cars = new List<GameObject>();

    /// <summary>
    /// The process begins with a set of
    /// individuals which is called a Population.
    /// Each individual is a solution to the problem you want to solve.
    /// An individual is characterized by
    /// a set of parameters(variables) known as Genes.
    /// </summary>
    public void InitPopulation()
    {
        alive = PopulationSize;
        if (cars.Count > 0)
        {
            //this sorts the networks and mutates them
            GenerateNewGeneration();
        }

        for (int i = 0; i < PopulationSize; i++)
        {
            if (cars.Count < PopulationSize)
            {
                GameObject g =
                Instantiate(carPrefab, carPrefab.transform.position
                , carPrefab.transform.rotation);
                cars.Add(g);
            }

            cars[i].GetComponent<Brain>().Network = networks[i];
            cars[i].GetComponent<BodyRenderer>()
            .SetColor(networks[i].GetHashCode());
            cars[i].GetComponent<Brain>().Surviving = true;
            cars[i].transform.position = carPrefab.transform.position;
            cars[i].transform.rotation = carPrefab.transform.rotation;
            cars[i].GetComponent<CarController>().RemoveAllForces();
            cars[i].GetComponent<Distance>().Reset();

            Timer.Reset();
        }

        Generation++;
    }
    /// <summary>
    /// generate the new population
    /// </summary>
    public void GenerateNewGeneration()
    {
        /*pause the simulation
        in case of exceeding the max number of generations*/
        if (Generation >= MaxGeneration)
        {
            PauseTheSimulation();
        }
```

```
        networks.Sort();
        CalculateAvgMeanAndBestFitness();
        FitnessList.Add((float)AverageFitness);
        //invoke the EndOfTheGeneration event
        EndOfTheGeneration.Invoke();
        SelectCrossoverAndMutatePopulation();
    }
```

Listing 11: Die Erzeugung einer neuen Population (29)

```
    /// <summary>
    /// This methode makes a copy of the best half of the neural network
    /// and removes the other half
    /// The new half will be mutated
    /// and a new generation will be created
    /// Mutation occurs to maintain diversity within the population
    /// and prevent premature convergence.
    /// </summary>
    public void SelectCrossoverAndMutatePopulation()
    {
        for (int i = 0; i < PopulationSize / 2; i++)
        {
        //make a deep copy of the neural network
        networks[i] = networks[i + cars.Count / 2]
        .Clone(new NeuralNetwork(layers));

        //crossover only on 5% of the population
        if (Random.Range(0, 99) >= 90)
        {
            networks[i] = networks[i]
            .Crossover(
            networks[Random.Range(PopulationSize / 2, PopulationSize)]
            );
        }
        //mutate the copy
        networks[i]. Mutate((MutationChance), MutationStrength);
        }
    }
```

Listing 12: Selektion, Mutation und Kombination der Neuronalen Netze

Die Fitnessfunktion

Ein größerer Fitnesswert eines Neuronalen Netzwerks erhöht die Chance, dass es für die nächste Generation überlebt. Der Fitnesswert wird wie folgt berechnet:

$fitness = distance/maxtime$ (vgl. Listing 13)

Wobei $distance$ die Länge der Strecke ist, die das Fahrzeug gefahren ist und $maxtime$ bezeichnet die maximale Zeit, in der das Fahrzeug fahren kann und wird von dem Benutzer eingegeben.

Daher werden durch den Neuroevolutionären Algorithmus die Neuronalen Netzwerke für Fahrzeuge ausgewählt, die in einer begrenzen Zeit die Länge der gefahrenen Strecke maximierten.

```
public void CalculateFitness()
 {
     if(Network != null)
     Network.Fitness = gameObject.GetComponent<Distance>().GetDistance()
             / GameObject.FindObjectOfType<AIController>().MaxTimeInSeconds;
 }
```

Listing 13: Die implementierte Fitnessfunktion

5 Experimente und Ergebnisse

Im Folgenden werden der Versuchsplan, die Durchführung sowie die Ergebnisse der zwei durchgeführten Experimente in der Simulation vorgestellt.

5.1 Untersuchung des Einflusses der Populationsgröße auf die Effizienz des entwickelten Algorithmus

Die Populationsgröße ist einer der wichtigsten Parameter bei der Anwendung von Genetischen Algorithmen (GAs). Diese kann in vielen Fällen die Qualität der erhaltenen Lösung stark beeinflussen. (30)
Beim Experiment zum Einfluss der Populationsgröße auf die Effizienz des entwickelten Algorithmus wurden die folgende exponentiell wachsenden Populationsgrößen von 2 bis 128 berücksichtigt: 2, 4, 8, 16, 32, 64 und 128. Die Einstellungen der Simulation wurden wie in Tabelle 3 festgelegt. Aufgrund der stochastischen Eigenschaften des verwendeten GA wurde zu jeder Populationsgröße eine Serie von 5 Läufen durchgeführt. Danach wurde den arithmetischen Mittelwert der Ergebnisse errechnet.

Tabelle 3: Experiment „Populationsgröße“: Eingabewerte

Max Generation	Population Size	Max Time in Sec	Mutation Strength	Mutation Chance
49	50	100	1	0.5

Die Ergebnisse der durchgeführten sieben Teilexperimente sind in der Abbildung 24 veranschaulicht. Die Abbildung zeigt, dass der Fitnesswert der Agenten mit der Zunahme der Populationsgröße zunimmt.

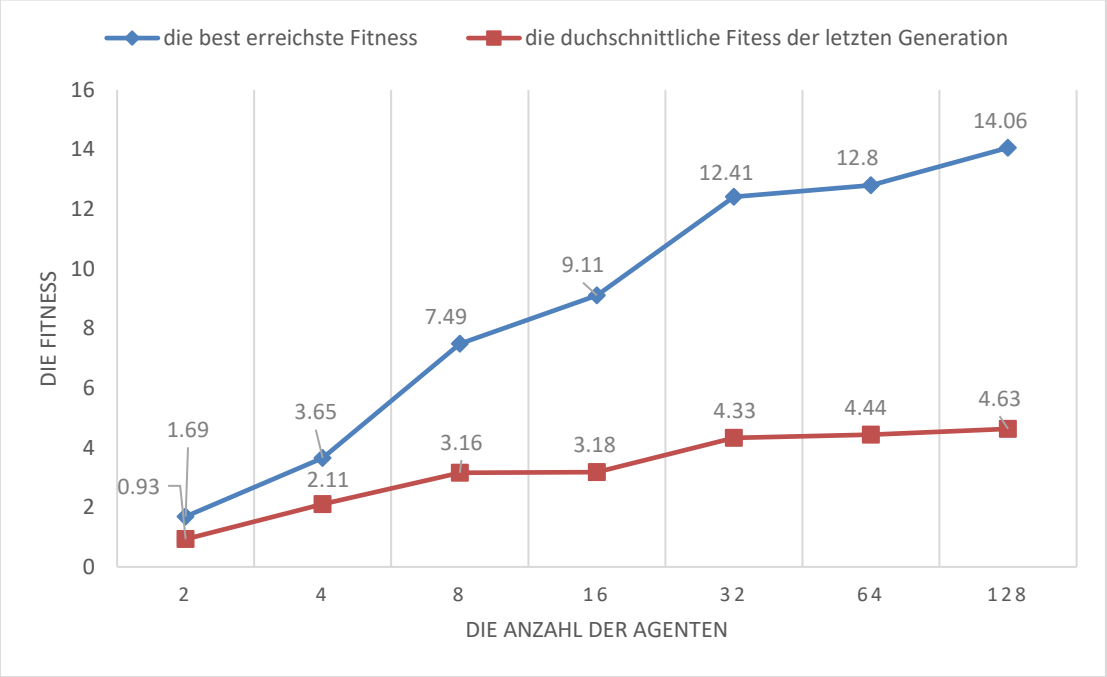

Abbildung 24: Zusammenhang zwischen der Populationsgröße und der erreichten Fitness

5.2 Untersuchung zur Generalisierbarkeit der vom Algorithmus gefundenen Lösungen

Die Fähigkeit zur Generalisierung einer Lösung ist einer der wichtigsten Vorteile Neuronaler Netze. Diese wird durch folgendes Experiment bestätigt. Dazu wurde eine Simulation mit den Einstellungen wie in Tabelle 4 gestartet. Dieses Experiment zeigt, ob die gefundenen Lösungen in der Lage sind, generalisiert zu werden.

Tabelle 4: Einstellungen für das Starten der Simulation

Max Generation	Population Size	Max Time in Sec	Mutation Strength	Mutation Chance	Road
49	50	100	1	0.5	Road1 vgl. Abbildung 12

Als die Generation 50 erreicht wurde, pausierte die Simulation (siehe Abbildung 25). In dieser Zeit wurden „Max Generation“ auf 59 und „Road“ auf „Road2“ umgestellt. Da-

nach wurde die Simulation fortgesetzt, bis die Generation 60 erreicht wurde. Obwohl die Strecke gewechselt wurde, gab es keine beträchtliche Änderung im Wert der besten Fitness einer Generation und im Wert der durchschnittlichen Fitness (siehe Abbildung 26).

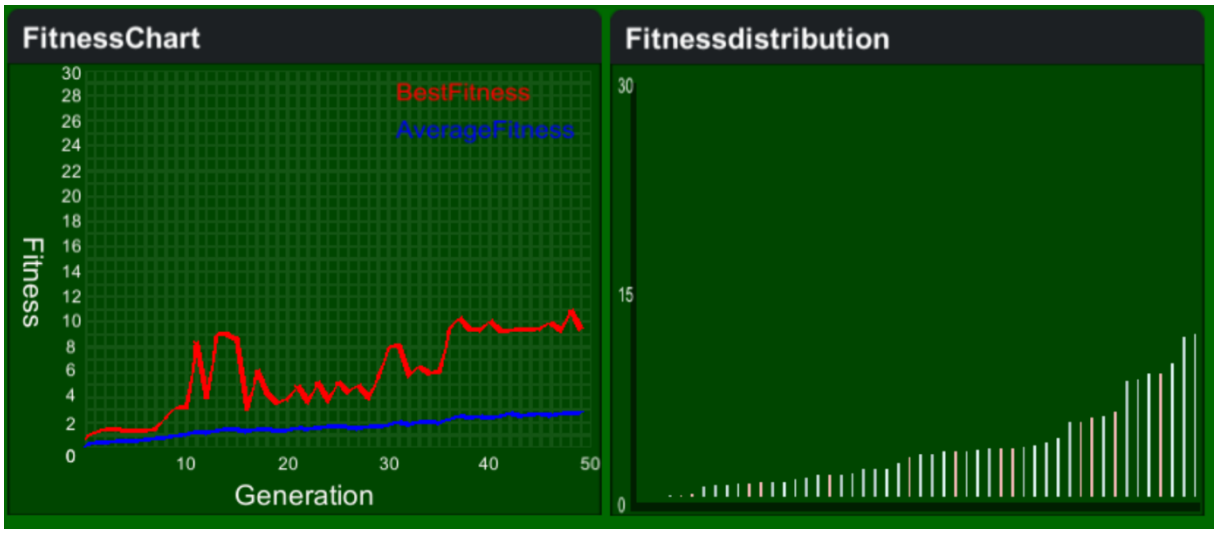

Abbildung 25: Fitnessdiagramm und Fitnessverteilungsdiagramm für die Simulation nach 50 Generationen auf Strecke 1

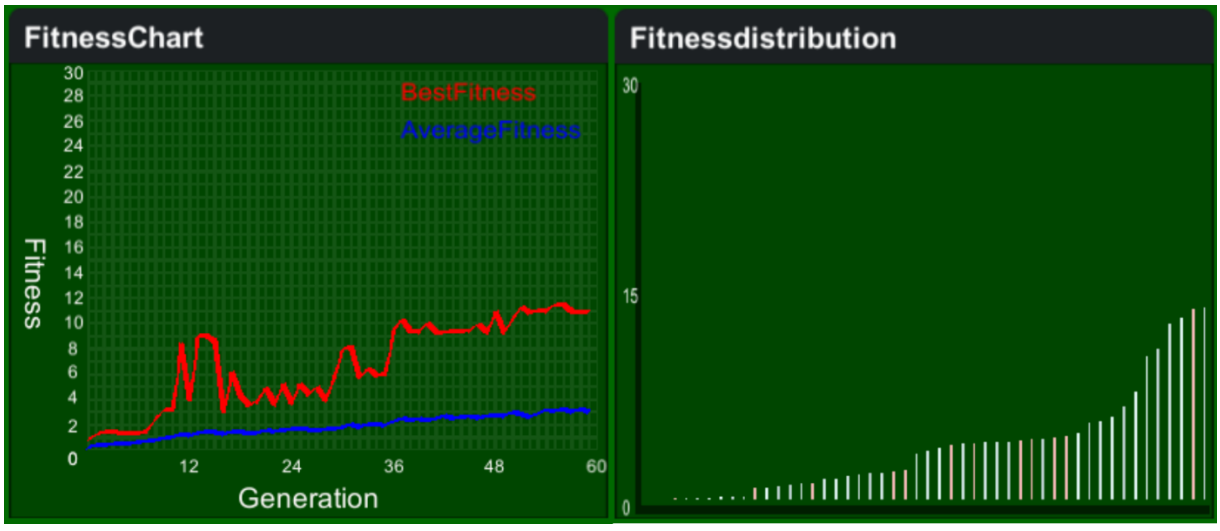

Abbildung 26: Fitnessdiagramm und Fitnessverteilungsdiagramm für die Simulation nach 50 Generationen auf die erste Strecke und 10 Generationen auf Strecke 2

6 Zusammenfassung

Im Rahmen dieser Arbeit wurden die Entwicklung eines Neuroevolutionären Algorithmus und einer Simulation eines selbstfahrenden Fahrzeugs dargestellt. Dieser Algorithmus sollte Fahrzeugsagenten ermöglichen, auf einer Strecke autonom zu fahren, ohne jemals vom Weg abzukommen.

Zur Erfüllung dieser Zielstellung wurden zunächst die relevanten theoretischen Grundlagen zu den Künstlichen Neuronalen Netzen, den evolutionären Algorithmen und neuroevolutionären Algorithmen präsentiert. Die Implementierung des Lernalgorithmus erfolgte in der Programmiersprache C# unter Verwendung des Unity-Game-Engines zur Simulation von Strecken, Fahrzeugen und Physische Reaktionen. Für das Schreiben des Codes wurde das Programm Visual-Studio verwendet.

Es wurde im Rahmen der Arbeit untersucht, welchen Einfluss die Populationsgröße auf die Effektivität des Algorithmus hat. Anhand dieser Untersuchung wurde festgestellen, dass der Fitnesswert der Agenten mit der Zunahme der Populationsgröße zunimmt. Damit wird deutlich, dass eine Lösung mit hohem Fitnesswert die Voraussetzung dafür ist, dass sich das Fahrzeug auf der Strecke besser orientieren kann. Darüber hinaus wurde ein Experiment zur Generalisierbarkeit der gefundenen Lösung präsentiert. Damit wurde gezeigt, dass es nach der Evolution von Agenten und trotz einer Streckenänderung, keine beträchtliche Änderung im Wert der besten Fitness einer Generation und im Wert der durchschnittlichen Fitness gab.

Aus den dargestellten Ergebnissen der Arbeit lässt sich schließen, dass sich ein Neuroevolutionärer Algorithmus zur Lösung einer Fahraufgabe in einem einfachen simulierten Szenario eignet.

7 Anhang

7.1 Abkürzungsverzeichnis

KNN	Künstliche Neuronale Netz
KI	Künstliche Intelligenz
NE	Neuroevolution
NEA	Neuroevolutionäre Algorithmus
GA	Genetische Algorithmen
EP	Evolutionäre Programmierung
ES	Evolutionsstrategien
GP	Genetische Programmieren
NEAT	NeuroEvolution of Augmented Topologies

7.2 Abbildungsverzeichnis

7.3 Tabellenverzeichnis

8 Literaturverzeichnis

1. **Chollet, Francois.** *Deep Learning with Python.* [ed.] 1st. s.l. : Manning Publications Co., 2017.

2. **Boersch, Ingo, Heinsohn, Jochen und Socher, Rolf.** *Wissensverarbeitung: Eine Einführung in die Künstliche Intelligenz für Informatiker und Ingenieure.* 2 edition. Heidelberg : s.n., 2007.

3. **Rodzin, Serguy, Rodzina, Olga und Lada, Rodzina.** *Neuroevolution: Problems, algorithms, and experiments.* Rostov-on-Don : International Conference on Application of Information and Communication Technologies (AICT), 2016.

4. **Döbel, Inga, et al.** Maschinelles Lernen. Eine Analyse zu Kompetenzen, Forschung und Anwendung. *bigdata-ai.fraunhofer.* [Online] https://www.bigdata-ai.fraunhofer.de/content/dam/bigdata/de/documents/Publikationen/Fraunhofer_Studie_ML_201809.pdf.

5. **Patterson, Josh and Gibson, Adam.** *Deep Learning A Practitioner's Approach.* s.l. : O'Reilly Media, Inc., 2017.

6. **Chi Nhan, Nguyen und Oliver, Zeigermann.** *Machine Learning kurz & gut.* 2018.

7. *A System Based on Artificial Neural Networks for Automatic Classification of Hydrogenerator Stator Windings Partial Discharges.* **Oliveira, Rodrigo M. S. de, et al.** 2017, Journal of Microwaves, Optoelectronics and Electromagnetic Applications, S. 628-645.

8. **Ibm.com. [Online] 2020. [Zitat vom: 12. 3 2021.] https://www.ibm.com/cloud/learn/neural-networks#toc-how-do-neu-vMq6OP-P.**

9. Pohlheim, Hartmut. *Evolutionäre Algorithmen.* Berlin : Springer, 1999.

10. Nissen, Volker. *Einführung in evolutionäre Algorithmen.* Braunschweig : Vieweg, 1997.

11. Masrour, Tawfik, El Hassani, Ibtissam and Cherrafi, Anass. Artificial intelligence and industrial applications. Cham : Springer, 2021, p. 21.

12. *Designing neural networks through neuroevolution.* Stanley, Kenneth O., et al. 2019, Nature Machine Intelligence, S. 24-35.

13. Stanley, Kenneth. Neuroevolution: A different kind of deep learning. *O'Reilly Media.* [Online] 13. 7 2017. https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/.

14. *Evolving Neural Networks through Augmenting Topologies.* Stanley, Kenneth O. und Miikkulainen, Risto. 2002, Evolutionary Computation.

15. *Evolutionary Computation for Reinforcement Learning.* Whiteson, Shimon. 2012, Reinforcement Learning: State of the Art, S. 5.

16. Technologies, Unity. Unity - Manual: Physics. *Docs.unity3d.comDocs.unity3d.com.* [Online] https://docs.unity3d.com/2019.3/Documentation/Manual/PhysicsSection.html.

17. Asset Store (Customers). *support.unity.com.* [Online] 2020. https://support.unity.com/hc/en-us/articles/210142503-What-is-the-Unity-Asset-Store-and-how-do-I-purchase-Assets-.

18. Unity - Manual: Scenes. *Docs.unity3d.com.* [Online] 2019. https://docs.unity3d.com/Manual/CreatingScenes.html.

19. Unity - Manual: GameObjects. *Docs.unity3d.com.* [Online] 23. 2 2021. https://docs.unity3d.com/Manual/GameObjects.html.

20. Unity - Manual: Creating and Using Scripts. *Docs.unity3d.com.* [Online] 22. 2 2021. https://docs.unity3d.com/2021.1/Documentation/Manual/CreatingAndUsingScripts.html.

21. Unity - Manual: Rigidbody. *docs.unity3d.com.* [Online] 23. 2 2021. https://docs.unity3d.com/Manual/class-Rigidbody.html.

22. Unity - Manual: Wheel Collider. *Docs.unity3d.com.* [Online] 23. 2 2021. https://docs.unity3d.com/Manual/class-WheelCollider.html.

23. Unity 2020.1 improvements. *Unity.* [Online] 14. 7 2020. https://unity3d.com/de/beta/2020.1b.

24. Einführung in C# - Leitfaden für C#. *Docs.microsoft.com.* [Online] 28. 1 2021. https://docs.microsoft.com/de-de/dotnet/csharp/tour-of-csharp/.

25. Visual Studio IDE, Code-Editor, Azure DevOps und App Center - Visual Studio. *Visual Studio.* [Online] [Zitat vom: 22. 03 2021.] https://visualstudio.microsoft.com/de/.

26. Modular Track. *assetstore.unity.com.* [Online] 12. 11 2019. https://assetstore.unity.com/packages/3d/environments/modular-track-85356.

27. ARCADE: FREE Racing Car. *assetstore.unity.com.* [Online] 13. 1 2021. https://assetstore.unity.com/packages/3d/vehicles/land/arcade-free-racing-car-161085.

28. Simple Car Controller in Unity Tutorial. *Youtube.com.* [Online] 9. 6 2020. [Zitat vom: 2021. 3 22.] https://www.youtube.com/watch?v=Z4HA8zJhGEk.

29. UI Widgets. *assetstore.unity.com.* [Online] 29. 7 2020. https://assetstore.unity.com/packages/tools/gui/ui-widgets-55542#description.

30. Building a neural network framework in C#. ***Medium.*** **[Online] [Zitat vom: 21. 03 21.] https://towardsdatascience.com/building-a-neural-network-framework-in-c-16ef56ce1fef.**

31. ***Effects of population size on the performance of genetic algorithms and the role of crossover.*** **Zhang, Yu-an, et al. 2010, Artificial Life and Robotics.**

32. What is Genetic Operators. ***igi-global.com/.*** **[Online] https://www.igi-global.com/dictionary/an-intelligent-process-development-using-fusion-of-genetic-algorithm-with-fuzzy-logic/12073.**

33. ***Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning.*** **Petroski, Felipe, et al. 2018.**

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Ort, Datum

Potsdam, 22.03.2021

Unterschrift

(Mulham Alesail)