CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Bachelorstudiengang Informatik

Bachelorarbeit

# Simulation selbstfahrender Autos mithilfe von künstlichen neuronalen Netzen und evolutionären Algorithmen

vorgelegt von

**Eike Stein**

Gutachter

**Prof. Dr. H.-J. Appelrath**
**Cornelius Ludmann**

Oldenburg, 17. August 2016

# INHALT

# 1 Einleitung

Im Jahr 2015 starben alleine in Deutschland 3475 Menschen durch Verkehrsunfälle [1]. Bestehende Sicherheitssysteme beschränken sich in der Regel auf die technische Unterstützung des Fahrers. Dazu zählen Antiblockiersysteme (ABS), Elektronisches Stabilitätsprogramme (ESP), Brake Assist Systems (BAS) und je nach Ausstattungsgrad noch weitere. Dem gegenüber begann sich in den letzten Jahren ein Zweig der Informatik zu entwickeln, mit dem Ziel, die Steuerung von Autos vollständig durch Computer zu realisieren. Zu den führenden Forschungseinrichtungen in diesem Bereich zählen auch große Unternehmen wie Google und Tesla. Für eine erfolgreiche Umsetzung autonomer Autos, müssen eine Reihe von Themen, wie Sensorverarbeitung, Bildverarbeitung und Psychologie, behandelt werden. Die besondere Herausforderung bei autonomen Autos, ist das Zusammenspiel vieler verschiedener Teilbereiche, so zu koordinieren, dass ein sicheres und zuverlässiges Fahren ermöglicht wird.

Aufgrund der verschiedenen Umwelteinflüsse ist es in der Regel nicht möglich auf alle erdenkbaren Situationen eine entsprechende Reaktion fest einzuprogrammieren, sodass Algorithmen zum Einsatz kommen müssen, die auch auf neue Gegebenheiten angemessen reagieren können. Einen solchen Ansatz verfolgen künstliche neuronale Netze. Inspiriert von neuronalen Verbindungen im Gehirn, versuchen diese gewünschtes Verhalten zu erlernen. In dieser Arbeit soll untersucht werden, inwieweit es möglich ist, mithilfe von künstlichen neuronalen Netzen Sensordaten eines simulierten Autos zu verarbeiten und die Steuerung dessen zu übernehmen. Konkret soll eine möglichst schnelle, sichere und *menschenähnliche* Fahrweise erreicht werden.

Zu Beginn der Arbeit soll in die zugrundeliegenden Technologien eingeführt werden, sowie einen ersten Überblick über die Simulationsumgebung gegeben werden. Es folgt eine Einordnung dieser Arbeit in Bezug zu anderen wissenschaftlichen Ausarbeitungen. Anschließend wird das Konzept der Arbeit vorgestellt. Dies beinhaltet den allgemeinen Ablauf der Simulation, sowie der Beschreibung wie die verwendeten Technologien konkret zum Einsatz kommen. Im darauffolgenden Kapitel wird der Entwurf der Simulation und ihrer Komponenten sowie des künstlichen neuronalen Netzes vorgestellt. Danach wird die Implementierung erläutert. Im Anschluss wird eine Ergebnisevaluation durchgeführt, die die Resultate auf fehlerfrei zurückgelegte Strecke,

Geschwindigkeit und Fahrverhalten hin untersucht. Außerdem wird ein allgemeines Fazit gezogen und ein Ausblick für zukünftige Arbeiten beschrieben.

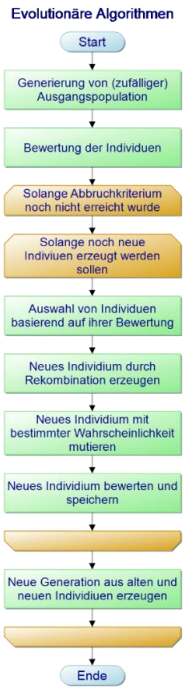
# 2 Grundlagen

## 2.1 Autonomes Fahren

Unter autonomem Fahren bezeichnet man grundsätzlich Autos, Busse, Lastwagen oder andere Verkehrsteilnehmer, die teilweise oder vollständig durch Computer gesteuert werden. Einer der ersten Beiträge auf diesem Gebiet wurde 1977 von *Tsukuba Mechanical Engineering Laboratory* in Japan geleistet. Damals konnte ein Auto weißen Straßenmarkierungen auf einem abgesperrten Testgelände folgen [2]. Mittlerweile beschäftigen sich vor allem große Unternehmen wie Google und Tesla mit der Entwicklung [3] [4]. Damit autonome Fahrzeuge in der Lage sind, sich in ihrer Umgebung zurecht zu finden, kommen eine Reihe von Sensoren zum Einsatz: Kameras, Ultrasound, GPS, Laser und einige weitere [2]. Die Aufgabe der Software ist es, die Daten, die die einzelnen Sensoren liefern zu verarbeiten und anschließend das Fahrzeug entsprechend zu kontrollieren. Es gibt eine Reihe verschiedener Lösungsstrategien die genutzt werden können. Sogenannte künstliche neuronale Netze ist einer davon [5].

## 2.2 Evolutionäre Algorithmen

Evolutionäre Algorithmen kommen häufig dort zum Einsatz, wo es zwar möglich ist eine potentielle Lösung für ein Problem zu bewerten, es aber sehr schwer ist eine solche Lösung zu konstruieren [6]. Die Idee dabei orientiert sich an der Evolutionstheorie der natürlichen Selektion nach Charles Darwin, wo die besten Individuen überleben und Nachkommen produzieren, die generell etwas besser sind als die Generationen vor ihnen. So werden nach und nach nur solche Individuen existieren, die am besten in ihrer Umgebung zurechtkommen. Wie auch bei den künstlichen neuronalen Netzen ist es nicht möglich die volle Komplexität in ein Computermodell zu übertragen. Es wird vielmehr die grundsätzliche Idee genutzt und angewendet. Die einzelnen Individuen bestehen meist nur aus einem Array an Zahlen. Diese Zahlen symbolisieren Merkmalsausprägungen und sind vergleichbar mit den Genen in der DNS [7]. Ein Evolutionärer Algorithmus läuft in der Regel wie folgt ab: [8]



1. Zunächst wird eine, meist zufällige, Ausgangspopulation generiert. In den meisten Fällen sind die Individuen nicht als Lösungskandidaten einsetzbar. Es gibt allerdings Variationen, bei denen Wissen über den Lösungsraum in die Generierung mit einfließt, was die Qualität der ersten Generation an Lösungen verbessern kann.

2. Nun werden alle Individuen anhand einer sogenannten Fitnessfunktion bewertet. Beispielsweise könnte das Ziel eines genetischen Algorithmus sein, ein möglichst aerodynamisches Auto zu modellieren. Dabei wären die einzelnen Gene die Positionen der Ecken der Karosseriekomponenten. Die Fitnessfunktion wäre dann wie viel Luftwiderstand das Auto bei unterschiedlichen Geschwindigkeiten aufweist. Ob der Fitnesswert durch Computermodelle oder im Windkanal errechnet wird, ist dabei unerheblich. Ausschlaggebend ist nur, dass ein Wert errechnet wird, der die Güte eines Lösungskandidaten adäquat beschreibt.

3. Anschließend werden anhand verschiedener Auswahlverfahren Individuen anhand ihrer Bewertung selektiert. Meist wird die Auswahl zufällig gewichtet getroffen.

4. Die in Schritt 3 ausgewählten Lösungskandidaten werden dann paarweise (in einigen Fällen auch tripel- oder quadrupelweise) miteinander kombiniert. Dies geschieht indem zum Teil die Gene des einen, dann die Gene des anderen ausgewählt und aneinandergefügt werden. So würde $[a, b, c]$ und $[x, y, z]$ Beispielsweise zu $[a, y, z]$ oder $[x, y, c]$ werden. Welche Gene von welchem Individuum genommen werden wird i.d.R. zufällig entschieden. Die resultierenden Individuen können als *Kinder* verstanden werden.

5. Damit der erreichbare Lösungsraum nicht ausschließlich von den Kombinationsmöglichkeiten der Ausgangsindividuen abhängt, werden die *Kinder* nun mit einer gewissen Wahrscheinlichkeit mutiert. Dabei ändert sich meistens ein Gen um einen zufälligen Wert. So wird versucht den gesamten Lösungsraum erreichbar zu machen.

6. Die *Kinder* werden jetzt anhand der Fitnessfunktion bewertet und gespeichert. Es werden solange Kinder erzeugt bis eine festgelegte Anzahl erreicht wurde.

7. Aus allen erzeugten *Kindern* und *Eltern* werden jetzt wieder mit einem (häufig stochastischen) Auswahlverfahren diejenigen Individuen ausgewählt, die in

die nächste Generation übernommen werden sollen. Es wird wieder mit Schritt 3 fortgefahren und der Ablauf wiederholt sich.

Ein Evolutionärer Algorithmus läuft prinzipiell unbegrenzt lange, allerdings gibt es eine Reihe möglicher Abbruchkriterien, die entscheiden wann eine weitere Ausführung keinen Sinn mehr macht oder zu *teuer* wird. *Teuer* in dem Sinne, als dass angenommen wird, dass die Ausführung Ressourcen wie Zeit oder Geld kostet. Ein mögliches Abbruchkriterium wäre das eine bestimmte Anzahl an Generationen durchlaufen wurden, oder dass über längere Zeit kein besseres Individuum erzeugt werden konnte. Die gewählten Abbruchkriterien sind domainspezifisch, d.h. für jede Problemstellung muss das Abbruchkriterium neu gewählt werden.

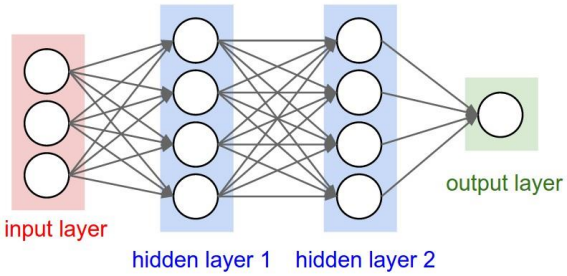
Ein Problem der Evolutionären Algorithmen ist die Parameterbestimmung. Es müssen eine Reihe von Werten im Voraus festgelegt werden, die jeweils nicht trivial von dem Lösungsraum, der Größe der Population und anderen Faktoren abhängen können. Beispielsweise ist es nicht möglich eine einheitliche optimale Mutationswahrscheinlichkeit zu empfehlen. In vielen Fällen bietet es sich auch an die Werte im Laufe des Algorithmus anzupassen was die Festlegung weiter erschwert. Es bleibt also festzuhalten, dass EAs zwar flexibel in der Anwendung sind, jedoch nicht einfach ohne Anpassung zu jeder Problemstellung eine Lösung finden.

## 2.3 Künstliche neuronale Netze

*"The computer is incredibly fast, accurate, and stupid. Man is unbelievably slow, inaccurate, and brilliant. The marriage of the two is a challenge and opportunity beyond imagination." – Stuart Walesh*

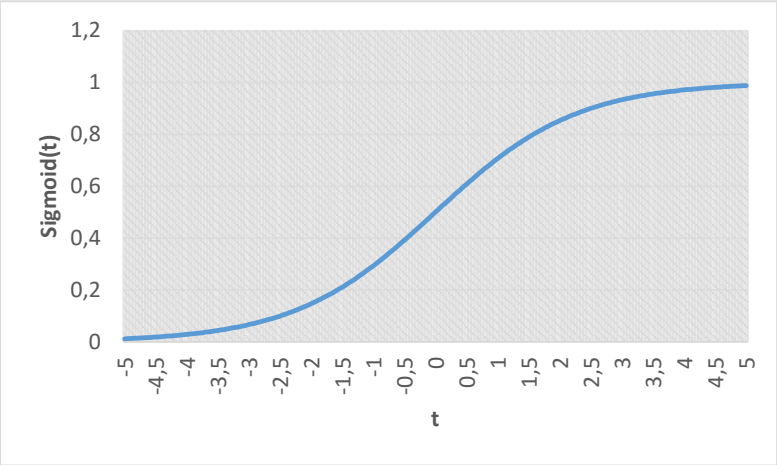
Künstliche neuronale Netze versuchen die Brücke zu schlagen, zwischen dem Intellekt von Menschen und der Rechengeschwindigkeit von Computern. Diese Netze sind inspiriert von den neuronalen Verbindungen im Gehirn. Es werden jedoch nur die grundsätzlichsten Eigenschaften übernommen und viele biologische Facetten ignoriert.

Der Aufbau jedes neuronalen Netzes ist grundsätzlich gleich. Es gibt eine Eingabeebene (*input layer*) die einen Eingabevektor akzeptiert. Über diesen Weg werden Daten an das neuronale Netz übergeben. Das biologische Äquivalent wären zum Beispiel die Augen, die Farb- und Helligkeitsinformationen wahrnehmen und an Neuronen im Gehirn weiterleiten. Die Daten des Eingabevektors werden nun an die nächste Ebene im Netz propagieren, die erste, sogenannte, *hidden layer*. Diese verarbeitet die Daten und leitet sie weiter an die nächste *hidden layer*, bis schließlich die letzte Ebene erreicht wird und die Ergebnisse ausgelesen werden können (die *output layer*). Jedes Element des Eingabevektors wird genau an ein *Neuron* der Eingabeebene geleitet. Jedes dieser Neuronen ist üblicherweise mit jedem Neuron der nächsten Ebene verbunden.




Die Ausgabe eines Neurons, also welcher Wert an die nächste Ebene weitergeleitet wird, errechnet sich mithilfe einer Aktivierungsfunktion. Die Funktion hat das Ziel die Ausgabe immer im gleichen Intervall zu halten. So könnte ein Neuron aufgrund der Eingangskonfiguration einen Wert annehmen, der unproportional groß oder klein ist. Mithilfe der Funktion wird der Wert jedoch wieder in das Intervall [0,1] oder [-1,1] projiziert. Eine Aktivierungsfunktion die häufig zum Einsatz kommt, ist die

sogenannte Sigmoid-Funktion. Sie hat die Form $\frac{1}{1+e^{-t}}$, wobei $t$ dabei der ursprüngliche Wert ist. Ihr Verlauf skizziert sich so:




Desweitern sind die Verbindungen gewichtet. Das bedeutet das jeder Wert der von der Aktivierungsfunktion berechnet wurde mit einer bestimmten Gewichtung mit in den Eingabewert eines Neurons eine Ebene weiter fließt. Der letztendliche Eingabewert ergibt sich aus der Summe aller gewichteten Ausgabewerte der Neuronen der vorherigen Ebene. Somit ergibt sich folgende Formel für ein Neuron mit dem Index *i* in Ebene *k*:

$$Neuron_{i,k} = Sigmoid(\sum_{j=0}^{n} Ausgabe(Neuron_{j,k-1}) * Gewicht_{(j,k-1),(i,k)})$$

So errechnet sich der Ausgabewert jedes Neurons, mit Ausnahme denen der ersten Ebene, die ihren Wert explizit durch die Daten gesetzt bekommen. Wie sich die Anzahl der *hidden layers* festlegt und wie viele Neuronen sich jeweils in ihnen befinden, ist variabel und hängt von der Komplexität des Einsatzgebietes ab. Ein einfaches *Und-Gatter* lässt sich beispielsweise mit 2 Eingabeneuronen und einem Ausgabeneuron realisieren und benötigt somit gar keine *hidden layers*. Möchte man ein *Exklusiv-Oder-Gatter* nachstellen benötigt man hingegen schon eine *hidden layer*. Intuitiv lässt sich das damit erklären, dass die Eingabedaten in Verbindung zueinander gesetzt werden müssen.

Desweitern werden in der Regel *Bias*-Neuronen implementiert. Diese speziellen Neuronen haben als Ausgabewert immer 1 und ermöglichen so auch Informationen aus einem Eingabevektor zu gewinnen, bei dem alle Werte 0 sind. Ein Beispiel dafür wäre ein *NOR-Gatter*. Die Ausgabe soll unter anderem dann 1 sein, wenn alle Eingangswerte 0 sind. Ohne *Bias-Neuronen* ist es nicht möglich die Verbindungen

im Netz so zu gewichten, dass aus einem 0-Eingabevektor ein *nicht-0-Ausgabewert* wird.

$$I_0 = 0, I_1 = 0, dann\ I_0 * w_0 + I_1 * w_1 = 0$$

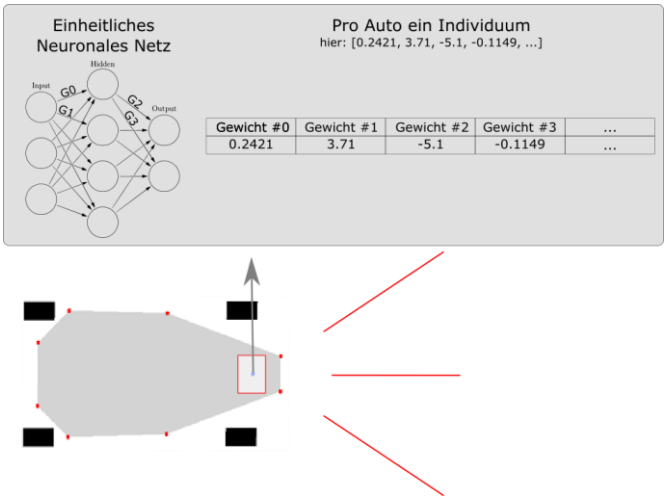
Damit die Ausgabe eines neuronalen Netzwerks überhaupt sinnvoll verwendet werden kann, muss das Netz zunächst trainiert werden. Dazu werden Trainingsdaten erzeugt, an denen das Netz trainiert werden kann. Wurde ein vorher festgelegtes Trainingsziel erreicht, kann das Netz nun an Daten außerhalb des Trainingsdatensatzes getestet werden. So könnte zum Beispiel die Börsendaten der letzten Wochen und Monate als Trainingsdatensatz genutzt werden, da hierfür bekannt ist, wie sich die Aktienkurse tatsächlich verändert haben. Im Trainingsprozess wird jetzt versucht zu erreichen, dass das neuronale Netz die Aktienkurse möglichst präzise vorhersagt. Ist das Training abgeschlossen, kann das Netz für Vorhersagen aktueller Aktienkurse eingesetzt werden. In der Realität gibt es in diesem Prozess einige Hürden, deren Bewältigung nicht ganz einfach ist.

In fast allen Fällen bleibt der Aufbau des Netzwerks über den gesamten Trainingszeitraum gleich, nur die einzelnen Gewichte ändern sich. Für die Bestimmung der Gewichte gibt es eine Reihe verschiedener Ansätze, die unterschiedliche Vor- und Nachteile mit sich bringen. Häufig kommt der *Back-Propagation-Algorithmus* zum Einsatz. Die Idee bei diesem Algorithmus ist es, die Ausgabe des Netzwerks mit einer vorher festgelegten Ausgabe zu vergleichen. Je größer der Abstand zur gewünschten Ausgabe je größer der Fehler. Dieser Fehlerwert wird dann von der Ausgabeebene durch die Ebenen zurück propagiert und die Gewichte werden dabei korrigiert. Der Vorteil liegt vor allem in der einfachen Implementierung, allerdings ist das Trainieren sehr zeitaufwendig. Ein weiteres Problem ist, dass man zunächst Trainingsdaten benötigt, die jedem gegebenen Datensatz genau einen Ausgabevektor zuordnen. Das ist zwar häufig kein Problem, wie an dem Beispiel mit den Aktienkursen gezeigt, aber nichtsdestotrotz gibt es Situationen in denen die Erzeugung der gewünschten Ausgabedaten nicht ohne weiteres möglich ist. Angenommen das Ziel ist es ein künstliches neuronales Netz als Steuerungseinheit für ein Weltraumfahrzeug zu trainieren, dass später im Falle eines Verbindungsabbruchs selbstständig die Umgebung auf einem Himmelskörper erkundet. Zwar ist es durchaus denkbar, dass mithilfe von Simulationen ein Datensatz mit verschiedenen Sensorwerten generiert werden kann, allerdings stellt das Festlegen der gewünschten Ausgabewerte ein großes Problem

dar. In solchen Fällen können häufig evolutionäre Algorithmen eingesetzt werden. Die einzelnen Individuen stellen dabei jeweils eine Gewichtskonfiguration dar. Jedes *Gen* entspricht einem Verbindungsgewicht. Die Bewertungsfunktion ist dabei nicht direkt an die Ausgabe gekoppelt, sondern vielmehr an das Verhalten welches von den Ausgabewerten ausgeht. So könnte bei dem Weltraumfahrzeug die zurückgelegte Strecke ein Faktor für die Bewertung eines Individuums sein; ein Wert der ebenfalls in Simulationen bestimmt werden kann.

## 2.4 Simulationsumgebung

Nachdem in EAs und KNNs eingeführt wurde, soll nun deren Rolle in der Simulation erläutert werden. Die Simulation benutzt GPS-Koordinaten bekannter Rennstrecken um einen virtuellen Rundkurs zu erzeugen. Auf diesem Rundkurs werden dann Autos platziert, die mithilfe von Abstandssensoren ihre Umgebung wahrnehmen können. Die Autos besitzen jeweils ein künstliches neuronales Netz um die Sensorwerte in Geschwindigkeit und Lenkrichtung zu übertragen. Der Aufbau der KNNs unterscheidet sich dabei nicht zwischen den Autos nur die Gewichte der einzelnen Verbindungen. Die Gewichte werden mithilfe von einem EA trainiert. Jedes Individuum in dem EA entspricht einer Gewichtskombination. Dabei gibt jedes Gen das Gewicht genau einer Verbindung im neuronalen Netz an. Das bedeutet, dass sich die Autos nur in der Gewichtskonfiguration der Verbindung ihrer neuronalen Netze unterscheiden und somit die Individuen es sind, die das eigentliche Fahrverhalten der Autos bestimmen.





Es wird zunächst eine zufällige Startpopulation generiert. Nun wird für jedes Individuum ein Auto mit der entsprechenden Gewichtskonfiguration auf einem

ausgewählten Rundkurs platziert. Anschließend wird die Simulation gestartet und die neuronalen Netze reagieren auf die Sensordaten und steuern das Auto durch den Rundkurs. Kollidiert das Auto dabei mit der Streckenbegrenzung bricht die Simulation sofort ab und die Bewertung findet statt. Alternativ beendet sich die Simulation auch nach einer festgelegten Zeit. Wie genau die Bewertung stattfindet wird im Kapitel *Simulation* näher beschrieben. Grundsätzlich folgt aber aus großer zurückgelegter Strecke mit hoher Geschwindigkeit eine hohe Bewertung. Nachdem das Fahrverhalten jedes Individuums der Population bewertet wurde, wird die neue Generation erzeugt und die Simulation startet erneut. Nach einigen Generationen hat sich dann hoffentlich ein erfolgreiches Fahrverhalten entwickelt.

# 3 Einordnung der Arbeit

Im Folgenden soll die Ausarbeit und ihre Teilthemen in Bezug zu anderen wissenschaftlichen Arbeiten gestellt werden. Insbesondere wird dabei auf autonomes Fahren, evolutionäre Algorithmen, sowie künstliche neuronale Netze eingegangen.

Eine, für diese Arbeit sehr wichtiges Buch ist „An introduction to Neural Networks“ von Ben Kröse und Patrick van der Smagt [9]. Darin beschreiben die Autoren die wichtigsten Eigenschaften von neuronalen Netzwerken und wie diese festgelegt werden können. In Bezug zu evolutionären Algorithmen stellt „Introduction to Evolutionary Algorithms“ von Xinje Yu und Mitso Gen eine gute Übersicht dar. Welche Fortschritte auf dem Gebiet des autonomen Fahrens gemacht wurden und welche Technologien dabei zum Einsatz gekommen sind werden in dem Artikel „Autonomous driving in urban environments: approaches, lessons and challenges“ von Mark Campbell, Magnus Egerstedt, Jonathan P. How und Richard M. Murray vorgestellt.

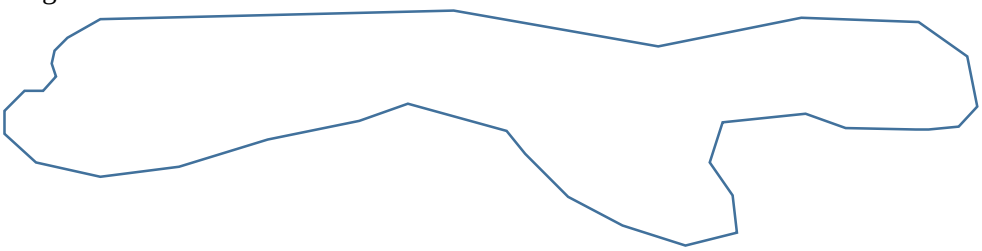
Neben diesen drei Ausarbeitungen, wird an entsprechenden Stellen auf weitere wissenschaftliche Abhandlungen verwiesen. Diese Arbeit verknüpft das Wissen dieser drei Teilgebiete miteinander und versucht so die Ausgangsfrage beantworten zu können. Eine vollständige Übersicht aller verwendeten Arbeiten findet sich im Literaturverzeichnis am Ende dieser Arbeit.

# 4 Konzept

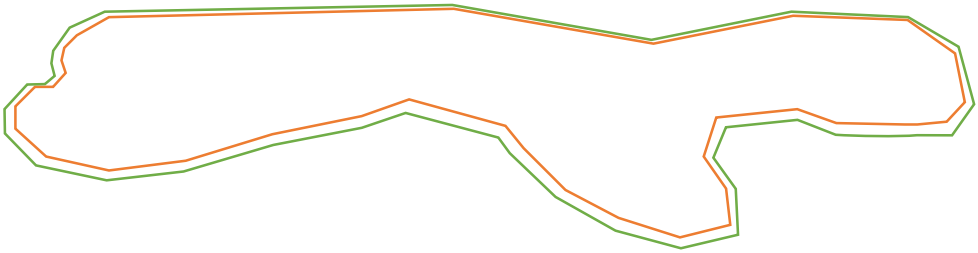
Zur Beantwortung der Frage, ob sich neuronale Netze zum Steuern von Fahrzeugen eignen, muss zunächst ein Rahmen geschaffen werden, der das Testen erlaubt. Die nächstliegende Lösung ist die Entwicklung einer Simulation. Ziel dieser Simulation ist es eine Strecke bereit zu stellen auf welcher Autos fahren und anhand ihres Fahrverhaltens hin analysiert werden können.

## 4.1 Streckengenerierung

Da der normale Straßenverkehr bei der Beantwortung der Frage außenvorgelassen wird, gibt es auch keinen Grund das Fahrzeug auf simulierten Stadtkursen fahren zu lassen. Vielmehr können GPS-Koordinaten von bekannten Rennstrecken genutzt werden. Diese sind meist vom Verlauf abwechslungsreicher und bieten somit mehr Spielraum für die Optimierung des Fahrverhaltens. Eine mögliche Quelle dieser GPS-Koordinaten stellt die Webseite GPSies dar. Dort können Benutzer Streckendaten hinterlegen und anschließend kostenlos heruntergeladen werden. So stehen viele Strecken der Formel 1 kostenlos bereit. Da es sich bei allen Strecken um Rundkurse handelt, können die GPS-Koordinaten als Eckpunkte eines Polygons aufgefasst werden:



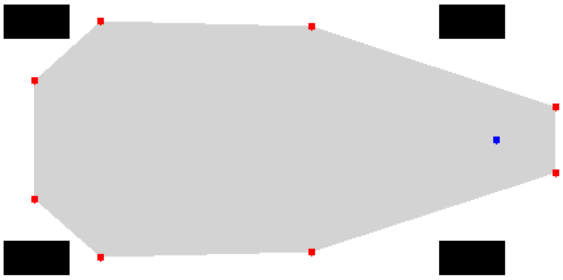
Für eine befahrbare Strecke reicht dies jedoch alleine nicht aus. Es muss eine innere und äußere Streckenbegrenzung definiert werden. Dies kann über die Streckenbreite realisiert werden. Dabei kann das generierte Polygon jeweils verkleinert und vergrößert werden; jeweils um die Hälfte der gewünschten Streckenbreite. So entsteht ein inneres und ein äußeres Polygon, die als Streckenbegrenzung genutzt werden können:



Das innere, orangene Polygon stellt dabei das verkleinerte, das äußere, grüne das vergrößerte Polygon dar. Die ursprüngliche Strecke liegt genau zwischen diesen beiden Polygonen. So können die GPS-Koordinaten in befahrbare Strecken umgewandelt werden.

## 4.2 Fahrzeug

Zunächst muss allerdings das Fahrzeug konstruiert werden und entschieden werden über welche Wege dieses auf die Umgebung reagieren kann. Da die Simulation aus Effizienzgründen ausschließlich im zweidimensionalen Raum abläuft, kann die Form des Fahrzeuges, ähnlich wie bei der Strecke, als Polygon aufgefasst werden. Zusätzlich kann über vier weitere Vektoren die Position der Reifen angegeben werden. Eine denkbare Form wäre:



Die roten Punkte geben dabei die Eckpunkte[1] an, die Mittelpunkte der schwarzen Rechtecke die Positionen[2] der Reifen. Die Form und die relative Position der Reifen, haben nicht nur eine kosmetische Relevanz, sondern beeinflussen auch die Fahrphysik. Ein zu geringer Abstand zwischen der Vorder- und Hinterachse führt dazu, dass das Fahrzeug schnell ins Schleudern gerät, wohingegen ein zu großer Abstand die Manövrierbarkeit einschränkt. Die gewählte Form des Fahrzeuges, sowie die Position der Reifen haben sich in empirischen Tests als geeignete Balance herausgestellt.

[1] (-0,45, -1,825), (-0,9, -1,325), (-0,85, 0,275), (-0,25, 2,125), (0,25, 2,125), (0,85, 0,275), (0,9, -1,325), (0,45, -1,825)
[2] (-0,9, 1,5), (0,9, 1,5), (-0,9, -1,8), (0,9, -1,8)

Zusätzlich zu der geometrischen Komponente der Fahrzeugbeschreibung, muss auch definiert sein, wie das Fahrzeug die Umgebung wahrnehmen kann. Dafür kommen Sensoren zum Einsatz, die den Abstand von einem Punkt des Fahrzeuges aus in eine gegebene Richtung bis zur nächsten Wand messen. Naheliegend ist es, die Startposition so zu wählen, dass sie ungefähr der Position eines menschlichen Fahrers entspricht. So nimmt das Fahrzeug auch nur das wahr, was auch einem Menschen an Informationen zur Verfügung stehen würde. Die Position[3], die ungefähr dem eines Menschen entspricht ist in obiger Abbildung durch einen blauen Punkt markiert. Von diesem Punkt aus werden dann in der Simulation die Abstände gemessen und stehen als Umgebungsinformation zur Verfügung.

## 4.3 Neuronales Netzwerk

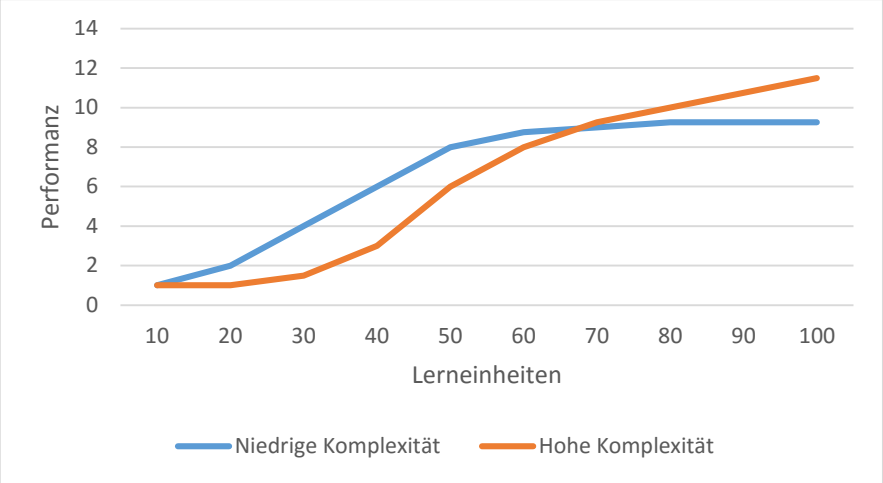
Da die Frage ist, ob die Steuerung von einem künstlichen neuronalen Netz übernommen werden kann, komm ein solches Netz auch als Fahrer in der Simulation zum Einsatz. Als Eingabedaten dienen die gemessenen Abstandswerte der Sensoren. Die Ausgabe des Netzes setzt sich aus zwei Werten zusammen. Der erste Wert stellt einen Faktor dar, der die zu erreichende Zielgeschwindigkeit angibt. Diese errechnet sich aus dem Faktor multipliziert mit der maximalen Geschwindigkeit des Autos. Aufgrund der Aktivierungsfunktion liegt der Faktor stets im Intervall $[0,1]$. Der zweite Wert gibt die Lenkrichtung an. Für diese Ausgabe wird eine Aktivierungsfunktion gewählt, die den ursprünglichen Wert in das Intervall $[-1,1]$ projiziert. *-1* entspricht dabei einer maximalen Auslenkung nach links, *+1* nach rechts.

Die Rohdaten der Umgebung bestehen aus einer Reihe meterbasierter Entfernungsmessungen, die sich so nicht unmittelbar für die Verwendung von neuronalen Netzen eignen. Aus diesem Grund wird eine maximale Entfernung festgelegt, die von den Sensoren erfasst werden kann. Die Eingabedaten errechnen sich dann über $\frac{gemessene\ Entfernung}{maximale\ Entfernung}$. So werden die Werte normalisiert und befinden sich immer zwischen $[0,1]$ und können vom Netzwerk verarbeitet werden.

Die Wahl der Richtigen Topologie bei neuronalen Netzen ist häufig eine Balance zwischen Lerngeschwindigkeit und der maximal erreichbaren Performanz. Wird die Netzstruktur zu simpel gestaltet, werden zwar recht schnell annehmbare Ergebnisse

[3] (0,1.675)

erzielt, jedoch können diese auch trotz längerem Lernprozess nicht maßgeblich verbessert werden. Auf der anderen Seite bedeutet eine komplexe Topologie auch ein zum Teil deutlich erhöhter Zeitaufwand, bis eine gewisser Schwellwert überschritten wird. Jedoch kommt es im Gegensatz zu einem einfacheren Aufbau




nicht so schnell zur Stagnierung der Trainingsergebnisse. Angewendet auf die gegebene Situation, bedeutet dies, dass einfachere Netze schneller in der Lage sind, ohne Kollision mit der Streckenbegrenzung, das Fahrzeug eine vollständige Runde auf einem Rundkurs fahren zu lassen, jedoch wird die Ideallinie nur selten verfolgt. Dies ändert sich auch trotz längerem Training nicht mehr deutlich; vielmehr wird das bestehende Fahrverhalten effizienter: die Kurven werden etwas zügiger durchfahren und auf Geraden wird eine höher maximale Geschwindigkeit erreicht. Bei Netzen mit *hidden layers* dauert es zu Beginn des Lernprozesses länger, bis eine vollständige Runde erfolgreich gefahren werden kann. Der Vorteil ist, dass sich bei andauerndem Training nicht nur die durchschnittliche Geschwindigkeit erhöht, sondern auch der gewählte Weg optimiert wird. Dieser Umstand lässt sich über folgenden Graphen visualisieren:

Dabei handelt es sich nicht um tatsächliche Daten. Die Graphik soll nur dazu dienen den Sachverhalt deutlicher darzustellen. Es bleibt festzuhalten, dass es keine grundsätzlich optimale Topologie zugeben scheint und die Entscheidung vielmehr auf Basis der gewünschten Lerngeschwindigkeit und maximal bestem Fahrverhalten getroffen werden muss. Empirisch hat sich ein neuronales Netz mit einer *hidden layer* als akzeptabler Kompromiss herausgestellt.

Vergleichbar mit der Entscheidung für die Topologie des Netzwerkes ist die Festlegung der Anzahl an Abstandssensoren. Auch hier muss ein Gleichgewicht zwischen ausreichend Umgebungsdaten und dem damit verbundenen Anstieg an zu

trainierenden Verbindungen im neuronalen Netz gefunden werden. Außerdem ist es intuitiv von Vorteil die Sensoren jeweils paarweise zu platzieren, sodass das *Sichtfeld* symmetrisch ist. So verdoppelt sich jedoch auch die Anzahl der Verbindungen von der Eingabeebene zur ersten *hidden layer*. Wie auch bei der Netzstruktur, löst man dieses Problem am besten durch Testen verschiedener Konfigurationen. Eine mögliche Lösung wäre die Verwendung von sechs[4] Sensoren, die jeweils in unterschiedliche Richtungen messen. Die Wahl dieser Sensoren ist allerdings nicht als ein allgemeingültiges Optimum für Simulationen dieser Art aufzufassen. Wie viele Sensoren das beste Gleichgewicht darstellen, hängt vor allem von der gewählten Netztopologie ab. Ein komplizierteres Netz, kann unter Umständen auch mit weniger Sensordaten ein akzeptables Fahrverhalten entwickelt, wohingegen ein simpler gestaltetes Netz diesen Umstand nicht ohne weiteres kompensieren kann. Deshalb ist es notwendig bei Veränderung der vorgeschlagenen Daten nicht nur die Topologie oder die Anzahl an Sensoren zu ändern, sondern immer beide zusammen.

## 4.4 Physikalische Berechnungen

Die eigentliche Simulation benötigt eine Komponente, die die physikalische Berechnung ausführt. Eine grundsätzliche Neuentwicklung ist hierbei nicht notwendig. Stattdessen kann die *Farseer Physics Engine* verwendet werden. Diese Bibliothek stellt Klassen und Methoden zur Verfügung, die Objekte im zweidimensionalen Raum realistisch bewegen können. Objekte werden dabei über ihre Form beschrieben. Die gewählte Karosserie lässt sich beispielsweise als Polygon darstellen. Des Weiteren ist es möglich unterschiedliche Objekte über *Joints* oder zu Deutsch *Gelenke* zu verbinden. So können die Reifen relativ zu der Karosserie fixiert werden und sich nur in ihrer Ausrichtung ändern.

Die physikalische Berechnung läuft über Zeitschritte ab. Das bedeutet das keine kontinuierliche Berechnung erfolgt, sondern vielmehr immer nach einem bestimmten Intervall. Die verwendete Engine stellt hierfür die *Step*-Funktion bereit. Als Parameter wird die verstrichene Zeit seit dem letzten Aufruf übergeben. Da eine Echtzeitausführung in der Simulation nicht notwendig ist, kann auch ein fiktiver, zu hoher Wert übergeben werden. Dies hat die Folge, dass die Simulation bedeutend schneller abläuft, als dies in der Realität der Fall ist. So kann ein Fahrzeug, ein

[4] Relativ zur Fahrrichtung: -40°, -20°, -4°, 4°, 20°, 40°

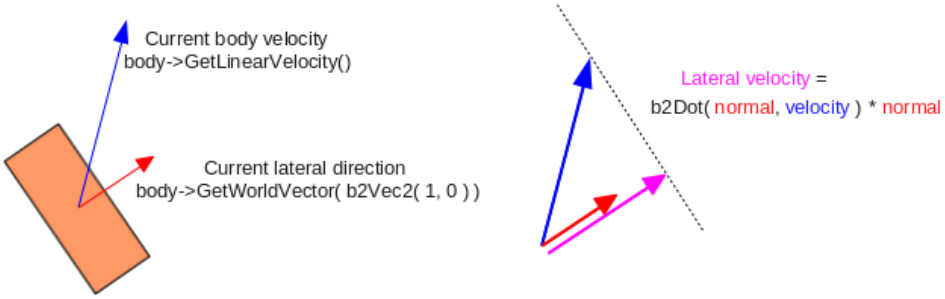
trainierte Fahrverhalten vorausgesetzt, innerhalb weniger Sekunden eine vollständige Runde auf dem Nürburgring fahren; dies würde in der Realität mehrere Minuten dauern. Der Vorteil dabei ist, dass so die Simulation schneller abläuft und sich gute Lösungen zeitlich schneller von schlechten abgrenzen. Der übergebene Wert kann jedoch nicht beliebig hoch gewählt werden. Ab einem gewissen Punkt kann es passieren, dass das Fahrzeug durch eigentlich unpassierbare Streckenbegrenzungen navigieren kann. Dies liegt daran, dass beim Aufruf der *Step*-Funktion zunächst die Position und Rotation der Objekte in der Simulation anpasst und erst im Anschluss eine Kollisionserkennung erfolgt. Ein anschauliches Beispiel ist die Flugbahn einer Pistolenkugel. Befindet sich die Kugel in einem gegebenen Zeitschritt noch wenige Millimeter vor einem Stück Papier, so wird sie, falls der Zeitschritt zu groß ist, anschließend bereits hinter dem dünnen Papier sein. Für die Physikengine existieren nur die beiden Situationen, getrennt voneinander. Dass die Kugel um von der ersten Position zu er zweiten zu gelangen, dass Blatt Papier hätte durchdringen müssen, ist eine Tatsache, die so ohne weiteres nicht berücksichtigt wird. Somit kommt es zu keiner Kollision und das Papier bleibt intakt. Angewendet auf die Fahrsimulation, bedeutet das, dass Fahrzeuge bei zu großem Zeitschritt Streckenbegrenzungen durchdringen können ohne das eine Kollision erkannt wird. Da ein Kontakt mit einer Begrenzung allerdings fatale Folgen in der Realität haben könnte, muss dies von der Simulation entsprechend erkannt und verarbeitet werden können. Es gibt Ansätze wie so eine Situation verhindert werden kann, jedoch sind diese relativ Prozessorzeitintensiv und sind so keine sinnvolle Option. Stattdessen muss das Zeitintervall so gewählt werden, dass es nicht zu dieser Problematik kommen kann. Dieses Intervall hängt auch von der Größe des Autos und der Dicke der Streckenbegrenzung ab. Demnach muss das Intervall situationsspezifisch gewählt werden. In der Fahrzeugsimulation kommt es bei Tests mit einem Intervall von 100 Millisekunden zu keinen Problemen.

### 4.4.1 Fahrzeugphysik

Welche Aufgabe die Physikengine allerdings nicht übernimmt, ist das berechnen der Fahrzeugphysik. Zwar werden die Kräfte von den Reifen auf die Karosserie übertragen und so auch auf die anderen Reifen, jedoch wird die Reibung der Reifen ohne weiteres nicht berücksichtigt. So ist es beispielsweise möglich, dass Fahrzeug sich auf der Stelle dreht und die Reifen über den Boden gleiten oder aber dass beim

Lenken das Fahrzeug schlichtweg weiter geradeaus fährt. Es gibt keine Möglichkeit der Physikengine eine Reifenkomponente hinzuzufügen.

Um die Besonderheiten bei der Fahrzeugphysik zu berücksichtigen, werden nach jedem Aufruf der *Step*-Funktion eine Reihe zusätzliche physikalische Berechnungen durchgeführt. Als erstes wird die relative Geschwindigkeit der einzelnen Reifen angepasst. Dazu wird ein Impuls entgegen der lateralen Geschwindigkeit angewendet, dessen Stärke genau so groß ist, sodass jegliche laterale Bewegung neutralisiert wird. Folgende Abbildung verdeutlicht den Ablauf:




Die Formel für den Impuls lautet wie folgt: [9][5]

$$Impulse = -Dot(normal, velocity) * normal * wheelmass$$

Da die Simulation aus der *Top-Down*-Sicht betrachtet wird, muss zusätzlich verhindert werden, dass die Reifen sich ohne Lenkvorgang neu ausrichten. Insbesondere bei höheren Geschwindigkeiten, wo größere Kräfte auf das Fahrzeug einwirken, kann es sonst passieren, dass das Auto ins Schlingern gerät oder vollständig die Kontrolle verliert. Um dies zu verhindern wird ein Drehimpuls gegeben, der eine eventuell vorhandene Rotation ausgleicht. Dazu wird die *Trägheit*[6] der Reifen mit der negativen Winkelgeschwindigkeit multipliziert und als Impuls angewendet. Diese beiden Impulse ermöglichen, dass sich das Fahrzeug realitätsnah steuern lässt. Zunächst muss jedoch das Auto beschleunigt werden können. Dies wird realisiert, indem die Position des Gaspedals als Wert im Intervall [0,1] interpretiert wird. Das neuronale Netz besitzt als Ausgabe einen Geschwindigkeitsfaktor der genau in dieses Intervall fällt. Dieser Faktor wird anschließend mit der maximalen Geschwindigkeit des Fahrzeugs multipliziert. Ist das Ergebnis größer als die aktuelle Geschwindigkeit soll das Fahrzeug beschleunigt, ist sie hingegen kleiner abgebremst werden. Die maximale Kraft die der Motor

[5] *Dot* ist dabei das Skalarprodukt zweier Vektoren. Sowohl der Normalvektor als auch der Geschwindigkeitsvektor können von der Physikengine erfragt werden.
[6] Die Trägheitseigenschaft eines Objekts wird von der Physikengine festgelegt und kann ausgelesen werden.

erreichen kann, wird im Falle einer Beschleunigung auf die entsprechenden Reifen, je nach Antriebsart, gleichmäßig aufgeteilt. In der Simulation wird Heckantrieb verwendet, jedoch können prinzipiell auch Allrad- und Vorderantrieb genutzt werden. Die Kräfte werden jeweils entlang der Ausrichtung der Reifen angewendet. Soll anstatt zu beschleunigen gebremst werden, wird die maximale Motorkraft mit 1,5 multipliziert und entgegen der aktuellen Bewegungsrichtung angewendet. Dies hat den Hintergrund, dass Fahrzeuge in der Realität meist stärker bremsen, als beschleunigen können.

Es kann passieren, dass die Umgebungsdaten bei Simulationsbeginn bei einer bestimmten Netzwerkgewichtung als Folge haben, dass der Geschwindigkeitsfaktor nahezu null ist. In diesem Fall würde das Fahrzeug kaum beschleunigen und die Umgebungsdaten sich auch kaum ändern. So kann praktisch nicht untersucht werden, ob die Lenkeigenschaften des neuronalen Netzes im Gegensatz zum Geschwindigkeitsfaktor unter Umständen akzeptable Ergebnisse erzielen würden. Deshalb wird ein minimaler Geschwindigkeitsfaktor vorgegeben. In der Simulation haben die Fahrzeuge eine Höchstgeschwindigkeit von 300 Kilometer pro Stunde; eine minimale Geschwindigkeit von 30 Kilometer pro Stunde erscheint akzeptabel. Aus diesem Grund wird der minimale Geschwindigkeitsfaktor auf 0,1 festgelegt. Das bedeutet, dass selbst wenn das neuronale Netzwerk eigentlich null als aktuelle Zielgeschwindigkeit besitzt, das Fahrzeug trotzdem auf 30 km/h beschleunigt. So wird gewährleistet, dass auch bei suboptimaler Reaktion auf die Umgebung in Bezug zur Geschwindigkeit, trotzdem das Lenkverhalten untersucht werden kann. Diese Einschränkung hat zur Folge, dass besonders zu Beginn der Simulation bedeutend schneller brauchbare Ergebnisse erzielt werden. Intuitiv wird so zunächst vor allem das Lenkverhalten und erst anschließend die Geschwindigkeit optimiert.

Nachdem die physikalische Simulation der Reibung der Reifen und der Geschwindigkeitsänderung erfolgt ist, wird nun als nächster Schritt das Lenken realisiert. Um das Drehen des Lenkrades zu simulieren, wird eine maximale Drehgeschwindigkeit eingeführt. Empirisch wurde der Wert 300 Grad pro Sekunde festgelegt. Wobei das Lenkradausrichtung eins zu eins der Reifenausrichtung entspricht: ein Grad Lenkradänderung bedeutet ein Grad Reifenausrichtungsänderung. Das neuronale Netzwerk gibt über das zweite Ausgabeneuron an, welche Auslenkung erzielt werden soll. Der Zielwinkel errechnet sich demnach über die Formel:

$$Winkel = Faktor * Maxwinkel * Einschränkung$$

Der maximale Winkel beträgt zwölf Grad. Und somit können die Reifen eine Ausrichtung von -12 bis +12 Grad erreichen. Der E*inschränkung*-Faktor wird dynamisch berechnet und hängt von der aktuellen Geschwindigkeit des Fahrzeuges ab. Er liegt stets im Intervall $[0,1]$. Die Motivation hinter diesem Faktor ist die Einschränkung der Lenkausrichtung bei hohen Geschwindigkeiten. Ansonsten wäre es möglich mit 300 km/h die engsten Kurven der Rennstrecke ohne Kontakt mit der Streckenbegrenzung zu durchfahren. In der Realität verhindert vor allem Haftungsverlust der Reifen und hohe Fliehkräfte ein solches Fahrverhalten. Durch den beschriebenen Faktor wird jedoch, mit weniger rechenintensiven Operationen, ein vergleichbares Ergebnis erzielt.

Anschließend wird die Differenz zwischen aktuellem Winkel der Vorderreifen und dem gewünschten Winkel berechnet. Falls diese Differenz größer als die maximal zulässige Winkeländerung ist, wird sie entsprechend eingeschränkt und daraufhin mit dem aktuellen Winkel der Reifen addiert. Das Ergebnis stellt die neue Winkelausrichtung dar und wird mittels der *Joints*, mit denen die Reifen an der Karosserie befestigt sind, eingestellt.

Als letzter Schritt wird allgemeine Reibung der Reifen und Luftwiderstand berechnet. Dazu wird eine Kraft auf die Karosserie angewendet, die entgegen der aktuellen Fahrtrichtung wirkt. Diese Kraft ist abhängig von der Geschwindigkeit. Je schneller sich das Fahrzeug bewegt, desto höher ist auch der Luftwiderstand. Nachdem alle physikalischen Berechnung durchgeführt beginnt der Ablauf von Neuem.

## 4.5 Simulationsablauf

Die Simulation läuft zyklisch ab. Bevor sie beginnt werden die Fahrzeuge auf die Startposition einer ausgewählten Rennstrecke platziert. Nach dem Start der Simulation werden immer wieder dieselben Schritte ausgeführt. Zunächst wird die Umgebung aus Sicht eines Fahrzeuges mittels der Sensoren erfasst. Diese Werte werden dann an das jeweilige künstliche neuronale Netz weitergegeben und die beiden Ausgabewerte berechnet. Der erste Wert wird nun in die Zielgeschwindigkeit umgerechnet der zweite Wert in die Zielausrichtung der Vorderreifen. Über die eben beschriebenen Verfahren werden diese Ergebnisse dann entsprechend in der physikalischen Simulation angewendet. Die Kräfte der Luft- und Reifenwiderstände

werden berechnet und ebenfalls berücksichtigt. Nachdem die Kräfte nun an den entsprechenden Komponenten der Simulation wirken, wird die *Step*-Funktion aufgerufen und die Engine errechnet den neuen Zustand der Simulation. Nun wird wieder die Umgebung über die Sensoren erfasst der der Ablauf beginnt von vorne.

Falls jedoch das Fahrzeug mit einer Streckenbegrenzung kollidiert oder die Simulation länger als zwei Minuten läuft[7] wird die Simulation für dieses Fahrzeug abgebrochen und das Fahrverhalten wird bis zu diesem Punkt bewertet. Dies geschieht über die zurückgelegte Strecke. Um zu messen wie viele Meter das Auto tatsächlich auf der Rennstrecke zurückgelegt hat, werden in regelmäßigen Abständen Sensorlinien auf der Strecke platziert. Beim Überfahren wird die Kollision registriert und ein Zähler inkrementiert. In der Simulation werden diese Sensoren alle zehn Meter angebracht. Bei einer Streckenlänge von einem Kilometer entspricht das genau hundert Sensoren. Soll das Fahrverhalten nun bewertet werden wird der Wert der Zählvariable durch die Gesamtzahl aller Sensoren auf der Strecke dividiert. Dies hat den Hintergrund, dass die Bewertung von verschiedenen Strecken vergleichbar sein soll. Schafft ein Fahrzeug auf einer Strecke eine vollständige Runde und auf einer anderen Strecke ebenso, dann sollte auch die Bewertung gleich sein. Würde die Zählvariable direkt als Bewertung verwendet werden, würden bei längeren Strecken die Bewertung stark von der bei kürzeren Strecken abweichen. Verschiedene Strecken sind unterschiedlich schwierig zu durchfahren, aber so kann trotzdem immer erkannt werden, wenn eine komplette Runde gefahren wurde; nämlich genau dann wenn die Bewertung größer/gleich eins ist.

### 4.5.1 Berechnung der nächsten Generation

Kam es bei allen Fahrzeugen zu einer Kollision oder sind zwei Minuten in der Simulation verstrichen, werden die künstlichen neuronalen Netze anhand ihrer erreichten Bewertung sortiert. Die besten neuronalen Netze werden unverändert in den nächsten Simulationsablauf übernommen. Die verbleibenden Plätze werden mit einer gewichteten Zufallsauswahl festgelegt. Die Bewertungen aller Netze werden addiert. Anschließend wird eine Zufallsvariable zwischen null und dem errechneten Wert generiert. Nun werden die Bewertungen von dem besten Netz zu dem

[7] zur Erinnerung: ein Simulationsschritt wird immer 100 Millisekunden weitergerechnet, unabhängig davon wie viel Zeit für die Berechnung wirklich vergangen ist. Diese zwei Minuten beziehen sich also nicht auf *echten* zwei Minuten, sondern auf die Zeit der Simulation, die in aller Regel bedeutend schneller vergeht als die reale Zeit. Eine äquivalente alternative Darstellung wäre, dass die Simulation immer höchstens 1200 Iterationen besitzt.

schlechtesten nach und nach überprüft. Ist die aktuelle Bewertung addiert mit allen bereits überprüften Bewertungen größer als die generierte Zufallszahl, dann wird das aktuelle neuronale Netz in die nächste Simulationsiteration übernommen. Dieses Verfahren ist unter dem Namen *Roulette-Selection* oder *Propotional-Selection* [10] bekannt und stellt ein Selektionsverfahren aus dem Themengebiet der evolutionären Algorithmen dar. Der Vorteil bei dieser Selektionsvariante in dieser Situation ist, dass auch neuronale Netze ausgewählt werden die unter Umständen zwar keine unmittelbar hohe Bewertung erreicht haben, aber eventuell nur sehr knapp eine bedeutend höhere Bewertung verfehlt haben, indem sie vielleicht nur eine Kurve am Anfang der Strecke ein wenig zu eng durchfahren haben. Das berücksichtigen solcher Netze erlaubt eine größere genetische Diversität in der Population und so eine Vielzahl verschiedener Ansätze für das optimale Fahrverhalten. Außerdem werden so lokale Optima vermieden, da Kandidaten einer größeren *Fläche* des Lösungsraums berücksichtigt werden.

Die neuronalen Netze die über diese gewichtete Zufallsauswahl selektiert wurden, werden zunächst leicht abgeändert oder *mutiert*. Dazu werden die Gewichte der einzelnen Verbindungen leicht verändert. Wie groß diese Veränderung ausfällt ist ebenso zufällig, wie die Auswahl welche Verbindung verändert werden soll. Ist eine Verbindung für eine *Mutation* ausgewählt worden, wird die Distanz der Veränderung über eine Normalverteilung errechnet. Der Vorteil bei diesem Verfahren liegt darin, dass es in der Regel nur zu kleinen Veränderungen der Gewichte kommt und so vergleichbar mit dem Hill-Climb-Algorithmus sich die Bewertung der neuronalen Netze auf ein (lokales) Optimum zu bewegt. Auf der anderen Seite kommt es gelegentlich auch zu größeren Veränderungen. So können lokale Optima überwunden werden und neue Lösungsansätze generiert werden. Beispielsweise könnte es sein, dass zunächst die beste Lösung zwar recht gut die Kurven durchsteuern kann, jedoch immer mit maximaler Geschwindigkeit fährt. Die kleinen Veränderungen würden auf lange Sicht gesehen, das Kurvenverhalten weiter verbessern, jedoch bedarf es einer großen Veränderung des Geschwindigkeitsverhaltens um auch schärfere Kurven ohne Kontakt mit der Streckenbegrenzung durchfahren zu können[8]. Diese großen Anpassungen treten aufgrund der Verteilung der Zufallszahlen in einer Normalverteilung nur selten auf,

[8] Erinnerung: bei hohen Geschwindigkeiten ist das Lenkverhalten stark eingeschränkt.

deshalb bleibt der Hauptfaktor hinter der Verbesserung der Lösungen kleine iterative Anpassungen.

Für die Definition der Normalverteilung wird die Standardabweichung oder die Varianz benötigt. Ebenso muss die Wahrscheinlichkeit festgelegt sein, mit der ein Verbindungsgewicht verändert werden soll. Falls immer nur eine festgelegte Anzahlt mutiert werden, können bestimmte Optima nur schwer überwunden werden. Am Beispiel das stets nur eine Verbindung mutiert werden soll wird dies intuitiv klar. So muss das Kurvenverhalten angepasst werden, wenn die Geschwindigkeit reduziert wird, ansonsten führt die veränderte Geschwindigkeit und die damit einhergehende geänderte Lenkmöglichkeit dazu, dass Kurven nun beispielsweise zu eng gefahren werden und es so mit der Innenwand zur Kollision kommt. Die benötigte gleichzeitige Veränderung mehrerer Gewichte stellt die Motivation hinter einer Variablen Anzahl an Mutationen dar.

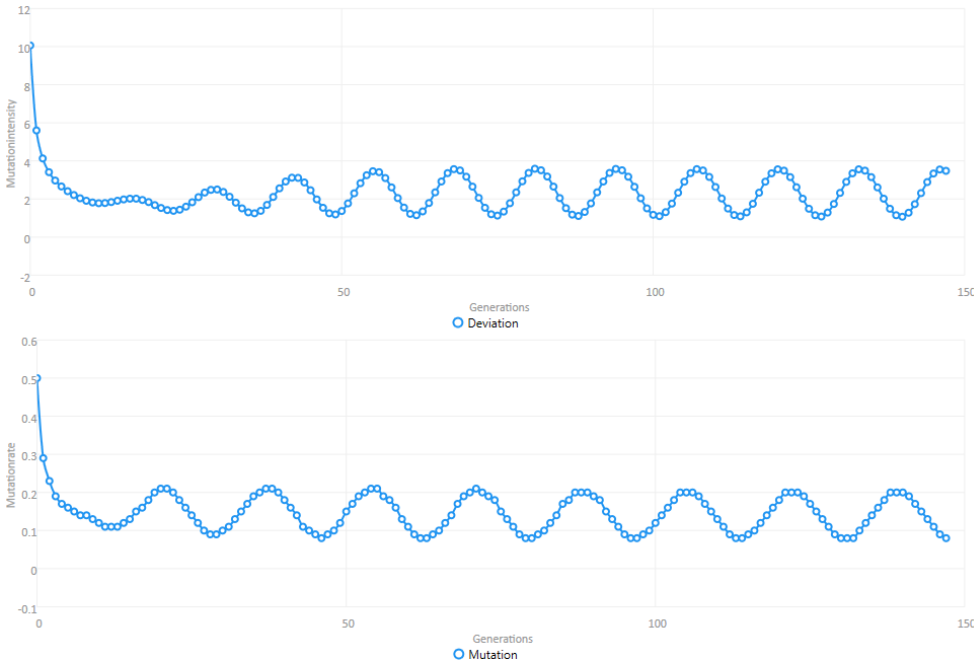
#### 4.5.1.1 Mutation

Wie groß die Chance auf Mutation pro Gewicht sein sollte, hängt auch von der Standardabweichung ab. In verschiedenen Situationen müssen verschiedene Kombinationen gewählt werden. Ist ein neuronales Netz bereits recht erfolgreich bedarf es nur noch kleinerer Änderungen um das Ergebnis weiter zu verbessern. Befinden sich alle Lösungskandidaten jedoch in einem lokalen Optimum muss es zu recht großen Veränderungen kommen um brauchbare Alternativen zu finden. Hierbei sollte die Anzahl der Gewichtsänderungen variabel sein. Vielleicht reicht eine einzige, dafür recht drastische Gewichtsänderung bereits, oder es müssen gleichzeitig viele Gewichte angepasst werden. Welche Mutationsrate mit welche Mutationsintensität das beste Ergebnis für eine gegebene Situation liefert, lässt sich schwer über festgelegte Regeln definieren. Für einen menschlichen Betrachter mag vollkommen klar sein, dass sich die aktuellen Fahrverhalten in einem lokalen Optimum befinden und dementsprechend große Veränderungen benötigt werden, um dies zu überwinden. Diese Beobachtung algorithmisch zu errechnen ist jedoch sehr schwer. Die einfachste Lösung wäre dieses Problem mehr oder weniger zu ignorieren und feste Werte für Mutationsrate und -intensität zu wählen. Da die Auswahl stets auf Wahrscheinlichkeiten basiert, wird die benötigte Kombination, um das lokale Optimum zu verlassen, auf lange Sicht irgendwann eintreten. Empirisch hat sich ein alternativen Verfahren als überlegen herausgestellt. Dabei werden die beiden Variablen zyklisch verändert. Der Verlauf der Funktion ist der

einer Sinuskurve. Jedoch haben beiden Variablen eine unterschiedliche Periodenlänge. So wird erreicht, dass nahezu jede Kombinationsmöglichkeit erreicht werden kann. Nach jeder Simulationsiteration wird die Mutationsrate und -intensität anhand dieser Idee verändert. Die Amplitudenhöhe hängt von der Anzahl der Simulationsiterationen ab, die seitdem das beste Ergebnis das letzte Mal überboten wurde, vergangen sind. So wird versucht auf lokale Optima entsprechend zu reagieren und eine größere Veränderung wahrscheinlicher zu machen. Als Kurve skizziert ergibt sich folgende Abbildung:

Zunächst wird mit relativ großen Werten begonnen, um eine große Variabilität zwischen den neuronalen Netzen sicherzustellen. Anschließend beginnen sich die Werte der beiden Variablen entsprechend dem beschriebenen Verfahren zu verändern. Man sieht recht deutlich wie die unterschiedliche Periodenlänge der beiden Kurven dazu führt, dass ein Großteil der Kombinationsmöglichkeiten auch erreicht wird. Als Periodenlänge bieten sich zwei teilerfremde Zahlen an, da so das kleinste gemeinsame Vielfache der triviale Fall der Multiplikation der beiden Zahl ist und so eine große Anzahl an Kombinationen eintritt bevor es zur Wiederholung kommt. Primzahlen sind grundsätzlich immer teilerfremd, deshalb eignen sich die Zahlen 5 und 7 als Periodenlängen.

Nachdem nun alle Ansätze vorgestellt wurden, können diese nun zusammengeführt werden. So ergibt sich ein Rahmen mit der die Ausgangsfrage beantwortet werden




kann. Um die Auswertung zu erleichtern, bietet es sich zunächst jedoch an ein

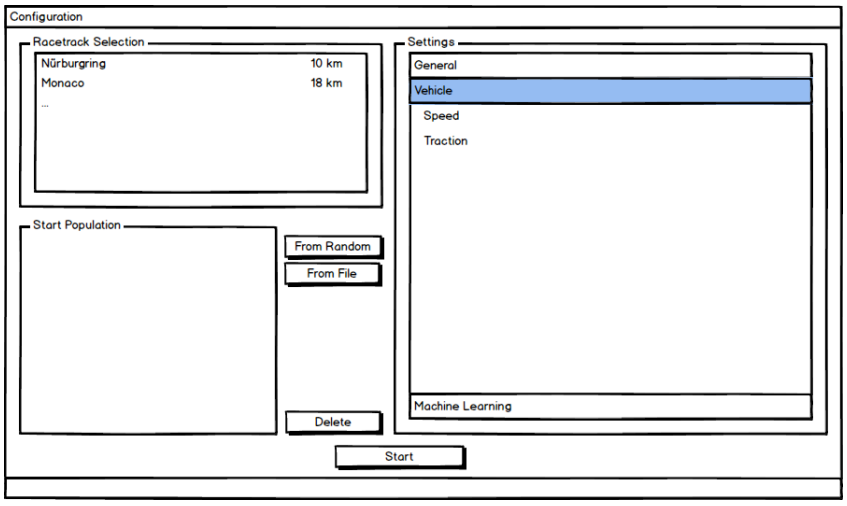
Hilfsprogramm zu schreiben, welches die beschriebenen Verfahren implementiert und visuell aufbereitet. So wird nicht nur die korrekte Beantwortung der Fragestellung ermöglicht, sondern es lassen sich auch weitere Schlussfolgerungen ziehen. Im Folgenden wird der Entwurf eines solchen Hilfsprogramms vorgestellt.

# 5 Entwurf

Bevor das Hilfsprogramm implementiert werden kann, sollte zunächst mit Methoden der Softwaretechnik ein Entwurf angefertigt werden. Eine Anforderungsdefinition entfällt in diesem Fall jedoch, da die typische Klient-Softwareentwickler-Beziehung nicht gegeben ist. Auch ohne dieses Hilfsprogramm könnte die Ausgangsfrage beantwortet werden, somit sind praktisch alle Anforderungen optional und dienen nur zur besseren Visualisierung der Ergebnisse. Das Programm wird in C# implementiert. Als Darstellungsframework kommt die *Windows Presentation Foundation (WPF)* zum Einsatz. Die Visualisierung der Ergebnisse wird mit dem OpenGL-Wrapper *OpenTK* umgesetzt. Zunächst wird das Programm aus Benutzersicht und anschließend aus technischer Sicht entworfen.[9]

## 5.1 Benutzersicht

Beim Programmstart soll es zunächst möglich sein, eine Reihe von Einstellungen festzulegen, sowie eine der geladenen Rennstrecken auszuwählen und zu entscheiden wie sich die Startpopulation der neuronalen Netze zusammensetzt. Zu diesem Zweck wird ein Fenster dargestellt indem die entsprechende Konfiguration eingestellt werden kann. Eine mögliche Visualisierung wäre zum Beispiel folgende:

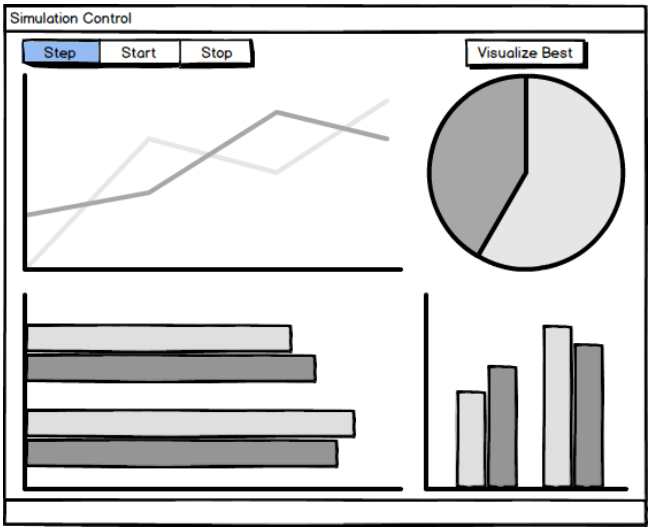



Zu den Einstellungsmöglichkeiten gehören unter anderem die maximale Geschwindigkeit und die Beschleunigung des Fahrzeugs, sowie Mutationsrate und

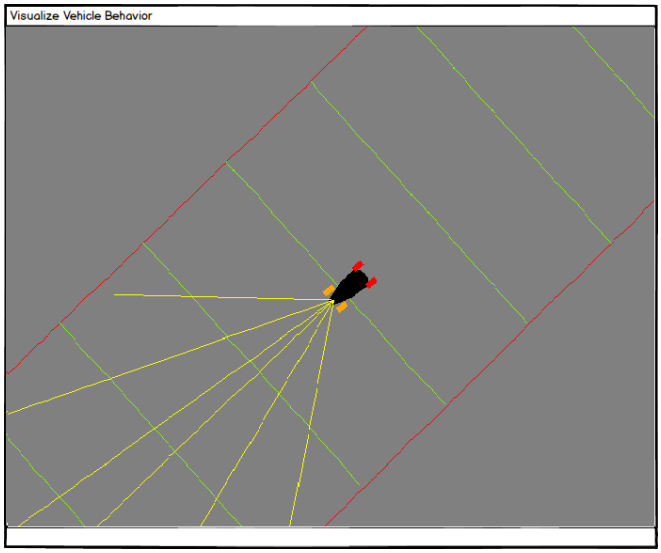
[9] Benutzersicht meint die optischen Komponenten des Hilfsprogramms, technische Sicht hingegen den Aufbau der Klassen und deren Beziehung untereinander.

Anzahl an Sensoren. Prinzipiell können alle Variable eingestellt werden, die nicht aus technischen Gründen auf einen festen Wert gesetzt werden müssen. [10]

Nachdem der Benutzer die Konfiguration abgeschlossen hat, wird über einen Klick auf den Start-Button das Auswertungsfenster geöffnet. In diesem Fenster kann die Simulation gestartet und gestoppt werden, sowie eine Reihe an Informationen abgerufen werden, die den aktuellen Stand und den Verlauf der Simulation widerspiegeln. Eine Form der möglichen Visualisierung ist folgende:




Über den Button *Visualize Best* kann das momentan bestbewertete Fahrverhalten dargestellt werden. Dazu wird in einem neuen Fenster in Echtzeit ein Fahrzeug mit dem entsprechenden neuronalen Netz visualisiert.




[10] Hiermit sind sogenannte *Magic Numbers* gemeint. Variablen dessen Werte sich nicht errechnen lassen, sondern manuell festgelegt werden müssen.

Unter Umständen bietet es sich an auch einzelne Individuen zu analysieren und das Fahrverhalten manuell zu untersuchen. Deshalb wird, nachdem der Benutzer ein Individuum in einem der anderen Diagramme ausgewählt hat, eine Auswertung für ein konkretes Individuum dargestellt. Diese Auswertung beinhaltet zurückgelegte Strecke, Durchschnittsgeschwindigkeit, Errechnete Bewertung und die Zeit, die verstrichen ist, bevor es zum Abbruch der Simulation kam. Außerdem wird das neuronale Netzwerk mit den Gewichten des ausgewählten Individuums dargestellt. In dieser Übersicht können dann auch manuell ausgewählte Individuen der Population gespeichert und visualisiert werden. Da eines der Evaluationskriterien die subjektive Beurteilung des Fahrverhaltens ist, kann über diesen Weg die Veränderung des Fahrverhaltens im Laufe des Trainingsprozesses protokolieret und analysiert werden.

Welche Daten die anderen Diagramme widerspiegeln und wie diese konkret visualisiert werden, steht im engen Zusammenhang mit dem technischen Entwurf. Aus diesem Grund kann auf die Diagramme erst im Anschluss an die Erläuterung dessen eingegangen werden.

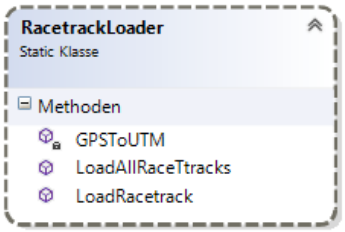
## 5.2 Technischer Entwurf

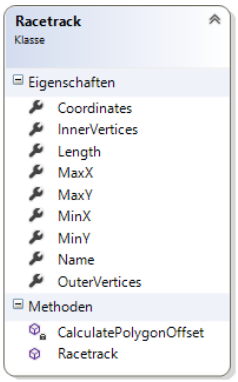
### 5.2.1 Streckendaten

Wie im Konzept beschrieben, liegen die Streckendaten als GPS-Koordinaten vor. Sie werden im XML-Format bereitgestellt und können so recht simpel programmiertechnisch ausgelesen werden. Eine heruntergeladene Datei hat folgenden Aufbau:

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <track name="Nürburgring">
3.    <trkpt lat="50.33818" lon="6.94986"/>
4.    <trkpt lat="50.33884" lon="6.9491"/>
5.    <trkpt lat="50.33928" lon="6.9485"/>
6.    <trkpt lat="50.33928" lon="6.94812"/>
7.    <trkpt lat="50.33906" lon="6.94778"/>
8.    <trkpt lat="50.3385"  lon="6.94724"/>
9.    <trkpt lat="50.33799" lon="6.94667"/>
10.   <trkpt lat="50.3376"  lon="6.94583"/>
11.   <trkpt lat="50.33729" lon="6.94485"/>
12.   <!-- ... -->
13. </xml>
```

Um diese GPS-Koordinaten auf eine zweidimensionale Ebene zu projizieren, müssen sie zunächst konvertiert werden. Dies geschieht über die Klasse *RacetrackLoader*. Diese Klasse bietet die Möglichkeit, alle Strecken eines Ordners auf der Festplatte zu laden und in *Racetrack*-Objekten zu repräsentieren. Ein *Racetrack* besitzt demnach nur noch die nutzbaren zweidimensionalen Koordinaten. Bei der Erzeugung des *Racetrack* wird außerdem das innere und äußere Polygon als Streckenbegrenzung berechnet (siehe Konzept). Außerdem wird der kleinste und größte *x*- und *y*-Wert aller Koordinaten gespeichert, um die Rennstrecke korrekt visualisieren zu können (Verschiebung der Kamera zu den entsprechenden Koordinaten).




### 5.2.2 Simulationseinstellungen

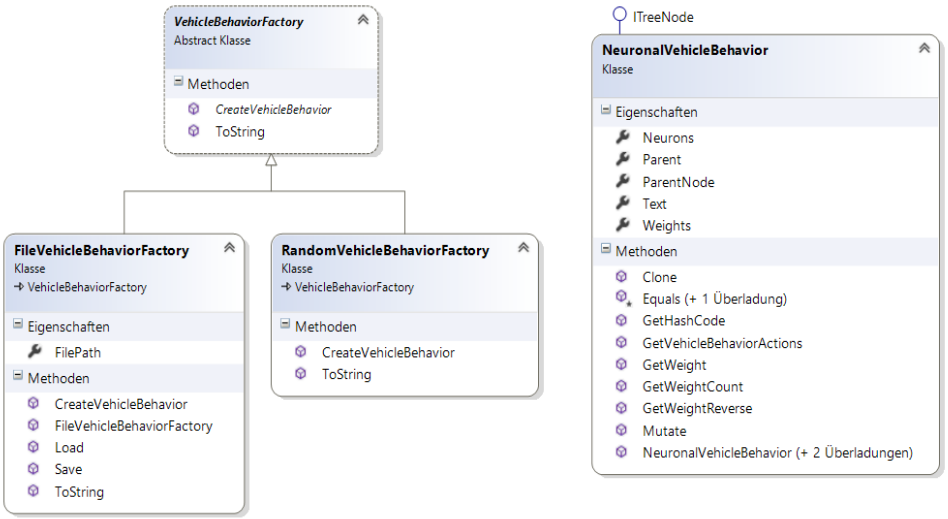
Wie im Kapitel 5.1 bereits beschrieben, verfügt das Hilfsprogramm über eine Reihe an Einstellungen, die vom Benutzer angepasst werden können. Um die Größe der benötigten Klassen möglichst klein zu halten, werden die Einstellungen, angelehnt an ihre Gruppierung in der Benutzeroberfläche, in Unterklassen aufgeteilt. Es gibt eine verwaltende Klasse, die das Laden und Speichern der Einstellungsdatei auf der Festplatte übernimmt und die Unterklassen, sowie allgemeine Einstellungen der Simulation kapselt. Trotz der Aufteilung sind die Klassen immer noch recht umfangreich. Deshalb wird an dieser Stelle auf eine graphische Repräsentation verzichtet. Grundsätzlich ist der Aufbau, wie man ihn im Hinblick auf die einstellbaren Eigenschaften in der Benutzeroberfläche erwartet. Jede Einstellung entspricht einem Attributfeld in der entsprechenden Klasse.

Damit die dargestellten Werte der Einstellungen in der Benutzeroberfläche konsistent mit den gespeicherten Werten bleiben, kommt Databinding zum Einsatz. Das Darstellungsframework *WPF* bietet die Möglichkeit mithilfe von Events bei Veränderungen in der Oberfläche direkt die Werte in den gebundenen Attributfeldern entsprechend anzupassen. Aber auch die andere Richtung wird unterstützt. So wird beim Neuladen der Einstellungsdatei automatisch die Oberfläche mit den korrekten Werten aktualisiert. Wie genau diese Databinding Funktionalität genutzt werden kann, findet sich im Anhang. Wichtig ist anzumerken,

dass Textfelder in der Benutzeroberfläche ausschließlich Werte vom Typ *String*, also eine Zeichenfolge, annehmen können. Viele Einstellungen sind jedoch Fließkommazahlen und müssen demnach zunächst umgewandelt werden. *WPF* konvertiert eine Reihe an Datentypen automatisch, allerdings scheitert der Automatismus an selbstdefinierten Repräsentationen. Eine solche Repräsentation findet sich bei der Einstellung über die Anzahl der Neuronen in den *Hidden-Layers*. Die gewählte Darstellung trennt die Anzahl der Neuronen jeweils mit einem Komma voneinander. Ein neuronales Netz mit drei *Hidden-Layers* mit jeweils sechs Neuronen würde beispielsweise folgendermaßen in der Benutzeroberfläche dargestellt werden: *6,6,6*. Diese Darstellung kann nicht unmittelbar konvertiert werden und eine Umwandlung muss manuell erfolgen. Die genauen Details sind für das Verständnis nicht weiter notwendig, dieses Beispiel sollte nur verdeutlichen, dass unter Umständen der Quellcode an einigen Stellen komplexer erscheinen mag, als es auf den ersten Blick nötig getan hätte, diese Beobachtung jedoch nicht korrekt ist.

Die Einstellungsdatei wird mithilfe der XML-Serialisierung gespeichert und geladen. Damit wird die Struktur der Klassen in ein XML-Schema übertragen und die Werte als XML-Attribute gespeichert. Beim Programmstart wird die Einstellungsdatei geladen. Falls noch keine vorhanden ist, wird sie automatisch mit Standardwerten erstellt. Der Benutzer kann mit einem Klick auf den *Save*-Button die Einstellungsdatei überspeichern. Beim nächsten Programmstart werden dann die veränderten Einstellungen geladen.

Die Auswahl der Rennstrecke wird, wie auch bei den Einstellungen, über Databinding realisiert. Die aktuell ausgewählte Rennstrecke in der



Benutzeroberfläche wird automatisch in einem Attributfeld in der entsprechenden Klasse gespeichert. Bevor der Benutzer die eigentliche Simulation starten kann, muss eine Rennstrecke ausgewählt werden. Außerdem muss vor dem Start die Ausgangspopulation konfiguriert werden. Hierzu kommt das *Factory-Pattern* zum Einsatz:

Dem Benutzer stehen zwei *Factories* zur Verfügung. Zum einen kann angegeben werden, dass ein bereits trainiertes neuronales Netz von der Festplatte geladen werden und in die Startpopulation übernommen werden soll. Als zweite Option kann ein neuronales Netzwerk auch zufällig auf Basis der gewählten Einstellungen erzeugt werden. Dabei werden die Gewichte mit zufälligen Werten initialisiert. Die Basisklasse *VehicleBehaviorFactory* besitzt die abstrakte Methode *CreateVehicleBehavior*, die von der *FileVehicleBehaviorFactory* und der *RandomVehicleBehaviorFactory* implementiert werden. Der Rückgabewert ist dabei ein *NeuronalVehicleBehavior*. Nach der Erzeugung des *NeuronalVehicleBehavior* existiert kein Zusammenhang mehr mit der *Factory*. So können neuronale Netze, die aus einer Datei gelesen wurden genauso mutiert werden, wie dies bei zufällig erzeugten Netzen der Fall ist. Bei dem Laden eines neuronalen Netzwerkes aus einer Datei ist es wichtig darauf zu achten, dass das Netzwerk im richtigen Format vorliegt. So kann es sein, dass zu einem früheren Zeitpunkt mehr *Hidden-Layers* in den Einstellungen festgelegt worden sind, als dies in der aktuellen Ausführung des Programms der Fall ist. Deshalb kann es vorkommen, dass es eine unterschiedliche Anzahl an Gewichten und Eingabeneuronen gibt, was zur Laufzeit zu Fehlern führen wird.

Der Aufbau der *NeuronalVehicleBehavior*-Klasse ist so gewählt, dass der Wert an den Neuronen auch der Berechnung von Eingabedaten in Ausgabedaten persistiert. So kann in der Benutzeroberfläche gegebenenfalls eine Echtzeitdarstellung der einzelnen Verbindungen und Neuronen erfolgen, um die, durch Training erreichten, Gewichte besser analysieren zu können. Zu diesem Zweck werden die Neuronen als zweidimensionaler, verzweigter[11] *Array* repräsentiert. Dies hat den Hintergrund, dass in jeder Ebene unterschiedlich viele Neuronen vorhanden sein können und demnach bei einem normalen zweidimensionalen *Array* einige *Array*-Elemente leer stehen würden. Die Gewichte werden als eindimensionaler *Array* gespeichert. Die

[11] *Verzweigt* oder Englisch *jagged* bedeutet bei *Arrays*, dass sie unterschiedliche Anzahl an Spalten oder Zeilen haben können. Anschaulich handelt es sich bei ihnen um *Arrays* von *Arrays*.

Zuordnung von Gewicht zu Verbindung erfolgt über eine fortlaufende Nummer. Eine alternative Darstellungsweise wäre, die Gewichte als eigene Klasse zu repräsentieren und dort die Position von Ausgangs- zu Zielneuron zu speichern. Dieser Lösungsansatz hat jedoch den Nachteil, dass die Abfrage bestimmter Verbindungen nicht sehr performant ist, da unter Umständen alle Gewichte durchsucht werden müssen, bis das Gewicht zu einer gegebenen Verbindung gefunden werden kann. Bei der fortlaufenden Zuordnung kann dies vermieden werden. Bei einer Abfrage von einem Gewicht für eine Verbindung aus der Ebene mit dem Index *i* in die Ebene mit dem Index *i+1* und dem Index der Neuronen von *j* und *k* kann der Index des Gewichts im Array direkt über den Aufbau des Netzwerkes berechnet werden, ohne alle Gewichte zu durchsuchen. Bei der Berechnung der Ausgabe des Netzwerkes, wird nach jedem Zugriff auf den Gewicht-*Array* der Zugriffsindex inkrementiert. Daraus resultiert, dass die Verbindungen implizit durchnummeriert sind und die Zuordnung eindeutig über den Index im *Array* gegeben ist. Auch das Speichern und Laden wird so stark vereinfacht, da ausschließlich alle Gewichte hintereinander ohne Positionsinformationen im Netzwerk gespeichert werden können. Der Aufbau des Netzwerkes folgt aus den gewählten Einstellungen, wie Anzahl an Sensoren und Hidden-*Layers* und muss deshalb nicht mitgespeichert werden, wenngleich so die Kompitabilitätsprüfung zwischen gewählter Einstellung und gespeichertem Netzwerk vereinfacht werden würde. Die einzige mögliche Überprüfung ist die Anzahl an Gewichten. Wenn die Einstellung aber so angepasst werden, dass der Aufbau des Netzwerkes sich zwar ändert, nicht aber die Anzahl der Gewichte, kann nicht überprüft werden, ob ein gespeichertes Netzwerk denselben Aufbau, wie das momentan eingestellte Netzwerk besitzt. Zusätzlich müssten Informationen über *Hidden-Layers* und Sensoranzahl gespeichert werden. Wenn sich aber nicht die Anzahl der Sensoren ändert, sondern die Richtung, in die sie die Umgebung abtasten, dann wird das Netzwerk trotzdem nicht genauso funktionieren, wie es das beim Speichern tat. Genauso verhält es sich, wenn man die maximale Geschwindigkeit, das Beschleunigungsverhalten, die Fahrzeugform oder nahezu jede beliebige andere Einstellung ändert. Die einzige Lösung wäre die Einstellungsdatei immer zusammen mit dem neuronalen Netz zu speichern, was aber aufgrund der Menge an möglichen Einstellungen eine speicherverschwendende Lösung darstellen würde. Aus diesem Grund wird auf jegliche Form der Kompatibilitätsprüfung verzichtet. Diese Aufgabe wird dem Benutzer überlassen.

Neben den Neuronen und Gewichten, besitzt die *NeuronalVehicleBehavior*-Klasse zwei weitere Eigenschaften. Mithilfe die *Parent*-Eigenschaft wird gespeichert, aus welchem Netzwerk das Netzwerk entstanden ist. Da der Trainingsprozess über Mutation neue neuronale Netzwerke erzeugt, kann über diese Eigenschaft nachverfolgt werden, wie groß die genetische Diversität ist. Wird in jeder Generation jedes neue Netzwerk aus genau einem Netzwerk generiert, ist das ein Zeichen dafür, dass der Selektionsalgorithmus überarbeitet werden sollte. Grundsätzlich entspricht jedes Netzwerk einem Lösungsansatz für die Aufgabe möglichst schnell und sicher auf einer ausgewählten Strecke zu fahren. Zwar werden nicht alle Lösungskandidaten gleich erfolgreich sein, aber trotzdem wahrscheinlich unterschiedliche Lösungsansätze kodieren. So ist ein Fahrverhalten eher langsam, fährt aber dafür relativ genau die Ideallinie, wohingegen ein anderes Fahrverhalten vielleicht eher auf Geschwindigkeit optimiert ist. Wird nur eines dieser beiden Fahrverhalten als Ausgangspunkt für die nächste Generation genommen, dann besitzen ab dem Zeitpunkt alle Individuen in der Population ein sehr ähnliches Fahrverhalten. Unter Umständen stellt sich aber heraus, dass das alternative Fahrverhalten auf lange Sicht bessere Ergebnisse erzielt hätte. Ohne ein Individuum in der Population was auf diesem Fahrverhalten basiert, ist es aber eher unwahrscheinlich, dass über zufällige Mutation plötzlich dieses bessere Fahrverhalten erreicht wird. Wahrscheinlicher wäre es, wenn das alternative Fahrverhalten in der Population erhalten geblieben wäre. Aus diesem Grund ist es von Vorteil sich der Vererbungsstruktur der Individuen bewusst zu sein. Über die *Parent*-Eigenschaft kann dieser Vererbungsverlauf visualisiert werden und so Grundlage für gegebenenfalls nötige Anpassungen am Selektionsalgorithmus sein. Diese Visualisierung wird in Form von einem Baumdiagramm in dem Simulationsfenster dargestellt.

Neben der Vererbungsstruktur ist es auch wichtig zu wissen, wie sich die Bewertung im Laufe der Zeit verändert. Stagniert beispielsweise die Bewertung zu schnell, müssen Anpassungen an Mutationsrate und -stärke vorgenommen werden. Aber auch wenn die Variation der Bewertungen von Generation zu Generation zu groß ist, kann das ein Zeichen sein, dass die Mutationseinstellungen anders festgelegt sein sollten. Das Protokoll über die Bewertungen der Simulation werden in Form eines Liniendiagramms ebenfalls in dem Simulationsfenster dargestellt.

Die Festlegung der Mutationsrate und -stärke gestaltet sich, wie in Kapitel *Konzept* bereits beschrieben, als nicht ganz trivial. Aus diesem Grund wird neben der

Vererbungsstruktur und des Bewertungsverlaufs auch die Mutationsrate und -stärke visuell dargestellt. So kann untersucht werden, welche Kombinationen der beiden Werte durchschnittlich zur größten Verbesserungen führt. Als Darstellungsform bietet sich auch hier ein Liniendiagramm an.

Als viertes und letztes Diagramm wird das neuronale Netzwerk eines ausgewählten Individuums dargestellt. So kann untersucht werden, welche Verbindungsgewichte im Laufe der Simulation verstärkt oder abgeschwächt werden. Auch können anhand der Darstellung Rückschlüsse auf die Funktionsweise des Netzwerks gezogen werden. Diese vier Diagramme stellen neben der subjektiven Beurteilung des Fahrverhaltens, mittels des Visualisierungsfensters, die Grundlage für die spätere Evaluation und Auswertung im Hinblick auf die ursprüngliche Frage dar.

### 5.2.3 Simulation

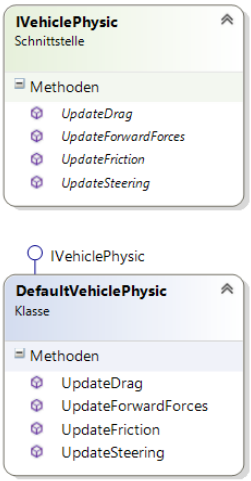
Der grundsätzliche Simulationsablauf wurde bereits im Kapitel *Konzept* vorgestellt. Was noch nicht erläutert wurde, ist wie sich dieser Ablauf in den Rest des Programms integriert.

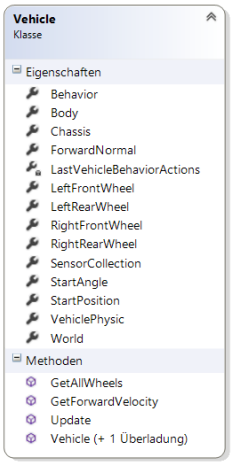
Wird die Simulation zum ersten Mal gestartet sind einige Schritte von Nöten, damit die Simulation fehlerfrei durchlaufen werden kann. Als erster Schritt werden so viele Simulationsumgebungen erstellt, wie ausgewählte Individuen in der Population gibt. Die Motivation hinter diesem Schritt ist, dass so jedes Individuum in einer eigenen Simulationsumgebung agieren kann und somit nicht auf die anderen Fahrzeuge reagieren muss[12]. Außerdem können so alle Simulationsumgebungen parallel ausgeführt werden. Auf modernen Computerarchitekturen kommen nahezu ausschließlich Mehrkernprozessoren zum Einsatz, die Berechnungen in der Regel vollständig parallel durchführen können. Startet man alle Simulationen gleichzeitig kommt es auf einem modernen acht-Kern Prozessor zu einer durchschnittlichen Prozessorauslastung von ungefähr 95%. Werden alle Fahrzeuge in einer einzigen Simulationsumgebung platziert und diese dann gestartet, wird derselbe Prozessor nur zu etwa 20% ausgelastet. Prozessorauslastung alleine ist sicher nicht das entscheide Kriterium, wenn es um die Beurteilung der Effizienz eines Programms geht, was aber zusätzlich berücksichtigt werden muss ist, dass auch die Dauer der physikalischen Berechnungen zunehmen, wenn alle Fahrzeuge zusammen simuliert

[12] Das Ignorieren anderer Teilnehmer in der Simulation kann zwar implementiert werden, allerdings muss dann bei jeder Kollision der *Sensorstrahlen* mit einem Objekt zusätzlich überprüft werden, ob es sich dabei um eine Wand oder ein anderes Fahrzeug handelt. Diese Überprüfung entfällt in diesem Fall.

werden. Es müssen mehr Kollisionen erkannt werden, obwohl viele ohnehin verworfen werden (Kollision zwischen zwei Fahrzeugen wird ignoriert). Misst man die Dauer bis eine Generation vollständig simuliert wurde, bei beiden Ansätzen, stellt sich heraus, dass die parallele Ausführung fast zehnmal so schnell ist, wie die Ausführung bei einer einzigen Simulationsumgebung. Diese Entwurfsentscheidung konnte natürlich erst nach der Implementierung getroffen werden, dennoch ist dieser Vorgriff an dieser Stelle nötig, um die ausgewählte Option zu motivieren.

Als zweiter Schritt werden die einzelnen Simulationsumgebungen initialisiert. In die Physikengine werden die Polygone der Rennstrecke geladen und das Fahrzeug wird



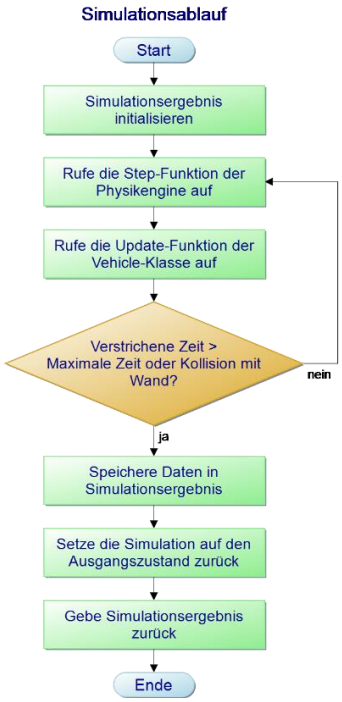

jeweils erstellt und mit den Sensoren ausgestattet. Außerdem wird es an der Startposition auf der Rennstrecke platziert.[13] Die Fahrverhalten werden entsprechend der Auswahl des Benutzers erstellt (aus Datei geladen oder zufällig generiert). Jedes Fahrzeug besitzt eine Reihe von Eigenschaften die initialisiert werden müssen. Jedem Fahrzeug wird im Konstruktor als Parameter genau ein Fahrverhalten übergeben. Neben der eigentlichen Steuerung werden auch die Räder und eine Physikkomponente erzeugt. Die Physikengine wird mithilfe von Dependency-Injection an die *Vehicle*-Klasse übergeben. Die Fahrzeug Klasse erwartet nur eine abstrakte Definition der benötigten Methoden, die konkrete Klasse mit der Implementierung wird zur Laufzeit übergeben. Nachdem die *Step*-Funktion aufgerufen wurde, wird die *Update*-Funktion der *Vehicle*-Klasse aufgerufen. Diese signalisiert dann der Fahrzeugphysik, dass eine erneute Berechnung notwendig ist. Über diesen Weg können auf einfache Art und Weise weitere alternative Implementierungen für die Fahrzeugphysik implementiert und zur Simulation hinzugefügt werden. An dieser Stelle sei jedoch erwähnt, dass kein besonderer Fokus auf die Erweiterbarkeit dieses Hilfsprogramms gelegt wurde. Welche Funktionen zur Verfügung stehen sollen, war von Anfang an definiert. Die Entwicklung von flexiblen Softwaresystemen erfordert einen nicht unerheblichen

[13] Die ersten Koordinaten der Rennstrecke dienen als Startpunkt. Ob dies dem tatsächlichen Start auf den originalen Rennstrecken entspricht, hängt davon ab, wie die GPSies Benutzer die Rennstrecke eingestellt haben.

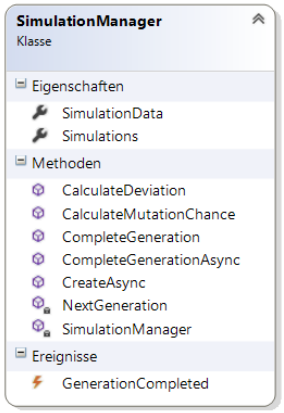
höheren Entwicklungsaufwand, der sich im Zeitrahmen dieser Arbeit nicht umsetzen ließe.

Die *Wheel*-Klassen verwalten die *Joints* zu der Hauptkarosserie und kapseln so unnötige Informationen aus der *Vehicle*-Klasse heraus. Auf die verbleibenden Eigenschaften der Fahrzeug-Klasse werden an dieser Stelle nicht weiter eingegangen. Bei ihnen handelt es sich hauptsächlich um Informationen über die physikalischen Eigenschaften, die von der Fahrzeugphysik genutzt werden können.

Die bis jetzt abgehandelten Schritte werden nur einmalig beim Start der Simulation ausgeführt. Es handelt sich bei ihnen in erster Linie um die Initialisierung der Simulation mit den entsprechenden Komponenten. Im Folgenden werden diejenigen Abläufe vorgestellt, die sich, wie im Kapitel *Konzept* beschrieben, zyklisch wiederholen. Mit dem Aufruf der *CompleteGeneration*-Methode der *SimulationManager*-Klasse wird der Start eines neuen Zyklus eingeleitet. Jede Ausführung der Teilsimulationen der einzelnen Fahrzeuge werden in Threads geladen, die anschließend gleichzeitig gestartet werden. Der Ablauf einer einzelnen Simulation lässt sich über folgenden Programmablaufplan visualisieren:



Sobald alle Teilsimulationen terminiert sind, werden die Ergebnisse vom *SimulationManager* gesammelt und die Vorbereitung auf die nächste Ausführung beginnt. Dazu wird das neuronale Netzwerk mit der besten Bewertung unverändert in die nächste Generation übernommen. So wird sichergestellt, dass die Population sich im Laufe des Trainings zumindest nicht verschlechtert und die insgesamt beste Lösung erhalten bleibt. Experimente mit der Übernahme mehrerer unveränderter neuronaler Netze zeigte in der Regel ein etwas langsameres Training. Die restlichen neuronalen Netze werden entsprechend dem im Kapitel *Konzept* vorgestellten Verfahren selektiert und mutiert in die nächste Generation übernommen. Bei der Mutation wird ein neues Netzwerk erstellt, allerdings ein Verweis auf das Netzwerk, welches die Grundlage für die Mutation darstellte, gespeichert, damit der Vererbungsverlauf, wie erwähnt, dargestellt werden kann. Die Mutationsrate und -stärke werden wie beschrieben berechnet. Anschließend werden die Fahrzeuge in der Simulationsumgebung mit den neuronalen Netzen der neuen Generation ausgestattet und die nächste Generation kann gestartet werden. Vorher werden die gesammelten Ergebnisse jedoch in der Benutzeroberfläche dargestellt. Wurde die Simulation nicht im Einzelschrittmodus gestartet, wird nun erneut die *CompleteGeneration*-Methode aufgerufen und die nächste Generation wird berechnet.




Der hier vorgestellte Entwurf fokussiert sich vor allem auf die Simulation. Die Klassen die an dieser Stelle nicht behandelt wurden, können im Anhang in Form von Klassendiagrammen gefunden werden. Dies bezieht sich in erster Linie auf Klassen die für die visuelle Darstellung der Ergebnisse verantwortlich sind, sowie Hilfsklassen, die keine zentrale Rolle in dem Simulationsablauf einnehmen. Wie genau der Ablauf bei der Berechnung der Sensorwerte ist, und wie die Ausgabe des neuronalen Netzwerkes bestimmt wird, wird im folgenden Kapitel *Implementierung* beschrieben. Außerdem wird der Programmablaufplan genauer erläutert. Der Hintergrund ist, dass eine Erklärung ohne auf die konkrete Implementierung einzugehen die Vorgänge nicht zufriedenstellen darlegen könnte.

# 6 Implementierung

Nachdem im vorherigen Kapitel der Entwurf der Architektur des Programms vorgestellt wurde, folgt nun die Implementierung. Dabei wird besonderer Fokus auf die konkrete Simulation und die benötigten Klassen gelegt und weniger auf die visuelle Darstellung oder optionale Funktionen.

## 6.1 Platzierung der Fortschrittssensoren

Die Fortschrittssensoren dienen der Entfernungsmessung der Fahrzeuge auf den Rennstrecken. Sie werden alle zehn Meter platziert und erlauben so, die zurückgelegte Strecke der Fahrzeuge zu protokollieren. Wie genau sie zur Bewertung der Individuen der jeweils aktuellen Population beitragen wurde bereits beschrieben, deshalb soll im Folgenden erläutert werden, wie sich genau die Positionierung und Ausrichtung berechnet. Als Daten über die Rennstrecke liegen eine begrenzte Anzahl an Koordinaten, sowie die innere und äußere Streckenbegrenzung vor. Die Herausforderung hierbei ist trotz dieser begrenzten Koordinaten, alle zehn Meter, unabhängig vom Abstand der gegebenen Koordinaten, einen Fortschrittssensor zu platzieren.

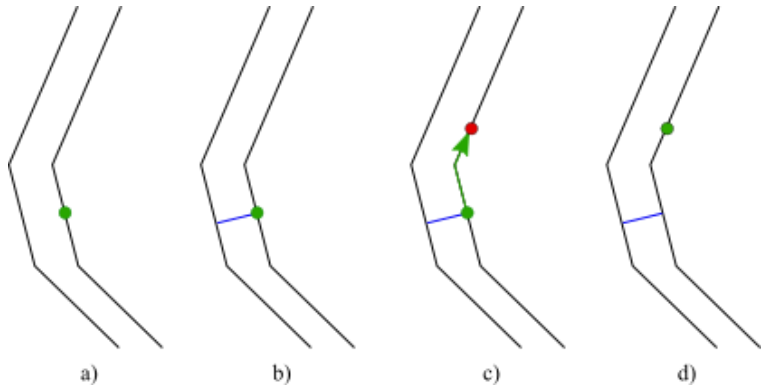
Die Sensoren werden mit folgendem Code generiert:

```
private void CreateSensors()}
{
    var sensors = new List<Line>();
    double length = innerVertices.GetTotalLength();
    double stepSize = SensorSpacing;
    double currentStep = stepSize;

    while (currentStep + stepSize < length)
    {
        var line = GetLineOnVertices(innerVertices, currentStep);
        CreateProgressSensor(line, sensors.Count);
        sensors.Add(line);
        currentStep += stepSize;
    }
    Sensors = sensors;
}
```

Die grundsätzliche Idee ist es, die innere Streckenbegrenzung zu interpolieren und so die Platzierung zu vereinfachen. Die Methode *GetTotalLength* ermittelt die Gesamtlänge der gegebenen Vektoren. In diesem Fall die Länge der inneren Fahrbahnbegrenzung. In der Variablen *stepSize* wird der, in den Einstellungen festgelegte, Abstand der Sensoren gespeichert. Solange nicht die komplette innere

Abgrenzung traversiert wurde, wird anschließend zyklisch folgender Ablauf durchgeführt:




Die Ausgangssituation ist in der Abbildung ganz links dargestellt. Nachdem Ermitteln der Sensorposition und -ausrichtung und der Platzierung des Sensors ergibt sich eine Situation wie sie in der zweiten Abbildung dargestellt ist. Nun wird entlang der inneren Streckenbegrenzung um den festgelegten Sensorabstand fortgefahren. Der rote Kreis in Abbildung stellt die neue Position dar. Nun beginnt der Ablauf von vorne.

Die Startposition der Sensorlinie ist immer die aktuelle Position auf der inneren Begrenzung. Die Ausrichtung ergibt sich, indem der Vektor von der Startposition zur nächsten Ecke des Polygons um 90° entgegen des Uhrzeigersinns rotiert wird. Die Länge des Vektors wird normiert und anschließend mit der Breite der Strecke multipliziert. Der Endpunkt der Sensorlinie ergibt sich, wenn der errechnete Vektor und der Vektor der Startposition addiert werden. Nun ist der Sensor vollständig definiert. Das hinzufügen zur Physikengine ist trivial und wird nicht weiter erläutert.

Die Berechnung der Position des nächsten Sensors gestaltet sich hingegen komplexer. Die *GetLineOnVertices*-Methode erhält die absolute bereits zurückgelegte Strecke auf der inneren Streckenbegrenzung als Parameter. Nun wird solange die Abgrenzung traversiert, bis der nächste Eckpunkt weiter entfernt ist, als die übergebene aktuelle Position. So wird genau zwischen den beiden Eckpunkten des Polygons gestoppt, zwischen denen auch die nächste Position des Sensors liegen soll. Es wird die verbleibende Distanz gemessen, die noch zum Erreichen dieses Punktes fehlen. Genau diese Entfernung wird dann in Richtung des nächsten Eckpunktes projiziert. Der so ermittelte Punkt stellt die neue Position des Sensors dar. Nun wird wieder orthogonal auf der Streckenbegrenzung der Sensor platziert

und der Ablauf wiederholt sich, bis die vollständige Strecke mit Sensoren ausgestattet ist.

## 6.2 Künstliche neuronale Netze

Wie in den vorherigen Kapiteln bereits erläutert beginnt die Simulation mit der Erzeugung der neuronalen Netze auf Basis der Auswahl des Benutzers. Zur Festlegung der Gewichte kommen sogenannte *Delegates* zum Einsatz. In C# kann mithilfe von *Delegates* Funktionen als Parameter zur Laufzeit übergeben werden. Sie definieren die Anzahl und die Typen der Parameter der Methode und den Typ des Rückgabewertes, nicht aber die konkrete Implementierung. Vergleichbar sind sie mit *Interfaces* die eine einzige Methode beinhalten. Für die Generierung der Gewichte der neuronalen Netze wird im Konstruktor eine Methode erwartet, die als Parameter den Index des zu generierenden Gewichts erwarten und darauf basierend das konkrete Gewicht für diese Verbindung zurückgibt.

Soll ein Netzwerk hingegen aus einer Datei geladen werden, so können die Gewichte aus einer Datei über den übergebenen Index adressiert und so einzeln an den Konstruktor übergeben werden. Der Vorteil dieses Ansatzes ist es, dass ein einziger Konstruktor ausreicht um beide Konstruktionsweisen zu implementieren. Außerdem können so einfache neue Methoden zur Generierung der Gewichte dem Programm hinzugefügt werden. Beispielsweise ein partielles Laden eines neuronalen Netzwerkes aus einer Datei, bei der die verbleibenden Gewichte zufällig erzeugt werden. Im Konstruktor wird anschließend berechnet wie viele Gewichte benötigt werden und ein *Array* mit der entsprechenden Größe erzeugt. Dieses wird dann iterativ mit den Rückgabewerten der übergebenen Methode gefüllt. Die Reinfolge in der die Gewichte ausgelesen werden ist immer gleich so reicht einzig die Position im *Array* der Gewichte aus, um sie den Verbindungen des Netzwerkes zuzuordnen.

Die Berechnung der Ausgabe erfolgt von der Eingabeebene zur Ausgabeebene hin. Die Neuronen werden dabei von *oben nach unten*[14] in der jeweiligen Ebene durchlaufen und ihr Ausgabewert berechnet. Die Berechnung des Wertes eines Neurons erfolgt auf folgende Art und Weise:[15]

---

[14] O*ben nach unten* meint hier, dass der Neuronen anhand ihres Index im Array in aufsteigender Reinfolge berechnet werden. Stellt man sich das Netzwerk als Tabelle vor, so würde zunächst die Reihe von oben nach unten durchlaufen werden, bevor die nächste Spalte errechnet wird.

[15] Die Variable *i* gibt den Index der aktuellen Ebene an. Die Variable *j* den Index des zu berechnenden Neurons in der aktuellen Ebene.

```
double sum = 0;
for (int k = 0; k < Neurons[i - 1].Length; k++)
{
    sum += Neurons[i - 1][k] * Weights[weightIndex++];
}
Neurons[i][j] = ActivationFunction(sum, i, j);
```

Die Aktivitätsfunktion wird auf Basis der Position des Neurons bestimmt. Da die Ausgabe des Netzwerkes in einem bestimmten Intervall liegen muss[16], wird für die Ausgabeneuronen die Aktivitätsfunktion gesondert festgelegt. Für alle anderen Neuronen wird die Funktion genutzt, die vom Benutzer in den Einstellungen festgelegt wurde. Nachdem alle Neuronen berechnet wurden, werden die Werte der beiden Neuronen in der letzten Ebene ausgelesen und dem Konstruktor der *VehicleBehaviorActions*-Klasse übergeben. Diese Klasse speichert die Ausgabe und bietet die Grundlage, auf welche später die Fahrzeugphysik die physikalischen Kräfte berechnet.

## 6.3 Physikengine und Fahrzeugphysik

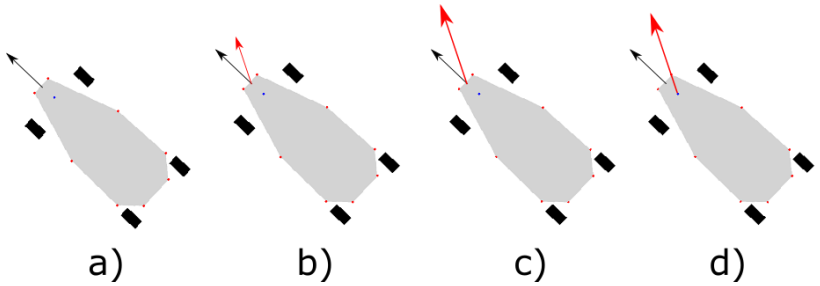
Im *Konzept* und *Entwurf* wurde bereits auf die verwendete Physikengine eingegangen, jedoch der konkrete Umgang mit dieser nicht näher erläutert. Dies wird im Folgenden anhand der Fahrzeugklasse und -physik nachgeholt.

Für die allgemeine physikalische Berechnung kommt, wie erwähnt, die *Farseer Physics Engine* zum Einsatz. Bevor Objekte von der Engine simuliert werden können, müssen sie einem *World*-Objekt hinzugefügt werden. Dieses *World*-Objekt verwaltet alle Komponenten der physikalischen Simulation, sowie eine Reihe weiterer Einstellungen wie Gravitation. Für dieses Projekt wird für die Gravitation ein null-Vektor gewählt, da die Simulation aus der Draufsicht realisiert ist. So werden keine zusätzlichen Kräfte auf die Elemente der Simulation angewendet, außer die, die manuell hinzugefügt wurden. Nachdem die *Step*-Funktion der Physikengine aufgerufen wurde, folgt die manuelle Berechnung der neuen Kräfte. Zunächst müssen dafür jedoch die Sensorwerte ermittelt werden. Dies geschieht über die Klasse *SensorCollection*. In ihr wird die Methode *GetSensorValues* implementiert. In den Einstellungen sind die verschiedenen Winkel der Richtungen relativ zur Fahrtrichtung gespeichert, in jene die Distanz gemessen werden soll. Der Ablauf lässt sich über folgenden Code darstellen:

[16] Zur Erinnerung: [-1,1] für die Richtung und [0,1] für die Geschwindigkeit.

```
public IEnumerable<SensorValue> GetSensorValues()
{
    foreach (var angle in SensorAngles)
    {
        var direction = Vehicle.ForwardNormal.Rotate(angle);
        var endPoint = SensorStart + direction * ViewDistance;
        float value = 1;
        Vehicle.World.RayCast((hit,fraction) =>
        {
            if (hit is not Wall)
                return -1; //ignore
            if (value > fraction)
                value = fraction; //save distance
            return -1;
        }, SensorStart, endPoint);
        yield return new SensorValue(angle, value);
    }
}
```

Es wird über jeden angegebenen Winkel iteriert. Zunächst wird die aus dem relativen Winkel zur Fahrtrichtung des Autos ein Vektor berechnet, der die absolute Richtung im zweidimensionalen Raum beschreibt.




In der Abbildung ganz links ist der Normalrichtungsvektor des Fahrzeugs als schwarzer Pfeil dargestellt. Der Startpunkt des Vektors ist hier als Koordinatenursprung anzusehen. In der zweiten Abbildung ist dieser um einen Winkel rotiert worden (roter Pfeil). Für die Rotation kommen folgende Formeln zum Einsatz:

$$x_{neu} = x_{alt} * \cos(winkel) - y_{alt} * \sin(winkel)$$
$$y_{neu} = y_{alt} * \sin(winkel) + y_{alt} * \cos(winkel)$$

Der rotierte Vektor wird nach der Multiplikation mit der maximalen Sichtweite (zu sehen in der dritten Abbildung), auf die Position des Sensors verschoben, wie in der letzten Abbildung dargestellt wird.

Über die Funktion *RayCast* der bereits erwähnten *World*-Klasse kann nun von der Position der Sensoren in dem Fahrzeug analysiert werden, durch welche Objekte ein theoretischer Lichtstrahl dringen müsste um am errechneten Endpunkt anzukommen. Der erste Parameter ist dabei wieder ein *delegate* das hier in der

Kurzschreibform[17] implementiert ist. Diese *inline*-Methode wird von der *RayCast*-Funktion immer dann aufgerufen, wenn es auf dem Weg von der Start- zu der Endposition des simulierten Lichtstrahls zu einer Kollision mit einem Element der Engine kommt. Die Methode überprüft dann, ob es sich bei dem Objekt um eine Streckenbegrenzung handelt. Ist dies nicht der Fall wird die Kollision ignoriert. Dies ist Beispielsweise der Fall, wenn der Lichtstrahl die Reifen durchdringen muss. Falls hingegen tatsächlich eine Kollision mit einer Wand vorliegt, dann wird abgefragt, ob die Kollision näher am Fahrzeug festgestellt wurde, als bereits vorher erkannte Kollisionen. Dies hat den Hintergrund, dass in den Spezifikationen der *Farseer Physics Engine* nicht garantiert wird, dass die Kollisionen in der Reinfolge auftreten, wie dies in der Realität der Fall wäre. So könnte zuerst eine entfernte Wand erkannt werden, bevor das durchdringen des nahegelegenen Reifens notwendig wird. Durch das Vergleichen der Entfernungen wird realisiert, dass nur die nächstgelegene Streckenbegrenzung gespeichert wird. Die Entfernung ist hierbei nicht in Metern gegeben, sondern als Anteilswert zur maximalen Entfernung. Angenommen die Sichtweite beträgt zweihundert Meter, und in hundert Meter Entfernung befindet sich eine Wand. Dann würde der als Parameter übergebene Anteilswert für diese Kollision 0,5 betragen.

Nachdem dieses Verfahren für alle eingestellten Winkel durchgeführt wurde, werden die Ergebnisse in Form von *SensorValue*-Objekten an den Aufrufer zurückgegeben.[18] In diesem Fall erfolgte der Aufruf aus der *Update*-Funktion der *Vehicle*-Klasse heraus. Die ermittelten Resultate werden daraufhin als *VehicleBehaviorInput* dem aktuellen neuronalen Netzwerk des Fahrzeuges übergeben. Wie das Netzwerk aus diesen Daten die Ausgabe berechnet wurde bereits im Abschnitt *Künstliche neuronale Netze* dieses Kapitels beschrieben.

Die Ausgaben des neuronalen Netzwerkes werden, bevor sie der Fahrzeugphysik übergeben werden, mit den vorherigen Werten verrechnet:

$$faktor_{gemittelt} = faktor_{neu} * (1 - glättung) + faktor_{alt} * glättung$$

Der Parameter *glättung* wird über die Einstellungen festgelegt. Standardmäßig beträgt dieser 0,2. Die Motivation hinter diesem Schritt ist es spontane

[17] Auch unter dem Namen Lambda-Expression bekannt.

[18] In C# ist es möglich Methoden zu programmieren, die erzeugte Objekte iterativ an den Aufrufer zurückgeben. Dies wird über das Schlüsselwort *yield* implementiert. Neben geringerem Programmieraufwand, werden die Ergebnisse auch erst dann abgefragt, wenn sie benötigt werden. Genaueres dazu findet sich unter dem Link: https://msdn.microsoft.com/library/9k7k7cf0.aspx

Veränderungen in der Ausgabe des neuronalen Netzwerkes abzufedern. Ein Beispiel hierfür ist das Kurvenverhalten. Kommt das Fahrzeug zu einer Kurve wird ein Sensor von einem Simulationsschritt zum nächsten einen deutlich anderen Wert annehmen. In dem vorherigen Schritt trifft der Sensorstrahl noch auf die Streckenbegrenzung im nächsten bereits nicht mehr. Somit wird der Wert, den dieser Sensor annimmt, deutlich größer. Das neuronale Netzwerk sollte idealerweise darauf reagieren, indem es in die Kurve einzulenken beginnt. In dem darauffolgenden Simulationsschritt hat das Fahrzeug dann die Fahrtrichtung angepasst. Jetzt kollidiert der Sensorstrahl der soeben noch an der Innenbegrenzung vorbei reichte, wieder mit dieser. Der Wert des Sensors ändert sich wieder stark, was bedeutet, dass das neuronale Netzwerk keine Kurve mehr erkennen kann. Statt weiter der Kurve zu folgen, wird das Fahrzeug erneut geradeaus weiterfahren. Bis der Sensorstrahl wieder an der Streckenbegrenzung vorbei reicht. Dieser Ablauf wiederholt sich. Und auch wenn nach jeder Wiederholung das Fahrzeug im Ergebnis doch weit genug eingelenkt hat um die Kurve zu durchfahren, entsteht ein deutlich sichtbares Pendeln des Fahrzeuges. Um diesem Pendeln entgegen zu wirken wird die jeweilige Ausgabe mit dem letzten Ausgabewert verrechnet. Wurde im letzten Simulationsschritt eine vollständige Auslenkung nach links angestrebt, so wird über dieses Verfahren verhindert, dass im nächsten Schritt komplette Rechtsauslenkung das Ziel darstellt. Die Konsequenz ist, dass das neuronale Netzwerk nur verzögert auf die Umgebung reagieren kann. Jedoch ist, wie folgende Auswertung zeigt, die Verzögerung gering genug, um keine nennenswerte Einschränkung darzustellen.

| Simulationsschritt | Auslenkung |
|---|---|
| 1 | -1 |
| 2 | 0,6 |
| 3 | 0,92 |
| 4 | 0,984 |
| 5 | 0,9968 |
| 6 | 0,99936 |

Bei einer Anfangsauslenkung von -1 und der eigentlichen neuen Auslenkung von 1, wird nach nur drei Simulationsschritten bereits eine neue Auslenkung von 0,92 erreicht. Nach fünf Simulationsschritten ist der Unterschied zwischen gemitteltem und eigentlichem Wert vernachlässigbar gering. Hinzu kommt, dass das Zeitintervall zwischen den einzelnen Simulationsschritten so klein ist, dass in der Echtzeitvisualisierung nur rund 50 Millisekunden vergehen, bis nahezu kein Unterschied mehr festgestellt werden kann. Wie groß die Dämpfung von altem und

neuem Wert maximal sein kann, stellt ein Kompromiss zwischen stabilem Fahrverhalten und schnellen Reaktionsvermögen dar. Wird die Dämpfung zu groß gewählt, kann das neuronale Netz, selbst in der Theorie, die Kurven nicht mehr ohne Kontakt mit der Streckenbegrenzung durchfahren. Auf der anderen Seite bedeutet ein zu kleiner Wert ein unruhiges Kurven- und Beschleunigungsverhalten. Der Standardwert von 0,2 stellt zwar ein gutes Mittel zwischen diesen beiden Extremen dar, jedoch kommt es trotzdem nach wie vor zu einer abrupten Fahrweise. Genauer wird auf dieses Problem im Kapitel *Evaluation* eingegangen.

Nachdem die neuen Werte für den Geschwindigkeitsfaktor sowie die Lenkrichtung bestimmt und diese von der Fahrzeugphysik in Form von Kräften und Impulsen auf das Fahrzeug angewendet wurden, ist der Simulationsschritt abgeschlossen. Falls die maximale Zeit noch nicht überschritten ist und es in diesem Schritt zu keiner Kollision mit der Streckenbegrenzung kam, folgt nun unmittelbar die Berechnung des nächsten Simulationsschrittes.

## 6.4 Optimierung durch Parallelisierung

Im Kapitel *Entwurf* wurden bereits erste Optimierungsmaßnahmen in Form von Parallelisierung vorgestellt. An dieser Stelle wird etwas genau auf die Thematik eingegangen.

Zur Parallelisierung von Programmabläufen stehen neben herkömmlichen *Threads* in C# sogenannte *Tasks* zur Verfügung. Die Tasks wurden im Rahmen der *Task Parallel Library* (TPL) eingeführt. Sie fungieren auf einer abstrakteren Schicht. In dieser Simulation werden deshalb streng genommen unmittelbar keine *Threads* verwendet, sondern die optimierten *Tasks*. Die *TPL* verwaltet das Ausführen von *Tasks* und gewährleistet, dass nicht unnötig viele Threads im Hintergrund gestartet wurden. Werden mehr *Threads* ausgeführt, als es Prozessorkerne gibt, kommt es in der Regel zu Effizienzproblemen. Damit alle *Threads* die gleiche Prozessorzeit erhalten, schaltet das Betriebssystem, recht zeitintensiv, automatisch zwischen den ausgeführten *Threads* um. Wurden nur so viele *Threads* gestartet, wie es auch Prozessorkerne gibt, wird dieses Umschalten minimiert. Die *TPL* plant die Ausführung der gestarteten *Tasks* so, dass jeder *Task* in genau einem *Thread* ausgeführt wird, aber höchstens so viele wie es aus Effizienzgründen Sinn macht. Aus diesem Grund ist es unproblematisch jede Teilsimulation in einem eigenen *Thread*, beziehungsweise *Task* auszuführen, obwohl vielleicht im Kapitel *Entwurf*

zunächst ein anderer Anschein erweckt wurde. Auf eine detaillierte Beschreibung der *TPL* wird an dieser Stelle verzichtet, da sie für den weiteren Ablauf des Programms nur eine nebensächliche Rolle einnimmt.[19]

Nachdem nun die Kernkomponenten des Hilfsprogramms durch dieses Kapitel und das Kapitel *Entwurf* beschrieben wurden, können im nächsten Kapitel *Evaluation* die Ergebnisse der Simulationen in Hinblick auf die Ausgangsfrage ausgewertet werden.

[19] Unter https://msdn.microsoft.com/library/dd460717 findet sich die offizielle Dokumentation.

# 7 Evaluation

Im Anschluss an die Beschreibung des Entwurfs und der darauf aufbauenden Implementation, werden im Folgenden die Resultate, die mit dem so erstellten Programm gesammelt wurden ausgewertet. Die Ausgangsfrage, ob sich künstlichen neuronalen Netze, die mit Methoden des Themengebiets der evolutionären Algorithmen trainiert wurden, sich für die Steuerung von Fahrzeugen eignen, steht dabei im Vordergrund.

## 7.1 Fehlerfrei zurückgelegte Strecke

Die Fehlerfrei zurückgelegte Strecke stellt das Hauptkriterium für die Bewertung der einzelnen neuronalen Netze dar. Häufig kommt es, besonders in den ersten Generationen, zu dem Fall, dass keines der Individuen in der Lage ist, bestimmte komplexere Streckenabschnitte ohne Kollision mit der Streckenbegrenzung zu durchfahren. Aus diesem Grund gleichen sich die Individuen der Population zunächst an. Es gibt in der Regel einige wenige Individuen, die bis zu einer besonders engen Kurve fahren können wohingegen die verbleibenden Lösungskandidaten bereits vorher kollidieren. Durch den gewählten Selektionsalgorithmus, werden in jeder Generation mit hoher Wahrscheinlichkeit die Lösungen ausgewählt, die die Strecke am weitesten erfolgreich durchfahren können. Aber auch schlechter bewertete Individuen werden berücksichtigt um die genetische Diversität aufrecht zu erhalten. Da aber über mehrere Generationen keine der Individuen der benötigten Mutation unterlaufen, um den problematischen Streckenabschnitt erfolgreich zu durchfahren, summiert sich die Selektionswahrscheinlichkeit für die besseren Lösungskandidaten auf. Bis nahezu jedes Individuum das gleiche Fahrverhalten besitzt und alle an derselben Stelle scheitern. So wird die Chance, dass die benötigte Mutation eintritt vergrößert. Dies geschieht auf Kosten der Vielfalt im Fahrverhalten. Gelingt einem neuronalen Netz, sich so zu entwickeln, dass es auch den komplexen Streckenabschnitt durchfahren kann, stellt, je nach Strecke, der Rest der Runde ebenfalls kein Problem mehr dar. Dies liegt darin begründet, dass häufig das schwere Hindernis, die letzte untrainierte Situation darstellt, auf die das Netzwerk lernen muss korrekt zu reagieren.

Problematisch ist, dass ab diesem Zeitpunkt, nahezu alle Individuen, aufgrund der beschriebenen Selektionswahrscheinlichkeit, eine sehr ähnliche Struktur aufweisen. Dementsprechend langsam fällt häufig das anschließende Training der Individuen

aus. Und auch kann es passieren, dass die Simulation in einem lokalen Optimum stagniert, da keines der Individuen aufgrund der Großen Ähnlichkeit untereinander mehr die benötigten Eigenschaften besitzt, um ein anderes Optimum zu finden. So wird das bestehenden Fahrverhalten weiter optimiert, aber das globale Optimum nicht immer erreicht. Beispielsweise kann die Lösung, der es gelang als erstes das problematische Hindernis zu durchfahren, dies geschafft haben, indem es sich immer eng an der rechten Streckenbegrenzung gehalten hat. Dass dadurch Linkskurven nur sehr weitläufig durchfahren werden stellt einen Zustand dar, der sich nur sehr schwer wieder austrainieren lässt. Besonders wenn alle Individuen der Population diese Eigenschaft teilen.

Nichtsdestotrotz gelingt es in nahezu allen Fällen innerhalb weniger[20] Generation mindestens einmal eine Runde zu durchfahren. Weitere Ansätze um die genetische Diversität zu erhalten, wurden nicht untersucht.

## 7.2 Geschwindigkeit

In der Regel nimmt die Geschwindigkeit aufgrund der gewählten Bewertungsfunktion eine untergeordnete Rolle ein. Ob ein Individuum in zehn Sekunden Simulationszeit zwanzig oder einundzwanzig Fortschrittssensoren überfährt, macht keinen Unterschied, wenn es anschließend mit der Streckenbegrenzung kollidiert. Aus diesem Grund zeigt sich ein recht variables Bild in der Geschwindigkeit, bis das erste Individuum eine vollständige Runde absolviert hat. Erst dann beginnt der Prozess der Geschwindigkeitsoptimierung. Denn nun ist der Selektionsdruck nicht mehr in erster Linie auf der kollisionsfreien Absolvierung der Strecke, sondern vor allem auf der Erreichung einer möglichst hohen Distanz in den standardmäßig erlaubten zwei Minuten Fahrzeit. Wie nah das Fahrzeug dabei auf der Ideallinie fährt, ist nach wie vor wichtig, jedoch macht dies einen eher geringen Unterschied aus, wenn an anderes neuronales Netz auf den ausgedehnten Geraden das Fahrzeug auf maximal Geschwindigkeit beschleunigt.

Um das Trainieren von Fahrrichtung und Geschwindigkeit mehr miteinander zu kombinieren, wurde versucht in der Bewertungsfunktion die Durchschnittsgeschwindigkeit mit zu berücksichtigen. Dies führte aber meist dazu, dass die Fahrzeuge mit immer größerer Geschwindigkeit an problematischen

[20] Je nach Anzahl der Individuen meist zwischen zwanzig und vierzig Generationen. Dies entspricht in etwa einer Trainingszeit von zwei Minuten auf einem modernen achtkerne Prozessor.

Streckenabschnitten mit der Streckenbegrenzung kollidierten, anstatt zu lernen diese erfolgreich zu durfahren. Die Gewichtung von Geschwindigkeit und zurückgelegter Strecke stellte sich als nicht triviales Problem heraus. Sobald der Einfluss der Durchschnittsgeschwindigkeit groß genug wurde, um einen nennenswerten Unterschied zu machen, kam es erneut zu dem Problem der Maximierung der Kollisionsgeschwindigkeit, anstelle der Maximierung der allgemeinen Geschwindigkeit an den Stellen der Strecke, wo dies Sinn ergeben würde. Aus diesem Grund wurde letztendlich doch die Bewertungsfunktion verwendet, die ausschließlich die zurückgelegte Strecke als Bewertungskriterium verwendet.

Die Geschwindigkeit der Fahrzeuge macht, auch nach längerem Training, den Eindruck, als wären weitere Optimierungen durchaus möglich. Besonders auf kurvenarmen Streckenabschnitten, wo ein zügiges durchfahren der kleinen Kurven problemlos möglich wäre, kommt es in der Regel trotzdem sofort zur deutlichen Geschwindigkeitsreduktion. An dieser Stelle sind weitere Optimierungen der Bewertungsfunktion und unter Umständen sogar der Fahrzeugphysik notwendig, um bessere Ergebnisse zu erzielen.

## 7.3 Fahrverhalten

Die Einschätzung des Fahrverhaltens, kann ausschließlich subjektiv erfolgen. Aus diesem Grund erfolgt die Einschätzung in erster Linie auf Basis der visuellen Darstellung der Fahrverhalten in der Simulation. Das Problem, dass die Fahrzeuge pendeln und in einigen seltenen Fällen sogar die Kontrolle über das Fahrzeug verloren wird[21], zeigt sich trotz der eingebauten Dämpfung der Lenkeingriffe des neuronalen Netzes nach wie vor. In den bestbewerteten Individuen ist dieses Pendeln zwar minimiert, da hierbei schlichtweg eine längere Strecke zurückgelegt wird, als wenn das Fahrzeug vollends geradlinig fahren würde und Pendeln somit ein Nachteil bedeutet, nichtsdestotrotz kann in einigen Situationen das Problem trotzdem deutlich beobachtet werden. Problematisch für das Absolvieren der Strecke ist dies nicht und gerade deshalb gibt es auch keinen Selektionsdruck, der dem Pendeln entgegenwirken würde. Versuche algorithmisch das Pendeln zu identifizieren und in der Bewertungsfunktion zu integrieren brachten keine zufriedenstellenden Ergebnisse. Es kam immer dort zu Problemen wo schnell hintereinander die Richtung aufgrund des Streckenverlaufs geändert werden musste. Die

[21] Siehe entsprechenden Abschnitt im Kapitel *Implementierung*.

Unterscheidung, ob eine Richtungsänderung notwendig war oder nicht, konnte nicht akzeptabel erkannt werden. Würde das neuronale Netzwerk zu Steuerung von Fahrzeugen verwendet werden, die Menschen befördern, würde diese mit dem abrupten Fahrverhalten, vermutlich unzufrieden sein. Hinzu kommt, dass das neuronale Netz nicht trainiert wird, Sicherheitsabstände zu den Streckenbegrenzungen einzuhalten. Definiert man die Streckenbegrenzungen als Rasenflächen wird dies sicherlich weniger ein Problem für menschliche Insassen darstellen, als in Fällen wo das neuronale Netzwerk das Fahrzeug nur wenige Zentimeter an Häuserwänden oder anderen massiven Begrenzung vorbei manövriert. Die beiden vorgestellten Probleme stellen vermutlich das größte Hindernis dar, wenn es darum geht, dass künstliche neuronale Netzwerk, wie es hier vorgestellt wird, die Steuerung von Fahrzeugen mit menschlichen Passagieren übernehmen zu lassen.

## 7.4 Fazit

Trotz der erwähnten Probleme, ist die Ausgangsfrage, ob sich künstliche neuronale Netze zum Steuern von Fahrzeugen eignen, mit *ja* zu beantworten. Dass die Problemstellung nur im Rahmen einer Simulation getestet wurde und dass die Fahrzeugphysik nicht exakt der Realität entspricht, bedeutet jedoch, dass die Ergebnisse sich nicht ohne weiteres auf reale Fahrzeuge anwenden lassen. Gezeigt werden konnte nur, dass für das Steuern von Fahrzeugen in dem eingeschränkten Umfang der Simulation, neuronale Netze eine vielversprechende Lösung darstellen. Für die Anwendung im Straßenverkehr, müssen zunächst eine ganze Reihe weiterer Technologien zum Einsatz kommen und damit verbundene Probleme gelöst werden.

## 7.5 Ausblick

In dieser Arbeit wurde nur ein kleiner technologischer Aspekt von autonomen Autos untersucht. Es konnte gezeigt werden, dass künstliche neuronale Netze ein vielversprechendes Werkzeug darstellen, um das behandelte Teilproblem, mit einigen Einschränkungen, zu lösen. Es ist nun zu untersuchen, inwieweit die Bewertungsfunktion oder andere Teile der Simulation angepasst werden müssen, um die erwähnten Hindernisse zu überwinden. Außerdem wäre es sinnvoll die Ausgangsfrage nicht ausschließlich über Simulationen zu beantworten versuchen, sondern reale Fahrzeuge durch neuronale Netze steuern zu lassen. So könnten die Möglichkeiten deutlicher von den Grenzen abgegrenzt werden: welche Probleme der Steuerung können ausschließlich durch neuronale Netze gelöst werden? Wo müssen

andere Technologien eingesetzt werden? Können diese eventuell kombiniert werden? Die Beantwortung dieser Fragen ist aber nicht mehr Teil dieser Ausarbeitung.

# 9 Anhang

## 9.1 Klassendiagramme

# LITERATUR


[1] „statista.com," [Online]. Available: http://de.statista.com/statistik/daten/studie/185/umfrage/todesfaelle-im-strassenverkehr/. [Zugriff am 08 Mai 2016].

[2] M. K. Alex Forrest, „wpi.edu," 1 Mai 2007. [Online]. Available: http://www.wpi.edu/Pubs/E-project/Available/E-project-043007-205701/unrestricted/IQPOVP06B1.pdf. [Zugriff am 31 Mai 2016].

[3] A. M. Kessler, „nytimes.com," 19 März 2015. [Online]. Available: http://www.nytimes.com/2015/03/20/business/elon-musk-says-self-driving-tesla-cars-will-be-in-the-us-by-summer.html?_r=0. [Zugriff am 31 Mai 2016].

[4] „google.com," [Online]. Available: https://www.google.com/selfdrivingcar/. [Zugriff am 31 Mai 2016].

[5] Vijay John, Toyota Technological Institute, „Pedestrian detection in thermal images using adaptive fuzzy C-means clustering and convolutional neural networks," in *IAPR International Conference*, Tokyo, 2015.

[6] F. Streichert, „Universität Tuebingingen," April 2002. [Online]. Available: http://www.ra.cs.uni-tuebingen.de/mitarb/streiche/publications/Introduction_to_Evolutionary_Algorithms.pdf. [Zugriff am 16 08 2016].

[7] M. G. Xinjie Yu, „Representation and Evaluation," in *Introduction to Evolutionary Algorithms*, Springer, 2010, pp. 15-16.

[8] „Simple Genetic Algorithm Infrastructure," in *Introduction to Evolutionary Algorithms*, Springer, 2010, pp. 17-23.

[9] iforce2d, „Top-down car physics," 2014.

[10] M. Majurnder, „Artificial Neural Network," in *Impact of Urbanization on Water Shortage in Face of Climatic Aberrations*, Springer Singapore, 2015, pp. 49-54.

[11] Cliparts.co, „cliparts.co," [Online]. Available: http://cliparts.co/cliparts/M8T/GAB/M8TGABAia.png. [Zugriff am 1 Juni 2016].

[12] M. G. Xinjie Yu, „What are Evolutionary Algorithms used for?," in *Introduction to Evolutionary Algorithms*, Springer, 2010, p. 3.

# ABBILDUNGEN

# ERKLÄRUNG

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 14.08.16
Ort, Datum