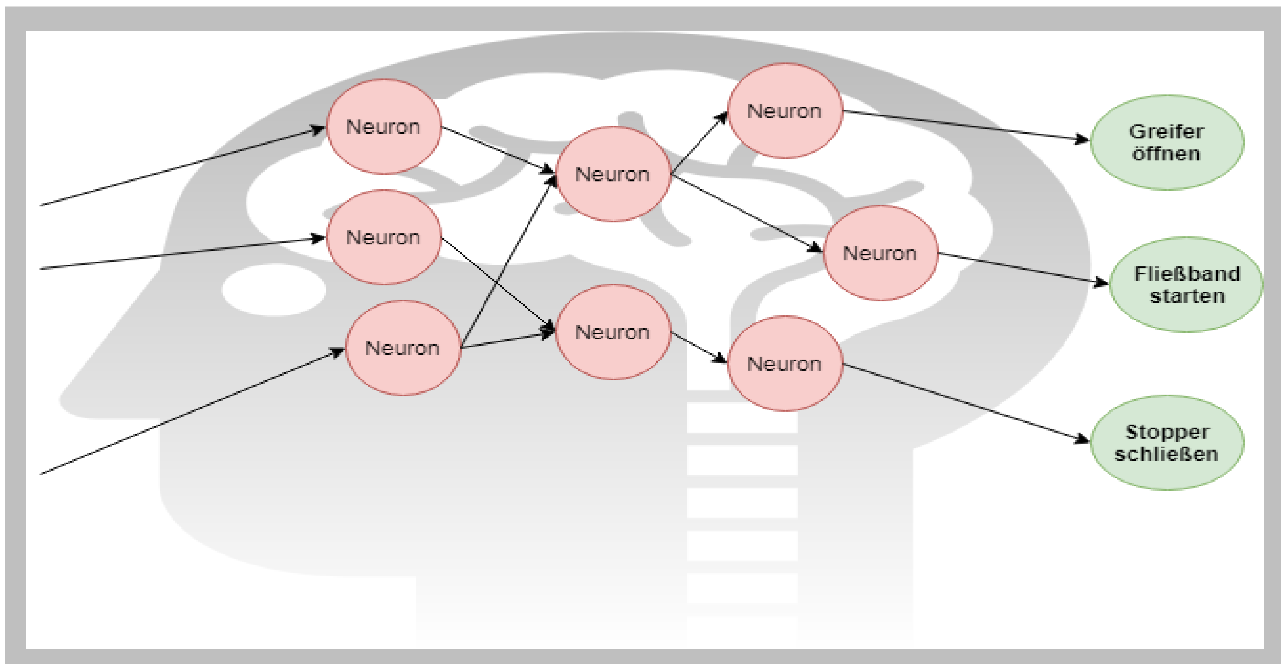
FAPS

Automatisches Ableiten von Steuerungsabläufen durch semantische Modellierung von Funktionen und Methoden der künstlichen Intelligenz

Bachelorarbeit im Studiengang Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Fertigungsautomatisierung und Produktionssystematik
Prof. Dr.-Ing. J. Franke



Bearbeiter: Patrick Lohnert, 21915680

Betreuer: M. Sc. Markus Brandmeier
M. Sc. Dennis Müller
Prof. Dr. M. Kohlhase

Abgabetermin: 25.09.2018
Bearbeitungszeit: 5 Monate

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 25.09.2018

____

Patrick Lohnert

Inhaltsverzeichnis

1 EINLEITUNG 1
1.1 Problemstellung und Stand der Wissenschaft 1
1.2 Ziele und Aufbau der Arbeit 2

2 METHODEN ZUR PROBLEMLÖSUNG 3
2.1 Vergleich von Methoden 3
2.2 Neuronale Netze 4
2.3 Verwendung von neuronalen Netzwerken in vergleichbarem Kontext 5
2.4 Handlungsbedarf 5

3 AUFBAU UND STRUKTUR DES PROGRAMMS 7
3.1 Allgemeine Struktur 7
3.2 Hauptkomponenten der Backend-Anwendung 7
3.3 Darstellung der Anlage 9
3.3.1 Schemenhafte Darstellung 9
3.3.2 Webdarstellung 10
3.3.3 XML-Darstellung 11
3.4 Phasen der Ausführung 12
3.5 Implementierung von neuronalen Netzen 13
3.6 Evolutionärer Algorithmus 15
3.6.1 Unterteilung in Spezies 16
3.6.2 Entfernen von Spezies 16
3.6.3 Vermehren von neuronalen Netzwerken 17
3.6.4 Arten von Mutation 18
3.7 Generierung von Steuerungsprogrammen 19

4 VALIDIERUNGSBEISPIEL UND EVALUATION 21
4.1 Beispielablauf an der Demonstratoranlage am FAPS 21
4.2 Anpassungen des Programms an das Beispiel 21
4.3 Beispielhafte Auswertung eines neuronalen Netzes 22
4.4 Ergebnisse 23
4.5 Evaluation und Verbesserungsvorschläge 24

5 WEITERE ANWENDUNGSGEBIETE UND AUSBLICK IN DIE ZUKUNFT 26

LITERATURVERZEICHNIS 28

ANHANG A 29

Abbildungsverzeichnis

Abbildung 1: Neuronales Netzwerk und dessen Bestandteile 4
Abbildung 2: Schemenhafte Darstellung der virtuellen Anlage 9
Abbildung 3: Aufbau der Webseite 10
Abbildung 4: Aufbau der XML-Datei 11
Abbildung 5: Verwendung von Eingabewerten 14
Abbildung 6: Beispielhaftes neuronales Netzwerk 14
Abbildung 7: beispielhafte Knotenmutation 18
Abbildung 8: Demonstratoranlage des FAPS [5] 21
Abbildung 9: beispielhaftes neuronales Netz 22

Abkürzungsverzeichnis

IO-Knoten	Inputknoten und Outputknoten
ML	Machine Learning
KI	Künstliche Intelligenz

1 Einleitung

Im folgenden Kapitel soll auf die Notwendigkeit zur automatisierten Ablaufprogrammerstellung aufmerksam gemacht und deren Vorteile dargestellt werden. Ebenfalls soll der Aufbau der Arbeit beschrieben werden.

1.1 Problemstellung und Stand der Wissenschaft

Steigende Variantenvielfalt und steigender Fachkräftemangel führen dazu, dass eine kleine Zahl an Mitarbeitern eine hohe Programmkomplexität und einen hohen Änderungsaufwand beherrschen muss. Heutzutage müssen qualifizierte Mitarbeiter on- oder offline teilweise jeden einzelnen Befehl und jede Variablenbelegung händisch eingeben, wodurch bei komplexen Aufgaben fehlerhafte und nicht optimierte Abläufe erstellt werden können. Hierdurch entstehen einerseits hohe Personalkosten, andererseits wird Produktionskapazität verschwendet. Wie lange kann die Komplexität noch steigen bis sie durch den Menschen nicht mehr beherrscht werden kann?

Machine Learning – kurz ML - ist eine Methode, die sich in der Industrie immer weiter ausbreitet. Im Vordergrund steht hierbei die automatische Generierung von Wissen. Oft wird ML in Verbindung mit neuronalen Netzen eingesetzt, welche flexibel an alle Themenbereiche angepasst werden können. Um die Funktionalität von Anlagen in einem neuronalen Netz abbilden zu können und damit je nach Dauer der Lernphase, mehr oder weniger gut optimierte Programmabläufe generieren zu können, muss neben neuronalen Netzen auch eine Anwendung implementiert werden, die diese Netze koordiniert. Ebenfalls sollte ein bereits trainiertes Netz in relativ kurzer Zeit mit Änderungen der Anlage umgehen können und dadurch eine hohe Verfügbarkeit schaffen. Da neue Produkte meist lange im Voraus geplant werden, könnte diese Anwendung offline bereits lange vor der Anlageninbetriebnahme den Ablauf-Code generieren und somit einen Stillstand der Maschinen verhindern.

Der Einsatz von neuronalen Netzen zur Ablaufgenerierung wird dadurch erschwert, dass für den Lernvorgang des Netzes kaum Ideallösungen existieren, mit welchen vom Netzwerk gelieferte Lösungen verglichen werden könnten. Somit wird ein anderer Bewertungsalgorithmus für diese Lösungen benötigt.

Achim Baier erklärt in einem Artikel über KI und ML, dass es an der Zeit ist, diese Themen in den Maschinenbau einzubinden. Er schildert, dass die Voraussetzungen für den Einsatz von Methoden dieser beiden Gebiete nun erfüllt sind. Dabei ist der technische Fortschritt im Bereich Hardware und die starke Vernetzung von Systemen ausschlaggebend. Große IT-Unternehmen haben mit Unterstützung von Universitäten und Forschungsinstituten eine grundlegende Architektur an Software im Bereich KI und ML zur freien Verfügung gestellt. Hieran können sich nun kleinere

Maschinenbau-Unternehmen bedienen und mit Hilfe von Softwareentwicklern komplexe Programme erzeugen. [1]

1.2 Ziele und Aufbau der Arbeit

Mit dieser Arbeit soll ein Grundstein für die automatisierte Ablaufgenerierung geschaffen werden, wobei ein funktionsfähiger Prototyp als Resultat erstellt werden soll. Als Ziel der Arbeit sollen Steuerungsabläufe für die Demonstratoranlage in den Gebäuden des FAPS generiert werden. Hierfür wird ein Programm entwickelt, welches den momentanen Zustand der Anlage und den gewünschten Endzustand als Input erhält und in Bezug auf die Reihenfolge und Durchlaufzeit einen möglichst guten Ablauf generieren soll.

Da der Umfang einer Bachelorarbeit bei weitem nicht ausreicht, um dieses neue Themengebiet komplett zu umfassen, wird diese Arbeit hinsichtlich Wieder- und Weiterverwendung gestaltet.

Zu Beginn wird in Kapitel 1 die Notwendigkeit zur Automatisierung der Ablaufgenerierung geschildert. In Kapitel 2 werden Methoden zur Lösungsfindung in Bezug auf Qualität und Dauer verglichen, wobei nach Komplexität der Anlage unterschieden werden muss. Das Kapitel 3 befasst sich mit dem Aufbau des zu entwickelnden Programms und erläutert dessen Funktionsweise. Anschließend wird in Kapitel 4 das entstandene Programm an einer simulierten und einer realen Anlage getestet und das Ergebnis bewertet. Ebenfalls werden Vorschläge zur Verbesserung des Programms und die entstandenen Erfahrungen geteilt. Zuletzt werden in Kapitel 5 weitere Anwendungsgebiete von künstlicher Intelligenz im Maschinenbau aufgezeigt und ein Ausblick in die Zukunft gegeben.

2 Methoden zur Problemlösung

In diesem Kapitel sollen verschiedene Methoden, um automatisiert Steuerungscode zu generieren, verglichen werden und auf geeignete Methoden genauer eingegangen werden.

2.1 Vergleich von Methoden

Bei der Erzeugung von Steuerungsabläufen beeinflussen viele Faktoren wie gut eine Methode geeignet ist, daher sollte die Verwendbarkeit und Eignung dieser Techniken und Algorithmen verglichen und geprüft werden. Steuerungsabläufe können von verschiedenster Komplexität sein. Einfache Abläufe bestehen aus einer kleinen Anzahl an Schritten, wobei diskrete Attribute aus einem kleinen Wertebereich verwendet werden. Diese Abläufe sind von geringerer Priorität für diese Arbeit sollten aber ebenfalls automatisch generierbar sein. Im Gegensatz hierzu werden Abläufe benötigt, welche sich aus einem großen Pool von komplexen Funktionen bedienen können und hierfür Attributwerte aus einem beliebig großen Wertebereich verwenden können. Die entstehende Komplexität führt zu Einbußen der Ergebnisqualität und/oder der Laufzeit.

Heutzutage werden Steuerungsabläufe in vielen Firmen per Hand an der Maschine oder remote an einem Computer entwickelt. Dies kann je nach Komplexität der Anlage und Können des Entwicklers zu verschieden guten Ergebnissen führen. Wird ein schlechtes Ergebnis nicht durch eine Simulation getestet kann dies zu Schäden und Ausfällen der Anlagen führen. Ebenfalls entstehen hierbei hohe Personalkosten und Stillstandzeiten der Maschinen. Selbst ein erfahrener Programmierer kann nicht immer den perfekten Ablauf bilden, wodurch die Produktion mit nicht optimierten Durchlaufzeiten arbeiten muss.

Mithilfe des sogenannten Brute-Force-Algorithmus können alle möglichen Kombinationen an Funktionen und Parametern durchlaufen und der ideale Ablauf gefunden werden. Mit einem Grundwissen in der Kombinationslehre ist leicht zu erkennen, dass die Anzahl an Kombinationen sehr schnell sehr groß werden kann und daher der Algorithmus nur für sehr einfache Abläufe geeignet ist.

Von einem Rechner können wesentlich mehr Abläufe in einer bestimmten Zeitspanne bewertet werden, als durch ein menschliches Gehirn. Menschen fällt es allerdings leichter irrelevante Abläufe herauszufiltern und zielorientiert in Richtung einer guten Lösung zu entwickeln, sodass die Anzahl an zu bewertenden Abläufen stark verringert wird. Nun sollen die Vorteile beider bisher genannten Methoden kombiniert werden. Um die Effizienz eines Rechners zu steigern, können neuronale Netzwerke eingesetzt werden, die zielgerichtet sinnvolle Abläufe generieren können. Durch den Lernvorgang der Netze wird die Generierung von schlechten Abläufen verringert. Nun muss die Ergebnisqualität geprüft werden.

2.2 Neuronale Netze

Die Bestandteile eines neuronalen Netzes sind Input-Knoten, Knoten im sogenannten Hidden-Layer, Output-Knoten und gewichtete Kanten zwischen diesen. Abbildung 1 zeigt ein solches neuronales Netz, inklusive dessen Bestandteile.

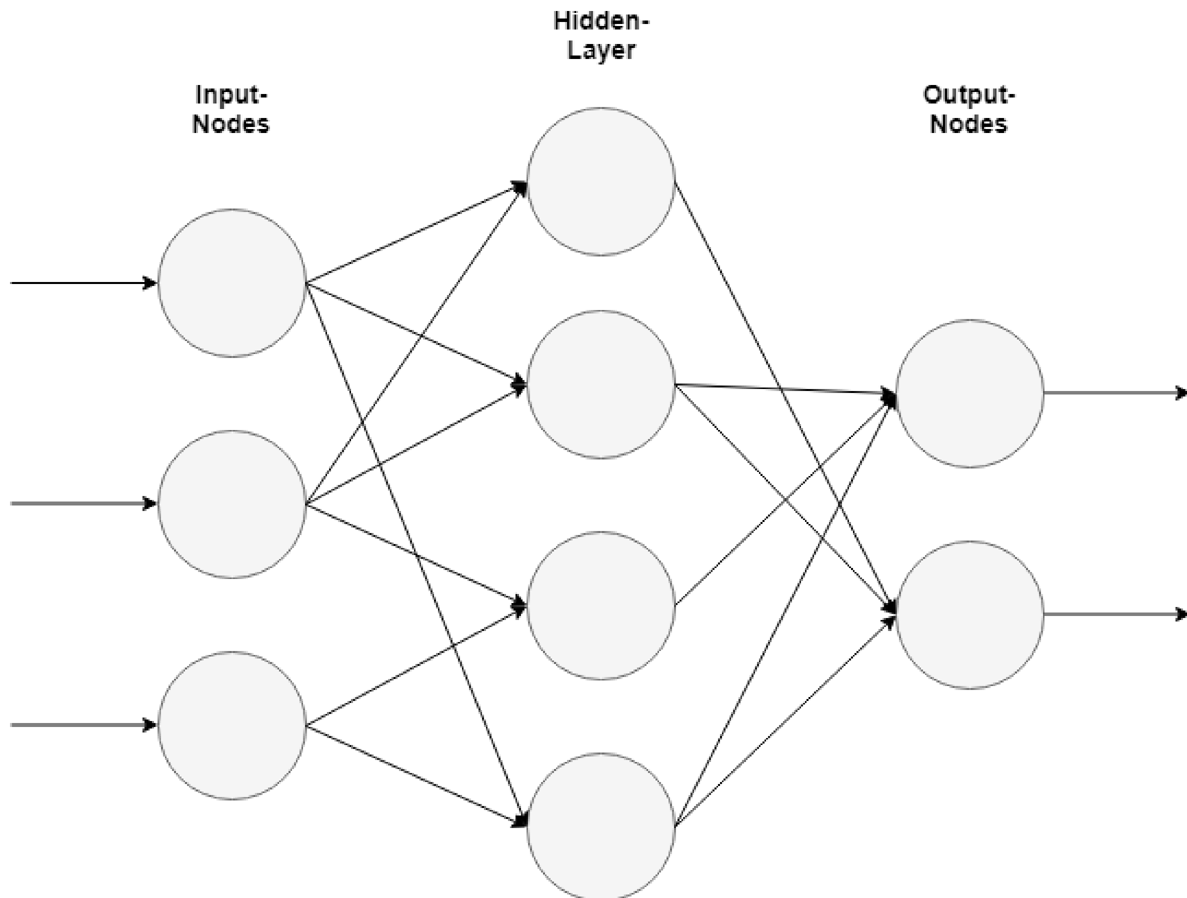


Abbildung 1: Neuronales Netzwerk und dessen Bestandteile

In- und Outputknoten werden von nun an durch den Begriff IO-Knoten abgekürzt. Bei der Auswertung werden zu Beginn bestimmte Werte an die Input-Knoten übergeben. Von diesen werden die Werte über Kanten zu weiteren Knoten gegeben, wobei der Gewichtungswert der Kante multipliziert wird. Die Ergebnisse werden bei den entsprechenden Knoten aufsummiert und anschließend meist mit einer Sigmoid-Funktion behandelt. Die Werte werden so lange weitergegeben bis sie Output-Knoten erreichen, von welchen das Ergebnis abzulesen ist. Ebenfalls kann eine Aktivierungsfunktion verwendet werden, um festzulegen, ob ein Output-Knoten aktiv ist. Eine solche Funktion wird immer dann verwendet, wenn das Ergebnis für eindeutige Entscheidungen genutzt werden soll. Neuronale Netze können zyklisch oder azyklisch aufgebaut sein.

Da meistens keine optimale Gewichtung bekannt ist, muss das Netz vor seiner Verwendung eine Lernphase durchleben, wobei das Netz meistens Probleme mit bekannten Lösungen berechnet, um im Anschluss die vom Netz gelieferte mit der bekannten Lösung vergleichen zu können. [2]

2.3 Verwendung von neuronalen Netzwerken in vergleichbarem Kontext

Das Ziel vieler Videospiele ist es, eine Spielfigur durch geschickte Kommandoeingabe über Hindernisse in das Ziel zu manövrieren und dabei möglichst viele Punkte zu sammeln. Hierbei können durch Eingabe falscher Kommandos Kollisionen und schlechte Ergebnisse erzielt werden. Zu jedem Zeitpunkt des Spielverlaufs ist somit nur ein Teil der möglichen Kommandos sinnvoll, sodass der Spieler eine Art Ablauf durch zeitlich abgestimmte Eingaben erzeugen muss, um das Spielziel zu erreichen.

Dieser Sachverhalt hat viele Gemeinsamkeiten mit der Generierung von Steuerungscodes für Anlagen, wobei eine Fachkraft Funktionen mit bestimmten Parametern in die korrekte zeitliche Abfolge einreiht.

Die Arbeit „Evolving Mario to Maximize Coin Score Using Neat and Novelty" von Amy und Jeremy Han befasst sich damit, mithilfe von Neuroevolution of augmenting Topologies (NEAT) und Novelty Search eine möglichst hohe Punktzahl bei einem bekannten Videospiel zu erreichen, wobei die Eingabe der Kommandos durch eine KI erfolgt. Sie prüfen, ob die beiden Algorithmen geheime Räume finden, welche die Punktzahl enorm steigern, aber schwierig zu erreichen sind. Im Anschluss prüfen sie welcher Algorithmus schneller ist. Sie zeigen, dass die Geschwindigkeit von der Umgebung und Schwierigkeit abhängt. [3]

Da die Aufgabe der automatischen Generierung von Steuerungsabläufen dem Sachverhalt dieses Videospiels stark ähnelt und hierbei das Problem offensichtlich gut gelöst wird, gilt es nun zu prüfen, ob sich die hierbei genutzten Methoden auf diese Aufgabe übertragen lassen.

Diese Arbeit konzentriert sich auf NEAT und bezieht sich dabei auf einen Artikel des MIT Press Journals. Dieser beschreibt verschiedenste Möglichkeiten die evolutionäre Entwicklung von neuronalen Netzwerken zu beschleunigen. Hierbei sticht besonders die Eingliederung von neuronalen Netzen in verschiedene Sparten vergleichbar mit Rassen in der Natur heraus. [4]

2.4 Handlungsbedarf

Die Anzahl der Menschen beziehungsweise Verbraucher steigt rapide, während Modellintervalle immer kleiner werden, sodass Produkte in kürzerer Zeit entwickelt und

anschließend in größeren Zahlen produziert werden müssen. Die Anforderungen an Maschinenbauunternehmen steigen stets an und erhöhen den Druck auf diese. Hinzu kommt, dass die Anzahl der Fachkräfte begrenzt ist.

Es ist nur eine Frage der Zeit bis die Unternehmen die hohe Nachfrage nicht mehr decken können, daher sollten Fachkräfte weitestgehend durch automatisierte Vorgänge entlastet werden und die freiwerdende Arbeitskraft in den noch nicht automatisierbaren Bereichen eingesetzt werden.

Viele Bereiche der Industrie sind bereits automatisiert und erzeugen mit einer verhältnismäßig geringen Anzahl an Arbeitern einen großen Output. Die Generierung von Steuerungsprogrammen und ähnliche Aufgaben der Entwicklung von Produkten sind nur geringfügig automatisiert, könnten aber, wie diese Arbeit zeigt, beinahe vollautomatisiert realisiert werden. Für Unternehmen bedeutet dies ein hohes Kosteneinsparungspotential und gibt ihnen die Möglichkeit den Fachkräftemangel in anderen Bereichen zu decken. Neben den Lohnkosten, die eingespart werden können, bietet die automatisierte Generierung der Steuerungsprogramme die Chance Maschinenstillstände der Onlineprogrammierung, also der Erzeugung von Programmen an der Maschine, komplett zu verhindern.

Da viele Methoden der KI schon jetzt eingesetzt werden können, besteht die Chance schon jetzt Erfahrung in diesen Bereichen zu sammeln und diese in den kommenden Jahren zu verbessern, wodurch das Wachstum der Industrie stark angekurbelt werden kann. Für einzelne Unternehmen bedeutet dies einen großen Wettbewerbsvorteil und die Möglichkeit dies zu einer ihrer Kernkompetenzen zu entwickeln. Der anfängliche Investitionsbedarf kann durch den baldigen Einsatz der Automatisierung schnell gedeckt werden.

Wird die Forschung an Methoden der künstlichen Intelligenz vorangetrieben, so können diese bald flexibel in allen möglichen Bereichen der Technik und Forschung eingesetzt werden, wodurch viele Probleme schneller gelöst werden können. Die Ausbildung von guten Forschern dauert mehrere Jahrzehnte, wohin im Gegensatz die Anpassung von Netzwerken an Forschungsprojekte wesentlich schneller zu vollbringen ist. Hält die zeitlich konstante Verbesserung von Prozessoren an, lässt sich nur erahnen, wann Computer menschliche Forschung unwirtschaftlich machen, daher ist es sinnvoll Forschungskapazität in die Weiterentwicklung der KI zu stecken.

3 Aufbau und Struktur des Programms

Kapitel 3 befasst sich mit dem Aufbau und der Strukturierung des Programms, welches unter Angabe des momentanen Zustands und des gewünschten Endzustands der Anlage, automatisch den Steuerungscode generieren soll.

3.1 Allgemeine Struktur

Das Programm ist unterteilt in eine Java-Backend-Anwendung, inklusive eines Webservers und eine Javascript-Frontend-Anwendung in Form einer Webseite. Die Kernaufgaben der Webseite sind die Darstellung der Anlage in ihrem momentanen Zustand und die Interaktion mit dem Nutzer, beispielsweise kann dieser hierüber steuern, ob die neuronalen Netze trainiert werden oder zum Generieren von Steuerungscodes eingesetzt werden sollen. Die Backend-Anwendung kümmert sich um die Population der Netzwerke, die Interpretation von Anlagenzuständen und die Bereitstellung eines Server-Sockets, der zur Kommunikation mit der Webseite dient und bei Bedarf Dateien an diese verschickt. Die Unterteilung in zwei Anwendungen hat den entscheidenden Vorteil, dass die Anlage hierdurch remote gesteuert und eine Simulationsdarstellung angezeigt werden kann.

3.2 Hauptkomponenten der Backend-Anwendung

Die Aufgabe der Kommunikation mit der Webseite wird von einer Server-Klasse übernommen, die für jeden neuen Nutzer einen neuen Thread startet. In diesem Thread werden die Eingaben und Anforderungen des Nutzers behandelt und an die entsprechenden Komponenten weitergeleitet.

Die Anlagenlogik wird durch verschiedene Klassen implementiert:

- **Part**: Diese abstrakte Klasse dient als Vorlage für Teile-Klassen, wie beispielsweise die Greifer-Klasse „Grab“. Durch Verwendung der Methode „addOperation(Operation)“ können Teile-Operationen, wie das Bewegen des Greifers, hinzugefügt werden. Für jedes funktionale Teil muss die Funktion „distanceToState(String, int, int)“ überschrieben werden, die unter Angabe des Attributnamens, dem Ist- und Sollzustand einen Kostenwert liefert.

- **Plant**: Ein Objekt dieser Klasse lädt die Teile aus der entsprechenden Datei im XML-Format. Ebenfalls ist das Objekt für die Berechnung des Steuerungsablaufs mit Hilfe eines neuronalen Netzes verantwortlich.

- **PartState**: Diese Klasse dient zur Verknüpfung aller Attribute eines Teils mit den entsprechenden Attributwerten. Ebenfalls enthält sie eine Methode, die die Kostenfunktion jedes Attributs aufruft und das Ergebnis aufsummiert.

- **PlantState**: Jedes Objekt dieser Klasse enthält den Zustand „PartState“ aller Teile der Anlage und die Funktion „distanceToOtherState(PlantState)“, die die Gesamtkosten zwischen den momentanen und den Zielzuständen berechnet. Hierbei werden die Kosten aller Teile summiert und das Ergebnis letztendlich als Bewertung des Netzwerkes genutzt.

- **Operation**: Diese Klasse enthält ein Attribut des Typen „Function<Plantstate, OperationResult>“, welchem eine Funktion zugewiesen werden muss. Diese erhält einen PlantState und gibt ein OperationResult zurück. Außerdem enthält sie eine Liste von Bedingungen „ArrayList<Condition>“, die vor der Ausführung der Operation erfüllt sein müssen. Diese Klasse beschreibt somit die Operationen von Teilen der Anlage.

- **OperationResult**: Ein Objekt dieser Klasse enthält den Zielzustand der Operation und den Steuerungscode, der zu diesem Zustand führt.

- **Condition**: Diese Klasse enthält ein Attribut des Typen „Function<PlantState, Boolean>“, welchem eine Funktion zugewiesen werden muss. Diese Funktion erhält den momentanen Anlagenzustand und gibt zurück, ob die Bedingung erfüllt ist. Mit dieser Klasse werden Vorbedingungen für Operationen ermöglicht.

- **PlantTester**: Diese Klasse ist entscheidend für die Lernphase der Anwendung. Sie enthält 6 Zielzustände und eine Methode, die den Ablauf vom momentanen Zustand der virtuellen Anlage, zu diesen Zielzuständen simuliert und bewertet. Die Bewertung wird in Kapitel 3.7 erläutert.

Die Implementierung des evolutionären Algorithmus und den neuronalen Netzwerken ist in folgende Klassen gegliedert:

- **NetworkEvolver**: Diese Klasse dient zur Koordination aller folgenden Klassen und enthält eine Liste aller Netzwerke. Ebenfalls entscheidet sie welche Netzwerke überleben und welche zu schwach sind beziehungsweise sich zu langsam verbessern.

- **NeuralNetwork**: Ein Objekt dieser Klasse enthält alle Informationen über das neuronale Netz. Darunter fallen Input-Neuronen, Output-Neuronen, Gene und Neuronen. Ebenfalls enthält jedes Objekt dieser Klasse seine spezifischen Mutationschancen.

- **Species**: Diese Klasse hat die Aufgabe die neu entstandenen Netzwerke in Spezies einzuteilen und innerhalb dieser Spezies die Netzwerke nach den Kosten zu sortieren.

- **Gene**: Diese Klasse stellt das Verbindungselement zwischen Neuronen dar. Gene enthalten ein Gewichtungsfaktor-Attribut, eine Innovationsnummer und Indizes ihrer In- und Output-Neuronen.

- **Neuron**: Diese Klasse ist das Gegenstück zur Klasse „Gene“ und ist in dieser Implementierung notwendig, um die Eingangswerte aufzusummieren, umzuwandeln und weiterzugeben. Außerdem dient sie zur Koordinierung der Menge an Eingangswerten und enthält eine Funktion, die als Ergebnis liefert, ob alle Eingangswerte angekommen sind.

3.3 Darstellung der Anlage

Wie bereits in Kapitel 3.1 geschildert, besteht die Applikation aus zwei Anwendungen und erzeugt somit Synchronisierungs- und Kommunikationsaufwand, deshalb werden nun mehrere Darstellungen und Schemata aufgeführt, die diesen Aufwand behandeln.

3.3.1 Schemenhafte Darstellung

Eine Komplexitätsreduktion wird durch eine schemenhafte Darstellung erzielt. Komplexe Teile werden nach ihrer Funktion zusammengefasst und durch simple Objekte und Operationen abgebildet. Die Beispielanlage dieser Arbeit wird auf 2 Fließbänder reduziert, welche je 4 Positionen besitzen. Zwischen diesen 8 Positionen kann ein Werkstückträger im Uhrzeigersinn verschoben werden. Des Weiteren gibt es 4 Greiferpositionen, zwischen welchen flexibel gewechselt werden kann. Dieses Schema wird in Abbildung 2 dargestellt.

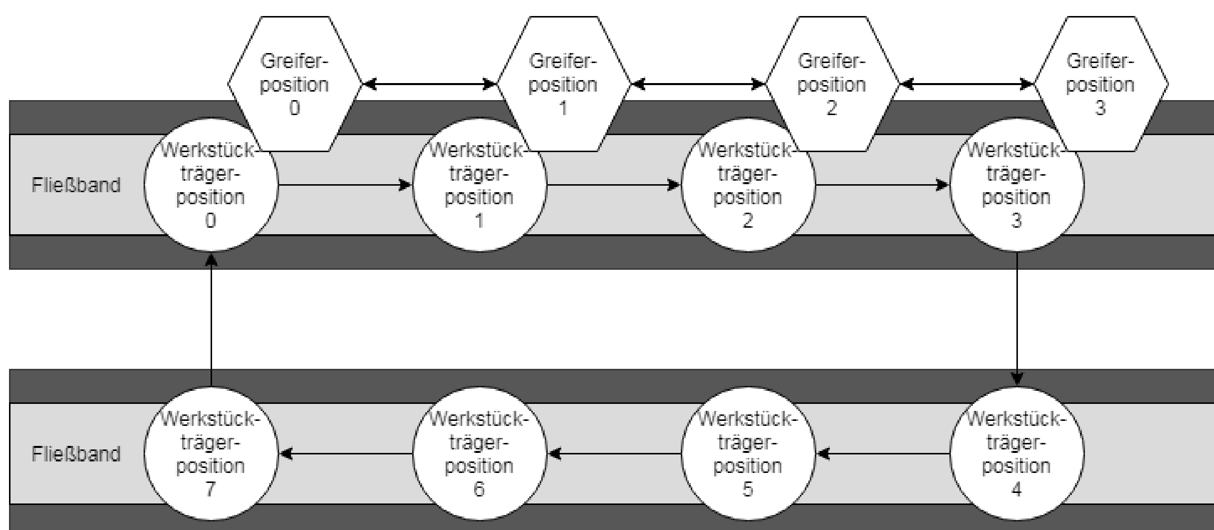


Abbildung 2: Schemenhafte Darstellung der virtuellen Anlage

3.3.2 Webdarstellung

Die Frontend-Anwendung liegt in Form einer Webseite vor, die eine 3D-Darstellung der Anlage enthält. Abbildung 3 zeigt den Aufbau dieser Webseite. Die Anzeige kann mit Hilfe der linken Maustaste gedreht werden und durch das Mausrad vergrößert oder verkleinert werden.

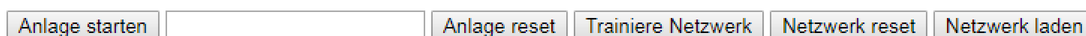


Abbildung 3: Aufbau der Webseite

Ebenfalls stellt die Webseite weitere Möglichkeiten der Interaktion mit der Anlage und der Netzwerkpopulation zur Verfügung. Ein vorher trainiertes Netzwerk kann durch den Knopf „Netzwerk laden“ geladen werden. Der Trainingsvorgang kann durch Drücken des Knopfes „Netzwerk trainieren“ beliebig oft gestartet und durch erneutes Drücken gestoppt werden. Durch das Stoppen wird der Speichervorgang ausgelöst und das Netzwerk wird durch den Server in einer Datei gespeichert. Der Knopf „Netzwerk reset“

setzt die Datei auf ihren Urzustand zurück und ermöglicht ein erneutes Training von Grund auf.

Wird in das Textfeld der Index – in dieser Implementierung zwischen 0 und 5 – einer Testanlage des NetworkEvolver-Objekts eingetragen und der Knopf „Anlage starten" gedrückt, erfolgt ein Post-Request über das Netzwerk, welcher vom Server eine Antwort erhält. Diese Antwort enthält einen Body mit einem für die Webseiten-Skripte verständlichen Steuerungscode. Serverseitig wird der Steuerungscode für die Webseite auf eine analoge Weise wie der Steuerungscode der tatsächlichen Anlage generiert. Dies wird in Kapitel 3.7 behandelt.

3.3.3 XML-Darstellung

Die Webdarstellung benötigt Informationen über das darzustellende Modell der Anlage. Hierfür bietet sich die folgende XML-Struktur aus Abbildung 4 an, da auf recht einfachem Weg Dateinamen und Attribute, wie Typ, Name und Position jedes Teils gespeichert und abgerufen werden können.

```
<?xml version="1.0"?>
<plant>
    <part id="1">
        <type>Grab</type>
        <name>Grab 1</name>
        <model>linearunit3.obj</model>
        <position>0</position>
        <modelposition>7.27 4 -1.16</modelposition>
    </part>
    <part id="2">
        <type>Conveyor belt</type>
        <name>Conveyor belt</name>
        <model>Lineareinheit_fest_und_Band.obj</model>
        <modelposition>0 0.29 0</modelposition>
    </part>
    <part id="3">
        <type>Workpiece carrier</type>
        <name>Workpiece carrier1</name>
        <model>workpiececarrier.obj</model>
        <position>7</position>
        <modelposition>7.27 0 -6.03</modelposition>
    </part>
    <part id="4">
        <type>Workpiece carrier</type>
        <name>Workpiece carrier2</name>
        <model>workpiececarrier.obj</model>
        <position>0</position>
        <modelposition>7.27 0 -1.16</modelposition>
    </part>
</plant>
```

Abbildung 4: Aufbau der XML-Datei

Die Datei bietet ebenfalls die Möglichkeit die Daten für Front- und Backend-Anwendungen an einem Ort zu speichern, somit können Änderungen automatisch auf beiden Seiten synchronisiert werden.

Nun soll die Struktur genauer erläutert werden:

- Der Oberknoten „<plant>“ enthält alle Teile „<part>“.
- Die Knoten „<part>“ besitzen eine einzigartige ID und weitere Attribute.
- Das Attribut „<type>“ gibt der Anwendung vor, welche Unterklasse von Part für die Instanziierung verwendet werden muss.
- Das Attribut „<name>“ ist eine Möglichkeit zur Unterscheidung zwischen mehreren Teilen gleichen Typens.
- Das Attribut „<model>“ beinhaltet den Dateinamen des 3D-Modells. In dieser Implementierung sind dies Modelle im „.obj“-Format.
- Die Positionsattribute „<position>“ und „<modelposition>“ sind verschieden zu interpretieren. „<position>“ ist ein optionales Attribut, welches die Position im Schema aus Kapitel 3.3.1 beschreibt. „<modelposition>“ ist das Attribut, das die Position des Modells im 3-diminsionalen Raum beinhaltet.

Es ist möglich weitere Attribute hinzuzufügen, um mehr Funktionalität zu ermöglichen, allerdings muss das Programm dementsprechend angepasst werden. Die XML-Datei wird in der Klasse „Plant“ importiert.

3.4 Phasen der Ausführung

Vor Beginn der Ausführung muss das Anlagenschema durch Unterklassen der Klasse "Part" implementiert werden. Hierbei werden die einzelnen Teile inklusive ihrer Operationen in das Programm eingebracht und bieten somit eine hohe Flexibilität und Kompatibilität. Verschiedenste Anlagen können durch eine relativ einfache Programmierung implementiert werden.

Die Evolution wird mit minimalen neuronalen Netzen begonnen, da sich neue Strukturen inkrementell bilden können und hierbei gegenüber dem Start mit zufälligen Netzwerken ein zeitlicher Vorteil entsteht, da der Suchraum verkleinert wird. [4]

Da das neuronale Netz zu Beginn nur aus IO-Knoten besteht, benötigt dieses eine relativ lange Lern-/ Evolutionsphase, welche dazu dient, möglichst gute Strukturen aufzubauen. Während dieser Phase bilden sich immer wieder neue Verbindungen und Knoten, mit dem Ziel, die Funktionsweise der Anlage im Aufbau des Netzes zu speichern und möglichst gute Entscheidungen treffen zu können. Um lernen zu können, benötigt das Netz Testszenarien in Form von Start und Endzuständen und einen Bewertungsalgorithmus für die Qualität der Lösung.

Um nun Steuerungscode zu generieren, kann das Netz entweder speziell auf diesen Fall trainiert werden oder, insofern es bereits ausreichend viele Evolutionsstufen hinter sich hat, direkt mit den gewünschten Daten gespeist werden. Die Qualität der Lösung

steht in Relation mit der Zeit, die für das Lernen des Netzwerks verwendet wurde, da dieser evolutionäre Algorithmus stochastischer Natur ist und somit eine gewisse Anzahl an Generationen benötigt, um zuverlässig gute Ergebnisse liefern zu können.

3.5 Implementierung von neuronalen Netzen

Neuronale Netze können auf viele Weisen implementiert werden und sollten auf den Anwendungsfall angepasst werden. Die kleinste Menge an Informationen, durch die ein Netzwerk konstruiert werden kann, ist die Menge von Kanten, die die Knoten über eindeutige Indizes referenzieren und einen Gewichtungsfaktor besitzen. Ebenfalls wird die Menge der Indizes von IO-Knoten benötigt. In objektorientierten Programmiersprachen wie Java kann jedes Knoten-Objekt die Kanten referenzieren, die von ihm wegführen und jedes Kanten-Objekt den Knoten, zu dem es führt, allerdings würde diese Implementierung einen sehr großen Aufwand beim Kopieren von Netzwerken erzeugen. Da bei einem evolutionären Algorithmus offensichtlich viele Generationen von Netzwerken kopiert oder gekreuzt werden, wird in dieser Arbeit nur eine Liste von Kantenobjekten „ArrayList<Gene>“ und jeweils eine Liste an Integer-Werten der Indizes von IO-Knoten gespeichert. Für Auswertungszwecke wird ebenfalls für jeden Index ein Neuron in eine Liste eingereiht. Dieser Index entspricht entweder dem Listenplatz oder einem zufälligen einzigartigem Wert.

Die Netzwerke sind frei von Zyklen, um eine geregelte Auswertung zu gewährleisten. Dies führt dazu, dass bei äquivalenten Netzwerken, die unterschiedlich abgespeichert sein können, bei gleicher Eingabe die gleiche Ausgabe generiert wird.

Die Input-Knoten werden nach der Summe an möglichen Zuständen aller Teile der Anlage erzeugt. Für jeden Zustand, den ein Attribut annehmen kann gibt es zwei Input-Knoten. Einer dieser Knoten referenziert sich auf den Ist-Zustand und einer auf den Soll-Zustand. Gibt es beispielsweise eine Anlage mit einem Teil, welches ein Boolean-Attribut besitzt, dann besitzt das entsprechende neuronale Netz vier Input-Knoten. Die Anzahl der Output-Knoten richtet sich nach der Anzahl aller möglichen Operationen der Teile der Anlage. Ebenfalls gibt es einen weiteren Output-Knoten der eine „tue nichts“-Operation verkörpert.

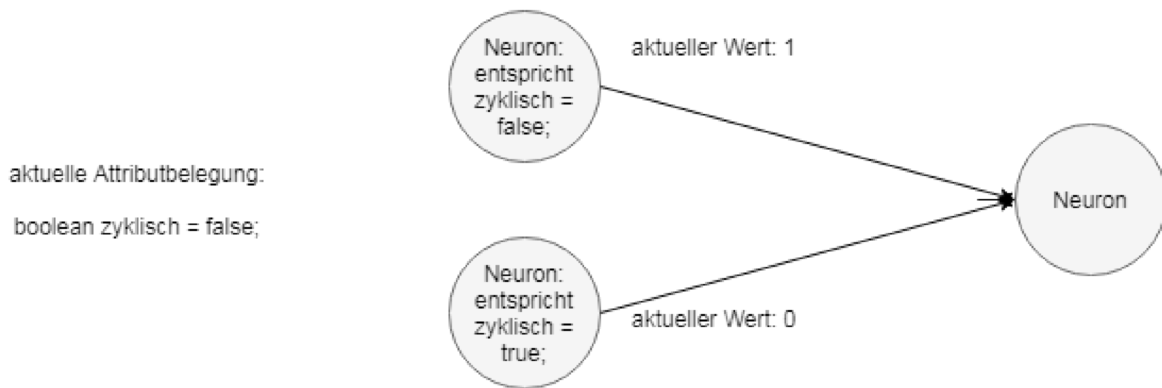


Abbildung 5: Verwendung von Eingabewerten

Um ein neuronales Netz auszuwerten, muss eine Eingabe gemacht werden. Für diese Eingabe werden alle Input-Knoten mit dem Wert 0 belegt, außer die Knoten, die den aktuellen Attributzuständen entsprechen. In Abbildung 5 wird eine beispielhafte Eingabe gezeigt. Für das Attribut „boolean zyklisch" existieren vier Neuronen, um die entsprechenden Attributzustände abzudecken. Da der aktuelle Attributwert „false" ist, wird der Wert des entsprechenden Neurons auf 1 gesetzt. Die Neuronen für den Zielzustand wurden in dieser Abbildung nicht dargestellt.

Zu Beginn der Evolution werden Netzwerke erzeugt, die nur aus diesen IO-Knoten bestehen, welche bei ihrer Erzeugung Indizes erhalten, die über den gesamten Evolutionsprozess für alle Netzwerke konsistent und gleich bleiben. Von nun an werden diese Netze Netzwerkstümpfe genannt.

Bei dem Kopiervorgang von Netzwerken wird zu Beginn ein Netzwerkstumpf mit den bekannten IO-Knoten-Indizes erzeugt. Anschließend wird über die Liste der Gene des zu kopierenden Netzwerks iteriert und für jeden Eingangs- oder Ausgangsindex geprüft, ob er in der Liste der Knoten enthalten ist. Ist er nicht enthalten wird ein neuer Knoten mit dem entsprechenden Index erzeugt.

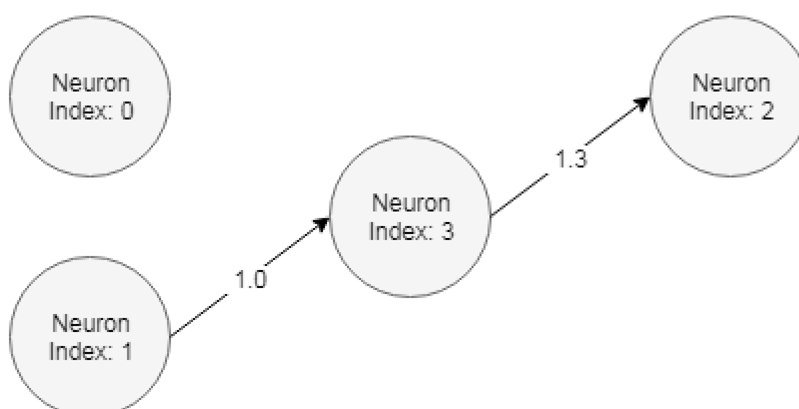


Abbildung 6: Beispielhaftes neuronales Netzwerk

Abbildung 6 zeigt ein Beispielnetzwerk. Ein NeuralNetwork-Objekt dieses Netzwerks besitzt folgende Attributwerte:

- Die Liste der Indizes der Input-Neuronen: [0, 1]
- Die Liste der Indizes der Output-Neuronen: [3]
- Eine Liste, die zwei Gene enthält:
  - Gen 1: Eingangsindex 1, Ausgangsindex 3, Gewichtungsfaktor 1.0
  - Gen 2: Eingangsindex 3, Ausgangsindex 2, Gewichtungsfaktor 1.3
- Eine Liste, die 4 Neuronen enthält mit den Indizes 0 bis 3

Diese Implementierung ist selbst für große Netzwerke sehr Speicherplatzeffizient und ermöglicht schnelle Kopiervorgänge. Hierdurch wird die Lernphase verkürzt beziehungsweise die Leistung in einer gewissen Zeit erhöht.

3.6 Evolutionärer Algorithmus

In dieser Arbeit wird ein Algorithmus verwendet, der sich sehr ähnlich zur natürlichen Evolution verhält. Hierbei wird mit 300 neuronalen Netzen in Generation 1 begonnen, welche nur aus IO-Knoten bestehen und keinerlei Kanten besitzen. Um einen noch näheren Bezug zur Natur zu erreichen werden Knoten auch als Neuronen und Kanten auch als Gene bezeichnet. Ziel jedes Generationszyklus besteht darin, die besten Netzwerke herauszufiltern und zu vermehren.

Damit sich neuronale Netze verbessern können, muss es eine Art von Mutation geben. Wenn zufällige, vom Computer ausgewählte Werte einen bestimmten Grenzwert überschreiten kommt es zu Mutation, das heißt es bilden sich neue Gene und/oder Neuronen oder die Gewichtungsfaktoren der Gene verändern sich, somit verändert sich das neuronale Netz.

Der einfachste Gedanke ist die X-besten Netzwerke auszuwählen und zu vermehren bis die Gesamtzahl an Netzen wieder 300 erreicht, allerdings werden hierbei Netzwerke aus dem Weg geräumt, die gewünschte Gene enthalten aber insgesamt schlechter abschneiden als andere. Eine bessere Strategie ist, die Netze in Spezies aufzuteilen und die X-besten jeder Spezies mit anderen Netzwerken, welche in ihrer Spezies zu den X-besten Zählen, zu vermehren, da dadurch sich langsamer entwickelnde Spezies, welche das Potential zu einer insgesamt besseren Leistung besitzen, ebenfalls die Chance auf Entwicklung erhalten. Bei diesem Programm werden bei jeder Generation die Spezies ausgelöscht, deren bestes Netzwerk im Vergleich zu dem Besten der anderen Spezies einen unterdurchschnittlichen Leistungswert erzielt. Kurz gesagt: Die schlechtere Hälfte aller Spezies wird entfernt um Platz für neue Spezies zu schaffen.

3.6.1 Unterteilung in Spezies

Um die Frage zu beantworten, wann ein Netzwerk einer bestimmten Spezies angehört, muss geklärt werden wann Netzwerke - in diesem Kontext auch Genome genannt - gleiche Gene besitzen. Entsteht bei der Mutation ein neues Gen, erhält dieses eine sogenannte Innovations-ID. Um die Ähnlichkeit zweier Netzwerke zu berechnen wird nun die Anzahl unterschiedlicher Gene, also Gene, die eine unterschiedliche Innovations-ID besitzen und das Delta der Gewichtungsfaktoren gleicher Gene berechnet. Unterschreiten diese Werte einen bestimmten Grenzwert werden die Netze der gleichen Spezies zugeordnet.

3.6.2 Entfernen von Spezies

Da die maximale Anzahl an Netzwerken auf 300 begrenzt ist, müssen bestehende Netzwerke gelöscht werden um Platz für neue Netze zu schaffen. Die Aufgabe besteht darin, diejenigen Netze zu entfernen, die das geringste Potential besitzen ein Ergebnis zu erzielen, das dem aktuellen Evolutionsgrad gerecht wird.

Der Algorithmus, der in dieser Implementierung verwendet wird, ist in zwei Phasen unterteilt und hat den folgenden Aufbau:

Phase 1:

1. Berechne den durchschnittlichen Rang aller Netzwerke einer Spezies.
2. Summiere alle durchschnittlichen Ränge innerhalb einer Spezies und speichere diesen Wert in der Variablen „avarageRankSum“.
3. Die Spezies, deren bestes Netzwerk die gleichen oder niedrigere Kosten als das zehntbeste Netzwerk besitzen, überleben.
4. Spezies, die sich in den letzten 15 Generationen verbessert haben, dürfen nicht gelöscht werden.
5. Ebenfalls überleben die Spezies, deren durchschnittlicher Rang („avarageRank“) folgende Gleichung erfüllt:

$$\frac{avarageRank}{avarageRankSum * networkCount} >= 1$$

mit:		
	networkCount	Gesamtanzahl an Netzwerken
	avarageRank	durchschnittlicher Rang der Netzwerke einer Spezies
	avarageRankSum	Summe der durchschnittlichen Ränge der Netzwerke einer Spezies

Phase 2:

1. Iteriere über die überlebenden Spezies und addiere für jede Spezies mit genau einem Netzwerk den Wert 1, für alle anderen den Wert 2, da anschließend alle Spezies auf ihre besten zwei Netzwerke reduziert werden.
2. Ziehe nun diesen Wert von 300 ab und das Ergebnis gibt die Anzahl noch zu erzeugender Netzwerke an. Anhand dieser Anzahl werden nun neue Netzwerke erzeugt, indem eine zufällige Spezies ausgewählt wird und ein neues Netzwerk generiert. Dieses wird allerdings noch nicht in eine Spezies eingeordnet.
3. Entferne nun alle Netzwerke aus allen Spezies, bis auf die jeweiligen besten zwei.
4. Ordne die neu generierten Netzwerke in die Spezies ein und definiere gegebenenfalls neue Spezies.

Nun sollte die Population wieder 300 Netzwerke umfassen. Im Programm wird dieser Algorithmus durch die Methode „clearWeakSpecies()“ der Klasse „NetworkEvolver“ umgesetzt.

3.6.3 Vermehren von neuronalen Netzwerken

Nachdem die schlechteren Spezies aussterben soll die Population wieder ihren ursprünglichen Wert von 300 erreichen. Dieses Ziel wird durch zwei verschiedene Vorgänge erreicht:

1. Vermehrung von einzelnen Netzwerken:

   Das Netzwerk inklusive dessen Mutationsraten wird dupliziert und anschließend wird das neu entstandene Netz mutiert und erneut in die noch vorhandenen oder in eine neue Spezies eingestuft.

2. Kreuzung zweier Netzwerke:

   Die Gene beider Netzwerke, die die gleiche Innovations-ID besitzen, werden zufällig von einem der beiden Netze ausgewählt, einem neuen Netzwerk hinzugefügt und um die nötigen Neuronen ergänzt. Somit entsteht ein Netz mit den Gemeinsamkeiten der Eltern-Netze mit zufällig aus den Eltern-Genen ausgewählten Gewichtungsfaktoren. Die Mutationsraten werden kopiert und anschließend mutiert das Netzwerk.

Im Programm wird die Erzeugung neuer Netzwerke mit Hilfe von bekannten Spezies durch die Methode „breed()“ der Klasse „Species“ umgesetzt.

3.6.4 Arten von Mutation

Jedes neuronale Netz besitz verschiedene Mutationsraten, die beeinflussen welche Mutationen stattfinden werden. Die Raten sind Fließkommazahlen.

Die Mutationsraten sind wie folgt unterteilt:

- MutateConnection
- MutateLink
- MutateNode
- MutateStepSize

Jedes neue Netzwerk durchläuft einen Mutationszyklus und somit folgenden Ablauf:

1. Die Mutationsraten des Netzes werden zufällig vergrößert oder verkleinert. Dies geschieht auch für ältere Netze.
2. Wenn ein Zufallswert kleiner als **MutateConnection** ist, tritt eine Punktmutation auf, das heißt alle Gene erhalten die Chance neu gewichtet zu werden. Dies ist entweder abhängig vom bisherigen Gewicht und der **MutateStepSize** oder ihnen wird ein komplett neuer Wert zwischen –2 und 2 zugewiesen.
3. Der abgerundete Wert von **MutateLink** gibt vor, wie oft Linkmutation auftritt. Hierbei werden zufällig 2 Neuronen ausgewählt und durch ein Gen verbunden, insofern nicht beide Input-Neuronen sind.
4. Der abgerundete Wert von **MutateNode** entscheidet, wie oft eine Knotenmutation stattfindet. Bei der Knotenmutation wird ein Gen "aufgetrennt" und ein neues Neuron eingefügt. Diese wird in Abbildung 7 dargestellt.
5. Letztendlich werden die entstandenen Zyklen entfernt.

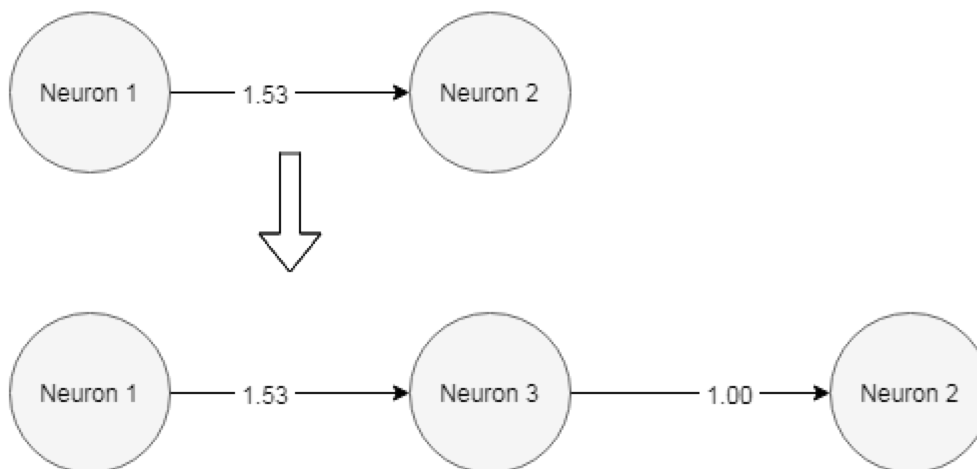


Abbildung 7: beispielhafte Knotenmutation

Die Abbildung 7 zeigt eine Knotenmutation, wobei Neuron 3 zwischen Neuron 1 und 2 und ein neues Gen zwischen Neuron 3 und 2 eingefügt werden.

3.7 Generierung von Steuerungsprogrammen

Die Möglichkeit zur freien Programmierung der Teile-Klassen, welche die abstrakte Klasse „Part“ erweitern, bietet den Vorteil Steuerungscode direkt und variabel im Java-Code einzubinden. Wie schon im Kapitel 3.2 erwähnt, existiert die Klasse „OperationResult“, welche das Resultat von Operationen ist. Die Klasse „OperationResult“ enthält für die Webseitensteuerung und die Anlagensteuerung je ein String-Attribut, das über das Netzwerk an das entsprechende Ziel gesendet werden kann.

Ein fertiges Steuerungsprogramm entspricht in dieser Implementierung einer Liste von „OperationResult“-Objekten. Bei der Erzeugung dieser Liste ist zu unterscheiden, ob dies für die Lernphase oder das Endergebnis geschieht. In der Lernphase müssen Abläufe generiert werden, die zur Bewertung herangezogen werden. Hierbei wird für jeden Zielzustand aus der Liste im „PlantTester“ wie folgt ein Ablauf generiert:

1. Setze die Gesamtkosten auf 0. Wiederhole die Schritte 2. – 7. solange bis sie „maxIteration“-mal – in der Implementierung 50-mal – wiederholt wurden oder die „tue nichts“-Operation ausgewählt wurde.
2. Entspricht der Ist-Zustand dem Soll-Zustand, dann springe zu Punkt 8.
3. Werte das momentane Netzwerk mit dem aktuellen und dem Zielzustand als Eingabe aus.
4. Suche den Output-Knoten mit dem höchsten Wert und wähle die entsprechende Operation aus.
5. Entspricht diese Operation der „tue nichts“-Operation, dann springe zu Punkt 8.
6. Addiere den Wert 1 zu den Gesamtkosten und überprüfe ob die Vorbedingungen für die Operation aus Punkt 4 erfüllt sind. Wenn sie nicht erfüllt addiere den quadrierten Wert von „maxIteration“ zu den Gesamtkosten.
7. Wende die Operation auf den Ist-Zustand an und springe zu Punkt 2.
8. Berechne durch die Funktion „distanceToOtherState(PlantState)“, wie weit das Ergebnis vom Zielzustand entfernt ist und multipliziere den Wert „maxIteration“ mit der Rückgabe der Funktion. Addiere das Gesamtergebnis auf die Gesamtkosten.

Bei diesem Algorithmus sollen ungültig ausgeführte Operationen und zu früh beendete Erstellvorgänge durch höhere Kosten bestraft werden. Der Wert „maxIteration“ wird in Punkt 8 multipliziert um eine sofortige „tue nichts“-Operation auszuschließen, welche sonst kostengünstiger wäre als die meisten Abfolgen. Da fehlerhafte und unvollständige Abläufe so hoch bestraft werden, werden meistens erst ausführbare und fehlerfreie Abläufe generiert und erst im Anschluss die Reihenfolge der Befehle optimiert, da zusätzliche gültige Operationen nur relativ gering bestraft werden.

Um einen Steuerungscode mit einem fertig trainierten Netzwerk zu erzeugen, muss die Methode „getControlFlow(PlantState)“ der Klasse „Plant“ aufgerufen werden.

Hierbei werden ähnlich zu dem oben beschriebenen Algorithmus solange die besten Operationen ausgewählt bis der Zielzustand erreicht ist, oder die „tue nichts“-Operation ausgewählt wurde. Ungültige Operationen werden übersprungen, da sie zu Schäden an der Anlage führen könnten.

4 Validierungsbeispiel und Evaluation

In diesem Kapitel soll das Programm an einem Beispiel validiert und im Anschluss in Hinblick auf Aufwand, Geschwindigkeit und Qualität der Lösung bewertet werden.

4.1 Beispielablauf an der Demonstratoranlage am FAPS

In den Gebäuden des FAPS steht eine Anlage, die für diese Arbeit herangezogen und in Abbildung 8 dargestellt wird.

Für diese Arbeit wurde die Funktionalität dieser Anlage teilweise eingeschränkt. Der Portalroboter kann vier Positionen einnehmen. Das Band 1 wird komplett ignoriert und die Bänder 2 und 3 besitzen je vier Positionen, welche in Bezug auf die Position der Stopper gewählt wurden. Die Funktionsweise der Hubeinheiten wird auf einen Wechsel der Werkstückträgerposition reduziert.

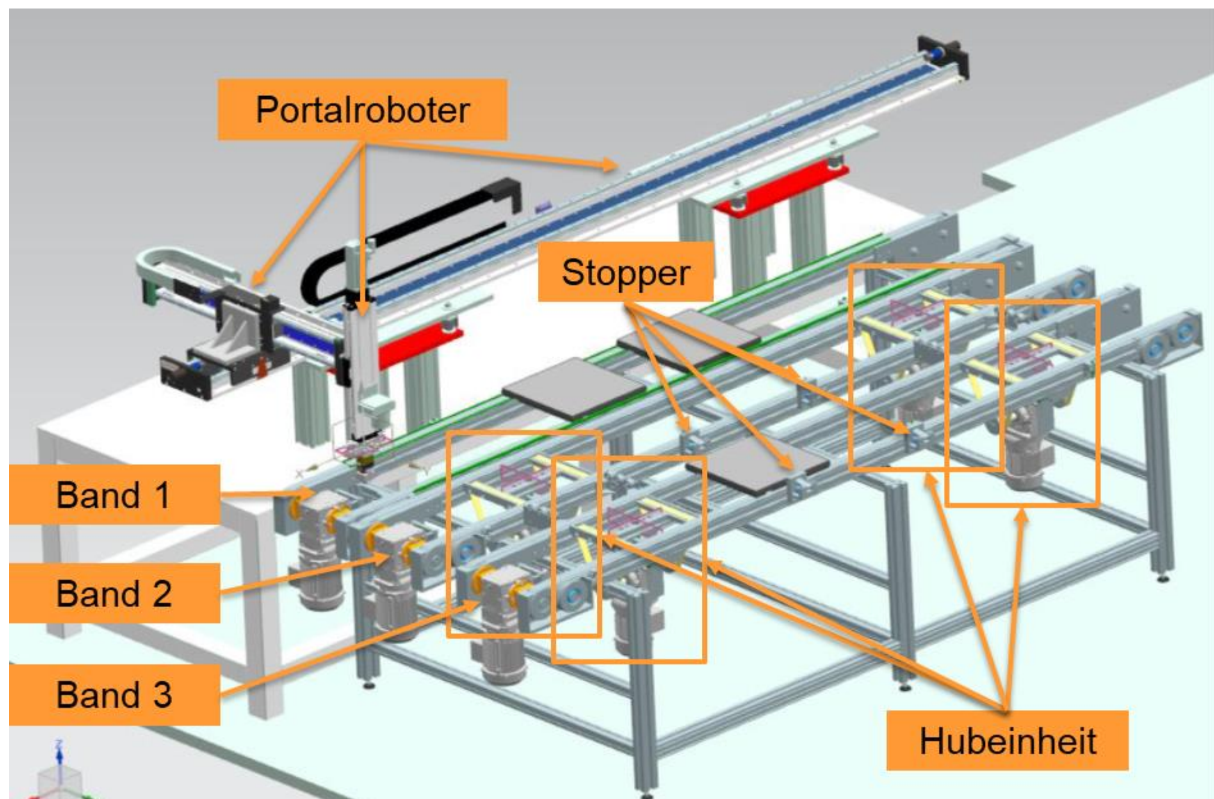


Abbildung 8: Demonstratoranlage des FAPS [5]

4.2 Anpassungen des Programms an das Beispiel

Um das Programm an die Funktionsweise der Demonstratoranlage anzupassen, wurden drei Klassen implementiert.

Die Klasse „ConveyorBelt“ stellt Funktionen zum Starten und Stoppen der Fließbänder zur Verfügung, wobei nur ein stillstehendes Band gestartet und ein sich bewegendes Band gestoppt werden kann.

Die Klasse „Grab“ implementiert die Bewegung des Greifers auf vier Positionen, die jeweils nur von der Nachbarposition erreichbar sind. Zukünftig kann die Funktionalität auf viele Positionen, einen Greiferwechsel und den Greifvorgang erweitert werden.

Die Klasse „WorkpieceCarrier“ schafft die Funktionalität eines Werkstückträgers und dessen Bewegungsfähigkeit auf dem Fließband. Ebenfalls wird durch Vorbedingungen abgefangen, dass mehrere Stückträger aufeinander laufen. Die Funktionalität von Stoppern wird in dieser Klasse umgesetzt.

Ebenfalls wurden die Variablen des evolutionären Algorithmus auf die Anlage angepasst.

4.3 Beispielhafte Auswertung eines neuronalen Netzes

Die Abbildung 9 zeigt ein beispielhaftes neuronales Netzwerk, welches nun ausgewertet werden soll. Um das Beispiel einfach zu halten werden die Eingangswerte der Input-Neuronen als 1 angenommen.

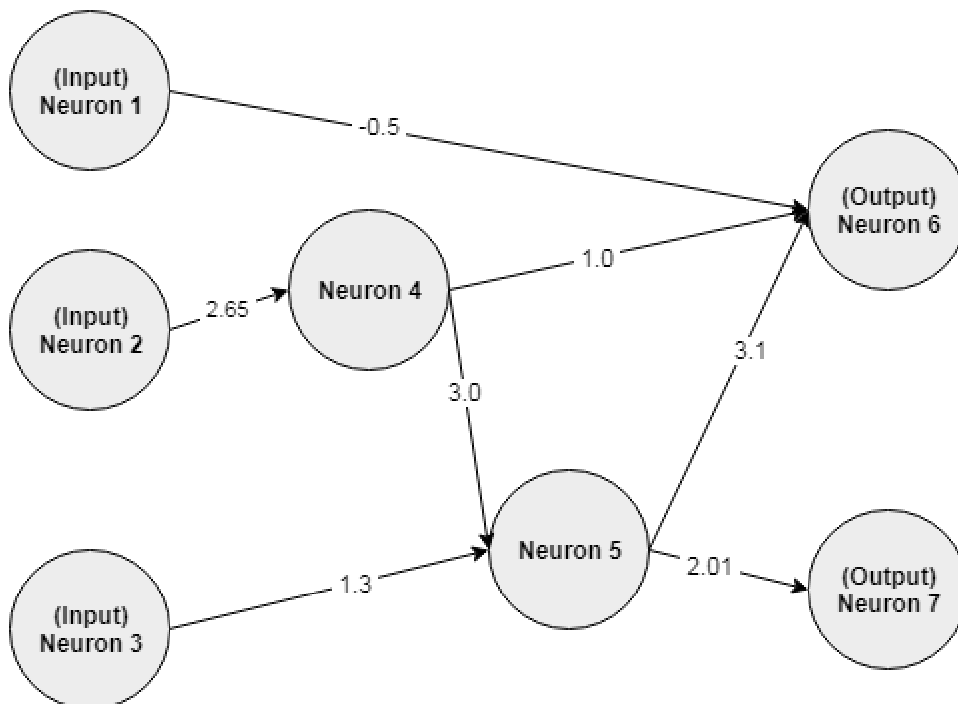


Abbildung 9: beispielhaftes neuronales Netz

Die Auswertung neuronaler Netze in dieser Arbeit wurde wie folgt implementiert:

1. Lese die Eingangsparameter in die Inputneuronen ein.
2. Wiederhole die Punkte 3 bis 6 bis alle Neuronen ausgewertet sind.
3. Iteriere über alle nicht ausgewerteten Neuronen.
4. Haben alle Vorgängerknoten beziehungsweise -neuronen ihren Wert an das Neuron weitergeben, dann wende eine Sigmoid-Funktion auf die Summe dieser Werte an.
5. Multipliziere das Ergebnis dieser Funktion jeweils mit dem Gewichtungsfaktor der Kanten beziehungsweise Gene und leite dieses Ergebnis an die Nachfolgeknoten des Neurons weiter.
6. Kehre zurück zu Punkt 2.

Wenn nun das Beispiel nach diesem Algorithmus ausgewertet wird, enthält zu Beginn jedes Input-Neuron den Wert 1. Die Neuronen 5, 6 und 7 können nicht ausgewertet werden, da noch nicht alle ihre Vorgängerknoten ausgewertet wurden, somit bleibt nur Neuron 4 zur Auswertung übrig. Der Wert in Neuron 2 wird mit dem Gewichtungsfaktor des Genes zwischen Neuron 4 und 5 multipliziert und Neuron 4 erhält somit den Wert $2{,}65$. Um nun den Wert aus Neuron 4 weitergeben zu können muss eine Sigmoid-Funktion auf diesen angewendet werden.

Die **Sigmoid-Funktion (Sigmoide Aktivitätsfunktion)** ist wie folgt definiert:

"A sigmoid function is a bounded differentiable real function that is defined for all real input values and that has a positive derivative everywhere." [6]

In diesem Beispiel wird folgende Funktion verwendet:

$$\mathrm{f(x)} = \frac{1}{e^{-0.1 \times \mathrm{x}} + 1}$$

Für $\mathrm{f}(2{,}65)$ wird somit das Ergebnis $0{,}57$ zurückgegeben, welches nun multipliziert mit $1{,}0$ an das Neuron 6 und multipliziert mit $3{,}0$ an das Neuron 5 weitergegeben wird. Neuron 5 erhält ebenfalls den Wert aus Neuron 3 über ein Gen mit Gewichtungsfaktor $1{,}3$. Somit werden die beiden Produkte $(0{,}57 * 3{,}0) = 1{,}71$ und $(1{,}0 * 1{,}3) = 1{,}3$ aufsummiert und das Ergebnis von der Sigmoid-Funktion ausgewertet. Letztendlich gibt Neuron 5 den Wert $0{,}57$ weiter. Dieser Algorithmus wird nun auch auf Neuron 6 und 7 angewendet und das Netzwerk ist anschließend fertig ausgewertet.

4.4 Ergebnisse

Beim Durchlaufen von etwa 100000 Generationen der Evolution, liefert das Programm für jeden Test-Zielzustand ein ausführbares Ergebnis ohne ungültige Operationen. Die jeweiligen Zielzustände werden allesamt erreicht. Es ist offensichtlich, dass die

optimalen Abläufe noch nicht generiert werden, da beispielsweise der Greifer von der Zielposition wegfährt, um anschließend wieder zu dieser zu fahren. Diese Zyklen erhöhen die Gesamtkosten, aber anhand der Kosten kann nicht abgelesen werden, ob der ideale Ablauf generiert wird.

Wird ein bereits trainiertes Netz zur Generierung eines Ablaufs mit neuer Startposition eingesetzt, so wählt es auch ungültige Operationen und führt bisher bei Tests zu keinen korrekten Endzuständen.

Die Berechnung einer Generation dauert in den meisten Fällen zwischen 100 und 200 Millisekunden. Nach 100.000 Generationen haben die besten Netzwerke etwa 100 Neuronen und 50 Gene. Zum Ende der Lernphase hin reduziert sich die Zahl der Spezies auf zehn, insofern das zehntbeste Netz sich kürzlich verbessert hat. Anschließend steigt die Zahl der Spezies langsam wieder, da sich die 10 besten Spezies nur sehr langsam verbessern und neue Spezies mit diesen gleichziehen.

4.5 Evaluation und Verbesserungsvorschläge

Wird das Fließband als bereits laufend angenommen und keine Möglichkeit gegeben, das Band zu stoppen, liefert das Programm wesentlich schneller gute Ergebnisse. Hierdurch wird aufgezeigt, dass der Mensch die Lernphase durch das Entfernen trivialer Operationen wesentlich beschleunigen kann.

Oft werden neue Gene zu Netzwerken hinzugefügt, welche Zyklen erzeugen. Da am Ende eines Mutationszyklus Zyklen und somit Gene entfernt werden, ist dieser Vorgang oft aufwändiger als nützlich. Ebenfalls wird bei der Zyklenentfernung das Netzwerk komplett ausgewertet, wodurch der doppelte Aufwand entsteht. Daher sollte geprüft werden, ob zyklische Netzwerke bessere oder schnellere Ergebnisse liefern. Insofern sie das nicht tun, sollte der Entfernprozess von Zyklen und der Auswertungsprozess kombiniert werden. Ebenfalls sollte bedacht werden, ob durch Mutation Kanten entfernt werden dürfen. Dies gibt dem Algorithmus mehr Freiheit und kann die Evolution möglicherweise beschleunigen. Hierdurch können Knoten entfernt werden, die im Verlauf der Evolution zufällig hinzugefügt wurden und das Netzwerk verschlechtern.

Da das Programm Netzwerke nur von einem Startzustand aus trainiert, sollten die Test-Zielzustände auch jeweils als Startzustände verwendet werden. Hierbei werden aus sechs Testfällen 30, wodurch jede Generation der Evolution mehr Rechenzeit in Anspruch nimmt und ebenfalls mehr Generationen benötigt werden, aber wesentlich vielseitigere Netzwerke entstehen können.

Multithreading ist eine Möglichkeit die Rechenzeit für jede Generation zu verkürzen. Ein Anwendungsfall könnte so aussehen, dass das Programm zeitgleich mehrfach ausgeführt wird. Dies bedeutet, dass es mehrmals 300 Netzwerke gibt, welche unabhängig der anderen Populationen trainiert werden können und anschließend

können die besten Netze der Populationen gekreuzt werden. Ein weiterer Anwendungsfall ist das zeitgleiche Auswerten mehrerer Netze durch beispielsweise einen Arbeiterpool. Hierdurch verkürzt sich die Auswertungszeit drastisch.

Die Abbildung von der Funktionalität einer Anlage auf die Programmebene sollte flexibler möglich sein. Beispielsweise sollte der Greifer beliebig viele Positionen im 3-diminsionalen Raum einnehmen können, insofern der reale Greifer dies auch kann. Dies kann neue Probleme erzeugen und die Notwendigkeit einer Änderung der gesamten Programmstruktur nach sich ziehen.

Es steckt viel Optimierungspotential in der Auswahl der überlebenden Spezies und Netzwerke. Besitzt eine Spezies das elftbeste Netzwerk, welches minimal schlechter sein kann als das Beste, kann sie trotzdem aussterben, insofern sie auch viele schlechte Netzwerke enthält. Hierbei wird ersichtlich, warum nicht nur die Spezies mit dem besten Netzwerk unbeeinflusst von ihren anderen Netzwerken überleben sollte. Die Anhebung auf die Spezies mit den zehn besten Netzwerken hat bereits die Effizienz erhöht, ist aber nicht die allgemein beste Lösung.

Im Aussortierverfahren innerhalb der Spezies steckt ebenfalls ein hohes Optimierungspotential. Hat eine Spezies ein sehr gutes und ein sehr schlechtes Netz überleben in dieser Implementierung trotzdem beide, daher sollte nach einer besseren Methode gesucht werden.

Das Netzwerk hat einen Ausgabe-Knoten, der angibt ob der Ablauf abgeschlossen ist. Dieser „tue nichts“-Knoten bedeutet einen Mehraufwand für die Evolution und bietet nur den Vorteil, dass das Netzwerk erkennen könnte, ob der Start- gleich dem Endzustand ist. Daher sollte dieser Knoten entfernt werden und anschließend die Ergebnisse verglichen werden.

5 Weitere Anwendungsgebiete und Ausblick in die Zukunft

Dieses Kapitel soll auf die vielfältigen Anwendungsgebiete und Möglichkeiten von künstlicher Intelligenz und Machine-Learning im Bereich Maschinenbau aufmerksam machen und deren Vor- und Nachteile schildern. Ebenfalls soll ein Ausblick auf die zukünftige Entwicklung dieser Themen gegeben werden.

Schon heute wird darauf geachtet den Energieverbrauch zu senken und eine Umweltverschmutzung zu vermeiden. Zukünftig kann das Prinzip, welches in dieser Arbeit verwendet wird, dazu verwendet werden, die Steuerungsabläufe hinsichtlich Energieverbrauch und Verschleißverringerung zu optimieren. Dies kann schon in naher Zukunft umgesetzt werden, da hierfür nur die Kostenfunktionen in Abhängigkeit von den Operationen gesetzt werden müssten, welche momentan alle mit dem Wert 1 bewertet werden.

Die Verwendung von neuronalen Netzen in alltäglichen Produkten, wie beispielsweise Fahrzeugen, kann den Nutzungskomfort wesentlich erhöhen. Das Netzwerk kann die Verhaltensmuster des Nutzers lernen und ihm anschließend Vorschläge machen. Der Lernvorgang ist hier leicht umsetzbar, da die Ausgabe des Netzes mit den Handlungen des Nutzers verglichen werden kann und somit Feedback bekommt.

In der Finanzwelt können neuronale Netze eingesetzt werden, um Aktienkurse zu beobachten und anschließend Vorhersagen für diese Kurse treffen. Hierbei wird ebenfalls konstant Feedback an das neuronale Netzwerk geliefert. Nach einer entsprechend langen Lernphase kann es den Kauf und Verkauf von Aktien übernehmen und somit möglicherweise Vermögen für den Nutzer generieren.

Es gibt unzählige Anwendungsgebiete für neuronale Netze und evolutionäre Algorithmen, wobei jedes Gebiet gesondert auf Forschung angewiesen ist. Ebenfalls sollte zukünftig die Kombination von neuronalen Netzen mit beliebigen anderen Verfahren und Mustern untersucht werden. Beispielsweise können diese in Verbindung mit einem Brute-Force-Algorithmus arbeiten, um in jeden Schritt dem Algorithmus das wahrscheinlichste Auswahlobjekt vorzuschlagen. Dies lässt sich auch analog auf die Tiefensuche oder ähnliche Algorithmen übertragen.

Da das Ergebnis dieser Arbeit ein funktionierender Prototyp und kein optimiertes Produkt ist, kann die Dauer einzelner Evolutionsschritte sicherlich um ein Vielfaches verringert werden, wodurch wesentlich komplexere Aufgaben lösbar werden. Hinzu kommt, dass die Hardware zukünftig deutlich schneller werden wird und mehr Berechnungen in gleicher Zeit ermöglichen wird.

In ferner Zukunft könnte eine komplette Nachbildung eines menschlichen Gehirns und dessen Entwicklung mithilfe von neuronalen Netzen erschaffen werden, wobei dies mit Vorsicht zu betrachten ist. Schritt für Schritt wird sich die Technik in eine solche Richtung bewegen, aber es besteht die Möglichkeit, dass neuronale Netze von

neueren Technologien abgelöst werden. Aus diesem Grund sollte sich die Forschung nicht nur auf neuronale Netze fixieren, sondern auch verwandte Methoden in Betracht ziehen. Diese Arbeit hat gezeigt, dass die Arbeit, welche heute noch von Menschen erbracht wird, zukünftig von Computern übernommen werden kann, daher wird es vermutlich zu einer Verschiebung dieser Arbeitsplätze in die Forschung und Entwicklung kommen.

Literaturverzeichnis

Literaturverzeichnis

[1] SUSANNE NÖRDINGER. Darum bringen KI und Machine Learning den Maschinenbau voran [Zugriff am: 24. September 2018]

[2] MARTIN KNÖFEL. *Künstliche Neuronale Netze* [online], 2017 [Zugriff am: 23. September 2018]

[3] AMY HAN, J.H. *Evolving Mario to Maximize Coin Score Using Neat and Novelty* [online] [Zugriff am: 23. September 2018]

[4] KENNETH O. STANLEY & RISTO MIIKKULAINEN. Evolving Neural Networks through Augmenting Topologies. *The MIT Press Journals*, **2002**

[5] SEBASTIAN FEIKE. *Semantische Modellierung von Steuerungstechnischen Vorgängen im Kontext von Plug&Produce.* Projektarbeit. Erlangen/Nürnberg, 14. Juli 2017

[6] HAN, J. und C. MORAGA. *The influence of the sigmoid function parameters on the speed of backpropagation learning* [online] [Zugriff am: 24. September 2018]

Anhang A

Datenträger mit

Digitaler Version der Arbeit (als) Word und PDF
Eclipse Java Projekt und Frontend Dateien
Citavi Projekt
Quelldateien

Lebenslauf

Geburtsdatum: 11.06.1996

Ausbildung:

10/2014 – 09/2018: Bachelor-Studium an der Friedrich-Alexander-Universität

Praxiserfahrung:

06/2018 – 09/2018: Softwareengineer bei der e.solutions GmbH

02/2018 – 05/2018: Werkstudent bei der e.solutions GmbH

08/2014 – 09/2014 Werkstudent bei der Siemens AG
Und 04/2015 – 07/2015: