Gymnasium Olching
Georgenstr. 2
82140 Olching

Abiturjahrgang 2016

Seminararbeit

Rahmenthema des Wissenschaftspropädeutischen Seminars:

Maß und Zahl

Leitfach: *Mathematik*

Thema der Arbeit:

Genetische Programmierung und automatisierte Trainierung von neuronalen Netzwerken

von

Dominik Horn

Betreuende Lehrkraft: OStR Susanne Barth

Abgabetermin *(2. Unterrichtstag im November)*: 10. November 2015

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
Schriftliche Arbeit				x3	
Abschlusspräsentation				x1	
				Summe:	
				Gesamtleistung nach §61 (7) GSO = Summe : 2 (gerundet)	

_____

(Datum und Unterschrift der Kursleiterin)

Inhaltsverzeichnis

1. Einführung und Grundidee

1.1 Einleitende Bezugsherstellung

Selbst denkende Computer sind spannender denn je. Zahlreiche moderne Technologien, wie autonome Kraftfahrzeuge und Sprachassistenten verwenden Techniken um künstlich intelligent zu sein (Vgl. GolemDeepLearning 2015). Wie funktionieren diese? Wie viel Arbeitsaufwand bedarf die Implementierung einer künstlichen Intelligenz?

Ein vielversprechender Ansatz, bei der Suche nach Möglichkeiten künstliche Intelligenz zu implementieren, stammt aus dem Bereich des „Machine learning“. Kurzum wird versucht, das relativ simple Neuronen-Synapsen-Modell eines biologisches Gehirns in einem Computerprogramm zu implementieren. Dieses künstliche „Gehirn“ ist hierbei aus mehreren Neuronen aufgebaut. Daraus leitet sich auch der Name ab: Neuronales Netzwerk. Im Detail wird die Idee dahinter weiter unten im Text erklärt. Vorteile eines neuronalen Netzwerkes sind der niedrige Schwierigkeitsgrad beim Implementieren und die universelle Anwendbarkeit (Vgl. GolemDeepLearning 2015). Nachteilig ist der sehr hohe Aufwand beim Trainieren solcher Netzwerke. Es werden teils große Datenmengen benötigt, um zum gewünschten Ergebnis zu kommen. Ein Beispiel aus der Medizin: Ein neuronales Netzwerk soll eigenständig Patientendaten analysieren und ihre Anfälligkeit für verschiedene Krankheiten automatisch bewerten. Es muss also beispielsweise von selber den Schluss ziehen, dass Menschen, welche permanent unter hohem Stress leiden ein erhöhtes Risiko für Herzprobleme haben (Vgl. Stress 2015). Damit ein neuronales Netzwerk lernt, solche Zuordnungen eigenständig vorzunehmen, muss es mit ausreichend Patientendaten und dazu passenden Krankheitsdaten gefüttert werden. Dies muss manuell geschehen und ist oft zu aufwendig, beziehungsweise aufgrund von Datenmangel nicht realisierbar.

So genannte „Genetic Algorithms“, fortan GA genannt, könnten hier Abhilfe schaffen. Ein GA ist kurz gesagt eine heuristische Suchmethode, welche den Prozess der natürlichen Selektion versucht zu emulieren. Zusammengefasst wird eine Population an Subjekten beliebiger Art in einer simulierten Umgebung getestet, bewertet und gemäß dem Satz „Survival of the fittest“ - Überleben des am besten angepasstesten – selektiert und weiter entwickelt. Die Subjekte des Algorithmus sind dabei beliebige Objekte, welche aus „Chromosomes“ bestehen (Vgl. AI-Junkie-GA 2015). GAs werden oft bei

Optimierungs- oder Suchproblemen eingesetzt. Kombiniert man nun neuronale Netzwerke und GAs, eliminiert man theoretisch das aufwändige, händische Trainieren der künstlichen Intelligenz, da diese Aufgabe fortan der Computer selbstständig übernehmen kann.

Dieser Arbeit zugrunde liegt allerdings ursprünglich noch eine weitere Idee. GAs sind eine allgemeine Fassung eines Konzepts – nämlich virtuelle Evolution zur Problemlösung anzuwenden. Eine konkretere Form der GAs ist das so gennante „Genetic programming“, fortan GP. Subjekte einer Population bei GPs sind konkret Computerprogramme. Ein korrekt implementierter GP-Algorithmus kann theoretisch Computerprogramme von selber entwickeln (Vgl. TU-Chemnitz 2015). Dies geschieht wie oben genannt über den Ansatz der Evolution. Nachfolgend werden die Grundlagen - GAs, das GP und neuronale Netzwerke - genauer erklärt.

1.2 Erklärung der Grundkonzepte

1.2.1 Genetische Algorithmen

Ein genetischer Algorithmus sieht folgendermaßen aus: Zu Beginn der Laufzeit des Algorithmus wird eine Startgeneration von variabler Größe, bestehend aus ähnlichen Subjekten erzeugt. Diese wiederum bestehen aus so genannten „Chromosomes“, welche ihre Eigenschaften und ihr Verhalten bestimmen (Vgl. AI-Junkie-GA 2015). Jedes einzelne dieser Subjekte durchläuft eine Testphase, in welcher überprüft wird wie gut es seine Aufgabe erfüllt. Die Bewertung erfolgt dabei über eine so genannte „fittest function“ (Vgl. AI-Junkie-GA 2015), welche je nach Zweck des zu erzeugenden Subjekts anders aussehen muss. Basierend auf dieser Bewertung wird das/die beste(n) Subjekt(e) ausgewählt und mit ihm/ihnen durch zufällige Mutation und/oder Kreuzung eine neue Population erzeugt. Dieser durchläuft erneut die Testphase, wird bewertet, selektiert und mutiert. Das ganze erfolgt so lange bis ein Subjekt gefunden wird, welches die Kriterien der fittest-function ausreichend erfüllt (Vgl. AI-Junkie-GA 2015).

1.2.2 Genetisches Programmieren

Genetische Programmierung ist eine spezielle Form der Genetischen Algorithmen. Statt allgemeiner Subjekte werden konkret Computerprogramme erzeugt, getestet, nach einer

fittest function bewertet, selektiert und mutiert (Vgl. TU-Chemnitz 2015).

Die fittest-function aus der Testphase würde in einem simplen Fall, bei dem ein Programm erzeugt werden soll das ein Wort auf die Konsole schreiben kann, zum einen bewerten wie fehlerbehaftet das Programm ist und wie nah seine Ausgabe an die gewünschte Ausgabe kommt. Für jede Abweichung vom gewollten Ergebnis vergibt der Algorithmus Punkte. Diejenige(n) Routine(n) mit den niedrigsten Punktzahlen werden behalten, sie kommen dem Ziel am nächsten. Alle anderen werden gelöscht. Anschließend wird durch ein beliebiges Verfahren eine neue Population aus dem/den Besten erzeugt, welche wieder die Tests durchlaufen muss. Der beschriebene GP-Algorithmus ist schemenhaft in Abbildung 1 zu sehen und wird so lange wiederholt, bis ein Programm in der Testphase den Wert 0, also ein perfektes Ergebnis nach den Kriterien der fittest-function, zugewiesen bekommt.

Ein mögliches Verfahren, um eine neue Population nach der Selektionsphase zu erzeugen, ist den Quelltext des besten Programms n-mal zu kopieren und jeweils an unterschiedlichen Stellen zufällig zu mutieren. Besser ist jedoch die Quelltexte der besten Programme zu kreuzen und erst dann anschließend zu mutieren. Statt einer einzigen Population könnten auch mehrere verschieden Populationen, also verschiedene Spezies, gleichzeitig diesen Algorithmus getrennt durchwandern. Dabei werden diese Spezies gelegentlich miteinander gekreuzt. Letzteres wirkt einigen, später genauer diskutierten Problemen mit der genetischen Programmierung präventiv entgegen.

1.2.3 Künstliche neuronale Netzwerke

Künstliche neuronale Netzwerke sind vom Computer simulierte „Gehirne“. Sie bestehen aus einzelnen Neuronen, welche jeweils über gewichtete Verbindungen mit anderen Neuronen verbunden sind (Vgl. AI-Junkie-NeuralNetworks 2015). Ein oder mehrere Neuron(en) dienen als „Input neuron(s)“ oder Eingabeneuronen – über diese werden Daten in das neuronale Netzwerk eingespeist. Parallel zu den Eingabeneuronen gibt es so gennante „Output neuron(s)“ - ein oder mehrere Ausgabeneuronen. Über diese werden vom neuronalen Netzwerk verarbeitete Daten entnommen. Zwischen der Aus- und Eingabeschicht befinden sich die „hidden layers“ - die versteckten Schichten (Vgl. AI-Junkie-NeuralNetworks 2015). Eingespeiste Daten wandern von den Eingabeneuronen über ihre Verbindungen zu den Neuronen der versteckten Schichten, welche sich die Daten von Schicht zu Schicht jeweils auch über ihre Verbindungen durchreichen.

Schlussendlich gelangen die Daten so an die Ausgabeneuronen. Das ganze ist in Abbildung 2 veranschaulicht. Die Daten werden währenddessen von jedem Neuron mittels simpler Mathematischer Berechnungen verarbeitet.

Jedes Neuron einer Schicht ist mit jedem Ausgang jedes Neurons der vorherigen Schicht über eine multiplikatives Gewicht verbunden verbunden. Im einfachsten Fall schickt ein Neuron den Wert „1“ an alle Neuronen welche mit seinem Ausgang verbunden sind, wenn alle Werte, welche an seinem Eingang angekommen sind aufsummiert und multipliziert mit ihren individuellen Gewichtungen einen vorher fest gesetzten Grenzwert überschreiten (Vgl. Abbildung 3). Diese neuronalen Netzwerke werden auch „binäre neuronale Netzwerke“ genannt (Vgl. AI-Junkie-NeuralNetworks 2015). Eine weitere Möglichkeit ist eine „Sigmoid“-Funktion zu verwenden, um Signalausgaben dynamisch zu gestalten (Vgl. AI-Junkie-NeuralNetworks 2015). In Abbildung 4 wird eine „Sigmoid“ Funktion gezeigt. Mathematisch sieht diese so aus:

$$\frac{1}{1+e^{-(\frac{a}{p})}}$$

a steuert die Aktivierung des Neurons und p steuert die Kurve: Umso kleiner umso mehr schaut sie aus wie die Stufenfunktion aus Abbildung 3.

1.3 Fragestellung

Die ursprüngliche Frage, welche sich im Bezug zu diesem Thema stellte war, inwiefern ein Computerprogramm selber Computerprogramme schreiben kann. Bei einer Recherche im Internet auf der Webseite „youtube.com“ fand ich ein Video, in dem ein Programm gezeigt wurde, welches eigenständig simplen Quelltext entwickeln konnte. Durch Benutzen der Google-Suchmaschine und eigenes Ausprobieren entstand dann der erste Prototyp, welcher einfache Computerprogramme in der Sprache „Brainfuck“ autonom mittels eines genetischen Algorithmus, speziell das genetische Programmieren, erstellen kann. Anschließend an diesen Prototypen sollte erörtert werden, inwiefern neuronale Netzwerke mit genetischer Programmierung, beziehungsweise genetischen Algorithmen kombiniert werden können, um eventuell eine sich selbst verbessernde künstliche Intelligenz zu schaffen. Dass es sich hierbei um eine wahre, selbstbewusste und wirklich denkende Intelligenz handeln würde war damals bereits außer Frage. Hierfür fehlt Rechenleistung und Zeit. Trotzdem entstand ein zweiter Prototyp, welcher diesen Ansatz

der automatischen Trainierung/Programmierung von neuronalen Netzwerken durch genetische Algorithmen vielversprechend umsetzt. Darauf aufbauend stellte sich die Frage wie weit dieser Ansatz taugt, und das sollte mithilfe eines dritten Prototypen herausgefunden werden.

2. Praktische Arbeit

2.1 Der erste Prototyp

2.1.1 Grundidee

Der erste Prototyp sollte mithilfe genetischer Programmierung automatisiert kleine Programme in der Computersprache „Brainfuck“ entwickeln, welche ein kurzes, vorher vom Benutzer festgelegtes Wort auf dem Bildschirm anzeigen können.

2.1.2 Theorie: Brainfuck

Brainfuck ist eine von Urban Müller entwickelte, Turing-vollständige Programmiersprache, welche aus 8 Befehlszeichen besteht (Vgl. muppetlabs 2015). Diese 8 Zeichen, und ihre Funktion, können in Abbildung 5 gesehen werden. Ein Brainfuck-Programm hat Zugriff auf eine vorher definierte Anzahl an Speicher-Bytes, welche es mit beliebigen Informationen füllen kann.

2.1.3 Umsetzung des ersten Prototypen

Der erste Prototyp ist in der Computersprache „C“ geschrieben.
Zu Beginn der Laufzeit des ersten Prototypen wird eine Population an 1000 zufälligen Quelltexten mit der Länge von 100 Befehlszeichen erzeugt. Jeder dieser Quelltexte wird von einem selbst geschriebenen, so genannten „Interpreter“ anschließend ausgeführt. Das bedeutet, dass das Programm nicht von einem „Compiler“ in Machinensprache übersetzt

wird und dann direkt auf dem Prozessor des Computers läuft, sondern die Sprache von einem Programm, dem Interpreter, direkt interpretiert wird.

Das Ergebnis nach dem Lauf des Programms wird jeweils bewertet. Hierzu schaut sich der Algorithmus an, was der interpretierte Quelltext auf dem Bildschirm ausgeben würde. Um zu bestimmen wie nah ein gegebenes Programm am Ziel ist ein vorher definiertes Wort auf dem Bildschirm anzuzeigen wird schlicht die lexikalische Abweichung der Ausgabe des Programms von dem Zielwort berechnet. Das heißt, dass jedem Buchstaben eine Zahl von 0 bis 25, welche seine Position im Alphabet repräsentiert, zugewiesen wird. Die Buchstabennummer jedes Buchstaben, welcher vom Programm ausgegeben wurde wird subtrahiert von der Buchstabennummer jedes Buchstaben des Zielwortes. Die Beträge der erhaltenen Werte werden aufsummiert. Für jeden Buchstaben der im Vergleich zum Zielwort fehlt oder zu viel ist wird auf den so erhaltenen Wert noch einmal ein variabler Zahlenwert aufsummiert. Schlussendlich erhält man so einen Abweichungswert für jedes Quelltextfragment. Ist dieser null gibt das Programm das gewünschte Wort perfekt aus. Aus dieser Logik wird die fittest-function gefolgert: Dasjenige Programm mit dem niedrigsten Abweichungswert ist das momentan beste.

Von diesem werden im nächsten Schritt 1 000 Kopien produziert, welche an zwei je unterschiedlichen Stellen im Quelltext zufällig mutiert werden. Der Algorithmus läuft so lange bis ein Programm mit einem Abweichungswert von „0“ bewertet wird. Dieses ist das gewünschte Programm.

2.1.4 Testergebnisse und Fazit

Abbildung 6 zeigt einen Durchlauf des ersten Prototypen. Es ist zu erkennen, dass der Benutzer das Wort „hallo“ als Zielwort festgelegt hat. Der Prototyp generiert wie oben beschrieben zunächst eine Startpopulation aus 1000 zufälligen Brainfuck-Programmen. Das beste Programm aus dieser sieht so aus: „-++.-++++.+.++.-+.+++-+-+++++---+---+-----+-++--+++-++++.+-+++..++--+..--+--+--.-..++-++++--+.+++.+++.+“ und gibt die Zeichenkette „befhhloopplkkor“ aus. Damit hat es einen relativ hohen Abweichungswert.

Nach zehn Generationen hat das beste Programm eine Bewertung von nur drei Punkten erreicht, das heißt die lexikalische Abweichung beträgt drei Zeichen. Dieses Programm gibt das Wort „gblln“ aus. Die nächsten 130 Generationen verändert sich der beste Quelltext nicht, er zeigt „hbllo“ auf dem Bildschirm an und weicht ergo um ein Zeichen

vom Wort „hallo“ ab. Schlussendlich findet der Algorithmus eine Lösung: Das Programm „-++++-+--++--+-++++++.+-++-+----+---++------+.+-++++++++++-++..+++--++.+--+-+-+-+-+-+-----+----+--+++“ würde das gewünschte Wort „hallo“ auf dem Computerbildschirm anzeigen.

Beim Betrachten dieses Durchlaufs zeichnet sich bereits ein Problem des Algorithmus ab. Über 130 Generationen „hängt“ der Algorithmus an einer fast perfekten Lösung, welche durch zufällige Mutationen nur schwer zu verbessern ist. Änderungen verschlechtern nämlich im Normalfall aufgrund von der Wahrscheinlichkeitsverteilung die Kopien des pro Durchlauf besten Programms, darum dauert das Finden einer Lösung lange. Im schlimmsten Fall kann der Algorithmus sogar, aufgrund von der Beschaffenheit von Brainfuck und der Fittest-Function, komplett stecken bleiben. Findet der Algorithmus beispielsweise ein Programm, dass eine niedrige Bewertung erhält aber in keinem Fall durch zwei zufällige Änderungen verbessert werden kann wird es nie mehr vorwärts gehen und als einziger Ausweg bleibt ein Neustart des Prototypen.

Ein weiteres Problem ist, dass die Laufzeit des Algorithmus wegen seiner zufälligen Beschaffenheit nicht vorhersehbar ist. In seltenen Fällen ist das gewünschte Programm bereits in der zufälligen Startpopulation vorhanden und die Laufzeit ist praktisch gleich 0. In anderen Fällen kann die Laufzeit auch unendlich lang werden, wenn wie oben besprochen der Algorithmus hängt.

Dennoch lässt sich sagen, dass der erste Prototyp seine Aufgabe erfüllt hat. Er zeigt auf, dass mithilfe von genetischer Programmierung die automatisierte Entwicklung von Computerprogrammen zumindest im kleinen Maßstabe definitiv möglich ist.

2.2 Der zweite Prototyp

2.2.1 Grundidee

Der zweite Prototyp sollte mit einem angepassten Algorithmus automatisiert ein neuronales Netzwerk entwickeln, welches einen virtuellen Käfer so über den Computerbildschirm steuert, dass dieser möglichst viele Nahrungsstücke in einer simulierten Welt einsammelt.

2.2.2 Theorie

Dem neuronalen Netzwerk werden zwei Zahlenwerte, je zwischen null und eins als Eingangswerte eingespeist. Einer repräsentiert die aktuelle Bewegungsrichtung als Winkel und der andere den Winkel zum nächsten Nahrungsstück. „0“ bedeutet hierbei beispielsweise rechts und „0.5“ links. Die Formel dazu lautet:

$$\frac{\textit{Winkel zwischen Richtung und Vektor nach Rechts}(1;0)}{360°}$$

Ausgeben soll das neuronale Netzwerk genau einen Zahlenwert, welcher die Bewegung des Käfers steuert. Durch sein Vorzeichen gibt dieser eine Drehrichtung und über die Größe seines Betrags eine Drehgeschwindigkeit an.

Die Anzahl der versteckten Schichten von Neuronen ist vorher nicht festgelegt und soll beim Programmstart durch Parametereingaben bestimmt werden. Dadurch kann die optimale Größe für den Versuchsaufbau experimentell angenähert werden.

2.2.3 Umsetzung des zweiten Prototypen

Der zweite Prototyp ist in der Programmiersprache „Java“ umgesetzt und verwendet die Slick2D Grafikbibliothek. In seinem Konzept lehnt es sich an das Programm von der Website AI-Junkie-NeuralNetworks (Vgl. AI-Junkie-NeuralNetworks 2015) an.

Zunächst musste eine simulierte Testumgebung geschaffen werden. Diese besteht aus allen Käfern der aktuellen Population, sowie einer vorher festgelegten Anzahl an Nahrungsstücken, welche an zufälligen Positionen nahe der Mitte der Simulierten Fläche angesiedelt sind. Abbildung 7 zeigt den Prototypen zur Laufzeit. Dieser ist von der Softwarearchitektur her aufgebaut wie ein Videospiel. Das heißt, dass sechzig mal pro Sekunde die Simulation einen kleinen Schritt vorwärts schreitet und so eine flüssige, video- artige Darstellung zustande kommt. Bei jedem dieser Schritte werden die Eingabeparameter – Bewegungsrichtung und Richtung zum nächsten Nahrungsstück – für jeden Käfer neu ermittelt und dann die Ausgabe seines rekursiv aufgebauten neuronalen Netzwerkes berechnet. Diese Ausgabe wird anschließend verwendet, um den Käfer entsprechend zu drehen. Er bewegt sich dann um einen kleinen Schritt festgelegter Größe, genau:

$$\frac{3\,Pixel}{60\,Schritte}$$

in die neue Richtung. Das bedeutet, dass er sich nach sechzig Schritten, also einer ganzen Sekunde um genau drei Pixel bewegt hat.

Eine Testphase dauert 58 823 Schritte:

$$\frac{1\,000\,000\,Schritte}{\left(\frac{1\,000\,Millisekunden}{60\,Simulationsschritte\ pro\ Sekunde}\right)} = \frac{1\,000\,000\,Schritte}{17\,ms\ pro\ Simulationsschritt} = 58\,823$$

Das heißt nach ~16 Minuten ist eine Generation, beziehungsweise ein Durchlauf des Algorithmus beendet. Um mit dem bloßen Auge und im Rahmen der menschlichen Aufmerksamkeitsspanne einen Fortschritt sehen zu können wurden zwei Methoden zur Beschleunigung implementiert. Einmal lässt sich durch Drücken der F3 Taste die Anzahl der Simulationsschritte pro Sekunde signifikant erhöhen und zum anderen können 50 000 Schritte der Simulation mit der F1 Taste direkt übersprungen werden.

Das aufsammeln von Nahrungsstücken geschieht, wenn ein Käfer über das Nahrungsstück läuft. Sammelt ein Käfer ein Nahrungsstück auf, erscheint an zufälliger Stelle in der simulierten Umgebung ein neues.

2.2.4 Testergebnisse und Fazit

In den meisten Testläufen ist nach ein paar Generationen eindeutig zu sehen, dass die einzelnen Käfer sich im Vergleich zum anfänglichen Zustand sehr viel Zielstrebiger auf die Nahrungsstücke zu bewegen. Dies ist dem sehr einfach Versuchsaufbau geschuldet, da der Schritt von dem anfänglichen, zufälligen neuronalen Netzwerk hin zu dem gewünschten Ergebnis sehr klein und daher das schnelle Finden einer Lösung sehr wahrscheinlich ist. Basierend auf diesen Erkenntnissen lässt sich nicht prognostizieren, wie gut der Algorithmus bei komplizierteren Aufgaben funktioniert.

2.3 Der dritte Prototyp

2.3.1 Grundidee

Da der Algorithmus im zweiten Prototypen vielversprechend funktioniert hat, wird der Ansatz im dritten Prototypen einfach hoch skaliert. Aufgabe des automatisiert zu findenden neuronalen Netzwerk soll es sein, selbständig eine Mario-artige Spielfigur in einem selbst geschriebenen 2D-Computerspiel zu steuern. Das Spiel ist dabei folgendermaßen aufgebaut: Die Spielfigur muss durch eine zweidimensionale Welt, vorbei an statischen und dynamischen Hindernissen, in ein Ziel gesteuert werden. Dazu kann sie springen und sich nach links sowie nach rechts bewegen.

2.3.2 Umsetzung des dritten Prototypen

Der dritte Prototyp ist in der Programmiersprache Java geschrieben und bedient sich des Slick2D Grafik-„Frameworks“.

Ein Computerspiel besitzt immer eine so genannte „Gameloop“ - eine Spielschleife. Diese sorgt dafür, dass das Computerspiel periodisch in einem gewissen Intervall Spieler-Eingaben entgegen nimmt, den anzuzeigenden Zustand unter Berücksichtigung der Spielphysik und Spiellogik neu berechnet und anschließend das neue Bild des Spielwelt auf den Computerbildschirm ausgibt. Idealerweise geschieht dies sechzig mal pro Sekunde, also in einem Intervall von siebzehn Millisekunden.

Die Physik wird über eine so genannte „Physics Engine“ realisiert (Vgl. Gamedev-tutsplus 2015). Diese kümmert sich darum, dass die einzelnen Objekte eines Spiels den physikalischen Regeln, wie Gravitation, folgen. Auch Kollisionen einzelner Objekte werden von ihr aufgelöst. Im Fall des dritten Prototypen kann die selbst entwickelte Physics-Engine sogar Auskunft darüber geben, ob zwischen zwei Objekten ein Hindernis beziehungsweise eine „line of sight“ ist. Das heißt mit ihr kann ermittelt werden ob beispielsweise eine Gegner die Spielfigur sehen kann. Implementiert ist letzteres mithilfe von „Raycasting“ (Vgl. Redblobgames 2015). Verwendet wird dieses Verfahren für die Errechnung der Eingabedaten für das neuronale Netzwerk. Dieses bekommt Daten über alle Dinge, welche es in der 2D-Spielewelt realistisch sehen kann.

Die grafische Anzeige erfolgt wie eingangs erwähnt mittels des Slick2D-Frameworks. Diese Bibliothek hält unter anderem vorgefertigte Methoden bereit, um zweidimensionale

Grafiken zu laden und mit dem OpenGL-Standard über die Grafikkarte anzuzeigen.

Die Spielfigur wird momentan leider noch händisch und nicht von einem neuronalen Netzwerk gesteuert. Dies ist zum einen einer Verkalkulierung und dem Zeit- und Datenverlust durch defekte Computer geschuldet. Der Arbeitsumfang des dritten Prototyps ist dennoch groß und aus diesem Grund inkludiert.. Ursprünglich war für diesen Abschnitt ein dreidimensionales Computerspiel geplant, welches von einem neuronalen Netzwerk gesteuert werden sollte. Die Engine für das 3D-Spiel, welche sich um Anzeige, Physik, Logik et cetera gekümmert hätte, sollte selbst geschrieben sein. Der enorme Aufwand eines solchen Unterfangens war ab einem gewissen Punkt nicht mehr stemmbar, ich entschloss mich also lieber zu versuchen, noch etwas vergleichbares in 2D zu entwickeln. Testphase und Auswertung müssen für den dritten Prototypen zum aktuellen Zeitpunkt ausfallen.

3. Gesamtfazit und Problemdiskussion

3.1 Probleme der genetischer Algorithmen

Beim Testen des ersten Prototypen ist bereits ein gravierender Fehler der genetischen Algorithmen aufgefallen. Alle genetischen Algorithmen sind von der Laufzeit her unberechenbar, da sie sehr viel auf Zufall und damit Wahrscheinlichkeiten basieren. Selbst bei dem sehr einfachen Anwendungsfall des ersten Prototypen treten bereits große Diskrepanzen auf. Einmal wird das gesuchte Programm direkt in der ersten Generation gefunden, ein anderes mal dauert es ein paar Sekunden bis Minuten. Die Folgerung: Skaliert man den Algorithmus hoch, das heißt verwendet man ihn für schwierigere Aufgaben kann es sehr lange dauern, bis er diese Lösen kann.

Auch während der Algorithmus rechnet können bereits Probleme auftreten, zum Beispiel dass der Algorithmus stecken bleibt. Dieser Fall tritt dann auf, wenn in zu geringem Maße mutiert wird. Ist die Wahrscheinlichkeit, ein besseres Subjekt durch den Mutationsprozess zu finden, gleich null kann logischerweise kein Fortschritt mehr erfolgen. Zu Deutsch: Der Sprung vom jetzigen besten Zustand zum nächst Besseren kann nicht in einer Mutationsphase geschehen. Der Algorithmus dreht sich in diesem Fall im Kreis. Ein Beispiel aus der Evolution zur Illustration: Es ist davon auszugehen, dass immer nur genau eine Eigenschaft eines Lebewesens verändert werden kann pro Mutationsphase, das heißt pro Reproduktion. Gestartet wird mit einem Fisch ohne Beine und Ziel ist es ein Landlebewesen mit Beinen zu schaffen. Das Dilemma ist offensichtlich: Entweder der Fisch hat Beine oder kann an Land atmen. Einzeln ist jede dieser Eigenschaften für den Fisch hinderlich. Die fittest-function der biologischen Welt („Survival of the fittest“) bewertet den Fisch mit Beinen, der aber aufgrund seiner Lunge nicht an Land atmen kann genauso schlecht wie den Fisch der wegen seiner Lunge an Land leben muss, sich aber aufgrund der mangelnden Beine nicht fortbewegen kann. Der Ansatz Beine beziehungsweise eine Lunge zu haben wird also durch die natürliche Selektion verworfen und der evolutionäre Sprung vom Fisch zum Landlebewesen mit Beinen kann nicht erfolgen. So ähnlich kann es sich auch mit Subjekten von genetischen Algorithmen verhalten.

3.2 Probleme neuronaler Netzwerke

Neuronale Netzwerke müssen im Normalfall von Hand mit einer großen Menge Daten trainiert werden das zu tun, was sie sollen. Mithilfe eines genetischen Algorithmus kann dies in gewissen Fällen eliminiert werden. Verwendet man dieses Verfahren tun sich allerdings neue Probleme auf, welche im vorigen Abschnitt erläutert wurden.

Neuronale Netzwerke begeistern mit ihren vielen Möglichkeiten und grenzen bereits heute an scheinbar echte künstliche Intelligenz (Vgl. GolemDeepLearning 2015). Ihre Unberechenbarkeit, gepaart mit ihren Kompetenzen und der Möglichkeit der Selbstverbesserung wirft allerdings brennende Fragen auf. Droht eine Roboterapokalypse? Wie werden weitaus intelligentere Maschinen uns gegenüber handeln?

3.3 Abschließendes Fazit

Die beiden ersten Prototypen beweisen wie verblüffend einfach die Umsetzung einer scheinbar semi-intelligenten Maschine ist. Auch wenn die Ansätze rudimentär erscheinen und im kleinen Maßstab arbeiten ist durchaus denkbar, dass zukünftige Fortschritte auf den erforschten Gebieten und die steigende Rechenleistung moderner Computersysteme in nicht allzu ferner Zeit echte künstliche Intelligenz hervorbringen. Diese Aussicht ist gleichermaßen interessant und beängstigend, wenn man an aktuelle Science-Fiction Filme denkt. Renommierte Wissenschaftler, wie Stephen Hawking, warnen sogar aktiv vor künstlichen Intelligenzen und dem damit verbunden, drohenden Ende der menschlichen Zivilisation (Vgl. HawkingBBC 2015).

4. Zukunftsausblick

Meine Arbeit ist aufgrund des beschränkten Maßstabs trotzdem ethisch vertretbar, da sie keine echte künstliche Intelligenz mit unbegrenzten Kompetenzen in dem mir möglichen Rahmen entstehen lassen kann. Aus diesem Grund werde ich den dritten Prototypen fertig stellen und testen. Eventuell folgt dann tatsächliche, anwendungsbezogene Arbeit. Das heißt, ich werde erforschen inwiefern genetische Algorithmen kombiniert mit neuronalen Netzwerken Probleme aus der echten Welt lösen können.

5. Abbildungen und Quelltext

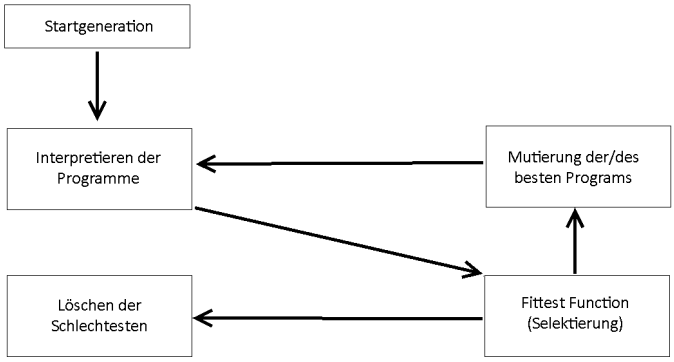

Abb. 1: genetische Programmierung schemenhaft dargestellt

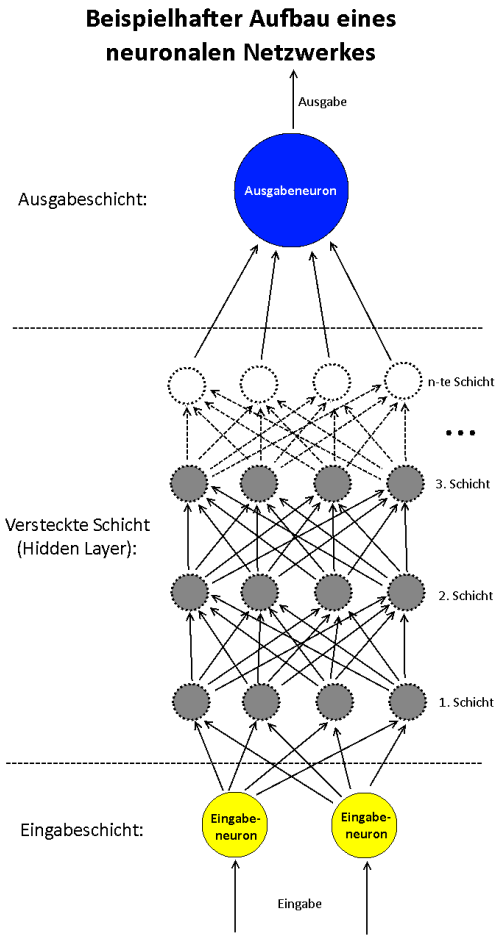

Abb. 2: Beispielhafter Aufbau eines neuronalen Netzwerkes (Vgl. AI-Junkie-NeuralNetworks 2015)

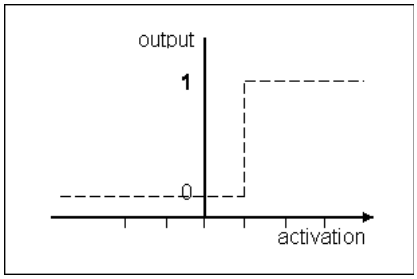

Abb. 3: Ausgabe eines virtuellen Neurons bei festgesetztem Schwellwert (Vgl. AI-Junkie-NeuralNetworks 2015)

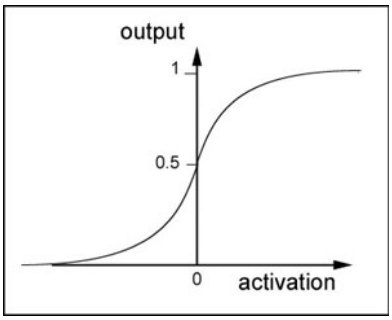

Abb. 4: Ausgabe eines virtuellen Neurons durch Sigmoid-Funktion (Vgl. AI-Junkie-NeuralNetworks 2015)

Befehlszeichen	Bedeutung
>	Erhöhe den Pointer um 1
<	Verringere den Pointer um 1
+	Erhöhe den Wert des Bytes auf das der Pointer zeigt
-	Verringere den Wert des Bytes auf das der Pointer zeigt
.	Gib den Wert des Bytes, auf das der Pointer zeigt, aus
,	Lese ein Byte von der Konsole und speichere dieses Byte an der Adresse auf die Pointer zeigt
[	Springe hinter die zugehörige „]“, wenn der Wert des Bytes auf das Pointer zeigt 0 ist
]	Springe zurück zu der zugehörigen „[“, es sei den der Wert des Bytes auf das Pointer zeigt ist 0

Abb. 5: Der Brainfuck Befehlszeichensatz (Vgl. Muppetlabs 2015)

```
$ ./protyp1.exe
Please enter desired program output: hallo
--------------------------------------------------------------------------------
1. Verifying that functions work properly (WARNING: methods can falsely pass test)...
2. Generating initial population
Best: "-++.-++++.+.++.-+.+++-+-+++++---+---+-----+-++--+++-++++.+-+++..++--+..--+--+--.-..++-++++--+.+++.+", output: "befhhloopplkkor", desired: "hallo"
3. Done Initializing
--------------------------------------------------------------------------------

Generation     Score                   Output

    10     |    3    |   "gblln"
    20     |    1    |   "hbllo"
    30     |    1    |   "hbllo"
    40     |    1    |   "hbllo"
    50     |    1    |   "hbllo"
    60     |    1    |   "hbllo"
    70     |    1    |   "hbllo"
    80     |    1    |   "hbllo"
    90     |    1    |   "hbllo"
   100     |    1    |   "hbllo"
   110     |    1    |   "hbllo"
   120     |    1    |   "hbllo"
   130     |    1    |   "hbllo"
   140     |    1    |   "hbllo"

Found Solution :D ... Summary:
        -> Program: "-++++-+--++--+-++++++.+-++-+----+---++------+.+-++++++++-++..+++--++.+--+-+-+-+-+-----+----+--+++"
        -> Output: "hallo"
        -> Desired Output: "hallo"
        -> scoring: 0
```

Abb. 6: Ein Durchlauf des ersten Prototypen

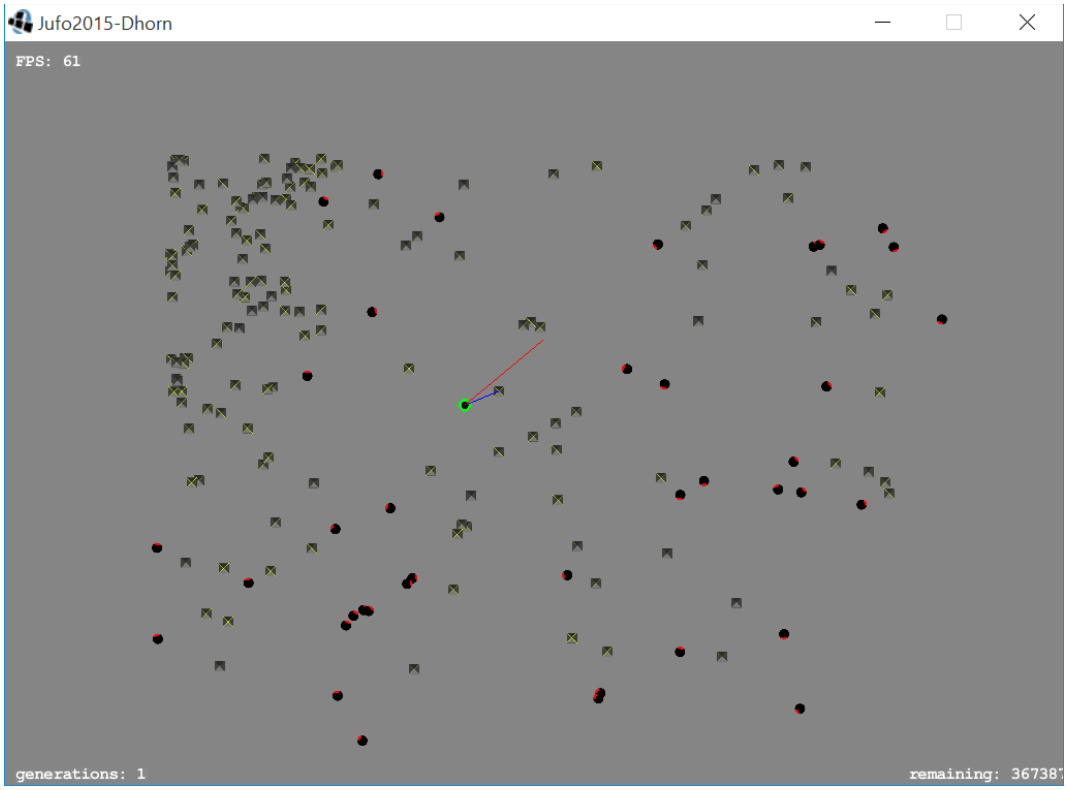

Abb. 7: Prototyp 2 zur Laufzeit. Die Käfer sind schwarz/rot und die Nahrungsstücke als graue Kästen dargestellt (Vgl. AI-Junkie-NeuralNetworks 2015)

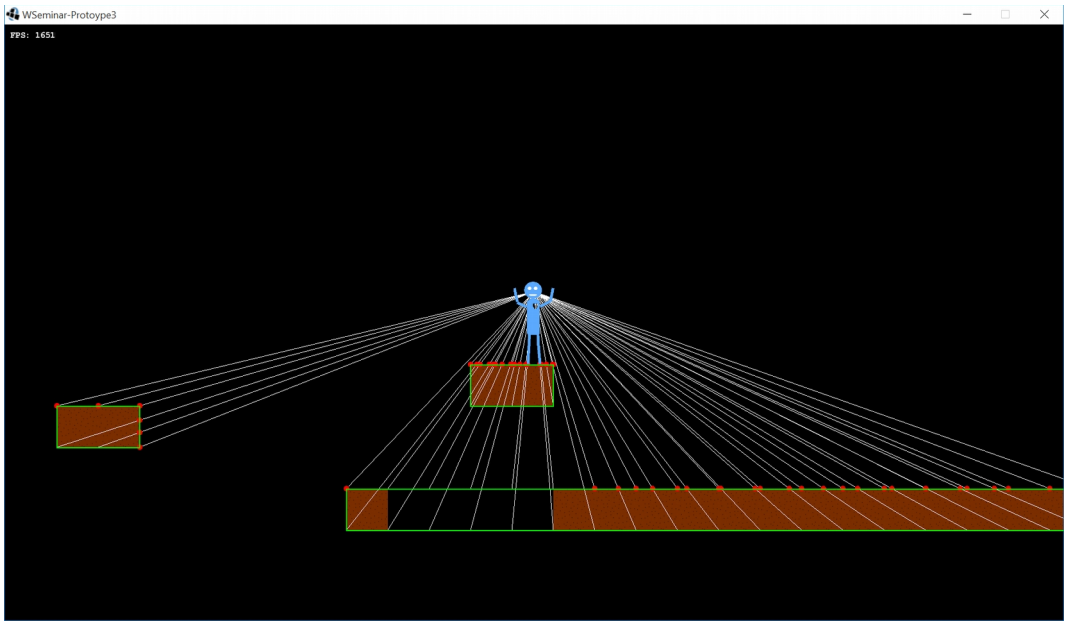

Abb. 8: Grafische Darstellung der „Raycasting“ Implementierung im dritten Prototypen. Die braunen Kästchen sind Plattformen der Spielwelt, auf welchen andere Objekte stehen können. Die grünen Linien stellen die Ränder der Plattformen dar. Diese werden für den Zweck des „Raycastings“ bei jedem Programmstart erneut berechnet. Der Spielcharakter ist in hellblau zu sehen. Die von seinen Augen ausgehenden, weißen Linien sind die so genannten „Rays“, welche seine Sichtlinien darstellen. In rot zu sehen sind die jeweiligen Punkte, an denen ein „Ray“ ein undurchsichtiges physikalisches Objekt versucht zu durchdringen. Es ist im Vergleich zu Abbildung 9 deutlich zu erkennen, dass diejenigen Plattformen, welche für die Spielfigur nicht zu sehen sind, ausgeblendet werden.

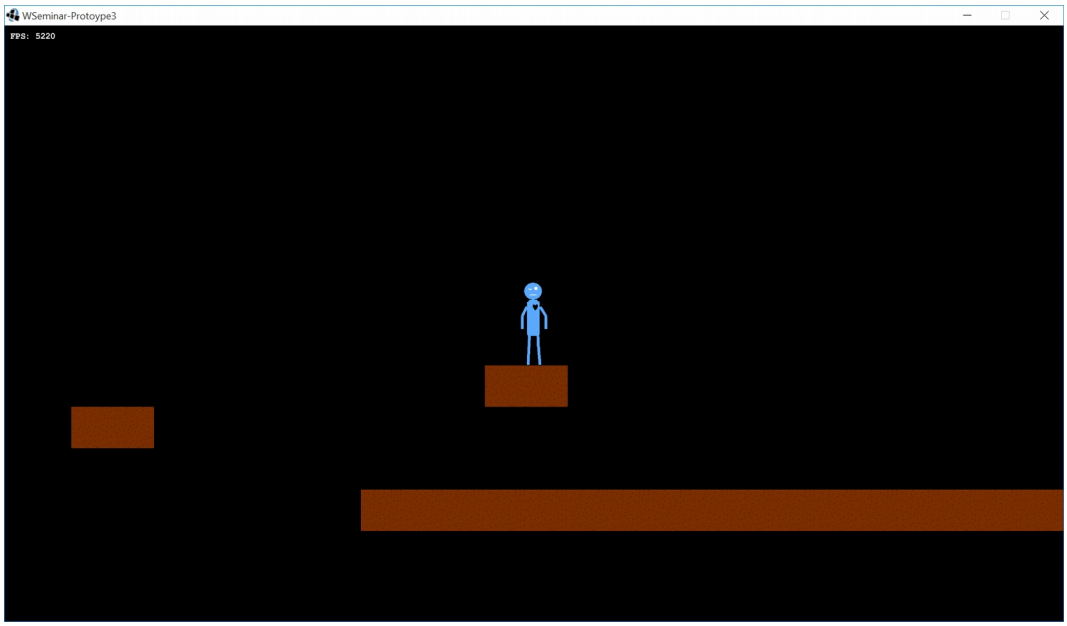

Abb. 9: Grafische Darstellung der Spielszenerie ohne „Raycasting“. Alles wird ungeachtet der Sichtlinien angezeigt.

6. Bibliographie

- GolemDeepLearning (Hrsg., 2015): „Maschinen, die wie Menschen lernen“ http://www.golem.de/news/deep-learning-maschinen-die-wie-menschen-lernen-1510-116468.html (Stand 06.10.2015)

- Stress (Hrsg. 2015): „Stress and Heart Disease“ http://www.stress.org/stress-and-heart-disease/ (Stand 04.11.2015)

- AI-Junkie-GA (Hrsg., 2015): „Genetic Algorithms in Plain English“ http://www.ai-junkie.com/ga/intro/gat1.html (Stand 04.11.2015)

- TU-Chemnitz (Hrsg., 2001/2002): „Genetische Programmierung in C++ https://www.tu-chemnitz.de/informatik/ThIS/downloads/courses/ws01/gp/heinich.pdf (Stand 04.11.2015)

- Muppetlabs (Hrsg., 2015): „The Brainfuck Programming Language“ http://www.muppetlabs.com/~breadbox/bf/ (Stand 31.10.2015)

- AI-Junkie-NeuralNetworks (Hrsg., 2015): „Neural Networks in Plain English“ http://www.ai-junkie.com/ann/evolved/nnt1.html (Stand 20.04.2015)

- Gamedev-tutsplus (Hrsg., 2015): „How to Create a Custom Physics Engine: The Basiscs and Impulse Resolution“ http://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331 (Stand 04.11.2015)

- Redblobgames (Hrsg., 2015): „2d Visibility“ http://www.redblobgames.com/articles/visibility/ (Stand 04.11.2015)

- HawkingBBC (Hrsg., 2014): „Stephen Hawking warns artificial intelligence could end mankind“ http://www.bbc.com/news/technology-30290540 (Stand 07.11.2015)

7. Quelltext

Aufgrund des Umfangs der Quelltexte der einzelnen Prototypen wurden diese nicht mit abgedruckt. Sie können online unter den jeweils angegebenen Links eingesehen werden:

1. Prototyp:	https://github.com/DominikHorn/WSeminar_Prototyp1
2. Prototyp:	https://github.com/DominikHorn/WSeminar_Prototyp2
3. Prototyp:	https://github.com/DominikHorn/WSeminar_Prototyp3

8. Unterschriebene Erklärung

Ich habe diese Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt.

_____

(Ort, Datum und Unterschrift des Schülers)