Maschinelles Lernen durch Selektion

Ein Projekt von Simon Köhl, Q2, St. Michael Gymnasium Bad Münstereifel

Kurzfassung:

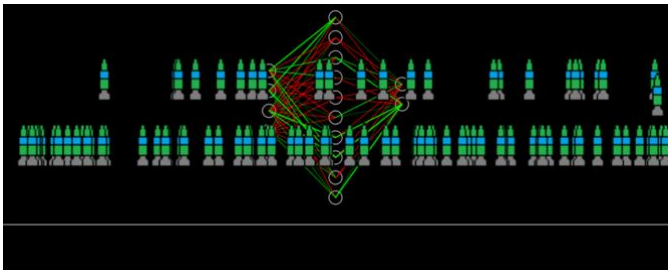
Das Ziel dieses Projektes ist es, die Möglichkeiten und Fähigkeiten künstlicher Intelligenz zu demonstrieren und auf den Prüfstand zu stellen.

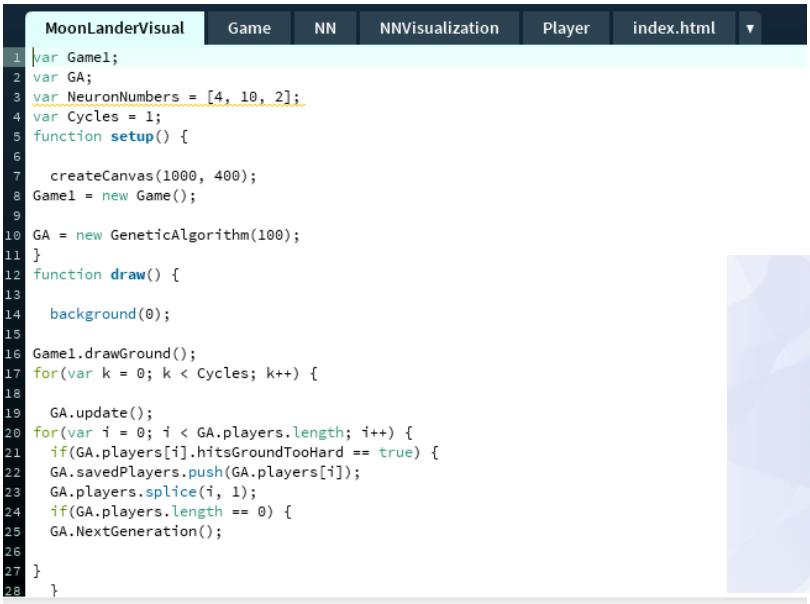
Hierfür wurde zuerst ein neuronales Netzwerk entwickelt, welches dann in Verbindung mit genetischen Algorithmen gebracht wurde und schließlich anhand mehrerer selbst programmierter 2D-Spiele getestet wurde. Das daraus resultierende Programm lernt also selbstständig, Spiele so gut wie möglich zu spielen.

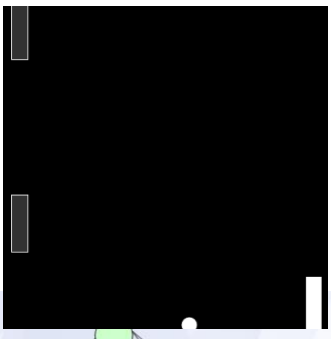
Zentrale Aspekte dieser Arbeit liegen also bei:

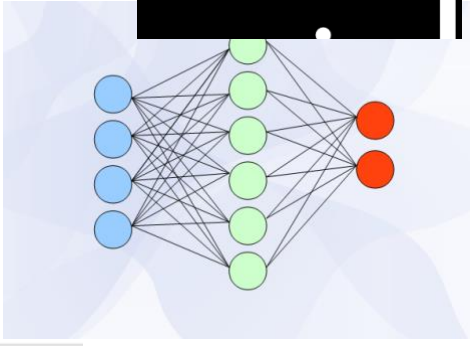
- der Entwicklung und Programmierung des neuronalen Netzwerkes
- der Implementierung des genetischen Algorithmus
- der Funktionsweise der 2D-Spiele

und der Performance des Programmes in verschiedenen Spielen.









Inhaltsverzeichnis:

1. Einleitung

1.1 Inspiration

Da ich mich persönlich sehr für Informatik und vor Allem für den Teilbereich der künstlichen Intelligenz interessiere, kam mir die Idee, meine privaten Interessen mit Jugend Forscht zu verbinden und ein Projekt zu entwickeln. Die ursprüngliche Idee lag hier bei einem Programm, welches anhand eines physikalisch möglichst genauem Spiel lernt, eine Rakete zu steuern. Hiermit sollten das fast schon Science-Fiction-ähnliche Potenzial und damit die ungeheuren Fähigkeiten dieser Technologien klar dargestellt werden. Da ich mich zwar schon des Öfteren mit neuronalen Netzwerken, allerdings weniger mit der Technik des Reinforcement Learning beschäftigt habe, war dieses Gebiet für mich quasi Neuland. Dementsprechend versuchte ich zuerst, mich an das Problem anzunähern beziehungsweise überhaupt einen funktionierendes Programm dahingehend zu schreiben. So waren die ersten Spiele, an denen die Programme getestet wurden sehr einfache Spiele, wie zum Beispiel Pong oder einfach ein 3 * 3 Raster, in dem zwei Felder dem Spieler Schaden zufügen und er zu einem Feld auf der anderen Seite des Rasters mit möglichst wenig Schaden gelangen musste. Diese Versuche beruhten auf dem Prinzip des Q-Learnings und als ich feststellte, dass sich hier keine Erfolge einstellten, da ich noch zu unerfahren mit diesem Forschungsgebiet war, wechselte ich meine Strategie. Einer meiner ersten Versuche eine künstliche Intelligenz zu programmieren liegt bereits ungefähr drei Jahre zurück. In diesem Versuch lernte das Programm nicht durch neuronale Netzwerke, sondern durch Evolution, so hatte ich die ersten Erfahrungen mit genetischen Algorithmen. Ich programmierte diesen Versuch nach einer Video-Anleitung des YouTube-Kanals „The Coding Train“ und eignete mir somit durch „Learning-by-doing“ die Konzepte genetischer Algorithmen an. Durch diese Vorkenntnisse und die Tatsache, dass ich des Öfteren Videos über die sogenannte NeuroEvolution sah, versuchte ich also, das Projekt mithilfe von NeuroEvolution zu verwirklichen.

1.2 Realisierung

Um ein solches Programm realisieren zu können, müssen zuerst die wesentlichen Bestandteile des Programmes genannt werden. Dies wäre zuerst ein Spiel, welches erlernt werden soll. Zweitens ist ein neuronales Netzwerk zu nennen, die Kernfunktion, welche die Welt um den Spieler herum beobachten kann. An dritter Stelle steht der genetische Algorithmus, welcher das Lernen erst ermöglicht.
Als Testplattform für das Programm wurden insgesamt drei Spiele programmiert: „Pong“, „Flappy Bird“ und „MoonLander“. In Kapitel 2.3 wird mehr auf die einzelnen Spiele eingegangen werden.
Das neuronale Netzwerk wurde selbst programmiert, es wurde keine Zweitlibrary benutzt, lediglich für die Matrixoperationen wurden wieder Videos von „The Coding Train“ zurate gezogen.
Das Wissen über den genetischen Algorithmus habe ich mir ebenfalls durch „The Coding Train“ erschlossen und angewendet.
Anschließend an die Entwicklung wurde das Programm auf verschiedene Aspekte

getestet und die Ergebnisse dieser Tests wurden dokumentiert und interpretiert.
Die Idee, sich ein Beispiel an der Evolution zu nehmen und durch Evolution die neuronalen Netzwerke an die Umgebung anzupassen, gibt es bereits seit 1994. So gab es beispielweise 1994 (siehe Q1) das Experiment „Genetic lander: An experiment in accurate neuro-genetic control“ von Edmund Ronald und Marc Schoenauer (siehe Q2).
Bei der Durchführung stellten sich allerdings einige Problematiken heraus. War ein Spiel beispielsweise nicht ordnungsgemäß programmiert und beinhaltete unbekannte Bugs, konnte es vorkommen, dass die künstliche Intelligenz diese Bugs durch Zufall fand. Ein Beispiel für einen dieser Bugs wird im Kapitel „Problematiken“ genannt werden.
Es galt also, herauszustellen, wie und ob dieses Projekt zu realisieren ist und die allgemeine Fragestellung hierbei liegt darin, wie effizient und effektiv dieses Programm in Relation zum mit der Entwicklung verbundenen Aufwand zur Lösung von Problemen ist.

2. Funktionsweise

2.1 Neuronales Netzwerk

Ein „neuronales Netzwerk“ (auch: „neuronales Netz“, englisch: „neural network“) oder „künstliches neuronales Netz (kurz: „KNN“, englisch: „ANN“) ist eine stark vereinfachte Simulation biologischer Prozesse im Gehirn. Auf diese biologischen Prozesse wird hier nicht genauer eingegangen werden, da dies den Rahmen dieser schriftlichen Arbeit sprengen würde. Die biologische Modellannahme allerdings lässt sich wie folgt beschreiben: Die Grundbestandteile des Gehirns setzen sich zusammen aus Neuronen und Synapsen. Die Neuronen sind durch Synapsen miteinander verbunden, jedes Neuron hat im Modell mehrere Eingänge und nur einen Ausgang, an diesem Ausgang können allerdings wieder mehrere Synapsen sitzen. Wenn die Signale, die das Neuron über die Eingänge als elektrisches Signal empfängt, einen bestimmten Schwellenwert erreichen, feuert dieses und gibt somit ebenfalls ein elektrisches Signal weiter, im Netzwerk gegebenenfalls sogar an andere Neuronen. Das Netzwerk lernt dann durch die Gewichtung verschiedener Eingänge. Diese Gewichtung findet durch die Änderung der Synapsen statt.
Im KNN ist ein Neuron eine Funktion und eine Synapse ein sogenanntes „Gewicht“ beziehungsweise „Weight“, welches durch einen Zahlenwert, meistens zwischen -3 und 3 repräsentiert ist. Der Output eines Neurons wird als Aktivierungswert bezeichnet. Der Aufbau des hier benutzten KNN besteht aus einzelnen „Layern“, also Schichten, in denen unterschiedlich viele Neuronen sitzen. Im „Input-Layer“ werden verschiedene Spielwerte eingespeist, auf dessen Basis das Programm die Umwelt im Spiel betrachten und verstehen kann. So kann zum Beispiel eine Videoaufnahme des Spiels als Input gelten, indem jedes einzelne Neuron den normalisierten

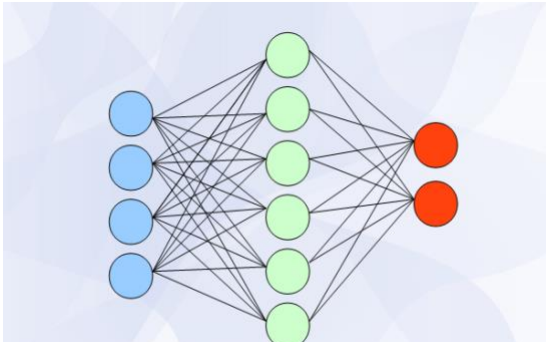


Abbildung 1: Modell eines KNN

Pixelwert des momentanen Bildes annimmt. Ein vereinfachtes Beispiel hierfür wäre ein 4x4-Bild als Input, also 16 Pixel, somit auch 16 Neuronen. Jeder Pixel kann einen Wert von 0 bis 255 annehmen (vereinfacht) und da die Werte der Neuronen meistens zwischen 0 und 1 liegen müssen, werden diese Pixelwerte normalisiert, indem alle Werte durch 255 geteilt werden. Technisch gesehen sitzen in diesem „Input-Layer“ bei meinem Programm keine Neuronen, da hier einfach die Werte als Aktivierungswerte genommen werden, im ersten Layer gibt es also noch keine „Aktivierungsfunktion“. Die „Aktivierungsfunktion“ ist ein essenzieller Bestandteil eines Neurons, denn durch die Funktion wird erst der Schwellenwert implementiert. Je nachdem welche Aktivierungsfunktion gewählt wird, kann so durch eine Kombination linearer Funktionen eine nicht-lineare Funktion angenähert werden. Dies ist die Kernaufgabe eines neuronalen Netzwerkes, denn es ist nichts anderes als eine mathematische Funktionsannäherungsmaschine, oder im Englischen: „function approximator“. Die Kombination findet durch das Vernetzen mehrerer Layer statt, was uns zu den sogenannten „Hidden-Layers“ (siehe Abb. 1 alle grünen Neuronen) führt. Diese sitzen zwischen dem Input- (siehe Abb. 1 alle blauen Neuronen) und dem Output-Layer (siehe Abb.1 alle roten Neuronen). Die Anzahl der Hidden-Layer kann stark variieren, je nach Komplexität des Problems wird sie allerdings relativ klein gehalten, da sonst verschiedene Probleme auftreten können, aber mehr dazu später. Durch einen Algorithmus namens „Feed-Forward“ wird also, dem Namen gemäß, der Input durch das Netzwerk in Richtung Output-Layer gefüttert. Der Output-Layer besteht dann wiederum aus Neuronen, welche wieder Werte von 0 bis 1 annehmen können, und auf Basis dieser Werte werden dann Aktionen im Spiel getätigt. Ein neuronales Netzwerk nimmt also einfach verschiedene Beobachtungen aus einem Spiel und entscheidet darauf basierend, was getan werden soll. Die verschiedenen Layers sind untereinander mit Weights verbunden, also liegen zwischen zwei Layern immer Weights, und wenn diese Weights immer von jedem Neuron im vorherigen zu jedem Neuron im nächsten Layer gehen, ist das Netzwerk „fully connected“. Die Weights haben ebenfalls einen Einfluss auf die Werte, da der Netzinput (welcher der Input für die Aktivierungsfunktion ist) eines Neurons im momentanen Layer die Summe aller Aktivierungswerte der Neuronen aus dem vorherigen Layer multipliziert mit den jeweiligen Gewichten. Da in einem Layer aber meistens mehr als nur ein Neuron liegt und jedes Neuron mit jedem Neuron aus dem vorherigen Layer verbunden ist, wäre die einzelne Berechnung der Werte sehr rechenaufwendig. Darum werden die Werte durch Matrixoperationen berechnet, worauf später in der Implementierung weiter eingegangen wird.
Die Gewichte verändern also die Werte, die nach vorne durchgegeben werden, also haben sie auch einen Einfluss auf die Endwerte. Da diese Weights anfangs randomisiert sind mit Werten zwischen -3 und 3, wird das neuronale Netzwerk also sehr wahrscheinlich keinen sinnvollen Output leisten. Die volle Kraft neuronaler Netzwerke wird nämlich erst im Lernprozess entfaltet, da hier anhand verschiedener Informationen verallgemeinert werden kann und somit bereits vorhandenes Wissen auf ähnliche, allerdings nie zuvor gesehene Spielzustände angewendet werden kann. Dieses Lernen manifestiert sich im graduellen Verändern und Anpassen der Weights, sodass der Output so geändert wird, dass das Programm vernünftige Aktionen unternimmt. Hier stellt sich allerdings das Problem, wie bestimmt werden soll, ob die graduelle Veränderung einen positiven oder negativen Effekt auf die Performance des neuronalen Netzwerkes hat. An diesem Punkt kommen genetische Algorithmen zutrage, welche im Folgenden Kapitel weiter erläutert werden.

2.1.1 Implementierung

Die neuronalen Netzwerke in diesem Projekten erhalten als Input keine Bildwerte, sondern lediglich Werte wie beispielsweise die eigene x- und y-Position und die x- und y- Position des Balles (am Beispiel von „Pong“). Dies wurde so gewählt, damit der Rechenaufwand relativ klein bleibt und mögliche Probleme einfacher erkannt und behoben werden können, da der Input so auf einer niedrigeren Abstraktionsebene liegt. Das Spiel wird gestartet, allerdings mit mehreren Spielern, anstatt einem, aufgrund des genetischen Algorithmus. Jeder einzelne Spieler hat ein individuelles KNN, also ein eigenes Gehirn, welches Entscheidungen trifft. Die Parameter sind anfangs komplett zufällig gewählt. Die Outputs des jeweiligen KNN bestimmen dann die Aktionen des jeweiligen Spielers im Spiel. Wenn zum Beispiel Neuron 1 im Output-Layer am stärksten feuert, der Aktivierungswert also am größten ist, wird dementsprechend die Aktion, die mit Neuron 1 assoziiert ist, getätigt.

2.1.2. Programmierung

Das komplette Programm wurde in JavaScript geschrieben, mit der Library „p5.js“ welche für die grafische Darstellung der Spielmechanik zuständig ist.

Der erste Schritt bei der Programmierung eines KNN ist die Initialisierung der Werte.

Code 1:

```
function NeuralNetwork(NeuronNums) {
        this.Layers = [];
        this.Layers[0] = new Layer(NeuronNums[0], 4, 0);
        this.LayerCount = NeuronNums.length;
        for( var i = 1; i < this.LayerCount; i++) {
                this.Layers[i] = new Layer(NeuronNums[i], this.Layers[i-
        1].NeuronNumber, 1);
        }
}
```

Erläuterung:

- **function NeuralNetwork(NeuronNums)**

Bei der Erstellung eines KNN werden verschiedene Werte benötigt. NeuronNums ist hier ein Array mit der Anzahl an Neuronen pro Layer, die insgesamte Länge dieses Arrays ist die Anzahl der Layer. Wenn NeuronNums zum Beispiel NeuronNums = [4, 7, 3] ist, so hat das KNN insgesamt drei Layer, der erste Layer hat vier, der zweite sieben und der dritte drei Neuronen.

- **this.Layers = [];**

  this.Layers[0] = new Layer(NeuronNums[0], 4, 0);

In diesen beiden Zeilen werden zuerst die Layer des KNN als leeres Array definiert, in der zweiten Zeile wird diesem Array bei Index 0, also bei dem ersten Platz im Array, ein Layer hinzugefügt mit der Neuronenanzahl NeuronNums[0], also welcher Wert auch immer an erster Stelle in NeuronNums steht. Die Zahl 4 ist bei diesem Layer ohne Bedeutung, da diese die Anzahl der Neuronen aus dem vorherigen Layer beschreibt, da dies allerding der erste Layer ist, gibt es keinen vorherigen Layer. Die Zahl 0 ist hier ein Schalter, welcher bestimmt, ob der Layer Gewichte bekommen soll oder nicht. Da dies wie gesagt der erste Layer ist, und die Gewichte eines Layers immer die zwischen ihm und dem vorherigen Layer sind, benötigt Layer[0] keine Gewichte.

- **this.LayerCount = NeuronNums.length;**

  for(var i = 1; i < this.LayerCount; i++) {

  this.Layers[i] = new Layer(NeuronNums[i], this.Layers[i-

1].NeuronNumber, 1);

  }

In diesen drei Zeilen werden die restlichen Layer definiert, indem zuerst die Anzahl der Layer in eine lokale Variable geschrieben wird und anschließen über das Layers-Array iteriert wird und für jeden Eintrag ein neuer Layer eingefügt wird.

Die genaue Beschreibung des Layer-Objektes wird ausgelassen, da dies sonst den Rahmen dieser schriftlichen Arbeit sprengen würde.

Damit die Inputs auch nach vorne durch das KNN propagiert werden können, benötigt das neuronale Netzwerk noch eine Feed-Forward-Funktion:

Code 2:

```
this.FeedForward = function(Input) {
        this.Layers[0].Outputs = Input;
        for(var i = 1; i < this.LayerCount; i++) {
                this.Layers[i].integsum = MatMul(this.Layers[i].Weights,
        this.Layers[i - 1].Outputs, this.Layers[i].NeuronNumber);
                for(var j = 0; j < this.Layers[i].Outputs.length; j++) {
                        this.Layers[i].Outputs[j] =
                this.activateSigmoid(this.Layers[i].integsum[j] +
                this.Layers[i].Biases[j]);
                }
        }
return this.Layers[this.Layers.length-1].Outputs;
```

}

Erläuterung:

- **this.FeedForward = function(Input) {**

Es wird eine Funktion im Objekt „NeuralNetwork“ definiert, welche als Parameter das Array „Input“ erhält. Dieses Array besteht aus den Werten, welche aus dem Spiel an das KNN weitergegeben werden.

- **this.Layers[0].Outputs = Input;**

Der Output des ersten Layers wird dem Input gleichgesetzt, dieser Schritt sorgt dafür, dass die Werte in das KNN eingespeist werden.

- **for(var i = 1; i < this.LayerCount; i++) {**

```
this.Layers[i].integsum = MatMul(this.Layers[i].Weights, this.Layers[i -
1].Outputs, this.Layers[i].NeuronNumber);
```

Mithilfe eines for-Loops wird über alle Layer mit Ausnahme des ersten iteriert. Der erste Teil dieses for-Loops besteht daraus, dass die „integsum“, also der Netzinput für jeden Layer berechnet werden. Dies geschieht durch eine Matrixmultiplikation der Weights des momentanen Layers mit den Outputs des vorherigen Layers.

- **for(var j = 0; j < this.Layers[i].Outputs.length; j++) {**

```
this.Layers[i].Outputs[j] = this.activateSigmoid(this.Layers[i].integsum[j]
+ this.Layers[i].Biases[j]);
}
```

Der zweite Teil des for-Loops besteht aus einem weiteren for-Loop, in dem alle Outputs des momentanen Layers durch die Aktivierungsfunktion (hier eine Sigmoid-Funktion), welche den Netzinput addiert mit dem Bias als Input erhält, berechnet.

- **return this.Layers[this.Layers.length-1].Outputs;**

Somit ist der Output des momentanen Layers bestimmt und es kann zum nächsten Layer gesprungen werden.
Die benutzte Hilfe für diesen Code kann im Videoquellenverzeichnis unter V1 gefunden werden. Die Quelle für die Matrix-Library kann ebenfalls im Videoquellenverzeichnis unter V1 sowie im Quellenverzeichnis unter Q6 gefunden werden.

2.2 Genetischer Algorithmus

Genetische Algorithmen orientieren sich an der biologischen Evolution. Diese besteht darin, dass durch graduelle Veränderungen im Genom die Angepasstheit eines Individuums verändert wird, und durch natürliche Selektion das besser angepasste Genom ausgewählt wird, da aus einer besseren Anpassung höhere Überlebenschancen resultieren. Genetische Algorithmen bestimmen also aus einem Pool vorhandener Genome die besten und konstruieren aus diesen Informationen den Genpool der nächsten Generation. Sie werden also meistens zur Optimierung eingesetzt (siehe Q3) und dienen in diesem Projekt in Kombination mit neuronalen Netzwerken als Lernfunktion. So besteht das Genom eines Spielers aus den Weights seines neuronalen Netzwerkes, also den Parametern, welche verändert werden, um

den bestmöglichen Output zu liefern. So bekommt also jeder Spieler anfangs zufällige Parameter zugewiesen, und es wird evaluiert, welcher Spieler mit diesen zufälligen Parametern am besten angepasst ist, also den höchsten Score bekommt. Diesem Spieler wird nun im Genpool eine höhere Chance zugewiesen, für die nächste Generation ausgewählt zu werden, also werden in der nächsten Generation mehr Spieler besser angepasst sein. Durch Mutationen, welche darin bestehen, dass mit einer zufälligen Rate zufällige Gewichte um einen zufälligen Wert geändert werden, finden so, abermals zufällig, kleinste Veränderungen statt. Zeichnen sich diese als vorteilhaft aus, wird die Chance für dieses Genom höher sein, in der nächsten Generation vertreten zu sein. Sind die Veränderungen allerdings negativ, wird die Chance sehr gering sein, das Genom stirbt als sozusagen aus. Somit kommt es zu einer graduellen Verbesserung, wodurch die Spieler das gegebene Spiel meistern und immer höhere Scores erzielen können, das Programm hat also gelernt, das gegebene Spiel zu spielen.

2.2.1 Implementierung

Der genetische Algorithmus bekommt in diesem Projekt eine Anzahl an Spielern pro Generation und erzeugt somit mehrere Spieler. Diese Spieler spielen dann das jeweilige Spiel und wer verliert, wird in einem Array namens „savedPlayers“ gespeichert. Wenn alle Spieler verloren haben, werden die besten Spieler bestimmt und deren Reproduktionschancen erhöht. Dies geschieht auf Basis der Berechnung der Fitness, welche einfach durch den jeweiligen Score repräsentiert wird. Nachdem dies passiert ist, wird die nächste Generation von Spielern auf Basis dieser Chancen erzeugt und mutiert und die Schleife startet von vorne.

2.2.2 Programmierung

Code 3:

```
class GeneticAlgorithm {
	constructor(TOTAL1) {
		this.savedPlayers = [];
		this.players = [];
		this.TOTAL = TOTAL1;
		for( var i = 0; i < this.TOTAL; i++) {
			this.players.push(new Player());
		}
		this.FitnessSum = 0;
		this.GenerationCount = 0;
	}
```

Erläuterung:

- `constructor(TOTAL1) {`
  `this.savedPlayers = [];`
  `this.players = [];`
  `this.TOTAL = TOTAL1;`

Zuerst werden in der Konstruktorfunktion die Werte initialisiert.
„this.savedPlayers“ wird zuerst als leeres Array initialisiert, ebenso wie „this.players“.
„this.TOTAL“ wird gleich TOTAL1 gesetzt, welches die maximale Spieleranzahl repräsentiert. „this.savedPlayers“ ist wie gesagt das Array mit den gespeicherten Spielern, „this.players“ das Array mit den momentan spielenden Spielern.

- **for(var i = 0; i < this.TOTAL; i++) {**
  this.players.push(new Player());
}

In diesem for-Loop wird über das Spielerarray iteriert und es werden so viele Spieler hinzugefügt, wie anfangs angegeben wurden.

- **this.FitnessSum = 0;**
  this.GenerationCount = 0;

Der Variablen „this.FitnessSum“ wird der Wert null zugewiesen, sie wird später für die Berechnung der Fitness verwendet werden. „this.GenerationCount“ ist die Anzahl der Generationen und da am Anfang noch null Generationen gelebt haben, wird ihr ebenfalls der Wert null zugewiesen.

Code 4:

```
NextGeneration() {
    this.CalculateFitness();
    for(var i = 0; i < this.TOTAL; i++) {
        this.players.push(this.PickTwo());
    }
    this.GenerationCount++;
    console.log("Current Generation: " + this.GenerationCount);
}
```

Erläuterung:

- **NextGeneration() {**

Die Funktion „NextGeneration“ wird definiert.

- **this.CalculateFitness();**

Die Funktion “this.CalculateFitness” wird aufgerufen. Diese Funktion berechnet die Angepasstheit der Spieler an die Umgebung und ordnet dieser eine Auswahlwahrscheinlichkeit zu. Auf sie wird nicht tiefer eingegangen werden, da dies den Rahmen sprengen würde.

- **for(var i = 0; i < this.TOTAL; i++) {**
  this.players.push(this.PickTwo());
  }

In diesem for-Loop wird über die momentanen Spieler iteriert und für jeden Spieler werden zwei Elternspieler ausgesucht. In diesem Beispiel wird ein CrossOver-Algorithmus anstelle eines puren Mutations-Algorithmus verwendet. Auf diese Auswahl-Algorithmen wird ebenfalls nicht tiefer eingegangen werden um die

Erläuterung angemessen kurz zu halten. Zusammengefasst lässt sich sagen, dass diese Algorithmen auf Basis der in CalculateFitness berechneten Wahrscheinlichkeiten neue Spieler aus savedPlayers auswählen und mutieren.
- **this.GenerationCount++;**

console.log("Current Generation: " + this.GenerationCount);

Anschließend wird die Anzahl an Generationen um eins erhöht und die momentane Generation in die Konsole geschrieben. Dies dient zum Auswerten und Interpretieren der Effizienz, da aufgrund dieser Daten bestimmt werden kann, wie viele Generationen benötigt werden, um einen bestimmten Score zu erreichen.

Die Quellen für diesen Algorithmus lassen sich unter V1 im Videoquellenverzeichnis finden.

2.3 Spiele

Die einzelnen Spiele, an denen das Spiel getestet wurde, sind größtenteils Kopien von bekannten 2D-Spielen. So sind zwei dieser Spiele sehr bekannte 2D-Spiele, zum ersten Pong, dann Flappy Bird.
Es wurden deshalb 2D-Spiele verwendet, weil die Kausalitäten in 2D-Spielen meistens nicht so abstrakt wie in 3D-Spielen sind und somit die Spiele für die neuronalen Netzwerke schneller und einfacher zu erlernen sind. Dies sorgt für schnellere Berechnungen, da kleinere Netzwerke verwendet werden können und die allgemeine Performance der KNNs somit verbessert werden kann.

2.3.1 Pong

Das Spiel Pong (1972 von Atari veröffentlicht und von Allan Alcorn entwickelt (Q4)) war eines der ersten Videospiele überhaupt und wurde noch auf TV-Bildschirmen gespielt, welche mit einem Joystick-System verbunden waren. Es zeichnet sich durch seinen charakteristischen, simplen Aufbau aus, da lediglich zwei sogenannte „Panel“ und ein Ball, im Originalspiel auch ein Score auf dem Bildschirm zu sehen sind. Zwei Spieler spielten so gegeneinander, indem versucht wurde, mit dem Joystick das eigene Panel so zu steuern, dass der Ball daran abprallt und nicht daran vorbei aus dem Fenster fliegt. Ein Computer kann ein Panel ebenfalls durch einfache, fest einprogrammierte Logik steuern. In diesem Projekt wird ein Panel von solch einem Logikcomputer gespielt, gegen ihn spielt die NeuroEvolution-KI. Der Unterschied zwischen beiden besteht darin, dass die KI außer den gegebenen Werten nichts über das Spiel weiß und es somit komplett von neuem erlernen muss. Das Gegnerpanel kann nichts erlernen, da dieses „Verhalten“ fest einprogrammiert ist. Die Spieler bekommen lediglich die eigene y-Position, die Position des Balles und die y-Position des Gegners als Input. Die KNNs haben also vier Inputs, einen Hidden-Layer mit zehn Neuronen und zwei Outputs: Hoch oder runter.

2.3.2 Flappy Bird

Flappy Bird wurde 2013 veröffentlicht und von Dong Nguyen entwickelt (Q5). Es ist ein relativ modernes Smartphone-Spiel, in dem der Spieler einen Vogel steuert, welcher bei jedem Tippen des Spielers ein kleines Stück nach oben fliegt. Der Vogel

fliegt auf Rohre zu, denen er ausweichen muss. Das Spiel ist besonders dafür bekannt, dass ein hoher Score sehr schwer zu erreichen ist, da der Abstand zwischen den Rohren meist sehr gering ist und die Flughöhe des Vogels sehr gut abgeschätzt werden muss. In der Originalversion des Spiels durfte der Vogel weder die Rohre, noch die Decke, noch den Boden berühren, in dieser Version wurde diese Funktion allerdings nicht eingebaut, was zu Problemen im Lernprozess der KI führte. Da dies allerdings eine gute Möglichkeit ist, Problematiken mit KIs darzustellen, wurde das Spiel so belassen und die Probleme werden später weiter veranschaulicht werden. Die Spieler bekommen die eigene y-Position, den Abstand zu den nächsten beiden Rohren und die jeweilige Länge der beiden Rohre als Input. Die KNNs haben also jeweils vier Inputs, zwei Hidden-Layer mit 20 und 12 Neuronen und zwei Outputs: Fliegen oder nicht fliegen.

2.3.3 MoonLander

MoonLander wurde für dieses Projekt entwickelt und ist an das Spiel „Lunar Lander“ angelehnt. Es hat weniger einen Spielcharakter, da dies das eigentliche Ziel dieser Projektarbeit beinhaltet, nämlich einem neuronalen Netzwerk das Steuern einer Rakete beizubringen. In diesem Spiel geht es darum, eine Rakete zu landen, allerdings hat man nur eine bestimmte Füllung Treibstoff. Wenn dieser Treibstoff leer ist, fällt die Rakete zu Boden. Trifft die Rakete den Boden zu hart, muss man von vorne starten, falls eine gute Landung gelingt, wird die Rakete wieder an den Startpunkt gesetzt, der Score bleibt allerdings erhalten. Nach jeder Landung wird eine neue zufällige Tankfüllung bestimmt, somit muss das KNN lernen, mit verschiedenen Tankfüllungen umzugehen und die Steuerung soweit zu verallgemeinern, dass eindeutig wird, dass mehr Schub gleich mehr Kraftstoffverbrauch ist. Die Spieler bekommen die eigene Höhe (y-Position), Tankfüllung und Geschwindigkeit als Input. Das KNN hat also drei Inputs, einen Hidden-Layer mit zehn Neuronen und zwei Outputs: Schub oder kein Schub.

3. Performance

Im folgenden Kapitel wird die Performance und der Nutzen dieses Programmes in Relation zum Aufwand anhand von Versuchen bewertet werden.

3.1 Versuch zur Bestimmung der allgemeinen Anwendbarkeit

Versuchsablauf:
Das Programm wird bei allen drei Spielen getestet und es wird beobachtet, wie viele Generationen durchschnittlich
benötigt werden, um das Spiel zu erlernen. Ein Spiel gilt dann als erlernt, wenn mindestens ein Spieler so gut in dem gegebenen Spiel ist, dass in der nächsten Zeit keine neuen Generationen auftreten werden, da dieser Spieler das Spiel immer weiter spielt.

Beobachtung: Tabelle 1

Spiel	Versuch 1	Versuch 2	Versuch 3	Versuch 4	Versuch 5	Versuch 6	Versuch 7	Durchschnittliche Anzahl an Generationen
Pong	29	7	24	30	12	12	5	17
MoonLander	34	22	215	251	15	15	234	~ 112
Flappy Bird	%	%	%	%	%	%	%	%

Wie in Tabelle 1 zu erkennen ist, wurde das Spiel Pong mit Abstand am schnellsten gelernt. Nach durchschnittlich 17 Generationen gilt Pong als erlernt. Das Spiel MoonLander dagegen gilt erst nach durchschnittlich 112 Generationen als erlernt, was sowohl an der sehr kurzen Generationsdauer, als auch an der größeren Komplexität des Spiels liegen kann. Flappy Bird konnte nicht erlernt werden aufgrund eines Bugs, der bedingt, dass schlecht angepasste Vögel manchmal trotzdem einen sehr hohen Score erzielen (siehe Kapitel „Problematiken“).

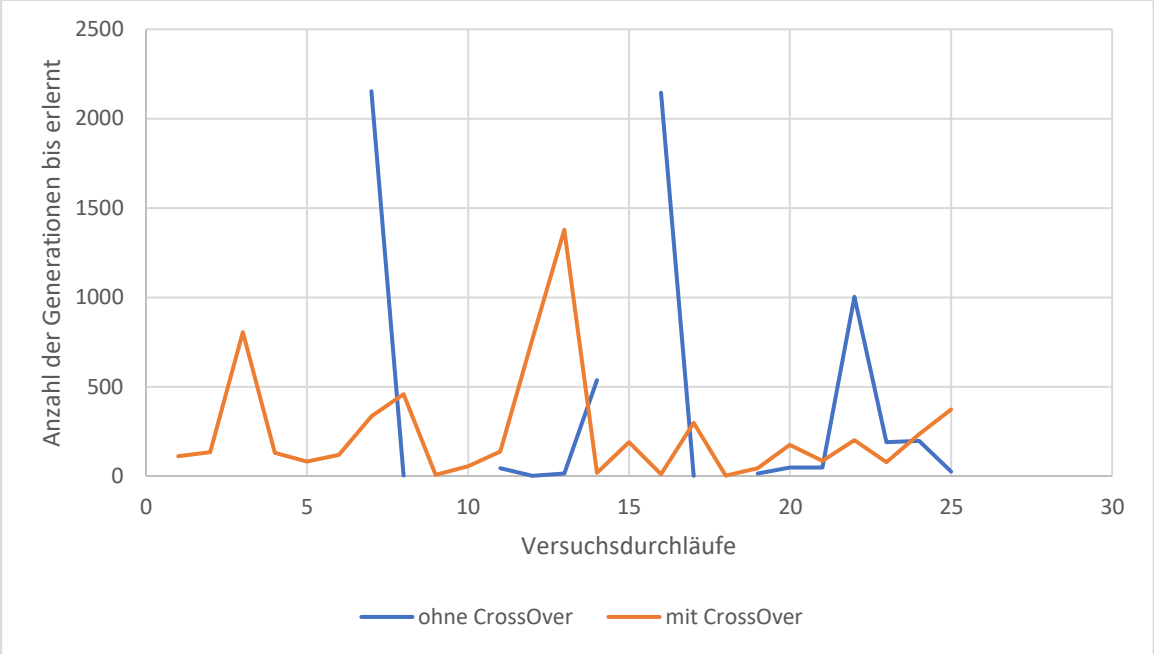
Ergebnis:
Das Programm kann auf mehrere Spiele angewendet werden, allerdings muss das Spiel so bugfrei wie möglich programmiert sein. Außerdem kann es, je nach Komplexität des Spiels, sehr lange dauern, bis ein Spiel erlernt ist. Dieses Problem kann allerdings gelöst werden, indem die Simulationsgeschwindigkeit angepasst wird. So dauerte bei diesem Versuch das Trainieren des Programmes auf MoonLander meistens ungefähr zehn Sekunden, da in diesem Programm die Möglichkeit einer Beschleunigung der Simulation möglich ist.

Vorteile:
Abstrahiert bedeutet das, dass nicht nur Spiele, sondern auch andere verschiedenste Dinge erlernt werden können, wie zum Beispiel physikalische Phänomene, die optimale Stoffzusammensetzung für ein Medikament, oder eben die treibstoffeffizienteste Steuerung einer Rakete.
Nachteile:
Die gegebene Umwelt muss so exakt wie möglich modelliert und simuliert werden, um ungewollte Lernentwicklungen zu verhindern.

3.1 Versuch zur Bestimmung der besseren Auswahlmethode für die nächste Generation

Versuchsablauf:
MoonLander wird einmal mit purer Mutation und einmal mit CrossOver trainiert und die benötigte Generationenanzahl wird über den Versuchsdurchläufen aufgetragen. Wenn das Programm nach 4000 Generationen das Spiel immer noch nicht erlernt hat, gilt das Spiel als ineffizient lernbar. Dies ist durch eine Lücke im Graphen zu erkennen.

Ergebnis: Graph 1



Beobachtung:
Die Versuchsdurchläufe ohne CrossOver weisen sehr häufig Lücken auf, das Spiel wurde nur in 17 Fällen erlernt, also gibt es hier eine Fehlerrate von 32%. Auch sind die Generationen, die zum Erlernen benötigt werden häufig mehr als bei den Versuchsdurchläufen ohne CrossOver. Hier lässt sich in diesen 25 Versuchsdurchläufen keine einzige Lücke finden, das Spiel konnte also in 100% der Fälle erfolgreich erlernt werden. Außerdem ist die durchschnittlich benötigte Generationenanzahl hier wesentlich kleiner, das Spiel konnte also schneller erlernt werden.

Ergebnis:
Die Methode des CrossOvers ist nicht nur wesentlich zuverlässiger, sondern optimiert ebenfalls die benötigte Zeit um das Programm zu trainieren.

4. Problematiken

Während des Entwicklungs- und Testungsprozesses kam es zu einigen Problematiken. Ein Beispiel hierfür ist die fehlerhafte Programmierung des Spiels „Flappy Bird“. Dadurch, dass es dem Spieler erlaubt war, die Decke und den Boden zu berühren in Kombination mit der Tatsache, dass manche Rohre nur eine Lücke zwischen der oberen Röhre und dem Boden bildeten, kam es dazu, dass die Spieler diese Fehler ausnutzten. Sie tätigten keine Aktionen, sondern glitten auf dem Boden umher, bis sie eine Röhre trafen. Durch diese Strategie konnten allerdings so viele Punkte gemacht werden, dass diese Spieler in Folgegenerationen häufiger vertreten waren. Da die Strategie beziehungsweise das Verhalten dieser Spieler allerdings für den Rest der Strecke unnütz war, kamen sie meist nur ein paar Rohre weit, bis sie verloren. Dennoch waren sie in diesem Szenario die Spieler mit den meisten Punkten. Falls also doch irgendwann ein Spieler das richtige Verhalten durch Zufall entwickelte und weiter kam als die anderen, hielt dieser Erfolg nicht lange an, da in

der nächsten Runde aufgrund der Fehler die untätigen Spieler wieder weiter kamen. Daraus ist zu folgern, dass das Programm möglicherweise in der Lage gewesen wäre, das richtige Verhaltensmuster für diese Umgebung zu entwickeln, durch die fehlerhafte Simulation eben dieser Umgebung kam es allerdings zur Zufriedenstellung durch ein lokales Maximum.
Dieses Phänomen ist auch in der Biologie zu beobachten: Wenn ein Lebewesen sich auf einem lokalen Maximum an eine Umgebung angepasst hat, ist es zwar nicht das beste Individuum, das es jemals in diesem Habitat geben wird, aber jeder Schritt in Richtung dieses globalen Maximums wäre zuerst ein Rückschritt, da das lokale Maximum zuerst verlassen werden müsste (Q7).
Ein weiteres Problem bei der Entwicklung dieses Projektes bestand darin, dass die Zeiten, in denen das Programm ohne Beschleunigung der Simulation die Spiele erlernte, meist mehrere Stunden betrugen und so schwer ein Fortschritt feststellbar war. Dieses Problem konnte jedoch durch das Hinzufügen der Beschleunigung gelöst werden.

5. Fazit

Zusammenfassend lässt sich sagen, dass durch die Technologie der NeuroEvolution die Möglichkeit besteht, dass Computer nicht nur Spiele, sondern auch komplexe Sachzusammenhänge erlernen und abstrahieren können. Mit der richtigen Simulation können somit Prozesse sowohl automatisiert, als auch optimiert werden. Dadurch kann es beispielsweise zu massiven Kosteneinsparungen bei der Produktion von Gütern, aber auch zu Revolutionen in der Medizin, Technik und Wissenschaft im Allgemeinen kommen. Dabei ist ebenfalls zu sagen, dass dieses Programm sehr flexibel ist und daher der Nutzen den Aufwand bei Weitem übertreffen. Denn es handelt sich hier um eine allgemeine Lösung für verschiedenste Probleme, welche, je nach Konfigurierung, sehr zuverlässig ist. Zusätzlich ist durch die Natur der KNNs zu sagen, dass durch die Generalisierung von bereits gelernten Daten für eine höhere Fehlertoleranz in den Testdaten sorgt. Natürlich ist es dadurch auch möglich, dass die KNNs selbst Fehler produzieren, dennoch ist die Effizienz dieser Technik schwer zu übertreffen. Die Möglichkeit, dass ein Computer nicht nur fest programmierten Regeln folgen, sondern auch ganze Verhaltensmuster erlernen und übernehmen kann, öffnet Möglichkeiten, deren Grenzen schwer vorstellbar erscheinen.

Quellenverzeichnis:

Q1: https://en.wikipedia.org/wiki/Neuroevolution

Q2: Edmund Ronald, Marc Schoenauer. Genetic lander: An experiment in accurate neuro-genetic control.Proc. PPSN III, Oct 1994, Jérusalem, France. pp.452 - 461, 10.1007/3-540-58484-6_288. hal-01079614

Q3: „Evolutionäre Algorithmen werden vorrangig zur Optimierung oder Suche eingesetzt.“ Zitatquelle: https://de.wikipedia.org/wiki/Evolutionärer_Algorithmus

Q4: https://de.wikipedia.org/wiki/Pong

Q5: https://de.wikipedia.org/wiki/Flappy_Bird

Q6: https://github.com/CodingTrain/Toy-Neural-Network-JS/blob/master/lib/matrix.js

Q7: „Markl Biologie, Oberstufe“, Prof. Dr. Jürgen Markl, Ernst Klett Verlag, 1. Auflage, 2010

Videoquellenverzeichnis:

V1: https://www.youtube.com/playlist?list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh

V2: https://www.youtube.com/playlist?list=PLRqwX-V7Uu6Yd3975YwxrR0x40XGJ_KGO