Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Projektbericht

Gaming AI

vorgelegt von

Friedrich Braun
Valentin Krön

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Wirtschaftsinformatik	Informatik
Matrikelnummer:	6252218	6700970
Betreuer:	Eugen Betke	Julian Kunkel

Hamburg, 2017-01-01

Inhaltsverzeichnis

Konventionelle AI's in Spielen haben jeweils einen konstanten Schwierigkeitsgrad. Nicht nur können erfahrene Spieler diese AI's überlisten, viele Spieler erreichen irgendwann eine Spielstärke, die über jeder AI liegt. In der Konsequenz wird jede AI auf Dauer langweilig. Eine neue, lernfähige AI könnte den Spielspaß steigern und die Qualität der Spiele erhöhen.
Dieser Text stellt den Bericht zu einem Projekt, das genau dieses Problem angeht, dar. Innerhalb des Projektes bauten wir ein sehr vereinfachtes RTS und eine AI, die erlernen sollte dieses gut zu spielen. Der AI liegt ein neuronales Netz zugrunde; das Training erfolgte über einen genetischen Algorithmus.
Diese Arbeit macht die ersten Schritte in die Entwicklung einer anpassungsfähigen AI und untersucht 3 Bewertungsmethoden des Lernfortschritts der AI's.

1 Einleitung

In diesem Kapitel gehen wir auf das Projektziel näher ein und erläutern warum wir das Projektziel ändern mussten.

Zurzeit sind die meisten AI für beliebige Computerspiele noch fest einprogrammierte Skripte. Diese AI's können somit nicht lernen und verlieren damit auf Dauer gegen menschliche Spieler. Um dies aufzufangen, also dem Menschen immer eine Herausforderung bieten zu können, werden die stärksten AI's mit unfairen Vorteilen versehen. Unser Projekt zielte nun auf einen anderen Ansatz: Die AI soll lernfähig sein und dadurch dem Menschen überlegen bleiben.

1.1 Ursprüngliches Projektziel

Unser ursprüngliches Projektziel bestand darin eine AI zu entwickeln und diese mit Hilfe eines genetischen Algorithmus zu trainieren. Diese AI sollte auf die Spring-RTS-Engine [1] angewandt werden. Die Absicht bestand darin zu zeigen, dass eine AI auf diese Weise lernen kann RTS-Spiele gut, eventuell sogar besser als menschliche Spieler, zu spielen.

1.2 Gründe Für die Änderung des Projektziels

Das Projektziel wurde im Laufe des Projekts aus den folgenden Gründen angepasst:

1. Einer der Gründe lag im unübersichtlichen Aufbau der Spring-Engine [1] selbst. Zwar bietet die Spring-Engine eine Lobby an, jedoch lassen sich nur wenige Spiele tatsächlich starten und selbst bei diesen lassen sich keine externen AI's einbinden.

2. Dadurch fehlte uns eine Vorlage, auf der wir unsere AI hätten aufbauen können; insbesondere fehlte uns Wissen zu den Schnittstellen zwischen unserer AI und der Spring-Engine [1]. Einige Spiele haben im Code für das Spiel direkt eingebundene AI's, die auch lauffähig sind. Diese AI's halfen uns jedoch auch nicht weiter, gerade da sie direkt ins Spiel hineinkompiliert sind. Dies bedeutet nämlich nicht nur, dass die AI selber kaum zu Debuggen ist, man muss auch direkt am Spiel selber mit der Sprache lua arbeiten.

Da die Auseinandersetzung mit dem Aufbau eines Spieles für die Spring-Engine und mit der Sprache lua den Projektrahmen gesprengt hätte, entschieden wir uns für eine Änderung der Zielsetzung.

1.3 Neues Projektziel

Prinzipiell ist es irrelevant im welchen Spiel man die AI entwickelt. Spring hatte sich lediglich angeboten, weil es ein OpenSource Projekt ist und wir die Engine um unseren Code einfach hätten erweitern dürfen. Die Komplexität des Projekt macht es dennoch schwierig es umzusetzen. Deshalb haben wir haben das Konzept, das wir umsetzen wollten, aus dem ursprünglichen Ziel extrahiert und auf eine andere Weise umgesetzt, d.h. statt es in Spring umzusetzen haben wir beschlossen unser eigenes, vereinfachtes, Spiel zu bauen und in diesem das Training durchzuführen.

2 Grundlagen

Dieses Kapitel beschreibt die Entwicklungsumgebung Ünityünd die in der Arbeit verwendeten Algorithmen.

2.1 Unity

Unity [3] ist eine Spieleentwicklungsumgebung, die für nicht kommerzielle Zwecke, wie z.B. Forschung und Lehre, kostenlos nutzbar ist. Unity setzt kein großes Vorwissen voraus und bietet, wegen großer Bekanntheit, auch viele Onlinetutorials.
Das wichtigste in Unity sind GameObjects. Aus diesen wird die ganze Spielwelt aufgebaut. Während Unity viele Komponenten für die GameObjects vorgefertigt bereitstellt, decken diese jedoch nicht jeden Spezialfall ab. Hierzu kann man eigene Skripte in C# oder Javascpript, bzw. mitlerweile Unityscript, schreiben.

2.2 Neuronale Netze

Ein Neuronales Netz besteht aus mehreren Schichten sogenannter Neuronen. Unter Neuron versteht man ein Element, dass mehrere Inputwerte bekommt, diese, meistens durch Summation, akkumuliert und dann entweder ein Signal weitersendet oder gerade nicht. Dabei ergibt sich aus den weitergesendeten Signalen einer Schicht der Input für die nächste Schicht, wobei der Input der ersten Schicht von außerhalb kommt und der Output der letzten Schicht die Ausgabe des Netzes darstellt. Alle Verbindungen zu Neuronen sind hierbei mit einem Gewicht belegt, das einen Faktor darstellt.
Meistens wird ein Neuronales Netz als Klassifikator eingesetzt. Trainiert wird dann mit Trainingsdaten, deren Klasse bekannt ist. Man lässt das Netz diese Daten neu klassifizieren und rechnet, bei falscher Klassifikation, den Fehler zurück.
Neuronale Netze bieten aber mehr Anwendungsmöglichkeiten, bei denen sich für einen bestimmten Input kein gewünschter Output angeben lässt; in diesem Fall sind andere Trainingsmethoden erforderlich.

2.3 Genetischer Algorithmus

Ein genetischer Algorithmus ist ein Optimierungsalgorithmus für Probleme bei denen mehrere Werte zusammenspielen. Die Vorgehensweise basiert auf biologischen Genen und Chromosomen (daher der Name). Natürliche Chromosome setzen sich aus vielen Genen zusammen, wobei sich erst im Zusammenspiel der Gene die Wirkung ergibt. Pflanzen sich natürliche Lebewesen fort, so geben sie ihre Gene weiter. Bei höheren Lebewesen werden dabei Gene von zwei Elternteilen neu kombiniert. Hierbei gilt, dass je fitter ein Lebewesen ist, desto wahrscheinlicher pflanzt es sich fort. Diese Vererbung passiert nicht fehlerfrei, es können sogenannte Mutationen, spontane Veränderungen einzelner Gene, auftreten.
Für den Anwendungsfall eines genetischen Algorithmus werden diese Prinzipien nun vereinfacht umgesetzt. Es gibt eine Menge sogenannter Individuen, die jeweils einen Lösungskandidaten für das Problem darstellen. Hierzu beinhalten sie ein Set von Genen in Form eines Chromosoms. Die Gene geben Belegungen von den Problemvariablen an. Die Güte jedes Individuums, also wie gut es das Problem löst, wird durch einen sogenannten Fitnesswert ausgedrückt.
Die Optimierung findet nun dadurch statt, dass neue Individuen geschaffen werden. Ein neues Individuum entsteht durch die Rekombination der Chromosomen zweier Elternteile, ähnlich wie bei der natürlichen Fortpflanzung. Hierbei kann auch Mutation auftreten. Die Wahl der Eltern hängt von dem Fitnesswert ab. Ein Vererbungsschritt, also das Erzeugen einer festen Menge neuer Individuen, wird als eine Generation bezeichnet. Nach dem Erzeugen neuer Individuen wird die Menge aller Individuen wieder auf einen feste Größe reduziert. Die genaue Umsetzung jedes diese Schritte hängt von der genauen Zielsetzung ab.

3 Aufbau

In diesem Kapitel beschreiben wir kurz den Aufbau der wichtigsten Bausteine des Projekts.

3.1 Aufbau der Simulation

Die Simulation ist ein sehr einfaches RTS. Sie ist ausgelegt auf genau zwei Spieler. Jeder der Spieler hat ein Hauptgebäude (repräsentiert durch große, farbige Quadrate in den Spielfeldecken), dass in regelmäßigen Zeitabständen Einheiten (kleine, farbige Kapseln) erstellt (Abbildung 3.1).

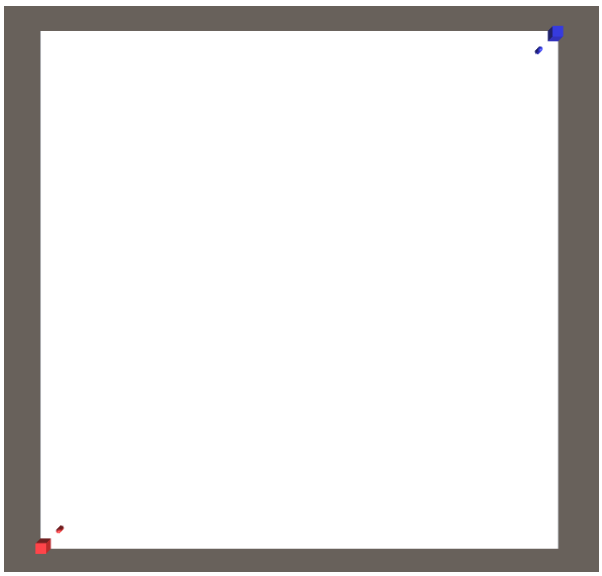


Abbildung 3.1: Ein Screenshot vom Spielfeld.

Die Einheiten können zu Positionen auf dem Spielfeld laufen. Übergeben wird dabei nur die Zielposition, die eigentliche Bewegung übernimmt Unity.
Sind Einheiten dicht genug an gegnerischen dran, also innerhalb eines vordefinierten Radius (r_{kill}), so können sie Schaden an diesen verursachen. Wenn Schaden verursacht wurde, so benötigt die Einheit eine vordefinierte Menge Zeit, bis sie wieder Schaden verursachen kann, andere Funktionen sind dadurch nicht eingeschränkt. Wichtig ist hierbei, dass beliebig viele Einheiten gleichzeitig angreifen können, genauer alle Einheiten, die zu einem Zeitpunkt angreifen können und wollen greifen gleichzeitig an. Dadurch

ist es möglich, dass sich zwei Einheiten gegenseitig töten. Fallen die Lebenspunkte einer Einheit auf oder unter null, so wird diese gelöscht.
Die Einheiten haben eine Sichtweite (r_{view}) in der sie sowohl die gegnerische als auch freundliche Einheiten sehen können.
Es gibt genau einen Einheitentyp. Die Simulation beinhaltet keinerlei Mechaniken, die Ressourcen in irgendeiner Form verwenden.

Da wir kein Spielende durch Sieg eines Spielers garantieren konnten, beschlossen wir jedem Match eine feste Zeitdauer zu geben. Diese musste lang genug für das Entfalten einer Taktik sein ohne dabei das Match unnötig in die Länge zu ziehen. Als Kompromiss wählten wir zehn Minuten pro Match.

3.2 Aufbau der AI

Da bei einem RTS die Anzahl der Einheiten, und damit die Anzahl der Möglichkeiten, variabel ist, wurde die Entscheidung getroffen die AI als Schwarmintelligenz zu designen. Jede Einheit hat einen Sichtradius und gibt diese Information als Input an die AI. Diese wiederum generiert daraus über ein neuronales Netz eine Entscheidung. Damit verhalten sich alle Einheiten gleich bei gleicher Eingabe. Durch den Aufbau als Schwarmintelligenz ist die AI problemlos skalierbar. Zudem benötigen wir keinerlei Koordination der Einheiten, da diese sich aus der Schwarmintelligenz von alleine ergibt.

3.3 Aufbau des Trainingsalgorithmus

Das Training der AI erfolgt dadurch, dass das neuronale Netz trainiert wird. Im Detail heißt das: Wir verwenden die Gewichte des neuronalen Netzes als Gene in einem genetischen Algorithmus. Dabei müssen wir nicht zwischen Geno- und Phänotyp unterscheiden. Die Rekombination erfolgt über einen zufälligen Schnitt. Die Mutation findet mit einer gewissen Wahrscheinlichkeit statt, wenn sie stattfindet so wird genau ein Gen durch ein zufälliges neues ersetzt.

Der Fitnesswert basiert auf der Anzahl lebender Einheiten, eigener wie gegnerischer, bei Spielende. Die genaue Umsetzung haben wir zu Auswertungszwecken variiert, weswegen wir in der Auswertung auch noch einmal genauer darauf eingehen.

4 Umsetzung

In diesem Kapitel beschreiben wir kurz die Umsetzung der wichtigsten Bausteine des Projekts.

4.1 Umsetzung der Simulation

Die Simulation entwickelten wir in Unity [3], die Skripte in C# .
Im Folgenden werden wir viele der Werte und unsere Entscheidung dazu etwas genauer erläutern.
Die Einheiten haben 100 Lebenspunkte, ein Angriff verursacht 25 Schaden und nach einem Angriff muss 10 Sekunden gewartet werden. Dieses 1-zu-4-Verhältnis erschien uns ein guter Kompromiss daraus, dass Einheiten weder zu schnell noch zu langsam sterben sollten. Das Hauptgebäude sollte deutlich schwieriger zu töten sein und hat deswegen 1000 Lebenspunkte.
Das Spielfeld hat eine Größe von 500 mal 500. Die Angriffsreichweite (r_{kill}) beträgt 50. Dies ist wieder ein Kompromiss dazwischen, dass die Einheiten sich finden müssen sollen und auch finden können. Aus der gleichen Überlegung heraus wählten wir einen Sichtradius (r_{view}) von 100, also das Doppelte des Angriffsradius. Wichtig war uns beim Sichtradius, dass Einheiten sich sehen können bevor sie sich angreifen können.
Das Hauptgebäude erzeugt jede Minute eine Einheit, die erste davon bei Spielbeginn. Dies soll eine kontinuierliche Produktion von Einheiten simulieren, da wir dazu keine Mechanik in das Spiel einbauen wollten. Deswegen dauert ein Match genau genommen auch 10 Minuten und 2 Sekunden, damit die letzte (11.) Einheit auch auf jeden Fall erzeugt wird. Daraus folgt natürlich, dass nur Einheiten erzeugt werden solange das Hauptgebäude besteht, also keine Einheiten mehr erzeugt werden wenn das Hauptgebäude zerstört ist.

4.2 Umsetzung der AI

Wie bereits im Aufbau erwähnt, ist der Kern der AI ein neuronales Netz. Dazu suchten wir uns eine passende C#-Bibliothek [2], die uns die notwendige Funktionalität bereitstellt.

Eine große Hürde bei der Verwendung eines neuronalen Netzes bestand darin, dass das neuronale Netz als Input Floats erwartet und den Output auch in Form von Floats liefert, was uns dazu zwingt den Input auf Floats herunter zu brechen. Die Umsetzung in der von uns verwendeten Bibliothek verwendet den Wertebereich null bis eins.

Als Input verwenden wir die eigene Position der Einheit (pos_x, pos_y) auf der Ebene, die wir einfach linear reskalieren. Dazu kommen die Einheiten, inklusive der Hauptgebäude, im Sichtradius der Einheit, sowohl freundliche als auch feindliche, deren genaue Zuordnung wir im Folgenden aufschlüsseln: Je ein Inputneuron verwenden wir um zu signalisieren, ob das jeweilige Hauptgebäude (Freund/Feind) zu sehen ist (found_friendly_building, found_enemy_building). Ist dies der Fall, so gibt ein weiteres Neuron eine ungefähre Entfernungsabschätzung an (dist_to_friendly_building, dist_to_enemy_building). Zusätzlich verwenden wir noch je ein Neuron um die Anzahl der Freunde (num_of_friends) bzw. Feinde (num_of_enemies) im Sichtbereich zu codieren; dies geschieht indem wir diese Anzahl durch die maximal mögliche Anzahl teilen. Insgesamt kommen wir somit auf acht Inputneuronen.

$$\text{Input} = \begin{pmatrix} \text{pos\_x} \\ \text{pos\_y} \\ \text{num\_of\_friends} \\ \text{num\_of\_enemies} \\ \text{found\_friendly\_building} \\ \text{found\_enemy\_building} \\ \text{dist\_to\_friendly\_building} \\ \text{dist\_to\_enemy\_building} \end{pmatrix} \tag{4.1}$$

Um die AI sinnvoll einsetzen zu können, brachen wir zuerst einmal die Entscheidungsmöglichkeiten des Menschen bzw. der AI auf zwei Funktionen herunter:

- SetDestination: Bekommt die Koordinaten eines Punktes in der Ebene und setzt die Zielposition der Einheit auf diesen Punkt. Die Einheit läuft dann zum Zielpunkt.
- SetTarget: Bekommt eine andere Einheit oder ein Hauptgebäude als "target". Solange dieses "target" existiert, läuft die Einheit darauf zu und, sollte es feindlich sein, greift es an.

$$\text{Output} = \begin{pmatrix} < \text{SetDestination}() \mid \text{SetTarget}() > \\ \text{target\_pos\_x} \\ \text{target\_pos\_y} \\ \text{friend} \mid \text{enemy} \\ \text{target\_building} \\ \text{target} \end{pmatrix} \tag{4.2}$$

Daraus leitet sich der Output ab. Das erste Outputneuron entscheidet darüber welche der beiden Funktionen, SetDestination oder SetTarget, ausgeführt wird. Zwei Neuronen bestimmen dann die Position des Zielpunktes; es erfolgt wieder eine lineare Reskalierung. Drei weitere Neuronen dienen dann zur Bestimmung des "target". Das erste davon legt fest, ob ein Freund oder Feind zum "target" wird, das zweite legt fest ob das jeweilige Hauptgebäude zum "target" wird und das dritte legt fest welche Einheit zum "target" wird, wenn es nicht das Hauptgebäude ist. Dabei fangen wir natürlich die Fälle ab, bei

denen sich ein ”target” ergeben würde, das nicht existiert. Somit kommen wir auf sechs Outputneuronen.

Wir entschieden uns mit einem Hidden-Layer zu experimentieren. Die Entscheidung zehn Neuronen in diesem Layer einzubauen trafen wir, weil aus unserem Erfahrungsschatz hervorgeht, dass man die Größenordnung der Layer zu einander nicht variiert.

Dadurch kommen wir auf 140 Gewichte $(8 \cdot 10 + 10 \cdot 6)$.

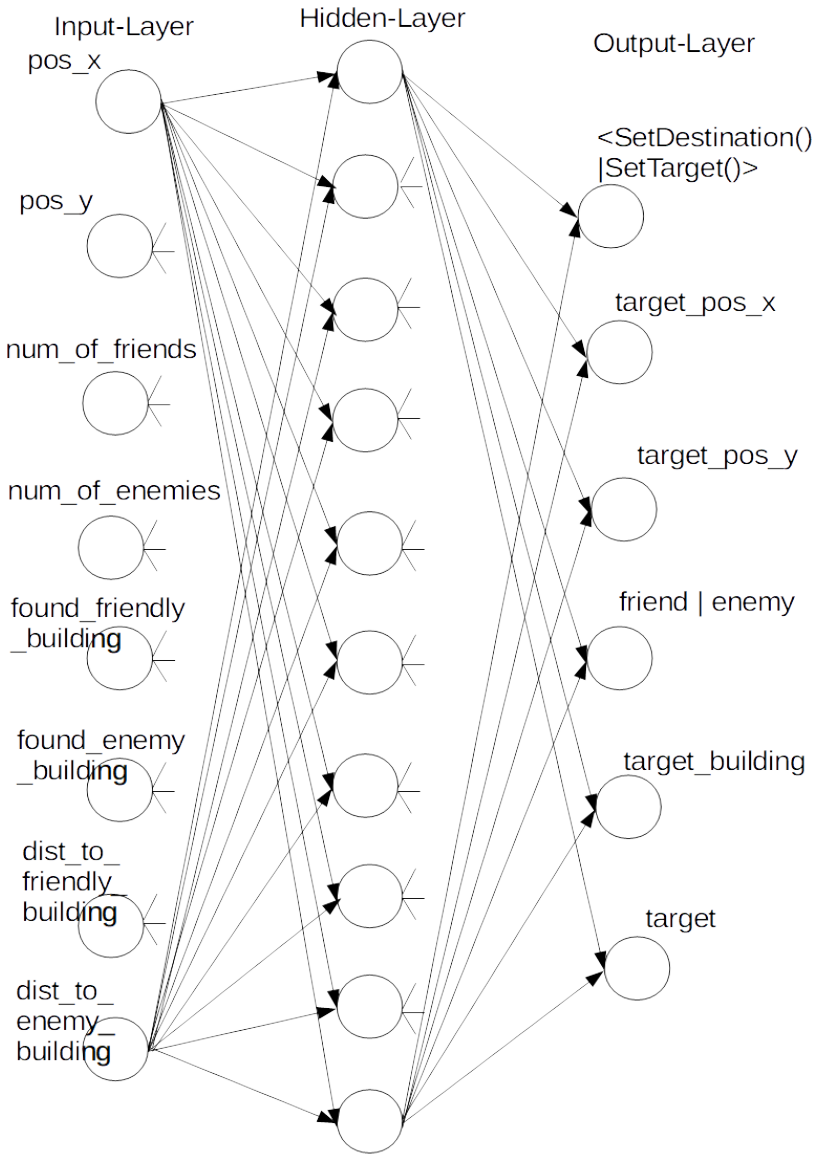


Abbildung 4.1: Eine Skizze des neuronalen Netzes

Die von uns verwendeten Gewichte liegen im Wertebereich von minus eins bis eins.

4.3 Umsetzung des Trainingsalgorithmus

Das Training des neuronalen Netzes wird von einem genetischen Algorithmus durchgeführt. Dabei beinhalten die Gene jeweils genau einen Gewichtswert. Da die Wertebereiche der Gewichte paarweise voneinander unabhängig sind, benötigen wir keine Kodierungsfunktion und laufen auch nicht Gefahr bei der Rekombination oder Mutation ungültige Individuen zu erzeugen. Bei der Erzeugung eines neuen Gens erzeugt dieses in seinem Konstruktor einen zufälligen, legalen Wert. Ein Chromosom beinhaltet nun eine Liste von Genen. Dies bedeutet explizit, dass wir keine weiteren Informationen im Chromosom speichern. Das Individuum wiederum verbindet ein Chromosom mit einem Fitnesswert. Verwaltet werden die Individuen von der Klasse PopulationManager.

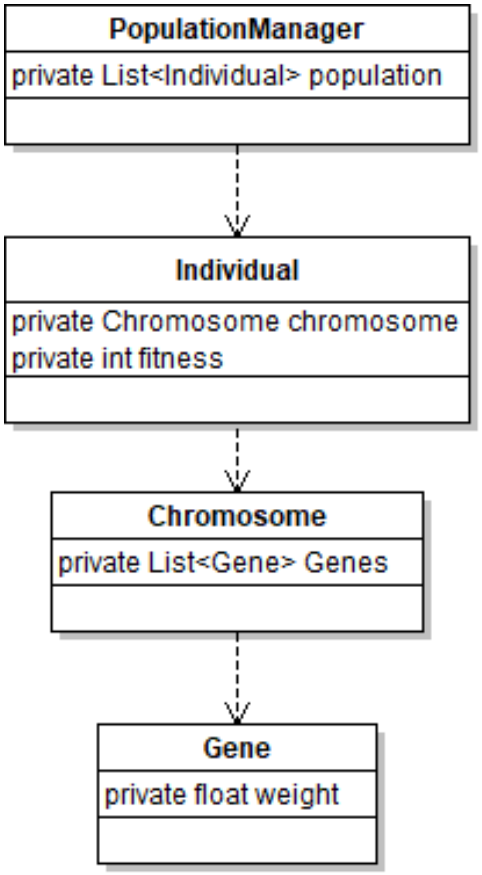


Abbildung 4.2: Ein vereinfachtes UML-Diagramm der beteiligten Klassen

Diese Klasse stellt alle wichtigen Funktionen bereit und bildet die Schnittstelle zur Außenwelt.

Der Fitnesswert ergibt sich daraus, dass jedes Individuum der aktuellen Population gegen alle anderen Individuen der selben Population genau ein Match spielt, woraufhin dann über alle Spielergebnisse summiert wird (Gleichung (4.3)). Ein Spielergebnis berechnet sich aus den eigenen und gegnerischen Einheiten bei Spielende. Da die genaue Formel für die Auswertung variiert wurde, werden wir an der Stelle darauf noch genauer

eingehen. Um sicher zu stellen, dass wir nur positive Fitnesswerte erhalten, verwenden wir ein Offset.

$$\text{Fitnesswert} = \sum_{i=0}^{n} \text{Spielergebnis}_i \tag{4.3}$$

Die Rekombination verwendet den Fitnesswert zur Auswahl der Eltern. Die Auswahl der Eltern erfolgt zufallsbasiert, wobei die Auswahlwahrscheinlichkeit eines Individuums proportional zu seinem Fitnesswert ist. Hierbei vermeiden wir explizit, dass ein Individuum mit sich selbst rekombiniert wird. Die eigentliche Rekombination erfolgt nun über einen zufälligen Schnitt, dies bedeutet, dass wir eine zufällige Position auswählen, wobei bis zu dieser Position die Gene des ersten Elternteils und nach dieser Position die Gene des zweiten Elternteils übernommen werden. Dabei wird sichergestellt, dass von jedem Elternteil mindestens ein Gen übernommen wird.

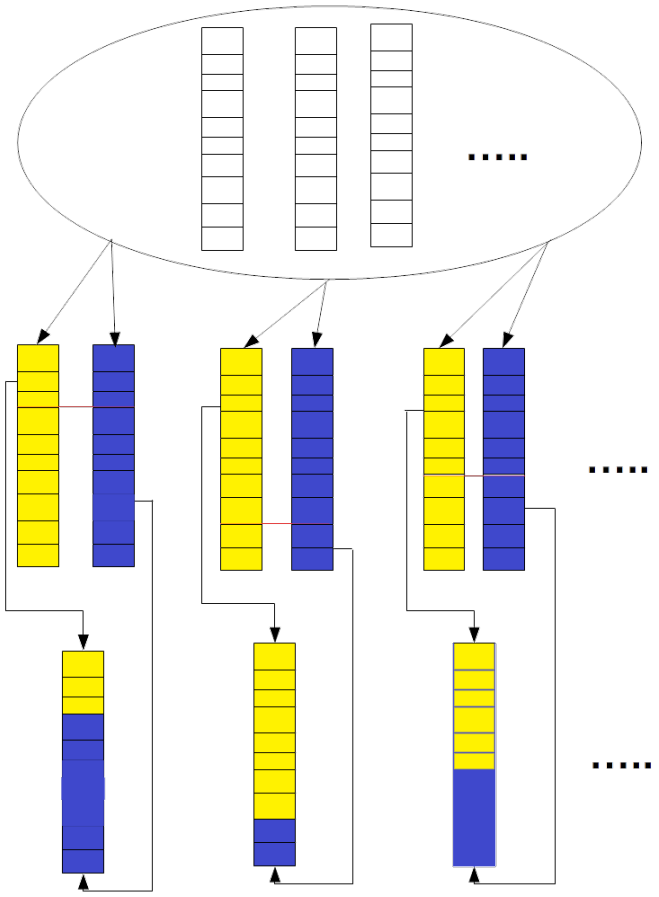


Abbildung 4.3: Eine Skizze des Rekombinationsvorgangs

Diese neu entstandenen Gensätze werden dann jeweils mit einer gewissen Wahrscheinlichkeit mutiert. Erfolgt die Mutation, so wird ein zufällig ausgewähltes Gen durch

ein neues Gen ersetzt. Danach werden die neuen Gensätze dann in Chromosomen und Individuen verpackt.

Nach Rekombination und Mutation haben wir somit neue Individuen in der Population. Diese neue Population wird nun wieder bezüglich der Fitness ausgewertet. Nach dieser Auswertung selektieren wir die fittesten Individuen. Dies ist darüber realisiert, dass wir die Individuen absteigend nach ihrem Fitnesswert sortieren und dann nur die vordersten Individuen übernehmen, sprich die hinteren Individuen löschen. Damit sind wir dann wieder bei der gleichen Populationsgröße wie vor der Rekombination.

Startet man nun das Trainingsprogramm, so wird ein neuer PopulationManager erzeugt, der wiederum seine Startpopulation von sechzehn Individuen zufällig erzeugt. Dabei ist zu beachten, dass diese Individuen zuerst bezüglich der Fitness ausgewertet werden, bevor der Algorithmus mit ihnen arbeitet.

Jeweils vor der Erzeugung neuer Individuen wird die aktuelle Population in Textdateien rausgeschrieben. Danach werden 32 neue Individuen, wie oben beschrieben, erzeugt. Damit sind wir bis zur Selektion bei einer Populationsgröße von 48 Individuen. Dann erfolgen die Auswertung nach dem Fitnesswert sowie die Selektion.

Zu beachten ist hierbei, dass von außen nur die Matches aufgerufen werden, will heißen: Das Ergebnis des letzten Matches wird an den PopulationManager übergeben und die Gewichten des nächsten Matches werden ausgelesen. Der PopulationManager achtet dabei selber auf die korrekte Zuordnung der Matches, sowie darauf, dass nach spielen aller Matches einer Generation der Generationschritt vollzogen wird.

5 Auswertung

Für die Evaluation standen wir vor einem großen Problem: Die Fitness-Funktion bezieht sich nur auf die direkte Konkurrenz und hat keinerlei Abhängigkeit zur einer objektiven, statischen Größe. Somit ist ein Ansteigen der Fitnessfunktion kein eindeutiges Zeichen für einen Fortschritt, was uns dazu zwingt eine andere Form der Evaluierung anzuwenden. Deswegen ließen wir das jeweils beste Individuum jeder Generation (null bis zehn) gegen jedes Individuum der nullten Generation antreten, um dann diese Matches nach Kills und Verlusten auszuwerten.

Insgesammt testeten wir drei Varianten der Fitnessfunktion, um einen Eindruck davon zu erhalten, welche sich am besten eignet. Wie erwartet bevorzugen die drei Varianten jeweils ein unterschiedliches Verhalten. Im Folgenden werden nun die drei Varianten vorgestellt und ausgewertet. Die Namen der Varianten ergeben sich daraus wie viel eigene und gegnerische Einheiten Wert sind (Wert_eigene zu Wert_gegnerische).

5.1 Variante 1: 1zu1

Diese Variante bewertet verlorene eigene Einheiten genauso schwer wie verwehrte [1] gegnerische Einheiten. Die Motivation dieser Variante war, dass dies uns als natürlichste Bewertung einer KI in diesem Spiel erschien; es wird weder Offensive noch Defensive bevorzugt und sorgt für einen ausgeglichenen Spielstil.

Abbildung 5.1 zeigt die durchschnittlichen Kills des jeweiligen Individuums gegen die nullte Generation. Dabei ist die x-Achse die Generation, aus der das jeweilige Individuum stammt, und die y-Achse der Durchschnittswert, der gegen die nullte Generation erzielten Kills. Die Punkte sind dabei die Einzelwerte, während die Gerade eine lineare Regression aller Werte ist, also den Trend anzeigt.

Die Stagnation von der nullten bis zur zweiten Generation liegt höchstwahrscheinlich daran, dass das Abschießen von Gegnereinheiten erst durch eine Mutation erlernt werden musste. Ab der dritten Generation zeichnet sich ein Aufwärtstrend ab. Die Einbrüche innerhalb dessen sind der Fitnessfunktion geschuldet: Ein Individuum kann innerhalb einer Population besser sein als in einer anderen; schließlich muss man sich begegnen, um sich bekämpfen zu können. Trotzdem gehen wir davon aus, dass sich auf längere Sicht, sprich mehr Generationen, der Aufwärtstrend fortsetzen würde.

Wir sind bisher nicht auf Verluste eingegangen, da in dieser Variante die nullte Generation keinem Individuum Verluste zufügen konnte.

[1] Es ist möglich das Hauptgebäude zu zerstören, wodurch keine neuen Einheiten entstehen. Gezählt werden die Einheiten bei Spielende, anstatt der getöteten bzw. verlorenen Einheiten, wodurch dieser Effekt den Punktestand beeinflusst.

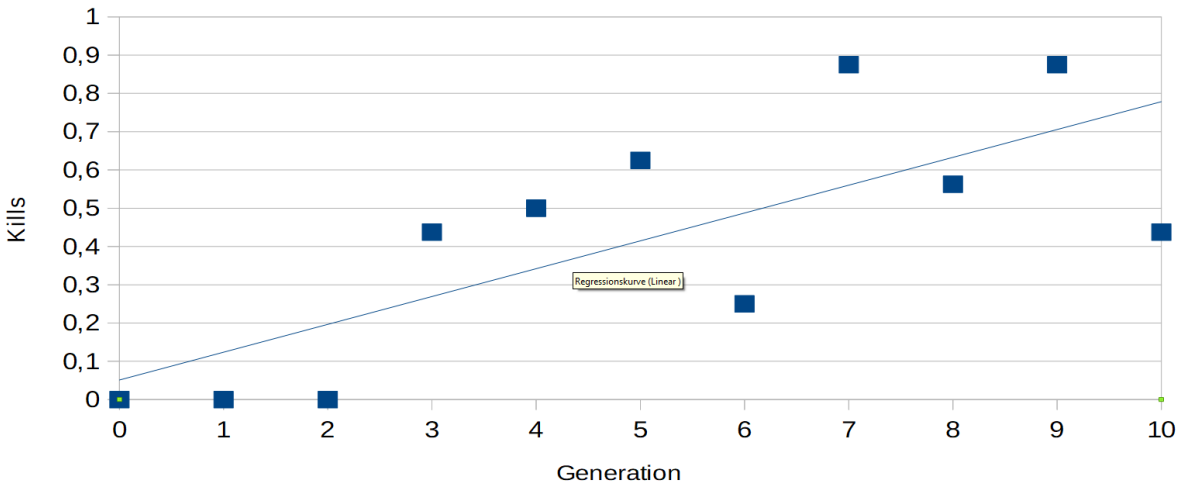


Abbildung 5.1: Durchschnittskills1zu1

5.2 Variante 2: 1zu2

Diese Variante bewertet verwehrte gegnerische Einheiten doppelt so schwer wie verlorene eigene Einheiten. Die Motivation dieser Variante war einen aggressiveren Spielstil zu bevorzugen, damit aktive Strategien extrinsisch motiviert werden.

Die Abbildung 5.2a zeigt die durchschnittlichen Kills des jeweiligen Individuums gegen die nullte Generation, die Abbildung 5.2b zeigt die durchschnittlichen Verluste.

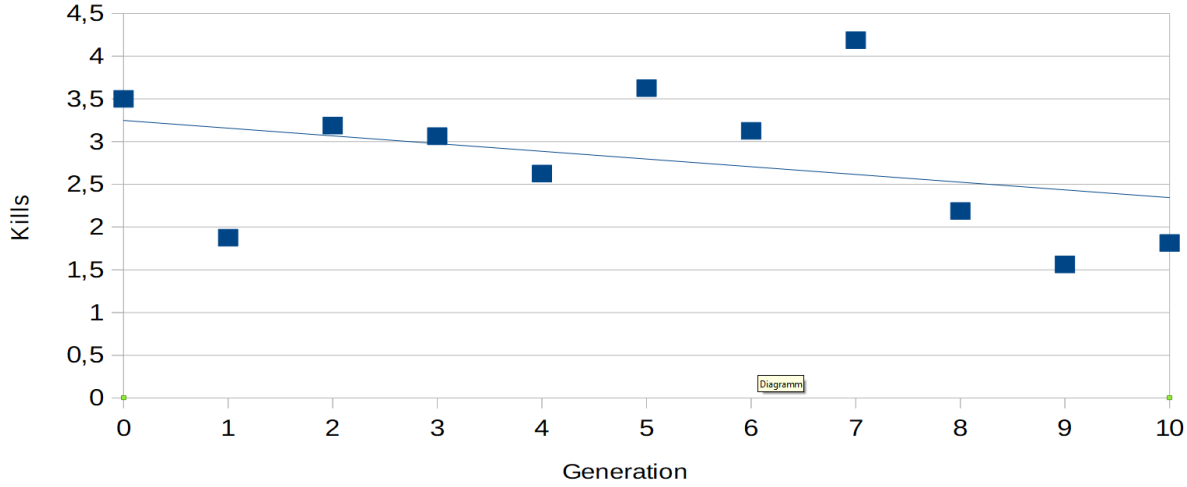
Beide Graphiken haben gemeinsam, dass sich kein klarer Gesamttrend herauslesen lässt. Allerdings ist klar zu erkennen, dass bei dieser Variante die nullte Generation zufälligerweise wesentlich aggressiver als andere nullte Generationen ist. Diese Aggressivität bleibt bis zur siebten Generation deutlich sichtbar. Wahrscheinlich ringt diese Variante um eine Balance zwischen aggressiver Vernichtung des Gegners und Minimierung eigener Verluste. Dies hängt natürlich eng mit der generell aggressiven Grundstimmung der Population zusammen. Da Kills, in dieser Variante, mehr zählen als Verluste geht diese aggressive Grundstimmung auch nicht verloren. Ab der achten Generation scheint die Vermeidung von Verlusten überhand über das Erzielen von Kills zu gewinnen, wodurch sich die Aggressivität senkt. Dies hat zur Folge, dass zwar die Kills zurückgehen jedoch die Verluste deutlich stärker abnehmen.

5.3 Variante 3: 0zu1

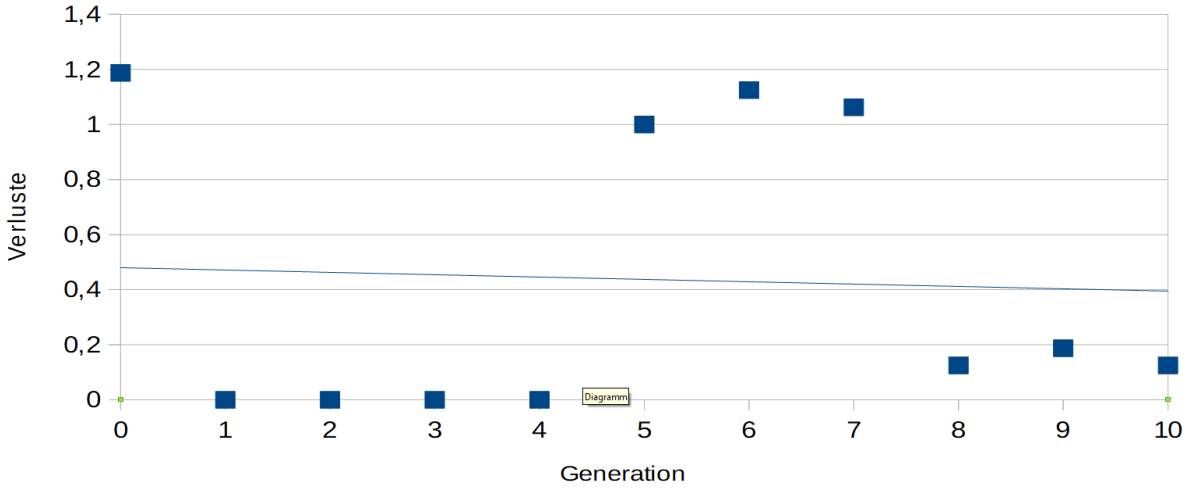
Diese Variante bewertet nur verwehrte gegnerische Einheiten und ignoriert verlorene eigene Einheiten. Die Motivation dieser Variante war einen aggressiven Spielstil zu forcieren, um damit eine hohe Aktivität zu schaffen.

Die obere Graphik zeigt die durchschnittlichen Kills des jeweiligen Individuums gegen die nullte Generation. Die Achsen entsprechen der ersten Abbildung (5.1).

Wieder haben wir keine Grafik zu Verlusten, weil es keine gab.



(a) Durchschnittskills1zu2



(b) Durchschnittsverluste1zu2

Abbildung 5.2: Trainingsfortschritt

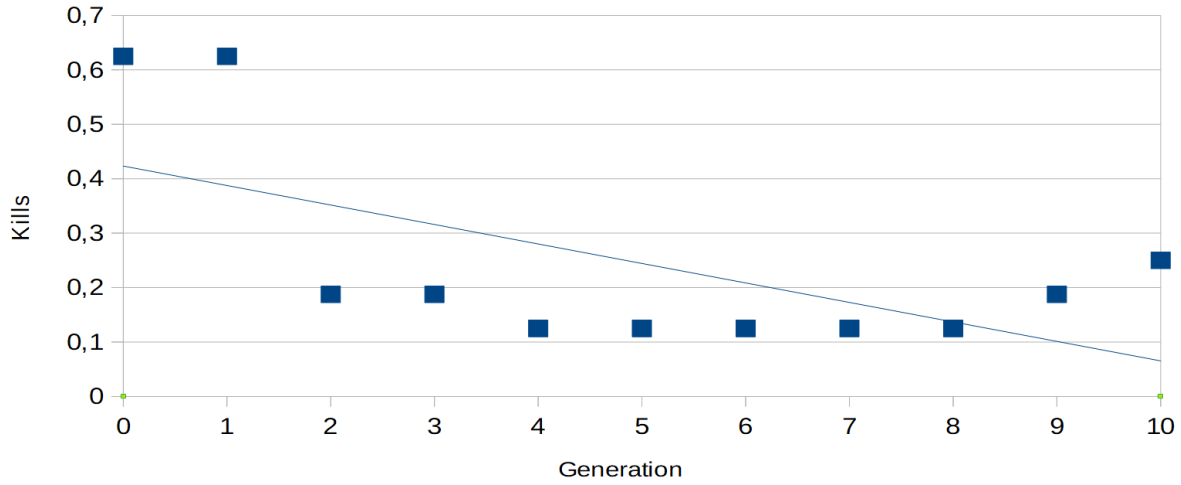


Abbildung 5.3: Durchschnittskills0zu1

Das erste, was an dieser Grafik auffällt, ist der starke Abfall nach der ersten Generation. Dies ist ein Indikator dafür, dass sich zuerst ein passiver Spielstil durchsetzen konnte. Da für uns auch am Fitnesswert nicht abzusehen ist, ob dies tatsächlich zur Vermeidung von Verlusten geführt hat, ist diese Entwicklung schwer einzuschätzen. Jedenfalls sind passive Individuen auch schwieriger anzugreifen.

Ab der neunten Generation setzt dann ein Aufwärtstrend ein. Bei dieser Variante ist klar zu erkennen, dass ohne Abstrafung von Verlusten sich passive Individuen unerwartet gut in der Population halten können.

5.4 Fazit

Während sich bei der 1zu1-Variante eine stetige Verbesserung abzuzeichnen scheint, ist dies bei den anderen Varianten nicht der Fall. An dieser Stelle fehlen leider weitere Trainingsdaten, also mehr Generationen und Wiederholungen, die auf Grund des Erstellungsaufwands nicht zu bewerkstelligen waren. Zudem ist die Generationsgröße eher klein. Trotzdem ist ein gewisser Trainingserfolg zu erkennen. Bei 1zu1 steigt die Zahl der durchschnittlichen Kills, bei 1zu2 balancieren sich Kills und Verluste über die Generationen aus.

Die 1zu1-Variante zeigte, trotz schlechtem Start, gute Fortschritte. Sie entspricht auch unserem intuitiven Verständnis von einer guten Strategie. Somit empfehlen wir, dass bei Weiterführung der Arbeit diese Variante weiter fortgesetzt wird.

Die 2zu1-Variante bringt eine gewisse Diversität mit. Bei einer Weiterführung der Arbeit empfehlen wir diese Variante zumindest als Vergleich zur 1zu1-Variante weiter zu verfolgen.

Die 0zu1-Variante konnte keine nennenswerten Ergebnisse erzielen. Dies könnte sich aber mit weiterem Training ändern. Daher empfehlen wir bei Weiterführung der Arbeit diese Variante zu Vergleichszwecken, zumindest vorerst, beizubehalten.

Desweiteren empfehlen wir die Erweiterung des Spiels um etwas, dass die AI intrinsisch motiviert und um das sie mit dem Gegner kämpfen kann. Zudem könnte ein höherer Detailgrad beim Input für das neuronale Netz für bessere Einzelfallunterscheidungen und damit -entscheidungen sorgen.

Wir gehen, nach Auswertung unserer Tests, davon aus, dass sich eine schwarmintelligenzbasierte Gaming-AI für ein RTS durch einen genetischen Algorithmus trainieren und optimieren lässt. Sogar unsere, doch sehr schlichte, AI war in der Lage reaktives Verhalten zu erlernen. Damit könnte sie, unter der Voraussetzung, dass sie ausreichend trainiert wird, gegen jede statisch vorprogrammierte AI gewinnen. Zudem macht ja gerade der Aspekt der Schwarmintelligenz die AI skalierbar. Damit ist sie prinzipiell auch auf kompliziertere RTS übertragbar.
In Summe haben wir eine einfache AI für ein sehr einfaches Spiel geschaffen, die sowohl gemeinsam als auch nahezu unabhängig voneinander erweiterbar sind.

6 Wie geht es weiter?

Letztendlich war unsere Zeit doch sehr begrenzt für den Umfang, den wir dem Projekt gerne verliehen hätten. Wir verloren viel Zeit damit Grundlagen zu schaffen auf denen wir eigentlich hätten aufbauen wollen. Wir hätten sowohl mehr Tiefe im Spiel (z.B. Ressourcen) haben als auch die AI erweitern (z.B. detaillierterer Input) wollen. Besonders wichtig hierbei wäre eine Spielmechanik, die etwas beinhaltet, worum man kämpfen muss (z.B. Kontrollpunkte). Dies hätte der AI eine intrinsische Motivation gegeben aktiv zu spielen. Mit einer solchen intrinsischen Motivation wäre es auch leichter möglich gewesen die Testspiele zu reduzieren (statt jeder gegen jeden z.B. jeder gegen 5 zufällige) ohne dabei die Testdaten durch extremes Übergewicht des Zufalls zu korrumpieren.
Sollte man dieses Projekt erweitern bzw. fortsetzen, so empfehlen wir Testdaten zu generieren, während im Hintergrund bereits die Weiterentwicklung fortgesetzt wird. Damit würde nicht so viel Zeit für etwas verloren gehen, das eigentlich im Hintergrund laufen kann. Zudem würde es einem auch laufend Hinweise geben in welche Richtung weiterzuarbeiten ist.
Würden wir dieses Projekt nochmal von vorne angehen, so würden wir Spring schneller verwerfen. Dies würde uns mehr am Ende Zeit geben uns auf die AI zu konzentrieren. Eventuell würden wir auch kein lauffähiges Spiel mehr bauen sondern eine reine Simulation. Dies würde uns zwar mehr Zeit für die AI geben, jedoch nimmt es Möglichkeiten den Fortschritt nach zu vollziehen und den Spielumfang zu erweitern. Im Sinne der Fertigstellung des Projektes wäre das also sehr sinnvoll, in Hinblick auf eine Erweiterung oder Fortsetzung des Projektes durch andere wäre es sehr hinderlich.
An und für sich haben wir die Basis geschaffen, auf der man unser Projekt, im Sinne der ursprünglichen Intention, erst sinnvoll durchführen kann. Wir mussten eine Basis schaffen, um die AI auf irgendetwas anwenden zu können. Sollte jemand unsere Arbeit fortsetzen wollen, so empfehlen wir auf der von uns geschaffenen Basis (und dem Code) aufzusetzen und tatsächlich die Arbeit fortzuführen, statt sie nur mit unseren Erkenntnissen neu zu beginnen.

Appendices

Abbildungsverzeichnis

Literatur

[1] Spring Community. *Spring is a free RTS game engine*. 2018. URL: https://springrts.com (besucht am 27. 04. 2018).

[2] Franck Fleurey. *C# Neural network library*. 2018. URL: http://franck.fleurey.free.fr/NeuralNetwork (besucht am 27. 04. 2018).

[3] Unity Technologies. *Unity*. 2018. URL: https://unity3d.com (besucht am 27. 04. 2018).