Anwendung von Künstlicher Intelligenz

Am Beispiel von Q-Learning und Neuralen Netzen

Fin Hendrik Eckhoff
Osterweg 10
26419 Schortens
☏+49 4416/84249
✉fheckiostiem@gmail.com

Jonas Michael
Am Junkernberg 1
26419 Schortens
☏+49 4416/7488499
✉jomichael@web.de

Facharbeit im Seminar ‚Einführung in die Programmierung‘

Mariengymnasium Jever

Schuljahr 2018/2019
Jahrgangsstufe Q2
Jonas Michael & Fin Hendrik Eckhoff

Anwendung von KI C++ und C#
Am Beispiel von Q-Learning und Neuralen Netzen

Seminarfach: ‚Einführung in die Programmierung'
Kursnummer: Sem 6
Kurslehrer: Herr Hermann Olliges
Datum der Themenerstellung: 01.10.2018
Abgabedatum: 12.11.2018

1. EINLEITUNG

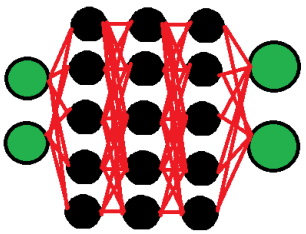
In den letzten Jahren wird die Bedeutung von Künstlicher Intelligenz immer größer. Überall hört man von KI, die schon die besten Menschen in verschiedenen Spielen geschlagen hat. Von KI, die uns bald alle ersetzen wird; und es werden sogar Stimmen laut, die behaupten, dass wir in ein paar Jahren nur noch Sklaven von KI sein werden. Um zu verstehen, weshalb Künstliche Intelligenz so erfolgreich ist und warum viele Menschen diese so fürchten, müssen wir ihr Prinzip und ihre Vorgehensweise verstehen.

2. KÜNSTLICHE INTELLIGENZ AM BEISPIEL EINES NEURALEN NETZWERKS UND GENTISCHEN ALGORITHMEN *(F.H.ECKHOFF)*

Um die Funktionsweise eines Neuralen Netzwerkes zu verstehen, gucken wir uns eine einfache Form an. Dabei übernimmt die Künstliche Intelligenz die Aufgabe eines Spielers des altbekannten Atari Spieles „Pacman“.

2.1. Definition Neurales Netzwerk

Neurales Netzwerk ist ein Überbegriff für verschiede Arten von Künstlichen Intelligenzen, die alle aber ähnlich aufgebaut sind. Sie sind alle an die Funktionsweise des menschlichen Gehirns angelehnt. Der Standard-Aufbau ist immer derselbe, es gibt sog. Neuronen, in denen Daten aufgenommen, summiert, kurzfristig gespeichert und mit irgendeiner Mathematischen Funktion in einen Bereich zwischen -1 und 1 gebracht werden. Zwischen den einzelnen Neuronen gibt es Verbindungen, welche Weights genannt werden. Weights gehen immer von jedem Neuronen einer Schicht zu jedem nächsten Neuronen. Klassische Formen des Neuralen Netzwerkes haben drei Hauptabschnitte: die Input-Neuronen: diese nehmen Usereingaben oder auch irgendeine andere Form der Daten auf. Als zweite Gruppe gibt es die Hidden-Neuronen, welche das Herzstück der Künstlichen Intelligenz sind. Hier passiert die „Magie“. In den Hidden-Layern geht es hauptsächlich um die Weights zwischen den Neuronen, die bestimmen, was das Neurale Netzwerk macht und wie gut es darin ist. Als letztes gibt es die Output-Neuronen, die dafür da sind, die Informationen auf ein gefordertes Format herunter zu brechen. Man braucht zum Beispiel für eine Künstliche Intelligenz, die ein Auto fährt wahrscheinlich ein Neuron, das das



Gaspedal kontrolliert und von 0 (gar nicht treten) bis 1 (voll treten) Werte ausgeben kann, eins, welches die Bremse steuert und genau wie das Gaspedal gesteuert wird und vielleicht eins, welches das Lenkrad steuert und von -1 (ganz links) bis 1 (ganz rechts) geht. Also braucht eine Auto Künstliche Intelligenz mindestens 3 Output-Neuronen. Die Weights zwischen zwei Neuronen werden am Anfang zufällig gesetzt. Damit das Netzwerk lernen kann müssen über mehrere Lehrnzyklen die Weights immer mehr angepasst werden, um ein optimales Ergebnis zu liefern. Dafür gibt es zwei mögliche Lernprozesse. Bei der Ersten, dem „supervised learning“ ist es nötig, das gewünschte Ergebnis zu kennen, da hier alle Eingaben zum Lernen mit einem sog. Label versehen werden, das die Ausgabe enthält[1], was zum Beispiel bei Handschrifterkennung der Fall ist. Es wird bei einem falschen Ergebnis das Zielergebnis genommen und rückwärts durch das Netzwerk geführt um zu sehen, welche Weights sich wie sehr ändern müssen. Die zweite Möglichkeit ist „unsupervised learning“, wenn das Ziel nicht genau zu definieren ist, wie es zum Beispiel bei einem Auto der Fall ist. Dann ist es nötig, eine so genannte Rewardfunktion zu haben, welche am Ende bestimmt, wie gut das Netzwerk war. Jedes Netzwerk fängt erstmal mit zufälligen Weights an. In dem Beispiel bedeutet das, dass das Auto erstmal zufällig Gas gibt, bremst und lenkt, aber nachdem das Neurale Netz auf verschiedenen Weisen die Weights immer weiter verändert hat, wird es irgendwann losfahren. Dieser Fortschritt wird dann verstärkt, indem das Neurale Netzwerk weiter verändert wird, ohne seine gelernten Fähigkeiten zu verlieren. Durch Verstärkung von positiven Eigenschaften und aussortieren von schlechten wird auf lange Sicht aus dem anfangs zufälligen Neuralen Netz ein guter, wenn nicht annähernd perfekter, Autofahrer.

2.2. Definition Genetischer Algorithmus

Bei Genetischen Algorithmen handelt es sich um eine Möglichkeit die Weights eines Neuralen Netzwerks zu verändern, ohne dass das Zielergebnis bekannt ist. Dabei bedient man sich wieder einem Beispiel aus der Natur. Bei dieser Methode wird das Prinzip des Überlebens des Stärkeren angewendet. Es werden mehrere Netzwerke erstellt (die Population), alle mit ihren individuellen zufälligen Weights. Dann wird jedes auf das spezielle Problem angewendet, sei es Autofahren, Schach spielen oder auch Pacman. Dann wird über die Rewardfunktion bestimmt, welche Netzwerke am besten waren. Die kann über verschiedenste Möglichkeiten geschehen. Bei

[1] Vgl. Klose, Olivia. “Machine Learning (2) - Supervised versus Unsupervised Learning”. *Olivia’s Blog*. 24. Februar 2015. 11. November 2018 <http://www.oliviaklose.com/machine-learning-2-supervised-versus-unsupervised-learning/>

Computerspielen ist es recht einfach, weil diese meistens mit einem eingebauten Score kommen. Aber alles kann als Rewardfunktion dienen; zur Not auch menschliche Kontrolle. Eine festgelegte Zahl der besten Netzwerke, aber mindestens zwei, wird dann zur weiteren „Fortpflanzung“ genommen. Die Gesamtheit aller Weights wird als Genom bezeichnet. Bei dem sogenannten Crossover werden zufällige Teile des Genoms der Eltern genommen und kombiniert.

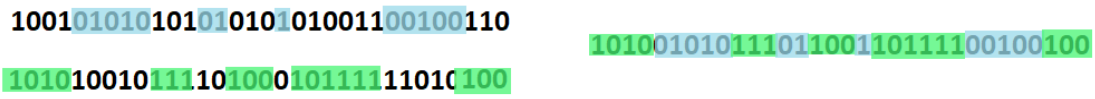


Abb. 1: schematische Darstellung des Crossovers in einem genetischen Algorithmus

Damit Inzucht verhindert wird, werden zwischendurch auch zufällige Mutationen eingebaut. Die Aufgabe des Programmierers liegt darin, auszuwählen, wie viele Netzwerke zu einer Population gehören. Außerdem muss er entscheiden, wie häufig eine Mutation vorkommen soll. Ist diese Zahl zu hoch, wird die Künstliche Intelligenz sich nie verbessern, weil jedes Neurale Netz zu unterschiedlich von seinem gut funktionierendem Vorgänger ist. Wird die Zahl jedoch zu klein gewählt, wird es nie über das erste Stadium der Zufälligkeit hinaus kommen.

2.3. Beispiel: Pacman

Das Spielprinzip des bekannten Atari-Spiels „Pacman“ ist recht einfach. Der Spieler steuert einen gelben Kreis, den sogenannten Pacman. Dieser muss sich in einem zweidimensionalen Labyrinth bewegen. Dabei muss er versuchen, Punkte zu sammeln. Die Herausforderung des Spiels besteht darin, dass sich in dem Labyrinth auch noch Geister bewegen. Sobald der Pacman einen Geist berührt, ist das Spiel zu Ende. In diesem Beispiel wird das Spiel etwas verändert. Zum einen ist die Grafik deutlich einfacher, dazu zählt, dass das Labyrinth nur durch ein einfaches Array aus ‚#‘ und ‚|‘ dargestellt wird. Der Pacman ist kein gelber Kreis mit Mund sondern nur ein ‚H‘. Auch die Geister sind anders als in dem Original. Es gibt nur noch einen Geist und dieser wird nur mit einem ‚E‘ gezeichnet. Der Spieler startet unten in der Mitte. Der Gegner startet oben links. Die vorhin erwähnten Punkte werden hinter dem Gegner erstellt. In diesem Beispiel wird der Gegner durch einen einfachen Wegfindungsalgorithmus gesteuert; er sucht einfach den kürzesten Weg zum Spieler. Die Künstliche Intelligenz wird dann die Kontrolle über den Pacman haben.

```
10  char map[18][32] = {
11      "+------------------------------+",
12      "|                              |",
13      "|                              |",
14      "|## ########### ##   #########|",
15      "|   |                          |",
16      "| | |### |  |           |      |",
17      "| |       |  | |###  |   |  |  |",
18      "| | #####|  | |      ## |      |",
19      "| |           |###  |      |  |",
20      "| |##### ###         ##       |",
21      "|          ######  ####### ###|",
22      "|                              |",
23      "|# ### ####      ###   #######|",
24      "|                              |",
25      "|                              |",
26      "|                              |",
27      "|                              |",
28      "+------------------------------+"
29      };
```

Aus ersichtlichen Gründen wird in dieser Arbeit nur auf die essentiellen Teile des Codes und des gesamten Projektes eingegangen. Außerdem wird es einige Funktionen geben, die nicht erklärt werden. Dabei handelt es sich meistens um Funktionen, die in dieser Verwendung des Netzes nicht verwendet werden. Wenn man das Netz anders erstellt oder verwendet, können diese Funktionen verwendet werden, dies ist aber in der aktuellen Version nicht der Fall.

2.3.1. Neurales Netzwerk

In folgenden wird das Kernstück der Künstlichen Intelligenz erklärt, das Neurale Netzwerk. Es wird zuerst definiert, was ein Weight ist. Es wird deklariert als long double. Als nächstes wird geklärt, was Neuronen sind. Jedes Neuron hat Informationen darüber, wie viele Neuronen nach ihm kommen, auch hat es ein Vector aus Werten, genannt tempStorage. Dieser speichert alle Werte, die das Neuron von vorherigen Neuronen bekommen hat. Aus diesen wird mit Hilfe der Funktion calculateSum ein storedValue errechnet; die Funktion calculateSum geht durch den gesamten Vector tempStorage und addiert alle Werte. Danach wird eine Sigmoidfunktion angewandt, welche das Ergebnis in einen Bereich zwischen -1 und 1 bringt. Der wichtigste Bestandteil des Neurons ist im Vector weights gespeichert. Dieser besteht aus Weights, welche einfache doubles sind. Am Anfang des Netzes wird weights über die Funktion FirstFillWeights mit Zufallszahlen gefüllt. Die Definition des Vectors weights ist: von diesem Neuron alle ausgehenden Verbindungen. Das heißt, jedes Neuron hat die Verbindungen nach vorne gespeichert.

Als nächstes wird die Klasse Net erklärt. Das Net hat einige Informationen darüber, wie es aufgebaut ist und es wird eine Liste von Eingangs Werten definiert. Der Kernbestandteil sind natürlich die einzelnen Neuronen. Diese werden in dem Vector neuralesNetz gespeichert. Man kann sehen, dass dieser Vector nicht nur eine Liste von Neuronen ist, sondern eine Liste aus Listen an Neuronen. Das kommt daher, dass das Netz zweidimensional ist. Es besteht aus einer Reihe an Layern (Ebenen). Diese Layer sind selbst auch eine Liste an Vectoren; somit macht es Sinn, das Netz als Liste aus Listen zu deklarieren. Die Funktionen der Klasse sind recht selbsterklärend. ShowWeights wurde nur zur Übersicht bei der Entwicklung genutzt, genauso wie flush und showSteps. Die tatsächlich verwendeten Funktionen sind: Save, diese Funktion speichert das Neurale Netz bzw. die weights in einer „.weights“ Datei. LoadWeights kann diese Werte aus der Datei einlesen, sodass man ein vorgefertigtes Netz verwenden kann. GetWerteFromFile liest die anfangs Werte aus einer Datei ein. PutWerteToNN schreibt diese eingelesenen Werte dann in die ersten

Neuronen des Netzes. Und FeedForward „sagt“ dann jedem Neuron, es solle seine Werte mit den entsprechenden weights verrechnen und dann die nächsten Neuronen weiter geben.

In dem Code passiert auch ein wichtiger Schritt, auch wenn dieser fast schon unscheinbar versteckt zwischen den Zeilen steht. Es wird das Eigentliche Netz, das Objekt der Klasse Net erstellt und myNet genannt.

Als nächstes schauen wir und das eigentliche Programm an, die main Funktion. Dort wird als erstes einiges an Einstellungen vorgenommen. Es wird die Anzahl an Nachkommastellen festgelegt und es werden die nötigen Dateien in dem Filestream geöffnet. Da das Netzwerk am Ende verschiedenste Dateien öffnen soll, weil es mehrere Neurale Netze geben wird, wird die Information, welche Dateien benutzt werden, von einem anderen Programm übergeben. Dies geschieht über tempargv1. Dann öffnet das Programm zum einen die Datei mit der Endung .GEN, dort wird später das interpretierte Ergebnis ausgegeben. Die Endung .weights steht für das eigentliche Netz, welches geladen wird. Und .saveoption ist nochmal dasselbe wie .weights aber es kann manchmal, wenn der PC sehr ausgelastet ist, zu Zugriffsüberschneidungen kommen, wobei das Netz gelöscht wird; daher wird immer eine Sicherheitskopie angelegt. Dann wird zur Sicherheit geprüft, ob die .weights Datei ordnungsgemäß geöffnet wurde.

Im Anschluss wird der strukturVektor erstellt. Dieser bestimmt, wie viele Layer und Neuronen das Netz hat. Man könnte auch diese Information entweder aus der Datei auslesen, oder über die Konsole weiter geben lassen. Der Einfachheit halber wird es hier aber direkt im Code deklariert. Das Netzwerk ist ein 12,15,15,4er Netzwerk das heißt, dass es 4 Layer hat; 12 Input-Neuronen dann zwei Layer aus 15 Hidden-Neuronen, und dann 4 Output-Neuronen. Die nächste Abfrage wird wichtig, wenn man das Netz zusammen mit der Umgebung laufen lässt. Dort wird geklärt, ob ein neues Netzwerk mit zufälligen Weights erstellt oder aus einer bestehenden Datei eingelesen werden soll.

Dann wird die Funktion ErstelleNetz aufgerufen, welche z.B. feststellt, wie viele Neuronen das Netz hat. Es wird aber auch ein vector aus Neuronen erstellt (ein Layer) und in den vector neuralesNetz von myNet eingefügt. Die Funktion fügt pro Layer die aus dem strukturVektor hervorgehende Anzahl an Neuronen hinzu; und, wenn das Netz noch nicht existierte, führt die Funktion für jedes Neuron des Netzes die Funktion FirstFillWeights aus.

Als nächstes in der main Funktion werden, wenn nötig, die weights aus der Datei eingelesen. Beim Speichern in der Datei wird vom ersten Neuron „oben links“ angefangen, und dann das erste Layer abgearbeitet. Bei jedem Neuron wird mit der ersten Verbindung angefangen. Somit steht in der Datei als erstes das Weight vom ersten Neuron zum ersten Neuron des 2. Layers und als letztes das Weight vom letzten Neuron des vorletzten Layers zum letzten Neuron des letzten Layers.

Danach wird das Netz einmal gespeichert (eigentlich nur für den Fall, dass das Netz neu erstellt wurde).

Danach werden die Werte aus der Datei eingelesen, mit PutWerteToNN in die ersten Neuronen geschrieben und, da die Werte in dem storedValue der Neuronen sein müssen und nicht nur im tempStorage, wird einmal die Funktion FeedForward ausgeführt.

Als Nächstes muss der Wert, den das Programm bekommen hat, einmal das gesamte Netz durchlaufen. Dafür wird für jeden Layer des Netzes einmal die Funktion calculateSum bei jedem Neuron aufgerufen und danach FeedForward. So „bewegt“ sich der Wert von Layer zu Layer.

Als letztes werden die Ergebnisse über writetofile in eine Datei geschrieben. Diese Funktion macht nichts anderes, als zu überprüfe, welche der Neuronen die höchste Aktivierung hatten, welche am stärksten angesteuert wurden. In diesem Fall wird ein ‚U‘ in die Datei geschrieben, wenn der erste Neuron den höchsten Wert hat, ein ‚R‘ beim zweiten, ein ‚D‘ beim dritten und ein ‚L‘ beim vierten. Diese Buchstaben werden später für die Bewegungsrichtungen des Pacman stehen. Sie sind in keiner bestimmten Reihenfolge, da diese auch egal ist, da das Netz diese dann mitlernt. In dem jetzigen Zustand kann das Netz, mit ein paar kleinen Veränderungen, für alles Mögliche verwendet werden. Wenn man z.B. die Anzahl der Input- und Output-Neuronen ändert, kann man es auch für Bilderkennung verwenden. Dabei nimmt man z.B. an, dass jedes Eingansneuron für die Helligkeit eines Pixels eines 16x16 großen Graustufenbildes steht und die Ausgangsneuronen stehen für 12 verschiedene Tiere, die das Netz erkennen kann. Dann müsste man nur den strukturVektor auf z.B. 256,100,100,12 ändern. Daher ist diese Form der künstlichen Intelligenz sehr praktisch; es ist quasi eine Allzweckwaffe.

2.3.2. Genetischer Algorithmus

Der Genetische Algorithmus ist der Teil des Programms, der das Lernen ermöglicht. Der Code dazu befindet sich in Anhang 2. Das Programm liest die Werte aus den

„.return“ Dateien aus, speichert sie in dem Vector allRets und sucht dann die vier besten Werte.

Die vier Netze, die die besten Werte erzielt haben, werden eingelesen und per crossover kombiniert. Dabei hat jedes Netz dieselbe Wahrscheinlichkeit. Dann werden alle „.weight“ Dateien mit dem neuen Genom überschrieben. Es wird auch zufällige Mutation verwendet .Die Mutation wird durch eine Zufallszahl von 0 bis 200 entschieden. Nur wenn diese Zahl 0 ist, wird eine Mutation durchgeführt. Das heißt, die Wahrscheinlichkeit liegt bei 1/200 oder auch 0.5%. Bei 465 Genomen pro Netz macht das ca. 2.3 Mutationen pro Crossover.

Diese gesamte Prozedur sorgt dafür, dass die vier besten Netze zusammengefügt und zu mehreren neuen Netzten kombiniert werden, in der Hoffnung, dass alte Fehler dadurch behoben werden.

2.3.3. Weitere wichtige Bestandteile des Projekts

2.3.3.1. Pacman

Die Pacman Spieleumgebung ist der einzige Teil des Projekts, der nicht komplett von mir selber geschrieben wurde. Der Ruhm dafür gebührt dem Nutzer „rmxhaha“. Die Datei dazu heißt „main.exe“. Die Änderungen am Code betreffen das Einlesen der Richtung aus einer Datei und auch das Schreiben der der Informationen in die „.map“ Datei für das Neurale Netz. Auch das Punktesystem wurde überarbeitet und die Visualisierung wurde ebenfalls entfernt.

2.3.3.2. Die „Augen“ des Pacman

Hier liegt eines der größten Probleme dieses Netzes. Am besten sollte das Neurale Netz dieselben Informationen bekommen wie ein Mensch der das Spiel spielt. Also das gesamte Spielfeld überblicken. Das Problem ist, dass das selbst bei einem recht kleinen Spielfeld wie hier (18x32) insgesamt 576 Eingaben bedeutet. Das wären dann etwa 300.000 Weights die alle einen Einfluss auf das Netz haben. Dies wäre einfach zu viel für dieses Projekt, da das Lernen durch den Genetischen Algorithmus dadurch deutlich länger dauern würde.

Die Lösung ist in diesem Fall, dass Pacman nur in Richtung der Achsen Informationen erhält. Er sieht also was über, unter, rechts und links von ihm ist. Dies passiert in den Funktionen, die mit „calculate“ anfangen. Es wird zuerst der Abstand zum nächsten Objekt gemessen und dann der ASCII Wert in die „.map“ Datei geschrieben.

2.3.4. Das Belohnungssystem

Das ursprüngliche Programm sah vor, für jeden vom Gegner hinterlassenen, gegessenen Kreis ein Punkt zu vergeben. Das Problem ist allerdings, dass Pacman am Anfang nie einen Kreis treffen wird. Daher wurde hier eine „rundenbasierte“ Belohnung eingeführt. In jeder Runde (jedem Zug) wird ein Wert erhöht, der am Ende zu den gegessenen Punkten addiert wird. Auch soll ein Verhalten, welches zum Fressen eines Punktes führt, auf jeden Fall erhalten bleiben; daher wird pro Kreis nicht mehr Eins sondern 60 addiert.

2.3.5. Gencheck

Um den Fortschritt des Pacmans beobachten zu können, wurde dieses Programm geschrieben. Es ist eine langsamere und daher beobachtbare Version des Hauptprogramms. Die Bedienung ist einfach: man nehme entweder eine Datei aus dem „best“ Ordner, oder ein anders Netz, welches „.best“ benannt wurde und kopiere es in den „check“ Ordner. Dann muss man die Datei nur noch „0.best“ nennen und „Gencheckmain.exe“ starten. So kann man sehen, wie sich Pacman mit dem entsprechenden Netz verhält.

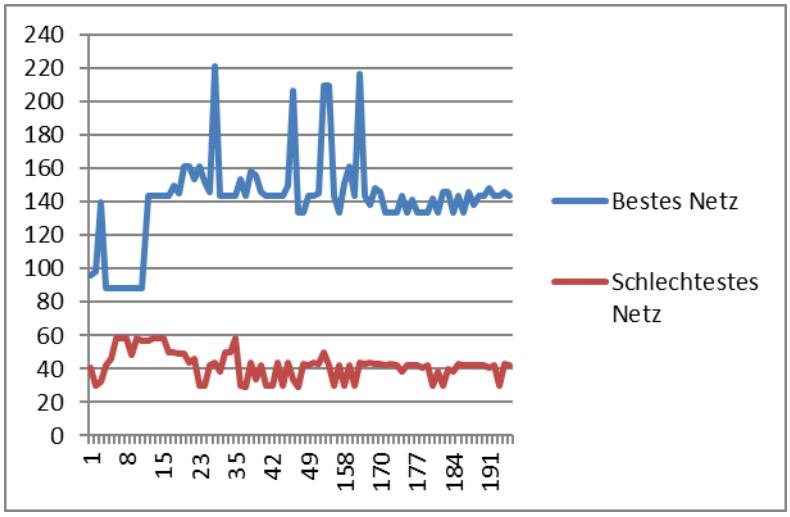
2.3.6. Startumgebung

Da eine mehrfache Wiederholung des Durchlaufs erwünscht ist, braucht das Programm eine Umgebung. Diese wird durch die Programme „SUN.exe“ und „runPacman.exe“ gegeben.

SUN steht für „Software unabhängiges Neurales Netz“ (man braucht keine Zusatzprogramme außer Windows oder Wine). Dieses Programm führt runPacman in diesem Fall 1000 Mal aus. RunPacman wiederum führt das Spiel 100 Mal aus (da es 100 Netze gibt).

2.4. Daten Auswertung

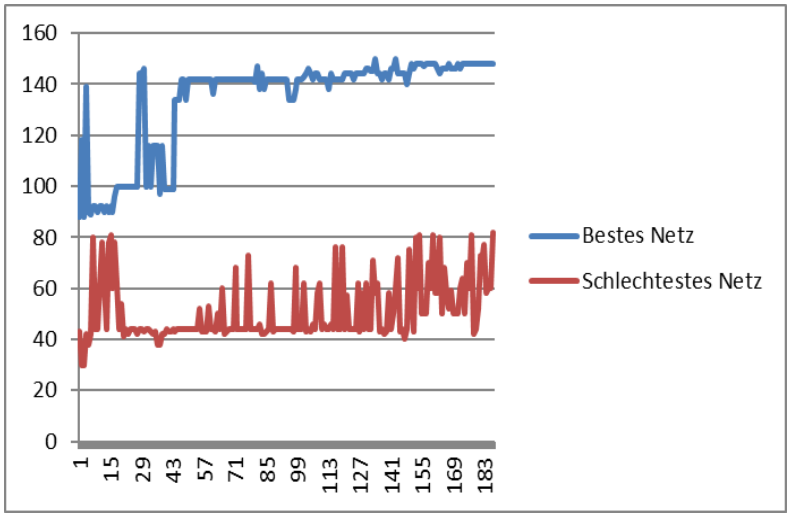
Im Folgenden werden ein paar repräsentative Ergebnisse vorgestellt und kurz analysiert.



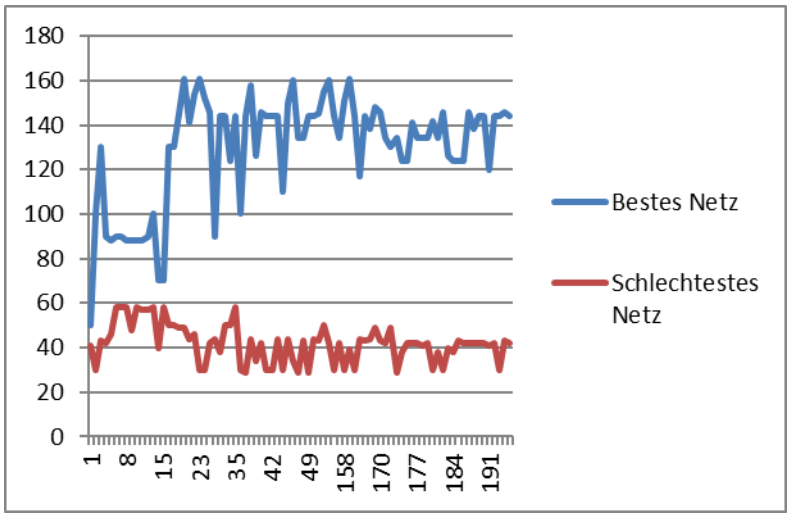
Durchlauf 1 (1/200 Mutation 220 Runden)

In dem Diagramm wurden alle falschen Werte entfernt.

Es ist zu sehen, dass nach ca. 45 Runden keine Verbesserung zu erkennen ist. Diese Beobachtung lässt sich mit Gencheckmain.exe bestätigen. Pacman läuft nach rechts oben und bleibt in der Ecke bis er vom Gegner ausgelöscht wird. Dies lässt sich möglicherweise mit einer höheren Mutationsrate verbessern, da dann mehr Vielfalt in den möglichen Entscheidungen herrscht.



Durchlauf 2 (1/50 Mutation 221 Runden)

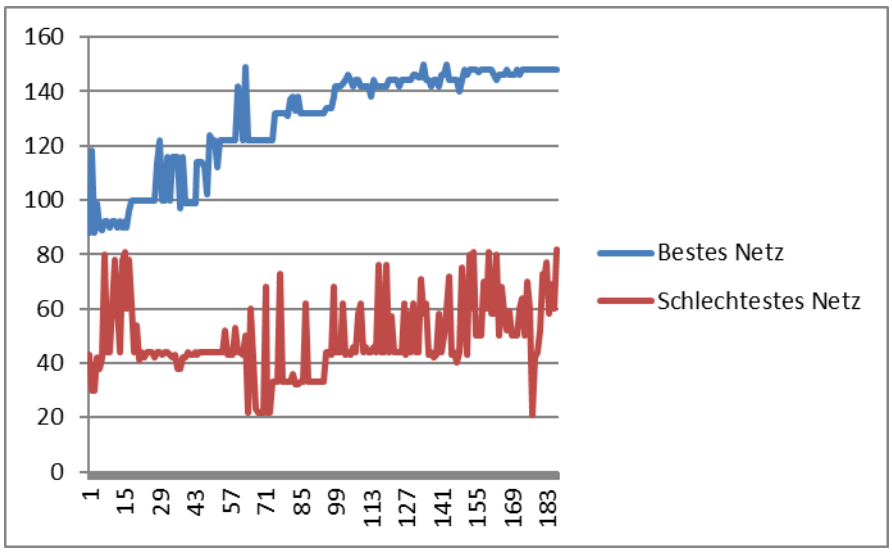


Mit Ausnahme einiger Spitzen, bei denen nicht ganz klar ist, ob es sich um Fehler handelt, erreicht das

Netz recht schnell seinen Stagnationspunkt. Auch hier scheint wieder eine zu geringe Mutationsrate zu herrschen.

Durchlauf 3 (1/25 Mutation 243 Runden)

Bei diesem Durchgang ist zu sehen, dass eine so hohe Mutationsrate zu einem fast zufälligen Ergebnis führt. Dies tritt auf, da sich eine hohe Wahrscheinlichkeit besteht, dass gute funktionierende Features wieder aus dem Netz entfernt werden. Eine weitere Herabsetzung der Mutationswahrscheinlichkeit (sprich hohe Mutationsrate) ist somit nicht ratsam. Führen wir also noch einen letzten Durchlauf durch mit einer noch kleineren Mutationswahrscheinlichkeit als bei Durchlauf 1.



Durchlauf 4 (1/250 Mutation 200 Runden)

Es zeigt sich, dass der Verlauf flacher ist als bei allen anderen, was sich einfach durch die geringere Veränderung durch die geringere Mutationsrate erklären lässt.

Trotzdem erreicht die Population nach ca. 100 Runden einen Punkt, ab dem kaum eine Veränderung zu verzeichnen ist.

2.5. Fazit

Nach ca. 170 Stunden Simulationen lässt sich generell sagen, dass eine geringe Mutationsrate (<1/150) eine sichere aber auch immer langsamer werdende Steigerung des Ergebnisses führt. Je mehr Mutationen passieren, desto unzuverlässiger wird eine Verbesserung. Interessant ist das Maximum, welches zu beobachten ist. Es ist definitiv nicht das Limit des Spiels. Es ist wahrscheinlicher, dass hier ein zwar kurzfristig gutes Verhalten gelernt wurde, welches aber zu einem solchen Ergebnis führt. Mit „Genchecker.exe“ lässt sich ein solches Muster teilweise erkennen. Oft kommt es vor, dass Pacman in eine Ecke läuft. Das passiert, weil das zu Beginn die beste Methode ist, um lange zu überleben. Dies ist allerdings nicht immer die beste Option. Auch sieht man bei fast allen Netzen, dass dieses temporäre Maximum eigentlich nach ca. 50 Durchläufen erreicht ist. Bei den meisten KI’s ist festzustellen, dass es an diesem Punkt einen Sprung gibt. Daraus lässt sich schließen, dass es einen Kernbestandteil gibt, der allen nachfolgenden Netzen ein wichtiges verhalten beibringt. In diesem Fall ist es vermutlich die Fähigkeit, gerade in eine Richtung zu laufen, sprich die Netze haben zumindest gelernt, unsinniges „feuern“ der Neuronen zu beenden.

Das Problem hat wahrscheinlich auch damit zu tun, dass das Netz die Informationen nicht „verstanden“ hat, da Objekt und Entfernung komplett Zusammenhangslos übergeben werden. Man könnte z.B. einen Vector aus zwei Zahlen als eine Information übergeben, die dann auch zusammen verarbeitet werden. Auch die Übergabe des ganzen Feldes, die zuvor ausgeschlossen wurde, wäre eine Option. Dann würde zwar das Lernen sehr lange dauern, solche Probleme wären dann aber ausgeschlossen.

Schlussendlich muss der Programmierer entscheiden, was er möchte. Ein neurales Netz kann, aufgrund seiner Ähnlichkeit zum Gehirn, fast alles lernen, was ein Mensch lernen kann, sodass es in seinen Anwendungsmöglichkeiten sehr flexibel ist.

3. KÜNSTLICHE INTELLIGENZ: ANWENDUNG DES Q-LEARNING ALGORITHMUS AN TICTACTOE*(J. MICHAEL)*

3.1. Definition

Q-Learning ist eine Form des MachineLearnings, die auf der Unterteilung der Umgebung in Zustände basiert. Sie zeichnet sich neben der Lernweise des Agenten[2] vor allem durch die im Normalfall auf zuvor definierte Aktionen beschränkte Interaktion mit der virtuellen Umgebung aus. Dabei unterscheidet man zwischen einer deterministischen Umgebung und einer stochastischen Umgebung, wobei bei ersterer die gewählte Aktion immer eintritt, bei letzterer dagegen zu einer bestimmten Wahrscheinlichkeit eine andere Aktion durchgeführt wird. Mit jener letzteren Umgebung ließe sich auch Risikoabwägung darstellen. Dabei erfordert jedoch eine stochastische Umgebung, dass es sich um einen begrenzten MDP[3] handelt. Andernfalls ist die Lernmethode nicht anwendbar, da, wie zuvor erwähnt, alle Zustände im Voraus klar definierbar sein müssen.

Q-Learning erhält seinen Namen durch das Q-Array, kurz für „Quality-Array“. Es stellt die Qualität einer Aktion ***a*** im Zustand ***S*** dar[4]. Zusätzlich gibt es in vielen Fällen ein R-Array, das die tatsächlichen Belohnungen beim Erreichen bestimmter Zustände beinhaltet. Diese Belohnungen müssen, anders als die temporären Belohnungen, zuvor definiert werden. Oftmals wird allerdings nur dem Erreichen eines Endzustands (und oft einer Abschlussaktion) ein Wert wie „100“ zugewiesen. Beim klassischen Q-Learning werden hier aber auch die Verknüpfungen zwischen zwei Zuständen dargestellt (0 bei existierender Verbindung, -1 bei nicht existierender Verbindung). Bei steigender Komplexität und besonders bei hoher Zahl an Zuständen werden andere Wege gewählt, um die Belohnungen und Verbindungen zuzuweisen[5].

Die zuvor genannte Version des Q-Learning ist der hier in leicht abgewandelter Form verwendete Lernmechanismus. Es existiert jedoch eine weitere Variante des Q-Learning, bei der kein Q-Array existiert[6]. Bei dieser Variante werden die Q-Werte durch ein neurales Netzwerk[7] berechnet. Sie wird besonders verwendet, wenn die

---

2 Siehe „Agent“, Kap. 5, S.26

3 Siehe „Markov Decision Process“, Kap. 5, S.26

4 Vgl. Matiisen, Tambet. „Demystifying Deep Reinforcement Learning“. *Computational Neuroscience Lab Institute of Computer Science, University of Tartu.* 19. Dezember 2015. 11. November 2018. <http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/>

5 Siehe Kap. 3.2 und 3.3.2

6 Siehe Kap. 4

7 Siehe Kap. 2

Zahl der Zustände zu groß zur effizienten Berechnung ist. Dies ist zum Beispiel bei dem 3x3x3 Rubik's Cube der Fall, aber auch, wenn in einem Zustand vorherige Zustände mit eingebunden werden[8].

In dieser Variante, als auch bei der regulären Variante gibt es Möglichkeiten, den Lernprozess zu verbessern. Diese Variante nennt sich „ε-greedy Exploration“[9] und beschreibt, dass es einen sich zeitlich verringernden Wert ε gibt, der die Wahrscheinlichkeit angibt, zufällig eine Aktion zu wählen oder das Gelernte anzuwenden. Dabei soll möglichst effizient die beste Folge von Aktionen gefunden werden, um den/die Zielzustände zu erreichen. Im Beispiel für Q-Learning[10] wird diese Methode noch nicht verwendet, die Vorteile dieses Lernmechanismus werden dagegen im Beispiel mit TicTacToe dargestellt.

3.2. Das Beispiel mit drei Schaltern

Als sehr einfaches Beispiel für ein solches Programm dient eine Reihe von drei Schaltern, die als Ziel alle nach oben (entspricht "AN" bzw. 1) geschaltet werden sollen, wobei jede Schalterkombination ein Zustand ist. In unserer Q-Tabelle erhalten wir damit die $2^3 = 8$ Kombinationen (in der Reihenfolge der Zustand der Schalter (0/1)) als Reihenbeschriftung und als Spaltenbeschriftung die Aktion "Schalter *a* umlegen", weshalb wir folgende Tabelle erhalten:

a/s	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0

Tabelle 1 Diese Tabelle stellt die Zuordnung Zustand (Reihe) und Aktion „Schalter x umlegen“ (Spalte) dar

Zur besseren Übersicht über die Verbindungen der Zustände lässt sich auch ein Flussdiagramm zeichnen. Daran lässt sich auch die Lernweise des Programms nachvollziehen.

---

8 Vgl. Matiisen, Tambet. „Demystifying Deep Reinforcement Learning”

9 Tokic, Michel. „Adaptive ε-greedy exploration in reinforcement learning based on value differences“. *Tokicblog*. 2010. 11. November 2018. <http://www.tokic.com/www/tokicm/publikationen/papers/AdaptiveEpsilonGreedyExploration.pdf>

10 Siehe Kapitel 3.2

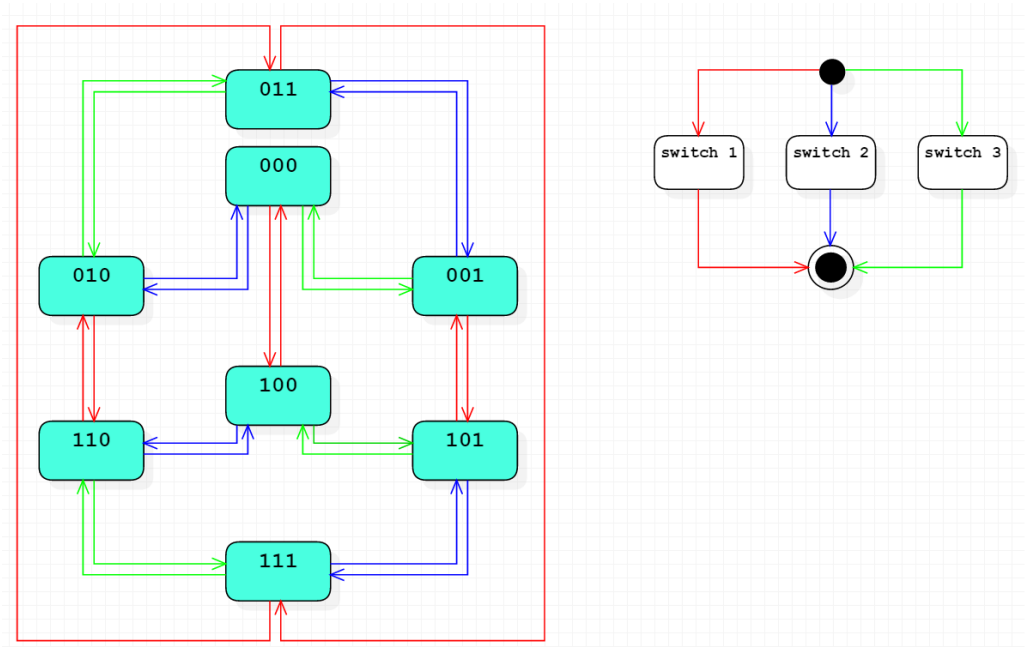


Abbildung 1 Die Abbildung stellt die Übergänge zwischen den Zuständen ohne deren Gewichtung dar

Wie in der Legende auf der rechten Seite angedeutet entspricht ein roter Pfeil dem Umlegen des ersten Schalters (im Programm Schalter 0), Blau dem Umlegen des zweiten Schalters und grün dem Umlegen des dritten Schalters.

Wie bereits in der Erklärung des Q-Learnings erwähnt wurde, werden beim Lernen den Aktionen bei bestimmten Zuständen bestimmte Belohnungen zugewiesen. Wenn man das Beispielprogramm ausführt, erhält man die folgende Abbildung.

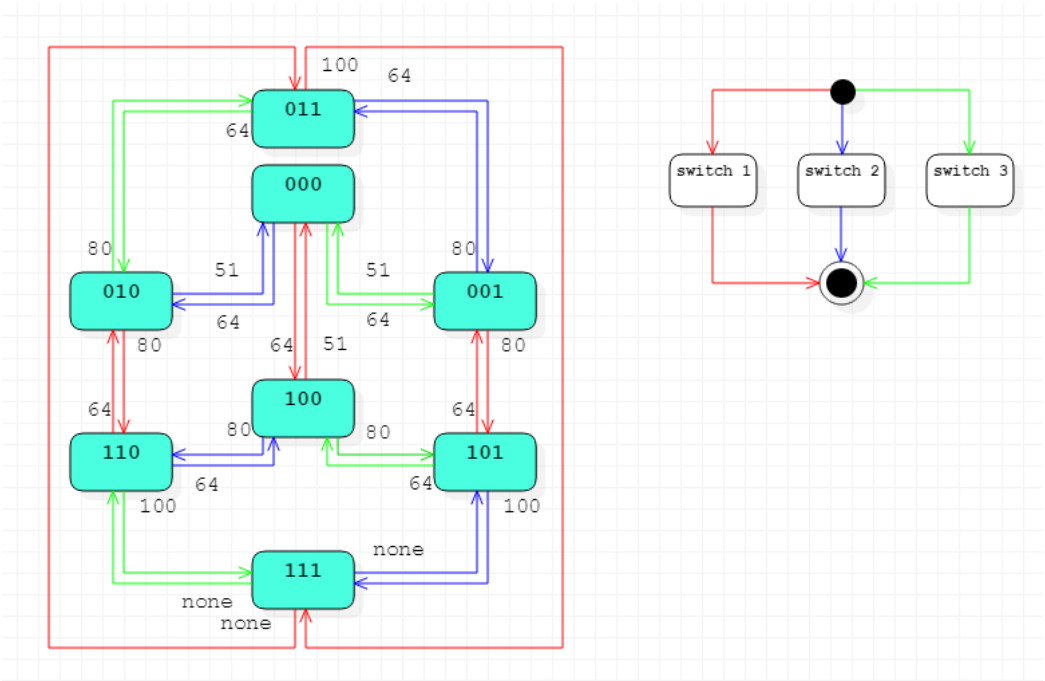


Abbildung 2 Das Flussdiagramm mit der Gewichtung

Wird nun zufällig ein Startzustand gewählt (der nicht „111“ ist, da das unser Endzustand ist), kann durch die Wahl der Aktion mit der höchsten Belohnung stets der kürzeste Weg zum Ziel ermittelt werden. Der Agent folgt also dem Belohnungsgradienten aufwärts.

3.2.1. Der Code

Der Code des Programms (siehe Anhang: QLearning -> QLearningEX) ist leicht zu erklären. Die elementaren Funktionen des Programms sind neben der *Main()* Funktion vor allem *SetupQ()* [siehe Z. 53 - 78], *Learn()* [siehe Z. 80 - 122] und *PrintSequences()* [siehe Z. 124 - 154].

Zunächst zu *SetupQ()*: wir verwenden hier die in Z. 29 – 32 definierten Zustände der Schalter (als Strings), um das Q-*Dictionary*[11] *(Quality,* nachfolgend trotzdem als Q-Array bezeichnet*)* und das R-*Dictionary (Reward*, nachfolgend trotzdem als R-Array bezeichnet*)* zu füllen. Dazu setzen wir die Strings aus unserem *statesString*-Array als *Keys*[12] und ein double-Array als *value*. Das double-Array ist drei doubles lang, sodass jeder der drei Aktionen ein double zugewiesen werden kann. Das Q-Array wird dabei an jeder Stelle mit 0 gefüllt, da die Werte erst beim Lernen erhalten werden. Das R-Array dagegen wird je nach Stelle mit 100 oder 0 bezeichnet. Wird der Wert für den Zustand 4, 5 oder 6 zugewiesen („110“, „101“, „011“), wird bei der jeweils zielführenden Aktion (2, 1, 0) die Belohnung hinzugefügt. Die Belohnung wird hier schon zugewiesen, da es keine *Abbruch*-Aktion gibt, die im Zielzustand durchgeführt werden muss und so die Simulation abbricht. Deshalb dient die jeweils letzte Aktion, die zum Erreichen des Zielzustandes führt, als Abbruch-Aktion, sodass ihr bereits eine Belohnung zugewiesen werden muss.

Die *Learn()*-Funktion beginnt mit zwei *Debug*-Statements, die in der Konsole zeigen, ob *SetupQ()* erfolgreich war [siehe Z. 82 + 83]. Anschließend wird eine *Random*-Instanz erzeugt, die im Laufe des Lernprozesses zufällig den Anfangszustand festlegt. Zusätzlich werden weitere Variablen zugewiesen: **int** nextAction wird beim Lernen mit einer Zufallszahl]0, 3] belegt, **string** nextState dient als Zwischenspeicher für den nächsten Zustand, **int** tr wird als Zähler der Trainingsdurchläufe genutzt, **double** hV (highestValue) dient als Zwischenspeicher für den höchsten Wert der Belohnungen nach der Durchführung der Aktion. Die **doubles** v und r dienen

[11] Siehe „Dictionary“, Kap.5

[12] Siehe „Dictionary“, Kap.5

ebenfalls als Zwischenspeicher für den neuen Wert des Q-Arrays (v) und dem R-Array Wert (r).

Anschließend werden 500 Trainingsrunden durchgeführt. Genau genommen sollten 6 Runden ausreichen, wenn jeder Weg gewählt wird, da es von „000“ aus drei mögliche Aktionen gibt, die zum Ziel führen, dann vom nächsten Zustand aus zwei, und dann nur noch eine. Also gibt es 3! = 6 Wege, von „000“ zu „111“ zu kommen. Da es sich jedoch um zufällige Wahl der Aktionen handelt, kann nicht darauf vertraut werden, dass jeder Weg exakt ein Mal gewählt wird. Deshalb wird bewusst eine deutlich höhere Rundenzahl gewählt, um definitiv ein konvergierendes Q-Array zu erhalten. Konvergierend wird das Array bezeichnet, weil durch die Berechnung des neuen Q-Werts durch die Bildung der Differenz sich der Wert immer weiter seinem „richtigen“ Wert annähert. Auf diese Weise ändert sich irgendwann der Wert nicht mehr, was bedeutet, dass das Array konvergiert ist.

Ein Trainingsdurchgang läuft folgendermaßen ab: zunächst wird zufällig eine Anfangskombination aus dem statesString-Array gewählt [Z.95] und dem agentState (public Variable) zugewiesen, sowie die Trigger auf die jeweiligen Werte gesetzt [Z.96] werden. Anschließend werden solange Entscheidungen getroffen, bis der Endzustand „111“ erreicht wird. Zunächst wird zufällig eine Aktion (0 – 2) gewählt [Z.100] und der Trigger an der Stelle umgelegt wird [Z. 101, siehe „ChangeTrigger“], wobei der String der neuen Kombination NACH umlegen des Triggers zurückgegeben wird, welcher dann dem Zwischenspeicher nextState zugewiesen wird. Mit den erhaltenen Informationen (Zustand, Aktion, nächster Zustand) kann nun die Qualität der Aktion bestimmt werden. Dafür wird zunächst die höchste Belohnung im nächsten Zustand aus allen Aktionen ermittelt [siehe Z. 104].

Auch die Belohnung kann ermittelt werden [Z. 105], wodurch mit der Formel

$$Q(s,a,s')_{neu} = Q(s,a,s')_{alt} + \alpha * \left[R(s,a) + \gamma * \mathrm{MAX}(\mathrm{s}' \mid 0, \mathrm{s}' \mid 1, \mathrm{s}' \mid 2) - Q(s,a,s')_{alt}\right]$$ [13]

der neue Q-Wert errechnet werden kann.

Dabei sind $s' \mid n$ mit $n = \{0,1,2\}$ die Q-Werte der Aktionen von 0 bis 2 vom Zustand s' aus. Die Werte für $R(s,a)$ sowie für $\mathrm{MAX}(\mathrm{s}' \mid 0, \mathrm{s}' \mid 1, \mathrm{s}' \mid 2)$ wurden in einer Variable gespeichert. Unser γ wird zuvor definiert und dient lediglich zur Abwägung des Risikos bei stochastischen Umgebungen bzw. der Erstellung des Gradienten. Damit

[13] *s* entspricht dem aktuellen Zustand, *a* der gewählten Aktion, s' der nächsten Aktion

wird bestimmt, wie wertvoll Langzeiterfolge sind im Vergleich zu sofortiger Belohnung. Der zweite vordefinierte Wert ist α, der als Lernrate bezeichnet wird. Der Wert liegt, genau wie γ, zwischen 0 und 1 und beschreibt, wie groß der Einfluss eines größeren bzw. kleineren Wertes auf den neuen Q-Wert ist. Je kleiner α ist, desto langsamer, aber desto sicherer nähert sich der Agent dem richtigen Q-Wert an. Das Q-Array konvergiert also langsamer, erleidet dabei aber auch weniger Schwingungen um den Zielwert herum.

Zuletzt wird die Aktion ausgeführt und der neue Zustand des Agenten auf die neue Position der Trigger gesetzt [Z. 111]. Dann wird nur noch überprüft, ob der Endzustand erreicht wurde [Z. 112f]. Ist dies der Fall, wird die while-Schleife verlassen, anderenfalls wird eine neue Aktion ausgewählt.

Wurden alle 500 Trainingsdurchläufe durchgeführt, werden die Ergebnissequenzen durch *PrintSequences()* ausgegeben. Dabei werden alle 7 Anfangszustände durchlaufen (alle außer dem Zustand „111“) [siehe Z. 127 – 130, 135] und wie zuvor solange Aktionen ausgewählt, bis der Zustand „111“ erreicht wurde.

Dabei werden die Aktionen jedoch nicht zufällig gewählt, sondern stets die Aktion mit der höchsten Qualität, also dem höchsten Q-Wert. Haben mehrere Aktionen dieselbe Quality, wird die erste Aktion gewählt, die die Quality besitzt. Dieser Vorgang ist selbsterklärend und findet sich in der Funktion *FindHighestIndex(double[])* [Z. 155 – 171]. Zuletzt wird der **String** agentState auf die neue Kombination der Trigger gesetzt. Auch hier wird die Schleife beendet, wenn der neue Zustand „111“ ist. Zuletzt wird nur dieser Zustand ausgegeben, da sonst die Ausgabe beim letzten Zustand vor „111“ endet.

Alle weiteren Funktionen sind durch ihre Anwendung selbsterklärend oder dienen lediglich der Ausgabe von wichtigen Informationen für mögliche Debug-Arbeiten.

3.3. TicTacToe

3.3.1. Das Spiel

TicTacToe ist ein bekanntes und relativ einfaches Spiel für zwei Personen. Auf einem quadratischen Feld mit den Maßen 3x3 wechseln sich die Spieler ab und setzen ihr Zeichen – einen Kreis oder ein Kreuz – in eins der neun Felder. Das Ziel des Spiels ist es, drei Felder in einer bestimmten Form mit seinem Zeichen in bestimmter Konfiguration zu markieren. Die Felder können in einer Reihe, Spalte oder diagonal liegen. Der Spieler, dem dies zuerst gelingt, gewinnt. Sind alle Felder belegt, und kein

Spieler hat eine solche Kombination erzielt, so ist das Spiel unentschieden ausgegangen.

3.3.2. Zustände in TicTacToe

Durch seine zweidimensionale Struktur lassen sich die Zustände nicht so leicht als Zahl darstellen. Eine Herangehensweise ist es, das zweidimensionale Array in ein eindimensionales umzuwandeln. Man erhält so, wie auch zuvor, einen String, der aus Zahlen von 0 bis 2 besteht. 0 repräsentiert dabei ein nicht ausgefülltes Feld, 1 die eigenen und 2 die gegnerischen Zeichen (siehe Kapitel 3.3.4). Wie zuvor auch kann die Zeichenfolge als Zahl in bestimmter Base gesehen werden. Da es 1, 2 und 3 in der Zeichenfolge gibt, ist damit die Base 3 sinnvoll. Das Problem dabei ist jedoch, dass nicht jeder Zustand eintreten kann, da die Spieler abwechselnd handeln, sodass die Differenz der Anzahl der Zeichen nie größer als 1 sein kann. Dadurch ist nicht jede Zahl im Array vorhanden, sodass die Zahlen nicht den Index widerspiegeln. Aus diesem Grund wird eine weitere Möglichkeit angewandt, bei der die Zustände als Instanzen der Klasse *„state“* als Membervariablen die Zeichenfolge, die aus einer Aktion resultierenden nachfolgenden Zustände, sowie einen Index besitzen. Besonders das Array für die nachfolgenden Zustände ist wichtig, weil es die Auswahl der Zustände deutlich einfacher macht.

Dabei muss beachtet werden, dass in der ersten Dimension des Arrays eine „0“ für das Hinzufügen einer „1“ an der gewählten Stelle steht und eine „1“ für das Hinzufügen einer „2“. Dies folgt daraus, dass die Zustände nicht die tatsächliche Struktur des Spielfelds darstellen, sondern diese umwandeln, sodass für jeden Spieler die eigenen Zeichen mit „1“ du gegnerische Zeichen mit „2“ markiert werden. Auf diese Weise treffen die Spieler in derselben Situation dieselbe Entscheidung.

3.3.3. Der Code

Alles, was für die Funktionen des Spiels notwendig ist, ist in der Klasse „*field“* (siehe Anhang: QLearning -> TicTacToeQLearning) zu finden. Diese Klasse besitzt viele Methoden, die in drei Teile geteilt werden können: *Funktionen des Spiels, Generierung der Zustände* und *hilfreiche Funktionen*, die den Umgang erleichtern. Erstere Gruppierung beinhaltet lediglich *SetPosition()*[siehe Z. 123]. Diese Funktion hat die Funktion, das Feld mit dem übergebenen Index mit dem übergebenen Zeichen (repräsentiert durch 1 und 2) zu füllen, sofern das Feld frei ist, oder den Inhalt zu überschreiben, wenn die Erlaubnis über den übergebenen **bool** übergeben wird. Der Inhalt der Felder wird nur bei der Generierung der Zustände (siehe Kapitel 3.3.5) ignoriert. Im normalen Spielverlauf muss ebenfalls der aktuelle Zustand, der in der

Variablen State[] currentState gespeichert ist, geändert werden. Da *currentState[0]* die Ansicht von Spieler 1 darstellt, muss, wenn Spieler 1 eine Aktion ausführt, der Index der ersten Dimension „0“ sein, bei einer Aktion von Spieler 2 eine „1“. Genau umgekehrt verhält es sich bei *currentState[1]*. Der aktuelle Spieler (mit 1 bzw 2) wird in Form von **int** value der Funktion übergeben. Für Spieler 1 muss lediglich 1 subtrahiert werden, sodass man die Zuweisung 1 → 0 bzw 2 → 1 hat. Für Spieler 2 ist dies nicht so offensichtlich, aber man stellt fest, dass 2 →0 und 1→1 die Eigenschaften von Modulo 2 sind. Somit muss also der Wert für Spieler 2 % 2 genommen werden, um den richtigen Nachfolgezustand zu erhalten. Zuletzt wird zur Sicherheit überprüft, ob eine ungültige Aktion ausgeführt wurde, indem geguckt wird, ob für beide Spieler der Nachfolgezustand existiert. Ist dem nicht so, wird ein Fehler ausgegeben.

Zur Generierung der Zustände zählen deutlich mehr Funktionen, bei denen oftmals nur kleine Aufgaben übernommen werden. Die wichtigste Funktion ist die Funktion *GetCombinations(string, bool)*, mit der die Generierung der Zustände begonnen wird. Zunächst wird überprüft, ob bereits Kombinationen generiert wurden. In diesem Fall ist der **bool** gotCombinations true, und die Funktion wird beendet [Z.145f]. Andernfalls wird die Funktion weiter durchlaufen und es wird zunächst grob überprüft, ob der übergebene Pfad gültig ist, indem nach der Zeichenfolge „:/“ gesucht wird. Diese ist für jeden Pfad unerlässlich.

Sofern der Pfad gültig scheint, wird die angegebene Datei geöffnet und ein **StreamReader** für die Datei angelegt, mit dem aus der Datei ausgelesen werden kann [Z. 151f]. Mithilfe des StreamReaders wird nun in einer while-Schleife solange aus der Datei ausgelesen, bis NULL zurückgegeben wird. Die ausgelesenen Strings werden dann in einer Liste gespeichert.

Mit diesen Strings muss noch etwas weitergearbeitet werden, weshalb anschließend zu Z. 198 gesprungen wird.

Wird kein gültiger String eingegeben, werden die Kombinationen generiert. Dafür wird zunächst mithilfe einer sich selbst aufrufenden Funktion jeder mögliche Zustand generiert, in dem sich das Feld befinden kann, ohne die Spielregeln zu beachten. Dabei werden $3^9 = 19683$ Kombinationen generiert.

Dieser Vorgang funktioniert so, dass zunächst mithilfe der zuvor genannten Funktion *SetPosition()* mithilfe des Überschreibens jeweils die übergebene Position *layer* mit einem Wert von 0 bis 2 beschrieben wird. Anschließend wird die Funktion erneut

aufgerufen, wobei layer um 1 erhöht wird, sodass alle 9 Positionen (von 0 bis 8) beschrieben werden. Wird die Funktion mit *layer = 9* aufgerufen, wird die Funktion *GetCombination(0)* aufgerufen, welche die aktuelle Kombination aus Zahlen in Form eines Strings zurückgibt. Diese Strings werden dann in einem Array gespeichert. Anschließend ruft sich die Funktion nicht noch einmal selbst auf, sondern bricht ab, sodass in der Schicht darunter der Wert des dazugehörigen Felds um 1 erhöht wird. Ist dieser Wert 2, ist die Schicht darunter dran, und so weiter.

Auf diese Weise werden alle möglichen Zustände gefunden, sowie unmögliche Zustände. Diese werden ab Z. 167 entfernt. Zunächst sind unmögliche Zustände Zustände, bei denen die Differenz der Züge größer als 1 ist, denn um diesen Zustand zu erreichen, muss ein Spieler zweimal hintereinander am Zug sein, und das ist nicht möglich. Die Strings, die zu diesen Zuständen gehören, werden ihrerseits in einer Liste gespeichert. Anschließend werden alle Elemente, die in dieser Liste sind, aus der ersten Liste entfernt, sodass nur die möglichen Zustände übrig bleiben.

Hilfreiche Funktionen beinhaltet verschiedene Funktionen, die die Interaktion mit den Zuständen erleichtern. Dazu zählt zum Beispiel die Funktion *Printable*, die den übergebenen String eines Zustands in eine lesbare Form, also das 3x3 Feld umwandelt. Dazu gibt es aber auch noch andere Funktionen, die genau das machen, was der Name sagt: *GetReward* gibt die Belohnung des aktuellen Zustands zurück, *getCombination* gibt den Zustand als String zurück und *GetCurrentState* gibt den aktuellen Zustand aus einer bestimmten Sicht zurück. Diese Funktionen sind soweit selbsterklärend und bedürfen keiner weiteren Erklärung.

3.3.4. Änderungen am Lernvorgang

Eine Änderung am Lernvorgang ist die erzwungene Entscheidung. Da viele Zustände und demnach noch mehr Aktionen möglich sind, wird hier zur Unterstützung des Lernvorgangs jede mögliche Aktion in jedem möglichen Zustand geübt.

Anders als in dem Beispiel mit den Schaltern darf der Agent nicht zwei Entscheidungen nacheinander treffen, sondern muss sich mit seinem Gegner abwechseln. Aus diesem Grund muss der Abschnitt $R(s,a)+\gamma * \mathrm{MAX}(s' \mid 0, s' \mid 1, s' \mid 2)$ als Quality umdefiniert werden. Es geht hierbei nicht mehr NUR darum, dass mithilfe der MAX-Funktion der höchste Wert der nachfolgenden Aktionen gefunden werden soll, sondern es muss die Qualität des eigenen Zustands bestimmt werden, indem die Qualität des gegnerischen Zustands

beachtet wird. Die eigentliche Formel lautet also eher $Q(s,a,s')_{neu} = Q(s,a,s')_{alt} + \alpha * [Quality(s,a,s') - Q(s,a,s')_{alt}]$

Diese Quality kann nun unterschiedlich beschrieben werden. Eine Möglichkeit ist, zwei Zustände „in die Zukunft“ zu gucken, wenn der aktuelle Spieler wieder an der Reihe ist. Dafür wird der Zustand nach der aktuellen Aktion betrachtet und dann die Aktion, die der Gegner am wahrscheinlichsten wählt, ausgewählt. Dann wird der Zustand, der aus dieser Aktion folgen würde, ausgewählt und dann die MAX-Funktion für diesen Zustand betrachtet. Auf diese Weise kann ermittelt werden, wie gut die Aktionen sind, die dem Spieler als nächstes zur Verfügung stehen. Das einzige Problem an dieser Betrachtung ist die Tatsache, dass diese Überlegung von zwei variablen Werten abhängt: dem besten Wert des Wert des Gegners und dem besten Wert des nächsten eigenen Zustands. Ändert sich der Wert für den Gegner also, fehlt die Grundlage für den aktuellen Q-Wert. Damit kann nicht zuverlässig konvergiert werden, da sich die zugrunde liegenden Werte durchgehend ändern. Konvergiert das Q-Array nicht, ist damit die Lernmethode nicht mehr brauchbar.

Eine weitere und weitaus einfachere Möglichkeit ist es, die Quality als Gegenteil der Quality des gegnerischen Zustands zu betrachten. Genauer bedeutet das, dass wie zuvor auch die MAX-Funktion auf den nächsten Zustand angewandt wird (wobei die Sicht des Gegners gewählt wird, um die Entscheidung „aus seinen Augen“ zu treffen). Mit der Überlegung, dass ein besserer Wert für den Gegner schlechter für den aktuellen Spieler ist, kann nun eine von zwei Möglichkeiten ausgewählt werden: entweder wird mit -1 multipliziert, oder aber von dem höchsten Wert des Q-Arrays die Qualität des Zustands zuvor subtrahiert. Mit letzterer Methode wird verhindert, dass es negative Werte gibt, da per Definition mindestens 0 als Ergebnis der Subtraktion übrig bleibt. Gerade weil negative Werte vermieden werden, wird hier die zweite Möglichkeit gewählt. Negative Werte werden vermieden, da eine unmögliche Aktion einen negativen Q-Wert hat.

Eine letzte Änderung soll dafür sorgen, dass „unentschieden“ nicht für die Spieler attraktiver wird als zu gewinnen. Dafür wird mit jeder Aktion ein „Preis“ gefordert, sodass lange Spiele (die zu einem unentschieden führen) weniger attraktiv werden. Hat ein Spieler also in einem Zustand die Wahl, zu gewinnen, oder das Spiel zu verlängern, ist die Qualität der Aktion, die zum Sieg führt deutlich höher als die, die das Spiel verlängert. Die volle Formel lautet also:

$$Q(s,a,s')_{neu} = Q(s,a,s')_{alt} + \alpha * [[100 - (R(s,a) + \gamma * \mathrm{MAX}(s'|0, s'|1, s'|2)) - step] - Q(s,a,s')_{alt}]$$

Betrachtet man die Formel, kann man in Bezug auf das Lernverhalten feststellen, dass es drei verschiedene Variablen gibt, die das Lernverhalten beeinflussen: α, γ und *step*. Um den Einfluss darzustellen wird mit verschiedenen Werten trainiert und die Ergebnisse zusammengefasst.

3.4. Lerndaten

Allgemein gilt für die Legenden: Die Dateien werden folgendermaßen benannt:

γ * 100 – α * 10 – step.txt.

Auf der y-Achse wird die Summe der Siege durch die aktuelle Rundenzahl geteilt. Das bedeutet, je weniger Siege erlangt werden, desto niedriger ist der Graph. Insgesamt werden 4000 Daten von je einer Struktur a, g und Stepp dargestellt, wobei der Datensatz deutlich gekürzt wurde und nur etwa jeder 89-te Eintrag beachtet wurde, um das Diagramm sinnvoll darzustellen. Zur Interpretation ist wichtig, zu beachten, dass ein fallender Graph einer Ansammlung von Unentschieden entspricht, ein steigender Graph einer Ansammlung von Siegen. Anhand dieser Verläufe können Aussagen darüber getroffen werden, ob das Programm gelernt hat, ein Unentschieden zu erreichen und wo die Fehler liegen.

3.4.1. Veränderung der Kosten pro Aktion

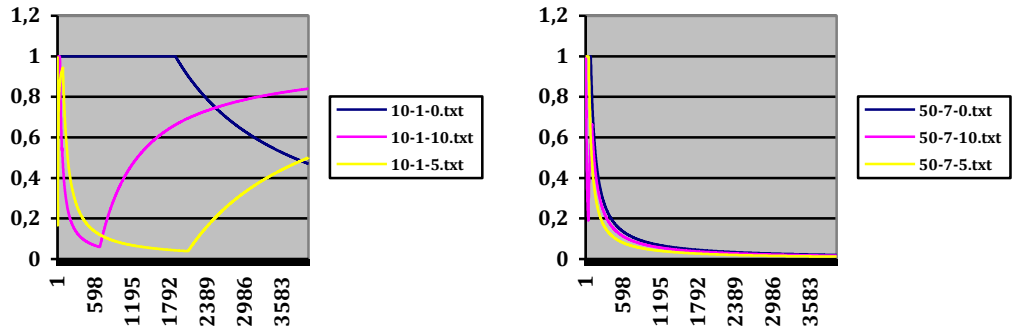


Diagramm 1 Veränderung der Kosten pro Aktion

In diesem Szenario wurden die Kosten pro Aktion variiert. Auffällig ist dabei, dass im linken Diagramm die Graphen, bis auf den mit einem stepPrice von 0, zunächst fallen, dann aber bis zum Ende der Simulation steigen. Im rechten Diagramm dagegen sind die Verläufe sehr ähnlich. Begründet werden kann das damit, dass bei niedrigem γ die Belohnungen in der Zukunft als deutlich weniger wichtig eingeschätzt werden als die Belohnungen, die unmittelbar bevorstehen. Für eine Strafe, bzw. einen negativen Preis ist das System gleich. Weil größere direkte Belohnungen und Strafen einen

größeren Einfluss als ferne oder geringe Belohnungen und Strafen, steigt auch der Graph mit Kosten pro Aktion von 10 deutlich früher als der Graph mit Kosten von 5.

3.4.2. Veränderung von γ

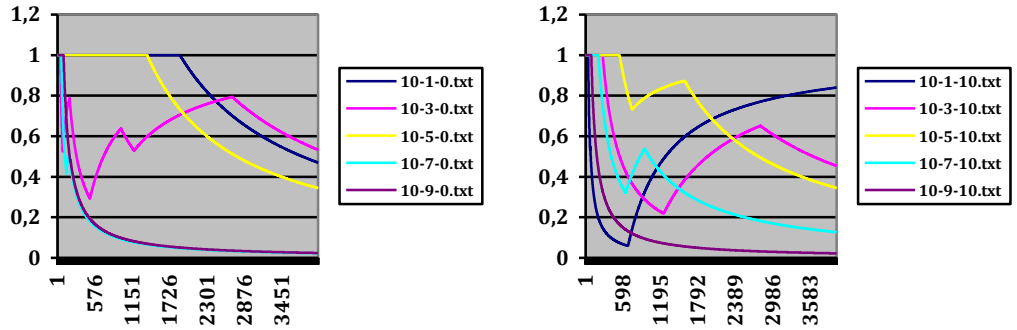


Diagramm 2 Veränderung von γ

Ebenso wie zuvor wurden zwei Parameter konstant gehalten. Dieses Mal wird γ variiert, wodurch auffällig ist, dass bei höherem γ die Graphen eher niedriger verlaufen. Das lässt sich ebenso wie zuvor damit erklären, dass zukünftige Belohnungen einen sehr hohen Stellenwert erhalten, sodass schnell ein erfolgreicher Weg gefunden werden kann. Dasselbe Phänomen ist auch bei hohen Strafen zu beobachten, wobei bei zu geringem γ die Belohnungen in der Zukunft als zu gering eingeschätzt werden. Aus diesem Grund lässt sich bei α = 0,1 und γ = 0,1 beobachten, dass der Graph zunächst fällt, später aber, durch die Strafen und ein zu geringes α, der beste Weg der ist, der zum Sieg führt.

3.4.3. Veränderung von α

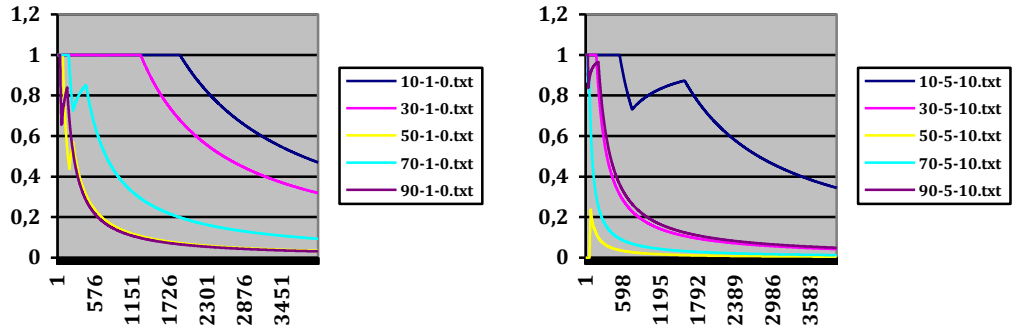


Diagramm 3 Veränderung von α

Für die Veränderung von α ist wichtig, dass ein größeres α dafür sorgt, dass die Differenz zwischen aktuellem und errechnetem Q-Wert stärker zu dem aktuellen Q-

Wert dazugerechnet wird. Dadurch treten zwei Phänomene auf: zum einen, und das kann man gut im linken Diagramm beobachten, wird ein optimales Ergebnis schneller erreicht. Es kann beobachtet werden, dass ca. ab Wert Nr. 2000 alle Graphen fallen, also eine Strategie zum Unentschieden gefunden wurde. Dabei sind jedoch die Graphen, die zu größerem α gehören, deutlich tiefer, da sie früher schon gefallen sind. Das zweite Phänomen ist, dass die Q-Werte um den optimalen Wert pendeln und so nicht die optimale Strategie ausgewählt werden kann, was man angedeutet am Graphen rechts mit γ = 0,1 sehen kann, da er zwischen 700 und 1700 wieder steigt.

3.5. Fazit

Die Lernweisen des Programms sind klar von den Parametern abhängig. Dabei sind je nach Ziel verschiedene Kombinationen sinnvoll. Klar ist aber auch, dass große α zu vermeiden sind und γ zwischen 0,5 und 1 sehr gut funktionieren.

Im Allgemeinen ist das Programm definitiv in der Lage, in verschiedenen Szenarien aus Zuständen den besten Weg zu wählen. Begrenzend wirken dabei die Anzahl der Zustände und der Aktionen, da bei einer höheren Zahl der Zustände oder einer höheren Zahl möglicher Aktionen auch die Anzahl der Durchgänge steigt, die benötigt wird, um alle Kombinationen zu durchlaufen, womit auch die Trainingszeit verlängert wird. Im Falle des TicTacToe fanden ca. 358.120 Trainingsdurchgänge mit je 9 Simulationen in 16 Minuten statt. Ein weiteres Beispiel für Q-Learning und vor allem dessen Limits ist der Rubik's Cube, der in der 2x2x2 Ausführung ca. 3.674.160 Zustände hat. Dazu kommen insgesamt 6 Seiten * 2 Richtungen, die durch 2 geteilt werden müssen, da oben nach rechts und unten nach links dasselbe Ergebnis liefert, = 6 Aktionen.

Es ergibt sich damit eine Zustandszahl für diesen Rubik's Cube, die $\frac{3674160}{8953} \approx 410{,}38$ mal so groß ist. Das dazugehörige Q-Array zu berechnen, würde dementsprechend ca. 410 Mal so lange dauern, was pro Parameterkombination eine Lerndauer von ca. 4,56 Tage oder 109 Stunden ergibt.

Daran sieht man, dass bei großen Zahlen das Q-Learning schwächelt. Dazu kommen auch die Speicheranforderungen, die je nach Anwendungen mehrere Gigabyte beinhalten können. Das Konzept dagegen ist einfache Mathematik, weshalb die Implementation des Algorithmus in ein Programm nicht sonderlich schwer ist. Aus diesem Grund steckt hier großes Potential, dass mit einer Möglichkeit, die Anzahl der Kombinationen zu verringern, in der Lage wäre, auch größere Probleme zu lösen.

Diese Möglichkeit gibt es tatsächlich bereits. Sie nennt sich Deep Q-Learning und kombiniert Q-Learning mit einem Neuronalen Netz, sodass die Formel des Q-Learning Algorithmus ein NN trainiert, dieselben Werte wie das Q-Array zurückzugeben. Wie bereits im ersten Kapitel festgestellt, sind NNs in der Lage, große Mengen an Daten zu verarbeiten und Gemeinsamkeiten zu erkennen, sodass, anders als beim Q-Learning, nicht jeder kleine Unterschied für einen vollständig neuen Zustand sorgt.

4. DEEP Q-LEARNING (F.H. ECKHOFF)

4.1. Das Problem

Wir haben jetzt also zwei KI's, das eine kann zufällige Aktionen machen und wird mit der Zeit immer besser, braucht aber sehr lange und kann auch nur begrenzt viele Informationen verarbeiten, das Andere kann die optimale Lösung finden, aber auch dort nur für eine begrenzte Anzahl an Wegen. Was also tun um eine sehr Komplexe Umgebung möglichst schnell und optimal zu lernen?

4.2. Die Lösung: Deep Q-Learning

Das Prinzip des Deep Q-Learning verwendet die besten Eigenschaften von beidem, sowohl Neurale Netze als auch Q-Learning. Neurale Netze sind sehr gut darin, Informationen zu verarbeiten. Allerding müssen diese Netze mit irgendwas trainiert werden. Da kommt das Q-Learning ins Spiel. Es werden die letzten paar Zustände gespeichert und wenn ein Punkt gemacht wurde, werden diese dann mithilfe der Formel dann als Trainingsdaten in die Backpropagation des Netzes gegeben. Die genaue Funktionsweise des hier verwendeten Netzes ist sehr komplex, weshalb sie im Folgenden kurz und stark vereinfacht erklärt wird.

4.2.1. Conv-Net

Die erste "Station" des Deep Q-Learnings ist das sogenannte „convolutional Network". Es dient normalerweise der Bilderkennung und funktioniert ähnlich wie das normale Neural Net, allerdings sind die Neuronen (hier Filter) gleichzeitig die weights. Diese filtern das das Eingangsbild nach bestimmten Mustern z.B. Strichen oder Kurven. Das nächste Layer filtert dann nach Kombinationen dieser Muster z.B. Ecken oder Kreise. Dies geht so weiter bis es dann normalerweise im letzten Layer die Muster z.B. „Hund" oder „Katze" sind. Hier wird jetzt an dem Bespiel nicht nach Tieren gesucht, sondern nur die Situationen klassifiziert.

4.2.2. Fully connected Layer

Die bis hier hin auf ein Format von 1x1 gebrachten Bilder werden dann als normaler Input in ein Fully connected Layer gegeben (Feed Forward). Dieses entscheidet dann aufgrund der Klassifizierung welche Aktion gemacht wird.

4.2.3. Q-Learning

Wenn nach ein paar Aktionen ein Punkt erzielt wurde, muss beachtet werden, dass wahrscheinlich die Aktion kurz vor Erzielung des Punktes nicht die einzig wichtige Aktion war, sondern dass auch die 10,20 oder 30 Aktionen vorher wichtig waren. Aus diesem Grund wird mithilfe von Q-Learning einigen vorherigen State-Aktion Paaren eine absteigende Wertigkeit zugeordnet. Diese wird dann rückwärts durch beide Netze gegeben um die Weights und Filter anzupassen.

4.3. Pong

Diese AI wird sich an dem Spiel „Pong“ versuchen. Der ursprüngliche Plan war eine Internet Seite als Quelle des Spiels zu verwenden, dies scheitert aber an einigen Problemen mit der Seite. Die Alternativlösung war dann sogar die bessere: ein eigenes Spiel schreiben. Dies bietet den Vorteil, dass man die „externe“ Deep Q-Learning (kurz DQL) AI mit einer eingebauten AI vergleichen kann.

4.3.1. Anwendung der AIs

Die Eingebaute funktioniert mit einem einfachen Neuralen Netzwerks, welches als Inputs die eigene y-Position und die x,y-Position des Balles bekommt.
Die DQL-AI erstellt von dem gesamten Spiel einen Screenshot und verarbeitet diesen. Um festzustellen ob ein Punkt erzielt wurde, und wenn ja, für wen, wird eine einfache Bilderkennung an der Position des Punktestands durchgeführt.

4.4. Auswertung

Die Auswertung der zu erkennenden Ergebnisse ist nicht ganz eindeutig. In den gemachten Versuch ist das Endergebnis nach ca. 1h 154:77 für die eingebaute AI. Es ist aber auch eindeutig ein aktiver Lernprozess zu erkennen, bei dem sich die DQL-AI aktiv zum Ball bewegt. Dieser tritt jedoch deutlich langsamer auf als bei der Eingebauten. Diese Verzögerung ist sehr wahrscheinlich auf die deutlich höhere Anzahl von Variablen bei dem DQL gegenüber der normalen AI zurückzuführen.
Es ist aber bemerkenswert, dass es tatsächlich möglich ist, einem Programm die Verbindung von Pixeln auf einem Bildschirm und einem abstrakten Konzept wie „Ball abwehren“ beizubringen.

Ein weiteres Problem ist Geschwindigkeit. Während die eingebaute AI selbst ohne Multithreading in ca. 0.005 Sekunden mit einer Entscheidung fertig ist und sogar künstlich verlangsamt werden muss, braucht die DQL-AI selbst mit Multithreading ca. 0.3 Sekunden. Dies reicht gerade für eine langsame Runde Pong aus, ist aber definitiv nicht geeignet für ein schnelles Spiel.

4.5. Fazit

Das Fazit, welches sich hier raus schließen lässt, ist ein deutliches Signal für die Entwicklung der AI. Nicht nur ist es innerhalb von wenigen Wochen möglich, ohne jeglichen Fremdcode eine komplette AI zuschreiben, die quasi vor jeden Bildschirm setzen kann und dann lernen wird, sondern ist dies sogar mit wenig Kenntnis von komplexen Formeln oder tiefem Verständnis von Programmierung möglich. Genau diese simple Möglichkeit, sich mit den Vorgängen ohne große Vorkenntnisse zu befassen, macht dieses Thema zukunftsorientiert.
Und gerade weil dem Menschen ein Großteil seiner Arbeit auf diese Weise abgenommen wird, ist zu erwarten, dass innerhalb der definierten Freiheiten die KI in allen Bereichen die Fähigkeit besitzt, besser als der Mensch zu sein. Der einzige Grund, sich vor KI zu fürchten, ist nur, wenn man davon ausgeht, ersetzt zu werden. Um eine zu abrupte gesellschaftliche Umstrukturierung zu verhindern, ist es deshalb wichtig, dass beim Einsatz von KI darauf geachtet wird, dass es in einem Rahmen geschieht, in dem die KI kontrolliert werden kann. Deshalb gehört zum Erfolg der KI ein Auge auf die Sicherheit.
Doch auch Scheitern und Neuanfangen gehört dazu, wie wir auf unserem Weg häufig feststellen durften. Deshalb soll diese Arbeit mit einem Zitat des 1955 verstorbenen Albert Einsteins enden:

> *Zwei Dinge sind zu unserer Arbeit nötig: Unermüdliche Ausdauer und die Bereitschaft, etwas, in das man viel Zeit und Arbeit gesteckt hat, wieder wegzuwerfen*[14]

[14] Steffler, Ralf. *Die Widersprüche der modernen Physik*. Ausgabe 5. Seite 300. [s.l.] : Verlag BoD – Books on Demand

5. DEFINITIONEN

Agent:	Agent ist eine im Bereich des MachineLearnings gebräuchliche Bezeichnung für ein Programm, das Aufgaben erfüllt. Zusammen mit künstlicher Intelligenz kann von einem intelligenten Agenten (engl. "intelligent agent") gesprochen werden, der zu einem bestimmten Grad "Autonomität, Mobilität, ein symbolisches Verständnis der Wirklichkeit“[15] besitzt, und “ die Fähigkeit, aus Erfahrung zu lernen und mit anderen Agenten und Systemen zu kooperieren"[16].
Markov Decision Process:	Bei einem stochastischen Prozess, der als MDP bezeichnet wird, handelt es sich um eine Verbindung von Zuständen und Aktionen. Diese besitzen Übergangswahrscheinlichkeiten, welche „nicht von dem Zeitpunkt […] und nur vom Zustand, in dem sich das System unmittelbar vor dem Zustandswechsel befindet, abhängen“[17]
Dictionary	Ein Dictionary (dt: Wörterbuch) ähnelt dem Prinzip eines Arrays. Der Unterschied ist, dass das Dictionary als index einen Wert annehmen kann, der kein Integer ist. Dieser neue Index wird *key* (dt: Schlüssel) genannt. Voraussetzung für einen erfolgreichen Zugriff ist, dass der *key* zuvor definiert wurde.

[15] Zwass, Vladimir. ”Agent, computer science”. *Encyclopedia Brittanica*. 17. November 2000, 11. November 2018 <https://www.britannica.com/technology/agent>

[16] Ebd.

[17] „Markov’sche Prozesse“. *Das große Tafelwerk.* 1. Aufl. 2014

6. LITERATURVERZEICHNIS

Klose, Olivia. “Machine Learning (2) - Supervised versus Unsupervised Learning”. *Olivia’s Blog*. 24. Februar 2015. 11. November 2018 <http://www.oliviaklose.com/machine-learning-2-supervised-versus-unsupervised-learning/>

„Markov’sche Prozesse“. *Das große Tafelwerk.* 1. Aufl. 2014

Matiisen, Tambet. „Demystifying Deep Reinforcement Learning”

Steffler, Ralf. *Die Widersprüche der modernen Physik.* Ausgabe 5. Seite 300. [s.l.] : Verlag BoD – Books on Demand

Tokic, Michel. „Adaptive ε-greedy exploration in reinforcement learning based on value differences“. Tokicblog. 2010. 11. November 2018. <http://www.tokic.com/www/tokicm/publikationen/papers/AdaptiveEpsilonGreedyExploration.pdf>

Zwass, Vladimir. ”Agent, computer science”. Encyclopedia Brittanica. 17. November 2000, 11. November 2018 <https://www.britannica.com/technology/agent>

7. ANHANG

7.1. Genutzte Konzepte

7.1.1. Klassen

```
18  class Auto{
19      public:
20          int laenge;
21          int breite;
22          string marke;
23          void beschleunigen(){
24              currentGeschwindigkeit++;
25              beschleunigungAnzeigen();
26          };
27
28      private:
29          int currentGeschwindigkeit;
30          void beschleunigungAnzeigen(){
31              //irgendwie zeigen, dass das auto beschleunigt wird
32          };
33  };
34
35
36  int main(){
37      Auto meinAuto;
38      meinAuto.beschleunigen();
39
40  }
```

In dieser Facharbeit werden mehrere Programme vorgestellt, die in den objektorientierten Programmiersprachen ‚C++' und ‚C#' geschrieben wurden. Ein Feature von C++ und C# ist die Verwendung von Klassen und Objekten. Klassen sind Datenstrukturen in diesen Programmiersprachen, In denen zusammengehörige Daten gebündelt werden. Zum Beispiel soll eine Straße simuliert werden. Dafür braucht man mehrere Autos. Hier in diesem Beispiel fängt man damit an, eine Klasse namens ‚Auto' zu deklarieren. Jede Klasse hat einen Public und einen Private Teil, zwischen denen der Unterschied ist, dass auf den Public Teil von außen zugegriffen werden kann. Das kann genutzt werden, um klar zu stellen, was in dem Programm erlaubt sein soll. Zum Beispiel soll die Geschwindigkeit ‚currentGeschwindigkeit' nicht direkt bearbeitet werden, da normalerweise ein komplexer Algorithmus dahinter steht. Nur Funktionen innerhalb des Klasse können auf den Wert zugreifen. In dem public Teil werden Daten wie zum Beispiel die Länge und Breite gespeichert, auf diese Werte können alle Funktionen in dem gesamten Programm zugreifen.

Es lässt sich ein Objekt von einer Klasse stellen. Dies wird außerhalb der Klasse gemacht und geschieht mit $Klassenname ‚Objektname'. Auf Funktionen innerhalb einer Klasse kann man mit Objektname.Funktionenname zugreifen. Der Vorteil dieser Methode ist, dass alle Werte und Funktionen, die zu einer Sache gehören. gebündelt sind. Es ist nicht unbedingt nötig, nimmt dem Programmierer allerdings viel Arbeit ab, da er sich sonst merken müsste welche Variable zu welchem abstrakten Objekt gehört.

7.1.2. Verkettete Listen

Unter einer verketteten Liste versteht man einen Datentyp, der eine große Anzahl an einzelnen Daten speichern kann. Man kann es sich tatsächlich wie eine Liste auf einem Blatt Papier vorstellen. Man kann am Ende weitere Einträge hinzufügen und in der Mitte welche löschen. Der Unterschied ist, dass die Einträge nicht unbedingt in einer Reihenfolge sein müssen. Es ist ausreichend, wenn an jedem Eintrag ein „Pfeil“ angehängt ist, der auf den nächsten Wert weist. Ein weiterer Unterschied ist, dass beim Programmieren der Typ der Daten immer in der Deklarierung mit angegeben werden muss. Am Beispiel von C++:

```
vector<int> eineListeVonInts;
```

Hier ist vector die Bezeichnung für verkettete Listen, ‚<int>‘ gibt den Datentyp an(in diesem Fall int, also Ganzzahl), und eineListeVonIntsder Name der Liste.

8. ERKLÄRUNG

Hiermit versichern wir, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt haben. Alle wörtlich oder sinngemäß den Schriften anderer entnommenen Stellen haben wir unter Angabe der Quellen kenntlich gemacht. Dies gilt auch für beigefügte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen. Uns ist bewusst, dass wir uns im Falle einer unbeabsichtigten oder vorsätzlichen Missachtung durch den fehlerhaften Umgang mit Quellen unter Umständen strafbar machen und die vorliegende Arbeit mit „ungenügend“ bewertet wird. Uns ist bewusst, dass unsere Arbeit mittels geeigneter Software auf Plagiate durchsucht werden kann.

Jever, den
Unterschriften

Hiermit erklären wir unser Einverständnis, dass diese Facharbeit der schulinternen Öffentlichkeit zugänglich gemacht werden kann. Dabei beinhaltet die veröffentlichte Fassung nicht den Anhang, der den Code enthält.

Jever, den
Unterschriften