VISUALISIERUNG EINESLABYRINTHSOLVERS AUF GRUNDLAGE EINES NEURONALEN NETZWERKS IN JAVA

Amin Boyaala

Informatik, Herr Faßbender, LK IF, 2019/2020

Inhaltsverzeichnis

Vorwort

Da ich erst relativ spät Fortschritte und Ergebnisse hatte, hatte ich keine Zeit eine GUI zu programmieren/ mich in GUI Programmierung reinzuarbeiten, das ist auch der Grund warum das Thema meiner Facharbeit nicht mit dem festgelegten übereinstimmt. Beinah alle Abbildungen sind Freihandzeichnungen und sind auf einem externen Blatt festgehalten. Diese Seiten sind nicht nummeriert und sind nicht im Inhaltsverzeichnis enthalten. Die Abbildungen sind Zeichnungen auf Basis von „Neural Networks: A Visual Introduction for Beginners by Michael Taylor“.

Einleitung

In dieser Facharbeit befasse ich mich mit dem Lösen eines Labyrinthes mittels eins neuronalen Netzwerks. Antrieb dafür war mein Wunsch mich mehr mit dem Thema maschinellen Lernen und Künstliche Intelligenz auseinanderzusetzen. Diese Facharbeit war also die ideale Möglichkeit mich in dieses Thema reinzuarbeiten.

Ich habe mich schon vorher sehr für neuronale Netzwerke interessiert und habe mir überlegt auf welches Problem ich es anwenden könnte. Ich habe mich für das Lösen eines Labyrinthes entschieden, da dies ein gute Anwendung für künstliche Intelligenz ist. Kann ich eine künstliche Intelligenz entwickeln, die in der Lage ist, ein Labyrinth zu lösen? Mit dieser Frage im Hinterkopf habe ich gestartet mich mit neuronalen Netzwerken auseinanderzusetzen. Mein Grundwissen kommt aus dem Buch „Neural Networks: A Visual Introduction for Beginners by Michael Taylor“ von Michal Taylor.

Was ist ein neuronales Netzwerk

Ein neuronales Netzwerk („Neural Network“, NN) oder auch künstliches neuronales Netzwerk („Artificial Neural Network“, ANN) genannt, ist ein maschineller Lernalgorithmus, es ist vom menschlichen Gehirn inspiriert.

Was ist maschinelles Lernen

Maschinelles Lernen ist ein Unterbegriff der künstlichen Intelligenz und dient dazu Programme zu Handlungen zu bringen, zu die es zu anfangs gar nicht programmiert wurde.

Maschinelle Lernalgorithmen verarbeiten Daten, um später Vorhersagungen zu machen oder Entscheidungen zu treffen. Einzelne Datenpunkte dieser Daten beinhalten eine Reihe von Attributen, also eine Dimension von Information. Als Beispiel für solche

Daten wäre das „Iris Data Set“[1], diese Daten beinhalten 150 Datenpunkte, die jeweils die Blütenblattlänge, die Blütenblattbreite, die Kelchblattlänge und die Kelchblattbreite speichert. Der fünfte Wert, also dem Ergebnis, ist die Klasse der Iris Pflanze. Die Lernalgorithmen trainieren mittels dieser Daten und können später Vorhersagungen zu neuen Daten aufstellen. So könnte ein trainiertes Modell mittels der Längen und Breiten der Blütenblätter und der Kelchblätter sagen welche Klasse von Irispflanze vorliegt.

Das oben genannte Beispiel ist ein Klassifizierungsproblem, hier stellt man sich die - Frage welcher Klasse die jeweiligen Datenpunkte zugehören. Maschinelles Lernen kann auch für Clusteranalyse und Regression genutzt werden.

Überwachtes, unüberwachtes und bestärkendes Lernen

Beim maschinellen Lernen gibt es verschiede Klassen. Überwachtes, unüberwachtes und bestärkendes Lernen. Beim Überwachten Lernen kriegt der Lernalgorithmus neben den Attributen, das richtige Ergebnis zum Trainieren. Ein neuronales Netzwerk zählt unter dieser Kategorie.

Beim unüberwachten Lernen erhält der Lernalgorithmus nur die Attribute. Der Algorithmus soll selbst Muster, Strukturen und Ausreißer in den Daten finden. Das Unternehmen Darktrace[2] nutzt unüberwachtes Lernen für die Sicherheit der Unternehmen seiner Kunden.

Beim bestärkenden Lernen erhält der Algorithmus Belohnungen und Strafen je nachdem welche Entscheidungen er getroffen hat. Der Algorithmus strebt nach der größtmöglichen Belohnung und erstellt sich eine Strategie, um diese zu erreichen.

Aufbau neuronaler Netzwerke

Neuronale Netzwerke sind von Gehirnen inspiriert. Genauso wie Gehirne Neuronen und Synapsen haben, besteht ein neuronales Netzwerk aus Nodes und Edges. Die Nodes sind in drei Schichten (Layers) eingeteilt: Den Input Layer, den Hidden Layer und den Output Layer. Jede Node aus einem Layer ist mit allen anderen Nodes des nächsten Layers über einer Edge verbunden (siehe Abb.1) und umgekehrt. Jede Node des Input Layers erhält ein Attribut des derzeitigen Datenpunktes. Jede Edge hat ein Gewicht („weight“) und verbindet zwei Nodes miteinander. Jede Node des Hidden und des Output Layers besitzen den Gesamtinput aus dem vorherigen Layer und eine

[1] Iris Data Set, https://archive.ics.uci.edu/ml/datasets/Iris, 26.04.2019, Creator: R.A Fisher, Donator: Michael Marshall
[2] unsupervised learning security company, https://www.darktrace.com/de/technologie/, 26.04.19

Aktivierungsfunktion. Sowohl der Hidden als auch der Output Layer besitzen je ein eine weitere Node, namens Bias, die mit jedem Node, des jeweiligen Layers über eine Edge verbunden ist.

Funktionsweise neuronaler Netzwerke

Bei einem neuronalen Netzwerk reisen die Daten von links nach rechts. Erst werden die Daten in den Input Layer eingesetzt, sodass jede Node des Input Layers je ein Attribut aus den Daten enthält. Im nächsten Schritt werden die Daten über die Edges zum nächsten Layer geschickt. Dabei werden die Daten mit dem Gewicht der Edges multipliziert. Jede Node des Input Layers schickt ihr Attribut multipliziert mit dem jeweiligen Gewicht zu jeder Node des Hidden Layer. Im Hidden Layer werden der Input der jeweiligen Nodes Net Input (der gesamte Input einer Node) genannt und durch ein $\sum$ dargestellt. Zum Net Input einer Hidden Node gehört noch der Output des Bias. Der Bias hat im Normalfall den Wert 1, der Output des Bias ist also 1 multipliziert mit dem Gewicht der jeweiligen Edge. Es wird zum Net Input des Input Layer lediglich das Gewicht der Edge vom Bias zur Node addiert (siehe Abb.2). Der Net Input der Node wird nun in die Aktivierungsfunktion eingesetzt. Das ist eine Funktion die den Net Input als eine Zahl in einem Intervall abbildet, sie wird durch ein σ dargestellt. Dies ist der Output der jeweiligen Node. Die Hidden Node gibt diesen Wert an alle Nodes des nächsten Layer, dem Output Layer, weiter. Diese werden wieder mit dem jeweiligen Gewicht verrechnet und an den Output Nodes (Nodes im Output Layer) weitergegeben. Dies addiert mit dem jeweiligen Bias ist der Net Input. Der Output des neuronalen Netzwerkes ist der Output der Output Nodes.

Lernprozess neuronaler Netzwerke

Ich möchte ihnen nun den Lernprozess erklären, deswegen werden wir nun diesen jetzt gemeinsam durchgehen

Das Lernen bei neuronalen Netzwerken liegt in dem Updaten der Gewichte der Edges. Zuerst rechnen wir den Error mittels einer Kostenfunktion (genaueres dazu später) aus. Die Kostenfunktion stellt die Abweichung unseres Netzwerkes vom richtigen Ergebnis als Zahl dar. Danach möchten wir herausfinden, wie die einzelnen Gewichte des neuralen Netzes diesen Error beeinflussen. Dafür berechnen wir die partiellen Ableitungen der einzelnen Gewichte (siehe Abb.3). Um dies auszurechnen nutzen wir die Kettenregel, da man mit dieser die partielle Ableitung einer verschachtelten Funktion ausrechnen kann, was gerade vorliegt, denn ein Neuronales Netzwerk ist

nichts anderes als eine große verschachtelte Funktion mit vielen Parametern. Mittels der Kettenregel berechnen wir die partielle Ableitung der einzelnen Gewichte des neuronalen Netzwerkes, mittels diesen wissen wir, wie das Gewicht verändert werden muss, damit die Differenz zwischen Output und Target verkleinert wird. Das tun wir, indem wir die berechnete partielle Ableitung, auch Gradient genannt, vom eigentlichen Gewicht abziehen. Unser Ziel ist es den geringstmöglichen Error zu erreichen. Um den Gradienten/ die partielle Ableitung für ein Gewicht herauszufinden, müssen wir die Kettenregel mehrfach, vom Output bis zum gewünschten Gewicht anwenden. Wir berechnen die partielle Ableitung vom Error im Verhältnis zum Output, vom Output zum Net Input und vom Net Input zum Gewicht (siehe Abb.4). Das Produkt der partiellen Ableitung ist der Gradient unserer jeweiligen Gewichte. Dieser mit der „Learning Rate“ multipliziert, wird vom Gewicht abgezogen, dass ist das neue Gewicht der Edge. Dieses Optimierungsverfahren wird Gradient Descent und Backpropagation genannt. „Learning Rate“ ist ein Parameter für neuronale Netzwerke. Dieser Parameter sagt unserem neuronalen Netzwerk, wie wichtig es jede Trainingseinheit nimmt. Wenn wir eine niedrige „Learning Rate“ wählen, spielen die einzelnen Trainingsrunden eine kleinere Rolle beim Korrigieren unserer Gewichte. Es gibt noch andere Parameter, aber wir brauchen für mein Projekt nur diese zu kennen.

Umsetzung

Der Plan ist es eine KI zu programmieren, die auf Basis eines neuronalen Netzwerkes sich durch ein Labyrinth manövriert. Ich programmiere in Java, eine objektorientierte Programmiersprache. Ich starte nun mit den Klassen für das Projekt.

Zuerst das Herzstück des Projektes, ein neuronales Netzwerk. Ich habe mich für ein neuronales Netzwerk mit drei Hilden Layer entschieden. Ein neuronales Netzwerk besteht aus mehreren Bestandteilen, wie Layers, Nodes und Edges. Diese Bestandteile habe ich als private Klassen in der Klasse des neuronalen Netzwerkes programmiert. Diese wären die Klasse Edge, die abstrakte Klasse Node und deren Unterklassen InputNode, HiddenNode und OutputNode. Die abstrakte Klasse Node habe ich benötigt damit die Klasse Edge mit allen drei Node Typen kommunizieren kann, ohne dass ich für jede Kombination eine extra Klasse anfertigen müsste. Zusätzlich habe ich noch eine private Function Klasse, die die Aktvierungsfunktion speichert.

Die Klasse Edge besitzt zwei Verweise auf Nodes, namens left und right. Diese Variablennamen spielen auf der Reise der Daten an (von links nach rechts). Neben

diesen Referenzen, speichert eine Edge ihr jeweiliges Gewicht, die Learning Rate und den Gradienten für das Gewicht. Der Gradient wird extra gespeichert und noch nicht direkt vom Gewicht abgezogen, da das aktuelle Gewicht noch für die Gradienten der nächsten Gewichte benötigt wird. Für den Transfer der Daten habe ich zwei Variablen angelegt, input und value. Die input Variable speichert den puren Input aus der left Node und value speichert den output der Edge, also den Input multipliziert mit dem Gewicht. Die Learning Rate wird jeder privaten Klasse beim Konstruktor übergeben. Die Methoden input, output, gradient und weightUpdate sind die Methoden, die die jeweiligen Werte berechnen/updaten und die Getter und Setter Methoden werden für den Transfer der Daten benötigt.

Die Klasse Node und deren Unterklassen besitzen zwei Arrays von Edge Objekten, wieder left und right. Diese speichern die Edges die zu den Nodes des anderen Layers verweisen. Ich benutze Arrays zum Speichern der Edges, da die Anzahl der gespeicherten Edges sich während der Laufzeit nicht ändern und nur einmal festgelegt werden. Genauso wie die Edge Klasse speichern auch die Nodes die Learning Rate. Das auch die Nodes die Learning Rate speichern, ist nötig für die Bias Edge. Der Bias und die Bias Edge speichere ich in einer einzelnen Variablen, anstatt in eine extra Node und Edge. Der Bias behält gewöhnlich den Wert 1 und daher spielt nur das Gewicht der Edge eine Rolle für den Net Input. Jede Node besitzt für den Datenverkehr eine netInput Variable und eine value Variable. Außerdem gibt es noch eine gradient und eine biasGradient Variable.

Die InputNode nutzt den Edge Array left, den bias, den biasGradient und die learningRate Variable nicht, dafür besitzt sie eine extra Variable, den input. Die InputNode erhält seinen Input von außen und nicht durch die Edge Klasse, wie die anderen zwei Node Unterklassen. Die Variablen input, netInput und value nutze ich alle für den gleichen Wert, der Input verändert sich nicht in der InputNode. Dennoch habe ich es für die Einheitlichkeit so gelassen.

Die HiddenNodes nutzen alle Variablen. Der netInput speichert den Output aller Edges vom vorherigen Layer, der durch die Edges gelaufen sind. Also alle Attribute multipliziert mit dem jeweiligen Gewicht zusammenaddiert. Der value ist dann das Ergebnis der Aktivierungsfunktion vom netInput. Als Aktivierungsfunktion nutze ich die Sigmoid Funktion[3] (siehe Abb.5), welche jegliche Zahl in eine Zahl zwischen 0 und

[3] Eingeführt von Pierre François Verhulst

1 umwandelt. Dafür rufe ich die statische Methode activation auf, denn diese berechnet für mich die Sigmoid Funktion. In der gradient Variable speichere ich den Teil der Kettenregel der für alle Gewichte vor dieser, also links von dieser HiddenNode, gleich ist (vgl. Abb.6). Der biasGradient ist der Gradient für das Gewicht der Edge des Bias. Dieser entspricht dem berechneten Wert der gradient Variable.

Ich programmierte die OutputNode ähnlich wie die HiddenNodes, jedoch besitzt die OutputNode kein Edge Array right, da der Output der OutputNode der endgültige Output des neuronalen Netzes ist. Ich habe der OutputNode noch eine extra Variable namens target gegeben, diese speichert das gewünschte Ergebnis der Trainingseinheit und nutzt diese für die Kettenregel. Auch diese Nodes nutzen die statische activation Methode aus der Function Klasse auf.

Ich nutze Methoden, die meistens gleichnamig mit den jeweiligen Variablen sind, um die Werte für die verschiedene Variablen auszurechnen/upzudaten. Der Rest sind selbsterklärende Getter und Setter Methode.

Die Function Klasse benutze ich, um die anderen Klassen mit der Sigmoid Funktion zu versorgen. Da ich die Sigmoid Funktion benutze, gibt mein neuronales Netzwerk Werte zwischen 0 und 1 heraus.

Ich speichere die einzelnen Layers des neuronalen Netzwerks in Arrays der Input- Hidden- und OutputNode Klasse. Die Edge Objekte verwalte ich über einen zweidimensionalen Array. Der Input Layer heißt a, die drei Hidden Layer nannte ich b, c und d und der Output Layer heißt e. Die Arrays für die Edge haben eine Kombination der Layer die sie verbinden als Name, so heißt das Array der Edge Objekte zwischen den Input und den ersten Hidden Layer ab. Dazu speichere ich den Input, den Output, das erwünschte Ergebnis Target, den gesamten Error und der Error der einzelnen Output Nodes, also der lokale Error in double Variablen und double Arrays. Auch das neuronale Netzwerk speichert die Learning Rate, da er diesen den einzelnen Objekten der Klasse Node und Edge übergeben muss. Im Konstruktor werden die Anzahl der Nodes für die einzelnen Layer und die Learning Rate übergeben. Im Konstruktor generiere ich auch alle Nodes und Edges, in den Nodes werden auch schon die Arrays für die Edge Objekte angelegt. Danach wird die connect Methode aufgerufen, diese verbindest nun alle Nodes mit allen Edges. Mein neuronales Netzwerk hat drei öffentliche Methoden. Die train Methode, die predict Methode und die printError Methode. Die train Methode verlangt als Parameter die Trainingsdaten und den

gewünschten Output. Das neuronale Netzwerk macht eine Vorhersage (mittels der forward Methode), berechnet den Fehler (mittels der calcError Methode) und korrigiert die eigenen Gewichte mit Hilfe von Backpropagation (backward Methode). Die predict Methode verlangt als Parameter nur Daten und kein Output und gibt eine Vorhersage zurück.

Mein neuronales Netzwerk nutzt als Aktvierungsfunktion die Sigmoid Funktion und zum Berechnen des Errors die Kostenfunktion MSE („Mean Squared Error“, siehe Abb.7). Ich nutze keinen weiteren Parameter für mein neuronales Netzwerk, bis auf die Learning Rate.

Kommen wir als nächstes zur Umgebung, dem Labyrinth. Die Klasse Environment speichert sich das Labyrinth als zweidimensionales double Array. Dazu speichert es die maximale Breite und Länge des Labyrinthes als Integer. Die Koordinaten des Spielers, der sich durch das Labyrinth manövriert wird in einzelnen Variablen gespeichert. Eine weitere Variable namens hidden speichert die schon gegangen Schritte. Überschreitet dieser Wert die Anzahl aller Felder im Labyrinth, dann gilt der Player als verirrt. Die Methode „lose“ gibt dann bei nachfrage true oder false zurück. Neben dem normalen Labyrinth gibt es noch ein gelöstes Labyrinth. Dieses zweidimensionale double Array speichert in jedem Feld die Distanz zum Ziel. Wände und unerreichbare Felder werden als -1 gespeichert. Hier habe ich die Breitensuche angewandt und sie rekursiv implementiert. Die Variable r dient als Schwellenwert. Wenn ich ein Labyrinth erstellen will, gehe ich die einzelnen Felder des 2D-Arrays durch und für jedes Feld erschaffe ich eine Zufallszahl zwischen Null und Zehn. Ist die Zufallszahl über den Schwellenwert r, erhält das Feld den Wert eines freien Feldes (space = 0), ist sie unter dem Schwellenwert r erhält das Feld den Wert einer Wand (wall = -1). Danach wende ich die Breitensuche an. Besitzt das Feld (0|0) im gelösten Labyrinth einen Wert der nicht -1 ist, dann ist das Labyrinth lösbar und kann benutzt werden. Bevor es jedoch benutzt werden kann, muss erst der Spieler und das Ziel eingesetzt werden. Da das Ziel immer an der gleichen Stelle ist, brauche ich dessen Koordinaten nicht extra in Variablen zu speichern. Es ist in der entgegengesetzten Ecke des Startpunktes, also die maximale Breite – 1 und die maximale Höhe -1. An diesen Punkt und an der Nullstelle setzen wir den Wert für Entitys ein (entity = 1). Das bedeutet, dass in dieser Stelle ein Spieler oder ein Ziel ist. Bei der Ausgabe des Labyrinthes starten wir vom Punkt (0|0) und geben die Werte der Felder Reihe für Reihe von links nach rechts aus. Im Gegensatz zu einem normalen Koordinatensystem ist bei der Ausgabe der Nullpunkt oben links und die y-

Achse geht nach unten, die x-Achse geht normal nach rechts (siehe Abb.8). Da zuerst nach der Zeile und dann nach der Spalte gefragt wird, muss ich beim 2D-Array erst die y- dann die x-Koordinate angeben. Die Environment Klasse kann verschiedene Daten zurückgeben: Das Labyrinth in einem 2D-Array, das Labyrinth in einem 1D-Array, das gelöste Labyrinth in einem 2D-/ 1D-Array, das direkte Umfeld des Spielers und den richtigen Schritt. Der richtige Schritt ist der gewünschte Output, es sagt welche Verschiebung, auf der y- und auf der x-Achse die Richtige ist.

Die letzte Klasse ist die Agent Klasse, sie speichert und trainiert das neuronale Netzwerk und nutzt dieses, um sich durch die Labyrinthe zu manövrieren. Neben dem neuronalen Netzwerk speichere ich noch ein Objekt der Klasse Environment und zwei Variablen, die zählen wie oft ein Labyrinth gelöst wurde (success) und wie oft sich der Agent verirrt hat (failed). Im Konstruktor wird das neuronale Netzwerk generiert und es wird eine Methode train aufgerufen, die das Netzwerk trainiert. Das neuronale Netzwerk wird mit acht Input Nodes, je 32 Hidden Nodes, vier Output Nodes und mit einer Learning Rate von 0.3 generiert. Ich plane dem neuronalen Netzwerk das direkte Umfeld als Input zu geben, daher die vier Input Nodes. Der Output besteht aus den Vorhersagungen für die vier möglichen Richtungen (unten, rechts, oben, links). Der Agent nutzt diese Vorhersagungen, um das Labyrinth zu lösen. In der „train“ Methode wird ein Labyrinth erzeugt und durchlaufen. Mittels der nextStep Methode aus der Environment Klasse wird immer der richtige Schritt gemacht. Das neuronale Netzwerk wird währenddessen immer mit dem direkten Umfeld (getData, von Environment), als Input und mit dem richtigen Schritt (getRightChoice, von Environment), als gewünschten Output trainiert. Der Output von „getRightChoice“ gibt aber nur zwei Werte wieder, die Verschiebung auf der x- und der y-Achse. Daher muss ich erst diese zwei Werte in die gewünschten vier Werte (unten, rechts, oben, links) umwandeln, dazu dient mir die convert Methode. Die „train“ Methode wiederholt dies mit 50 000 Labyrinthen. Neben der „train“ Methode gibt es auch die test Methode. Die test Methode bekommt einen Integer als Parameter übergeben. Die Methode löst so viele Labyrinthe wie der Parameter es vorgibt und gibt dann einen String zurück, der sagt wie viele der Labyrinthe gelöst wurden. Beim Lösen der Labyrinthe spielen zwei Methoden eine Rolle, die initialize und die nextStep Methode. Die initialize Methode generiert in der globalen environment Variable ein neues Labyrinth und ruft die rekursive nextStep Methode auf. Diese übergibt dem neuronalen Netzwerk die nötigen Daten und wandelt die Vorhersage des Netzwerks in einen Schritt (einer Verschiebung auf der x- y-Achse)

um. Ist beispielsweise die Vorhersage für ein Schritt nach unten am größten, wird dem Environment Objekt dies übergeben und das Labyrinth wird geupdatet (step, von Environment). Wenn das Labyrinth noch nicht gelöst ist, wird die Methode erneut aufgerufen, das ist der rekursive Aufruf. Wenn das Labyrinth gelöst wurde oder sich der Agent verirrt hat, werden die rekursiven Aufrufe gestoppt und die jeweilige Zählvariable erhöht sich, das sind die Abbruchbedingungen. Die show und die showNextStep Methode machen das gleiche, hier aber wird bei jedem Schritt die pause Methode aufgerufen und das Labyrinth auf der Konsole ausgegeben (printRawMaze, von Environment). Die pause Methode nutzt die TimeUnit Library, um eine Pause von 100 Millisekunden hervorzurufen.

Die TimeUnit und Math Library sind die einzigen importierten Elemente in meinem Projekt.

Beim auswählen des nächsten Schrittes („nextStep“, „showNextStep“), gibt es noch einen weiteren Faktor der eine Rolle spielt, das ist das Erkunden. Bei einer Wahrscheinlichkeit 10% geht der Agent ein Schritt nach unten oder rechts. Das dient dazu, dass im Falle einer „Fliegensituation“ der Agent nicht in einem Teufelskreis gefangen ist, mehr dazu bei „Probleme und Erkenntnisse“.

Ergebnisse

Für mein Projekt nutze ich folgende Parameter:

Neural Network:

- Input Nodes: 8, Hidden Nodes: 32 *3, Output Nodes: 4
- Learning Rate: 30%
- Kostenfunktion: MSE
- Aktivierungsfunktion: Sigmoid

Environment:

- 10 x 10
- Rate: 30%

Agent:

- Training: 50 000 Labyrinthe
- Exploration Rate: 20%
- Ein Schritt nach rechts: 10%
- Ein Schritt nach unten: 10%

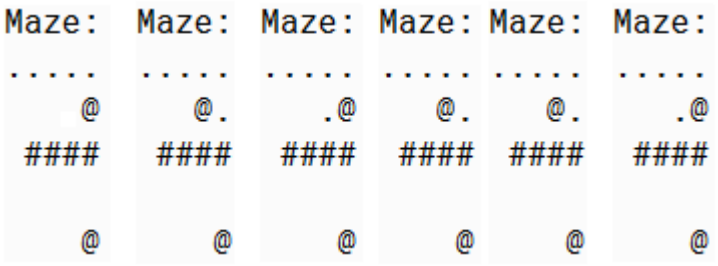
Dabei löst ein gut trainiertes neuronales Netzwerk etwas über 500 von 1000 Labyrinthen, ein schlecht trainiertes 390 von 1000. Im Durchschnitt werden 450 bis 490 von 1000 Labyrinthen gelöst.

Probleme und Erkenntnisse

Es hat sehr lange gedauert bis es erste Ergebnisse gab. Immer wenn ein Lösungsansatz nicht aufgegangen ist, habe ich mir einen komplexeren und schwierigeren Lösungsansatz genommen, bis zu dem Punkt wo ich diese nicht mehr umsetzen konnte. Das ist die Falsche Herangehensweise, die richtige wäre es das Problem zu vereinfachen, wie zum Beispiel ein kleineres Labyrinth zu nutzen. Als ich nun sah, dass einer meiner Lösungsansätze doch funktionieren kann, habe ich die Idee solange ausgebaut bis ich zu zufriedenstellende Ergebnisse kam. Nun kann meine KI jedes zweites bis drittes Labyrinth lösen

Meine KI ist wie eine Stubenfliege – Der nächste Schritt

Neuronale Netzwerke sind starke maschinelle Lernalgorithmen, jedoch sind sie in einem Bereich noch ausbaufähig, ihre Merkfähigkeit. Ein neuronales Netzwerk vergisst alles was es getan hat. Genauso wie eine Stubenfliege. Wenn eine Stubenfliege gegen ein Fenster fliegt, wird sie es nach paar Sekunden vergessen haben und erneut gegen das Fenster fliegen. Genauso ist es mit meinem neuronalen Netzwerk.



Mein neuronales Netzwerk erarbeitet sich eine Strategie mit dem es Labyrinthe löst. Wenn es in eine solche Situation gerät, verhält es sich wie eine Stubenfliege. Es geht in die Ecke, merkt dass es nicht mehr weiter geht, versucht das Hindernis zu umgehen, vergisst dies auf halben Weg und läuft wieder in die Ecke. Um solche Probleme in kleinen Umfang lösen zu können, dient die Erkundungsfunktion meiner KI. Sie hilft dem neuronalen Netzwerk aus diesem Teufelskreis rauszukommen. Der nächste Schritt wäre es sich jedes Feld zu merken, dass schon zweimal begangen worden ist und es wie eine Wand zu behandeln. Dadurch könnte die KI auch aus größeren Sackgassen entkommen.

Literaturverzeichnis

- Neural Networks: A Visual Introduction for Beginners by Michael Taylor, 2017, ISBN 9781549869136
- Iris Data Set, archive.ics.uci.edu/ml/datasets/Iris, 26.04.2019, Creator: R.A Fisher, Donator: Michael Marshall
- unsupervised learning security company, www.darktrace.com/de/technologie/, 26.04.19

Versicherung der selbständigen Erarbeitung

Ich versichere, dass ich die vorliegende Arbeit einschließlich evtl. beigefügter Zeichnungen, Kartenskizzen, Darstellungen u. ä. m. selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

_____, den _____

(Ort) (Datum)

_____

(Unterschrift)