Reinforcement Learning Schritt für Schritt in Schulprojekten

Rau, T.
Graf-Rasso-Gymnasium Fürstenfeldbruck

DOI: 10.18420/ibis-02-01-07

Zusammenfassung

In diesem Artikel werden kleine Unterrichtsprojekte vorgestellt, anhand derer man sich dem Thema *Reinforcement Learning* – einem Teilbereich des *Machine Learning,* wie auch überwachtes und überwachtes Lernen – in mehreren Schritten in der Sekundarstufe II nähern kann. Jeder Schritt stößt am Ende auf Grenzen, so dass sich jeweils die nächste Erweiterung anbietet. Am Anfang steht das manuelle Anlegen einer einfachen handschriftlichen Tabelle über entscheidende Züge in einem Spiel. Weil das nur bei einfachen Spielen geht, ist der nächste Schritt das Anlegen und Anpassen einer Q-Tabelle mit kontinuierlicher Evaluation sämtlicher Züge. Im dritten Schritt wird die für manche Fälle nicht mehr ausreichende Q-Tabelle durch ein Neuronales Netz ersetzt.

Einleitung

Wenn man eine Strategie für das Lösen von einer Gruppe von Aufgaben entwickeln will, gibt es verschiedene Möglichkeiten: Man kann sich selber einen Algorithmus überlegen, aber das erfordert Analyse und geht nur bei überschaubaren Aufgaben; man kann einen bewährten Algorithmus wie den Minimax-Algorithmus verwenden, aber der erfordert ebenfalls Analyse und eine Bewertungsfunktion. Ein ganz anderer Ansatz ist das Lernen durch Bestätigung, *Reinforcement Learning:* Das ist eine Form des maschinellen Lernens, bei dem ein Agent zu Beginn noch nicht weiß, was eine geeignete Strategie ist, aber durch Rückmeldungen aus einem System nach und nach lernt, die eigenen Entscheidungen abhängig vom Zustand des Systems so zu treffen, dass schließlich das gewünschte Verhalten gezeigt wird – ohne eigene Analyse (Sutton & Barto 1999, S. 3).

1. Naives Reinforcement Learning: Notieren von spielentscheidenden Zügen

Man kann es bereits als eine einfache Form von Reinforcement Learning betrachten, wenn man sich lediglich bei einem Spiel merkt, welcher Zug in welchem Spielzustand zu einem unmittelbaren Sieg oder einer Niederlage geführt hat, und diesen Zug dann bei zukünftigen Wiederholungen im selben Zustand entsprechend bevorzugt oder vermeidet. (In den Beispielen geht es insgesamt meist um Spiele, weshalb von Spielzustand und Zug die Rede ist; allgemeiner würde

man wohl von Zuständen und Entscheidungen sprechen.) Das heißt, man probiert einfach mehr oder weniger durch *brute force* alles aus und merkt sich für jede Kombination von Spielzustand, wie oft er zu Sieg oder Niederlage geführt hat. Damit lassen sich bereits einfache Spiele erlernen.

Beispiel: Mau-Mau-Regeln erkennen

Mau-Mau ist ein einfaches Kartenspiel, bei dem als grundsätzliche Ablageregel gilt: Man darf auf eine Karte eine andere Karte legen, wenn sie den gleichen Wert oder die gleiche Farbe (im Sinn von Pik, Herz, Karo, Kreuz) hat. Theoretisch sind auch andere Regeln denkbar, im Spiel Eleusis von Robert Abbott geht es sogar darum, induktiv solche Regeln zu erschließen. Wenn man einfach ausprobiert und mitzählt, welche Karte anlegbar ist und welche nicht, errät man schnell die Regel.

Das lässt sich exemplarisch vorführen oder in Partnerarbeit durchspielen. Sinnvoll ist dabei das Anlegen einer Tabelle, um die Ergebnisse festzuhalten; sie wird später ein wichtiges Element beim Reinforcement Learning darstellen. Dort heißt sie Q-Tabelle oder *Q table*, weil die bewertete *Qualität* der Entscheidungen darin notiert ist.

Beispiel: Minischach

Dieses in der Informatikdidaktik viel verwendete Spiel ist eine Art Mini-Schach auf einem 3x3-Spielfeld, mit drei Bauern auf jeder Seite, oben die Figuren der menschlichen Spielerin, unten die Roboter (oder, in anderen Fassungen, Krokodile) des KI-Agenten.

Gezogen werden die Bauern wie im Schach (ohne Ausnahmezüge); gewonnen hat man, wenn man (1) die gegenüberliegende Seite mit einer Figur erreicht hat oder (2) alle gegnerischen Figuren geschlagen hat oder (3) der Gegner keinen Zug mehr machen kann,

Die menschliche Spielerin 1 beginnt immer mit einem Zug der Menschen. Der Agent ist immer Spieler 2 mit den Robotern oder Krokodilen. Bei optimaler Strategie gewinnt Spieler 2 immer. Das ist am Anfang aber noch nicht offensichtlich, etwa wenn sich Schülerinnen und Schüler enaktiv als KI-Agent versuchen, dem Schokolinsen-Algorithmus folgend. Dabei werden keine Strichlisten geführt, sondern farbige Schokolinsen stehen für die einzelnen Züge, ihre Anzahl entspricht der Qualität des Zugs – bei einem Sieg kommt eine hinzu, bei einer Niederlage eine weg.

Es gibt insgesamt 37 verschiedene Spielzustände, denen der Agent begegnen kann, wovon 18 spiegelsymmetrische Varianten anderer Zustände sind. Am Ende sollte der Agent also eine Tabelle mit 37 gefüllten Zeilen erstellt haben. Zugmöglichkeiten gibt es je nach Notation mehr, wobei nicht alle Züge in jedem Spielzustand legal möglich sind; je nach Zustand stehen maximal 4 zur Auswahl. Bald lernt der Agent, das Spiel fehlerfrei zu spielen, so dass er immer gegen einen Menschen gewinnt.

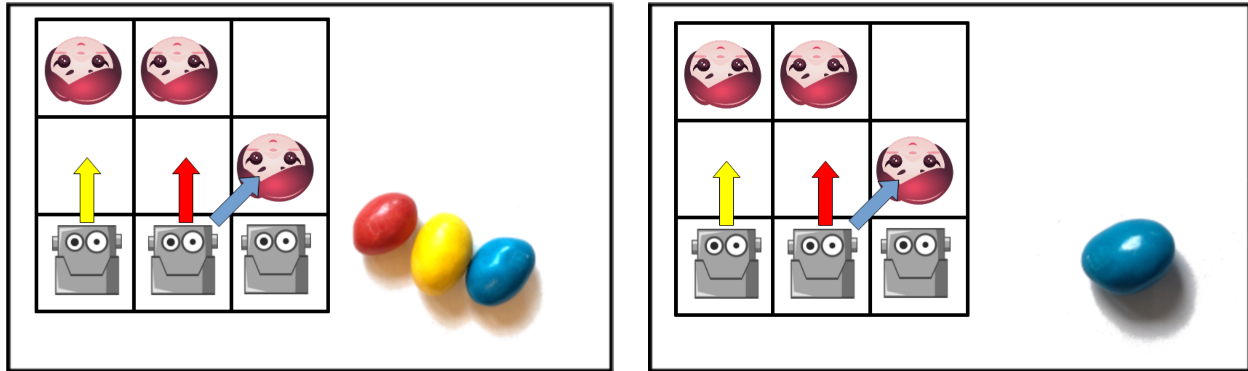

Abbildung 1: Ursprüngliche und spätere Bewertung der Züge: Alle drei Optionen sind am Anfang gleich bewertet, nach dem Lernen verbleibt nur eine.

Beispiel: Pong/Breakout

Selbst ein ganz simples Computer-Breakout oder Pong funktioniert. Ein Sieg ist dabei, den Ball mit dem Schläger zu erwischen, worauf der Ball abprallt; das wird mit 1 Punkt belohnt. Als Niederlage gilt, wenn der Ball die untere Linie berührt, was mit -1 Punkt bestraft wird.[1] Fürs Erste sei der Zustand des Systems die Differenz der x-Positionen von Ball und Schläger, eventuell durch 10 geteilt, um eine überschaubare Anzahl an Zuständen (je nach Spielfeldgröße um die 60) zu erhalten[2]. Die Zugmöglichkeiten sind: den Schläger 1. gar nicht, 2. nach links oder 3. nach rechts zu bewegen.

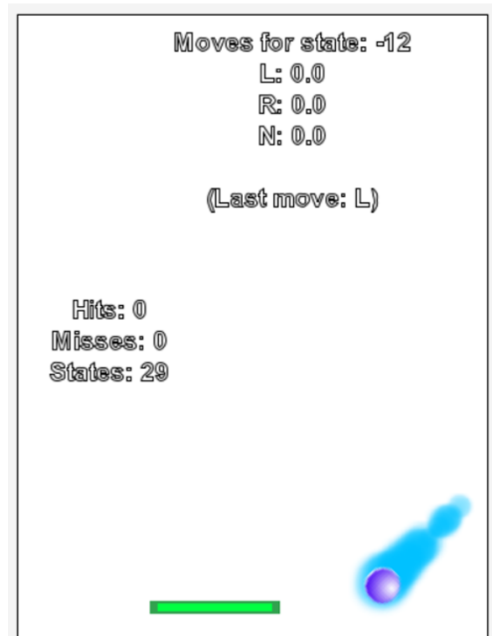

Abbildung 2: Spielsituation Breakout: Ball weit entfernt von Schläger

Es stimmt zwar, dass die KI im folgenden Zustand keinen Unterschied zwischen den Entscheidungen lernen kann, da alle Entscheidungen zu einer Niederlage, also dem Verpassen des Balls, führen:

---

[1] Man kann stattdessen aber auch in allen anderen Situationen belohnen: je kleiner der Abstand zwischen Ball und Schläger, desto mehr Belohnung. So lernt der noch Agent schneller, aber er lernt vielleicht weniger, Breakout zu spielen, sondern er lernt eben, den Schläger parallel zum Ball zu führen – man hat das Problem für ihn vereinfacht.

[2] Bei den bisherigen Spielen war die Wahl des Zustands unkompliziert, nämlich meist die vollständige Information über das System. Bei Breakout gibt es mehrere Möglichkeiten: 1. Die Differenz der x-Positionen von Ball und Schläger. Allerdings nimmt diese Reduktion bereits einen Teil der Lösungsstrategie voraus: sie muss etwas mit den x-Koordinaten zu tun haben. Wenn man als Zustand stattdessen 2. ein Tupel wählt aus x- und y-Position des Balls, seiner Flugrichtung, und der x-Position des Schlägers, also die gesamten Systeminformationen, dann gibt man weniger vor. Die Anzahl der Zustände ist jetzt allerdings viel größer, das naive Belohnungssystem versagt dabei fast ganz.

Die Bewegungen Links, Rechts oder Nichts werden alle ausprobiert und alle bestraft. da alle gleichermaßen zu einer Niederlage führen. Dennoch kann das Spiel gelernt werden: Wenn der Ball in einer anderen Situation knapp am Schläger ist, so dass die Entscheidung einen Unterschied macht, wird die richtige belohnt. Zwar ist es so nur eine kleine Anzahl an Zuständen, bei denen sinnvolle Entscheidungen gelernt werden, aber das System bewegt sich am Ende in dieser simplen Version immer nur innerhalb dieser Menge an Zuständen.

Gegenbeispiel: Tic-Tac-Toe

Auch auf Spiele wie das Streichholz-Wegnehm-Spiel Nim oder Tic-Tac-Toe lässt sich diese einfache Form des Reinforcement Learning anwenden, allerdings nicht sehr gewinnbringend: Der so trainierte Agent spielt schlecht. Die Analyse, warum er schlecht spielt, führt zu den Grenzen des bisherigen Systems und weckt das Bedürfnis nach einer Erweiterung.

Die KI kann zwar lernen, nie einen Zug zu machen, der zu einer unmittelbaren Niederlage führt – aber nur, sofern sich das überhaupt vermeiden lässt. Doch anders als bei den bisher betrachteten Spielen gibt es bei Tic-Tac-Toe Situationen, in denen *jeder* mögliche Zug zu einer Niederlage führt. Eine KI, die den Namen verdient, sollte in der Lage sein, solche Situationen zu vermeiden, aber das ist mit dem bisherigen System nicht möglich.

Wenn die menschlichere Spielerin 1 links oben beginnt, gibt es acht mögliche Züge, die alle mit dem neutralen Initialwert bewertet sind. Keiner dieser Züge kann unmittelbar zu Sieg oder Niederlage führen, so dass keiner dieser Züge in der Bewertung eine Änderung erfahren kann. Tatsächlich führen aber sieben der acht Möglichkeiten gegen eine geübte Spielerin 1 immer zu einer Niederlage.

<table>
<tr><td>1</td><td>2</td><td>2</td></tr>
<tr><td>2</td><td></td><td>2</td></tr>
<tr><td>2</td><td>2</td><td>2</td></tr>
</table>

Abbildung 3: Alle eingezeichneten Züge für 2 führen in der Situation 100|000|000 mittelbar zu einer Niederlage.

(Auch bei dem Minischach gibt es einen einzigen solchen Zustand, in den man als Agent gar nicht erst kommen möchte, weil jeder mögliche Zug zu einer Niederlage führt. In der Schokolinsenschreibweise heißt das, dass keine einzige Schokolinse mehr zur Verfügung steht. Allerdings lernt die KI aus anderen Gründen, diese Situation gar nicht erst aufkommen zu lassen, da es bereits zuvor einen eindeutigen und damit präferierten Gewinn-Zug gibt.)

2. Eigentliches Reinforcement Learning: Kontinuierliche Bewertung

Die Lösung für das Problem: Man bewertet nicht nur den Zug, der Sieg oder Niederlage gebracht hat, sondern auch den vorhergehenden Zug, und damit also: *jeden* Zug. Der Agent kann dabei nicht in die Zukunft schauen, welcher Zug ihm später schaden oder nutzen wird, aber in die Vergangenheit. Deshalb wird in der Q-Tabelle im *aktuellen* Zustand bewertet, wie gut der letzte Zug im *vorhergehenden* Zustand war, abhängig von – wie bisher – Belohnung oder Bestrafung, aber eben zusätzlich abhängig von den Optionen, die jetzt im aktuellen Zustand zur Verfügung stehen. Gibt es da nur noch Züge, die schlecht bewertet sind, wird der Zug im vorhergehenden Zustand, der einen hierhergebracht hat, abgewertet. Gibt es da mindestens einen hoch bewerteten Zug, wird der Zug, der einen im vorhergehenden Zustand hierhergebracht hat, aufgewertet.

Ein ungünstiger Zustand und wie man dorthin kommt

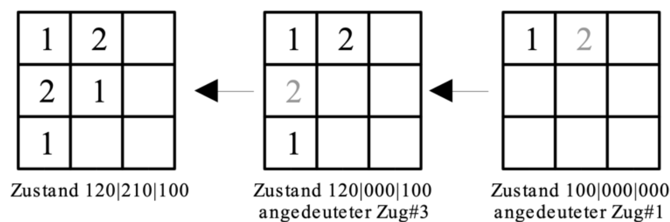

Abbildung 4: Drei mögliche Situationen für den Agenten/Spieler 2

Im Zustand 120|210|100 gibt es vier Zugmöglichkeiten für den Agenten 2, alle können zu einer unmittelbaren Niederlage führen, haben also einen niedrigen Wert in der Q-Tabelle. Dementsprechend müsste der Zug, der zu dieser Situation geführt hat, abgewertet werden, es handelt sich etwa um den Zug Feld #3 im Zustand 120|000|100. Allerdings sind die anderen fünf Züge in eben diesem Zustand ebenfalls schlecht bewertet, da sie unmittelbar zu einer Niederlage führen können. Demnach muss bereits der Zug, der zu diesem Zustand 120|000|100 führt, herabgesetzt werden, etwa Zug Feld #1 im Zustand 100|000|000.

So lernt die KI, diesen Anfangszug zu vermeiden, obwohl er erst später zu einer Niederlage führen wird.

Kontinuierliche Bewertung

Für diese kontinuierliche Bewertung des letzten Zugs im letzten Zustand unter Berücksichtigung von Belohnung, Bestrafung und den Optionen im aktuellen Zustand verwendet man die Bellman-Gleichung, hier in einer möglichen Version (nach Wang 2020):

$$Bewertung_{alt} = Bewertung_{alt} + Lernrate \cdot (\gamma \cdot Maximalwert_{neu} - Bewertung_{alt} + Belohnung)$$

Üblicherweise ist sie auch so formuliert:

$$Bewertung_{alt} = (1 - Lernrate) \cdot Bewertung_{alt} + Lernrate \cdot (Belohnung + \gamma \cdot Maximalwert_{neu})$$

Dabei ist $Bewertung_{alt}$ die Bewertung des bereits gemachten Zuges im vorherigen Zustand, $Maximalwert_{neu}$ die höchste Bewertung irgendeines Zuges im neuen, aktuellen Zustand. Diese Belohnung kann positiv oder negativ sein, in einem einfachen Fall beträgt sie zum Beispiel 1 bei einem Sieg und -1 bei einer Niederlage. Der Wert γ bezeichnet den Diskontierungsfaktor zwischen 0 und 1 (z.B. 0.5), der den zukünftigen Wert mehr oder weniger stark in das Ergebnis eingehen lässt, und die Lernrate zwischen 0 und 1 (z.B. 0.75) modifiziert die gesamte Werteänderung.

(Bei Lernrate 0 ändert sich nie etwas, ansonsten bestimmt die Lernrate, wie schnell die Bewertung im Falle keiner weiteren Änderungen konvergiert, und zwar zu $\gamma \cdot Maximalwert + Belohnung$. Bei $\gamma = 0$ konvergiert sie also allein zum Wert der aktuellen Belohnung, ohne dass die zukünftige Situation berücksichtigt wird.)

Wenn alle Züge in einem Zustand negative Werte haben, wird jeder Zug, der zu diesem Zustand führt, beim entsprechenden Durchgang im Wert herabgesetzt. Es spricht sich sozusagen herum, wenn ein Zustand eine Sackgasse darstellt.

Mit dieser Änderung lässt sich jetzt auch Tic-Tac-Toe erlernen.

Zustand	Zug	Bewertung
112 012 000	#3 #6 #7 #8	0.0 0.369140625 0.375 1.0
110 200 201	#2 #4 #5 #7	0.0 0.39825439453125 -0.75 -0.75
002 110 102	#0 #1 #5 #7	0.32967033384017147 -0.75 1.0 -0.75

Abbildung 5: Q-Tabelle für Tic-Tac-Toe, verkürzter Ausschnitt (~2100 Zustände, maximal 8 Optionen)

Exploratives Verhalten

Normalerweise wählt ein Agent während des Trainings als *greedy algorithm* immer den bestbewerteten Zug in einer gegebenen Situation. Allerdings ist es manchmal sinnvoll, auch andere Züge auszuprobieren. Deshalb ergänzt man oft eine Explorationsrate zwischen 0 und 1, etwa 0.05, die bestimmt, mit welcher Wahrscheinlichkeit ein zufälliger Zug statt des bestbewerteten ausgeführt wird.

3. Alltagsbeispiel selbstfahrende Autos: Weitere Probleme und Lösungen

Autonomes Fahren ist ein aktueller Kontext für Informatik und künstliche Intelligenz, auch in Computerspielen gibt es Fahrzeuge, die maschinell gesteuert werden. Deshalb bietet es sich an, zu schauen, wie weit man dabei mit Reinforcement Learning kommt. Die Aufgabe: Ein Auto soll einen Parcours entlang fahren können. Am Anfang soll das Fahren so weit wie möglich vereinfacht werden, später kann man immer noch weitere Elemente in das Modell einführen.

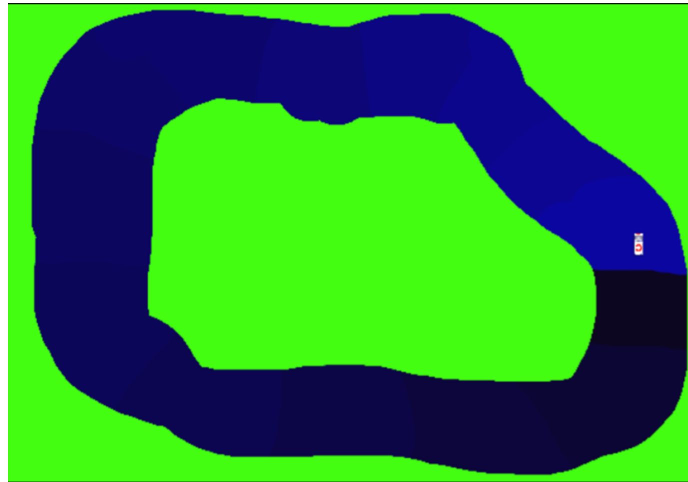

Abbildung 6: Eine einfache Fahrstrecke für ein Auto

- Es gibt nur ein Auto, keine weiteren, und keine Hindernisse. Das Auto wird als Punkt betrachtet.
- Das Auto kann jeweils nur links drehen und etwas fahren, rechts drehen und etwas fahren, oder geradeaus fahren. Es gibt keine Beschleunigung.
- Gefahren werden kann nur auf der Strecke, beim Betreten des grünen Bereichs außerhalb beginnt man wieder an der Startposition.

Umgesetzt wird das Projekt mit der didaktischen Entwicklungsumgebung Greenfoot.

Eine erste Lösung

Wieder muss man erstens entscheiden, was man als Zustand wählt. Eine naive, aber einfach umzusetzende Idee soll am Anfang stehen: x- und y-Koordinate des Autos und seine Drehrichtung. Alle drei bieten sich als typische Attribute eines Greenfoot-Actor-Objekts an. Es gibt dabei je nach Größe des Spielfelds als Obergrenze 640 * 400 * 8 Zustände, die aber nicht alle ausgeschöpft werden. Auf die Nachteile dieses Ansatzes wird später eingegangen.

Zweitens muss man entscheiden, welches Verhalten belohnt wird. Dabei reicht es, wenn das Verlassen der Bahn oder Fahren in die Gegenrichtung mit -1 bestraft und das Erreichen eines neuen Streckenabschnitts mit +1 belohnt wird. Die Streckenabschnitte sind farbig kodiert, erreicht das Auto einen dunkleren Abschnitt als den bisherigen, gilt das als Fortschritt.

Nach diesem System lernt der Agent relativ schnell, die Strecke abzufahren, im Bild sieht man die erste erfolgreiche Runde nach vielen Fehlversuchen (die jeweils zu einem Neustart am Anfang führen):

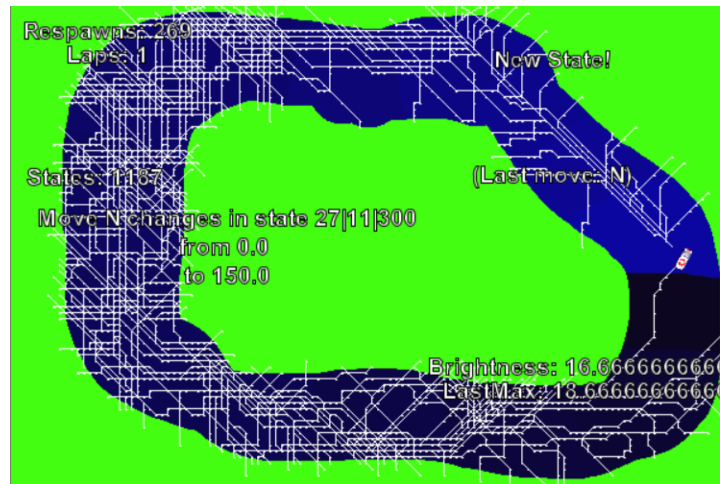

Abbildung 7: Erste vollständige Runde nach vielen erfolglosen Versuchen

Vielleicht sind Schülerinnen und Schüler bereits jetzt auf das Problem gestoßen, dass der Agent gelernt hat, genau eine Strecke abzufahren und bei anderen Strecken versagt. Als Test legt man einen zweiten Parcours an. Man kann zwar auch diesen trainieren, aber das Ergebnis lässt sich nicht übertragen. Es ist klar, dass das an der Wahl des Zustands liegt - die KI lernt einfach blind, an welcher absoluten Leinwandposition welches Verhalten gewünscht ist.

Eine zweite Lösung: Sensoren

Eine bessere Lösung ist, das Auto mit Sensoren auszurüsten, etwa drei oder fünf davon, die die Entfernung nach vorne und zu den Seiten messen. Abhängig von Länge und Anzahl der Sensoren gibt es als Obergrenze 25^5 mögliche Zustände, und auch mit diesem System lässt sich das Abfahren der Strecke lernen:

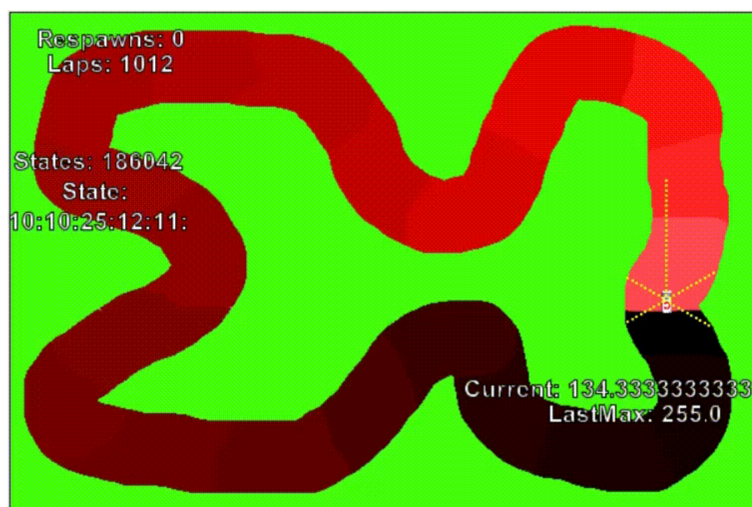

Abbildung 8: Auto mit fünf Sensoren auf schwierigerer Strecke

Allerdings ist die Übertragbarkeit auf andere Strecken immer noch gering. Bei einiger Überlegung wird die Ursache klar. Der Agent lernt zum Beispiel, sich in der Situation 05|25|20 (Abstände links, vorne, rechts) erfolgreich zu verhalten, und kann das auch übertragen auf genau diese Situation auf einer anderen Strecke. Aber der an sich ähnliche Zustand 05|25|19 stellt einen völlig neuen Eintrag in der Q-Tabelle dar, der neu gelernt werden muss und auf den sich die Ergebnisse ähnlicher, bereits bekannter Einträge nicht übertragen lassen.

Welche Lösungen gibt es für dieses Problem? Man könnte, wenn man das Lernen für abgeschlossen erklärt hat, bei unbekannten Zuständen nach dem jeweils ähnlichsten Zustand in der Q-Tabelle suchen. Oder man könnte so lange auf vielen verschiedenen Strecken trainieren, bis der Zustandsraum völlig erfasst und in die Q-Tabelle eingegangen ist. Oder man verwendet statt der Q-Tabelle ein Neuronales Netz.

Eine dritte Lösung: Neuronales Netz

Wenn die Kombinationsmöglichkeiten für eine Q-Tabelle zu umfangreich werden, hilft stattdessen ein Neuronales Netz, Ähnlichkeiten erkennen und zusammenfassen. Es besteht aus einzelnen vernetzten Neuronen, wobei die Eingabeknoten Werte annehmen und sie in der einfachsten Form modifiziert an eine versteckte Schicht von Knoten weitergeben, die ihrerseits aus den verschiedenen Inputs neue Outputs generieren und an eine Ausgangsschicht von Knoten weitergeben. Die Art der Modifikation wird dabei nach und nach lernend verbessert, bis ein Ziel erreicht ist. Statt der Q-Tabelle wird jetzt ein solches Netz verwendet.

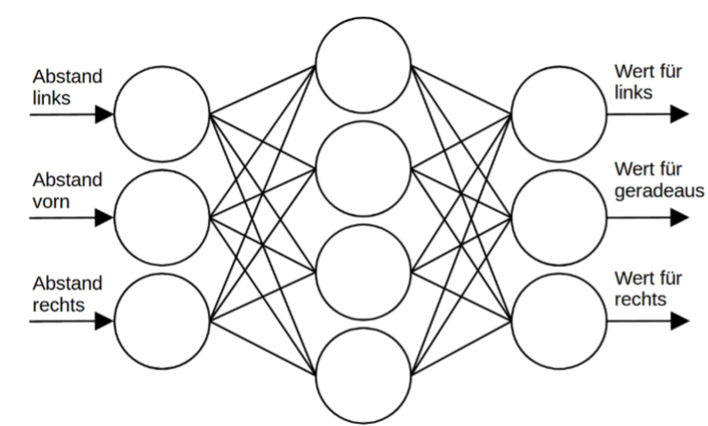

Abbildung 9: Neuronales Netz mit 4 versteckten Knoten, für ein Auto mit 3 Sensoren und 3 Fahrtmöglichkeiten

- Anzahl der Eingabeknoten: entspricht Anzahl der Sensoren beim Auto.
- Anzahl der Ausgabeknoten: entspricht Anzahl der möglichen Entscheidungen, im einfachen Fall drei: Geradeaus, Links, Rechts.
- Anzahl der versteckten Knoten, Lernrate, insbesondere Wahl der Initialbelegung und der Aktivierungsfunktion: hängt von der konkreten Situation ab.

Die Eingangswerte entsprechen den von den Sensoren gemessenen Werten. Die Ausgabewerte entsprechen dabei der Bewertung der verschiedenen Entscheidungen: Der Ausgabeknoten mit

dem höchsten Wert ist die Entscheidung, die am besten bewertet ist. Die Ausgabeknoten stehen für die Zugmöglichkeiten.

Das Trainieren des Netzes läuft ebenfalls über die oben genannte Bellman-Gleichung. Dabei ist $Bewertung_{alt}$ wieder die Bewertung des letzten Zuges im vorherigen Zustand. Sie ist zu ermitteln, indem man den vorherigen Zustand als Eingabe für das Neuronale Netz verwendet und die Bewertung aus dem Ausgabeknoten abliest, der dem Zug entspricht. Zum Beispiel könnten die Eingabewerte 05|25|20 sein, die Ausgabewerte 4|3|-2, der Zug selber war Zug 0 und dessen neue Bewertung (also die Ausgabe des ersten Knotens der Ausgabeschicht) 7.

Dann wird das Netz so trainiert, dass für die Eingabewerte des alten Zustands die Knoten der Ausgabeschicht nicht mehr das ursprüngliche Tupel 4|3|-2 ergeben, sondern jetzt zumindest näherungsweise 7|3|-2, also mit dem neu ermittelten Wert.

Das Ergebnis: Jetzt lässt sich das auf einem Kurs gelernte auch auf andere Kurse übertragen. Klar ist allerdings: Wenn ein Kurs nur Linkskurven enthält, wird das Auto auf einem Kurs, der auch Rechtskurven enthält, wenig erfolgreich sein –eine Hypothese, die man überprüfen kann.

Bereits mit drei Sensoren und vier Neuronen in der versteckten Schicht lässt sich ein Verhalten erlernen, das zum Abfahren zweier verschiedener Strecken ausreicht.

Warum überhaupt Reinforcement Learning?

Tatsächlich könnte man die Steuerung ganz ohne KI anlegen. Schließlich lässt sich für das bisherige System auch manuell ein Algorithmus finden und umsetzen: Wenn der Sensor vorne am größten ist, fahre geradeaus, sonst wenn er links am größten ist, mache die Linksbewegung, sonst die Rechtsbewegung. Die Vorteile, die Reinforcement Learning mit oder ohne Neuronalem Netz bieten, zeigen sich erst dann, wenn ein solcher Algorithmus für das optimale Verhalten nicht mehr so einfach manuell angelegt werden kann. Deshalb kann man Erweiterungsmöglichkeiten ausprobieren: Andere Autos, zusätzliche Hindernisse, Beschleunigung und Bremsen.

Quellen

Alle Webseiten/Links wurden zuletzt geprüft am 01.05.2023.

Michaeli, Tilman & Seeger, Stefan (2023): Schlag den Roboter – Maschinelles Lernen erfahren. https://computingeducation.de/proj-schlag-das-kroko/

Rau, Thomas (2023): KI: Reinforcement Learning 2. https://www.herr-rau.de/wordpress/2023/03/ki-reinforcement-learning-2.htm

Sutton, Richard & Barto, Andrew (1999): Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.

Wang, Mike (2020): Deep Q-Learning Tutorial: minDQN. A Practical Guide to Deep Q-Networks. https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

Lizenz

Dieser Artikel steht unter der Lizenz CC BY NC 4.0 zur Verfügung.

Den Greenfoot-Programmcode gibt es auf Anfrage.

Kontakt

Thomas Rau

Graf-Rasso-Gymnasium Fürstenfeldbruck

E-Mail: thomas.rau@ifi.lmu.de