Einführung in das Reinforcement Learning

Patrick Dammann

heute

Contents

1 Abstract

Diese schriftlichen Ausarbeitung zu meinem Seminar-Vortrag mit dem Thema "Einführung in das Reinforcement Learning" soll einen kurzen Überblick über das Thema Reinforcement Learning im Allgemeinen geben.

Der erste Teil erklärt dafür zunächst die generelle Problemstellung, für die Reinforcement Learning eine Lösung anbieten möchte. Hierbei soll erklärt werden, was ein Markov Entscheidungs-Problem ist, welche Notationen dafür benötigt werden und wie man es bei Vorhandensein des vollständigen Modells lösen kann.

Der zweite Teil der Ausarbeitung stellt Reinforcement Learning als Konzept vor, jede Probleme zu lösen wenn nicht alle Informationen direkt verfügbar sind. Hierbei werden hauptsächlich grundlegende Ideen von Algorithmen so wie Verbesserungen für eben jene vorgestellt, und zuletzt die aktuelle Relevanz des Konzepts bedingt durch Integrationsmöglichkeit von state-of-the-art-Methoden aufgezeigt.

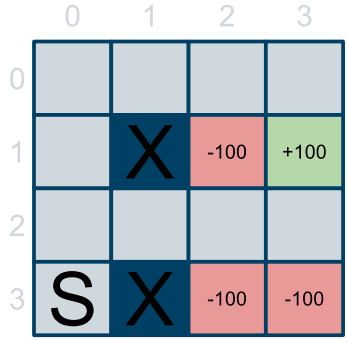
2 Markov Entscheidungs-Probleme

Markov Entscheidungs-Probleme (kurz **MDP** für *markov decision process*) sind ein Modell für Probleme, bei denen ein Agent versucht die größtmögliche Belohnung (*reward*) zu erzielen. Er bewegt sich dazu durch eine Menge von Zuständen (*states*), indem er aus einer Menge von Aktionen (*actions*) wählt. Welchen Zustand der Agent erreicht ist nicht deterministisch, die Wahrscheinlichkeiten sind jedoch nur von der gewählten Aktion und dem aktuellen Zustand abhängig. Die Belohnung, die der Agent für einen Übergang erhält, wird durch Ausgangs- und Endzustand des Übergangs so wie die gewählte Aktion bestimmt. Der Agent startet in einem Startzustand, es kann diverse Endzustände geben, nach deren Erreichen keine Aktionen mehr ausgeführt werden können.

Ziel ist es, eine Strategie (*policy*) zu finden, welche jedem Zustand die Aktion zuordnet, durch die der höchste Gesamtgewinn (also die höchste Summe über alle erzielten Gewinne) erwartet werden kann.

2.1 Ein Beispiel

In diesem Beispiel werden die Zustände durch die Felder eines 4×4-Grids visualisiert. Der Startzustand $(3, 0)$ ist durch ein großes S markiert, die Zustände mit einem X sind nicht erreichbar.



Die Menge der Aktionen besteht aus 4 Elementen, jedes symbolisiert einen Schritt in eine der vier Himmelsrichtungen. Beim Ausführen einer Aktion landet der Agent mit einer Chance von 0.8 ein Feld weiter in der gewählten Himmelsrichtung, so wie mit je einer Chance von 0.1 in einer der zwei orthogonalen Richtungen. Würde der Agent hierbei aif einem Feld landen, welches er nicht betreten kann (eines der mit X markierten, oder außerhalb des Grids), bleibt er auf seinem Feld.

Von den bunten Feldern aus führt jede Aktion mit einer Chance von 1 in den Endzustand. Der Agent erhält für diesen Übergang die auf dem Feld verzeichnete Belohnung (100 oder -100). Für jede andere Aktion wird eine Belohnung von -2 verbucht.

2.2 Notation

Zur Beschreibung eines MDPs werden einige Notationen benötigt, die hier angegeben werden. *(Zu einigen Notationen werden Beispiele zum oben genannten Beispiel angegeben.)* Zur Problemstellung selbst gehören:

- $S = \{s_1, \ldots, s_n\}$ ist die Menge der **Zustände**. Die Elemente der Menge können beliebig benannt werden.
- $A = \{a_1, \ldots, a_n\}$ ist die Menge der **Aktionen**. Auch diese Elemente können beliebig benannt werden.
- $T : S \times A \times S \to [0, 1]$ ist die **Übergangs-Funktion** (*transition function*). $T(s, a, s')$ beschreibt die Wahrscheinlichkeit, mit der der Agent von Zustand s in den Zustand s' wechselt, wenn er Aktion a ausführt. Eine allgemeinere, jedoch hier nicht verwendete Schreibweise wäre $T(s, a, s') = p(s'|s, a)$.

$$
\begin{aligned}
T(s_{0,2}, a_E, s_{0,3}) &= 0.8 && \text{(Agent wählt Osten und kommt dort an)} \\
T(s_{0,2}, a_E, s_{0,2}) &= 0.1 && \text{(Agent wählt Osten, geht nach Norden, kein valider Zug)} \\
T(s_{0,2}, a_E, s_{1,2}) &= 0.1 && \text{(Agent wählt Osten, geht nach Süden)} \\
T(s_{0,2}, a_E, s_{0,1}) &= 0.0 && \text{(Agent wählt Osten, Bewegung nach Westen unmöglich)} \\
T(s_{1,2}, \_, s_{end}) &= 1.0 && \text{(jede Aktion von bunten Feldern führt zum Endzustand)}
\end{aligned}
$$

- $R : S \times A \times S \to \mathbb{R}$ ist die **Belohnungs-Funktion** (*reward function*). $R(s, a, s')$ ordnet dem Übergang von Zustand s nach Zustand s' mit Hilfe von Aktion a eine Belohung zu.

$$
\begin{aligned}
R(s_{1,2}, \_, s_{end}) &= -100 && \text{(Belohnung von buntem Feld)} \\
R(s_{1,3}, \_, s_{end}) &= +100 && \text{(Belohnung von buntem Feld)} \\
R(s_{0,2}, a_N, s_{0,1}) &= -2 && \text{(jede Bewegung kostet 2 Belohung)}
\end{aligned}
$$

Zum Lösen der Probleme werden folgende Notationen verwendet:

- $\pi(s)$ ist eine **Policy**, die dem Zustand s die optimale Aktion zuordnet. Hierbei beschreibt π eine Policy im Allgemeinen, π^* die optimale Policy.

- $\gamma \in [0, 1]$ ist der sogenannte **Discount-Faktor**. Für den Agenten werden nach jeder Aktion alle noch erreichbaren Belohnungen mit γ multipliziert, damit er kürzere Wege, die zur selben Belohnung führen, bevorzugen wird. Mit $\gamma = 1$ ist für den Agenten ein langer Weg ebenso gut wie ein kurzer, solange er die selbe Belohnung erhält. Mit $\gamma = 0$ wird der Agent nicht mehr vorausschauend planen, da für ihn nur die nächste Belohnung Wert hat. Es gilt also, einen geeigneten Trade-Off zu finden.

2.3 V-Values

Um ein MDP zu lösen, lässt sich auf ein Bewertungsschema für Zustände zurückgreifen. Hierbei ergibt sich die Bewertung eines Zustandes aus der **Gesamtbelohung, die der Agent zu erwarteten hat, wenn er eine Simulation in besagtem Zustand startet, die optimale Aktion ausführt und von dort an optimal handelt**. Die Markov-Bedingung erlaubt uns in jedem Zustand anzunehmen, wir hätten die Simulation gerade erst gestartet, da es keine Faktoren gibt, die von bereits vergangenen Ereignissen abhängen.

Da die Bewertung jedes Zustandes angibt, welche Belohnung von ihm aus zu erwarten ist, lässt sich die Bewertung wie folgt rekursiv definieren:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T\left(s, a, s'\right) \left[R\left(s, a, s'\right) + \gamma \cdot V^*\left(s'\right)\right]$$

In der eckigen Klammer befindet sich die unmittelbare Belohung, wenn der Agent von Zustand s über Aktion a in Zustand s' gelandet ist, addiert zur zu erwartenden Belohnung von Zustand s' aus (welche mit γ multipliziert wurde, da eine Aktion vorrüber ist).

Dieser Wert wird mit der Wahrscheinlichkeit des Auftretens dieses Übergangs multipliziert und durch Aufsummieren über alle möglichen Zustände s' (*man bedenke, dass T nur für Zustände, die von s über a erreicht werden können > 0 ist*) zum Erwartungswert für eine Aktion a.

Da V^* als erwartete Belohnung definiert ist, wenn der Agent immer optimal handelt, ist es der Erwartungswert der Aktion, von welcher die höchste Belohnung erwartet wird.

2.4 Value Iteration

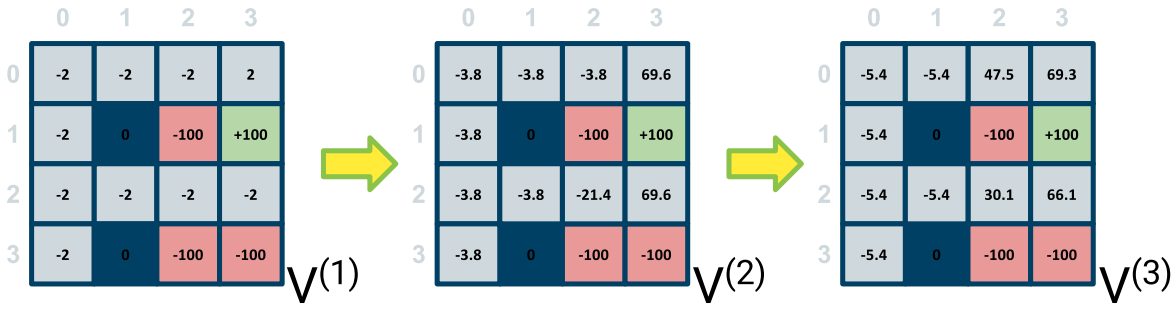
Durch die rekursive Definition von V^* lässt es sich nicht trivial berechnen. Als möglicher Lösungsansatz bietet sich daher die Value Iteration an. Hierzu wird das MDP zeitlich begrenzt und $V^{(k)}$ definiert als *Gesamtbelohung, die der Agent zu erwarteten hat, wenn er eine Simulation in besagtem Zustand startet, die optimale Aktion ausführt und von dort an optimal handelt,* ***aber nur noch k Schritte übrig hat***.

$V^{(0)}$ lässt sich einfach bestimmen, da es 0 für jeden Zustand sein muss. Auch $V^{(1)}$ folgt oft direkt aus der Problemdefinition. In unserem Beispiel wäre es -100 bzw $+100$ für die "bunten" Zustände und -2 für alle anderen, da jeder Schritt -2 Belohnung bringt.

Ein beliebiges $V^{(k)}$ lässt sich einfach berechnen. Dazu muss nur die Formel für V^*leicht angepasst werden:

$$V^{(k)}(s) = \max_{a \in A} \sum_{s' \in S} T\left(s, a, s'\right) \left[R\left(s, a, s'\right) + \gamma \cdot V^{(k-1)}\left(s'\right)\right]$$

Da sich das hintere V^* in der Gleichung ja auf Geschehnisse bezog, die einen Zeitschritt später passieren, muss es bei der Berechnung von $V^{(k)}$ durch ein $V^{(k-1)}$ ersetzt werden, da der Agent im zeitlich beschränkten MDP einen Zeitschritt später selbstverständlich nur noch einen Zeitschritt weniger zur Verfügung hat.



Für MDPs, deren maximal erzielbare Belohung nach oben beschränkt ist, wird $V^{(k)}$ für $k \longrightarrow \infty$ konvergieren, da sich auch mit mehr Schritten kein besseres Ergebnis mehr erzielen lassen wird. Durch ein $\gamma < 1$ kann quasi eine Beschränkung simuliert werden.

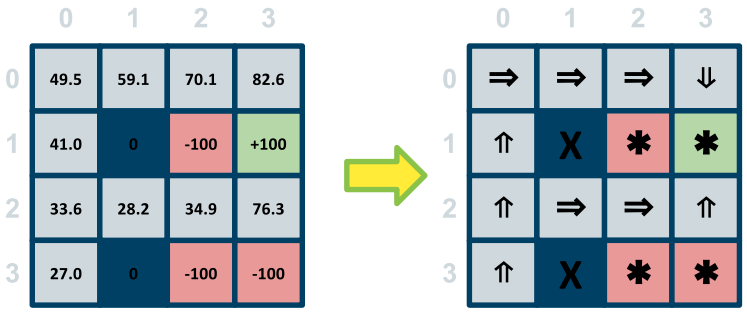
Für ein geeignet großes k gilt also $V^{(k)} = V^* \pm \varepsilon$ mit vernachlässigbarem ε (*z.B. kleiner als Maschinengenauigkeit oder kleiner als eine vorher gewählte Toleranz*), da $V^{(\infty)}$ genau unserer Definition für ein V^* im nicht zeitlich beschränkten MDP entspricht.

2.5 Policy Extraction

Mit Hilfe der optimalen V-Values V^* lässt sich jetzt eine optimale Policy für das MDP berechnen. Dazu simuliert man einen weiteren Schritt der Value Iteration. Anstatt nun aber tatsächlich das Maximum über alle a zu bestimmen, weisen wir dem aktullen Zustand die Aktion zu, die von ihm aus die erwartete Belohnung maximieren würde.

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma \cdot V^*(s') \right]$$

Auf diese Weise haben wir eine optimale Strategie erzeugt und somit das MDP gelöst.



3 Reinforcement Learning

Reinforcement Learning ist ein Machine-Learning-Konzept, in welchem ein Agent ohne anfängliche Informationen über sein Umfeld oder die Auswirkungen seines Handelns dazu trainiert werden soll, die maximale Belohnung zu erhalten.

3.1 Verbindung zu MDPs

Während sich zum Beispiel mit Value Iteration und Policy Extraction MDPs lösen lassen, bei denen alle Informationen vorhanden sind, bietet Reinforcement Learning eine Reihe von Algorithmen bei welchen vom Agenten keine Kenntnis von $T(s, a, s')$ oder $R(s, a, s')$ vorrausgesetzt werden. Der Agent muss sich also selbstständig über Aktionen durch die Zustände bewegen, um die verschiedenen Übergänge und deren Belohnungen und Wahrscheinlichkeiten selbst zu erfahren.

Da der Agent erst nach jedem Übergang die erhaltene Belohnung von extern mitgeteilt bekommt, lassen sich so bei der Problemmodellierung einfache Möglichkeiten implementieren, Belohnungen zu verteilen.

3.2 Model Based Approaches

Die erste Möglichkeit an dieses Problem heranzugehen wäre ein **model based approach**. Das bedeutet, der Agent versucht zunächst, die fehlenden Teile des Modells (also vor allem T und R) durch Samples zu approximieren. Sobald diese annährend bekannt sind, lässt sich das Problem mit beliebigen MDP-Algorithmen (wie zum Beispiel Value Iteration und Policy Extraction) lösen.

Dieser Ansatz erfordert sehr große Mengen an Samples welche zur Auswertung gleichzeitig zur Verfügung stehen müssen. Ausserdem erfordert das Sammeln der Samples einen großen Grad an Freiheit in der Bewegung des Agenten durch die Zustände, was zum Beispiel bei Anwendungen in der Robotik oft nicht gegeben ist. Bei sehr großen (oder sogar indiskreten) Zustands-Mengen ist dieser Ansatz also nur schwer umsetzbar.

3.3 Temporal Difference Learning

Als weiterer Ansatz bietet sich das **temporal difference learning** an. Die Idee dahinter ist, dass mit zufällig gewählten V-Values begonnen wird und diese langsam iterativ verbessert werden.

Nach jeder Aktion (und deren Feedback) überprüft der Agent, wie sich die ihm über V-Values versprochene erwartete Belohunung im Vergleich zur direkt erhaltenen Belohung und der nun in Aussicht stehenden erwarteten Belohnung verhält und passt die V-Values des eben verlassenen Zustandes geringfügig dementsprechend an.

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V\left(s'\right) - V(s) \right]$$

Die Bewertung von Zustand s wird um einen kleinen Teil ($\alpha < 0$) der Differenz zwischen versprochener ($V(s)$) und erfahrener ($r + \gamma V(s')$) Belohung angepasst. Die Lernrate (*learning rate*) α ist dabei ein wichtiger Hyperparameter und entscheident für die Konvergenz des Verfahrens. Für ein

sich verringerndes α_i (α im i-ten Zeitschritt) mit $\sum a_i = \infty$ und $\sum a_i^2 < \infty$ konvergiert das Verfahren in jedem Fall gegen die optimale Lösung (sofern es eine gibt). Aber auch für konstante, sehr kleine α konvergiert das Verfahren in der Praxis sehr oft.

Oft lässt sich der Rechenaufwand für das Lernen auch verkleinern, in dem beim Erkunden der Umgebung zunächst viele Samples gesammelt wurden und das Lernen danach auf allen gesammelten Samples am Stück passiert. Erst danach wird die Policy aktualisiert, damit wieder neue Samples gesammelt werden können. Dies wird bis zur Konvergenz wiederholt.

3.4 Q-Values/Q-Learning

Das explizite Aktualisieren der Policy kann entfallen, wenn die Policy direkt implizit mitgelernt wird. Diese Idee kann durch die sogenannten Q-Values umgesetzt werden. Hierbei werden statt einem Wert pro State $|A|$ Werte pro State gespeichert, jeweils einer pro ausführbarer Aktion.

Es ist also $Q(s,a)$ definiert als *Gesamtbelohung, die der Agent zu erwarten hat, wenn er eine Simulation in besagtem Zustand startet,* ***Aktion*** **a ausführt** *und von dort an optimal handelt*, was formal bedeutet:

$$Q^*(s,a) = \sum_{s' \in S} T\left(s,a,s'\right)\left[R\left(s,a,s'\right) + \gamma \cdot V^*\left(s'\right)\right]$$

Hierbei lässt sich sofort erkennen, dass sowohl die V-Values als auch die gewünschte Startegie implizit mitgeliefert wird:

$$V(s) = \max_{a \in A} Q(s,a)$$
$$\pi(s) = \operatorname*{argmax}_{a \in A} Q(s,a)$$

Viele beliebte Reinforcement-Learning-Algorithmen basieren auf dem direkten Lernen der Q-Values. Einer der einfachsten ist hierbei **Q-Learning**, welcher nur eine Form von Temporal Difference Learning auf Q-Values darstellt. Die Update-Regel wird dabei lediglich wie folgt abgeändert:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a' \in A} Q\left(s',a'\right) - Q(s,a)\right]$$

Selbstverständlich benötigt Q-Learning aber auch mehr Samples zum konvergieren, da $|A|$-mal so viele Parameter gelernt werden.

3.5 Exploration

Während der Agent die Zustände erkundet, hat er stets die Auswahl zwischen zwei generellen Lernstrategien. Er muss entscheiden, ob er bisher schlecht erkundete Wege durch den Zustandsraum besuchen möchte um mehr über sie herauszufinden, oder ob er sich an seine aktuelle Strategie hält um ihre Richtigkeit zu überprüfen und vielleicht weniger erforschte Zustände zu erforschen, die er erst durch bereits für gut befundene Entscheidungen erreicht.

Es gibt viele Methoden um einen geeigneten Trade-Off zwischen den beiden Extrema zu finden. Drei davon, welche auch beliebig untereinander kombiniert werden können, werden hier vorgestellt:

3.5.1 Startbedingungen

Zu Beginn vieler Reinforcement-Learning-Algorithmen werden Startwerte zufällig initialisiert. Wenn zum Beispiel die Q-Values allesamt höher initialisiert werden als der höchte Wert, von dem man glaubt, dass er von ihnen angenommen werden kann, so werden die ersten Updates sehr häufig zu Verringerungen der Werte führen. Dadurch ist der Agent in den ersten Iterationen eher geneigt Wege auszuprobieren, für die er noch keine Updates durchgeführt hat.

Selbstverständlich ist diese Technik nur in den ersten Interationen sinnvoll und sollte mit anderen Methoden kombiniert werden.

3.5.2 Die ϵ-greedy Exploration

Bei dieser Erkundungs-Technik wird der neue, namensgebende Hyperparameter $0 \leq \epsilon \leq 1$ eingeführt. Vor jeder Aktion wirft der Agent nun eine unfaire Münze, so dass er mit einer Chance von $(1 - \epsilon)$ nach seiner Strategie handelt, jedoch mit einer Chance von ϵ eine völlig zufällige Aktion durchführt.

Während ϵ zur Test-Zeit für gewöhnlich auf 0 gesetzt wird, kann es zur Trainingszeit je nach Bedarf konstant sein, mit der Zeit abnehmen oder anderweitig an die Gegebenheiten angepasst werden.

3.5.3 Erkundungs-Funktionen

Um die Erkundung tatsächlich gezielt in unbekannte Regionen zu lenken, lässt sich der Iterationsschritt in der Berechnung der Q-Values wie folgt anpassen:

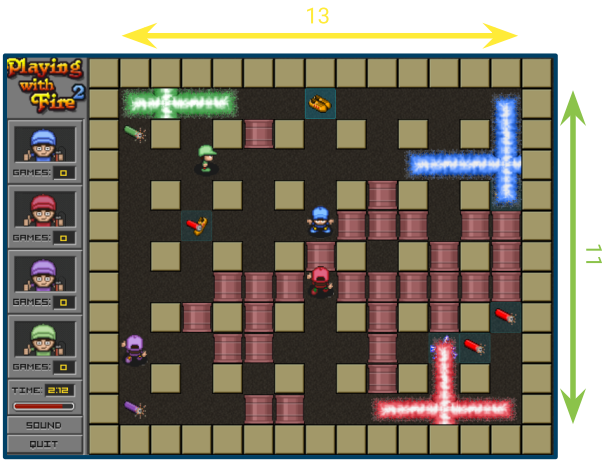
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a' \in A} \left(Q\left(s', a'\right) + E\left(s', a'\right) \right) - Q(s,a) \right]$$

Die Funktion $E(s', a')$ sinkt dabei jedes Mal, wenn die Aktion a' im Zustand s' ausgeführt wird. Sie kann zum Beispiel die Form $E(s,a) = {}^{k}/_{n_{s,a}}$ haben, wobei $n_{s,a}$ die Anzahl der Vorkommnisse des Zustand-Aktion-Tupels (s,a) ist.

Da $E(s,a)$ gegen 0 konvergiert, und fälschlicherweise gut klassifizierte Aktionen oft gewählt und im Wert verringert werden, sollte beim Konvergieren der Q-Learing-Methode der Wert der Erkundungsfunktion keinen merklichen Einfluss mehr auf die gelernte Strategie haben.

3.6 State Features

Gerade in realistischen Anwendungen wird die Anzahl verschiedener Zustände schnell unüberschaubar groß. Eine ungefähre Vorstellung lässt sich über diese grobe, obere Abschätzung der möglichen Zustände eines Flash-Spiels vermitteln:



Man stelle sich ein Szenario vor, in dem der Reinforcement-Learning-Agent lernen soll, ein einfaches Spiel im Stil des bekannten Videospiels “Bomberman” zu gewinnen.

Das Spielfeld in diesem Beispiel besteht aus $11 \cdot 13$ Feldern, welche jeweils 15 verschiedene Zustände annehmen können. Diese Zustände sind: leer, zerstörbarer Stein, unzerstörbarer Stein, Feuer von je einem der 4 Spieler, Dynamit von je einem der 4 Spieler oder eines von 4 sammelbaren Power-Ups. Allein dadurch gibt es schon $15^{11 \cdot 13} \approx 1.5 \cdot 10^{168}$ verschiedene Zustände, bereits mehr als das Quadrat aller Atome im Universum.

Wenn man jetzt noch betrachtet, dass in jedem dieser Zustände jeder der vier Spieler auf jedem der $44 \cdot 44$ Pixel jedes Feldes stehen könnte und möglicherweise die Zeit, die auf dem Timer auf die Sekunde genau abgelesen werden kann, die Spielweise des Agenten beeinflussen sollte und somit neue Zustände generiert, kommt man sogar auf eine Kardinalität von $15^{11 \cdot 13} \cdot \left(13 \cdot 11 \cdot 44^2\right)^4 \cdot (3 \cdot 60) \approx 1.6 \cdot 10^{192}$.

Natürlich ist dies nur eine obere Abschätzung, da einige der Zustände sicherlich unerreichbar sind. Die ungefähre Größenordnung zeigt jedoch gut auf, wie schnell die Zustandsmenge unermesslich groß werden kann.

Nicht nur, dass der Agent für jeden dieser Zustände eine eigene Aktion für die Strategie bzw. einen Q-Value lernen muss, er kann auch nicht über verschiedene Zustände hinweg abstrahieren und somit von Gelerntem über Zustände auf ähnliche Zustände schließen.

Aus diesem Grund lohnt es sich, statt eines Lookup-Tables für die Q-Values eine Q-Funktion zu approximieren. Eine Möglichkeit hierfür ist, nur bestimmte Features aus den Zuständen zu extrahieren und mit diesen über eine gewichtete Summe die Q-Values anzunäheren. Dies könnte beispielsweise diese Form haben:

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

An Stelle der Q-Values selbst werden nun die Gewichte zu ihrer Berechnung gelernt. Ein Update nachdem in Zustand s Aktion a ausgeführt wurde, könnte dabei so aussehen:

$$w_i \leftarrow w_i + \alpha \left[r + \gamma \max_{a' \in A} Q\left(s', a'\right) - Q(s,a) \right] \cdot f_i(s,a)$$

Natürlich hängt hierbei der Erfolg der Methode sehr stark von den gewählten Features ab.

3.7 Andere Algorithmen

Ausser naivem Q-Learning gibt es noch viele Verbesserungen am generellen Algorithmus, von denen zwei hier vorgestellt werden sollen:

3.7.1 SARSA

Der SARSA-Algorithmus (kurz für State-Action-Reward-State-Action) ist eine Abwandlung von Q-Learning, die statt des Q-Values für die optimale nächste Aktion den Wert der tatsächlich als nächstes ausgeführten Aktion nutzt, um in den Updates den nächsten Zustand zu bewerten.

Da der Einfluss von zum Beispiel Exploration Strategien für gewöhnlich gegen Ende des Trainings gegen null geht, weicht der Agent dann nur noch sehr selten von der für ihn optimalen Policy ab, wodurch der Unterschied zu Q-Learning gering wird und SARSA eine optimale Policy lernt.

3.7.2 Double Q-Learning

Q-Learning tendiert dazu, unter gewissen Umständen den Wert von Zustand-Aktions-Paaren zu überschätzen. Dies kommt unter anderem daher, dass sich durch das Suchen des Maximums in der Update-Regel Überschätzungen von einzelnen Werten durch viele Updates ziehen kann.

Um die Idee von Q-Learning zu skizzieren, sollte man sich folgende alternative Schreibweise der Update-Rule anschauen:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q \left(s', \underset{a' \in A}{\operatorname{argmax}}\ Q\left(s', a'\right) \right) - Q(s,a) \right]$$

In Double Q-Learning werden nun zwei Q-Functions gelernt. In jedem Update wird dann eine von beiden zufällig ausgewählt und bekommt den Namen Q_1, die andere heißt Q_2. Q_1 ist die Funktion, die in diesem Schritt geupdatet wird. Über sie wird auch ermittelt, welche Aktion im nächsten Schritt optimal wäre. Der Wert der optimalen Aktion wird jedoch aus Q_2 ausgelesen.

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha \left[r + \gamma Q_2 \left(s', \underset{a' \in A}{\operatorname{argmax}}\ Q_1\left(s', a'\right) \right) - Q_1(s,a) \right]$$

Wichtig hierbei ist, dass die Zuweisung von Q_1 und Q_2 vor jedem Update (oder zumindest regelmäßig) geschieht, damit beide Funktionen gleichmäßig aktualisiert werden.

3.8 Deep Q-Learning

Um gut und schnell Q-Values zu lernen benötigt man eine starke und einfach zu trainierende Methode zur Funktionsapproximierung, da bei realistischen Kardinalitäten der Zustands-Menge keine Lookup-Tables benutzt werden können. Häufig scheitern Versuche aber auch daran, dass von Hand extrahierte Features eines Zustandes dem Agenten nicht die Informationen zukommen lassen, die er für eine optimale Entscheidungsfindung benötigen würde.

In den letzten Jahren haben sich neuronale Netze mehr und mehr als Konzept der Wahl in beiden Problemfällen erwiesen. Sie können beliebige Funktionen approximieren, lassen sich gut und sehr einfach auf GPUs trainieren und lernen überraschend präzise Features zu extrahieren, vor allem wenn diese im Rest des Netzwerks weiterverarbeitet werden. Als kleinen Bonus haben sich in den letzten Jahren Netzwerk-Architekturen für etliche Problemfälle etabliert, weshalb es egal ist, ob die Umgebung dem Agenten den Zustand in Form von Zahlen, Bildern (Video-Kamera, Video-Spiel) oder sogar Audiospuren oder Text zukommen lässt.

Das Neuronale Netz bekommt beim Deep Q-Learning sämtliche Informationen der Umgebung als Input, möglicherweise zusätzlich sogar zwischengespeicherte Zusatzinfomationen wie die Aktionen und Zustände der letzten n Aktionen. Als Output wird ein $|A|$-dimensionaler Vektor erwartet, der die Q-Values zu allen Aktionen beinhaltet.

Da Neuronale Netze zum Trainieren ein Target brauchen, also ein Ergebnis, das der Trainierende statt des tatsächlichen Ergebnisses gewünscht hätte, wird als Target genau das definiert, was im Kapitel über Temporal Difference Learning als "erfahrene Belohnung" bezeichnet wurde, also hier $r + \gamma \max_{a' \in A} Q(s', a')$.

Da bei Neuronalen Netzen Training mit vielen Daten am Stück wesentlich effizienter umgesetzt werden kann, empfiehlt es sich hier besonders mit einer fixierten Strategie Erfahrung zu sammeln um dann mit dieser zu trainieren.

4 Fazit

Reinfoecement Learning stellt ein gutes Konzept dar, um Probleme zu lösen, welche sich als MDP formulieren lassen. Diese Klasse von Problemen mag im ersten Moment eher klein wirken, ist meiner Meinung nach jedoch viel größer als man denkt. Durch künstliche Erweiterung des State-Spaces (was durch Approximieren von Funktionen eher weniger ins Gewicht fällt) und zusätzlichen Variablen lässt sich Reinforcement Learning auch in Situationen verwenden, in denen die Zukunft entgegen der Markov-Eigenschaft sehr wohl von der Vergangenheit abhängt. Das Konzept der einen Aktion lässt sich ebenfalls sehr einfach auf mehrere parallele Aktionen transformieren, was das Verfahren noch universeller macht.

Reinforcement Learning stößt aber zum Beispiel dort an seine Grenzen, wo es keine natürlich definierten Belohnungen gibt und es Menschen sehr schwer fällt, geeignete und generalisierte Belohungen zu definieren. Auch in Fällen, in denen richtiges Handeln sehr verzögert belohnt wird, kann Reinforcement Learning versagen, da es zwar konvergieren könnte, dazu jedoch nicht hinnehmbare Mengen an Zeit benötigt.

5 Quellen

AI Course CS188 (University Berkley) http://ai.berkeley.edu/home.html
https://www.youtube.com/channel/UCB4_W1V-KfwpTLxH9jG1_iA/videos

Harmon, Mance E. and Stephanie S. Harmon. "Reinforcement Learning: A Tutorial." (1996).

Q-Learning (Code) https://github.com/farizrahman4u/qlearning4k

SARSA im Detail https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning/

Double-Q-Learning arXiv:1509.06461v3 [cs.LG]