Reinforcement learning - DADL

Niklas Jung, Lena Wagner

24. Januar 2023

1 Einleitung

Reinforcement learning (deutsch: verstärkendes Lernen) stellt neben *supervised* und *unsupervised learning* ein eigenes Teilgebiet des machine learning dar. Im Gegensatz zum supervised und unsupervised learning greift das reinforcement learning nicht auf einen statischen Trainingsdatensatz zurück, sondern lernt selbstständig in einer dynamischen Umgebung aus gesammelten Erfahrungen nach einem Belohnungsprinzip. Diese Art des Lernens ähnelt der biologischer Lebensformen.

Aktuelle Anwendungsgebiete sind beispielsweise Robotik, autonomes Fahren und Spiele wie Go (AlphaGo, 2016) und Atari Games (z.B. Snake). Letztes Jahr wurde m.H. von reinforcement learning ein neuer Algorithmus für die Matrixmultiplikation gefunden, der für viele Matrixgrößen schneller ist als von Menschen gefundene Algorithmen. (DeepMind, 'Discovering faster matrix multiplication algorithms with reinforcement learning',
https://doi.org/10.1038/s41586-022-05172-4).

2 Grundbegriffe

- **Agent** (*agent*): Die Entität bzw. der Algorithmus, der eine Umgebung wahrnehmen und Aktionen durchführen kann.
- **Umgebung** (*environment*): Die Umgebung, in der der Agent lernen soll. Zum Zeitpunkt t befindet sie sich im Zustand s_t aus der Menge der möglichen Zustände $\mathcal{S}$.
- **Aktionen** (*actions*): Die Menge $\mathcal{A} \ni a_t$ der möglichen Aktionen a_t, die der Agent durchführen kann. Der Zustand der Umgebung ändert sich nach einer durchgeführten Aktion, also $s_{t+1} = \delta(s_t, a_t)$ mit der Übergangsfunktion $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Die Menge an möglichen Aktionen kann durch den Umgebungszustand eingeschränkt sein, also $\mathcal{A} = \mathcal{A}_s$.
- **Belohnungsfunktion** (*reward function*): $r_t : \mathcal{S} \times \mathcal{A} \to \mathbf{R}$, wobei $r_t(s_t, a_t)$ die erhaltene Belohnung darstellt, wenn im Zustand s_t die Aktion a_t durchgeführt wurde. Beim Lernen soll $r_t > 0$ zu einer positiven und $r_t < 0$ zu

einer negativen Verstärkung der Bewertung der Aktion a_t im Zustand s_t führen.

- **Strategie** (*policy*): $\pi : \mathcal{S} \to \mathcal{A}$: die Strategie, mit der der Agent für einen gegebenen Zustand der Umgebung eine Aktion auswählt.

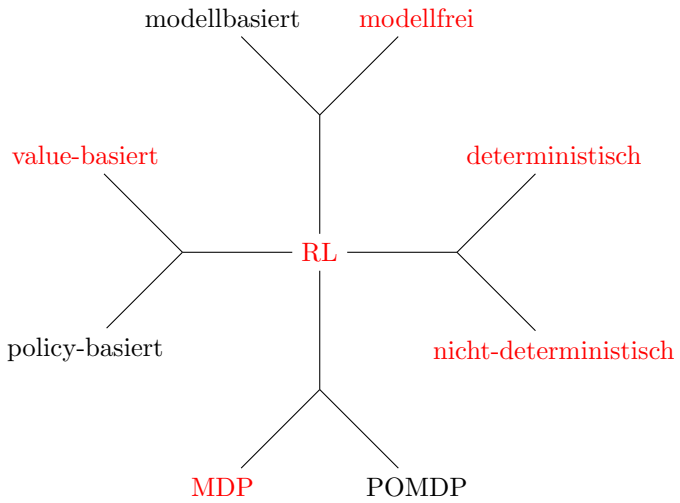

Abbildung 1: Übersicht Reinforcement Learning; wir befassen uns im Folgenden mit den rot markierten Teilgebiete.

3 Lernprozess

Ziel des Reinforcement Learnings ist es, dass der Agent basierend auf seinen Erfahrungen eine optimale Strategie erlernt, d.h. eine Strategie, die langfristig die Belohnung maximiert.

Verschiedene Annahmen führen zu unterschiedlichen Charakterisierungen des reinforcement learning, siehe hierzu Abb. **??**. Die Aktionen des Agenten können deterministisch oder nicht-deterministisch sein. Die Umgebung wird meist modelliert als *Markov Decision Process* (MDP), bei dem die zu erwartende Belohnung nur von dem momentanen Zustand und der gewählten Aktion abhängt. Entspricht die Beschreibung des Problems keinem MDP, zum Beispiel weil der tatsächliche Zustand des Agenten nicht exakt bekannt ist, bezeichnet man den Prozess als POMDP (*partially oberservable Markov decision process*).

Im Folgenden nehmen wir den einfachen Fall deterministische Aktionen und Markovscher Entscheidungsprozess an.

Lernprozess: Zu jedem diskreten Zeitpunkt t kennt der Agent den aktuellen Zustand s_t und führt eine Aktion a_t aus. Anschließend liefert die Umgebung eine Belohnung $r_t(s_t, a_t)$ und einen nächsten Zustand $s_{t+1} = \delta(s_t, a_t)$. Hierbei

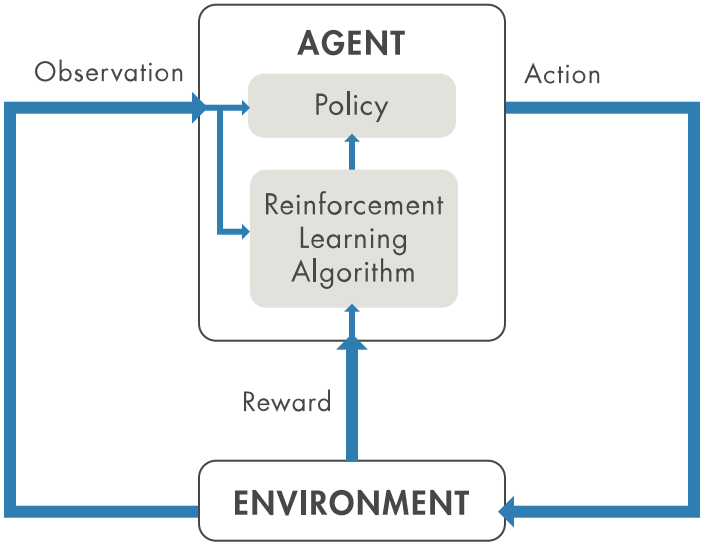

Abbildung 2: Aktions-Belohnungs-Schleife
Quelle: https://de.mathworks.com/solutions/deep-learning/deep-reinforcement-learning.html

sind die Funktionen r, δ Teil der Umgebung aber nicht notwendigerweise dem Agenten bekannt; im MDP hängen r, δ nur von dem aktuellen Zustand und der aktuellen Aktion ab, aber nicht von den vorherigen.

Die Aufgabe des Agenten ist es nun, die Strategie $\pi : \mathcal{S} \to \mathcal{A}$ für die nächste Aktion auszuwählen, wenn der aktuell beobachtete Zustand $s_t : \pi(s_t) = a_t$ bekannt ist. Die Strategie soll die **größte kumulative Belohnung** über die Zeit liefern:

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{1}$$

V^π wird auch Wertefunktion bzw. Bewertungsfunktion genannt. Gl. (**??**) ist die gewichtete Summe der Belohnungen. Der Agent startet bei s_t und folgt der Strategie $\pi : a_i = \pi(s_i)$ für $i \geq t$. Die Skontorate (*discount rate*) γ, $0 \leq \gamma < 1$, legt den relativen Wert von verzögerten und direkten Belohnungen fest. Hierdurch wird eine schrittweise Abschwächung der Bewertung für weiter weg liegende zukünftige Zustands-Aktions-Paare erreicht. ($\gamma = 0$ berücksichtigt nur die aktuellste Belohnung, bei $\gamma = 1$ werden alle zukünftigen Belohnungen gleichwertig in Betracht gezogen.) Ein typischer Wert für γ ist $\gamma \approx 0.9$

Im Lernprozess erlernt das Modell eine Strategie, die $V^\pi(s)$ für alle Zustände $s \in \mathcal{S}$ maximiert:

$$\pi^* = \arg\max_\pi V^\pi(s) \quad \forall s \in \mathcal{S} \tag{2}$$

und $V^*(s)$ ist die Wertefunktion der optimalen Strategie π^*.

Eine Strategie π^* heißt **optimal**, wenn für alle Zustände s gilt

$$V^*(s) \geq V^\pi(s) \tag{3}$$

Reinforcement learning entspricht also einem Optimierungsproblem. In der dynamischen Programmierung werden Optimierungsprobleme gelöst, indem Zwischenergebnisse über Teile von Strategien gespeichert und wiederverwendet werden. Für die optimale Strategie gilt das sog. Bellman-Prinzip:

Unabhängig vom Startzustand s_t und der ersten Aktion a_t müssen ausgehend von jedem möglichen Nachfolgezustand s_{t+1} alle folgenden Entscheidungen optimal sein.

Das Bellman-Prinzip wird beschrieben durch die **Bellman-Gleichung**:

$$V^*(s) = \max_a [r(s,a) + \gamma V^*(\delta(s,a))] \tag{4}$$

in Worten: Zur Berechnung von $V^*(s)$ wird die direkte Belohnung zu den mit dem Faktor γ abgeschwächten Belohnungen aller Nachfolgezustände addiert. Ist $V^*(\delta(s,a))$ bekannt, so ergibt sich $V^*(s)$ und somit die global optimierte Strategie π^* durch eine lokale Optimierung aller im Zustand s möglichen Aktionen a durch die Bellman-Gleichung.

Credit Assignment Problem: Ein fundamentales Problem in reinforcement learning ist die Schwierigkeit, die Belohnung am Ende einer Aktionssequenz den einzelnen Aktionen zuzuordnen, insbesondere wenn (am Anfang oder

bedingt durch das Problem) nur selten eine Belohnung erfolgt. Das Training mit solch einem *sparse reward setting* kann sehr schwierig sein. Traditioneller Lösungsansatz ist das sog. *reward shaping*: durch zusätzliche Belohnungen wird der Agent dabei unterstützt eine Strategie zu finden. Allerdings kann dies entscheidende Nachteile haben: Erstens ist die *reward shaping* Funktion zugeschnitten auf die Umgebung, also nicht generalisierbar. Weiterhin kann *reward shaping* zu overfitting oder einem unerwünschten Verhalten führen, zum Beispiel weil die Aufgabenstellung in unerwarteter Weise interpretiert wurde (*alignment problem*). Schließlich gibt es auch Fälle, in denen eine solche Beeinflussung des Lernprozesses nicht gewollt ist, da dies die Kreativität einschränkt (z.B. bei AlphaGo).

4 Q-Learning

Die direkte Bestimmung von π^* ist meist schwierig, da hierfür eine im Zustand s_t ausgeführte Aktion a_t bewertet werden soll, auch wenn nicht bekannt ist, wohin diese Aktion führt. Bei **Q-Learning** versucht man daher zunächst, eine Evaluationsfunktion $Q(s, a)$ zu bestimmen, aus der π dann abgeleitet werden kann. Diese Funktion nennen wir **Q-Funktion**. Sie ist definiert als

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a)), \tag{5}$$

wobei hier von deterministischem r und δ ausgegangen wird. Aus dieser Funktion lässt sich die Strategie

$$\pi^*(s) = \arg\max_a Q(s, a) \tag{6}$$

ableiten. Aus der Bellman-Gleichung (**??**) erhalten wir die Relation $V^*(s) = \max_{a'} Q(s, a')$. Damit können wir die Q-Funktion nun rekursiv darstellen als

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a'). \tag{7}$$

Im Folgenden werden mehrere Möglichkeiten, die Q-Funktion bei einer gegebenen Umgebung zu erlernen, vorgestellt.

4.1 Q-Wertetabelle

Ein einfacher Algorithmus zur Bestimmung der Q-Funktion bei deterministischem r und δ ist eine Q-Wertetabelle. Der Algorithmus startet mit einer Tabelle aus Werten $\hat{Q}(s, a)$, initialisiert mit $\hat{Q}(s, a) = 0$ für alle möglichen Paare (s, a). In jedem Schritt wird der folgende Algorithmus ausgeführt:

- Der momentane Zustand s wird beobachtet.
- Eine Aktion a wird ausgeführt.
- Der neue Zustand $s' = \delta(s, a)$ und die erhaltene Belohnung $r = r(s, a)$ werden beobachtet.

- Der Tabelleneintrag $\hat{Q}(s,a)$ an der Stelle (s,a) wird durch

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a') \tag{8}$$

  ersetzt.

Es lässt sich zeigen, dass unter der Annahme, dass $r(s,a)$ beschränkt ist und jeder mögliche Zustand unendlich oft besucht wird, die Funktion $\hat{Q}$ folgende Eigenschaften erfüllt:

- Die Werte $\hat{Q}(s,a)$ steigen monoton an und liegen unter dem optimalen Wert $Q(s,a)$.
- Bei unendlich vielen Iterationen dieses Algorithmus konvergiert $\hat{Q}$ gegen Q.

Der Agent muss hierfür die Funktionen r und δ nicht kennen, sondern kann die Q-Funktion nur anhand der spezifischen Werte $r(s,a)$, $\delta(s,a)$ (*samples*) dieser Funktionen beim Durchführen einer Aktion a in einem bestimmten Zustand s finden.

Exploration vs. Exploitation: Im Q-Learning Algorithmus stehen in jeder Iteration verschiedene Aktionen zur Auswahl. Wird zufällig eine Aktion ausgewählt, führt dies langfristig zu einem gleichmäßigen Erkunden (*exploration*) aller möglichen Strategien, allerdings mit einer sehr langsamen Konvergenz. Wählt der Agent hingegen die Aktion mit dem höchsten bereits gelernten $\hat{Q}$-Wert (Verwerten, *exploitation*), erreicht man schnelle Konvergenz, aber lässt möglicherweise Strategien unbesucht. Dies kann zum Erlernen nicht optimaler Strategien führen. Um diesem Tradeoff zu begegnen, bietet sich eine Kombination aus Exploration und Explotation an. Ein geeignetes Auswahlverfahren ist beispielsweise die *ε-greedy policy*, bei der mit einer Wahrscheinlichkeit von ε eine zufällige Aktion und mit einer Wahrscheinlichkeit von $1-\varepsilon$ die vorausgesagte optimale Aktion ausgewählt wird. Oft wird dabei ein hoher Erkundungsanteil am Anfang gewählt, der dann im Verlauf immer weiter reduziert wird.

4.2 Nichtdeterministische Umgebung und Temporal Difference (TD) learning

Im Allgemeinen sind r und δ für viele realistische Anwendungen keine deterministischen Funktionen mehr, sondern Wahrscheinlichkeitsverteilungen $p(r|s,a)$ und $P(s'|s,a)$. In diesem Fall verallgemeinert sich die kumulative Belohnung zu

$$V^{\pi}(s) = E\left\{ \sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, s_0 = s \right\}, \tag{9}$$

wobei $E\{X\}$ den Erwartungswert von X bezeichnet. Die Bellman-Gleichung für die Wertefunktion V^* der optimalen Strategie π^* verallgemeinert sich dann zu

$$V^*(s) = E_{\pi^*}\{r_0 + \gamma V^*(s_1) | s_0 = s\} \tag{10}$$

Wir können also, analog zum vorherigen Fall, die Q-Funktion definieren als

$$\begin{aligned} Q(s,a) &= E\{r + \gamma V^*(\delta)|s,a\} \\ &= E\{r|s,a\} + \gamma E\{V^*(\delta)|s,a\} \\ &= E\{r|s,a\} + \gamma \sum_{s'} P(s'|s,a) V^*(s') \\ &= E\{r|s,a\} + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \end{aligned} \tag{11}$$

Hierbei bezeichnet $E\{X|s,a\}$ den Erwartungswert von X am Zustand s unter der Aktion a. Allerdings gibt es hier keine Konvergenzgarantie für das im letzten Abschnitt gezeigte Verfahren zur Bestimmung der Q-Funktion mehr, da Belohnung und Nachfolgezustand bei gleichem Ausgangszustand s und Aktion a verschieden sein können, was zu einer alternierenden Folge führen kann. Um die Iteration zu stabilisieren, wird der alte gewichtete Q-Wert Gl.(**??**) addiert. Die Lernregel für $\hat{Q}$ aus (**??**) lautet nun

$$\hat{Q}_n(s,a) = (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s,a),a')] \tag{12}$$

$$= \hat{Q}_{n-1}(s,a) + \alpha_n \underbrace{[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s,a),a') - \hat{Q}_{n-1}(s,a)]}_{\text{TD-Fehler}} \tag{13}$$

mit einem zeitlich veränderlichen Gewichtungsfaktor α_n. Diese Art von Lernmethode heißt *Temporal Difference Learning* (TD-Learning). Oft wählt man $\alpha_n = \frac{1}{1+b_n(s,a)}$, wobei $b_n(s,a)$ angibt, wie oft zur $n$ten Iteration im Zustand s die Aktion a schon ausgeführt wurde.

Der Q-Learning Algorithmus für nicht-deterministische Prozesse erfolgt analog zum deterministischen Fall, wobei der Tabelleneintrag $\hat{Q}(s,a)$ durch Gl. (**??**) ersetzt wird.

4.3 Deep Q-Learning

Die oben vorgestellten Methoden versuchen, die Q-Funktion für jedes mögliche Zustand-Aktionspaar (s,a) einzeln zu approximieren. Dies ist für die meisten Anwendungen aufgrund von großen Zustands- und Aktionsräumen oft sehr unpraktikabel. Daher ist es üblich, die Q-Funktion ohne Kenntnis für alle möglichen (s,a) durch eine parametrisierte Funktion $\hat{Q}(s,a;\theta)$ zu approximieren. Ein Ansatz dafür ist *Deep Q-Learning*, bei dem $\hat{Q}(s,a;\theta)$ durch ein Neuronales Netz mit Gewichten θ dargestellt wird. Wir nennen $\hat{Q}(s,a;\theta)$ ein **Q-Netzwerk**.

Das Q-Nezwerk wird trainiert über das Minimieren der Verlustfunktion

$$L_i(\theta_i) = E\left\{(y_i - \hat{Q}(s,a;\theta_i))^2|s,a\right\}, \tag{14}$$

wobei

$$\begin{aligned} y_i &= E\{r|s,a\} + \gamma \sum_{s'} P(s'|s,a) \max_{a'} \hat{Q}(s',a';\theta_{i-1}) \\ &= E\{r + \gamma \max_{a'} \hat{Q}(s',a';\theta_{i-1})|s,a\} \end{aligned} \tag{15}$$

zu einem Iterationsschritt i. Ableiten dieser Funktion liefert

$$\nabla_{\theta_i} L_i(\theta_i) = E\left\{ \left(r + \gamma \max_{a'} \hat{Q}(s',a';\theta_{i-1}) - \hat{Q}(s,a;\theta_i) \right) \nabla_{\theta_i} Q(s,a;\theta_i) \middle| s,a \right\}. \tag{16}$$

Es kann nun also, beispielsweise über *Stochastic Gradient Descent*, die Verlustfunktion minimiert und damit eine Approximation für die optimale Q-Funktion in Form des trainierten Q-Netzwerks $\hat{Q}(s,a;\theta)$ gefunden werden.

Statt in jedem Schritt einzeln die Gewichte mit den gesammelten Erfahrungen zu updaten, bietet es sich für die konkrete Implementierung an, sogenanntes *experience replay* zu verwenden. Dabei werden zu jedem Iterationsschritt die Erfahrungen des Agenten $e_t = (s_t, a_t, r_t, s_{t+1})$ in einem Datenset $\mathcal{D}$ gespeichert. Aus diesem werden anschließend in jedem Schritt zufällige Minibatches zum Updaten der Gewichte von $\hat{Q}$ verwendet. Dies hat den Vorteil, dass deutlich weniger Korrelation bei den zum Updaten der Gewichte verwendeten Daten herrscht. Zum Auswählen der nächsten Aktion wird wieder, wie bei der Q-Wertetabelle, eine ε-greedy Policy verwendet. Statt die Q-Funktion direkt als Neuronales Netz zu approximieren, können wir ein Neuronales Netz verwenden, bei dem der Input eine Repräsentation des Zustands ist, und die Ausgabeschicht aus den vorhergesagten Belohnungen für alle einzelnen Aktionen besteht. Dies verringert u.U. die Laufzeit. Der Algorithmus sieht wie folgt aus:

```
Initialisiere Replay Memory D mit Kapazität N
Initialisiere Q mit zufälligen Gewichten
for episode = 1...M do:
    Beobachte Zustand s_1
    for t = 1...T do:
        Mit Wahrscheinlichkeit epsilon wähle zufällige Aktion a_t
        Sonst wähle a_t = max_a Q(s_t,a; theta)
        Führe Aktion a_t in Umgebung aus
        Beobachte Belohnung r_t und nächsten Zustand s_{t+1}
        Speichere Übergang (s_t,a_t,r_t,s_{t+1}) in D
        Nehme zufälligen Minibatch von Übergängen aus D
        Setze y_j = r_j für Terminalzustand s_{j+1}
        oder y_j = r_j + gamma * max_{a'} Q(s_{j+1},a'; theta), sonst
        Führe Gradient Descent-Schritte für (y_j - Q(s_j,a_j; theta) aus
    end for
end for
```

Mittels dieses Ansatzes lassen sich wesentlich größere Zustandsräume und damit wesentlich mehr praktisch anwendbare Probleme behandeln.

5 Quellen

1. Wolfgang Ertel: Grundkurs Künstliche Intelligenz, Springer, 2021
2. M.A. Carreia-Pepinan: Introduction to Machine Learning, EECS, University of California, 2019
3. V. Minh, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller: Playing Atari with Deep Reinforcement Learning
4. Uwe Lorenz: Reinforcement Learning, Springer, 2020
5. Stefan Richter: Statistisches und maschinelles Lernen, Kapitel 8, Springer, 2019
6. Tim Miller, lecture notes: COMP90054: Reinforcement Learning, https://gibberblot.github.io/rl-notes/intro.html, 2022