Deep Reinforcement Learning und Mobile Robotik
Bericht über das Forschungssemester WS 2021/22

Prof. Dr. Oliver Bittel

11.11.2022

Hochschule Konstanz - Technik, Wirtschaft und Gestaltung
Fakultät Informatik
Alfred-Wachtel-Str. 8
bittel@htwg-konstanz.de
http://www-home.htwg-konstanz.de/ bittel/

Zusammenfassung Reinforcement Learning (RL) ist ein zentrales Gebiet der Künstlichen Intelligenz und hat als Ziel, autonome Agenten zu entwickeln, die mit ihrer Umgebung interagieren, um durch Versuch und Irrtum optimale Verhaltensweisen zu erlernen. Klassische Verfahren wie Dynamische Programmierung und Q-Learning speichern das gelernte Verhalten in Tabellen und haben bei größeren Zustands- und Aktionsräumen ein Skalierungsproblem . In jüngster Zeit wurden RL-Verfahren sehr erfolgreich mit Tiefen Neuronalen Netzen kombiniert. Diese Verfahren, die auch Deep Reinforcement Learning (DRL) genannt werden, skalieren aufgrund der Approximationseigenschaft der Neuronalen Netzen wesentlich besser. Eine bemerkenswerte Vielfalt von Problemen aus der mobilen Robotik kann als RL-Problem formuliert und mit DRL-Verfahren gelöst werden.

Im Rahmen meines Praxissemesters wurden zum einen die klasssischen Verfahren als auch wichtige DRL-Verfahren wie DQN und A2C implementiert und auf einen Krabbelroboter angewendet und verglichen. In diesem Bericht werden die Algorithmen und die zugrunde liegende Mathematik ausführlich beschrieben. Dieser Bericht gibt nicht nur einen Überblick über das Forschungssemester sondern kann auch sehr gut als Einstieg in die DRL-Verfahren dienen.

1 Einleitung

Reinforcement Learning (RL, Verstärkungslernen) ist ein zentrales Gebiet der Künstlichen Intelligenz (KI) und hat als Ziel, autonome Agenten zu entwickeln, die mit ihrer Umgebung interagieren, um durch Versuch und Irrtum optimale Verhaltensweisen zu erlernen (SB18). Im Gegensatz zum überwachten Lernen (supervised learning) wird dem Agenten das optimale Verhalten nicht vorgeführt, sondern der Agent erhält durch Interaktion mit der Umgebung eine Belohnung (Verstärkung) für sein Verhalten. Daher rührt auch der Namen Reinforcement Learning oder Verstärkungslernen. Die Belohnung kann auch negativ sein für ein nicht gewünschtes Verhalten.

RL geht zurück auf die Optimierung technischer Prozesse mit *Dynamischer Programmierung* (DP) (Bel57). DP erfordert ein mathematisches Modell des Agenten und der Umgebung, was aber in der Praxis oft nicht bekannt ist (ein Vogel, der das Fliegen lernt, hat kein mathematisches Modell des Fliegens). Dieser gravierende Nachteil der DP wurde durch das modellfreie *Q-Learning* gelöst (WD92). Beim Q-Learning lernt der Agent sein Verhalten in Form einer Tabelle, in der für jeden Zustand, den der Agent einnehmen kann, und jeder durchführbaren Aktion eine Bewertung abgespeichert wird. Die Tabelle wird iterativ durch Interaktion mit der Umgebung gelernt. Das Q-Learning und andere tabellenbasierte

Verfahren haben jedoch ein Skalierungsproblem und sind auf Probleme mit einem kleinen Zustands- und Aktionsraum beschränkt. Der Zustand eines komplexeren Laufroboters kann sehr rasch eine Dimension von mehr als 30 erreichen (bedingt durch die Vielzahl seiner Gelenke). Die tabellenbasierten Verfahren stoßen dann zwangsläufig auf Speicherplatz- und Laufzeitprobleme.

Dank der Hardwareentwicklung insbesondere der von Hochleistungsgrafikkarten konnten im letzten Jahrzehnt *Neuronale Netze* mit einer Vielzahl von Schichten (*deep neural networks*) entwickelt und erfolgreich für Objekterkennung und Sprachverarbeitung eingesetzt werden (LBH15). Es war daher naheliegend, die Tabellen in den RL-Verfahren durch Neuronale Netze zu ersetzen. Die so entwickelten Verfahren werden *Deep Reinforcement Learning* (DRL) genannt. Die große Herausforderung bei DRL war es, Neuronale Netze, die überwacht trainiert werden (Experte gibt das zu Lernende vor), mit RL-Verfahren, die nur mit Belohnung und ohne Expertenwissen lernen, zu verknüpfen. Der Durchbruch von DRL-Verfahren wurde von der Firma Deepmind erzielt, die ein Agent (*Deep Q-Network*, DQN) entwickelt haben, der eine Vielzahl der klassischen Atari-Video-Spiele (siehe Abb. 1) auf Expertenniveau zu spielen lernte (MKS+15). Der Agent sieht dabei lediglich das Konsolenbild als RGB-Bild und den Score als Belohnungswert. Aktionen sind typische Joystick-Befehle. In DQN wird wegen der Verarbeitung von Bildern ein Faltungsnetz (*convolutional neural network*) eingesetzt. Ebenfalls herausragend war die Entwicklung des Agenten *AlphaZero* (SSS+17), der ohne Expertenwissen das Brettspiel GO spielen lernte. AlphaZero lernte sein Verhalten durch Spiele gegen sich selbst. Eine Belohnung erhielt der Agent erst am Ende des Spiels (gewonnen oder verloren)! AlphaZero hat gegen einen der besten Go-Spieler der Welt gewonnen. Bemerkenswert ist der unglaublich große Zustandsraum von GO von etwa 10^{171}. Im Durchschnitt sind 250 Aktionen je Zug möglich und insgesamt liegt die mittlere Spieldauer bei ca. 150 Halbzügen. Daher galt es lange Zeit als nicht möglich für GO (im Gegensatz zu Schach) einen Agenten auf Meisterniveau zu entwickeln.

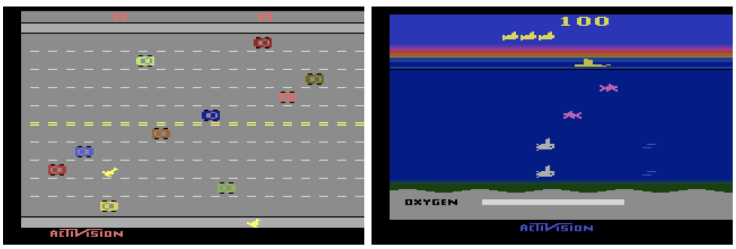

Abbildung 1: Zwei klasssische Atari-Spiele: *Freeway* und *Seaquest* aus der *Arcade Learning Environment* (BNVB12).

Eine bemerkenswerte Vielfalt von Problemen aus der mobilen Robotik kann als RL-Problem formuliert werden. Erst der Einsatz von DRL-Verfahren erlaubte es, Probleme mit einem größeren Zustandsraum zu lösen. Typische Beispiele sind das Lernen von Pick-And-Place- und andere Manipulations-Aufgaben mit einem Greifarm (siehe Abb. 2), das Lernen eines Laufverhaltens eines Laufroboters ((siehe Abb. 3) und die Fahrbahnverfolgung eines

autonomen Fahrzeugs (siehe Abb. 4). Ein Überblick findet sich in (KBP13), (Ama17), (ADBB17), (ITF+21) und (WLZ+20).

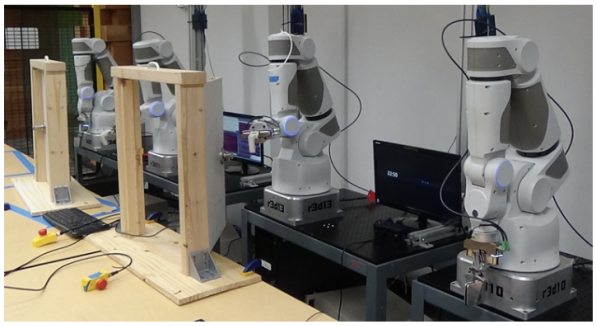

Abbildung 2: Zwei 7-DoF-Greifarmroboter, die ein Türöffnungsvorgang mit einer DQN-Variante lernen (GHLL17). Die Roboter kennen ihre Endeffektor-Pose und die Pose des Handknaufs.

Abbildung 3: Simulierter Vierbein-Roboter und humanoider Roboter, die mit DRL-Methoden das Laufen lernen (DCH+16). Der Vierbein-Roboter (linkes Bild) hat einen 8-dimensionalen und der humanoide Roboter (rechtes Bild) einen 28-dimensionalen Zustandsraum.

2 Motivation und Vorgehen

Schon seit vielen Jahren vertrete ich das Lehrgebiet *Mobile Robotik* im Bachelor *Angewandte Informatik* und im Master *Informatik*. Hinzugekommen ist seit SS 2021 eine neue Vorlesung über KI, bei der maschinelles Lernen ein zentraler Bestandteil ist. Aufgrund der jüngsten Erfolge im Bereich DRL und der vielfältigen Anwendbarkeit von DRL in der Robotik empfand ich es als wichtig und lohnenswert, das Gebiet des DRL zu erarbeiten, Anwendungen in der mobilen Robotik zu untersuchen und DRL in die Lehre zu integrieren.

Da die Lehre grundsätzlich praxisnah und mit motivierenden Beispielen begleitet sein sollte, wurde als Ausgangspunkt ein zweibeiniger Roboter (Abb. 5 (a)) gewählt. Dieser wurde in einem studentischen Projekt im Jahr 2020 entwickelt und war von einem einbeinigem Roboter der Hochschule Ravenburg-Weingarten inspiriert (Abb. 5 (c)). Da beim Reinforcement Learning der Roboter sehr lange mit der Umgebung interagieren muss, um genügend

Abbildung 4: Simuliertes Fahrzeug, das mit dem DRL-Verfahren *Deep Deterministic Policy Gradient* (DDPG) die Verfolgung einer Fahrbahnspur lernt (HLZ+21). Zu den Umgebungsinformationen des Fahrzeugs gehören die Längsgeschwindigkeit, die Quergeschwindigkeit, der Winkel zwischen der Fahrzeugrichtung und der Fahrbahn und der Abstand zur Mittellinie und zum Fahrbahnrand.

Erfahrung über den Erfolg von Aktionen zu sammeln, werden üblichweise simulierte Roboter eingesetzt. Auch wenn zusätzlich eine Hochleistungshardware eingesetzt wird, dauert bei hochdimensionalen Modellen wie z.B. einem humanoiden Roboter das Training in der Regel mehrere Stunden wenn nicht Tage. Bei unserem Krabbelroboter mit zwei Beinen mit je zwei Gelenken ergibt sich ein vergleichsweise kleiner 4-dimensionaler Zustandsraum. So konnte auch ohne Einsatz von Hochleistungshardware erwartet werden, dass die Trainingsdauer der Neuronalen Netze eher im Minutenbereich statt im Tagebereich liegen würde. Das war auch wünschenswert, da verschiedene RL- und DRL-Verfahren realisiert und untersucht werden sollten.

Aus diesen Überlegungen heraus wurde folgende Vorgehensweise gewählt.

1. Begonnen wurde mit der Entwicklung eines Simulationsmodells (siehe Abb. 5 (b)). Das zugrundeliegende mathematische Modell ist sehr einfach gehalten und geht von idealen Bedingungen wie starke Motorik ohne Reibungsverluste und ideal haftender Untergrund aus.
2. Da ein mathematisches Modell vorhanden war und der Zustandsraum von kleinerer Dimension ist, wurde zuerst Dynamisches Programmieren (DP) umgesetzt. DP hat den Vorteil, ein nachweislich optimales Verhalten zu erlernen. Das optimale Verhalten kann dann geschickterweise verwendet werden, um die anderen modellfreien Verfahren besser beurteilen zu können. Man beachte, dass DP bei einem realen Roboter aufgrund eines oft fehlenden Modells im Allgemeinen nicht möglich wäre.
3. Als nächstes wurde das tabellenbasierte Q-Learning realisiert, das ohne Modell auskommt. Hier wird deutlich werden, dass das Lernen in einem 4-dimensionaler Zustandsraum schon sehr aufwändig werden kann.

4. Als erstes DRL-Verfahren wurde das von DeepMind entwickelte DQN-Verfahren realisiert. Historisch war DQN eines der ersten sehr erfolgreichen Verfahren. DQN ist gewissermassen eine Variante des Q-Learnings. DQN lernt ähnlich wie das Q-Learning eine Wertefunktion, welche die in den verschiedenen Zuständen durchführbaren Aktionen bewertet. Das eigentliche Verhalten ergibt sich dann relativ einfach aus der gelernten Wertefunktion. Die Neuronalen Netze wurden mit dem *PyTorch*-Paket (PyT22) umgesetzt. Eine kleine Einarbeitungsphase in Tiefe Neuronale Netze war daher zuvor notwendig.
5. In der jüngsten Zeit sind DRL-Verfahren entwickelt worden, die direkt ein Verhalten lernen - ohne Umweg über eine Wertefunktion. Prominenter Vertreter ist das DRL-Verfahren *Advantage Actor Critic* (A2C) (MBM+16), das in einer abgewandelten Version auch bei ZeroAlpha zum Einsatz kam. Diese Verfahren neigen weniger zu einem Verhaltens-Bias wie die wertebasierte Verfahren, sind aber zeitaufwendiger. Um dieses Nachteil auszugleichen, wurde mit mehrere Simulatoren parallel trainiert (*Asynchronous Advantage Actor Critic*, A3C).

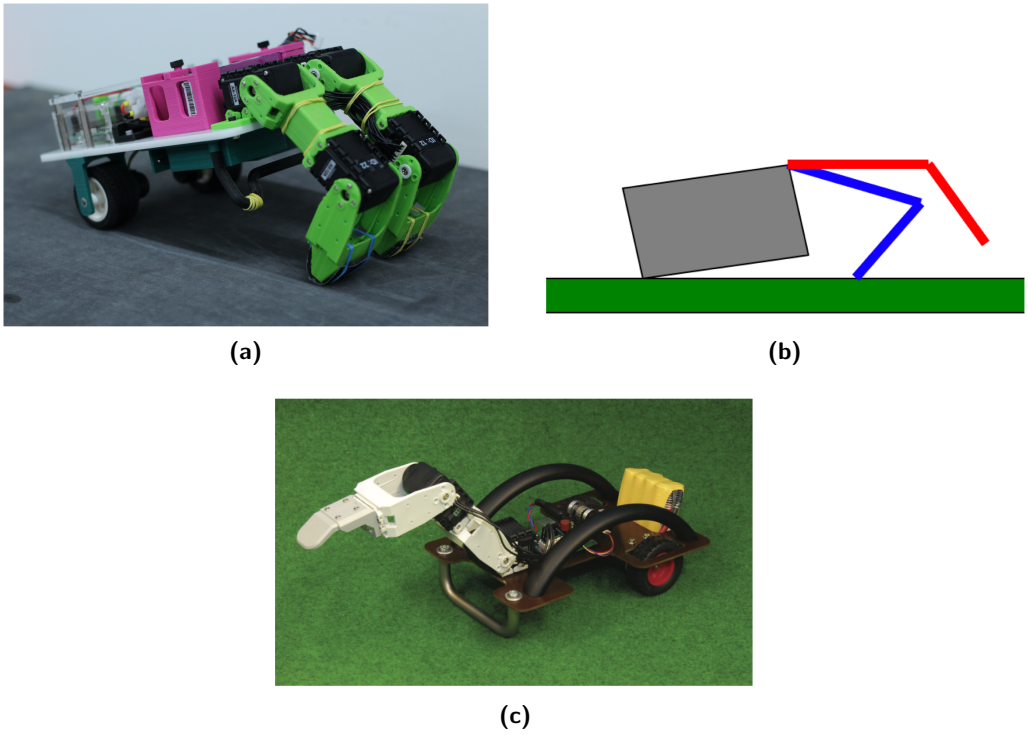

Abbildung 5: (a) An der HTWG Konstanz entwickelter zweibeiniger Krabbelroboter mit (b) Simulator. (c) An der HS Weingarten-Ravensburg entwickelter einbeiniger Roboter (TEF09).

Dieser Bericht beginnt zunächst mit den theoretischen Grundlagen zu RL wie Markov Decision Process (MDP) und Policies (Verhalten). Als Beispiel wird unser zweibeiniger Roboter herangezogen. Dann werden die vier oben genannten Verfahren vorgestellt. Die

erzielten Resultate werden verglichen und bewertet. Der Bericht schließt ab mit einer Darstellung der gewonnen Erfahrungen und den sich daraus ergebenden Konsequenzen für die Lehre.

3 Grundlagen

3.1 Markov Entscheidungsprozess und Krabbelroboter

Wir betrachten einen Agenten, der in diskreten Zeitschritten mit seiner Umgebung interagiert. In jedem Zeitschritt t beobachtet der Agent seinen *Zustand* $s_t \in \mathcal{S}$, führt eine *Aktion* $a_t \in \mathcal{A}$ aus und erhält im nächsten Zeitschritt einen reellwertigen *Reward* [1] (Belohnung) r_{t+1} und gelangt in einen Folgezustand s_{t+1} (siehe Abb. 6). $\mathcal{S}$ und $\mathcal{A}$ werden Zustands- bzw. Aktionsraum genannt. Der Zustand kann sowohl den internen Zustand des Agenten (z. B. Gelenkwinkel eines Greifarmroboters) als auch den Zustand der Umgebung umfassen. In den Verfahren, die hier betrachtet werden, sind der Zustands- und der Aktionsraum diskret.

Zustandsübergange als auch Rewards sind im Allgemeinen probabilistisch und werden durch eine Funktion $p(s', r|s, a)$ beschrieben. $p(s', r|s, a)$ gibt die bedingte Wahrscheinlichkeit des Folgezustands s' und des Rewards r an, falls der Agent im Zustand s die Aktion a auswählt (Notation ist an (SB18) angelehnt). Ein derartiges Agenten-Umgebungs-System wird auch *Markov Entscheidungsprozess* (Markov Decision Process, MDP) genannt.

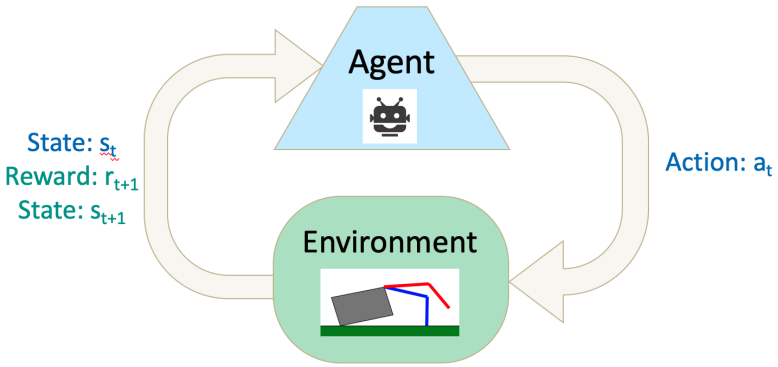

Abbildung 6: In jedem Zeitschritt t beobachtet der Agent seinen Zustand $s_t \in \mathcal{S}$, führt eine Aktion $a_t \in \mathcal{A}$ aus und erhält im nächsten Zeitschritt eine reellwertigen Reward r_{t+1} und gelangt in einen Zustand s_{t+1}.

Die Auswahl einer Aktion in einem Zustand wird vom Agenten durch eine sogenannte *Policy* (Verhalten) π gesteuert. Die Policy ist im allgemeinen ebenfalls probabilistisch und wird durch eine Funktion $\pi(a|s)$ beschrieben, die die Wahrscheinlichkeit angibt, dass die Aktion a im Zustand s gewählt wird. Wenn der Agent in einem Zustand s_1 gestartet wird,

[1] Einige Begriffe aus dem RL werden wir in Englisch belassen. Zum einen würden deutsche Übersetzungen den Sachverhalt nicht gut treffen und zum anderen sind fast alle Veröffentlichungen zu RL in englischer Sprache.

dann wird mit seiner Policy π eine sogenannte *Episode* bestehend aus Zustand, Aktion, Reward und Folgezustand generiert: $s_1, a_1, r_2, s_2, a_2, r_3, s_3, ...$ Die Summe der diskontierten Rewards, die der Agent ab einem Zustand s_t erhält, wird als *Return* R_t bezeichnet:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... \tag{1}$$

$\gamma < 1$ wird *Diskontierungsfaktor* genannt und üblicherweise auf $\gamma = 0.9$ gesetzt. Dadurch wird erreicht, dass der Return R_t immer endlich ist und weiter in der Zukunft liegende Rewards stärker abgeschwächt werden. Ziel des RL ist es, eine optimale Strategie π_* zu lernen, die immer Episoden generiert, die einen maximalen Return erwarten lassen.

$$\pi_* = \arg\max_{\pi} \mathbb{E}_\pi[R_t] \tag{2}$$

Dabei bezeichnet $\mathbb{E}_\pi[\cdot]$ den Erwartungswert einer Zufallsvariable unter der Voraussetzung, dass der Agent die Policy π verfolgt.

3.2 Krabbelroboter

Unser Krabbelroboter in Abb. 5 (a) und (b) dient als Beispiel eines MDP. Zustands- und Aktionraum sind diskret. Ein Zustand $s = (\theta_{l_1}, \theta_{l_2}, \theta_{r_1}, \theta_{r_2})$ beschreibt die Winkel der beiden Gelenken in beiden Beinen. l_1, l_2 bezeichnet die beiden Gelenke im linken Bein und r_1, r_2 die Gelenke im rechten Bein. Im ersten Beingelenk wurde ein Bewegungsbereich von 50 Grad mit einer Winkelauflöung (Diskretierung) von $\delta_1 = 10$ Grad und im zweiten Beingelenk ein Bewegungsbereich von 67.5 Grad mit einer Winkelauflöung von $\delta_2 = 7.5$ Grad festgelegt. Eine Aktion a ist ebenfalls ein 4-Tupel $a = (a_{l_1}, a_{l_2}, a_{r_1}, a_{r_2}) \in \{-1, 0, +1\}^4$, die für jedes Gelenk festlegt, ob es um $-\delta_1$, 0 oder $+\delta_1$ bzw. $-\delta_2$, 0 oder $+\delta_2$ gedreht werden soll. Damit ergibt sich eine Größe des Zustandsraums $\mathcal{S}$ und des Aktionsraums $\mathcal{A}$ von:

$$|\mathcal{S}| = 6 * 10 * 6 * 10 = 3600 \text{ und } |\mathcal{A}| = 3^4 = 81 \tag{3}$$

Die Zustandsübergänge sind sowohl im realen Modell als auch im Simulator deterministisch und lassen sich einfacherweise durch eine Funktion $\mathcal{T}(s, a) = s'$ beschreiben. Der Reward wird in einem ersten Ansatz durch den in einem Zeitschritt zurückgelegten Weg festgelegt. Im realen Modell wird der zurückgelegte Weg mit einem Drehencoder in der Hinterradachse gemessen. Die Messung hängt stark von der Griffigkeit des Untergrunds ab. Der entsprechende Reward ist daher probabilistisch mit unbekanntem Modell. In der Simulation wurde eine perfekte Griffigkeit vorausgesetzt. Der Reward ist damit deterministisch und kann als bekannte Funktion $\mathcal{R}(s, a) = r$ vorausgesetzt werden.

4 Dynamische Programmierung (DP)

Dynamische Prgrammierung (DP) umfasst eine Reihe von Algorithmen, die für MDP's eine optimale Policy berechnen können, wobei das Systemmodell (Funktion $p(s', r|s, a)$)

bekannt sein muss. Wir werden DP für unseren simulierten Krabbelroboter einsetzten, für den ein deterministisches Systemmodell vorliegt (siehe auch Abschnitt 3.2):

$$\text{Zustandsübergang } \mathcal{T}(s,a) = s' \text{ und Reward } \mathcal{R}(s,a) = r \tag{4}$$

Die zentrale Idee der DP ist die Benutzung einer *Value*-Funktion $V_\pi(s)$, die beschreibt, wie nützlich es ist, in Zustand s zu sein und dann die Policy π zu verfolgen. Gesucht ist eine optimale Value-Funktion V_*:

$$V_*(s) = \max_\pi V_\pi(s) \tag{5}$$

Auf Bellman (Bel57) geht zurück, dass sich V_* rekursiv charakterisieren lässt (Bellmansches Optimalitätsgleichung):

$$V_*(s) = \max_a [\mathcal{R}(s,a) + \gamma V_*(\mathcal{T}(s,a))] \tag{6}$$

Der optimale Wert eines Zustands s ergibt sich aus dem Reward $\mathcal{R}(s,a)$ und den diskontierten Wert des Folgezustands $\gamma V_*(\mathcal{T}(s,a))$ bei Ausführung der bestmöglichen Aktion a. γ ist der bereits eingeführte Diskontierungsfaktor (siehe Abschnitt 3.1). Ist die optimale Value-Funktion V_* bekannt, dann lässt sich daraus sehr einfach eine optimale Policy π_* ermitteln, die auch *greedy policy* genannt wird:

$$\pi_*(s) = \arg\max_a [\mathcal{R}(s,a) + \gamma V_*(\mathcal{T}(s,a))] \tag{7}$$

Algorithmisch läst sich V_* bestimmen, indem die Funktion $V_*(s)$ als große Tabelle (Feld) $V[s]$ implementiert wird und die Bellmansche Optimalitätsgleichung wiederholt angewendet wird, bis ein Konvergenzkriterium erreicht wird. Der Algorithmus ist als Pseudo-Code dargestellt und wird auch *Value Iteration* genannt (siehe Algorithmus 1).

Algorithm 1: Value Iteration zur Berechnung einer optimalen Policy π

input : Deterministisches Systemmodell $\mathcal{T}$ und $\mathcal{R}$
output: Optimale Policy π

```
1 Initialisiere Feld V[s] für jedes s ∈ S beliebig;
2 while Konvergenzkritzerium nicht erreicht do
3 |   for jedes s ∈ S do
4 |   |   V[s] ← max_a(R(s,a) + γ * V[(T(s,a))]);
5 |   end
6 end
7 gib greedy Policy π zurück:
8     π(s) = arg max_a(R(s,a) + γ * V[(T(s,a)])
```

Design der Reward-Funktion

Ganz entscheidend für das gelernte Verhalten ist die Definition der Reward-Funktion. In einem ersten Ansatz wurde eine Reward-Funktion definiert, die ein möglichst schnelles Vorwärtskommen des Agenten als Ziel definiert:

$$\mathcal{R}(s,a) = d \tag{8}$$

Dabei ist d der im Zustand s durch die Aktion a zurückgelegte Weg. Eine Evaluierung des gelernten Verhaltens zeigt einen zurückgelegten Weg von etwa 1.10 m bei 100 Zeitschritten. Dabei wurden ca. 31 Gehschritte durchgeführt (siehe youtu.be/txSWZV9jLn0). Es ist bemerkenswert, dass der Roboter den typischen wechselseitigen Gang (wie bei einem Menschen) und nicht ein Vorwärtshüpfen (wie bei einem Känguru) lernt. Etwas unschön ist die hohe Gehfrequenz (Trippelschritte). Daher wurde in einem zweiten Ansatz der Simulator mit einem Gehschritt-Zähler ausgestattet. Die Reward-Funktion wird erweitert um eine Bestrafung einer zu hohen Gehfrequenz:

$$\mathcal{R}(s,a) = d - 0.02 * n \tag{9}$$

Dabei ist n die Anzahl der Gehschritte und 0.02 ein empirisch ermittelter Gewichtungsfaktor. Als Gehschritt zählt die einmalige Berührung eines Beines mit dem Boden in einem Zeitschritt. Man beachte, dass die Zeitschritte mit 0.1 sec so klein sind, dass meistens $n = 0$ und in ganz wenigen Fällen $n = 1$ gemessen wird. Mit dieser Reward-Funktion wurde ein Verhalten gelernt, bei dem der Krabbelroboter 1.06 m zurücklegt, aber nur 11 Gehschritte durchführt (siehe Video youtu.be/9q_FlWlL8Qs). Dieses mit der Rewardfunktion (9) erzielte Gehverhalten wird als Referenz für die Beurteilung der weiteren Verfahren herangezogen.

5 Q-Learning

DP ist auf MDP's beschränkt, für die ein Systemmodell bekannt ist. Ist kein Systemmodell gegeben, dann muss der Agent durch Ausprobieren ein optimales Verhalten finden. Verfahren, die mit reinem Ausprobieren arbeiten, werden auch *Monte-Carlo-Verfahren* genannt. Monte-Carlo-Verfahren sind in der Praxis jedoch sehr ineffzient. Q-Learning kommt ebenfalls ohne Systemmodell und ist gewissermaßen eine Mischform aus DP und Monte-Carlo-Verfahren. Q-Learning gehört zu einer Klasse von Verfahren, die in der RL-Literatur auch *Temporal Difference Learning* (TD-Learning) genannt werden. Man beachte, dass bei unserem realen realen Krabbelroboter die Reward-Funktion nicht bekannt ist, und daher mit dem Q-Leaning- aber nicht mit dem DP-Verfahren trainiert werden kann.

Die zentrale Idee des Q-Learning ist die Benutzung einer *Action-Value*-Funktion $Q_\pi(s,a)$, die beschreibt, wie nützlich es ist, in Zustand s die Aktion a zu wählen und dann die Policy π zu verfolgen. Gesucht ist eine optimale Action-Value-Funktion $Q_*(s,a)$:

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \tag{10}$$

Der Agent lernt die optimale Action-Value-Funktion Q_* durch Interaktion mit der Umgebung. Dazu stellen wir uns vor, dass der Agent eine bereits (noch nicht optimal) gelernte Action-Value-Funktion Q durch eine Erfahrung verbessern möchte. Q wird als Tabelle $Q[s,a]$ verwaltet. Der Agent wählt mit Hilfe seiner Q-Tabelle im Zustand s eine Aktion a aus, führt sie aus, erreicht den Folgezustand s' und erhält einen Reward r (siehe auch Abb. 7).

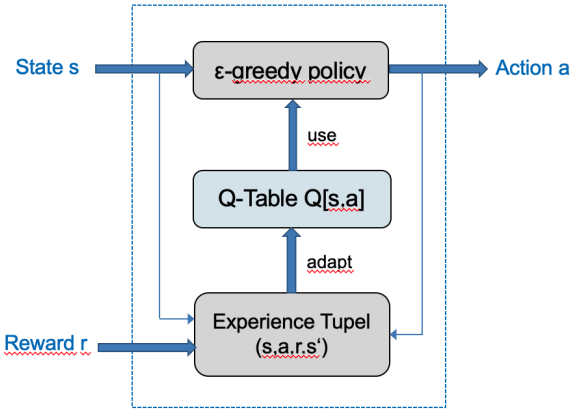

Abbildung 7: Q-Learning Agent.

Mit diesem Erfahrungstupel (s,a,r,s') verbessert der Agent die Tabelle Q:

$$Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a]) \tag{11}$$

Die Differenz $r + \gamma \max_{a'} Q[s',a'] - Q[s,a]$ drückt den Erfahrungsgewinn nach einem Zeitschritt aus und wird auch *Temporal Difference Error* (TD Error) genannt. α ist eine Lernrate und steuert, wie stark neue Erfahrungen berücksichtigt werden. Bei $\alpha = 1$ wird $Q[s,a]$ komplett durch die neue Erfahrung ersetzt während bei einem kleineren α eine Mischung aus altem Q-Wert und neuer Erfahrung gewählt wird. Erfahrungsgemäß wird $\alpha = 0.1$ gewählt.

Der Q-Learning-Algorithmus ist in Algorithmus 2 als Pseudo-Code dargestellt. Der Algorithmus generiert mehrere (einige Tausend) Episoden, wobei jede Episode auf N Zeitschritte (z.B. $N = 100$) begrenzt ist. Mit dem ϵ-Greedy-Schritt in Zeile 5 bis 7 wird mit einer Wahrscheinlichkeit von $P < \epsilon$ eine zufällige und sonst eine nach dem bisherigem Wissen beste Aktion gewählt (greedy policy) gewählt. Damit findet eine Abwägung zwischen Erkunden und Verwerten statt (siehe nächsten Abschnitt).

Algorithm 2: Q-Learning zur Berechnung einer optimalen Policy π

input : Beliebiger MDP-Agent. Systemmodell muss nicht bekannt sein.
output: Optimale Policy π

Initialisiere Feld $Q[s, a]$ für jedes $s \in \mathcal{S}$ und $a \in \mathcal{A}$ beliebig;
repeat
  initialisiere Zustand s beliebig;
  for *jeden Schritt in der Episode* **do**
    generiere zuällige Zahl $r \in [0, 1]$;
    if $r < \epsilon$ **then** wähle zufällige Aktion $a \in \mathcal{A}$;
    else wähle $a = \arg\max_a Q[s, a]$; // Greedy Policy
    Führe Aktion a aus und beobachte Reward r und Folgezustand s';
    $Q[s, a] \leftarrow Q[s, a] + \alpha(r + \gamma * \max_{a'} Q[s', a'] - Q[s, a])$;
    $s \leftarrow s'$;
  end
until *die Maximalanzahl von Episoden ist erreicht*;
gib greedy Policy π zurück:
  $\pi(s) = \arg\max_a Q[s, a]$

Erkunden vs. Verwerten

Eine der großen Herausforderungen bei RL-Verfahren ist die richtige Balance zwischen Verwerten (Exploitation) und Erkunden (Exploration) zu finden. Einerseits sollte der Agent auf bereits in der Vergangenheit bewährte Aktionen aufbauen (Verwerten). Andererseits sollte der Agent auch neues, unbekanntes Terrain untersuchen (Erkunden), um bessere Aktionen finden zu können. Im Algorithmus 2 wird die Balance durch den Parameter ϵ gesteuert. Diese Vorgehensweise wird auch ϵ-*greedy Strategie* genannt. Die Erfahrung hat gezeigt, dass es außerdem günstig ist, im Lernverfahren mit einem hohen ϵ-Wert zu beginnen und den Wert über die Zeit abzusenken (decaying ϵ) (siehe Abb. 8).

6 Deep Q-Network (DQN)

Q-Learning hat den gravierenden Nachteil, dass die Größe der Q-Tabelle exponentiell mit der Dimensionsgröße des Zustands- und Aktionsraums wächst. Zusätzlich wird die Tabellengröße durch die Feinheit der Diskretisierung der Zustände und Aktionen bestimmt (siehe auch Gleichung (3)). Selbst bei unserem vergleichsweise einfachen Krabbelroboter und der festgelegten Auflösung erreicht die Tabelle eine Größe von 3600*81 = 291600 Einträge. Es ist dann nicht mehr gewährleistet, dass Q-Learning in vernünftiger Zeit alle Tabelleneinträge hinreichend gut lernt. Hier bietet sich ein Approximationsverfahren an, dass nicht gelernte Tabellenbereiche aus gelernten Bereichen approximiert. Ein besonders erfolgreich eingesetzter Approximator sind Neuronale Netze. Wir ersetzen daher die Q-Tabellen durch Neuronale Netze und erhalten das sogenannte DQN-Verfahren. Im nächsten Abschnitt wird zunächst kurz auf Neuronale Netze und ihre Umsetzung mit PyTorch eingegangen.

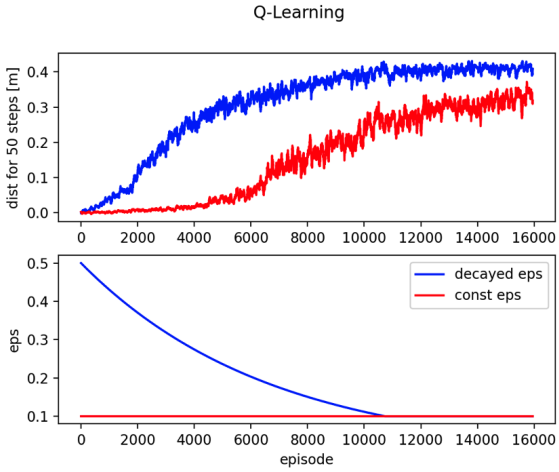

Abbildung 8: Abwägung zwischen Erkunden und Verwerten. Mit einem exponentiell abnehmenden ϵ (blaue Kurven unten) lernt der Agent wesentlich schneller ein schnelles Gehen (blaue Kurve oben). Mit einem konstanten ϵ ist der Lernfortschritt wesentlich kleiner (rote Kurven).

6.1 Neuronale Netze mit PyTorch

PyTorch ist eine Open Source Framework zur Erstellung Neuronaler Netze mit Python (PyT22). Da wir unsere Lernverfahren mit Python realisiert haben und PyTorch in der RL-Community stark verwendet wird, war die Entscheidung für PyTorch zwangsläufig. Sehr hilfreich bei der Umsetzung der Lernverfahren mit PyTorch war (Mor20).

Einfache mehrschichtige Neuronale Netze[2] bestehen aus einer Folge von Schichten, wobei jede Schicht aus einer Folge von Neuronen besteht (siehe Abb. 9 (a)). Die erste Schicht wird Eingabeschicht und die letzte Ausgabeschicht genannt. Jedes Neuron einer Schicht ist mit allen andern Neuronen der Folgeschicht mit einer gewichteten Kante verbunden. Die Eingabeschicht wird mit einem Eingabevektor x gespeist. Jedes Neuron berechnet eine gewichtete Summe und wendet dann eine sogenannte Aktivierungsfunktion an (Abb. 9 (b)). Als Ergebnis wird ein Ausgabevektor y in der Ausgabeschicht produziert.

Das Neuronale Netz berechnet letztendlich eine parameterisierte Funktion f_θ, wobei der Parametervektor θ alle Gewichte des Netzes zusammenfasst:

$$y = f_\theta(x) \tag{12}$$

Ziel beim Lernen ist es nun, die Gewichte θ so anzupassen, dass der Fehler zwischen Ausgabe y und einem Ziel (target) t minimiert wird. Die Anpassung der Gewichte geschieht durch Minimierung einer Fehlerfunktion $L(\theta) = (y - t)^2$ durch Gradientenabstieg:

[2] Hier werden nur die einfachen mehrschichtigen Netze (multi-layer perceptrons) betrachtet. Fortgeschrittene Netze wie z.B. Faltungsnetzwerke (convolutional networks) und LSTM-Netzwerke (long short-term memory networks), die bei der Verarbeitung von Bildern und Sprache wichtig sind, werden hier nicht betrachtet.

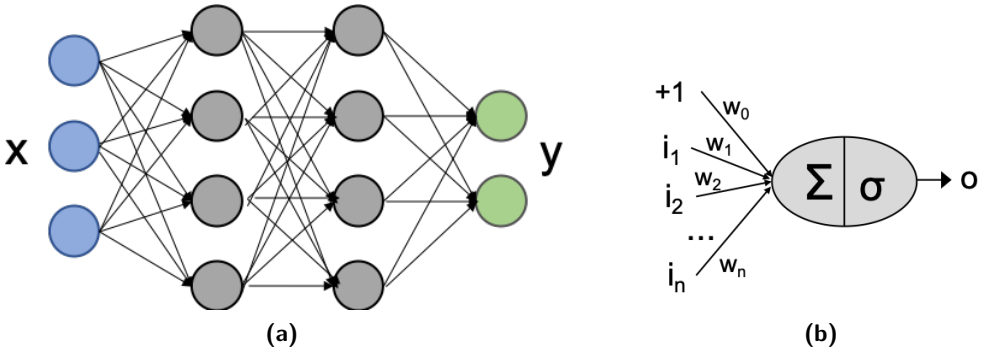

Abbildung 9: (a) Neuronales Netz mit 4 Schichten. Die Eingabeschicht besteht aus 3 Neuronen, die beiden verdeckten Schichten aus jeweils 4 Neuronen und die Ausgabeschicht aus 2 Neuronen. Man spricht auch von einem 3-4-4-2-Netz. (b) zeigt ein einzelnes Neuron, das für eine Eingabe $i = (i_1, i_2, ..., i_n)$ eine gewichtete Summe berechnet und dann eine Aktivierungsfunktion σ anwendet: $o = \sigma(\sum_{k=1}^{n} w_i i_k + w_0)$. w_0 ist ein Bias.

$$\Delta\theta = -\eta \cdot \nabla_\theta L(\theta) \tag{13}$$

η ist eine Lernrate. Die Berechnung der partiellen Ableitungen lassen sich durch das Backpropagation-Verfahren (siehe (RHW86)) besonders effizient lösen. In folgendem Listing wird das Netz aus Abb. 9 definiert, auf eine Eingabe x angewendet und für eine Zielausgabe t trainiert. Das Neuronale Netz wird als Klasse definiert, wobei im Konstruktor (Zeile 6 - 12) die Netzarchitektur (3-4-4-2-Netz) und die Aktivierungsfunktionen festgelegt werden. In diesem Beispiel wird das Netz nur für ein Zielvektor t trainiert (Zeile 34 - 36). Üblichweise erfolgt das Trainieren des Netzes für mehre Zielvektoren gleichzeitig.

```
1  import torch
2  import torch.nn as nn
3
4  class NeuralNetwork(nn.Module):
5      def __init__(self):
6          super().__init__()
7          self.fc1 = nn.Linear(3, 4)
8          self.act1 = nn.ReLU()
9          self.fc2 = nn.Linear(4, 4)
10         self.act2 = nn.ReLU()
11         self.out = nn.Linear(4, 2)
12         self.final = nn.Sigmoid()
13
14     def forward(self, x):
15         op = self.fc1(x)
16         op = self.act1(op)
17         op = self.fc2(op)
18         op = self.act2(op)
19         op = self.out(op)
20         y = self.final(op)
21         return y
22
23 # Netz defininieren
24 model = NeuralNetwork()              # 3-4-4-2-Netz
25 loss_function = nn.MSELoss()        # Fehlerfunktion
26 optimizer = torch.optim.Adam(model.parameters(), lr= 0.001) # Optimierungsverfahren
```

```
27
28 # Netz trainieren
29 optimizer.zero_grad()                # Gradienten zuruecksetzen
30 model.train()                        # Trainingsmodus
31 x = torch.tensor([1.0, 0.5, 2.0])    # Eingansvektor
32 t = torch.tensor([1.0, 1.0])         # Zielvektor
33 y = model(x)                        # Ausgabe berechnen
34 loss = loss_function(t, y)          # Fehler berechnen
35 loss.backward()                     # Backpropagation
36 optimizer.step()                    # Gewichte anpassen
37
38 # Netz anwenden:
39 model.eval()                        # Evaluierungsmodus
40 print(model(x))                     # Neue Netzausgabe ausgeben
```

6.2 DQN-Verfahren

Die zentrale Idee ist die Benutzung eines Neuronalen Netzes statt einer Q-Tabelle. Dieses Netz wird auch Q-Netz q_θ genannt, wobei θ die Gewichtsparameter sind. q_θ liefert für einen Zustand s eine Ausgabe $q_\theta(s)$. $q_\theta(s)$ ist ein Vektor und enthält für jede Aktion a den Q-Wert. Wir schreiben dafür kurz $q_\theta(s, a)$ (siehe Abb. 10).

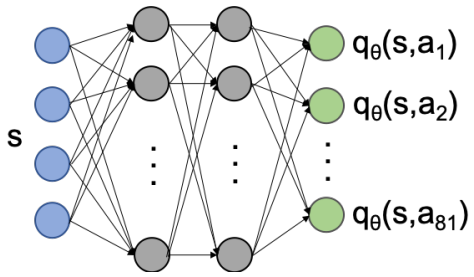

Abbildung 10: Beispiel eines Q-Netzes q_θ. Die Eingabe ist ein Zustand s. Die Größe der Eingabeschicht ist daher gleich der Dimension des Zustandsraums. Die Ausgabeschicht enthält für jede Aktion ein Neuron. Da bei unserem konkreten Krabbelroboter 81 unterschiedliche Aktionen möglich sind $(a_1, a_2, ...a_{81})$, gibt es entsprechend viele Ausgabeneuronen. Die Ausgabe des Neurons für die Aktion a_i wird $q_\theta(s, a_i)$ genannt und beschreibt den Q-Wert $Q(s, a_i)$.

Zwei besondere Merkmale des DQN-Verfahren sind für seinen Erfolg verantwortlich:

1. *Experience Replay*: Wenn der Agent ein Erfahrung (s, a, r, s') macht, dann ist es sinnvoll dieses Erfahrungstupel über die Zeit mehrere Male als Trainingsdaten zu verwenden. Dazu wird ein *Replay Buffer* als FIFO-Liste verwaltet, in der neue Erfahrungstupel abgespeichert und bei Überlauf die ältesten Erfahrungen gelöscht werden. Aus diesem Replay Buffer wird ein *Mini Batch* (kleine Teilmenge von Erfahrungstupel) für das Training zufällig ausgewählt. Da die Erfahrungstupel randomisiert gezogen werden, werden Korrelationen bei zeitlich aufeinanderfolgenden Erfahrungstupeln vermieden.
2. *Target Network:* Neuronale Netze - wie im vorigen Abschnitt dargestellt - werden überwacht trainiert. Es muss ein Zielvektor t (in unserem Fall eine Bewertung aller

Aktionen) vorgeben werden. Andererseits soll t durch die RL-Verfahren ja gerade gelernt werden. Eine zunächst naheliegene Lösung besteht einfach darin, das Neuronale Netz selbst zu nehmen, um ein Zielvektor zu produzieren. Es zeigt sich jedoch, dass das Lernen dann sehr instabil wird, weil Ausgabe y und Ziel t zu stark korreliert sind. Die vorgeschlagene Lösung ist ein zweites Netz - das sogenannte *Target-Netzwerk* - zu verwenden, um die Zielvektoren zu produzieren. Mit diesen generierten Zielvektoren wird das eigentliche Netzwerk - auch *Online-Netzwerk* genannt - trainiert. Das Target-Netzwerk dagegen wird nicht trainiert, sondern passt seine Gewichte verzögert an die Gewichte des Online-Netzwerks an.

Abb. 11 zeigt die Struktur und Funktionsweise des DQN-Agenten. Der DQN-Agent ist mit einem Replay Buffer ausgestattet und verfügt über ein Online-Netzwerk q_θ und ein Target-Netzwerk q_{θ^-} mit gleicher Architektur aber unterschiedlichen Gewichten. Nur das Online-Netzwerk wird trainiert. In jedem Zeitschritt wählt der Agent mit dem Online-Netzwerk eine Aktion a mit einer ϵ-Greedy-Strategie aus. Er beobachtet den Folgezustand s' und den Reward r. Das Erfahrungstupel (s, a, r, s') wird in den Replay Buffer abgelegt. In jedem n-ten Zeitschritt (z.B. n = 10), wird ein Mini-Batch zufällig aus dem Replay-Buffer gezogen. Mit Hilfe des Target-Netwerks werden die Parameter θ des Online-Netzwerks angepasst. Die prinzipielle Idee ist wie beim TD Learning (siehe (11)). Da jedoch eine Neuronales Netz trainiert wird, wird eine geeignete Fehlerfunktion $L(\theta)$ definiert, die mit dem dem Backpropagation-Verfahren minimiert wird. Für ein Erfahrungstupel (s, a, r, s') ist die Fehlerfunktion $L(\theta)$ wie folgt definiert:

$$L(\theta) = (r + \gamma \max_{a'} q_{\theta^-}(s', a') - q_\theta(s, a))^2 \tag{14}$$

Die Gewichte des Target-Netwerks werden verzögert vom Online-Netzwerk kopiert (soft update), was durch folgende Zuweisung mit einem kleinen Wert für Parameter τ (z.B. $\tau = 0.1$) geleistet wird:

$$\theta^- \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta^- \tag{15}$$

7 Advantage Actor Critic (A2C) und Asynchronous Advantage Actor Critic (A3C)

Die in den letzten Abschnitten vorgestellten Methoden finden optimale oder nahezu optimale Policies mit Hilfe von Value-Funktionen. In diesem Abschnitt stellen wir eine Methode vor, die eine Policy direkt lernt. Diese Methode gehört zur großen Kasse der Policy-basierten Verfahren. Eine besonders wichtige und erfolgreiche Variante sind die *Actor-Critic*-Methoden. Actor-Critic-Methoden verwenden ein Policy-Netzwerk, um eine möglichst gute Aktion zu wählen, und ein Value-Netzwerk, um die Policy zu bewerten und zu optimieren. Das Policy-Netzwerk wird auch *Actor* und das Value-Netzwerk *Critic* genannt.

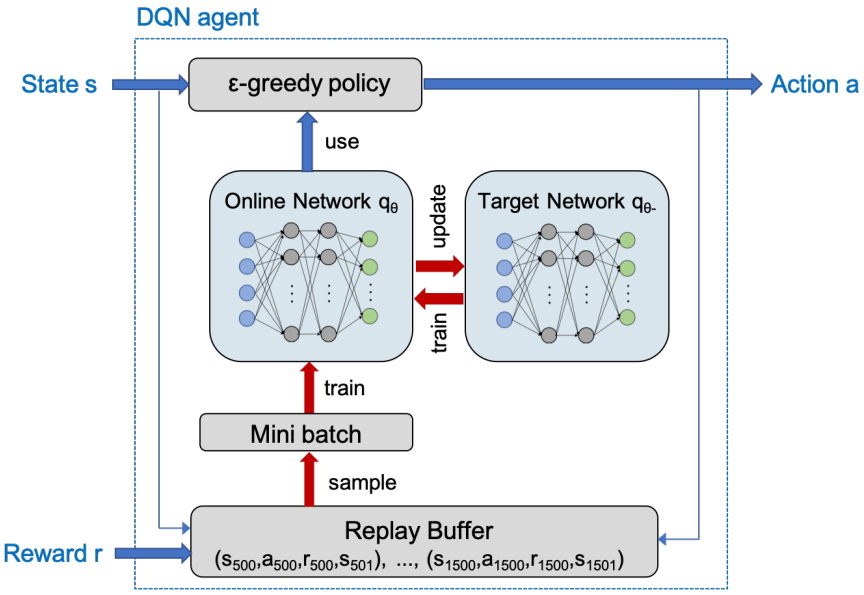

Abbildung 11: DQN-Agent mit Replay Buffer, Online-Netzwerk q_θ und Target-Netzwerk q_{θ^-}. Beschreibung siehe Text.

Algorithm 3: DQN-Lernverfahren zur Berechnung einer optimalen Policy π

input : Beliebiger MDP-Agent. Systemmodell muss nicht bekannt sein.
output: Optimale Policy π

n_s = Dimension von $\mathcal{S}$;
$n_a = |\mathcal{A}|$;
initialisiere ein Online Netzwerk q_θ mit n_s Eingabe- und n_a Ausgabe-Neuronen;
initialisiere ein strukturgleiches Target-Netzwerk q_{θ^-};
initialisiere Replay Buffer rb als FIFO-Liste;
repeat
  initialisiere Zustand $s \in \mathcal{S}$ zufällig;
  for *jeden Schritt in der Episode* **do**
    generiere zufällige Zahl $r \in [0, 1]$;
    if $r < \epsilon$ **then** wähle zufällige Aktion $a \in \mathcal{A}$;
    else wähle $a = \arg\max_a q_\theta(s, a)$; // Greedy Policy
    Führe Aktion a aus und beobachte Reward r und Folgezustand s';
    Füge Erfahrungstupel (s, a, r, s') zum Replay Buffer rb dazu;
    ziehe eine zufällige Teilmenge $(s^{(i)}, a^{(i)}, r^{(i)}, s'^{(i)})$ (Mini-Batch) aus rb;
    Berechne Fehler $L(\theta)$ mit Hilfe des Target-Netzwerk:
      $L(\theta) = \sum_i (r^{(i)} + \gamma \max_{a'} q_{\theta^-}(s'^{(i)}, a') - q_\theta(s^{(i)}, a^{(i)}))^2$;
    Minimiere Fehler $L(\theta)$ mit Gradientenabstieg;
    Soft-Update des Target-Netzwerks: $\theta^- \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta^-$;
    $s \leftarrow s'$;
  end
until *die Maximalanzahl von Episoden ist erreicht*;
gib greedy Policy π zurück:
  $\pi(s) = \arg\max_a q_\theta(s, a)$

Policy-basierte Methoden haben den Vorteil, dass die gelernten Policies einen geringen Bias aufweisen. Jedoch weisen sie eine hohe Varianz auf, da vergleichsweise wenig Trainingsdaten anfallen. Eine naheliegende Verbesserung ist daher der Einsatz parallel lernender Agenten, die sich ihr Wissen (d.h. Policy und Value-Netzwerk) teilen. Diese Verfahren werden auch *Asynchronous Advantage Actor Critic* (A3C) genannt.

Das Policy-Netzwerk π_θ berechnet ein probabilistisches Verhalten, indem es als Eingabe einen Zustand s nimmt und als Ausgabe eine Wahrscheinlichkeitsverteilung $\pi_\theta(s)$ aller Aktionen liefert. Wir verwenden die Schreibweise $\pi_\theta(a|s) = P(a|s)$ für die Wahrscheinlichkeit die Aktion a im Zustand s zu wählen (siehe Abb. 12 (a)). Das Value-Netzwerk V_ω berechnet für jeden Zustand den erwarteten Return für die aktuelle Policy.

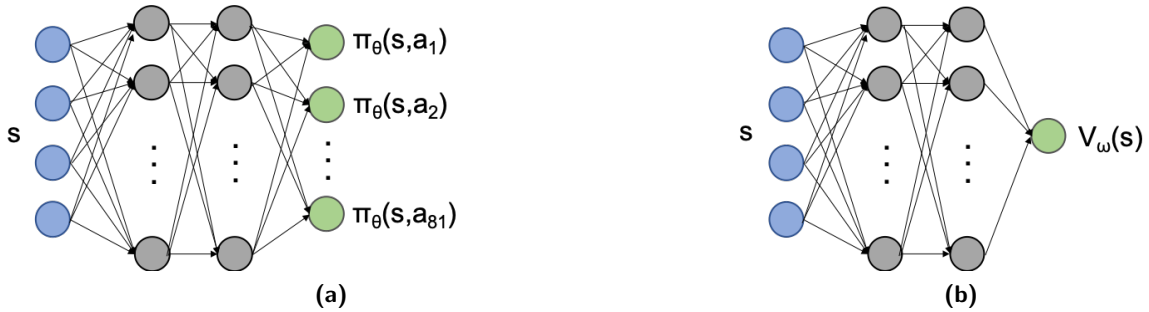

Abbildung 12: (a) Policy-Netzwerk π_θ (Actor) mit 4 Schichten für unseren Krabbelroboter. θ sind die Gewichte des Netzes. Das Policy-Netzwerk berechnet für jeden Zustand s und jeder Aktion a die Wahrscheinlichkeit $\pi_\theta(a|s) = P(a|s)$, dass die Aktion a im Zustand s gewählt wird. (b) Value-Netzwerk V_ω, wobei ω die Gewichte des Netzes sind.

Um das Policy-Netwerk π_θ zu trainieren führt der Agent mit seiner Policy π_θ eine komplette Episode mit N-Schritten durch:

$$s_1, a_1, r_2, s_2, a_2, r_3, s_3, ..., s_{N-1}, a_{N-1}, r_N, s_N \tag{16}$$

Danach wird für Aktionen, die vorteilhaft waren, die Wahrscheinlichkeit gewählt zu werden, erhöht. Dazu wird eine Performancefunktion $J(\theta)$ definiert, die mit Hilfe des PyTorch-Pakets durch Gradientenaufstieg maximiert wird. Die theoretische Grundlage dabei ist logistische Regression.

$$J(\theta) = \sum_{t=1}^{N-1} \log \pi_\theta(a_t|s_t) A_\omega(s_t, a_t) \tag{17}$$

$A_\omega(s_t, a_t)$ ist die sogenannte *Advantage*-Funktion, die den Fortschritt der aktuellen Episode bewertet, indem der aktuelle Return (neue Erfahrung) mit der Wert des Value-Netzwerks V_ω (alte Erfahrung) verglichen wird.

$$A_\omega(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{N-t-1} r_N + \gamma^{N-t} V_\omega(s_N) - V_\omega(s_t) \tag{18}$$

Das Value-Netzwerks V_ω soll möglichst gut die Returns berechnen. V_ω wird trainiert, in dem die die Fehlerfunktion $L(\omega)$ minimiert wird.

$$L(\omega) = \sum_{t=1}^{N-1} A_\omega(s_t, a_t)^2 \tag{19}$$

In Algorithmus 4 sind alle Schritte zusammengefasst. In Zeile 8 wird die jeweils nächste Aktion a_i probabilistisch ausgewählt. Dabei ist Ausgabe des Policy-Netzwerks $\pi_\theta(s_t)$ eine Wahrscheinlichkeitsverteilung aller Aktionen. Um die Exploration des Agenten zu unterstützen, wird in der Zeile 12 in der Performancefunktion $J(\theta)$ noch ein mit β gewichteter Entropieterm $H(\pi_\theta(s_t))$ eingeführt. Der Entropie H sorgt dafür, dass die Aktionen möglichst gleichwahrscheinlich bleiben. Die Entropie wird über die Zeit reduziert, so dass der Agent mit zunehmender Zeit sein gelerntes Wissen verstärkt verwendet (Exploitation).

Algorithm 4: A2C-Lernverfahren zur Berechnung einer optimalen Policy π

input : Beliebiger MDP-Agent. Systemmodell muss nicht bekannt sein.
output: Optimale Policy π

1 n_s = Dimension von $\mathcal{S}$;
2 $n_a = |\mathcal{A}|$;
3 initialisiere ein Policy Netzwerk π_θ mit n_s Eingabe- und n_a Ausgabe-Neuronen;
4 initialisiere ein Value-Netzwerk V_ω mit n_s Eingabe- und einem Ausgabe-Neuron;
5 **repeat**
6 initialisiere Zustand $s_1 \in \mathcal{S}$ zufällig;
7 **for** $t \leftarrow 1$ **to** $N-1$ **do** // führe eine N-Schritt-Episode durch
8 wähle $a_t \sim \pi_\theta(s_t)$;
9 Führe Aktion a_t aus und beobachte Reward r_{t+1} und Folgezustand s_{t+1};
10 **end**
11 Berechne Performance-Funktion $J(\theta)$:
12 $J(\theta) = \sum_{t=1}^{N-1} [\log \pi_\theta(a_t|s_t) A_\omega(s_t, a_t) + \beta H(\pi_\theta(s_t))]$;
13 Berechne Fehler-Funktion $L(\omega)$ (A_ω siehe Gleichung (18)):
14 $L(\omega) = \sum_{t=1}^{N-1} A_\omega(s_t, a_t)^2$;
15 Maximiere $J(\theta)$ und minimiere $L(\omega)$ mit Gradientenverfahren;
16 **until** *die Maximalanzahl von Episoden ist erreicht*;
17 gib greedy Policy π zurück:
18 $\pi(s) = \arg\max_a \pi_\theta(s|a)$

Das reine A2C-Verfahren hat den Nachteil, dass vergleichsweise wenig trainiert wird: die Erfahrungen einer Episode werden nur einmal zum Trainieren verwendet (Zeile 15). Daher ist die Lernperformance eher schwach. Um diesen Nachteil zu beheben, wird der A2C-

Algorithmus mehrfach parallel mit einem gemeinsamen Policy- und Value-Netzwerk gestartet (siehe Abb. 13). Dabei muss der parallele Zugriff auf die gemeinsamen Netzdaten synchronisiert werden. Das lässt sich in PyTorch einfach realisieren, in dem auf die Netze die Methode `share_memory()` angewendet wird:

```
actor = PolicyNetwork(...)
actor.share_memory()     # shared among processes
critic = ValueNetwork(...)
critic.share_memory()    # shared among processes
```

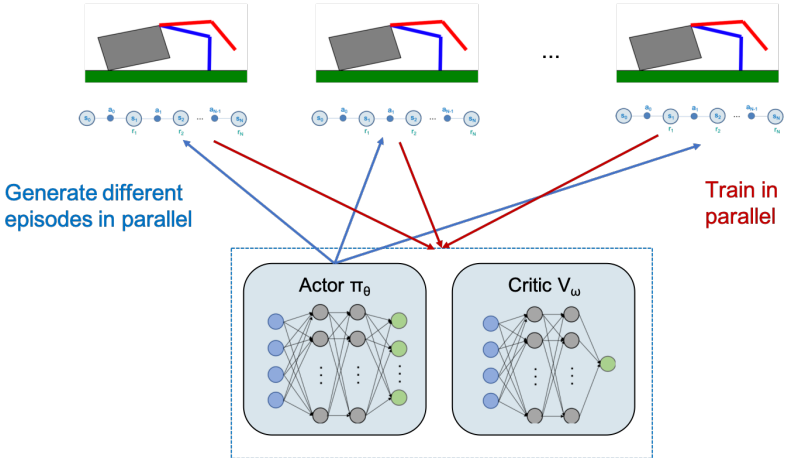

Abbildung 13: A3C-Agent. Der A2C-Algorithmus wird mehrfach parallel mit einem gemeinsamen Policy- und Value-Netzwerk gestartet.

8 Ergebnisse

Es wurden die 4 dargestellten Verfahren DP, Q, DQN und A3C implementiert und auf unseren Krabbelroboter angewendet. Der Zustands- und der Aktionsraum des Roboters wurde wie in Abschnitt 3.2 beschrieben diskretisiert. Damit hat der Zustandsraum die Größe 3600 und der Aktionsraum die Größe 81. Als Reward-Funktion wurde $\mathcal{R}(s, a)$ aus (9) gewählt. Daher wird ein Verhalten belohnt, das den Roboter möglichst schnell mit möglichst wenig Gehschritten bewegt (keine Trippelschritte). Bei den DRL-Verfahren sind sehr viele Netz- und Lernparameter einzustellen, so dass wir zunächst empirisch eine gute Wahl festgelegt und dann die Verfahren 5-mal mit unterschiedlich initialisierten Tabellen und Neuronalen Netzen trainiert haben. Die besten Ergebnisse sind in der Tabelle 1 festgehalten.

Der Krabbelroboter kann sich in Randzustände befinden, in denen nicht alle Aktionen erlaubt sind. In den Randzuständen sind manche Gelenkwinkel am Anschlag und dürfen nur in eine Richtung bewegt werden. In unserem Modell werden nicht erlaubte Aktionen nicht durchgeführt und ein Reward von $r = 0$ zurückgeliefert. Q, DQN und A3C lernen ein deterministische Verhalten (greedy-Strategie; siehe jeweils letzte Zeile in den Algorithmen). Falls sich der Roboter in einem Randzustand befindet, für den eine nicht erlaubte Aktion

als beste Aktion gelernt wurde, bleibt der Roboter aufgrund seines deterministischen Verhaltens stehen! Eine solche Episode wird als nicht erfolgreich gewertet. Die letzte Spalte der Tabelle zeigt die Erfolgsrate.

Q-Learning ist mit Abstand sowohl bei der Verhaltensperformance (Laufdistanz und Anzahl Gehschritte) als auch bei der Größe der Datenstruktur (hier Tabellengröße) am schlechtesten. A3C und DQN sind in etwa gleich performant, wobei A3C eine etwas bessere Erfolgsrate (99% statt 97%) hat. Im Vergleich zu dem durch DP gelernten optimalen Verhalten (siehe Video youtu.be/9q_FlWlL8Qs) sind die von DQN und A3C gelernte Verhalten fast gleich gut.

Verfahren	Tabelle od. Neuronales Netz	Hyperparameter Anz.	Einige wichtige Parameter	Trainingszeit in CPU [sec]	Mittlere Laufdistanz in m	Mittlere Anz. Gehschritte	Erfolgsrate in %
DP	V-Tabelle Gr = 3600	2		174	1.06	11.1	100
Q	Q-Tabelle Gr = 291600	7	16000 Episoden 50 Zeitschr./Episode	576	0.97	18.6	97
DQN	Online-Netzwerk Target-Netzwerk 4-128-128-81 Gew ≈ 27000	11	16000 Episode 50 Zeitschr./Episode Training jeden 10. Zeitschr. Replay Buffer size = 16000 Mini Batch size = 512	1092	1.06	12.0	97
A3C	Policy-Netzwerk: 4-128-128-81 Gew. ≈ 27000 Value-Netzwerk: 4-128-128-1 Gew ≈ 17000	7	36000 Episode 10 Zeitschr./Episode Anz. paralleler Prozesse = 8	1178	1.07	12.4	99

Tabelle 1: Vergleich der verschiedenen Lernverfahren. Gr = Tabellengröße und Gew = Anzahl Netzwerkgewichte. Das mit DP gelernte Verhalten ist nachweislich optimal und dient als Vergleich. Bei Q, DQN und A3C wurden 5 Trainingsversuche mit zufällig initialisierten Tabellen bzw. Neuronale Netze durchgeführt und das beste Ergebnis hier dargestellt. Laufdistanzen und Gehschritte beziehen sich auf 100 Zeitschritte (=10 sec) und sind gemittelt über alle Zustände. Die Erfolgsrate gibt den Anteil erfolgreicher Episoden an. Weitere Erklärungen im Text.

9 Resümee

9.1 Zusammenfassung und Ausblick

RL-Agenten lernen ein optimales Verhalten durch Interaktion mit der Umgebung. Klassische RL-Agenten speichern ihr Verhalten in Tabellen während DRL-Agenten ihr gelerntes Verhalten in Neuronale Netze speichern. In diesem Bericht wurden zunächst die Grundlagen von RL dargestellt und 4 verschiedene Verfahren implementiert und untersucht: DP, Q-Learning , DQN und A3C. DP ist ein klasssisches Verfahren, das ein Modell des Agenten und seiner Umgebung benötigt, dafür aber ein nachweislich optimales Verhalten berechnet. Die anderen Verfahren benötigen dagegen kein Modell, was für die Praxis sehr wichtig ist. Q-Learning hat den Nachteil, das Verhalten tabellenbasiert zu lernen, was schon bei kleineren Zustands- und Aktionsraumdimensionen zu beträchtlichen Tabellengrößen führt. Der

Lernerfolg leidet unter einer zu großen Tabellengröße, da beim Lernen kaum alle Tabellenbereiche hinreichend abgedeckt werden. Demgegenüber haben die DRL-Verfahren DQN und A3C, die mit Neuronalen Netzten arbeiten, diese Probleme wegen ihrer Generalisierungsfähigkeit nicht. DQN und A3C sind gegenüber der Dimensionsgröße der Zustands- und Aktionsräume des Agenten sehr robust.

In unserer Arbeit konnten wir die Verfahren für unseren Krabbelroboter in Python umsetzen. Für die Neuronalen Netze kam das PyTorch-Paket zum Einsatz. Das Lernen wurde an einem Simulator durchgeführt. Da ein Modell bekannt war, konnte das DP-Verfahren genutzt werden, um ein optimales Verhalten zu lernen. Es hat sich gezeigt, dass die mit DQN und A3C gelernten Verhalten gleichwertig im Sinne von Bewegungsgeschwindigkeit und Schrittgröße des Krabbelroboters (keine Trippelschritte) sind. DQN ist ein Value-basiertes Verfahren, d.h. es wird ein Neuronales Netz gelernt, das für jeden möglichen Zustand und jede mögliche Aktion eine Bewertung speichert. Die optimale Aktion kann dann sehr einfach über die sogenannte Greedy-Strategie ermittelt werden. A2C ist dagegen ein Policy-basiertes Verfahren. D.h. es wird direkt ein optimales Verhalten gelernt. A2C hat gegenüber DQN den Nachteil wenig Trainingsdaten zu produzieren. Dieser Effekt wird mit dem A3C-Verfahren ausgeglichen, indem mehrere Agenten parallel trainiert werden, die ihre Erfahrungen (d.h. Neuronale Netze) gemeinsam teilen. Das tabellenbasierte Q-Learning zeigt die schlechtesten Lernergebnisse, was klar auf die fehlende Approximationsfähigkeit zurückzuführen ist.

Zwei wichtige hier noch nicht betrachtete Aspekte sollen in nächster Zeit untersucht werden.

1. Alle hier behandelten Verfahren setzen diskrete Zustands- und Aktionsräume voraus. Viele interessante Aufgabenstellungen in der Robotik haben jedoch kontinuierliche (reellwertige) und hochdimensionale Aktionsräume. DQN und A3C können nicht ohne Weiteres auf kontinuierliche Bereiche angewandt werden, da diese Verfahren darauf beruhen, die Aktion zu wählen, die eine Value-Funktion maximiert. In (LHP+16) wird das vielversprechende Verfahren *Deep Deterministic Policy Gardient* (DDPG) beschrieben, das direkt eine kontinuierliche Policy als Neuronales Netz lernt. Dabei werden Ideen von DQN (Target-Netzwerk und Replay Buffer) und von A2C (Actor- und Crtic-Netzwerk) übernommen und kombiniert.
2. In Robotik-Anwendungen sollen letztendlich reale Roboter und keine Simulationsmodelle trainiert werden. Das Trainieren auf einem realen Roboter ist jedoch extrem zeitaufwendig und wurde bisher nur überzeugend umgesetzt, in dem mehrere Roboter parallel trainiert werden. Das scheidet aber oft aufgrund fehlender Resourcen aus. Ausserdem ist zu bedenken, dass falsche Aktionen die Maschine beschädigen können. Der Weg über ein Simulationsmodell ist daher fast zwangsläufig. Zwei grundsätzlich unterschiedliche Vorgehensweise sind naheliegend. Es wird ein vereinfachtes Simulationsmodell erstellt (wie in unserer Arbeit). Das für das Simulationsmodell trainierte Verhalten wird auf den realen Roboter transferiert und dann getunt. Bei der anderen Vorgehensweise wird

bei manuellem Betrieb des Roboters ein Modell des Roboters gelernt, das dann als Grundlage eines Simulationsmodell genommen werden kann.

9.2 Konsequenzen für die Lehre

RL-Verfahren haben haben eine besonders hohe Relevanz in der Robotik und haben als wichtiges KI-Gebiet eine hohen Motivationsfaktor bei den Studierenden. Einige Abschlussarbeiten konnten bereits im letzten Semester durchgeführt werden: ein Krabbelroboter mit alternativem Simulatiosmodell, Steuerung von Videospielen, Fahrspurverfolgung für ein autonomes Fahrzeug und eine Greifarmsteuerung für unseren Kuka YouBot (Abb. 14). Es bestand eine große Bereitschaft bei den Studierenden, die nicht ganz einfachen Verfahren zu erarbeiten und einzusetzen. Darüberhinaus haben die Erfahrungen gezeigt, dass DP, Q-Learning und DQN wesentlich einfacher zu begreifen sind als die Policy-basierten Verfahren wie A3C und andere DRL-Verfahren wie beispielsweise DDPG.

Abbildung 14: Kuka YouBot mit Mecanum-Rädern und 5 DoF-Greifarm. Sensorik besteht aus einem 270-Grad-Laser und einem RGB-Tiefen-Sensor.

Tabelle 2 zeigt die geplante Integration der hier vorgestellten Verfahren in die Lehre. Besonders in den Projekten und Abschlussarbeiten bietet sich die nährere Untersuchung der im vorherigen Abschnitt erwähnten Ausblicksthemen an.

Studiengang	Lehrveranstaltung	RL-Inhalte
Bachelor Angewandte Infomatik	Vorlesung KI	Einführung in RL, DP, Q und DQN
	Teamprojekt	Einfache DRL-Verfahren, realer Roboter
	Abschlussarbeiten	Einfache DRL-Verfahren, realer Roboter
Master Informatik	Mobile Robotik	Einführung in DRL-Verfahren
	Seminar *Mobile Robotik und Künstliche Intelligenz*	Aktuelle DRL-Verfahren
	Teamprojekt	Aktuelle DRL-Verfahren, realer Roboter
	Abschlussarbeiten	Aktuelle DRL-Verfahren, realer Roboter

Tabelle 2: Geplante Integration der hier vorgestellten Verfahren. Vertikal nimmt der Schwierigkeitsgrad der Themen zu.

Literaturverzeichnis

[ADBB17] Arulkumaran, Kai ; Deisenroth, Marc P. ; Brundage, Miles ; Bharath, Anil A.: A Brief Survey of Deep Reinforcement Learning. In: *IEEE Signal Processing Magazine* (2017), S. 1–16

[Ama17] Amarjyoti, Smruti: Deep reinforcement learning for robotic manipulation-the state of the art. In: *arXiv preprint arXiv:1701.08878* (2017)

[Bel57] Bellman, Richard: *Dynamic Programming*. Dover Publications, 1957

[BNVB12] Bellemare, Marc G. ; Naddaf, Yavar ; Veness, Joel ; Bowling, Michael: The Arcade Learning Environment: An Evaluation Platform for General Agents. In: *CoRR* abs/1207.4708 (2012)

[DCH+16] Duan, Yan ; Chen, Xi ; Houthooft, Rein ; Schulman, John ; Abbeel, Pieter: Benchmarking deep reinforcement learning for continuous control. In: *33rd International Conference on Machine Learning, ICML 2016* 3 (2016), S. 2001–2014

[GHLL17] Gu, Shixiang ; Holly, Ethan ; Lillicrap, Timothy ; Levine, Sergey: Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: *2017 IEEE international conference on robotics and automation (ICRA)* IEEE, 2017, S. 3389–3396

[HLZ+21] He, Rui ; Lv, Haipeng ; Zhang, Sumin ; Zhang, Dong ; Zhang, Hang: Lane following method based on improved DDPG algorithm. In: *Sensors* 21 (2021), Nr. 14

[ITF+21] Ibarz, Julian ; Tan, Jie ; Finn, Chelsea ; Kalakrishnan, Mrinal ; Pastor, Peter ; Levine, Sergey: How to train your robot with deep reinforcement learning: lessons we have learned. In: *International Journal of Robotics Research* 40 (2021), Nr. 4-5, S. 698–721

[KBP13] Kober, Jens ; Bagnell, J A. ; Peters, Jan: Reinforcement Learning in Robotics : A Survey. In: *The International Journal of Robotics Research* (2013)

[LBH15] LeCun, Yann ; Bengio, Yoshua ; Hinton, Geoffrey: Deep learning. In: *Nature* 521 (2015), Nr. 7553, S. 436–444

[LHP+16] Lillicrap, Timothy P. ; Hunt, Jonathan J. ; Pritzel, Alexander ; Heess, Nicolas ; Erez, Tom ; Tassa, Yuval ; Silver, David ; Wierstra, Daan: Continuous control with deep reinforcement learning. In: *International Conference on Learning Representations*, 2016

[MBM+16] Mnih, Volodymyr ; Badia, Adria P. ; Mirza, Lehdi ; Graves, Alex ; Harley, Tim ; Lillicrap, Timothy P. ; Silver, David ; Kavukcuoglu, Koray:

Asynchronous methods for deep reinforcement learning. In: *33rd International Conference on Machine Learning, ICML 2016* 4 (2016), S. 2850–2869

[MKS+15] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg ; Petersen, Stig ; Beattie, Charles ; Sadik, Amir ; Antonoglou, Ioannis ; King, Helen ; Kumaran, Dharshan ; Wierstra, Daan ; Legg, Shane ; Hassabis, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Nr. 7540, S. 529–533

[Mor20] Morales, Miguel: *Grokking Deep Reinforcement Learning*. Manning Publications, 2020

[PyT22] PyTorch: *An open source machine learning framework*. https://pytorch.org/. Version: 2022

[RHW86] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning representations by back-propagating errors. In: *Nature* 323 (1986), Nr. 6088, S. 533–536

[SB18] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018

[SSS+17] Silver, David ; Schrittwieser, Julian ; Simonyan, Karen ; Antonoglou, Ioannis ; Huang, Aja ; Guez, Arthur ; Hubert, Thomas ; Baker, Lucas ; Lai, Matthew ; Bolton, Adrian ; Chen, Yutian ; Lillicrap, Timothy ; Hui, Fan ; Sifre, Laurent ; Van Den Driessche, George ; Graepel, Thore ; Hassabis, Demis: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), Nr. 7676, S. 354–359

[TEF09] Tokic, Michel ; Ertel, Wolfgang ; Fessler, Joachim: The Crawler, A Class Room Demonstrator for Reinforcement Learning. In: *International FLAIRS Conference*, 2009, S. 160–165

[WD92] Watkins, Christopher J. C. H. ; Dayan, Peter: Q-learning. In: *Machine Learning* 8 (1992), Nr. 3, S. 279–292

[WLZ+20] Wang, Hao nan ; Liu, Ning ; Zhang, Yi yun ; Feng, Da wei ; Huang, Feng ; Li, Dong sheng ; Zhang, Yi ming: Deep reinforcement learning: a survey. In: *Frontiers of Information Technology and Electronic Engineering* 21 (2020), Nr. 12, S. 1726–1744