Hierarchical Reinforcement Learning in Robot Control

Malte Viering
Technische Universität Darmstadt
64289 Darmstadt, Germany
cu26mawy at rbg.informatik.tu-darmstadt.de

Abstract

Reinforcement Learning wird häufig für die Robotersteuerung verwendet. Ein Nachteil beim Reinforcement Learning ist, dass es bei hohen Dimensionen wegen des hohen Rechenaufwandes nicht geeignet ist. Ein weiteres Problem ist, dass mit Reinforcement Learning Probleme in der Regel nur auf das konkrete Problem hin optimiert werden aber nicht die daraus resultierenden Vor- und Nachteile weiterer Probleme betrachtet werden. Hierarchical Reinforcement Learning bietet eine mögliche Lösung für diese Probleme. Dieses Paper will einem mit (Hierarchical) Reinforcement Learning nicht vertrauten Leser einen Überblick über Hierarchical Reinforcement Learning und dessen Einsatz für die Robotersteuerung geben.

1 Einleitung

Robotern dynamisches Verhalten beizubringen ist keine triviale Aufgabe. Daher wird vermehrt maschinelles Lernen für die Robotersteuerung verwendet. In der Steuerung von Robotern kommt dabei sehr häufig Reinforcement Learning (RL) zum Einsatz. Dabei kann RL zur Optimierung von bereits gelerntem Verhalten oder zum Lernen von neuem Verhalten verwendet werden. In [1] kommt RL für Roboterfußball zur Anwendung. Apprenticeship Learning, das ähnlich zum RL ist, wird in [2] genutzt, um einem 4-beinigen Roboter das Gehen über schwieriges Terrain bei zu bringen. Damit ein Roboter das dynamische Aufstehen lernt, wird in [3] RL in Anspruch genommen.
RL leidet zum einen unter dem sogenannten Problem „curse of dimensionality“. Das meint, dass der Zustandsraum in Problemen mit höheren Dimensionen wie z. B. der Steuerung eines humanoiden Roboters zu groß ist, um das Problem in einem überschaubaren Zeitaufwand zu lösen. Ein weiteres Problem von RL ist, dass ein Problem immer nur lokal optimiert wird. Das kann zur Folge haben, dass z. B. zwei nacheinander durchgeführte Bewegungen nach der Optimierung immer noch zu teuer sind. Da die optimale Bewegung, wenn nur die erste Bewegung isoliert betrachtet wird, in einem Zustand enden kann, der für die zweite Bewegung sehr ungünstig ist.
Zur Verbesserung des RL wurde daher von einigen Wissenschaftlern vorgeschlagen, Hierarchical Reinforcement Learning (HRL) zu verwenden. In diesem Paper werden zunächst grundlegende Konzepte des RL vorgestellt. Auf dieser Grundlagen aufbauend wird dem Leser dann HRL theoretisch und anhand von angewandten Roboterbeispielen die praktische Umsetzungsmöglichkeit veranschaulicht.
Dafür gliedert sich der Rest des Papers wie folgt: In Kapitel 2 wird RL dargestellt. Im Besonderen wird der Markov Decision Process (MDP) erläutert. Für das Lernen mit MDP wird die Value Function Methode allgemein und konkret anhand einiger vorhandener Algorithmen erklärt. Des Weiteren werden Dynamic Movement Primitives (DMP) vorgestellt. Auch das RL mit DMP wird kurz aufgezeigt. In Kapitel 3 wird das HRL theoretisch beschrieben. Dafür werden zunächst mögliche Anpassungen des MDP oder der DMP dargestellt, so dass diese für HRL verwendet werden können. Daran anschließend wird darauf eingegangen, wie über diese Anpassungen gelernt werden kann. In Kapitel 4 werden einige Forschungsergebnisse vorgestellt, die HRL in der Robotersteuerung verwenden. Das Paper wird mit einer kurzen Zusammenfassung beendet.

2 Reinforcement Learning

In diesem Kapitel wird näher auf das Reinforcement Learning (RL) eingegangen. Dafür wird der (Semi-) Markov Decision Process (SMDP/MDP) beschrieben. Über dem (S)MDP sind viele RL-Algorithmen definiert. Für MDP wird RL anhand der Value-Funktion-Methode erläutert. Danach erfolgt eine Darstellung der Dynamic Movement Primitives, für die ein möglicher RL-Ansatz aufgezeigt wird.

2.1 (Semi-) Markov Decision Process

Viele RL-Algorithmen basieren auf dem Markov Decision Process [4] oder Abwandlungen dieses Modells. Der MDP besteht aus dem Tupel$(S, A, \{P_{\mathrm{sa}}\}, \gamma, R)$. Die einzelnen Elemente sind dabei wie folgt definiert:

- S ist die endliche Menge von Zuständen des MDPs.
- A ist die endliche Menge von Aktionen, dabei sind A_s alle Aktionen, die im Zustand s ausgeführt werden können.
- P ist die Übergangswahrscheinlichkeit; dabei gibt $P(s'|s, a)$ die Wahrscheinlichkeit an, vom Zustand s bei Aktion a in den Zustand s' zu wechseln.
- γ ist der Discount Factor. Der Discount Factor ist definiert in $[0, 1)$. Er beschreibt, wie stark eine zukünftig stattfindende Belohnung für den aktuellen Zustand relevant ist. Ein Discount Factor nahe bei 1 gibt an, dass zukünftige Belohnungen stark gewichtet werden.
- R ist die Belohnungsfunktion. $R : S \times A \to \Re$ ist die sofortige Belohnung bei der Aktion a im Zustand s.

Ein MDP funktioniert dabei wie folgt: Es wird in einem Zustand s_0 gestartet, in diesem Zustand wird eine Aktion a_0 ausgeführt. Als Folge der Aktion a_0 wird zufällig in einen Zustand s_1 gewechselt, s_1 ergibt sich dabei aus $\mathrm{P}_{\mathrm{s_0 a_0}}$. Dann wird eine weitere Aktion a_1 ausgeführt. Für das Ausführen der Aktion a im Zustand s erfolgt die sofortige Belohnung $R(s, a)$. Die Langzeitbelohnung der Aktionsausführungen $\{a_0$,a_1,a_2,...,$a_n\}$ vom Zustand s_0, gestartet im MDP, ist definiert als:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + ...,$$

dabei werden Belohnungen, die später erhalten werden, über den Discount Factor verringert. Ziel des RLs ist die erwartete Langzeitbelohnung

$$E[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + ...]$$

zu maximieren. Neben der hier verwendeten Notation $R : S \times A \to \Re$ gibt es noch die Notation $R : S \to \Re$. Bei $R : S \to \Re$ gibt es eine sofortige Belohnung im Zustand s unabhängig von der Aktion a, die ausgeführt wird, auch geschrieben als r_s oder einfach r, wenn s aus dem Kontext ersichtlich ist. $R : S \times A \to \Re$ ist bereits eine Vereinfachung von $R : S \times A \times S \to \Re$, in welcher die Belohnung vom Zustand s, der Aktion a und dem Zustand s', in den gewechselt wird, abhängig ist. Der Einfachheit halber ist es aber gebräuchlich, $R : S \times A \to \Re$ zu verwenden [5].
Wenn im Zustand s die Aktion a ausgeführt wird, gibt die Zufallsfunktion $P(s'|s, a)$ für alle s' an, wie wahrscheinlich in den Zustand s' gewechselt wird.
Die Policy-Funktion $\pi : S \to A$ bestimmt, welche Aktion a_i in einem Zustand s_i ausgeführt wird. Die Policy ist die Steuerungsfunktion, die für einen Zustand die beste Aktion ausführen soll. Wenn RL in der Robotik verwendet wird, ist die Policy π häufig die Steuerungsfunktion des Roboters.
Die Value-Funktion $V : S \to \Re$, die Langzeitbelohnung für einen Zustand, zu einer Policy π ist definiert als:

$$V^\pi(s) = E[R(s_0, \pi(s_0)) + \gamma R(s_1, \pi(s_2)) + \gamma^2 R(s_2, \pi(s_2)) + ...|s_0 = s].$$

Die Value-Funktion zu einer Policy weist jedem Zustand einen Wert zu, der der erwarteten Langzeitbelohnung beim Ausführen der Policy π entspricht.
Neben der Value-Funktion wird noch häufig die Action-Value-Funktion $Q : S \times A \to \Re$ verwendet. Diese ist für eine Policy π, wie folgt, definiert:

$$Q^\pi(s, a) = E[R(s_0, a_0) + \gamma R(s_1, \pi(s_2)) + \gamma^2 R(s_2, \pi(s_2)) + ...|s_0 = s, a_0 = a].$$

Dabei gibt die Action-Value-Funktion ähnlich wie die Value-Funktion die erwartete Langzeitbelohnung beim Ausführen einer Policy π an. Der Unterschied ist, dass die Action-Value-Funktion für einen Startwert, der aus einem Zustand und einer Aktion besteht, die Belohnung angibt. Das Hinzufügen einer Aktion zu dem Zustand ist für einige RL-Algorithmen notwendig.
Die Value-Funktion kann in die Bellman Gleichung umgeformt werden. Die Bellman Gleichung kann dazu verwendet werden, eine Lösung über dynamisches Programmieren zu finden. Die Bellman Form der Value-Funktion zu einer Policy π lautet:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \left[\sum_{s' \in S} P(s'|s, \pi(a)) V^{\pi}(s') \right].$$

Das Ziel des RLs ist es, die bestmögliche Policy π zu finden. Die optimale Value-Funktion, die den bestmöglichen Wert für die Value-Funktion angibt, kann definiert werden als:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s).$$

Für die iterative Berechnung der optimalen Value-Funktion kann diese auch in die Bellman Gleichung gebracht werden:

$$V^{*}(s) = \max_{a} \left(R(s, a) + \gamma \left[\sum_{s' \in S} P(s'|s, a) V^{*}(s') \right] \right).$$

Die Bellman Gleichung der Action-Value-Funktion lautet:

$$Q^{*}(s, a) = R(s, a) + \gamma \left[\sum_{s' \in S} P(s'|s, a) \max_{a' \in A_{s'}} Q^{*}(s', a') \right].$$

Aus der Bellman Gleichung für V und Q ergibt sich direkt eine Vorschrift für die iterative Bestimmung von V^{*} und Q^{*}. Die Vorschrift für die Iterative Berechnung von V^{*} ist:

$$V(s) \leftarrow \max_{a} \left(R(s, a) + \gamma \left[\sum_{s' \in S} P(s'|s, a) V^{*}(s') \right] \right).$$

Diese Formel wird iterativ für alle $s \in S$ angewendet, dabei konvergiert $V(s)$ zu $V^{*}(s)$. Dasselbe kann in ähnlicher Weise für $Q(s, a)$ durchgeführt werden. Die iterative Berechnungsformel lautet:

$$Q(s, a) \leftarrow R(s, a) + \gamma \left[\sum_{s' \in S} P(s'|s, a) \max_{a' \in A_{s'}} Q^{*}(s', a') \right].$$

Neben dem MDP spielt der Semi-Markov Decision Process (SMDP) eine wichtige Rolle beim (H)RL. Im SMDP wird der MDP in der Art generalisiert, dass es, wenn eine Aktion a im Zustand s ausgeführt wird, τ Zeit dauert, bevor in den nächsten Zustand s' gewechselt wird. Dabei ist τ eine Zufallsvariable. τ kann entweder ganzzahlige Werte oder reelle Werte annehmen, abhängig davon wird der SMDP als diskret oder kontinuierlich bezeichnet. Beim SMDP wird die Übergangsfunktion in der Art erweitert, dass $P(s', \tau|s, a)$ die Wahrscheinlichkeit angibt, beim Ausführen der Aktion a im Zustand s in den Zustand s' nach τ Zeit gewechselt wird. Exemplarisch wird hier die Value-Funktion zu einer Policy π in einem SMDP dargestellt. Die Value-Funktion ist definiert als:

$$V^{\pi}(s) = R(s, \pi(s)) + \left[\sum_{s' \in S, \tau} \gamma^{\tau} P(s', \tau|s, \pi(a)) V^{\pi}(s') \right].$$

Der Unterschied der Value-Funktion für SMDPs gegenüber MDPs ist die modifizierte Übergangswahrscheinlichkeit und dass der Discount-Faktor abhängig ist von der Zeit, die bis zum Wechsel in den nächsten Zustand s' vergeht.

2.1.1 Value Function Methoden

Wie bereits erwähnt, kann über die Bellman Gleichung V^{*} und Q^{*} iterativ berechnet werden. Wenn zu einem (S)MDP die Value- oder Action-Value-Funktion vorliegt, kann daraus die Policy berechnet werden. Es ist auch möglich direkt die Policy zu lernen, ohne den Umweg über die Value-Funktion zu nehmen. Das im späteren Verlauf vorgestellte Path Integral Verfahren ist z. B. ein

Policy-Lernverfahren. Neben der direkten Berechnung über die Bellman Gleichung gibt es effizientere Verfahren, um (S)MDPs zu lösen. Ein bekanntes Verfahren ist Q-Learning. Im Folgenden wird Q-Learning kurz vorgestellt.
Q-Learning ähnelt der bereits vorgestellten Updateformel. Es verwendet aber die direkte Belohnung und die Action-Value-Funktion des nachfolgenden Zustandes. Die Q-Learning-Gleichung ist:

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k \left[R(s,a) + \gamma \max_{a' \in A_{s'}} Q_k(s',a') \right].$$

Der Vorteil von Q-Learning ist, dass die Übergangswahrscheinlichkeit P_{sa} nicht benötigt wird. Dies ermöglicht das Lernen von Beispielen, auch wenn kein vollständiges Modell vorhanden ist. Q-Learning kann auch bei SMDP verwendet werden (weitere Informationen siehe [4]).
Diese Art des Lernens wird als Temporal Difference (TD) Lernen bezeichnet.

Q(λ) Beim Q(λ) [6] wird Q-Learning um den Eligibility Trace erweitert. Lernen erfolgt in zwei Phasen: In der ersten Phase werden alle Zustände S aktualisiert, in der zweiten Phase wird nur der aktuell betrachtete Zustand s aktualisiert. Dafür wird in der ersten Phase der TD-Fehler

$$\delta(s) = r + \gamma V(s') - V(s),$$

des Wechsels von s in s' berechnet. Dabei ist

$$V(s) = \max_{a \in A_s} Q(s,a)$$

der Action-Value-Funktionswert der besten Aktion a in s. Der Eligibility Trace wird für die Zustände wie folgt verringert:

$$e(S,A) \leftarrow \gamma\lambda e(S,A).$$

Dann wird die Action-Value-Funktion inkrementell über

$$Q(S,A) \leftarrow Q(S,A) + \alpha_Q e(S,A)\delta(s)$$

aktualisiert. Bei diesem Update wird neben dem Lernparameter α_Q der Eligibility Trace verwendet. Das sorgt dafür, dass Q für Paare (s,a), die in der nahen Vergangenheit besucht wurden, stärker aktualisiert wird als für Paare (s',a'), die länger nicht besucht wurden.
In der zweiten Phase wird der Wert des aktuell besuchten Zustandes (s,a) aktualisiert. Dafür wird zunächst auch hier der TD-Fehler

$$\delta'(s) = r + \gamma V(s') - Q(s,a)$$

berechnet. Dann erfolgt ein Update der Action-Value-Funktion

$$Q(s,a) \leftarrow Q(s,a) + \alpha_Q \delta'(s).$$

Als letztes wird der Eligibility Trace aktualisiert:

$$e(s,a) \leftarrow e(s,a) + 1.$$

Dabei wird der Eligibility Trace des Paares (s,a) um 1 inkrementiert und für alle anderen Paare nicht verändert.

Actor-Critic-Methode Die Actor-Critic-Methode [7] ist eine TD-Methode des RL, bei der die Value-Funktion und die Policy getrennt von einander betrachtet werden. Die Policy wird dabei als Actor und die Value-Funktion als Critic bezeichnet. Das Lernen erfolgt immer basierend auf der Policy. In der Robotik übernimmt der Actor die Steuerung des Roboters und wählt die Aktionen abhängig von der im Critic gelernten Value-Funktion.

2.2 Dynamic Movement Primitives

Eine Alternative zu MDP sind Dynamic Movement Primitives (DMP), die von Ijspeert et Al [8] vorgeschlagen wurden. Die Darstellung in diesem Kapitel orientiert sich an der Beschreibung von DMP aus [9]. In diesem Verfahren werden als Modell eine Menge von dynamischen Systemgleichungen verwendet. Die Beschreibung des Modells erfolgt über ein System von Differentialgleichungen. Die Differentialgleichungen werden über eine nicht lineare Störfunktion erweitert. Die Störfunktion

fließt über einen Parameter (genannt Shape-Parameter $\boldsymbol{\theta}$) in die Berechnung mit ein. Eine vereinfachte DMP kann durch folgende Gleichung dargestellt werden:

$$\frac{1}{\tau}\ddot{x}_t = \alpha\left(\beta\left(g - x_t\right) - \dot{x}_t\right) + \mathbf{g}_t^T\boldsymbol{\theta}. \tag{1}$$

Mit einer DMP wird dabei eine Bewegung von einer Startposition x_0 zu einer Endposition x_1 (g in (1)) beschrieben. Die Differenzialgleichung (in (1)) einer DMP besteht aus einer linearen Differentialgleichung (linker Teil von (1)), die durch eine Störfunktion $\mathbf{g}_t^T$ gestört wird [9]. Die Störfunktion sorgt für eine ruhige und beliebige Form der Bewegung. In diesem Paper werden DMPs nicht im Detail erklärt, für mehr Informationen siehe [8]. In diesem Zusammenhang ist von Bedeutung, dass eine DMP, wenn sie über die Zeit integriert wird, von einem Startpunkt x_0 zum Zielpunkt g konvergiert, sowie die Verwendung der erzeugten Trajektorie von $x, \dot{x}, \ddot{x}$ zur Steuerung des Roboters verwendet wird [9]. In vielen Roboterproblemen werden mehrere Bewegungen nacheinander ausgeführt. Diese werden bei DMPs modelliert, indem von Postion x_1 (Ende der ersten DMP) mit einer anderen DMP eine Bewegung zu einer Postion x_2 angegeben wird.
RL wird bei DMPs verwendet, um die Kosten der Bewegung zu reduzieren, dafür wird der Shape-Parameter beim Lernen optimiert.
Mit der Eigenschaft, dass DMPs eine Aktion von einem Startpunkt x_0 zu einem Ziel x_1 beschreiben, welche dabei über mehrere Zeitschritte gehen, haben DMPs Ähnlichkeit mit Optionen, die im Unterkapitel 3.1.2 beschrieben werden.

2.2.1 Path Integrals Algorithmus

Der Path Integrals Algorithmus (PI^2) [10] ist ein möglicher RL-Algorithmus, der das Optimieren von DMPs erlaubt. Alternativen sind z. B. [11] ein Policy-Gradient basiertes Lernverfahren oder PoWER [12] ein von Expectation-Maximization inspiriertes Lernverfahren. In PI^2 wird der Shape-Parameter $\boldsymbol{\theta}$ gegen eine vom Benutzer definierte Kostenfunktion

$$J(\tau_i) = \phi_{tN} + \int_{t_i}^{t_N} \left(r_t + \frac{1}{2}\boldsymbol{\theta}_t^T\mathbf{R}\boldsymbol{\theta}_t\right) dt$$

optimiert [9]. $J(\tau)$ gibt dabei die Kosten über eine Trajektorie τ an, die zum Zeitpunkt t_i startet und bis t_N geht. ϕ_{tN} bezeichnet die terminalen Kosten der Trajektorie τ_i. r_t benennt die direkten Kosten zum Zeitpunkt t. $\frac{1}{2}\boldsymbol{\theta}_t^T\mathbf{R}\boldsymbol{\theta}_t$ gibt die Steuerungskosten zum Zeitpunkt t an.
Das Lernen erfolgt, indem die DMP k mal mit leicht modifizierten Shape-Parametern $\boldsymbol{\theta} + \epsilon_{t,k}^{\boldsymbol{\theta}}$ ausgeführt wird. Dabei ist der Fehlerparameter $\epsilon_{t,k}^{\boldsymbol{\theta}}$ ein Zufallswert, der aus der Gauß-Verteilung gewählt wird. Die so gestörte DMP lässt sich schreiben als:

$$\frac{1}{\tau}\ddot{x}_t = \alpha\left(\beta\left(g - x_t\right) - \dot{x}_t\right) + \mathbf{g}_t^T\Big(\underbrace{\boldsymbol{\theta} + \boldsymbol{\epsilon}_{t,k}^{\boldsymbol{\theta}}}_{Shape\ Erforschung}\Big).$$

Über die Fehlerparameter werden leicht unterschiedliche Bewegungen erzeugt. Nach k Bewegungen mit unterschiedlichen Fehlerparametern wird die Policy (Shape-Parameter) aktualisiert. Das Ausführen einer DMP mit modifiziertem Shape-Parameter und anschließender Berechnung der Kostenfunktion wird auch als ein Roll-out bezeichnet.
Die Aktualisierung des Shape-Parameters in PI^2 wird im Folgenden erläutert und basiert auf der Beschreibung von [9]. $S(\tau_{i,k})$ beschreibt die Kosten zum Zeitpunk i in Roll-out k.

$$P(\tau_{i,k}) = \frac{\exp(-\frac{1}{\lambda}S(\tau_{i,k}))}{\sum_{l=1}^{K}\exp(-\frac{1}{\lambda}S(\tau_{i,l}))}$$

berechnet die Wahrscheinlichkeit des Roll-outs k / des Störparameters $\boldsymbol{\epsilon}_k$ des Roll-outs k. Dabei werden Roll-outs mit niedrigen Kosten eine hohe Wahrscheinlichkeit und Roll-outs mit hohen Kosten eine geringe Wahrscheinlichkeit zugewiesen.

$$\delta\boldsymbol{\theta}_{t_i} = \sum_{k=1}^{K}\left[P(\tau_{i,k})\mathbf{M}_{t_i,k}\boldsymbol{\epsilon}_{t_i,k}^{\boldsymbol{\theta}}\right]$$

berechnet den gewichteten Mittelwert über K-Roll-outs zum Zeitpunk t_i. Die Matrix $\mathbf{M}$ wird benötigt, um $\boldsymbol{\epsilon}$ in den Parameterraum zu transformieren.

$$[\delta\boldsymbol{\theta}]_j = \frac{\sum_{i=0}^{N-1}(N-i)w_{j,t_i}[\delta\boldsymbol{\theta}_{t_i}]_j}{\sum_{i=0}^{N-1}(N-i)w_{j,t_i}}$$

berechnet das Update eines Elements des Shape-Parameters. Dafür werden alle $[\delta\boldsymbol{\theta}_{t_i}]_j$ aufsummiert. Die Summierung wird evaluiert, indem Updates am Anfang stärker gewichtet werden als Updates im späteren Verlauf der Trajektorie. Es wird so vorgegangen, weil Änderungen am Anfang den gesamten weiteren Verlauf beeinflussen. Ebenso werden die $[\delta\boldsymbol{\theta}_{t_i}]_j$ nach der Aktivität w der zum Parameter j gehörenden Basisfunktion gewichtet [9].

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \delta\boldsymbol{\theta}$$

berechnet den neuen Shape-Parameterwert.

3 Hierarchical Reinforcement Learning

Es gibt viele Ansätze zum Hierarchical Reinforcement Learning (HRL), die sich teilweise in der Anwendung und Zielsetzung unterscheiden. Im Folgenden werden daher einige theoretische Ansätze aufgezeigt. Eine weitere Übersicht zu einigen Ansätzen im HRL ist in [4] zu finden.

3.1 Modelle für das HRL

In diesem Unterkapitel werden mehrere Modelle vorgestellt, die beim HRL verwendet werden können. Diese Modelle sind: 1. Feudal Reinforcement Learning, ein früher Ansatz zu HRL, der die allgemeine Idee des HRL gut beschreibt; 2. Options, die eine MDP um zeitlich verlängerte Aktionen erweitert; 3. die Hierarchical Abstract Machines (HAM), die einen MDP um hierarchische Policies erweitert; 4. MAXQ Value-Funktion Zerteilung, die eine (S)MDP in eine Hierarchie von SMDPs zerteilt; sowie 5. hintereinander ausgeführte DMPs. Dabei haben Options, HAMs und die MAXQ-Zerteilung gemeinsam, dass diese auf der Theorie von SMDPs arbeiten.

3.1.1 Feudal Reinforcement Learning

In [13] wird vorgeschlagen, eine Hierarchie von Managern zu verwenden. In dieser Hierarchie hat jeder Manager einen direkten Untermanager. Der Manager legt für seine Untermanager Zwischenziele fest. Die Untermanager können selber wieder Untermanager haben. Die Untermanager erhalten eine Belohnung, wenn sie das Zwischenziel erfüllen. Dabei ist es unbedeutend, ob der Manager sein Hauptziel mit Erlangen des Zwischenziels erreicht. Die Belohnung des Managers ist wiederum nur vom Erreichen seines Ziels abhängig. In [3] wird gezeigt, dass bei komplexeren Aufgaben die Lerngeschwindigkeit erhöht werden kann, wenn der Manager auch in dem Fall eine Belohnung erhält, wenn Untermanager die erhaltenen Zwischenaufgaben erfüllen. Ein Vorteil des hierarchischen Managerkonzeptes ist es, dass alle Manager das System nur in der Granularität ihrer Aktionen kennen müssen. In [13] wird Feudal Reinforcement Learning erfolgreich auf eine Labyrinthaufgabe angewendet, dabei konnte die Lerngeschwindigkeit verglichen mit einem RL-Ansatz gesteigert werden.

3.1.2 Options

In [14] wird ein Verfahren beschrieben, dass es erlaubt, einen MDP M = $(S, A, \{P_{\text{sa}}\}, \gamma, R)$ um zeitlich verlängerte Aktionen zu erweitern. Die zeitlich verlängerte Aktion wird Option genannt. Eine Option besteht aus dem Tupel(I, π, β), dabei ist

- I die Menge der Zustände, in der die Option gestartet werden kann. Es gilt, $I \subseteq S$, S ist die Menge der Zustände des MDPs M.
- π die Policy der Option.
- β die Terminierungsfunktion der Option. $\beta : S \to [0, 1]$ definiert für jeden Zustand im MDP M, wie wahrscheinlich die Option in diesem Zustand terminiert. Bei Markov Optionen wird angenommen, dass für alle s mit $\beta(s) < 1$ das $s \in I$ gilt.

Als Beispiel für eine Option ist ein MDP in der Abbildung 1 dargestellt. In diesem MDP können Optionen sein: „verlasse den Raum durch eine der beiden Türen“. Für die in der Grafik gezeigte Option wäre I: alle Zustände im Raum unten links. Der Funktionswert von β ist definiert als 0, für alle Zustände im Raum unten links und als 1, für alle anderen Zustände. In diesem Fall ist β eine deterministische Funktion. Die Policy π würde dann Aktionen ausführen, um den Raum nach oben zu verlassen. Die Idee des Beispiels ist von [14] übernommen.

Aktionen im MDP M können als Ein-Schritt-Optionen dargestellt werden [14], die genau das Verhalten der Aktion haben. Dies ermöglicht, die Aktionen des MDPs M weiter zu verwenden.
Ebenfalls werden in [14] Semi-Markov Optionen vorgestellt. Bei einer (Markov) Option hängt der nächste Zustand nur vom aktuellen Zustand ab. Wohingegen bei einer Semi-Markov Option der nächste Zustand von allen Zuständen abhängt, die ausgeführt wurden, seitdem die Option gestartet wurde. Diese Sequenz der bisher ausgeführten Zuständen in der Option wird in [14] mit History Ω bezeichnet. Eine Policy $\pi : \Omega \rightarrow A$ bei Semi-Markov Optionen arbeitet dann über die History anstatt nur über den aktuellen Zustand. Die Terminierungsfunktion $\beta : \Omega \rightarrow [0, 1]$ ist abhängig von der History.

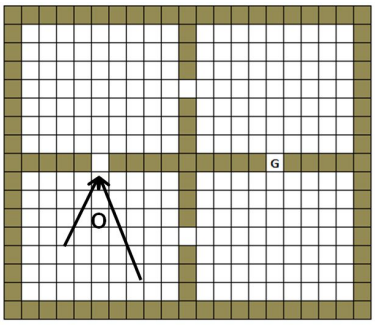

Abbildung 1: Beispiel einer möglichen Option in einem MDP. Die Idee des Beispieles ist aus [14] übernommen.

Damit Optionen im MDP M verwendet werden können, hat [14] Policies über Optionen O ($\mu : S \rightarrow O$) eingeführt. Wenn eine Policy über eine Option zu einem Zeitpunk t gestartet wird, führt diese Aktionen aus, bis die Option terminiert. Dabei kann eine Option, die von der Policy μ gestartet wurde, wiederum weitere Optionen aufrufen. Policies über Optionen sind in der Regel nicht Markov, auch wenn nur Markov Optionen verwendet werden [14].

In [14] wird das Theorem aufgestellt, dass ein MDP mit einem beliebigen Set an Optionen ein SMDP ist, wenn nur die Optionen ausgeführt werden. Dafür müssen die Optionen als unsichtbare Einheiten verwendet werden.

Damit Optionen als unsichtbare Einheiten betrachtet werden können, führt [14] das multi-time Modell ein. In diesem wird die Belohnung beim Ausführen einer Option o im Zustand s als

$$r_s^o = E\left\{r_{t+1} + \gamma r_{t+1} + ... + \gamma^{k-1} r_{t+k} | E(o, s, t)\right\}$$

definiert. Hier gibt $E(o, s, t)$ an, dass die Option o im Zustand s zum Zeitpunkt t initiiert wurde. Die Option läuft für k Zeiteinheiten.
Zusätzlich wird in [14] die Wahrscheinlichkeit, dass in den Zustand s' gewechselt wird, wenn sich der MDP in einem Zustand s befindet und die Option o ausgeführt wird, als

$$p_{ss'} = \sum_{k=1}^{\infty} p(s', k)\gamma^k$$

berechnet. Die Funktion $p(s', k)$ gibt die Wahrscheinlichkeit an, dass zum Zeitpunkt k in den Zustand s' gewechselt wird.
SMDP Algorithmen können mit Optionen nur dann verwendet werden, wenn diese als unsichtbare Einheit betrachtet werden [14]. [14] führt weiterhin Optionen ein, die „unterbrechbar“ sind. Auf diese wird in diesem Paper nicht näher eingegangen.

3.1.3 Hierarchical Abstract Machines

In [15] werden Hierarchical Abstract Machines (HAM) vorgestellt. HAM ist ein Ansatz, die Policies für einen MDP hierarchisch anzuordnen. Mit diesem Verfahren kann die Anzahl an möglichen Policies für einen MDP reduziert werden. Dafür werden in HAM abstrakte Maschinen eingeführt. Eine abstrakte Maschine stellt eine unvollständige Policy, die über einen endlichen Automaten realisiert ist, für einen MDP dar. Diese abstrakten Maschinen werden hierarchisch angeordnet. Das

Ausführen einer Aktion im Ursprung MDP hängt bei HAM vom aktuellen Zustand des MDPs und des aktuellen Zustandes der HAM ab. HAMs können die Anzahl an möglichen Policies für einen MDP einschränken und so das Lernen beschleunigen.
Eine Maschine für eine HAM ist definiert als eine Menge von Zuständen, eine Zustandsübergangsfunktion und eine Funktion, die den Startzustand einer Maschine bestimmt [15]. Die Maschinenzustände können von den folgenden 4 Typen sein. 1. Action-Zustand: es wird eine Aktion im MDP ausgeführt, 2. Call-Zustand: es wird eine andere Maschine aufgerufen, 3. Choice-Zustand: ein anderer Maschinenzustand wird nicht deterministisch ausgewählt und 4. Stop-Zustand: die aktuelle Maschine wird gestoppt und die Kontrolle an die nächst höhere Maschine weitergegeben [15]. Die Zustandsübergangsfunktion bestimmt den nächsten Zustand einer Maschine nach einem Call-Zustand oder nach dem Ausführen einer Aktion in einem Action-Zustand. Der Übergang ist dabei abhängig von den Maschinen und vom MDP Zustand; und er ist stochastisch [15].
Eine Policy für ein MDP, die konsistent zu einem für die MDP definierte HAM ist, trifft Entscheidungen, wenn sich die HAM in einem Choice-Zustand befindet [15]. Das restliche Verhalten ist über die HAM definiert.
H bezeichnet die Startmaschine der HAM und alle von der Startmaschine erreichbaren Maschinen. Die Anwendung der HAM H auf eine MDP M, $HM = H \circ M$ ist etwas vereinfacht definiert als:
Die Zustände von HM sind das Kreuzprodukt der Zustände in H und M. Für jeden Zustand in HM, in dem der Zustand der HAM H ein Action-Zustand ist, werden die Zustandsübergangsfunktionen von M und H zusammengelegt. In jedem Zustand von HM, in dem sich H in einem Choice-Zustand befindet, wird eine Aktion eingeführt, die nur die Zustandskomponente von H ändert. Die Belohnung in HM wird von den primitiven Aktionen in M genommen [15].

In [15] wird folgendes Theorem angegeben. Wenn M ein beliebiger MDP und H ein für M definierter HAM mit C der Menge der Choice-Zustände ist, dann existiert für M und H ein Entscheidungsprozess $reduce(H \circ M)$ mit den Zuständen C. Für diesen Entscheidungsprozess gilt, dass die optimale Policy für $reduce(H \circ M)$ zur optimalen Policy in M, die konsistent zur HAM H ist, korrespondiert.

Über dieses Theorem kann die Menge der möglichen Policies für ein MDP verringert werden. Beim Lernen mit HAM kann direkt im reduzierten Zustandsraum $reduce(H \circ M)$ gelernt werden [15].

3.1.4 MAXQ Value Function Decomposition

In [5] wird MAXQ Value Function Decomposition (einfach MAXQ), eine Zerteilung eines MDPs M, definiert. Der MDP M wird in eine endliche Menge von Zwischenaufgaben $\{M_0, M_1, ..., M_n\}$ aufgeteilt. Jede der Zwischenaufgaben stellt eine SMDP für ein Teilproblem dar. Die Zwischenaufgaben werden in einem Aufgabengraphen mit M_0 als Wurzelknoten aufgeteilt. Wenn M_0 gelöst wird, ist M gelöst. Eine Zwischenaufgabe M_i ist definiert über das Tuple$(T_i, A_i, \tilde{R}_i)$ [5]. Dabei ist

- T_i die deterministische Terminierungsfunktion der Zwischenaufgabe M_i. T_i legt für alle Zustände S von M fest, ob diese aktive Zustände S_i oder Terminierungszustände T_i sind.
- A_i die Menge der Aktionen der Zwischenaufgabe M_i. Dabei können diese primitiven Aktionen im MDP M, mehrere Aktionen im MDP M und der Aufruf einer weiteren Zwischenaufgabe sein. Kreise beim Aufrufen von Zwischenaufgaben sind nicht erlaubt.
- $\tilde{R}_i(s')$ die Pseudo-Belohnungsfunktion der Zwischenaufgabe M_i. Die Pseudo-Belohnungsfunktion legt eine Belohnung für das Erreichen eines Terminierungszustandes s' fest. Für Nicht-Terminierungszustände ist die Belohnung 0. Die Pseudo-Belohnungsfunktion wird nur während des Lernens verwendet.

[5]

Über den in Zwischenaufgaben aufgeteilten MDP wird eine hierarchische Policy $\pi = \{\pi_1, \pi_2, ..., \pi_n\}$ gelernt. Dabei gibt der Index an, für welche Zwischenaufgabe die Policy definiert ist. Eine Policy definiert für einen Zustand die primitive Aktion oder die Zwischenroutine, die aufgerufen wird [5].
Eine Besonderheit bei MAXQ ist, dass Zwischenaufgaben über einen Stack verfügen. Beim Aufrufen einer Zwischenaufgabe können diese Informationen übergeben werden. Der Zustand einer Zwischenaufgabe M_i ist daher definiert als (s, K) mit K Stackwert und Zustand s in M_i. Daraus resultierend können Zustände in einer Zwischenaufgabe nicht nur von allen vorherigen Zuständen der Zwischenaufgabe abhängig sein, sondern auch von den Zuständen, die vor der Zwischenaufgabe

ausgeführt wurden. Daher sind die Policies der Zwischenaufgabe nicht im Sinne des Zustandsraums des MDPs M Markov. Die Policies der Zwischenaufgaben sind aber Markov, wenn man den erweiterten Zustandsraum (Stack Parameter) betrachtet [4].

3.1.5 HRL mit DMPs

Mehrere DMPs können nacheinander ausgeführt werden. Die Endposition einer DMP beeinflusst die Kosten der nachfolgenden DMP. HRL kann in diesem Szenario dazu verwendet werden, zum einen die Endpostion der DMPs und zum andern für jede DMP den Shape-Parameter zu lernen und somit die summierten Kosten von mehreren hintereinander ausgeführten DMPs zu minimieren. Wenn zwei DMPs hintereinander ausgeführt werden, ist die Endposition der ersten DMP_1 die Startposition der zweiten DMP_2. Durch das Lernen einer neuen Endpositionierung der ersten DMP_1 ist es möglich, die summierten Kosten von DMP_1 und DMP_2 stärker zu minimieren, als wenn nur DMP_1 und DMP_2 mit RL optimiert werden.
Als Vorbereitung, um HRL mit DMPs nutzen zu können, wird der Zielparameter in der DMP Gleichung (1) um einen Störwert ϵ_k^g erweitert:

$$\frac{1}{\tau}\ddot{x}_t = \alpha(\beta((\underbrace{g + \epsilon_k^g}_{Ziel\ Erforschung}) - x_t) - \dot{x}_t) + \mathbf{g}_t^T(\underbrace{\boldsymbol{\theta} + \boldsymbol{\epsilon}_{t,k}^{\boldsymbol{\theta}}}_{Shape\ Erforschung}).$$

Der Zielparameter kann dabei so ähnlich wie der Shape-Parameter mit einer RL-Vorschrift optimiert werden.

DMP und Optionen Das Konzept der DMP hat Ähnlichkeiten zu Options. Eine DMP kann als eine zeitlich verlängerte, nicht unterbrechbare Aktion gesehen werden. Wenn DMPs im Kontext eines Option-Frameworks betrachtet werden, können die Elemente das Optionen-Tupel(I, π, β) für eine DMP von x_0 nach x_1, wie folgt definiert werden. Als Menge I der Startpunkte wird x_0 gewählt. Wenn die DMP nicht nur aus x_0 sondern auch aus mehreren Punkten gestartet werden kann, werden alle diese Punkte in die Menge I mit aufgenommen. Für die Terminierungsfunktion wird der Suchraum des Zielparameters x_1 als Terminierungszustände beschrieben. In [16] werden DMPs mit festem Zielpunkt in einem Option-Framework verwendet und darauf ein hierarchisches relatives Entropy Policy-Suchverfahren vorgestellt.

3.2 Lernen bei HRL

In diesem Unterkapitel wird auf das Lernen der im letzten Unterkapitel vorgestellten Modelle eingegangen.

3.2.1 Lernen bei HRL Modellen auf Basis von SMDP

Der Vorteil von HRL auf Basis von SMDPs ist, dass für SMDPs definierte RL-Algorithmen anwendbar sind. Im Folgenden wird kurz erläutert, wie in den vorgestellten Modellen das Lernen funktionieren kann.
Im Option-Framework kann z. B. Q-learning für SMDPs direkt angewandt werden, wenn die Optionen als eine zeitlich verlängerte, unzerbrechliche Aktion betrachtet werden. In [14] wird Q-learning für SMDPs zum Lernen von Labyrinthaufgaben verwendet.
In HAM kann direkt auf $reduce(H \circ M)$ RL-Verfahren angewendet werden, die RL-Verfahren müssen nur $reduce(H \circ M)$ und nicht den $H \circ M$ komplett betrachten. Zu beachten ist, dass ein Zustand aus dem Tupel(s, m) besteht, wobei s ein Zustand in M und m ein Zustand in H ist. Daher muss z. B. für Q-learning eine erweiterte Zustandstabelle eingesetzt werden [15].
Das Lernen mit MAXQ ist komplexer verglichen zu Options oder HAM. Das resultiert zum einen daraus, dass Zwischenaufgaben Parameter über den Stack erhalten können und daher muss der Parameter beim Lernen berücksichtigt werden. Zum anderen wird im MAXQ direkt eine Policy für alle Level gelernt.

3.2.2 Direktes Lernen bei hintereinander ausgeführten DMPs

Bei HRL mit DMPs, ohne diese im Options-Framework zu verwenden, können diese direkt z. B. mit einem modifizierten PI^2 gelernt werden. Dafür wird neben dem Shape-Parameter der Zielparameter gelernt. Das Lernen des Shape-Parameters wird nicht angepasst. Der Hauptunterschied beim

Lernen des Ziel-Störparameters gegenüber dem Shape-Parameter ist, dass nur die Kosten $S(\tau_{0,m,k})$ zum Zeitpunkt 0 verwendet werden; d. h., es werden genau die Kosten einer ganzen Trajektorie verwendet. Dies kann gemacht werden, da g konstant über den Verlauf einer Trajektorie ist und somit nicht von t abhängt. Im Folgenden werden die Updateschritte des Zielparameters vorgestellt. Die Kostenfunktion einer Trajektorie mit modifiziertem Zielparameter

$$S_{d,k} = \sum_{m=d}^{D} S(\tau_{0,m,k})$$

summiert die Kosten der aktuellen DMP d und aller folgenden DMPs des Roll-outs k auf.

$$P_{d,k} = \frac{\exp(-\frac{1}{\lambda} S_{d,k})}{\sum_{l=1}^{K} \exp(-\frac{1}{\lambda} S_{d,l})}$$

berechnet die Wahrscheinlichkeit für Roll-out k der DMP d, wobei die Wahrscheinlichkeit später als Gewichtungsfaktor verwendet wird. Wenn die Kosten im Roll-out k verglichen zu den anderen Roll-outs klein sind, wird eine hohe Wahrscheinlichkeit / hoher Gewichtungsfaktor zugewiesen. Sind die Kosten verglichen zu den anderen Roll-outs hoch, wird ein kleiner Wert errechnet.

$$\delta_g = \sum_{k=1}^{K} [P_{d,k} \epsilon_k^{g_d}]$$

berechnet den gewichteten Mittelwert aller ϵ, die zum Stören des Zielparameters eingesetzt wurden.

$$g \leftarrow g + \delta g \tag{2}$$

berechnet den neuen Zielparameter.

4 Anwendungen von HRL in der Robotik

4.1 Aufstehen eines Roboters

In [3] wird HRL erfolgreich verwendet, um einem Roboter das Aufstehen zu lehren. Bei diesem Roboter, siehe Abbildung 2, handelt es sich um einen 3 Link 2 Joint Roboter. Der Roboter ist nicht statisch am Boden befestigt, daher muss der Roboter beim Aufstehen und Stehen das Gleichgewicht halten. (Es gibt keine statische Lösung). Der Roboter lässt sich beschreiben durch die 4 Winkel $\theta_0, \theta_1, \theta_2, \theta_m$ und deren Änderung $\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_m$.

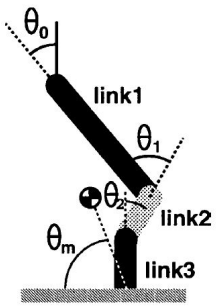

Abbildung 2: Skizze des Roboters, der in [3] das Aufstehen gelernt hat. Skizze übernommen aus [3].

HRL wird für das Lernen des Aufstehvorgangs verwendet, um das Lernen zu beschleunigen. Das schnellere Lernen ist von Vorteil, da es keine statische Lösung gibt, und das System auf die Bewegung reagieren muss.
In [3] wird das Lernen des Aufstehverhaltens in zwei Schichten aufgeteilt. Eine obere Schicht, die den gesamten Aufstehvorgang in einem grob diskretisierten Raum plant. Sowie der unteren Schicht, welche die von der oberen Schicht geplanten Schritte als Zwischenaufgaben erhält und diese versucht auszuführen. Diese Aufteilung entspricht dem Feudal Reinforcement Learning.
Höhere Ebene: Die Dimension in der höheren Ebene wird auf $X = (\theta_m, \theta_1, \theta_2)$ reduziert. Der Zustandsraum wird grob diskretisiert.
Untere Ebene: Es werden die dynamischen Variablen $X = (\theta_0, \theta_1, \theta_2, \dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2)$ des Systems als Dimension verwendet. Der Zustandsraum wird nicht diskretisiert.

4.1.1 Obere Ebene

Beim Lernen wird die nächste Aktion über die Boltzmann Verteilung ausgewählt:

$$P(a_s = a) = \frac{\exp\left[\beta Q(s,a)\right]}{\sum_{b \in A_s} \exp\left[\beta Q(s,b)\right]},$$

wobei A_s die Menge aller möglichen Aktionen im Zustand s ist. Als Belohnungsfunktion dient:

$$R(s) = R_{main} + R_{sub}.$$

Im Gegensatz zu [13] verwendet [3] eine Belohnung für das Erreichen eines Zwischenziels. [3] zeigt, dass die Verwendung einer Belohnung für die obere Schicht, wenn die untere Schicht ein Zwischenziel erreicht, das Lernen schneller und robuster machen kann. R_{main} gibt als Belohnung 1, wenn der Roboter steht. Beim Erreichen des Zwischenziel gibt R_{sub} eine Belohnung zwischen 0.25 und 0.5. Dabei richtet sich die Höhe der Belohnung nach der Höhe und Länge des Roboters bei dem Erreichen des Zwischenziels.

4.1.2 Untere Ebene

In der unteren Ebenen wird über einen kontinuierlichen Zustandsraum $X = (\boldsymbol{\theta} = \{\theta_0, \theta_1, \theta_2\}, \dot{\boldsymbol{\theta}} = \{\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2\})$ gearbeitet. Als RL-Algorithmus wird TD(λ) mit der Actor- Critic-Methode verwendet. Die untere Ebenen erhält eine Belohnung, wenn diese das Zwischenziel erreicht. Bei der Actor-Critic-Methode lernt der Actor die Steuerungsfunktion (Steuerung des Roboters) (für mehr Informationen siehe [3]) und die Critic die Value-Funktion. Da über einen kontinuierlichen Zustandsraum gearbeitet wird, wird eine Funktionsapproximation genutzt. In [3] wird dafür ein Incremental Normalized Gaussian Network (INGnet) [17] verwendet.
Im INGnet werden normalisierte Gauß'sche Basisfunktionen b_i für die Funktionsapproximation verwendet (für weitere Informationen siehe [3]). Die Value-Funktion wird berechnet als:

$$V(s) = \sum_i v_i b_i(s),$$

v_i ist das Gewicht zur Basisfunktion b_i. Der TD-Fehler berechnet sich in diesem Algorithmus wie folgt:

$$\delta(s) = r(s) - \frac{1}{\tau} V(s) + \frac{dV(s)}{dt},$$

$\tau = 0.5[s]$ ist eine Zeitkonstante. Beim Lernen kommt Eligibility Traces zum Einsatz. Der Eligibility Traces errechnet sich wie folgt:

$$\dot{e}_i(s) = -\frac{1}{K} e_i(s) + b_i(s),$$

mit $K = 0.1[s]$ als Zeitkonstante. Über die Lernrate α, den TD-Fehler und die Eligibility Traces ergibt sich die Aktualisierung der Value-Funktion wie folgt:

$$\dot{v}_i = \alpha_c \delta(s) e_i(s).$$

Die Berechnung des Actors kann in [3] nachgelesen werden.
Die direkte Belohnung $r(\boldsymbol{\theta}, \hat{\boldsymbol{\theta}})$ in der unteren Ebene wird über den Abstand der aktuellen Position $\boldsymbol{\theta}$ des Roboters zu der vom oberen Level definierten Zwischenzielposition $\hat{\boldsymbol{\theta}}$ definiert

$$r(\boldsymbol{\theta}, \hat{\boldsymbol{\theta}}) = \exp\left(-\frac{\left\|\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\right\|^2}{s_{\boldsymbol{\theta}}(=30)^2}\right) - 1.$$

Zusätzlich zu dieser Belohnung erhält die untere Ebene eine weitere Belohnung r_t, wenn diese die Position des Zwischenziels erreicht. Dies ist abhängig davon, wie nahe die Geschwindigkeit $\dot{\boldsymbol{\theta}}$ beim Erreichen des Ziels zu dem gewichteten Mittel aller Geschwindigkeiten ist, mit denen vorher das Zwischenziel erreicht wurde.

$$r_t = \begin{cases} \exp\left(-\frac{\|\dot{\boldsymbol{\theta}}(t) - \hat{\dot{\boldsymbol{\theta}}}\|^2}{s_{\dot{\boldsymbol{\theta}}}(=60)^2}\right) & \text{, wenn Zwischenziel zum wiederholten mal erreicht wird} \\ 0 & \text{, wenn das Zwischenziel das erste mal erreicht wird} \\ -1.5 & \text{, wenn der Roboter umfällt} \end{cases}$$

Beim Erreichen des Zwischenziels wird mit $\alpha\hat{\dot{\boldsymbol{\theta}}} + (1-\alpha)\dot{\boldsymbol{\theta}}$ das gewichtete Mittel gebildet. [3] verwendet für α den Wert 0.9.

4.1.3 Simulation und Ergebnis

Der hier vorgestellte HRL-Ansatz wurde von [3] initial in einem Simulator mit dem Roboter als Modell verwendet. Weitere Details zur Simulation wie z. B. die Änderungen der Lernparameter werden in [3] beschrieben. In der Simulation zum Lernen des Aufstehvorgangs hatte der Roboter in [3] maximal 1000 Versuche zur Verfügung. In 7 von 10 Durchgängen hat der Roboter das Aufrichten in weniger als 1000 Versuchen gelernt. Dabei hat der Roboter in der Simulation im Durchschnitt 749 Versuche mit einer Dauer von 30 Minuten benötigt. Im Durchschnitt hat der Roboter 4,3 Zwischenziele beim Lernen des Aufstehvorgangs verwendet. In einer vereinfachten Simulation eines nur 2 Link 1 Joint Roboters wurde in [3] gezeigt, dass der vorgeschlagene Ansatz nicht sehr anfällig bei der Wahl der Diskretisierung in der oberen Ebene ist. Weiterhin wird gezeigt, dass der Roboter schlechter lernt, wenn die Diskretisierung zu grob gewählt wird. In [3] werden keine Angaben gemacht, wie anfällig der 3 Link 2 Joint Roboter zur Diskretisierung der oberen Ebene ist.
In [3] wird beschrieben, dass der Roboter schneller und robuster lernt, wenn HRL anstatt RL angewandt wird. In diesem Vergleich wird für das RL die TD(λ) Actor-Critical-Methode mit einer Hand getunten Belohnungsfunktion genutzt.
Die erfolgreichen 7 Durchgänge, mit denen der Roboter im Simulator das Aufstehen gelernt hat, werden anschließend auf einen echten Roboter als Anfangswert übertragen. Bei 6 von 7 Initialwerten hat der Roboter nach durchschnittlich 164 Versuchen das Aufstehen gelernt.

4.2 Optimierung von hintereinander ausgeführten Roboterbewegungen.

In [9] wird HRL verwendet, um die Kosten einer Pick-and-Place-Aufgabe zu reduzierten. Dabei muss der Roboterarm zu einem Objekt bewegt werden, dieses greifen, dann zur nächsten Position bewegt werden (ein Regalfach weiter oben) und dort als letzten Schritt das Objekt wieder loslassen. Die Bewegung des Roboterarms zum Objekt und die Bewegung des Roboterarms mit dem Objekt vom unteren Fach zum oberen Fach wird mit DMPs modelliert. Die Pick-and-Place-Aufgabe kann der Roboterarm bereits ausführen, HRL wird zur Kostenminimierung verwendet. Wie das Greifen des Objekts erfolgt, wird in [9] nicht dargestellt. Der Greifvorgang wird für die Aufgabe einfach gehalten.
Konkret wird HRL verwendet, um die Form der zwei DMPs zu optimieren, d. h. die Kosten der Bewegung zu reduzieren, sowie neben der Form der DMPs die Endpositionen der DMPs zu lernen. [9] verwendet zum Lernen des Shape-Parameters der DMPs den PI^2 Algorithmus und zum Lernen der Zielparameter der DMPs den im Unterkapitel 3.2.2 vorgestellten Algorithmus. Die Endposition (Höhe mit der der Roboterarm das Objekt greift) der ersten Bewegung hat einen Einfluss auf die Kosten der ersten und zweiten Bewegung. Ziel des Lernens der Endpositionen ist es, die summierten Kosten über beide Bewegungen zu reduzieren. Dabei können die Kosten stärker reduziert werden als es ohne das Modifizieren der Endpostion möglich ist. Menschen fassen ein Objekt tief an, wenn sie es von einem niedrigeren in ein höheres Fach legen wollen. Andererseits greifen sie ein Objekt weiter oben, wenn sie beabsichtigen, es in ein tieferes Fach zu legen [9].

In [9] wird ein Roboterarm mit 3 DOF verwendet. Die Hand am Roboterarm hat 4 DOF. Die Position der Hand wird während der Armbewegung fixiert und daher beim HRL der 2 DMPs nicht berücksichtigt. Der Roboterarm hat ein System, das die Kollision des Arms oder des Objekts in der Hand mit dem Regal verhindert. Die Aufgabe wird über imitiertes Lernen initiiert. Der Roboter kann nach dem imitierten Lernen die Bewegung bereits erfolgreich ausführen. Es wird nur die Bewegung bezüglich der Kostenfunktion mit HRL optimiert. Der Suchraum der HRL-Aufgabe hat 12 Dimensionen, da der Arm 3 DOF hat, zwei DMPs betrachtet und für beide DMPs die Zielposition und der Shape-Parameter gelernt werden [9]. Als Kostenfunktion für die Trajektorie τ kommt folgende Funktion zum Einsatz:

$$J(\tau_i) = \phi_{t_N} + \frac{1}{N} \int_{t_i}^{t_n} (4G(t) + |\ddot{x}_{avoid}| + |\ddot{x}|/250)\, dt,$$

wobei

- $G(t)$ 1 ist, wenn der Roboterarm das Objekt nicht in der Hand hat;
- $|\ddot{x}_{avoid}|$ die Kosten angibt, wenn das Antikollisionssystem eingreifen muss, um dem Regal auszuweichen;
- $|\ddot{x}|$ die Beschleunigung des Arms beschreibt, der Arm soll sich nicht zu schnell bewegen;

ϕ_{t_N} die Höhe bezeichnet, aus der das Objekt am Ende los gelassen wird.

Für die Optimierung werden k = 5 Durchläufe für jedes Update der Parameter genutzt. Insgesamt werden über 60 Roll-outs gelernt.

Um die Effektivität des HRL bei der Pick-and-Place-Aufgabe zu evaluieren, testet [9] auf dieser Aufgabe weitere Verfahren: Optimieren des Shape- oder Zielparameters der DMPs der Bewegungen mit RL, sowie die Optimierung des Shape- und Zielparameters der DMPs nach einer Greedy-Strategie. Bei dieser Greedy-Strategie werden der Ziel- und Shape-Parameter nur zu den Kosten einer Bewegung optimiert. Beim Vergleich dieser Ansätze hat das HRL-Verfahren die beste Optimierung der summierten Kosten über beide DMPs nach 60 Roll-outs erzielt.
Der Roboter hat bei der Kostenoptimierung mit HRL gelernt, bei der Aufgabe, das Objekt in ein höheres Fach zu bewegen, das Objekt tiefer zu greifen; dieses Greifverhalten entspricht dem des Menschen [9].
Bei einer zweiten Pick-and-Place-Aufgabe, diesmal ein Objekt in ein tiefer gelegenes Fach zu bewegen, hat der Roboter bei der Kostenreduzierung mit HRL gelernt, das Objekt weiter oben zu greifen, was ebenfalls dem menschlichen Greifverhalten entspricht.

5 Fazit

Die Verwendung von HRL gegenüber RL hat für die Robotersteuerung Vorteile. In diesem Paper wurden Beispiele gezeigt, in denen HRL neben den theoretischen Vorteilen wie Einschränkung der möglichen Policies oder Beschleunigung des Lernens über zeitlich verlängerte Aktionen auch in der praktischen Anwendung der Robotersteuerung von Vorteil ist. Morimoto und Doya [3] haben mit HRL einem Roboter das Aufstehen beigebracht; eine Aufgabe, an der sie zuvor mit RL gescheitert sind. Stulp und Schaal [9] haben die summierten Kosten von mehreren hintereinander ausgeführten Roboterbewegungen mit HRL stärker reduzieren können als nur mit der Verwendung von RL auf die einzelnen Bewegungen. Weiterhin hat der Roboter damit dem Menschen ähnliches Verhalten gelernt. Trotz der Erfolge des HRL in der Robotersteuerung existieren noch offene Probleme. HRL wird zwar erfolgreich bei Problemen mit höheren Dimensionen verwendet, die verschiedenen Ebenen sind aber in der Regel immer noch fix und werden nicht gelernt.
Neben der Anwendung von HRL für die Robotersteuerung gibt es noch andere Ansätze, die erfolgreich zum Einsatz kommen.
Eine Alternative zum HRL ist (Hierarchical-) Apprenticeship Learning (AL). AL ist dem RL ähnlich und arbeitet auch über (S)MDPs. Im Unterschied zu RL wird aber keine benutzerdefinierte Belohnungsfunktion eingesetzt. Beim AL sind daher Beispiele nötig; von den Beispielen wird die Value-Funktion für die Zustände ermittelt. Hierarchical Apprenticeship Learning wird in [2] verwendet, um einem 4-beinigen Roboter das Laufen über „schwieriges“ Terrain bei zu bringen.
Eine weitere Alternative ist die Anwendung von RL auf den reduzierten Zustandsraum eines Roboters. Ein Zustandsraum kann beispielsweise hierarchisch aufgeteilt werden. Dies ermöglicht die Anwendung von RL auf einer der reduzierten Ebenen. Dieser Ansatz erlaubt es, RL auch bei Robotern mit höheren DOF zu verwenden, hat aber den Nachteil, dass das Lernen weniger flexibel ist. In [18] wird z. B. mit RL das Zentrum der Masse (CoM) eines Roboters gelernt. Eine Steuerungseinheit errechnet aus den Änderungen der CoM des Roboters die Änderungen für die Joints des Roboters. [18] konnte mit diesem Ansatz erfolgreich einem humanoiden Roboter eine Stoßbewegung beibringen. Exemplarisch wurde in [18] ein Spielzeugauto von einem humanoiden Roboter weggestoßen.

Acknowledgments

Für hilfreiche Diskussionen möchte ich Johannes Viering danken.

Literatur

[1] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, July 2009.

[2] J. Zico Kolter, Pieter Abbeel, and Andrew Y. Ng. Hierarchical apprenticeship learning with application to quadruped locomotion. In *NIPS*, 2007.

[3] Jun Morimoto and Kenji Doya. Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems*, 36(1):37 – 51, 2001.

[4] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. 13:2003, 2003.

[5] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[6] Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. In *Machine Learning*, pages 226–232. Morgan Kaufmann, 1996.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[8] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *In IEEE International Conference on Robotics and Automation (ICRA2002*, pages 1398–1403, 2002.

[9] Freek Stulp and Stefan Schaal. Hierarchical reinforcement learning with motion primitives. In *11th IEEE-RAS International Conference on Humanoid Robots*, 2011.

[10] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *J. Mach. Learn. Res.*, 11:3137–3181, December 2010.

[11] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomput.*, 71(7-9):1180–1190, March 2008.

[12] J. Kober and J. Peters. policy search for motor primitives in robotics. In *advances in neural information processing systems 22 (nips 2008), cambridge, ma: mit press*, 2009.

[13] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan Kaufmann, 1993.

[14] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[15] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 conference on Advances in neural information processing systems 10*, NIPS '97, pages 1043–1049, Cambridge, MA, USA, 1998. MIT Press.

[16] Christian Daniel, Gerhard Neumann, and Jan Peters. Hierarchical relative entropy policy search. *Journal of Machine Learning Research - Proceedings Track*, 22:273–281, 2012.

[17] Jun Morimoto and Kenji Doya. Reinforcement learning of dynamic motor sequence: Learning to stand up. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1721–1726, 1998.

[18] Takamitsu Matsubara, Jun Morimoto, Jun Nakanishi, Sang-Ho Hyon, Joshua G. Hale, and Gordon Cheng. Learning to acquire whole-body humanoid com movements to achieve dynamic tasks. In *ICRA*, pages 2688–2693, 2007.