Hochschule Bremerhaven

Exposé für eine Bachelorarbeit zum Thema:

Reinforcement Learning

Theoretische Grundlagen der tabellarischen Lernmethoden und praktische Umsetzung am Beispiel eines Ameisen-Agentenspiels

Autor:	Jan Löwenstrom
Matrikelnr.:	34937
Erstprüfer:	Prof. Dr.-Ing. Henrik Lipskoch
Zweitprüfer:	Prof. Dr. Mathias Lindemann

22. März 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

//TODO

2 Grundlagen

Bei dem Bestärkenden Lernen (*Reinforcement Learning*) interagiert ein Softwareagent (*Agent*) mit seiner Umwelt (*Environment*), die wiederum nach jeder Aktion (*Action*) Feedback an den Agenten zurückgibt. Dieses Feedback wird als Belohnung (*Reward*) bezeichnet, einem numerischen Wert, der sowohl positiv als auch negativ sein kann. Der Agent beobachtet zudem den Folgezustand (*State*) in dem sich die Umwelt nach der vorigen Aktion befindet, um so seine nächste Entscheidung treffen zu können. Ziel des Agenten ist es eine Strategie (*Policy*) zu entwickeln, so dass die Folge seiner Entscheidungen die Summe aller Belohnungen maximiert.

2.1 Markov Entscheidungsprozess

Die Umwelt wird in dieser Arbeit als Markov'scher Entscheidungsprozess (*Markov Decision Process, MDP*) definiert. Dieses Framework findet häufig Verwendung in der stochastischen Kontrolltheorie (Gosavi, 2009, S. 3) und bietet im Bezug auf das *Reinforcement Learning* Problem den mathematischen Rahmen, um u.a. präzise theoretische Aussagen treffen zu können Als *MDP* versteht sich die Formalisierung von sequentiellen Entscheidungsproblemen, bei denen eine Entscheidung nicht nur die sofortige Belohnung beeinflusst, sondern auch alle Folgezustände und somit auch alle zukünftigen Belohnungen (Sutton & Barto, 2018, S. 47). Ein Entscheidungsfinder muss somit das Konzept von verspäteten Belohnungen (*delayed rewards*) durchdringen. Vermeintlich schlecht erscheinende Entscheidungen in der Gegenwart können sich im Nachhinein als optimal herausstellen, angesichts der gesamten Handlung. Ein*e Skatspieler*in könnte z.B. alle Trümpfe direkt am Anfang spielen, um einen sofortigen Vorteil zu erhalten. Für den Spielausgang ist es aber womöglich besser, die Trümpfe für einen späteren Zeitpunkt aufzubewahren und zu Beginn „schlechte " Entscheidungen zu treffen, die dazu führen, ein paar Stiche zu verlieren.

Probleme, die als *MDP* definiert werden, müssen die Markov-Eigenschaft erfüllen, da diese gewissermaßen als Erweiterung von Markov-Ketten zu betrachten sind, mit dem Zusatz von Aktionen und Belohnungen. Bei den sog. Markov-Ketten führt das System

zufällige Zustandswechsel durch (Gosavi, 2009, S. 3). Dabei sind die Übergangswahrscheinlichkeiten zu den einzelnen Folgezuständen ausschließlich von dem aktuellen Zustand abhängig und nicht aufgrund des historischen Verlaufs (Gosavi, 2009, S. 3). Da die Markov-Eigenschaft eine essentielle Voraussetzung bei der Problemmodellierung ist, wird sie in Kapitel X näher erläutert. Die Beziehung zwischen Agent und Umwelt, kann durch folgendes Interface dargestellt werden (Sutton & Barto, 2018, S. 48):

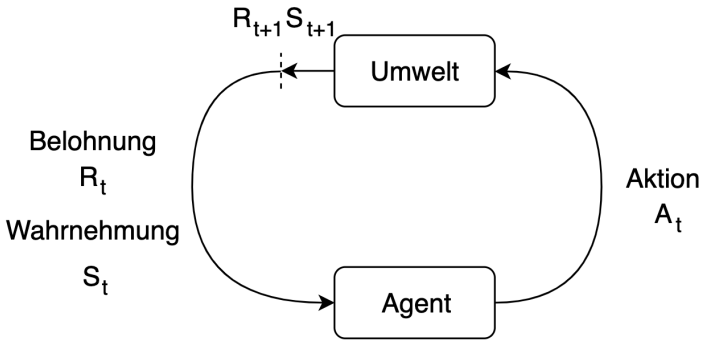

Abbildung 1: Agent-Umwelt Interface

Der Agent interagiert mit dem *MDP* jeweils zu diskreten Zeitpunkten $t = 0, 1, 2, 3, \ldots$. Zu jedem Zeitpunkt t beobachtet der Agent den Zustand seiner Umgebung $S_t \in \mathcal{S}$ und wählt aufgrund dessen eine Aktionen $A_t \in \mathcal{A}$. Als Konsequenz seiner Aktion erhält er einen Zeitpunkt später eine Belohnung $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ und stellt den Folgezustand S_{t+1} fest.

In der Literatur findet sich jedoch auch eine abweichende Definition im Bezug auf den Zeitpunkt der Belohnungsvergabe. C. J. C. H. Watkins (1989), Wiering & van Otterlo (2012) und Yu et al. (2009) z.B. binden die Belohnung R_t an das Zustands-Aktions-Paar (S_t, A_t). Die Definition R_{t+1} bei Aktion A_t von Sutton & Barto (2018) wird allerdings im Verlauf dieser Arbeit verwendet, da sie besser beschreibt, dass die Belohnung und der Folgezustand gemeinsam berechnet werden und einen Zeitpunkt später, nach Aktion A_t, für den Agenten sichtbar sind.

Das Zusammenspiel zwischen Agenten und MDP erzeugt somit folgende Reihenfolge (Sutton & Barto, 2018, S.48):

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.1}$$

Wird einfach nur von *MDPs* gesprochen, ist die endliche Variante (*finite MDP*) gemeint, bei dem die Mengen der Zustände, Aktionen und Belohnungen $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ eine endliche Anzahl an Elementen besitzen. In diesem Fall haben die Zufallsvariablen R_t und S_t wohl definierte, diskrete Wahrscheinlichkeitsverteilungen, die nur von dem vorigen Zustand und der vorigen Aktion abhängig sind. Die Wahrscheinlichkeit, dass die bestimmten Werte für diese Variablen $s' \in \mathcal{S}$ und $r \in \mathcal{R}$ eintreten, für einen bestimmten Zeitpunkt t und dem vorigen Zustand s und Aktion a, kann somit durch folgende Funktion beschrieben werden (Sutton & Barto, 2018, S.48):

$$p(s', r \mid s, a) \doteq Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}, \tag{2.2}$$

für alle $s', s \in \mathcal{S}, r \in \mathcal{R}$ und $a \in \mathcal{A}(s)$. Diese Funktion p definiert die sog. Dynamiken (*Dynamics*) eines *MDP*. Sie ist eine gewöhnliche deterministische Funktion mit vier Parametern $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. Das „|“ Zeichen kommt ursprünglich aus der Notation für bedingte Wahrscheinlichkeiten, soll hier aber andeuten, dass es sich um eine Wahrscheinlichkeitsverteilung handelt für jeweils alle Kombinationen von s und a (Sutton & Barto, 2018, S.49f):

$$\forall s \in \mathcal{S} : \forall a \in \mathcal{A}(s) : \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1 \tag{2.3}$$

Ist das Entscheidungsproblem nicht stochastischer Natur, sondern deterministisch, so ist für jedes Paar (s, a) genau ein bestimmtes Paar (s', r) gleich 1. Für alle weiteren Kombinationen von (s', r) im Bezug auf dieses Zustands-Aktions-Paar (s, a) ist p folglich gleich 0. Mit anderen Worten, wird im Zustand s die Aktion a gewählt, führt dies in jedem Fall zu einem bestimmten Folgezustand s'.

Sutton & Barto (2018) erläutern, dass das MDP Framework als extrem flexibel gilt und es demzufolge auf die unterschiedlichsten Probleme angewendet werden kann. Sie führen weiter aus, dass es die nötige Abstraktion für Probleme bietet, bei denen

unter Vorgabe eines Ziels mittels Interaktionen gelernt wird. Einzelheiten über das eigentliche Ziel, die Zustände oder die Form des Agenten sind dabei unerheblich. Letztendlich kommen die zwei Autoren zu dem Schluss, dass „jedes zielgerichtete Lernen auf drei Signale reduziert werden kann, die zwischen dem Agenten und der Umwelt ausgetauscht werden. Ein Signal repräsentiert die Entscheidung, die der Agent getroffen hat (die Aktion), ein Signal repräsentiert die Basis, auf der er zu dieser Entscheidung gekommen ist (der Zustand) und ein Signal definiert das zu erreichende Ziel (die Belohnung)“ (S. 50).

2.2 Markov-Eigenschaft und Zustandsmodellierung

Die Markov-Eigenschaft erhält ein eigenes Kapitel, da sie wichtig zum Verständnis dieser Arbeit ist und bei der Modellierung eines Reinforcement Learning Problems eine besondere Rolle spielt. Verbinden lässt sich dies sehr gut mit einem Einblick über die grundsätzliche Modellierung von Zuständen bei einem Reinforcement Learning Problem.

> The future is independent of the past given the present

Dieser Satz erscheint oft in der Literatur, wenn es um die Markov-Eigenschaft geht, so z.B. in den Arbeiten von Feldman & Valdez-Flores (2010), Kumar (2014), Capela et al. (2019) und Saul & Jordan (1999), oder auch in der Vorlesung der Stanford-Professorin Emma Brunskill (2019). Er fasst prägnat zusammen, was die Markov-Eigenschaft aussagt. Im Zusammenhang von MDPs lässt sich dieser Satz so übersetzen, dass ein Folgezustand nicht abhängig von Aktionen bzw. Zuständen in der Vergangenheit ist, sondern ausschließlich von dem aktuellen Zustand und der aktuell gewählten Aktion.

Sutton & Barto (2018) sehen die Markov-Eigenschaft als Einschränkung für die Zustände und nicht für den Entscheidungsprozess als solches. Ausschlaggebend ist, dass der Zustand, auf dessen Basis der Agent seine Entscheidung trifft, alle notwendigen Informationen der Vergangenheit beinhaltet, die für die Zukunft relevant sind (S.49). Die Umwelt ist somit nicht notwendigerweise gezwungen, Markov-konforme Zustände zu liefern. Brunskill (2019) wählt aufgrunddessen die Bezeichnung „Beobachtung“ (Observation O_t) als Feedback der Umwelt nach einer Aktion. Jene Beobachtungen können

anschließend durch eine interne Repräsentation zu Markov-Zuständen verarbeitet werden, die dann dem Entscheidungsfinder zugrunde liegen.

Folgendes Beispiel, basierend auf der Vorlesung von Brunskill (2019), liefert einen guten Einblick in die Zustandsmodellierung und der Problematik, die mit der Markov-Eigenschaft einhergeht.

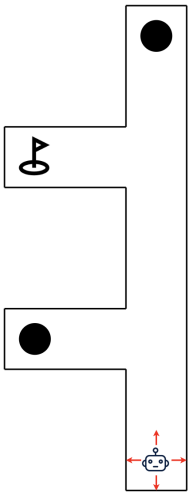

Abbildung 2: Zwei-Wege Beispiel zu der Markov-Eigenschaft

Gegeben ist ein beweglicher Roboter und eine Strecke mit zwei Korridoren. Der Roboter ist mit vier Sensoren ausgestattet, die jeweils eine Himmelsrichtung abdecken. Diese Sensoren sind in der Lage, angrenzende Wände zu erkennen und bilden den Zustand der Umwelt ab. Wahlweise ist der Zustand im Uhrzeiger definiert $\{N, O, S, W\}$, wobei 1 angibt, dass eine Wand erkannt wurde und 0, dass sich keine Wand in der unmittelbaren Nähe befindet. Es ergeben sich folglich 16 unterschiedliche Zustände, die der Agent unterscheiden und auf dessen Basis er Entscheidungen treffen kann (vier Aktionen: Fahrt in jeweils eine Richtungen). Der Roboter soll sein Ziel erreichen, markiert mit einer Flagge, ohne dabei in eine der beiden Fallen zu navigieren.

Eine potentielle Startposition, wie in Abb. 2 dargestellt, liefert somit den Zustand $\{0, 1, 1, 1\}$. Angenommen der Agent hat gelernt in diesem Zustand Richtung Norden zu fahren, dann ist der Folgezustand ebenfalls $\{0, 1, 1, 1\}$. Schließlich erreicht er den ersten

Korridor. Der westliche Sensor liefert folgerichtig 0 und der Zustand ist $\{0, 1, 0, 0\}$. Da der Agent nicht den ersten Korridor folgen darf, sondern dem zweiten, muss der Zustand $\{0, 1, 0, 0\}$ ebenfalls die Aktion „nach Norden fahren“ auslösen. Das Besondere hier ist jedoch, dass der Zustand bei dem zweiten Korridor identisch mit dem Zustand bei dem ersten Korridor ist und der Agent somit keine Chance hat, zu unterscheiden, vor welchem er sich gerade befindet, siehe Abb. 6. Er würde ebenfalls, wie schon bei dem ersten Korridor, weiter nach Norden und letztendlich in die Falle fahren.

Bezogen auf diesen Entscheidungsprozess ist die Modellierung der Zustände über den Sensorinput alleine nicht ausreichend, um die gestellte Aufgabe zu lösen. Die Kombination von Aufgabenstellung und dem Format der Zustände in dieser Form erfüllt insofern nicht die Markov-Eigenschaft, dass auf Basis der erkannten Zustände keine Möglichkeit besteht, die optimalen Entscheidungen zu treffen.

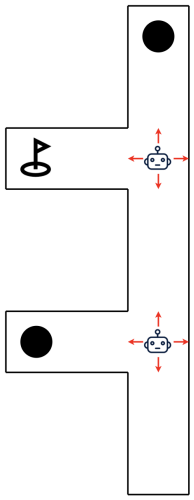

Abbildung 3: Zwei-Wege Beispiel Forts.

In der Theorie ist es jedoch möglich diesen Entscheidungsprozess als MDP umzumodellieren. Dabei werden die Sensordaten als Beobachtungen der Umwelt betrachtet und eine interne Repräsentation von Markov-Zuständen gepflegt. Möglich ist z.B. die gesamte Historie der Zustände und Aktionen zu speichern, damit der Roboter zurückverfolgen kann, wo er sich zur Zeit befindet. Ein Prozess als MDP zu definieren bedeutet aber gerade darauf zu verzichten, nämlich die gesamte Vergangenheit in einen Zustand zu verarbeitet. Denkbar ist auch, dass der Agent eine interne Repräsentation nach jeder Beobachtung pflegt und die Umwelt sukzessive nachbildet.

//TODO Schlussfolgerung; Modellierung
Letztendlich sollte

2.3 Belohnungen und Zielstrebigkeit

Das Besondere an dem Reinforcement Learning ist das Belohnungssignal (*Reward*), welches der Agent nach jeder Aktion erhält. Zu jedem diskreten Zeitpunkt wird dem Agenten eine Belohnung in Form einer einfachen Zahl $R_t \in \mathbb{R}$ zugestellt. Aufgabe eines jeden RL-Algorithmus ist es, die Summe aller gesammelten Belohnungen zu maximieren. Dabei ist entscheidend, dass der Fokus nicht ausschließlich auf die sofortigen Belohnungen gerichtet ist, sondern auf die erwartbare Summe aller Belohnungen über einen langen Zeitraum. Entscheidungen, die in der Gegenwart eine hohe sofortige Belohnungen versprechen sind verführerisch, können sich aber in der Zukunft in Bezug auf den gesamten Prozess als suboptimal herausstellen. (Sutton & Barto, 2018, S.53)

Eine Belohnungsfunktion wird in der Regeln von einem Menschen definiert und hat den größten Einfluss darauf, wie der Agent sich verhalten soll. Die Festlegung von Belohnung bei bestimmten Events ist die einzige Möglichkeit, die der Agent hat, zu verstehen, welches Ziel er verfolgen soll. Somit ist die Modellierung der passenden Belohnungsfunktion zur korrekten Abbildung der eigentlichen Aufgabenstellung von gravierender Bedeutung.

//TODO Belege Grundsätzlich gibt es zwei Ansätze, um eine Belohnungsfunktion zu formulieren. Verständlich werden diese durch ein Beispiel, bei dem ein Agent lernen soll, eine Partie Schach zu gewinnen. Die erste Möglichkeit besteht darin, dem Agenten ausschließlich eine Belohnung aufgrund des Spielausgangs zu geben. Er erhält +1 wenn er gewinnt, -1 bei einer Niederlage und 0 bei Unentschieden (und jeder Aktion zuvor). Auf den ersten Blick erscheint dieser Ansatz trivial, ist aber die direkte Übersetzung des Ziels in eine Belohnungsfunktion. Die größte erwartbare Summe aller Belohnungen erhält der Agent nur, wenn er lernt, das Spiel zu gewinnen. Größter Nachteil dieser Methode ist allerdings, dass der Agent keinerlei Hilfe oder Richtung bei dem Erkunden des Spiels erhält. Je größer der Zustands- und Aktionraum ist, desto länger braucht er um überhaupt einmal ein Spiel gewinnen zu können und zu lernen, welche Aktionen vorteilhaft sind und welche nicht.

Um dem entgegenzuwirken, werden dem Agenten bei der zweiten Möglichkeit feingranularere Belohnungen mitgeteilt, statt diese ausschließlich auf das Endresultat zu

reduzieren. Bestimmte Belohnungen zeigen dann, ob der Agent seinem Ziel näher gekommen ist oder eine ungünstige Entscheidung getroffen hat. Zum Beispiel könnte dem Agenten eine hohe Belohnung von +10 gegeben werden, wenn er die gegnerische Dame aus dem Spiel nimmt. Es ist auch denkbar, dass jede Spielfeldkonstellation bewertet wird. Dieser Ansatz benötigt somit spezielles Vorwissen über das Problem und kann sich zugleich sehr negativ auf das Verfolgen des eigentlichen Ziels auswirken. Der Agent könnte zum Beispiel nur lernen in jedem Spiel die Dame des Gegners zu schlagen und dabei trotzdem immer die Partie zu verlieren.

Die korrekte Modellierung der Belohnungsfunktion hat somit eine besondere Bedeutung. Sutton & Barto (2018) sind der Meinung, dass ein Schachagent nur angesichts des Spielaussgangs bewertet werden sollte und nicht aufgrund von Zwischenzielen wie z.B. dem Herausnehmen einer gegnerischen Spielfigur oder der Kontrolle über das Zentrum des Spielfelds (S. 53).

Für eine korrekte Übersetzung der Aufgabenstellung zu einer geeigneten Belohnungsfunktion gibt es keine klaren, formalen Regeln. Ein*e Designer*in muss auf Erfahrungswerte und einen gewissen Grad an Kreativität zurückgreifen. Soll ein Agent z.B. ein Labyrinth durchlaufen und so schnell wie möglich hinausfinden, dann muss nach jeder Aktion eine negative Belohnung von -1 verteilt werden. Somit wird der Agent gezwungen, auf direktem Wege den Ausgang zu erreichen. Würde lediglich für das Erreichen des Ausgangs eine positive Belohnung vergeben werden, dann wäre die Summe aller Belohnungen für jede Abfolge von Aktionen gleich. Der Agent „trödelt“. Hat er durch Zufall aus dem Labyrinth gefunden, so könnte er bei weiteren Durchläufen keinen effektiveren Weg finden, denn für ihn haben alle Aktionsfolgen den gleichen Nutzen.

Prinzipiell gilt, dass „das Belohnungssignal dazu dient, dem Agenten mitzuteilen *was* er erreichen soll, nicht *wie* er es erreichen soll“ (Sutton & Barto, 2018, S. 54).

2.4 Gewinn und Episoden

In Kapitel 2.1 wurde gezeigt, dass die Interaktion eines Agenten mit seiner Umwelt als bestimmte Abfolge beschrieben werden kann (2.1). In ihr werden letztendlich alle

Triple von Zustand, ausgeführter Aktion aufgrund dieses Zustands und anschließende Belohnung chronologisch aufgezeichnet. Ist diese Reihenfolge endlich, so wird sie auch als Episode (*Episode*) bezeichnet. Eine Episode fasst somit alle Informationen zusammen, die ein Agent erlebt, während er von einem beliebigen Startzustand aus anfängt die Umwelt zu erkunden. Das Ende einer Episode wird durch das Erreichen eines beliebigen Zielzustands erreicht. Ist eine Episode zu Ende, dann wird das Szenario zurückgesetzt und der Agent startet erneut im Startzustand. Episoden sind komplett unabhängig voneinander und erzeugen Abfolgen, die nicht durch vorrige Episoden beeinflusst sind.

Bisher wurde erwähnt, dass das Ziel eines Agenten sei, die Summe der zu erwartenden Belohnungen zu maximieren. Formal betrachtet, versucht er somit die Sequenz der Belohnungen, die er nach dem Zeitpunkt t erhält, den sog. erwarteten Gewinn (*Return*), zu maximieren. Im einfachsten Fall sieht G_t wie folgt aus, wobei T der finale Zeitstempel ist (Sutton & Barto, 2018, S.55):

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.4}$$

Bei episodialen Problemen lässt sich der Gewinn durch diese Addition von nachfolgenden Belohnungen für jeden Zeitpunkt t ermitteln. Grund hierfür ist, dass wärend der Berechnung, nach Abschluss der Episode, alle Belohnungen bekannt sind.

Jedoch existieren auch Probleme, die keine Endzustände definiert haben und daher einen sog. unendlichen Zeithorizont (*infinite horizon*) besitzen. Sie lassen sich nicht in natürliche Sequenzen unterteilen und werden auch mit „kontinuierlich“ betitelt, wobei dadurch ausschließlich beschrieben wird, dass die Interaktion zwischen Agenten und Umwelt kein definiertes Ende besitzt, die Zeitstempel sind weiterhin diskret. Folglich ist $T = \infty$, was wiederum bedeutet, dass der Gewinn unendlich ist.

Um diese kontinuierlichen und die zuvor beschriebenen episodialen Aufgaben im Bezug auf den Gewinn zu vereinheitlichen, wird das Konzept der Diskontierung (*discounting*) verwendet. Dabei gibt der Parameter γ, $0 \leq \gamma \leq 1$, Auskunft darüber, wie die Gewichtung zwischen sofortigen und zukünftigen Belohnungen verteilt ist. Der zukünftige diskontierte Gewinn, der durch die Aktion A_t maximiert werden soll,

berechnet sich somit wie folgt (Sutton & Barto, 2018, S.55):

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

Eine wichtige Erkenntnis ist, dass Gewinne aufeinanderfolgender Zeitpunkte in Verbindung stehen. Vor allem Algorithmen, die nach jedem Zeitstempel updaten, profitieren von dieser Eigenschaft. Sie verwenden den geschätzten Gewinn des Folgezustands, also G_{t+1}, zur Berechnung von G_t, dem geschätzten Gewinn des aktuellen Zustands. Dieses Verfahren, bei dem ein Schätzwert aufgrund eines anderen Schätzwertes aktualisiert wird, wird auch als *bootstrapping* bezeichnet.

Durch simple Umformung wird der Zusammenhang von Gewinnen deutlich (Sutton & Barto, 2018, S.55):

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.6}$$

Ist $\gamma = 0$, dann wählt der Agent seine Aktionen ausschließlich aufgrund der sofortigen Belohnung R_{t+1}. Je näher γ an 1 ist, desto „weitsichtiger“ wird der Agent, da der Gewinn für den Zeitpunkt t sich zusätzlich aus zukünftigen Belohnungen zusammensetzt. $\gamma = 1$ führt zu der gleichen Summe wie (2.4) und wird bei Problemen bestimmt, die Episoden erzeugen. Dadurch trifft der Agent seine Entscheidungen immer aufgrund jeglicher Konsequenzen in der Zukunft bzw. bis zum Ende der jeweiligen Episode. Um zu erreichen, dass die unendliche Summe in (2.5) bei kontinuierlichen Aufgaben einen endlichen Wert annimmt, muss $\gamma < 1$ gegeben sein.

Probleme mit unendlichem Zeithorizont können durch die Vergabe einer künstlichen Schranke zu einer episodialen Aufgabe umformuliert werden. Denkbar z.B. durch die Festlegung der maximalen Anzahl an Aktionen oder besuchten Zustände.

//TODO TD-Episodic tasks?! Weglassen?

Die Algorithmen der Monte-Carlo-Methoden, die in Kapitel 3.2 vorgestellt werden, können ausschließlich auf Basis von Episoden lernen. Jedoch existieren auch Methoden,

wie das Temporal-Difference-Learning, siehe Kapitel 3.3, die neben dem episodialen Lernen, zusätzlich in der Lage sind, mit kontinuierlichen Aufgaben zurechtzukommen.

2.5 Strategie und Nutzenfunktion

Fast alle Lernalgorithmen des Reinforcement Learning versuchen eine sog. Nutzenfunktion (*Value Function*) zu schätzen. Diese Funktion sagt aus, „wie gut“ es ist, dass sich der Agent in einem bestimmten Zustand befindet oder eine bestimmte Aktion in einem Zustand ausführt. Dabei bezieht sich das „wie gut“ darauf, welche Belohnungen in der Zukunft erwartbar sind, also wie groß der erwartete Gewinn ist. Zukünftige Belohnungen sind natürlicherweise abhängig davon, wie sich der Agent verhalten bzw. welche Entscheidungen er in der Zukunft treffen wird. Nutzenfunktion sind deshalb immer in Bezug auf eine bestimmte Strategie definiert (Sutton & Barto, 2018, S. 58).

Eine Strategie (*Policy*) kann als Abbildung verstanden werden, die jedem Zustand eine diskrete Wahrscheinlichkeitsverteilung über Aktionen zuordnet. Folgt der Agent einer Strategie π zum Zeitpunkt t, dann gibt $\pi(a \mid s)$ an, mit welcher Wahrscheinlichkeit $A_t = a$ ausgeführt wird, wenn $S_t = s$ (Sutton & Barto, 2018, S. 58). Neben solchen stochastischen Strategien, existieren auch simplere, deterministische Strategien, die jedem Zustand nur genau eine Aktion zuordnen, $\pi(s) = a$ (Brunskill, 2019).

Wie anfangs erwähnt, gibt es zwei Varianten der Nutzenfunktion. Die erste sagt aus, wie groß der Erwartungswert des Gewinns für den Zustands s ist, wenn in diesem gestartet und anschließend aufgrund der Strategie π gehandelt wird. Dieser *Zustands-Nutzen* kann für alle $s \in \mathcal{S}$ folgendermaßen definiert werden (Sutton & Barto, 2018, S. 58):

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \tag{2.7}$$

Die zweiten Variante gibt Auskunft darüber, wie groß der Nutzen ist, wenn im Zustand s gestartet, daraufhin die Aktion a ausgeführt und anschließend der Strategie π gefolgt wird. q_π wird auch als *Aktions-Nutzenfunktion* für die Strategie π bezeichnet und wird

formal ausgedrückt durch (Sutton & Barto, 2018, S. 58):

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{2.8}$$

//TODO der Erwartungswert auf ? //TODO funktionsapproximation hier?

2.6 Optimalität

Ein Reinforcement Learning Problem zu lösen bedeutet, eine Strategie zu finden, die den größten Gewinn bringt. Dabei lassen sich Strategien vergleichen, insofern, dass eine Strategie besser ist als eine andere, wenn der erwartete Gewinn für alle Zustände größer oder gleich ist (Sutton & Barto, 2018, S. 62f). Mit anderen Worten, $\pi \geq \pi'$ gilt, wenn $v_\pi(s) \geq v_{\pi'}(s)$ für alle $s \in \mathcal{S}$. Es existiert mindestens eine Strategie die besser oder gleich gegenüber allen anderen Strategien ist. Diese ist die optimale Strategie π_*. Optimale Strategien teilen die selbe (optimale) Zustands-Nutzenfunktion v_* und (optimale) Aktions-Zustands-Nutzenfunktion q_* (Sutton & Barto, 2018, S. 62f):

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.9}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.10}$$

Nutzenfunktionen sind, wie im vorigen Kapitel (2.5) erläutert, immer abhängig von einer bestimmten Strategie, da diese die gesammelte Erfahrung beeinflusst und somit auch die erwarteten, geschätzten Gewinne. Die optimale Nutzenfunktion kann jedoch auch ohne Referenz auf eine bestimmte Strategie beschrieben werden, da der Gewinn eines Zustands unter einer optimalen Strategie gleich dem erwarteten Gewinn für die beste Aktion in diesem Zustand ist. $v_*(s)$ referenziert somit $v_*(s')$, den besten Folgezustand, wodurch eine rekursive Beziehung zustande kommt. Eine optimale Nutzenfunktion v_* kann formal folgendermaßen beschrieben werden (Sutton & Barto, 2018, S. 63):

$$
\begin{aligned}
v_*(s) &= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}\, \pi_*[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')]
\end{aligned}
\tag{2.11}
$$

Diese Gleichung ist die sog. *Bellman Optimality Equation* und lässt sich auch als Gleichungssystem interpretieren, welches eine Gleichung pro Zustand bestitzt. Für ein Problem mit n Zuständen ergeben sich somit n Gleichung mit n Unbekannten (Sutton & Barto, 2018, S. 63). Eine Berechnung der optimalen Nutzenfunktion ist folglich in der Theorie möglich, jedoch muss die Übergangsfunktion p bekannt sein. Ist p gegeben wird von einem perfekten Modell gesprochen, eine Voraussetzung, die nicht immer erfüllt ist.

Selbst wenn die Dynamiken der Umwelt bekannt sind, kann die benötigte Rechenzeit zur Lösungen jedoch utopische Ausmaße annehmen. Bei einem Spiel wie „Backgammon“ sind die Regeln klar definiert, ein perfektes Modell ist demzufolge vorhanden, aber es existieren 10^{23} Zustände, was die mathematische Berechnung von v_* mittels der *Bellman Optimality Equation* praktisch unmöglich macht (Sutton & Barto, 2018, S. 66). Dennoch stellt sie ein wichtiges Fundament für das Reinforcement Learning dar, da die meisten Reinforcement Learning Algorithmen als annäherungsweises Lösungsverfahren verstanden werden können (Sutton & Barto, 2018, S. 66).

Methoden des Reinforcement Learnings, die die Umwelt als Blackbox betrachten, werden auch als *model-free* beschrieben. Sie benötigen keinen Zugriff auf die Übergangsfunktion p, denn es wird ausschließlich aufgrund der erhaltenen Belohnungen und Beobachtungen gelernt wird. Hierbei bezieht sich der Lernprozess darauf, wie nah die geschätzte Nutzenfunktion der aktuellen Strategie π an v_* bzw. q_* ist.

Die optimale Strategie lässt sich leicht ermitteln, wenn eine optimale Nutzenfunktion gegeben ist. Ist zum Beispiel v_* gegeben und befindet sich der Agent in Zustand s, dann muss er eine Aktion vorrausschauen, um den Folgezustand s' zu finden, der den maximalen Nutzen hat. Dieses Vorrausschen benötigt jedoch ein perfektes

Modell der Umgebung, um die Übergänge für jede Aktion zu berechnen. Das ist der ausschlaggebende Grund, warum bei *model-free* Methoden q_* berechnet wird. Denn dieser Nutzen umfasst implizit den Nutzen der Folgezustände für jede Aktion. Infolgedessen muss der Agent im Zustand s nur schauen, welche Aktion a und somit welches Zustands-Aktions-Paar den größten Nutzen hat und wählt genau jene Aktion.

2.7 Generalized Policy Iteration

Bei der Berechnung einer optimalen Strategie π_* spielt ein grundlegendes Konzept bei jeglichen Lernmethoden eine wichtige Rolle, die sog. *Generalized Policy Iteration (GPI)*. Dieses, durch Sutton & Barto (2018) geprägte, Prinzip beschreibt die Interaktion von zwei nebenläufigen Prozessen (S. 86). Ein Prozess sorgt dafür, dass die Nutzenfunktion beständig für die aktuelle Strategie wird. Er versucht das sog. Vorhersageproblem (*Prediction Problem*) zu lösen, bei dem die Nutzenfunktion v_π oder q_π für eine bestimmte Strategie geschätzt werden muss (Wiering & van Otterlo, 2012, S. 18). Jener Prozess wird als Strategieevaluation (*Policy Evaluation*) bezeichnet und unterscheidet sich je nach verwendeten Lernverfahren. *Model-based* Lernmethoden, bei denen ein perfektes Modell vorhanden ist, können den Nutzen für eine Strategie entweder direkt oder iterativ berechnen (Wiering & van Otterlo, 2012, S. 18). Hingegen benötigt die große Gruppe der *model-free* Methoden die gesammelte Erfahrung durch Interaktion mit der Umwelt. Hierbei konvergiert der geschätzte Gewinn zu dem tatsächlich erwartbaren Gewinn, solange jedes Zustands-Aktions-Paar unendlich oft besucht wird. Die Konvergenz lässt sich durch das „Gesetz der großen Zahlen“ (*Law of large numbers*) begründen (Sutton & Barto, 2018, S. 94). Ebendies sagt aus, dass die relative Häufigkeit eines Zufallsergebnisses bei zunehmender Anzahl der Ausführungen gegen die theoretische Wahrscheinlichkeit konvergiert. Für einen kompletten mathematischen Beweis siehe (Dekking et al., 2006, S. 181-189).

Das Wissen über den Nutzen der aktuellen Strategie π wird von dem zweiten Prozess genutzt, um eine verbesserte Strategie π' zu finden. Folgerichtig wird dieser Prozess als Strategieverbesserung (*Policy Improvement*) betitelt, der das sog. Kontrollproblem (*Control Problem*) zu lösen versucht (Wiering & van Otterlo, 2012, S. 18).

GPI an sich beschreibt lediglich, wie diese zwei Prozesse miteinander interagieren. Dabei konkurrieren sie auf der einen Seite, weil sie in unterschiedliche Richtungen ziehen. Eine Verbesserung der Strategie bei dem *Policy Improvement*, indem die Strategie gierig im Bezug auf die Nutzenfunktion gemacht wird, führt dazu, dass die evaluierte Nutzenfunktion für die verbesserte Strategie inkorrekt wird (Sutton & Barto, 2018, S. 86). Die erneute Evaluierung der Nutzenfunktion bei der *Policy Evaluation* sorgt indirekt dafür, dass die Strategie nicht mehr gierig ist (Sutton & Barto, 2018, S. 86).

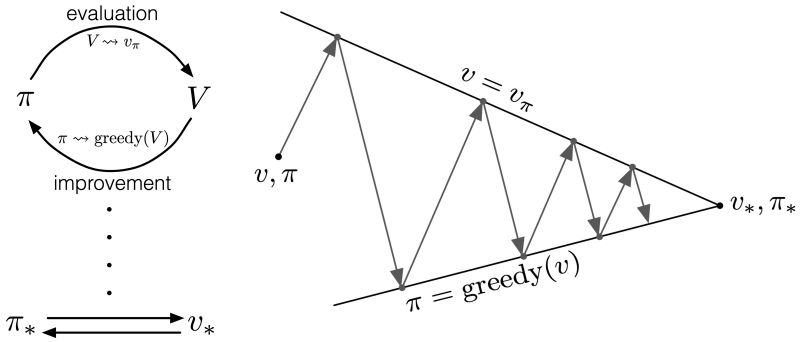

Abbildung 4: GPI nach (Sutton & Barto, 2018, S. 86f)

Auf der anderen Seite kooperieren sie jedoch in dem Sinne, dass beide Prozesse sich nur dann stabilisieren, wenn eine Strategie durch eine eigens evaluierte Nutzenfunktion gefunden worden ist, die zugleich gierig im Bezug auf genau diese ist, siehe Abb. 5. Sie haben somit ein gemeinsames Ziel; die optimale Nutzenfunktion bzw. die optimale Strategie zu finden.

2.8 Exploration-Exploitation Dilemma

Durch die Vergabe von Belohnungen und dem übergeordnetem Ziel eines Agenten so viele Belohnungen wie möglich zu sammlen, ergibt sich eine spezielle Problematik bei dem Reinforcement Learning, die bei anderen Lernmethoden des Maschinellen Lernens nicht vorhanden ist. Um den Gewinn zu maximieren, muss der Agent auf der einen Seite Aktionen bevorzugen, die sich in der Vergangenheit bereits als gut

herausgestellt haben. Er nutzt unvollständige Erfahrung, um so ausbeuterisch wie möglich zu handeln (*Exploitation*). Andererseits ist der Agent dazu gezwungen, neue Aktionen auszuprobieren, damit der Zustands- und Belohungsraum weiter erkundet wird, um bessere oder sogar optimale Entscheidungen in der Zukunft treffen zu können (*Exploration*).

Das Dilemma besteht darin, dass weder Exploration noch Exploitation ausschließlich verfolgt werden kann, ohne dabei die eigentliche Lernaufgaben zum Scheitern zu bringen. Dieses Exploration-Exploitation Dilemma wird von Mathematikern seit Jahrzehnten intensiv untersucht, bleibt allerdings ungelöst (Sutton & Barto, 2018, S. 3). Grundsätzlich muss ein Entscheidungsfinder eine Reihe von unterschiedlichen Aktionen ausführen und zunehmend jene bevorzugen, die sich als gut herausstellen. Dementsprechend muss eine Balance zwischen den beiden Prozessen gefunden werden. Eine Strategie, die ausschließlich ausbeuterisch handeln, wird auch als gierig (*greedy*) bezeichnet. Der Begriff „gierig“ bezeichnet in der Informatik eine Vorgehenweise, bei der immer die, zum Zeitpunkt der Wahl, vermeintlich beste Entscheidungen getroffen wird (Möller & Struth, 2004, S. 203). Dabei wird die Suche nach einem globalen Maximum komplett vernachläsigt. Auf den Kontext des Reinforcement Learning übertragen, wählt eine gierige Strategie für jeden Zustand immer jene Aktion, die den derzeitigen größten Nutzen besitzt. Nur wenn die Nutzenfunktion zu einer optimalen Nutzenfunktion konvergiert ist, ist eine gierige Nutzenfunktion auch gleichzeitig die optimale Strategie. Um jedoch die optimale Nutzenfunktion zu finden, muss erkundet werden.

Ein trivialer, aber denoch effektiver Ansatz ist es, die meiste Zeit gierig zu handeln, aber mit einer geringen Wahrscheinlichkeit ϵ eine zufällige Aktion auszuführen. Dabei spielen die geschätzen Nutzen der Aktionen keine Rolle und jede Aktion hat die gleiche Wahrscheinlichkeit ausgewählt zu werden. Zu vermerken ist, dass die gierige Aktion A_* ebenfalls in der Menge $\mathcal{A}(S_t)$ enthalten ist. Solche Strategien werden entsprechend als $\epsilon - greedy$ bezeichnet (Sutton & Barto, 2018, S. 28):

$$\pi(a|S_t) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{wenn } a = A_* \\ \epsilon/|\mathcal{A}(S_t)| & \text{wenn } a \neq A_* \end{cases} \tag{2.12}$$

2.9 Zusammenfasung

// TODO hier kommt die Zusammenfassung

3 Lernmethoden

3.1 Dynamische Programmierung

Die Algorithmen der Dynamischen Programmierung (*Dynamic Programming*, DP) sind im Rahmen dieser Arbeit nicht implementiert und weiter untersucht worden. Dennoch ist ein grundlegendes Verständnis für die DP von Vorteil, da elementare Bestandteile auch in den nachfolgenden Kapiteln zu den Monte-Carlo Methoden (3.2) und dem Temporal-Difference Learning (3.3) referenziert werden bzw. als Grundlage für das, in Kapitel 2.7 beschriebene, Konzept der *Generalized Policy Iteration* dienen.

Die grundlegende Idee der Dynamischen Programmierung ist die Aufteilung eines Optimierungsproblems in Teilprobleme. Dabei wird mit einem trivialem Problem gestartet und die optimale Lösung für jenes in einer Tabelle gespeichert, welches anschließend für die Lösung eines sukzessiv größer werdendes Problem verwendet wird (Mehlhorn & Sanders, 2008, S. 243).

Bezogen auf das Reinforcement Learning, ist dieses Problem die Suche nach der optimalen Strategie π_* bzw. der Lösung der *Bellman Optimality Equation*, siehe 2.11. Wie bereits in Kapitel 2.6 erwähnt, ist es möglich dieses Gleichungssystem zu lösen und somit v_* oder q_* linear zu berechnen. Mithilfe der DP erschließt sich jedoch ein iterativer Weg.

Da die Algorithmen der Dynamischen Programmierung Zugriff auf die Übergangswahrscheinlichkeiten der Umwelt sowie der Belohungsfunktion haben müssen, ist ein perfektes Modell der Umgebung Vorraussetzung. DP-Methoden sind folgerichtig immer *model-based*.

3.1.1 Strategieevaluierung

Wichtiger Bestandteil eines jeden RL Algorithmus ist die Berechnung der Zustands-Nutzenfunktion v_π (oder Aktions-Nutzenfunktion q_π) für eine willkürliche Strategie π. Die geschätzte Nutzenfunktion (unter einer bestimmten Strategie) gegen die wahren erwartbaren Werte der Gewinne streben zu lassen, wird auch als Strategieevaluierung (*Policy Evaluation*) bezeichnet (Sutton & Barto, 2018, S. 74).

Die Vorgehensweise der DP um dieses Vorhersageproblem (*Prediction Problem*) zu lösen, lässt sich folgendermaßen beschreiben. Zunächst wird eine willkürliche Nutzenfunktion v_0 gewählt, z.B. sind alle Nutzen zu Beginn mit 0 definiert. Die Bellman-Gleichung wird nun als Aktualisierungsregel angesehen, die Schritt für Schritt die geschätzten Nutzen verbessert. Es entsteht eine Reihenfolge von geschätzten Nutzenfunktionen v_0, v_1, v_2, ..., die letztendlich zu v_π konvergiert. Sutton & Barto (2018) definieren die Aktualisierungsregel wie folgt (S. 74):

$$\begin{aligned} \forall s \in \mathcal{S} : v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right] \end{aligned} \tag{3.1}$$

Mit Worten beschrieben, wird die Aktualisierungsregel in jeder Iteration auf alle Zustände $s \in \mathcal{S}$ angewendet. Dabei wird der alte Nutzen eines Zustands durch einen neuen Nutzen ersetzt, der auf dem erwarteten Nutzen aller Nachfolgezustände und der sofortigen Belohnung beruht, gewichtet nach den Übergangswahrscheinlichkeiten (Wiering & van Otterlo, 2012, S. 20). Hierbei ist festzustellen, dass die Aktualisierung des geschätzen Nutzens für einen Zustand, auf den ebenfalls geschätzen Nutzen der nachfolgenden Zustände stattfindet. DP-Methoden benutzen also das Prinzip des *bootstrapping* (Sutton & Barto, 2018, S. 89).

3.1.2 Strategieverbesserung

Ist eine suboptimale Strategie vollständig evaluiert worden, d.h. ist die Nutzenfunktion v_π präsent, dann kann diese genutzt werden, um eine bessere Strategie, π', zu finden.

Dazu wird zunächst q_π berechnet durch (Wiering & van Otterlo, 2012, S. 21):

$$q_\pi(s,a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a \right] \tag{3.2}$$

Für den Fall, dass $q_\pi(s,a)$ größer ist als v_π für ein $a \in \mathcal{A}$, dann ist es besser die Aktion a auszuführen, als jene Aktion, die durch die aktuelle Strategie π gewählt wird. Wird diese Verbesserung (*Policy Improvement*) für alle Zustände durchgeführt, dann ergibt sich die gierige (*greedy*) Strategie π', die die besten Aktionen basierend auf den Werten der aktuellen Aktions-Nutzentabelle ausführt.

3.1.3 Strategieiteration

Eine Methode, die die zwei Prozesse zum Evaluieren und der Verbesserung einer Strategie zusammenführt, ist die sog. Strategieiteration *Policy Iteration*, die ihren Ursprung in den Arbeiten von Bellman (1957) und Howard (1960) hat.

Bei der Strategieiteration wird zunächst eine willkürliche Strategie π_0 gewählt, die anschließend zu der Nutzenfunktionen v_{π_k} evaluiert wird. Ist dieser Schritt abgeschlossen, wird die Strategie gemäß der berechneten Aktions-Nutzenfunktion q_{π_k} (vgl. 3.2) verbessert, aus π_k folgt π_{k+1} und eine erneute Evaluation kann erfolgen. Die Schleife wird gestoppt, wenn für alle Zustände s gilt, dass $\pi_{k+1}(s) = \pi_k(s)$ (Wiering & van Otterlo, 2012, S. 22).

Die Strategieiteration generiert somit folgende Sequenz (Wiering & van Otterlo, 2012, S. 22):

$$\pi_0 \to v_{k_0} \to \pi_1 \to v_{k_1} \to \pi_2 \to v_{k_2} \to \pi_3 \to v_{k_3} \to \cdots \to \pi_* \tag{3.3}$$

3.1.4 Nutzeniteration

In Kapitel 3.1.1 wurde gezeigt, dass die Evaluierung einer Strategie mehrere Iteration durchlaufen muss, um letzendlich zu v_π zu konvergieren. Es ist jedoch möglich diesen Prozess vorzeitig abzubrechen, „ohne die Garantie der Konvergenz der Stratgieiteration zu verlieren“ (Sutton & Barto, 2018, S. 82).

Diese Erkenntnis macht sich die sog. Nutzeniteration (*Value Iteration*) zu nutzen, die bereits nach einer Iteration der Evaluation abbricht und den Schritt zur Verbesserung der Strategie direkt im Bezug auf diese Berechnung ausführt. Die Nutzeniteration fokusiert sich somit ausschließlich auf Schätzung der Nutzenfunktion und erzeugt im Vergleich zu der Strategieiteration folgende Sequenz (Wiering & van Otterlo, 2012, S. 23):

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \cdots \rightarrow v_* \tag{3.4}$$

3.1.5 Zusammenfassung

3.2 Monte-Carlo Methoden

Dieses Kapitel beschäftigt sich mit der Gruppe der Monte-Carlo Lernmethoden. Zunächst wird die Grundidee dieses *model-free* Ansatzes erläutert, d.h. wie die MC-Methoden das Vorhersageproblem lösen. Anschließend wird darauf eingegangen, wie mit dem Exploration-Exploitation Dilemma umgegangen wird, gefolgt von der Darstellung des Pseudocodes für den *first-visit*-Algorithmus.

Im vorrigen Kapitel wurde die Dynamische Programmierung vorgestellt. Ein Verfahren, welches ein komplettes Modell der Umgebung benötigt und somit als *model-based* bezeichnet wird. Die Hauptdisziplin des Reinforcement Learning ist jedoch die Suche nach optimalem Verhalten, wenn kein Zugriff auf die Dynamiken der Umwelt vorhanden ist (Wiering & van Otterlo, 2012, S. 27). Diese *model-free* Methoden lernen und approximieren aufgrund der Erfahrung, die sie durch die Interaktion mit der Umwelt erwerben. Dabei kann entweder mit der tatsächlichen Umwelt interagiert werden oder mit einer Simulation.

Im Bezug auf Simulationen erwähnen Sutton & Barto (2018) eine interessante Aussage. Zwar müsse ein Modell der Umwelt für eine Simulation vorhanden sein, aber sie behaupten, dass es in überraschend vielen Fällen möglich sei, Erfahrung aufgrund der erwünschten Wahrscheinlichkeitsverteilung zu erzeugen ohne dabei die komplette Wahrscheinlichkeitsverteilung für alle möglichen Übergänge zu kennen wie sie z.B. bei

der DP benötigt wird (S. 91).

// TODO: Lindemann; Beispiel? Würfel, BlackJack? Anstatt die Wahrscheinlichkeiten immer wieder komplett auszurechnen, kann man einfach einen Kartenstapel simulieren, von dem Karten entfernt werden. Meinen die beiden das?

3.2.1 Vorhersageproblem

Monte-Carlo Methoden lösen das Reinforcement Learning Problem über Durschnittsbildung der, durch Erfahrung gesammelten, Gewinne (Sutton & Barto, 2018, S. 91). Dabei werden die Nutzen und die Strategien ausschließlich nach einer abgeschlossenen Episode aktualisiert. Folgerichtig ist das Aktualisierungsverhalten *episode-by-episode* und nicht *step-by-step* (als nach jeder Aktion)(Sutton & Barto, 2018, S. 91). Das ist zugleich der Hauptunterschied zu den in Kapitel 4 vorgestelltem Temporal-Difference Learning, welches nach jeder Aktion die geschätzen Nutzen anpasst. Daraus folgt auch, dass Monte-Carlo Methoden ausschließlich auf episodiale Probleme anwendbar sind, TD Learning jedoch zusätzlich auch bei kontinuierlichen Aufgaben zum Einsatz kommen kann.

Zur Erinnerung, eine Nutzenfunktion gibt den Nutzen eines Zustands an, also den geschätzten erwartbaren Gewinn. Dabei ist der erwartete Gewinn eines Zustands die erwartbare Summe aller zukünftig, diskontierten Belohnungen, wenn von diesem Zustand aus gestartet wird. Um den erwarteten Gewinn zu schätzen, kann der Durschnitt über die, durch Erfahrung gesammelten, realen Gewinne gebildet werden.

Der Grundansatz ist dabei wie folgt. Eine Episode unter der Strategie π wird z.B. durch eine Simulation erzeugt. Es entsteht eine Reihenfolge von Triple (s, a, r). Kommt ein Zustand s innerhalb der Episode vor, wird auch von einem Besuch (*visit*) von s gesprochen (im Rahmen der Monte-Carlo Methoden). Der Gewinn für den Zustand s ist somit die Summe aller Belohnung nach dem ersten Besuch des Zustands s. Über alle, auf diese Weise gesammelten, Gewinne von s wird der Durschnitt berechnet, der -mit steigender Anzahl an Besuchen- gegen den tatsächlichen Nutzen von s strebt. Wird der Gewinn vom Start des ersten Besuchs von s berechnet, dann wird diese Methode auch als *First-Visit* bezeichnet. Konsequenterweise bezeichnet *Every-Visit* die Methode,

bei der für jegliche Besuche von s in einer Episode der Gewinn berechnet wird. Diese beiden Methoden sind sehr ähnlich und unterscheiden sich z.B. im Pseudocode nur durch eine Abfrage. Trotzdem haben sie unterschiedliche theoretische Eigenschaften (Sutton & Barto, 2018, S. 92). Diese sind jedoch im Rahmen dieser Arbeit nicht weiter dargestellt, da ausschließlich mit der *First-Visit* Variante garbeitet wird. //TODO für mehr Infos

In Kapitel 2.6 ist erwähnt worden, dass *model-free* Lernmethoden die Aktions-Nutzenfunktion berechnen. Grund hierfür ist, dass sie nicht in der Lage sind einen Schritt vorherzusehen, weil die Übergangsfunktion p nicht gegeben ist. Monte-Carlo Methoden können sowohl, wie im vorrigen Absatz erläutert, den Nutzen von Zuständen berechnen, als auch den Nutzen von Zustands-Aktions-Paaren. Der Unterschied besteht darin, dass nicht der Besuch von s entscheidend ist, sondern der Besuch des Paares (s, a).

3.2.2 Exploration

Da die Monte-Carlo Methoden mittels Durchschnittsbildung arbeiten, ist es unabdingbar, dass jeglichen Zustände respektive Zustands-Aktions-Paare ausreichend oft besucht werden. Wenn jedoch eine deterministische Strategie π gegeben ist, dann wird immer nur eine Aktion pro Zustand ausgeführt, nämlich jene mit dem derzeitigen höchsten, geschätzten Nutzen. Dies sorgt dafür, dass der Zustands- und Aktionsraum nicht ausreichend erkundet wird und der Algorithm sozusagen in einem lokalem Maximum festhängt.

Sutton & Barto (2018) stellen in ihrem Werk drei Ansatze vor, die die fortlaufende Exploration ermöglichen. Eine Möglichkeit ist die Verwendung einer ϵ-*greedy* Strategie, wie sie auch im Kapitel 2.8 vorgestellt wurde. Mit einer bestimmten Wahrscheinlichkeit ϵ wird nicht die vermeintlich beste Aktion gewählt (aktuell größter Aktions-Nutzen), sondern eine zufällige Aktion $a \in \mathcal{A}$. Diese Methodik erlaubt die fortlaufende Exploration und garantiert trotzdem eine Konvergenz zu einer optimalen Strategie, wenn ϵ im Laufe der Zeit verringert wird (Sutton & Barto, 2018, S. 201). Welche Auswirkungen die Werte von ϵ auf das Konvergenzverhalten haben und wie in eine „Verringerung im

Laufe der Zeit “ aussehen kann, wird im Rahmen der praktischen Umsetzung anhand des JumpingDino Beispiels in Kapitel 4.2 untersucht.

Um einen einen Überblick über weitere Vorgehensweisen zu der fortlaufenden Erkunden zu geben, werden auch die zwei weiteren Methoden nach Sutton & Barto (2018) kurz dargestellt (S.96-108).

Anstatt die Strategie so zu verändern, dass sie suboptimale Aktionen wählt, um alle Zustände oder Zustands-Aktions-Paare ausreichend oft zu besuchen, kann auch explizit mit einem bestimmten Zustand respektive Zustands-Aktions-Paar gestartet werden. Dabei muss jeder Zustand oder jedes Zustands-Aktions-Paar eine Wahrscheinlichkeit größer 0 haben, um als Start einer Episode ausgewählt zu werden. Dieser Ansatz wird auch als *Exloring Starts* bezeichnet.

Eine weitere Möglichkeit ist das sog. *off-policy learning*. Die Grundidee hierbei besteht darin, nicht eine Strategie zu benutzen, die teilweise exploriert (ϵ-*greedy*), sondern zwei Strategien zu verwenden. Eine konvergiert zu der optimalen Strategie und die andere exploriert den Zustands- und Aktionsraum, sammelt somit die Erfahrung. Die Strategie, die stetig verbessert wird, wird als Zielstrategie (*target policy*) bezeichnet wohingegen die Strategie, die die Episoden erzeugt als Verhaltensstrategie (*behavior policy*) bezeichnet wird (Sutton & Barto, 2018, S. 103). Das „off “ in *off-policy learning* bezieht sich darauf, dass die Erfahrung einer anderen, von der Zielstrategie abweichenden, Strategie dazu genutzt wird, um zu lernen.

3.2.3 Pseudocode

Der nachfolgende Pseudocode (Sutton & Barto, 2018, S. 101) zeigt die Vorgehensweise der Monte-Carlo Methoden zur Bestimmung der optimalen Strategie π_* auf Basis der Aktions-Nutzenfunktion q bzw. Q. Genauer wird die *First-Visit* Variante vorgestellt, die mithilfe einer ϵ-greedy Strategie, die die fortlaufende Erkunden garantiert. Wie in Kapitel 2.4 erläutert, sollte der Diskontierungsfaktor γ für episodiale Probleme den Wert 1 annehmen, um jegliches Handeln während einer Episode bei der Berechnung des Gewinns zu berücksichtigen.

Algorithm 1 On-policy first-visit MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$

1: Algorithm parameter: small $\epsilon > 0$
2: Initialize:
3: $\pi \leftarrow$ an arbitary ϵ-soft policy
4: $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
5: $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
6: Repeat forever (for each episode):
7: Generate an episode following $\pi : S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
8: $G \leftarrow 0$
9: Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
10: $G \leftarrow \gamma G + R_{t+1}$
11: Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$:
12: Append G to $Returns(S_t, A_t)$
13: $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
14: $A^* \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken arbitrarily)
15: For all $a \in \mathcal{A}(S_t)$:
16: $$\pi(a|S_t) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Eine Umformung zur *Every-Visit* Variante kann durch das Entfernen der Bedingung in Zeile 11 realisiert werden.

Dieser Ansatz der Monte-Carlo Methode folgt dem Schema der *Generalized Policy Iteration*, vgl. Kapitel 2.7. Der Prozess der Strategieevaluation findet nach jeder Episode statt und benötigt im Vergleich zu der Dynamischen Programmierung kein perfektes Modell. Da die Evaluation für jeden Zustand bzw. jedes Zustands-Aktions-Paar nach

nur einem Schritt (der Durschnittsermittlung des Gewinns) gestoppt wird und danach der Prozess der Strategieverbesserung (*Policy Improvement*) startet, errinert dieses Vorgehen an die Nutzeniteration aus der Dynamischen Programmierung, siehe Kapitel 3.1.4. Berechnete Werte für den Aktions-Nutzen jedes Zustands dienen als Grundlage für den Prozess der Strategieverbesserung, dem Verbessern der Strategie, im Fall der Monte-Carlo Methoden, nach jeder Episode.

3.2.4 BlackJack Beispiel*

3.2.5 Zusammenfassung

Monte-Carlo Methoden lernen Nutzenfunktionen durch die direkte Interaktion mit der Umgebung. Damit zählen sie zu den *model-free* Lernmethoden, die kein perfektes Modell der Umgebung benötigen. Zur Ermittlung des erwarteten Gewinns für ein Zustands-Aktions-Paars wird der Durschnitt über jegliche erhaltene Gewinne pro Episode gebildet. Somit findet die Evaluation und Verbesserung einer Strategie immer nur nach dem Abschluss einer Episode statt. Dies ist zugleich der Grund, warum Monte-Carlo Methoden ausschließlich auf episodiale Probleme anwendbar sind.

Da Aktionen auf Basis der temporär besten Aktions-Nutzen gewählt werden, ist eine ausreichende Exploration nicht gegeben, weil Gewinne vermeintlich suboptimaler Zustands-Aktions-Paare nicht gesammelt werden. Der Algorithmus verharrt in einem lokalem Maximum. Um dies zu verhindern, kann eine ϵ-greedy Strategie verwendet werden, die mit einer Wahrscheinlichkeit von ϵ eine zufällige Aktion ausführt.

Im Vergleich zu der Dynamischen Programmierung benötigen die MC Methoden kein perfektes Modell der Umgebung und können auf Basis von Simulationen lernen. Zugleich aktualiseren sie ihre geschätzten Nutzen nicht auf Basis von anderen geschätzten Nutzen, sie betreiben somit kein *bootstrapping*.

Im nächsten Kapitel werden Lernmethoden vorgestellt, die wie die MC Methoden kein perfektes Modell benötigt, aber wie die DP *bootstrappen* und somit in der Lage sind, nach jedem Zeitstempel ihre geschätzten Nutzen zu aktualiseren.

3.3 Temporal Difference Learning

Nachdem in den beiden vorrigen Kapitel die Methoden der Dynamischen Programmierung und des Monte-Carlo Ansatzes beleuchtet wurden, befasst sich dieses Kapitel mit der dritten großen Gruppe an Algorithmen, die das Reinforcement Learning Problem lösen, dem *Temporal-Difference Learning* (TD). Wie zuvor wird zunächst erläutert, wie diese Art der Algorithmen das Vorhersageproblem lösen. Anschließend werden zwei vollständige Algorithmen vorgestellt, die das Kontrollproblem, also die Suche nach einer optimalen Strategie, bewältigen. Zugleich werden Unterschiede und Parallelen der drei Lerngruppen aufgezeigt.

Um einschätzen zu können, welche zentrale Rolle das TD in dem Bereich des Reinforcement Learnings eingenommen hat, folgt ein Zitat von Sutton & Barto (2018):

> If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-dierence (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. (Sutton & Barto, 2018, S. 119)

Die Verbindung besteht zum einen daraus, dass das TD genau wie die Monte-Carlo Methoden direkt durch die Interaktion mit der Umwelt lernt, folgerichtig auch *model-free* sind. Zum anderen aktualisieren die TD Methoden, genauso wie bei der Dynamische Programmierung, ihre geschätzen Nutzen mit Hilfe weiterer geschätzen Nutzen, sie bedienen sich also ebenfalls dem Konzept des *bootstrapping* (Sutton & Barto, 2018, S. 119). Dadurch ist das TD in der Lage, seine Nutzentabelle nach jeder Aktion zu aktualisieren, *step-by-step*. Ein Warten auf das Ende einer Episode, wie bei den MC-Methoden, ist nicht notwendig. TD kann somit zusätzlich bei kontinuierlichen Problem zum Einsatz kommen (Sutton & Barto, 2018, S. 124).

3.3.1 Vorhersageproblem

MC- und TD-Methoden lösen das Vorhersageproblem beide durch gesammelte Erfahrung durch die direkte Interaktion mit der Umwelt. Sie folgen einer Strategie π und aktualisieren ihre geschätzen Nutzen V (oder Q) für v_π (respektive q_π) auf Grundlage

der erhaltenen (s, a, r) Triple. Doch wie schafft es das *Temporal-Differene Learning* im Gegensatz zu den Monte Carlo Methoden nach jedem Schritt zu aktualisieren und nicht auf das Ende einer Episode zu warten, somit nicht den Gewinn G_t zu benötigen?

Um diese Frage zu beantworten, wird zunächst ein neuer Parameter vorgestellt, die Schrittgröße α (*step-size parameter*). Dieser Parameter beeinflusst die Lernrate und sorgt konkret dafür, wie stark die Veränderung eines neu geschätzen Nutzens gewichtet wird. Des Weiteren wird der Begriff „Ziel“ (*target*), im Umfeld von TD auch TD-Ziel (*TD-target*), verwendet. Dieses Ziel sagt aus, zu welchem Wert die derzeitige Aktualiserung des Nutzen streben soll.

Monte-Carlo Methoden müssen bis zu dem Ende einer Episode warten, da erst dann der Gewinn G_t feststeht, der als Ziel für $V(S_t)$ benötigt wird (Sutton & Barto, 2018, S. 119). Eine vereinfachte formale Darstellung der Aktualisierungsregel für die *Ever-Visit* MC-Methode sieht wie folgt aus (Sutton & Barto, 2018, S. 119):

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t)\right] \tag{3.5}$$

Im Gegensatz dazu, müssen die TD-Methoden lediglich bis zu dem nächsten Zeitstempel warten, um eine Aktualiserung vorzunehmen. Dazu wird zum Zeitpunkt $t + 1$ sofort ein Ziel gebildet, welches aus der Belohnung R_{t+1} und dem geschätzen Nutzen $V(S_{t+1})$ zusammengesetzt ist. Die Aktualisierungsregel für die einfachste Form des TD lautet somit (Sutton & Barto, 2018, S. 120):

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{3.6}$$

TODO: TD Error weglassen, weil irrelevant für Bearbeitung oder näher erläutern? Richtwert für die Backpropagation bei Deep RL z.B.

Dabei wird der Teil in der eckigen Klammer auch als TD-Fehler (*TD-error*, δ) bezeichnet, der die Differenz zwischen dem geschätzen Wert S_t und dem besseren Schätzwert $R_{t+1} + \gamma V(S_{t+1})$ angibt (Sutton & Barto, 2018, S. 121):

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{3.7}$$

Statt dem Ziel G_t der MC-Methoden, ist das Ziel des TD-Learnings $R_{t+1} + \gamma V(S_{t+1})$. Da der Wert für $V(S_{t+1})$ ein geschätzer Wert ist, aber dennoch für die Aktualiserung verwendet wird, *bootstrappt* das TD-Learning. Dies ist notwendig, um nicht den realen Gewinn nach Abschluss einer Epsiode verwenden zu müssen, sondern diesen gewissermaßen aufspalten zu können und nur auf Basis der aktuellen Belohnung eine Anpassung vorzunehmen. Diese Aufspaltung basiert auf der fundamentalen Erkenntnis, dass Gewinne aufeinanderfolgender Zeitstempel in Verbindung stehen (siehe Kapitel 2.4) und somit gilt (Sutton & Barto, 2018, S. 120):

$$\begin{aligned} v_\pi &= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right] \end{aligned} \tag{3.8}$$

Anhand von 3.8 lässt sich der Zusammenhang der drei großen Gruppen von Lernmethoden sehr gut zusammenfassen. Die erste Zeile beschreibt den geschätzen Wert, den die Monte-Carlo Methoden als Ziel verwenden. Es handelt sich um einen Schätzwert, da der Erwartungswert unbekannt ist und stattdessen mit dem Durschnitt gesammelter Gewinne gerechnet wird.

Die Dynamische Programmierung benutzt den Schätzwert, der sich aus der dritten Zeile ergibt. Dabei bezieht sich das Schätzen nicht auf die eigentlichen Erwartungswerte, denn diese können berechnet werden, da ein perfektes Modell der Umgebung mit allen Übergangswahrscheinlichkeiten vorhanden ist. Ausschlaggebend ist, dass $v_\pi(S_{t+1})$ zum Zeitpunkt t nicht berechnet, sondern von dem derzeitige geschätzte Nutzen V_{t+1} Gebrauch gemacht wird (Sutton & Barto, 2018, S. 120).

Das TD-target ist eine Schätzung aufgrund beider Gründe, die in den zwei vorrigen Absätzen erläutert wurden. Es basiert auf der Sammlung von Erfahrung, um den Erwartungswert bzw. die Werte in der dritten Zeile von 3.8 schätzen zu können und gleichzeitig wird der derzeitg geschätze Nutzen V anstelle des wahren Wertes von v_π verwendet (Sutton & Barto, 2018, S. 120f).

Sutton & Barto (2018) kommen zu dem Schluss, dass das TD-Learning eine Vereinigung darstellt zwischen der Probenahme (*sampling*) der MC-Methoden und dem

bootstrapping der DP. Dabei führen die beiden Autoren weiter aus, dass es mit „Vorsicht und Vorstellungskraft “ möglich sei, die Vorteile beider Methoden (MC und DP) durch den Einsatz des TD-Learnings zu nutzen (S. 120f).

Dass das TD-Verfahren an sich, also die Verwendung der in 3.6 dargestellten Aktualisierungsregel, konvergiert ist durch die Arbeiten von Sutton (1988) und C. J. Watkins & Dayan (1992) bewiesen worden. Weitere Untersuchungen zu dem Konvergenzverhalten der tabularen TD-Methoden wurden zudem von Jaakkola et al. (1994) und Tsitsiklis (1994) durchgeführt.

3.3.2 SARSA

Nachdem die grundsätzliche Idee des *Temporal-Difference* Learnings im vorrigen Unterkapitel erläutert wurde, folgt nun der erste vollständige Algorithmus, um eine optimale Strategie π_* zu finden.

Wie schon bei den Algorithmen der DP und der MC-Methoden, wir das Kontrollproblem auch hier durch dem allgemeinen Leitbild der *Generalized Golicy Iteration*(GPI), siehe Kapitel 2.7, Folge geleistet. Das heißt, es werden Aktions-Nutzen geschätzt, $q_\pi(s, a)$, die dann als Grundlage für die Verbesserung der aktuellen (ϵ-*greedy*) Strategie benutzt werden. Entscheidend ist, dass nun eine Form der TD Aktualiserungsregel (3.6) für die Strategieevaluierung Verwendung findet.

In 3.6 wurde der Übergang von einem Zustand zu dem nächsten Zustand betrachtet, um den Zustands-Nutzen V zu ermitteln. Zur Berechnung von Q wird indes der Übergang von Zustands-Aktions-Paar zu Zustands-Aktions-Paar betrachtet, einschließlich der Belohnung, die bei diesem Übergang vergeben wird. Es ergibt sich das Quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, der diesem Algorithm seinem Namen gegeben hat. Die angepasste Aktualiserungsregel sieht somit wie folgt aus (Sutton & Barto, 2018, S. 129):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{3.9}$$

Eingebetten in einen gesamten Algorithmus, ergibt sich nach Sutton & Barto (2018) folgender Pseudocode (S.130):

Algorithm 2 Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameter: step size $\alpha \in (0, 1]$), small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+, a \in \mathcal{A}(s)$, arbitrarily except that
$Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'; A \leftarrow A';$
    until S is terminal

SARSA ist ein sowohl ein *on-policy* als auch ein *online* Algorithmus (Sutton & Barto, 2018, S. 129f). *On-policy*, weil die Strategie, die exploriert und die Strategie, die verbessert wird, dieselbe ist. Der Begriff *online* trifft für alle TD-Methoden zu, da sie eine Aktualiserung nach jedem Zeitstempel, also *step-by-step*, durchführen. In einigen Fallen kann dies ein entscheidener Vorteil gegenüber *offline* Methoden, wie denen der Monte-Carlo Familie, sein. Wenn bei einer gierigen Strategie eine Aktion den derzeitigen besten Nutzen hat, die dafür sorgt, dass der Agent in einem Zustand festhängt, dann kann die laufende Episode nicht abgeschlossen werden. *Online* Algorithm wie *SARSA* lernen hingegen während der Episode, dass eine solche Strategie suboptimal ist und wechseln zu einer anderen Strategie (Sutton & Barto, 2018, S. 130).

3.3.3 Q-Learning

Eines der frühesten Durchbrüche des Reinforcement Learnings war die Entwicklung des sog. *Q-Learnings*. Dieser Algorithmus wurde von C. J. C. H. Watkins (1989) vorgestellt und ist der am weitesten verbreitete RL-Algorithmus. Auf der Seite *arxiv.org*, einem Dokumentenserver für Papers aus dem naturwissenschaftlichen Bereich, liefert die Suche nach den Schlagworten „Reinforcement Learning“ in Verbindung mit „Q-Learning “ 774 Treffer, „SARSA “ hingegen nur 47 und „Monte-Carlo“ 135. Auch moderne Methoden des Deep-RLs basieren auf der Grundlage des *Q-Learnings*, z.B. das *deep Q-network* (DQN) Mnih et al. (2013), welches von *Google DeepMind* entwickelt wurde.

Der markanteste Unterschied des *Q-Learning* Algorithmus im Vergleich zu *SARSA* ist das Lernen *off-policy*. Im Zusammenhang möglicher Varianten zur fortlaufenden Exploration bei Monte-Carlo Methoden wurde dieses Prinzip im Abschnitt 3.2.2 bereits erörtert. Es geht dabei um die Eigenschaft, dass zwei unterschiedliche Strategien verwendet werden. Eine interagiert mit der Umwelt und die andere wird schrittweise verbessert bis sie schließlich zu der optimalen Strategie konvergiert ist.

Die erkundende Strategie kann wie bei *SARSA* und den MC-Methoden eine ϵ-*greedy* Strategie sein. Entscheidend ist, dass bei der Aktualisierung der Aktions-Nutzen, Q, direkt q_* approximiert wird, indem immer die aktuell beste Aktion des Folgezustands für die Berechnung des geschätzen Gewinns verwendet wird (Sutton & Barto, 2018, S.131):

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)] \tag{3.10}$$

Die Wahl der besten Aktion in $\gamma \max_a Q(S', a)$ sorgt dafür, dass die Strategie, die die Nutzentabelle aktualisiert, gierig (*greedy*) ist. Da die explorierende Strategie aber ϵ-*greedy* ist, unterscheiden sich diese zwei Strategien voneinander, wodurch sich konkret das *off-policy learning* erklären lässt. Ein kompletter Pseudocode für das *Q-Learning* kann folgerdermaßen dargestellt werden (Sutton & Barto, 2018, S. 131):

Algorithm 3 Q-Learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameter: step size $\alpha \in (0, 1]$), small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+, a \in \mathcal{A}(s)$, arbitrarily except that
$Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$;
    until S is terminal

3.3.4 Zusammenfassung

Das *Temporal-Difference Learning* stellt eine Verbindung zwischen der Dynamischen Programmierung und den Monte-Carlo Methoden dar. Es lernt durch die Interaktion mit der Umwelt, ist also *model-free* wie die MC-Methoden, bedient sich aber zeitgleich dem Konzept des *bootstrapping* wie es auch bei der DP zum Einsatz kommt. Dadurch ist das TD-Learning in der Lage, ohne ein perfektes Modell trotzdem nach jeder Aktion zu aktualisieren und braucht nicht auf das Ende einer Episode zu warten.

Die Suche nach der optimalen Strategie (Kontrollproblem) verläuft wie schon bei den MC-Methoden nach dem Prinzip der *Generalized Policy Iteration*. Konkret wurden zwei Algorithm zur Bestimmung der optimalen Strategie vorgestellt. Der *SARSA* Algorithmus lernt *on-policy* wohingegen das weit verbreitete *Q-Learning* eine *off-policy* Variante darstellt.

4 Praktischer Teil

4.1 Implementierung

4.1.1 Anforderungen

Zu Beginn und während der Umsetzung kristallisierten sich Anforderungen heraus, die zunächst aufgezählt und anschließend kurz erläutert werden:

- Gut gewählte Interfaces, die die Theorie widerspiegeln
- Determinismus; Wiederholbare Ergebnisse
- Visualisierung; GUI
- Erweiterbarkeit
- Sammlung von Statistiken
- Lernprozess speichern

Gut gewählte Interfaces. Die Interfaces sind so geschnitten, dass sie die grundlegenden Bestandteile des Reinforcement Learnings widerspiegeln. Dabei richtet sich die Terminologie an das Agent-Umwelt Interface, welches in Kapitel 2.1 vorgestellt ist.

Determinismus. Um zu gewährleisten, dass gesammelte Ergebnisse zum Verhalten von unterschiedlichen Algorithmen vergleichbar sind, muss die gesammte Implementierung determistisch sein und wiederholbare Ergebnisse liefern. Eines der wichtigsten Faktoren hierbei ist die Handhabung des *Random Number Generators*, der vor allem dafür benötigt wird, um „willkürliche“ Aktionen bei ϵ-*greedy* Strategien zu wählen. Gearbeitet wird ausschließlich mit der *RNG.java* Klasse, die mit Hilfe eines *static*-Kontruktors einmalig ein *Random*-Objekt anlegt und *seeded*.

Eine wichtige Erkenntnis ist außerdem, dass die HashMap in Java nicht deterministisch agiert im Bezug auf die Reihenfolge der Elemente. Die Reihenfolge ist jedoch entscheidend, da u.a. eine HashMap die Aktionen auf ihre Nutzen abbildet und das *KeySet* dieser Map dazu benutzt wird, um eine willkürliche Aktion zu wählen. In der Dokumentation zu der *HashMap*-Klasse heißt es: „This class makes no guarantees

as to the order of the map; in particular, it does not guarantee that the order will remain constant over time “ []Oracle (2020a). Um dennoch eine feste Reihenfolge zu garantieren, wird ausschließlich die *LinkedHashMap* als Implementationen des *Map*-Interfaces benutzt. Diese sichert eine konsistente, vorhersagebare Ordnung zu (Oracle, 2020b).

Die umgesetzten Algorithmen sind alle iterativ und somit *single-threaded*. Dennoch werden weitere nebenläufige Threads eingesetzt, um z.B. Laufzeitstatistiken zu sammeln. Auch das UI läuft in einem seperaten Thread. Es muss somit darauf geachtet werden, nur geeignete Aufgaben in andere Threads auszulagern, die den eigentlichen Lernprozess im Main-Thread nicht beeinflussen.

Visualisierung. Wenn über tausende Episoden gelernt wird, Millionen von Belohnungen verteilt und Aktionen ausgeführt werden, dann reicht eine simple Konsolenausgabe nicht mehr aus, um das Verhalten eines Algorithmus einzuschätzen. Eine Graphische Nutzungsoberfläche (*GUI*) ist erstellt worden, mit der Parameter während des Lernens gesteuert werden können. Außerdem wird die Umwelt visualisiert, die Nutzentabelle kann angezeigt werden und ein kontinuierlicher Graph zeigt die erhaltenen Belohnungen an.

Erweiterbarkeit. Der Aufbau der Implementierung erlaubt ein einfaches Hinzufügen von weiteren Lernszenarien. Hierzu muss lediglich ein *Enum*, welches den Aktionsraum repräsentiert und eine Klasse, die das *Environment*-Interface implementiert, anlegt werden. Ebenfalls können weitere RL Algorithmen ergänzt werden, indem von der abstrakten Klasse *Learning* bzw. *EpisodicLearning* abgeleitet wird. Letztendlich ergibt sich eine Art RL-Framework, welches unterschiedliche Umgebungen und Algorithmen bereitstellt.

Sammlung von Statistiken. Um z.B. Aussagen über das Konvergenzverhalten von unterschiedlichen Lernmethoden treffen zu können, ist es notwendig, Daten zu sammeln und auszuwerten. Das umgesetzte *Listener*-Pattern, bei dem die Algorithmen u.a. nach jedem Zeitstempel oder nach jeder kompletten Episode ein Event mit Informationen auslösen, erlaubt eine generische Datenerhebung für unterschiedliche Lernmethoden und Problemstellungen.

Lernprozess speichern. Alle Methoden des Reinforcement Learnings, die im Rahmen dieser Arbeit implementiert sind, laufen *single-threaded* und benötigen bei großen Zustands- und Aktionsräumen (100.000+ Zustände) lange Laufzeiten. Daher ist eine Speichern- und Laden-Funktion umgesetzt, die die aktuelle Nutzentabelle serialisieren und deserialisieren kann, um einen Lernvorgang zu einem späteren Zeitpunkt fortzusetzen.

4.1.2 Interfaces

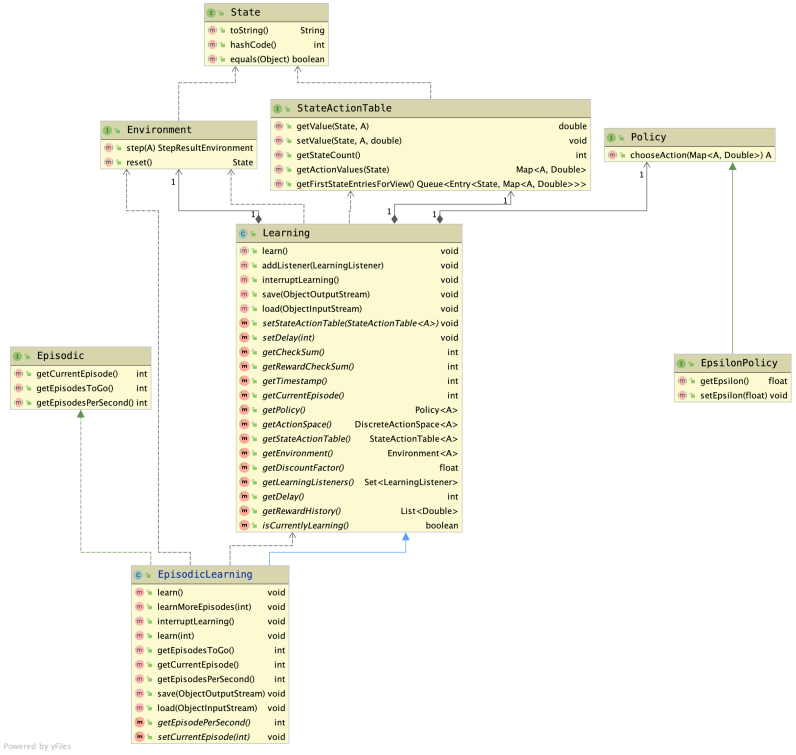

Abbildung 5: Darstellung der wichtigsten Interfaces

4.2 Jumping Dino

In diesem Kapitel wird das erste Problem- bzw. Lernszenario vorgestellt. Ursprünglich war geplant, dem Titel dieser Arbeit folge zu leisten und ausschließlich das Ameisen-

Agentenspiel („AntGame“) zu behandelt. Ursprünglich ist dieses Beispiel mit dem Namen „Jumping Dino“ entstanden, um die implementierten Algorithmen des RL bei einem scheinbar trivialeren Problem nachzuvollziehen.

Es stellte sich jedoch heraus, dass dieses episodiale Problem sehr gut geeignet ist, um das Konvergenzverhalten bei der Suche nach einer optimalen Strategie unter verschiedenen Bedingungen zu untersuchen. Vor allem eine Einschätzung darüber, welche Algorithmen (Monte-Carlo Methoden, *SARSA*, *Q-Learning*) für welche Problemszenarien praktikabel sind oder nicht, kann durch diese Untersuchung erlangt werden.

Im nachfolgenden Unterkapitel wird zunächst die Problemstellung erläutert und die möglichen Modellierungen der Umwelt vorgestellt. Anschließend werden die Zustands- und Aktionsräume für die einzelnen Szenarien definiert, die dann als Grundlage für die Ergebnisse dienen, die in Kapitel 5 aufgeführt werden. Abgeschlossen wird dieses Kapitel mit der Modellierung einer passenden Belohnungsfunktion.

4.2.1 Problemstellung

Der Name „Jumping Dino“ ist gewählt worden, weil das Beispiel an das bekannte Minigame „T-Rex Runner“ von Google angelehnt ist, welches immer im Google Chrome Browser erscheint, wenn keine Internetverbindung vorhanden ist. Zusammengefast geht es darum, dass ein Dino im richtigen Moment über Hindernisse springen muss, die fortlaufend, von dem rechten Bildschirmrand aus, auf ihn zukommen.

Die gesamte Spielwelt ist 800px breit, wobei der Dino stehts 50px vom linken Bildrand entfernt ist. Hindernisse und der Dino selbst sind als Quadrate definiert, mit der Seitenlänge 60px respektive 50px. Die maximale Höhe eines Sprungs, also der Abstand zwischen dem Boden und der unteren Kante des Dinos, beträgt 150px. Jeden Tick werden die Positionen der Akteure um einen gewisse Pixelanzahl angepasst, welches zugleich als Geschwindigkeit angesehen werden kann. Der Dino hüpft in allen Szenarien um 20px pro Tick, die Geschwindigkeiten der Hindernisse kann jedoch je nach Szenario variieren.

Auf eine aufwendige Visualisierung des Spielgeschehens wurde verzichtet, der Dino

wird lediglich als grünes Quadrat dargestellt, die Hindernisse als schwarzes Quadrat. Es ergibt sich folgende Umwelt:

Abbildung 6: Jumping Dino Umgebung

Bei jedem Tick kann der Agent zwischen zwei Aktionen wählen, JUMP und NOTHING. Befindet sich der Agent bereits im Sprung, so ist es dennoch möglich die Aktion JUMP auszuwählen, sie hat jedoch keinen Effekt. Eine Episode endet, wenn eine Kollision zwischen dem Dino und dem Hindernis registriert wird.

Für die Untersuchungen wird zwischen zwei Szenarien unterschieden:

Simple. Bei dieser Variante wandern die Hindernisse immer mit der gleichen Geschwindigkeit (30px pro Tick) nach links. Außerdem erscheinen sie immer im gleichen Abstand, d.h. erreicht die rechte Kante des Hindernisses den linken Bildschirmrand, so wird sofort ein neues Hindernis gespwant an der Position $x = 860$. Es ergeben sich 31 mögliche Positionen, in der sich ein Hindernis in der Umwelt befinden kann $(x = 860, x = 830, \ldots, x = -10, x = -40)$.

Advanced. Um die Umwelt anspruchsvoller zu gestalten, werden in diesem Szenario ein paar Änderungen vorgenommen. Statt einer festen Geschwindigkeit, bewegen sich die Hindernisse nun mit vier unterschiedlichen Geschwindigkeiten. Dies sorgt dafür, dass die Geschwindigkeit nun auch ein Faktor ist, der für einen Sprung des Dinos entscheidend ist. Bei sehr schnellen Hindernissen muss der Dino frühzeitig springen, um z.B. bei zwei aufeinanderfolgenden schnellen Hindernissen überhaupt in der Lage zu sein, das zweite Hindernis zu überspringen. Hingegeben muss er lernen, bei sehr langsamen Hindernissen erst sehr spät zu springen, um bei der Landung nicht zu kollidieren. Mit einer gleichen Wahrscheinlichkeit kann die Geschwindigkeit den Wert 10px, 21px, 48px oder 105px annehmen.

Eine weitere Anpassung bei dem *Jumping Dino Advanced* ist, dass die Hindernisse

nicht immer mit dem gleichen Abstand spawnen, sondern ebenfalls zufällig mit vier unterschiedlichen Werten ($x = 1630, x = 1694, x = 1718, x = 1814$). Insgesamt ergeben sich 827 mögliche Positionen, die ein Hindernis einnehmen kann.

4.2.2 Zustandsmodellierung

Für eine korrekt gewählt Zustandsmodellierung, die genug aber nicht redundante Informationen beinhaltet, um die optimalen Entscheidung zu treffen, ist zunächst die Problemstellung genauer zu betrachten.

In dem *Jumping Dino Simple* Szenario muss der Agent lernen, im richtigen Moment die Aktion JUMPING durchzuführen. Anders ausgedrückt, kann der Agent in der Theorie immer die Aktion NOTHING wählen, muss aber bei einem bestimmten Abstand zu dem Hindernis die Aktion JUMP ausführen. Im Grunde ist dies eine Schwellwertsuche, bei der vorerst angenommen wird, dass alleine die Distanz zu dem Hindernis ausreichend ist, um das Problem zu lösen.

Den Zustand nur anhand der Distanz zu verwirklichen ist bei der *Advanced* Variante nicht ausreichend, um optimale Entscheidung zu treffen. Schnelle Hindernisse müssen frühzeitiger übersprungen werden, langsame hingehen erst kurz vor einer Kollision. Eine zweite Information muss somit gegeben sein, die Geschwindigkeit der Hindernisse.

Im späteren Verlauf der Untersuchungen zu dem Konvergenzverhalten, die in Kapitel 5 näher erläutert werden, stellt sich heraus, dass der Zustand um eine boolische Flag erweitert werden muss, die aussagt, ob der Dino sich im Sprung befindet oder nicht. Es ergeben sich insgesamt drei unterschiedliche Zustandsmodellierungen, die auf folgenden Variablen beruhen:

- *dist*: *Integer*, Abstand zwischen rechter Kante des Dinos und linker Kante des Hindernisses
- *inJump*: *Boolean*, Boolischer Wert, ob sich der Dino in einem Sprung befindet und somit die Aktion *JUMP* keine Auswirkung hat.
- *obsSpeed*: *Integer*, Geschwindigkeit, mit der das Hindernis nach links wandert. Bei dem *Jumping Dino Advanced* kann diese Variable vier unterschiedliche Werte

annehmen.

$$Z_1 = \begin{bmatrix} dist \end{bmatrix}, \quad Z_2 = \begin{bmatrix} dist \\ inJump \end{bmatrix}, \quad Z_3 = \begin{bmatrix} dist \\ inJump \\ obsSpeed \end{bmatrix} \tag{4.1}$$

Die Kombination des Zustandraumes mit den zwei ausführbaren Aktionen ergibt die Anzahl der möglichen Zustands-Aktions-Paare, für die Werte in der Aktions-Nutzentabelle gespeichert werden. Es gibt zwei Szenarien *Simple* und *Advanced* und drei Zustandsmodellierungen Z_1, Z_2 und Z_3. Für das Szenario *Simple* ist die Modellierung Z_3 redundant, da die Variable *obsSpeed* nur einen Wert annehmen kann. Wie bereits erwähnt, muss der *obsSpeed* bei der *Advanced* Variante gegegen sein.

Wichtig zu erwähnen ist, dass diese Werte sich auf die tatsächlich möglichen (s, a)-Paare beziehen. D.h. für die *Simple* Variante ergeben sich für Z_2 in der Theorie 62 mögliche Zustände (Abstand des Hindernisses * im Sprung oder nicht, $31 * 2$). Die Nutzentablle registriert jedoch nur 58 Zustände, da einige Zustände gar nicht erreicht werden können. Für die Abstände $-40, -10, 20$ und 50 existieren nur die Kombinationen mit $inJump = true$. Der Dino musste vorher springen bzw. befindet sich im Sprung, da ansonsten die Episode zuvor bei einer Kollision vorzeitig beendet werden würde. Das erklärt die abweichenden Größen im Vergleich zu den Werten der theoretischen Kombinationen.

Gesamtanzahl aller gespeicherten Aktions-Nutzen:

$$G_{simple,Z_1} = 62, \quad G_{simple,Z_2} = 116, \quad G_{advanced,Z_3} = 4146 \tag{4.2}$$

4.2.3 Belohnungsfunktion

Im Kapitel 2.3 wurde darauf eingegangen, wie in eine Belohnungsfunktion gewählt werden sollte. Die Schlussfolgerung war, dass eine Belohnungsfunktion dem Agenten vermitteln soll, *was* er erreichen soll und nicht *wie* er es erreichen soll (Sutton & Barto, 2018, S. 54).

Die Aufgabe des hüpfenden Dinos besteht darin, die Episode so lang wie möglich zu überleben. Anders formuliert soll die Episode eine maximale Anzahl von Zeitstempeln

andauern. Um dieses Ziel als Belohungssignal zu modellieren, reicht es aus dem Agenten zu jedem Zeitpunkt t eine Belohnung von $+1$ mitzuteilen. Einzige Ausnahme ist hierbei das Erreichen eines Terminalzustandes. Kollidiert der Dino mit einem Hindernis, so erhält er die Belohnung $+0$ und die Episode ist beendet. Der Gewinn einer Episode richtet sich somit nach der Anzahl der Zeitstempel. Eine explizite Modellierung der Hindernisse oder die Vergabe von Belohnungen für das Überspringen dieser ist nicht notwendig, da der Agent implizit lernt, Hindernisse zu überspringen, um die Summe der Belohnungen zu maximieren.

//TODO

4.3 Ant-Game

4.3.1 Problemstellung

4.3.2 Zustandsmodellierung

4.4 Ergebnisse

5 Fazit

6 Ausblick

Literatur

Bellman, R. (1957). Dynamic programming: Princeton univ. press. *NJ*, *95*.

Brunskill, E. (2019). *Stanford cs234: Reinforcement learning - winter 2019.* Zugriff am 2020-02-10 auf `https://www.youtube.com/watch?v=FgzM3zpZ55o`

Capela, M., Céleri, L. C., Modi, K. & Chaves, R. (2019). *Monogamy of temporal correlations: Witnessing non-markovianity beyond data processing.*

Dekking, F., Kraaikamp, C., Lopuhaä, H. & Meester, L. (2006). *A modern introduction to probability and statistics: Understanding why and how.* Springer London. Zugriff auf `https://books.google.de/books?id=TEcmHJX67coC`

Feldman, R. M. & Valdez-Flores, C. (2010). Markov processes. In *Applied probability and stochastic processes.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Gosavi, A. (2009). Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, *21*. doi: 10.1287/ijoc.1080.0305

Howard, R. A. (1960). Dynamic programming and markov processes.

Jaakkola, T., Jordan, M. I. & Singh, S. P. (1994). Convergence of stochastic iterative dynamic programming algorithms. In *Advances in neural information processing systems* (S. 703–710).

Kumar, G. (2014). *Markov chains and markov jump processes* (Unveröffentlichte Dissertation). Indian Institute of Science Education and Research Kolkata.

Mehlhorn, K. & Sanders, P. (2008). *Algorithms and data structures: The basic toolbox.* Springer Science & Business Media.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. Zugriff auf `http://arxiv.org/abs/1312.5602`

Möller, B. & Struth, G. (2004). Greedy-like algorithms in modal kleene algebra. In R. Berghammer, B. Möller & G. Struth (Hrsg.), *Relational and kleene-algebraic methods in computer science.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Oracle. (2020a). *Oracle java documentation: Hashmap java^{TM} 8.* Zugriff auf `https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`

Oracle. (2020b). *Oracle java documentation: Linkedhashmap java^{TM} 8.* Zugriff auf `https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html`

Saul, L. & Jordan, M. (1999, 10). Mixed memory markov models: Decomposing complex stochastic processes as mixtures of simpler ones. *Machine Learning*, *37*, 75-87. doi: 10.1023/A:1007649326333

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, *3* (1).

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second Aufl.). The MIT Press. Zugriff auf `http://incompleteideas.net/book/the-book-2nd.html`

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine learning*, *16* (3), 185–202.

Watkins, C. J. & Dayan, P. (1992). Q-learning. *Machine learning*, *8* (3-4), 279–292.

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

Wiering, M. & van Otterlo, M. (2012). *Reinforcement learning: State-of-the-art.* Zugriff auf `https://ebookcentral.proquest.com`

Yu, J. Y., Mannor, S. & Shimkin, N. (2009). Markov decision processes with arbitrary reward processes. *Mathematics of Operations Research* (3), 737–757. Zugriff auf `http://www.jstor.org/stable/40538443`

Bellman Optimality Equation:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{6.1}$$