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

4. April 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 (**?**, 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 (**?**, 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 (**?**, S. 3). Dabei sind die Übergangswahrscheinlichkeiten zu den einzelnen Folgezuständen ausschließlich von dem aktuellen Zustand abhängig und nicht aufgrund des historischen Verlaufs (**?**, 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 (**?**, S. 48):

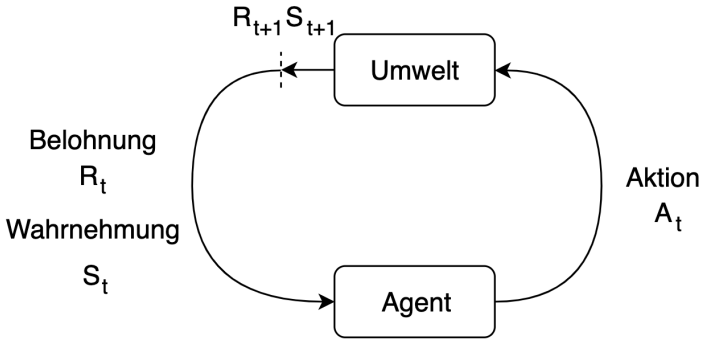

Abbildung 1: Agent-Umwelt Interface

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. **?**, **?** und **?** 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 **?** 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 (**?**, 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 (**?**, 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 (**?**, 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'.

? 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 **?**, **?**, **?** und **?**, oder auch in der Vorlesung der Stanford-Professorin Emma **?**. 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.

? 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. **?** 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 **?**, liefert einen guten Einblick in die Zustandsmodellierung und der Problematik, die mit der Markov-Eigenschaft einhergeht.

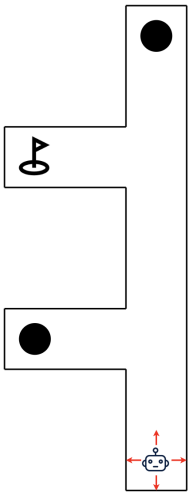

Abbildung 2: Zwei-Wege Beispiel zu der Markov-Eigenschaft

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. **??** 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. **??**. 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.

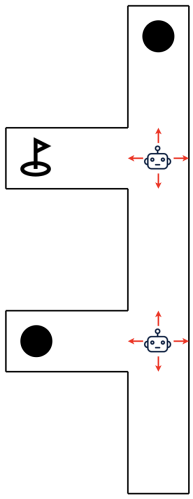

Abbildung 3: Zwei-Wege Beispiel Forts.

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. (**?**, 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. **?** 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“ (**?**, 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 (**??**). 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 (**?**, 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 (**?**, 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 (**?**, 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 (**??**) 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 (**??**) 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 **??** vorgestellt werden, können ausschließlich auf Basis von Episoden lernen. Jedoch existieren auch Methoden, wie das Temporal-Difference-Learning, siehe Kapitel **??**, 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 (**?**, 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 $S_t = s$ (**?**, S. 58). Neben solchen stochastischen Strategien, existieren auch simplere, deterministische Strategien, die jedem Zustand nur genau eine Aktion zuordnen, $\pi(s) = a$ (**?**).

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 (**?**, 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 (**?**, 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 (**?**, 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_* (**?**, 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 (**?**, 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 (**?**, 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 (**?**, 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 (**?**, 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 **?** 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 (**?**, 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 (**?**, 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 (**?**, 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 (**?**, 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 (**?**, 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 (**?**, S. 86). Die erneute Evaluierung der Nutzenfunktion bei der *Policy Evaluation* sorgt indirekt dafür, dass die Strategie nicht mehr gierig ist (**?**, S. 86).

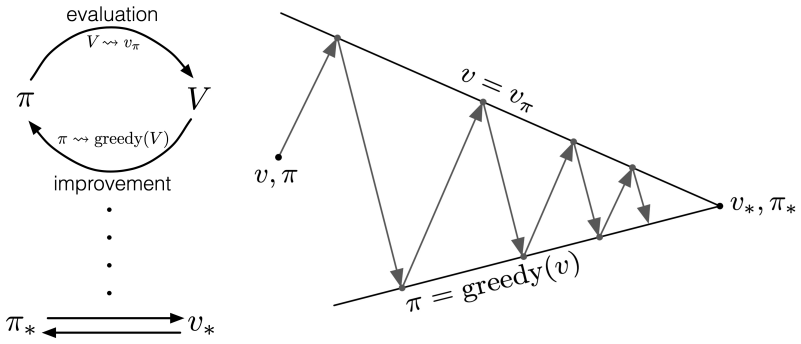

Abbildung 4: GPI nach (**?**, S. 86f)

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. **??**. 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 (**?**, 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 (**?**, 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 (**?**, 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 (**??**) und dem Temporal-Difference Learning (**??**) referenziert werden bzw. als Grundlage für das, in Kapitel **??** 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 (**?**, 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 **??**. Wie bereits in Kapitel **??** 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 (**?**, 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. **?** 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 (**?**, 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* (**?**, 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 (**?**, 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 **?** und **?** 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. **??**) 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 $\pi_{k+1}(s) = \pi_k(s)$ (**?**, S. 22).

Die Strategieiteration generiert somit folgende Sequenz (**?**, S. 22):

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

3.1.4 Nutzeniteration

In Kapitel **??** 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“ (**?**, 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 (**?**, 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 (**?**, 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 **?** 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 (**?**, S. 91). Dabei werden die Nutzen und die Strategien ausschließlich nach einer abgeschlossenen Episode ak-

tualisiert. Folgerichtig ist das Aktualisierungsverhalten *episode-by-episode* und nicht *step-by-step* (als nach jeder Aktion)(**?**, 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 (**?**, 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.

? 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 (**?**, S. 201). Welche Auswirkungen die Werte von ϵ auf das Konvergenzverhalten haben wird im Rahmen der praktischen Umsetzung anhand des JumpingDino Beispiels in Kapitel **??** untersucht.

Um einen einen Überblick über weitere Vorgehensweisen zu der fortlaufenden Erkunden zu geben, werden auch die zwei weiteren Methoden nach **?** 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 (**?**, 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 (**?**, 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: $Q(S_t, A_t) \leftarrow$ 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 **??**. 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 **??**. 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 **?**:

> If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. (**?**, 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* (**?**, 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 (**?**, 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 (**?**, S. 119). Eine vereinfachte formale Darstellung der Aktualisierungsregel für die *Ever-Visit* MC-Methode sieht wie folgt aus (**?**, 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 (**?**, 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 (**?**, 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 **??**) und somit gilt (**?**, 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 **??** 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 (**?**, 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 **??** schätzen zu können und gleichzeitig wird der derzeitg geschätze Nutzen V anstelle des wahren Wertes von v_π verwendet (**?**, S. 120f).

? 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 **??** dargestellten Aktualisierungsregel, konvergiert ist durch die Arbeiten von **?** und **?** bewiesen worden. Weitere Untersuchungen zu dem Konvergenzverhalten der tabularen TD-Methoden wurden zudem von **?** und **?** 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 **??**, 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 (**??**) für die Strategieevaluierung Verwendung findet.

In **??** 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 (**?**, 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 **?** 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 (**?**, 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 (**?**, 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 **?** 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) **?**, 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 **??** 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 (**?**, 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 (**?**, 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

Nachdem die theoretischen Grundlagen in den vorigen Kapitel ausführlich beleuchtet worden sind, folgt nun eine praktische Auseinandersetzung durch die Anwendung verschiedener Lernmethoden des Reinforcement Learnings auf zwei konkrete Problems-

zenarien.

Dabei wird der Leser zunächst mit der, von Grund auf erstellten, Implementierung vertraut gemacht. Anforderungen und Vorraussetzungen werden kurz erläutert und der Grundaufbau wird anhand der wichtigsten Interfaces dargestellt. Anschließend wird auf die beiden Problemstellungen eingegangen.

Neben einer detaillierten Beschreibung der Umwelt, wird der Fokus auf die Zustands- und Belohnungsfunktionsmodellierung gelegt. Geanuer geht es um die Fragestellung, wie eine Modellierung konkret auszusehen hat, damit Rl-Algorithmen auf das Problem anwendbar sind und letztendlich zu einem positiven Ergebnis, dem optimalen Verhalten, streben.

Die Bezeichnungen für die unterschiedliche Varianten der Problemstellungen sowie die zahlreichenden Zustands- und Belohnungsfunktionsmodellierung, die in diesem Kapitel verwendet werden, gelten als Grundlage für die Ergebnisse, die im nächsten Kapitel vorgestellt werden. Als Beispiel eine Analyse des Jumping Dino Problems: $\{FirstVisitMonteCarlo, Simple, Z_2, B_1\}$. Damit ist gemeint, dass die *First-Visit* Variante der Monte-Carlo Methoden als Lernalgorithmus auf das Problemszenarien Jumping Dino *Simple* unter Verwendung der Zustandsmodellierung Z_2 und der Belohnungsfunktion B_1, die jeweils in den Unterabschnitten „Zustandsmodellierung“ respektive „Belohnungsfunktion“ erläutert sind, angewendet wurde.

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 “**?**. Um dennoch eine feste Reihenfolge zu garantieren, wird ausschließlich die *LinkedHashMap* als Implementationen des *Map*-Interfaces benutzt. Diese sichert eine konsistente, vorhersagebare Ordnung zu (**?**).

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

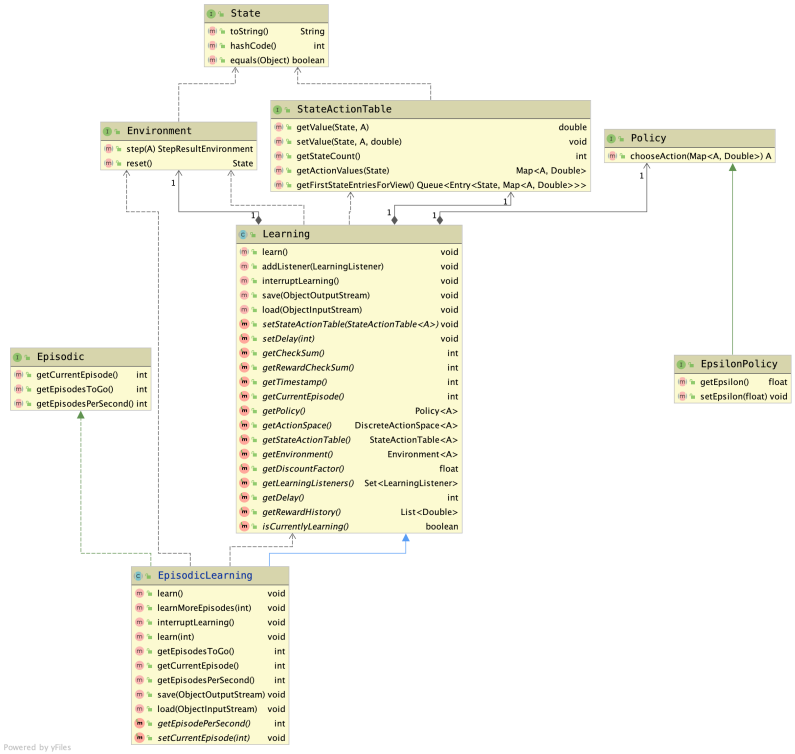

Abbildung 5: Darstellung der wichtigsten Interfaces

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 Pro-

blemszenarien 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 boolesche 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*, Boolescher 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 gegeben 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 muss vorher gesprungen sein 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

//TODO B1: +1 pro Zeitstempel +0 wenn Kollision B2: +1 pro Zeitstempel -1 wenn SPRINGEN im Sprung, +0 wenn Kollision B3: +0 pro Zeitstempel -1 wenn Kollision B4: +0 pro Zeitstempel, -1 wenn Springen im Sprung, -1 wenn Kollision Im Kapitel **??** 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 (**?**, 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.

Eine abweichende Belohnungsvergabe ergibt sich bei Lernmethoden, die nach jedem Schritt eine Anpassung der Nutzen vornehmen. Anstatt die Summe aller Belohnungen einer Episode zu maximieren, wie es die Monte-Carlo Methoden machen, basieren die Aktualisierung bei dem *Temporal-Difference Learning* auf der sofortigen Belohnung und dem geschätzen Nutzen des Folgezustands. Als Initialwert für sämtliche Aktions-Nutzen wird 0 gewählt. Statt dem Agenten positive Belohnungen zu vergeben, damit die Summe dieser auf Episodenbasis maximiert werden kann, wird nun mit negativen Belohnungen für fehlerhaftes Verhalten gearbeitet. Das bedeutet für das Jumpin Dino Spiel, dass für jeden Zeitstempel eine neutrale Belohnung von $+0$ verteilt wird, aber für den Fall einer Kollision -1. Dies hat zudem den positiven Effekt auf die Exploration, denn Aktionen, die zu einer Kollsion geführt haben, nehmen einen negativen Wert an und werden daraufhin von einer gierigen Strategie zunächst ignoriert, weil andere mögliche Aktionen für diesen Zustand einen höheren Nutzen haben, den Intilialwert 0.

Es soll zudem getestet werden, ob das Konvergenzverhalten beeinflusst wird, wenn der Agent negatives Feedback bekommt, wenn er die Aktion JUMP wählt, während er sich bereits in einem Sprung befindet.

Es ergeben sich somit vier Belohnungsfunktionen:

$$B_1 = \begin{bmatrix} +1 \text{ pro Zeitstempel} \\ +0 \text{ bei Kollsion} \end{bmatrix}, \quad B_2 = \begin{bmatrix} +0 \text{ pro Zeitstempel} \\ -1 \text{ bei Kollision} \end{bmatrix},$$
$$B_3 = \begin{bmatrix} +1 \text{ pro Zeitstempel} \\ +0 \text{ bei Kollsion} \\ -1 \text{ wenn JUMP im Sprung} \end{bmatrix}, B_4 = \begin{bmatrix} +0 \text{ pro Zeitstempel} \\ -1 \text{ bei Kollision} \\ -1 \text{ wenn JUMP im Sprung} \end{bmatrix} \tag{4.3}$$

4.3 AntGame

Nachdem im vorrigen Kapitel das episodiale Problem *Jumping Dino* vorgestellt wurde, folgt nun die Darstellung eines kontinuierlichen Problems, dem *AntGame*.

Inspiriert ist dieses Lernszenario durch die gleichnamige Semesteraufgabe in dem Wahlpflichtmodul „Agentensysteme“ der Hochschule Bremerhaven. Zur Lösung jener Aufgabe implementierte der Autor dieser Arbeit bekannte Wegfindungsalgorithmen wie den A-Star und Explorationsmechaniken, die sich an der Tiefen- und Breitensuche orientierten. Die Motivation für die Anwendung von Reinforcement Learning auf dasselbe Problem besteht darin, dass untersucht werden soll, ob RL-Algorithmen in der Lage sind, ähnliches Verhalten zu produzieren wie Algorithmen der klassischen Programmierung.

Konkret wird betrachtet, welche Schritte notwendig sind, um die ursprüngliche Aufgabenstellung zu einem RL-Problem umzuformulieren, damit u.a. die Markov-Eigenschaft erfüllt ist. Außerdem wird die Zielstrebigkeit untersucht und geschaut, ob der Agent implizit den A-Star Algorithmus lernen kann (Ausweichen von Hindernissen, direkter Weg zum Startpunkt).

Zunächst wird die originale Aufgabenstellung des Wahlpflichtkurses kurz vorgestellt. Anschließend werden Änderungen erläutert und die resultierende Umwelt präsentiert. Wie bei dem *Jumping Dino* Lernszenario wird abschließend der Zustands- und Aktionsraum definiert und die Modellierung einer Belohnungsfunktion erläutert.

4.3.1 Inspiration

Die ursprüngliche Aufgabenstellung des Wahlpflichtkurses „Agentensysteme“ ist stark an das Beispiel der „Wumpus-Welt“ von **?** angelehnt (S.197-200). In einer Rasterwelt, siehe Abb. **??**, befindet sich eine Ameise, die auf ihrem Nest spawnt. Ihr Ziel ist es, Futter zu suchen, es aufzusammeln und anschließend zu ihrem Nest zurückzulaufen, um es dort abzulegen. Dabei kann sie sich ausschließlich orthogonal bewegen, Hindernisse können ihren Weg blockieren und sie kann in Löcher fallen, die sie umbringen. Die Wahrnehmung der Ameise beschränkt sich auf die Erkenntnis auf welcher Art von Feld sie sich aktuell befindet sowie dem Futtergeruch bzw. Gestank orthogonaler Futterquellen respektive Löchern.

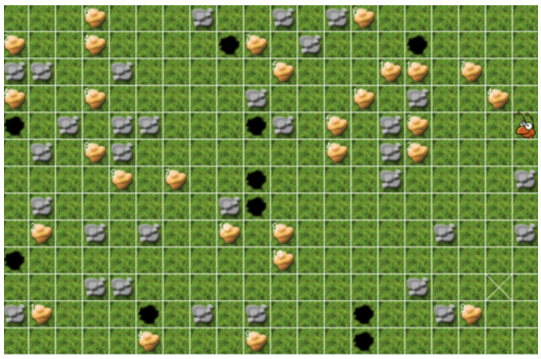

Abbildung 7: Rasterwelt

4.3.2 Problemstellung

Es gibt zwei entscheidende Gründe, weshalb das ursprüngliche Problemszenario umformuliert werden muss. Zum einen kann das Feedback der Umgebung die Markov-Eigenschaft nicht erfüllen und zum anderen ist die Komplexität des Problems für die tabellarischen Lernmethoden deutlich zu hoch.

Zunächst ist zu verdeutlichen, warum das originale Problem die Markov-Eigenschaft nicht erfüllen kann, die in Kapitel **??** näher beleuchtet ist. Eine vereinfachte Variante der Informationen, die die Umwelt dem Agenten zukommen lässt, sieht wie folgt aus:

Listing 1: bar

```
1 {
2     "state": "ALIVE",
3     "currentFood": 0,
4     "totalFood": 0,
```

```
5       "cell": {
6           "row": 0,
7           "col": 10,
8           "type": "START",
9           "food": 0,
10          "smell": 0,
11          "stench": 0
12      }
13  }}
```

Bei einer solchen Zustandsmodellierung kommt es zu einem ähnlichen Problem wie bei dem, in Kapitel **??** vorgestellten, Roboter-Beispiel. Angenommen, die Ameise befindet sich auf dem Feld mit den Koordinaten (0,10) und nimmt den Futtergeruch 1 wahr, dann kann die Ameise nicht logisch entscheiden, in welche Richtung sie sich bewegen muss. In einer Situation befindet sich das Futter auf dem nördlichen Nachbarsfeld, in einer anderen auf dem westlichen Nachbarsfeld. Für den Agenten sehen beide Situation identisch aus und er wird sich aufgrund des höhsten Aktions-Nutzens in beiden Fällen für dieselbe Aktion entscheiden.

Hinzu kommt, dass das Konzept von Fallen bzw. der Fallenerkennung relativ komplex ist und interne Berechnungen benötigt, um logische Schlussfolgerungen ziehen zu können. Bei der ursprünglichen Semesteraufgabe hat der Autor die Umwelt sukzessive in einer internen Repräsentation nachgebaut und anhand von rekursiven Berechnungen Rückschlüsse über mögliche und definitive Fallen zu ziehen.

Um zusätzliche Algorithmen zu vermeiden, die eine interne Repräsentation pflegen oder Observationen der Umgebung zunächst in valide Markov-Zustände umwandeln, wurde auf Fallen als Komponente verzichtet. Zudem ist der Agent nicht mehr in der Lage zu sterben, eine wichtige Änderung hin zu der Konstruktion eines kontinuierlichen Problems.

Finalisiert wird die neue kontinuierliche Umwelt dadurch, dass sich nur eine Futterquelle zur Zeit auf dem Spielfeld befindet. Legt die Ameise das Futter erfolgreich in ihrem Nest ab, so spawnt eine Futtereinheit erneut in einer zufällig gewählten Zelle.

Alle Tests wurden auf einer vordefinierten Rasterwelt durchgeführt, bei der die Hindernisse eine spezielle Anordnung haben. Gleichzeitig ist die Spielfeldgröße auf 8x8 Felder reduziert worden. Eine Visualisierung der Rasterwelt kann der Abbildung **??** entnommen werden, wobei der Start (das Nest) blau, freie Felder grün und Hindernisse rot markiert sind. Die aktuelle Position der Ameise wird mit einem „A“ gekennzeichnet, welches die Farbe Schwarz hat, wenn die Ameise kein Futter trägt bzw. Rot wenn sie Futter bei sich hat.

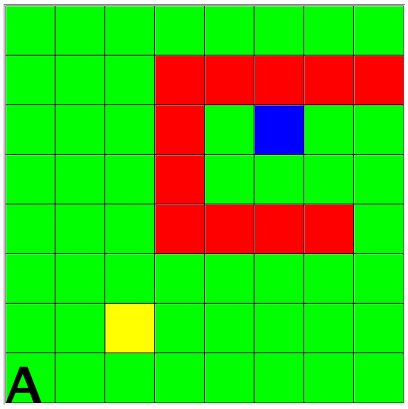

Abbildung 8: Rasterwelt neu

4.3.3 Zustandsmodellierung

Nachdem im vorrigen Unterkapitel die neue Version des *AntGames* definiert worden ist, beschäftigt sich dieses Unterkapitel mit der Frage, wie eine konkrete Zustandsmodellierung aussehen muss, die die Markov-Eigenschaft bedient und zu optimalen Verhalten führt.

Das ursprüngliche Konzept von „Gerüchen“ kann nicht Teil des Zustands sein, da Entscheidung dadruch nicht eindeutig sein können. Nimmt der Agent Futtergeruch wahr, dann befindet sich Futter auf einem der orthogonalen Nachbarsfelder. Doch in welche Himmelsrichtung er sich bewegen muss, ist unklar und Bedarf geplantes Erkunden. Bewegt sich die Ameise z.B. nach Norden und bemerkt, dass sie auf keinem Futter steht, dann sollte sie zurückgehen und anschließend das westliche Nachbarsfeld untersuchen. Nacheinander potentielle Futterquellen abzusuchen erfordert in jedem

Fall das Wissen, welche Felder bereits untersucht worden sind bzw. eine Historie besuchter Felder.

MDPs zeichnen sich jedoch gerade dadurch aus, dass Zustandswechsel nicht abhängig von der Historie der gewählten Aktionen ist, wie in Kapitel **??** und **??** beschrieben. Um zu gewährleisten, dass die Historie nicht Teil des Entscheidungsprozesses sein muss, um optimale Entscheidungen zu treffen, wird die gesamte Spielfeldkonstellation als Zustand gewählt. Dieser Ansatz ist inspiriert durch **?**, die in ihrer Arbeit vorstellen, wie sämtliche *Atari*-Spiele mithilfe von *Deep Reinforcement Learning* auf menschlichem Niveau gespielt werden können. Als Eingabe bzw. Zustand für den Algorithmus dient die gesamte Pixelkonfiguration des aktuellen Frames. Das Besondere hierbei ist, dass keinerlei Anpassungen an dem Programmcode oder dem Neuronalen Netz vorgenommen werden müssen, um alle 49 *Atari*-Spiele erfolgreich zu meistern.

Ein Zustand beeinhaltet somit die komplette Rasterwelt, d.h. ein zweidimensionales Array, welches alle Zellen mit ihrem jeweiligen Typ speichert (*START, FREE, OBSTACLE*) und der Information, ob Futter auf dieser Zelle liegt. Hinzu kommt die aktuelle Position der Ameise, also x- und -Koordinate und der booleschen Flag, ob die Ameise Futter trägt oder nicht.

Intern lässt sich der Zustand folgendermaßen beschreiben:

$$Z_1 = \begin{bmatrix} Cell[][]\ world \\ antPosition\ (x, y) \\ hasFood \end{bmatrix} \tag{4.4}$$

Der Aktionraum umfasst die Aktionen *MOVE_UP, MOVE_RIGHT, MOVE_DOWN, MOVE_LEFT, PICK_UP und DROP_DOWN*. Zusammen mit 2810 möglichen Zuständen für die definierte Rasterwelt ergibt sich die Gesamtanzahl der gespeicherten Aktions-Nutzen von:

$$G_{AntGame,Z_1} = 16860 \tag{4.5}$$

4.3.4 Belohnungsfunktion

In diesem Unterabschnitt wird eine abweichende Methode für die Belohnungsfunktion vorgestellt, die den Lernprozess deutlich beschleunigt.

Das Ziel des Agenten ist es, Futter auf direktem Weg zu seinem Nest zu bringen. Gemäß **?** und der ähnlichen Modellierung bei dem Jumping Dino Problem, sollte es ausreichen, dem Agenten lediglich eine positive Belohnung von +1 zu übergeben, wenn er das Futter auf sein Nest fallen lässt. Alle anderen Aktionen werden mit +0 belohnt. Und in der Tat, führt diese Modellierung zu einem annährend optimalen Ergebnis. „Optimal“ bedeutet im Zusammenhang des *AntGames*, dass die Ameise immer den direkten Weg zum Futter nimmt, es aufhebt und auf direkten Weg zurück zum Nest läuft, um es abzulegen. Sie läuft nicht gegen Hindernisse oder den Rand der Rasterwelt und legt auf keinem anderen Feld als das Nest ab.

$$B_1 = \begin{bmatrix} FOOD\_DROP\_DOWN\_SUCCESS = +1 \\ DEFAULT\_REWARD = 0 \end{bmatrix} \tag{4.6}$$

Diese einfache Modellierung der Belohnung hat jedoch zwei Nachteile. Der erste Punkt ist, dass sie nur annäherungsweise zu dem optimalen Verhalten konvergiert. In Kapitel XX wird eine Methodik vorgestellt, wie gemessen werden kann, ob die Ameise immer den direkten Weg nimmt oder nicht. Vorweggenommen, liegt die durschnittliche Anzahl an Zeitstempeln pro gesammelten Futter bei optimalen Verhalten bei circa 22,92. *QLearning* erreicht mit der einfachen Modellierung jedoch nur einen Durschnitt von circa 23,5. Ein maginal erscheinender Unterschied, der jedoch auffällig genug ist, damit ein Mensch ihn ohne Probleme erkennen kann, wenn er der Ameise zusieht. Es fehlt die explizite Modellierung, dass die Ameise nicht „trödeln“ soll. Dem Labyrinth-Beispiel von **?** nachempfunden, erhält die Ameise nach jedem Zeitstempeln eine negative Belohnung von -1, was letztendlich dazu führt, dass die Ameise optimal handelt.

$$B_2 = \begin{bmatrix} FOOD\_DROP\_DOWN\_SUCCESS = +1 \\ DEFAULT\_REWARD = -1 \end{bmatrix} \tag{4.7}$$

Zwar ist durch diese kleine Änderung sichergestellt, dass die gefundene optimale

Strategie π_* gleichzeitig auch das optimale Verhalten widerspiegelt, doch bleibt das zweites Problem, die Dauer bis zu der Konvergenz. *QLearning* benötigt rund 600000 Zeitstempel, um die ersten 1000 Futtereinheiten zu sammeln. Eine detaillierte Belohnungsfunktion ist in der Lage, den benötigten Aufwand deutlich zu verringern. Aktionen, die dem Ziel der Ameise deutlich widersprechen werden zusätzlich negativ belohnt, damit die Ameise schneller lernt, diese in der Zukunft nicht erneut auszuführen. Hierzu zählt beispielsweise das Laufen gegen Wände oder Hindernisse, das Ausführen der Aktion *PICK_ UP* auf einem Feld ohne Futter oder Futter auf einem anderen Feld als das Nest abzulegen. Durch diese Änderungen ist die Ameise während des Lernprozesses in der Lage, die ersten 1000 Futtereinheiten in nur circa 80000 Zeitstempeln zu sammeln.

Negative Belohnungen bei Ausführung einer unerwünschten Aktion werden mit dem *DEFAULT_ REWARD* addiert wird, wodurch die -2 zustande kommt. Grund hierfür ist, dass der Agent zwischen der normalen „Bestrafung“ eines vergangenen Zeitstempels und einer suboptimalen Aktion unterscheiden muss. Die detaillierte Belohnungsfunktion sieht wie folgt aus:

$$B_3 = \begin{bmatrix} FOOD\_PICK\_UP\_SUCCESS = +0 \\ FOOD\_PICK\_UP\_FAIL\_NO\_FOOD = -2 \\ FOOD\_PICK\_UP\_FAIL\_HAS\_FOOD\_ALREADY = -2 \\ FOOD\_DROP\_DOWN\_FAIL\_NO\_FOOD = -2 \\ FOOD\_DROP\_DOWN\_FAIL\_NOT\_START = -2 \\ FOOD\_DROP\_DOWN\_SUCCESS = +1 \\ RAN\_INTO\_WALL = -2 \\ RAN\_INTO\_OBSTACLE = -2 \\ DEFAULT\_REWARD = -1 \end{bmatrix} \tag{4.8}$$

TODO zusätzlicher Effekt, Diskontierungsfaktor, sonst rundungsfehler, wenn disc zu niedrig

4.4 Ergebnisse Jumping Dino

Dieser Kapitel trägt die Ergebnisse für das Problemszenario Jumping Dino in der *Simple* und *Advanced* Variante zusammen. Speziell untersucht wird die Auswirkung des Explorationsparameters ϵ auf das Konvergenzverhalten bei ϵ-*greedy* Strategien. Zudem sollen Erkenntnisse darüber gesammelt werden, wie sich die beiden Algoritmengruppen, Monte-Carlo Methoden und das *Temporal-Difference Learning*, in Effizienz und Vorgehensweise unterscheiden.

4.4.1 Konvergenzverhalten *Simple* Variante

Betrachtet wird zunächst die *Simple* Variante des Jumping Dino Spiels, bei dem sich Hindernisse stetig mit der gleichen Geschwindigkeit bewegen und mit dem gleichen Abstand zu dem Dino erscheinen, sobald ein Hindernis am linken Spielfeldrand verschwindet.

Methodik

Das Konvergenzverhalten zu untersuchen bedeutet in diesem Fall herauszufinden, wie lange ein Algorithmus braucht, um eine optimalen Strategie π_* zu finden. Da es sich um ein episodiales Problem handelt, kann das „wie lange“ durch die benötigte Anzahl an Episoden gemessen werden.

Die Verwendung von $\epsilon > 0$, sorgt dafür, dass mit der Wahrscheinlichkeit ϵ eine zufällige Aktion ausgeführt wird und optimales Verhalten somit nicht erreicht werden kann. Um dennoch herauszufinden, ab welcher Episode die gespeicherten Aktions-Nutzen q_* entsprechen, wird die ϵ-*greedy* in jeder zweite Episode durch eine *greedy* Strategie ausgetauscht und gemessen, ob der Agent länger als 300000 Zeitstempel überlebt ohne mit einem Hindernis zu kollidieren. Ist dies der Fall, dann ist die optimale Strategie gefunden worden und die Anzahl der Episoden/2 ist der benötigte Aufwand. Die Zahl 300000 ist willkürlich gewählt und muss lediglich groß genug sein, um sicherzustellen, dass der Agent fortlaufend Hindernisse überspringen kann und somit optimales Verhalten im Kontext der Aufgabenstellung erreicht worden ist.

Der *Random Number Generator* spielt eine entscheidene Rolle. Nicht nur für den Zufall bei ϵ-*greedy* Strategien, sondern auch bei der willkürlichen Bestimmung der besten Aktion bei gleichwertigen Aktions-Nutzen. Deswegen wurde das Experiment für jeden Paramter ϵ 100 Mal, mit unterschiedlichen *Random Seeds*, wiederholt, um daraus den Durschnitt mit Standardabweichung zu berechnen.

Für alle Tests, die mit Algorithmen des TD-Learnings durchgeführt worden sind, gilt der Diskontierungsfaktor $\gamma = 0.99$ und die Lernrate $\alpha = 0.9$.

Distanz und „inJump"-Information bei episodialer Belohnungsfunktion

Das Jumping Dino Lernszenario ist ein episodiales Problem bei dem eine Episode endet, wenn der Dino ein Hindernis berührt. Wie im Kapitel **??** erläutert, kann der Gewinn einer Episode gleich der Anzahl an Zeitstempeln sein, d.h. eine Belohnung $+1$ bei jedem Zeitstempel wird vergeben. Neben der Information, wie weit das Hindernis entfernt ist, wird zusätzlich mitgeteilt, ob der Dino sich im Sprung befindet oder nicht. Für die *First-Visit* Variante der Monte-Carlo Methoden ergibt sich folgendes Konvergenzverhalten:

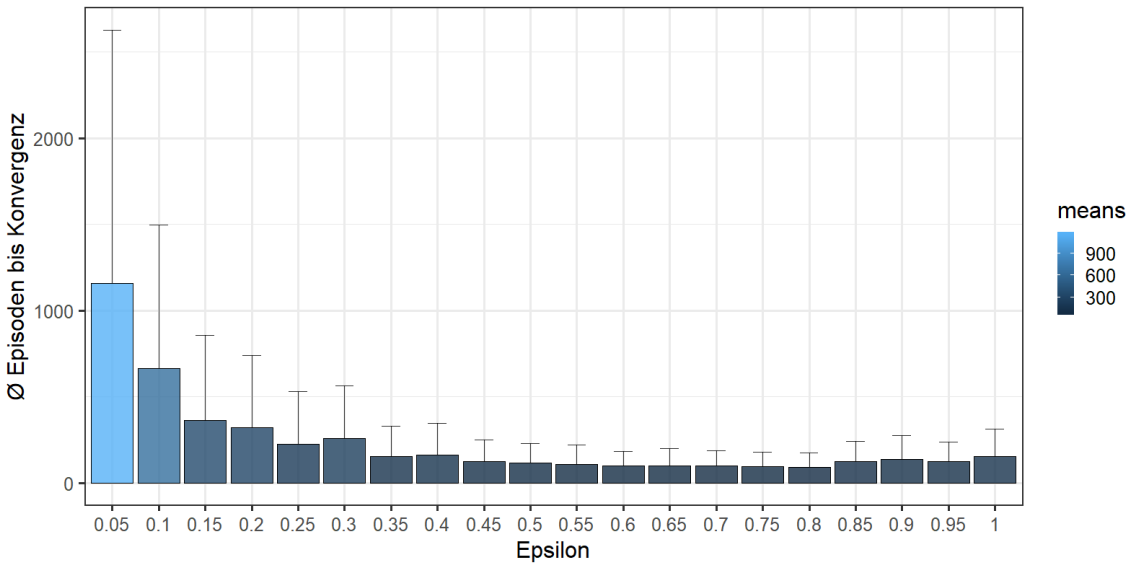

Abbildung 9: First-Visit Monte-Carlo, Z_2, B_1

Deutlich zu erkennen ist, dass geringe Werte für ϵ einen negativen Effekt auf die benötigte Anzahl an Epsioden hat. Für $\epsilon = 0.05$ werden durschnittlich 1159 Epsioden benötigt mit einer sehr hohen Standardabweichung von 1468. Dies lässt sich vor allem damit erklären, dass die meisten Episoden damit enden, dass der Dino gegen das zweite Hindernis springt. Das liegt daran, dass die initalen Werte für jegliche Aktions-Nutzen 0 sind und mit Willkür zwischen ihnen gewählt wird, was dazu führt, dass der Dino immer wieder springt. Dieses Verhalten sorgt bei einer neuen Episode (zufällig) dafür, dass er das erste Hindernis überspringt, aber durch die Sprungabstände permanent mit dem zweiten Hindernis kollidiert. Für diesen Fall ergibt sich ein anhaltender Gewinn von 56 (56 Zeitstempel bis zu der Kollision). Da die verwendete Belohnungsfunktion bewirkt, dass die Aktions-Nutzen einen positiven Wert annehmen, wenn sie in einer Episode ausgewählt werden, werden alternative Aktionen mit einer Wahrscheinlichkeit von $1 - \epsilon$ nicht versucht. Je größer ϵ, desto größer ist die Wahrscheinlichkeit, dass der Algorithmus aus seinem lokalen Maximum herausfindet und durch eine spezielle Aktionsreihenfolge lernt, das zweite Hindernis zu überspringen.

Bemerkenswert ist, dass komplett zufälliges Handeln ($\epsilon = 1$) nur durschnittlich 155 Episoden benötigt, um zu konvergieren. Dies ist ein Hinweis darauf, dass das Problem in der Tat sehr simpel ist, weil der reine *Brute-Force*-Ansatz nicht sonderlich schlechter abschneidet, als die besten Experimente ($0.5 <= \epsilon <= 0.8$) mit ungefähr 100 Episoden bis zu dem Erreichen der optimalen Strategie.

In Kapitel **??** wurde erklärt, dass die Belohnungsfunktion für das TD-Learning anders modelliert werden sollte als für die MC-Methoden. Um herauszufinden, ob eine Konvergenz dennoch stattfinden kann, wurden die gleichen Bedingungen, also Zustandsmodellierung Z_2 und Belohnungsfunktion B_1, bei dem Q-Learning untersucht. Letztendlich ist der Unterschied zwischen der Belohnungsfunktion B_1 und der Belohnungsfunktion speziell für das TD-Learning B_2 lediglich eine Verschiebung um -1. Das Verhältnis zwischen der Belohnung für jeden Zeitstempel und der Belohnung bei der Kollision ist dasselbe.

Neben der Standardinitalisierung der Aktions-Nutzen auf den Wert 0, wird zudem untersucht, ob ein Standardwert von $Integer.MAX\_VALUE$ einen Unterschied verursacht.

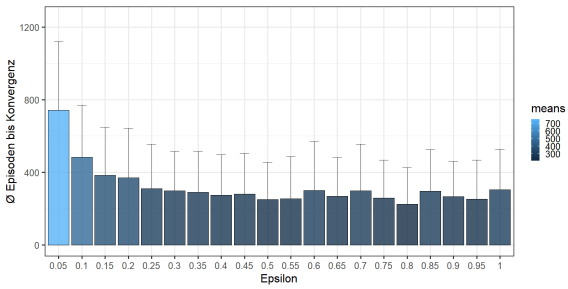

Abbildung 10: Q-Learning Z_2, B_1

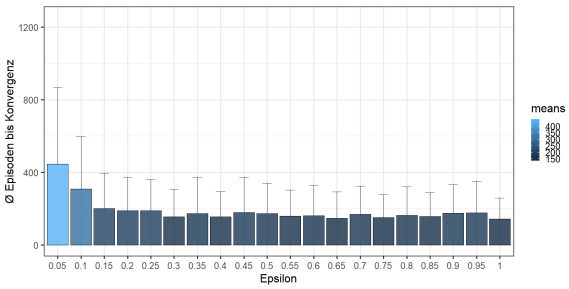

Abbildung 11: Q-Learning, Z_2, B_1,maxQ

Die Ergebnisse zeigen, dass das Q-Learning in der Lage ist, mit der gleichen, für episodiale Lernmethoden zugeschnittene, Belohnungsfunktion zu konvergieren. Andersherum ist dies nicht möglich. Monte-Carlo Methoden unter Verwendung der Belohnungsfunktion B_2 können nicht konvergieren, da jegliche Aktions-Nutzen den Wert -1 annehmen.

Für den Initialwert 0 für neue Aktions-Nutzen konvergiert das Q-Learning im Vergleich zu der *First-Visit* Monte-Carlo Methoden im Durschnitt langsamer. Der Durschnitt für die benötigten Episoden über alle Werte des Parameters ϵ für die MC-Methode beträgt 234, für das Q-Learning 320. Werden die Aktions-Nutzen allerdings mit dem $Integer.MAX\_VALUE$ initialisiert, dann ist das Q-Learning sogar performanter als die *First-Visit* Variante der MC-Methoden, mit einem Durschnitt von 188 Episoden bis zu der Konvergenz. Die Initialisierung der Aktions-Nutzen mit sehr hohen Werten begünstig die Erkundung des Zustand- bzw. Aktionsraumes, da die realen Nutzen kleiner sind als die intialen Werte. Eine Aktualisierung bedeutet eine Verringerung der Aktions-Nutzen der ausgeführten Aktion, wodurch nicht ausgewählte Aktionen den größten Nutzen bei der nächsten Entscheidung beibehalten und mit einer Wahrscheinlichkeit von $1 - \epsilon$ ausgewählt werden.

Angepasste Belohnungsfunktion für TD-Learning

Wie wichtig es ist, die Belohnungsfunktion korrekt zu modellieren und das nicht nur im Bezug auf das eigentliche Problem, sondern auch auf den verwendeten Lernalgorithmus, zeigen die nachfolgenden Ergebnisse.

Anstatt positive Verhalten zu belohnen und für jeden Zeitstempel $+1$ zu vergeben wird bei der Belohnungsfunktion B_2 nur Fehlverhalten, die Kollision mit dem Hindernis, bestraft. Um die Summe der Belohnungen zu maximieren, wie es jeder RL-Algorithmus versucht, muss somit eine Kollision vermieden werden. Durch den Diskontierungsfaktor $\gamma = 0.99$ bekommt der Agent die „Weitsicht“, um Entscheidungssequenzen zu erlernen, die eine Kollision und damit den Erhalt einer negativen Belohnung verhindern.

Die Anpassung der Belohnungsfunktion zeigt die Stärken der TD-Algorithmen *Q-Learning* und *SARSA*, die eine deutlich schnellere Konvergenz im Vergleich zu der *First-Visit* MC-Methode bieten.

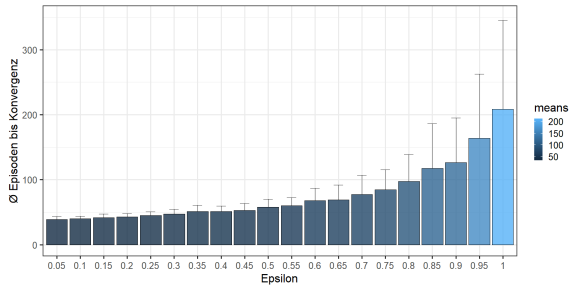

Abbildung 12: Q-Learning Z_2, B_2

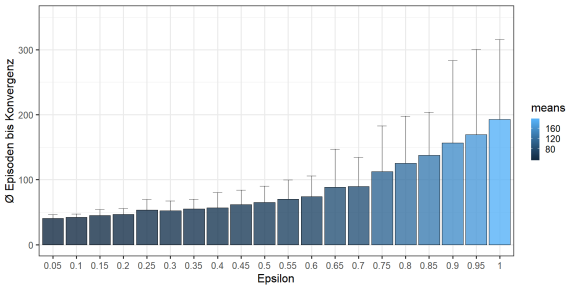

Abbildung 13: SARSA Z_2, B_2

Beide TD-Algorithmen konvergieren unter Verwendung von B_2 deutlich schneller als die MC-Methode, wobei *Q-Learning* mit 77 durschnittlichen Episoden marginal bessser abschneidet als *SARSA* mit 86 Episoden. Der deutliche Unterschied liegt im umgekehrten Trend, der durch den Paramter ϵ gegeben ist. Hierbei steigt der Aufwand bis zu Findung der optimalen Strategie mit größer werdenden ϵ. Beide Algorithmen erreichen bei $\epsilon = 0.05$ die schnellste Konvergenz (39 respektive 40 benötigte Episoden). Grund hierfür ist die Modellierung der Belohungsvergabe und den daraus resultierenden Aktions-Nutzen. Je geringer ϵ, desto konsistenter wird die Aktion mit dem aktuell höhsten Nutzen ausgewählt. Aktionen die bereits ausgeführt worden sind und letztendlich zu einer Kollision geführt haben, nehmen einen negativen Wert an, wodurch alternative Aktionen passiv zu der vermeintlich besten Aktion werden.

Fehlen der „inJump“ Information

Der naive Erstansatz des Autors für die Zustandsmodellierung des Jumping Dino *Simple* Lernszenarios bestand aus der Annahme, dass der Abstand zwischen Dino und Hindernis ausreichend sei, um das Problem lösen zu können. In diesem Unterabschnitt wird diskutiert, ob diese Information wirklich gegeben sein muss, bzw. warum es ohne sie nicht zu einer optimalen Entscheidungsfindung kommen kann.

Zu Beginn wurde angenommen, dass das Jumping Dino *Simple* Problem eine einfache Schwellwertsuche sei. Der Dino muss dabei nur den korrekten Abstand zu dem Hindernis finden, bei dem er die Aktion *JUMP* ausführen muss, bei allen anderen Abständen genügt die Aktion *NOTHING*. In der Theorie ist das auch der Fall, deswegen warf sich das Rätsel auf, warum die *First-Visit* Monte-Carlo Methode unter dieser Zustandsmodellierung nicht zu einer optimalen Strategie konvergieren konnte. In nur durschnittlich 7 von 100 Experimenten pro Paramterwert ϵ findet die *First-Visit* Variante das optimale Verhalten.

Bei genauerer Betrachtung lässt sich jedoch der Grund für hierfür finden. Der Dino schafft es beständig über das erste Hindernis zu springen. Dabei erfährt er bereits alle 31 möglichen Zustände. Kommt das zweite Hindernis auf ihn zu, so erlebt er erneut sämtliche Zustände. Entscheidend ist, dass die *First-Visit* Variante nur für den ersten Besuch eines Zustands-Aktions-Paares die Aktions-Nutzen anpasst, also für alle Paare für die Begegnung mit dem ersten Hindernis. Durch den Sprung über das erste Hindernis verschieben sich jedoch die Zeiträume, in dem der Dino sich im Sprung befindet und weil er durch die fehlende Information „inJump“ keine Möglichkeit hat zu wissen, ob die Aktion *JUMP* tatsächlich einen Sprung auslösen kann oder er sich z.B. gerade im Fall befindet, scheitert er unaufhörlich an dem zweiten Hindernis.

Die *First-Visit* Variante ist somit nicht auf diese konkrete Aufgabenstellung anwendbar. Doch schafft es die abgewandelte *Every-Visit* Variante, bei der alle Zustands-Aktions-Paare einer gesammelten Episode aktualisiert werden?

Die Antwort lautet *ja*, jedoch mit einer sehr hohen Varianz bzw. Standardabweichung zwischen den einzelnen Durchläufen bei gleichem ϵ. Beispielsweise gibt es in der Testgruppe $\epsilon = 0.3$ Durchläufe, die bereits nach nur 7 Episoden konvergiert sind und

solche, die bei einem anderen *Random Seed* jedoch über 65000 Episoden benötigen.

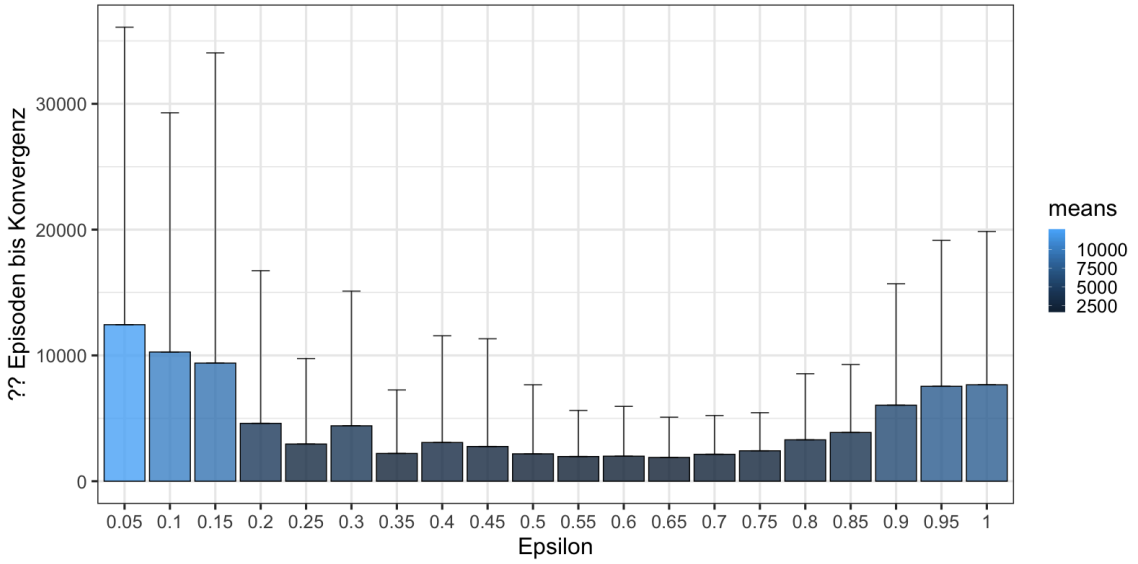

Abbildung 14: Every-Visit Monte-Carlo, Z_1, B_1

Q-Learning und *SARSA* sind ebenfalls in der Lage, ohne die Information, ob sich der Agent im Sprung befindet oder nicht, zu optimalen Verhalten zu konvergieren. Im Gegensatz zu dem *Every-Visit* Algorithmus konvergieren diese beiden TD-Algorithmen allerdings deutlich schneller.*Every-Visit* benötigt durschnittlich 4656 Episode im Vergleich zu dem Q-Learning, welches nur durschnittlich 210 Epsioden benötigt. Des Weiteren ist dieses konkrete Szenario das einizige, bei dem *SARSA*, mit durschnittliche 184 Episoden, besser abschneidet als das *Q-Learning*. In allen anderen Versuchen zeigte das *off-policy* Lernen ein schnelleres Konvergenzverhalten.

Auffällig ist, dass ϵ nur einen sehr geringen Einfluss auf das Konverhalten beider Algorithmen hat.

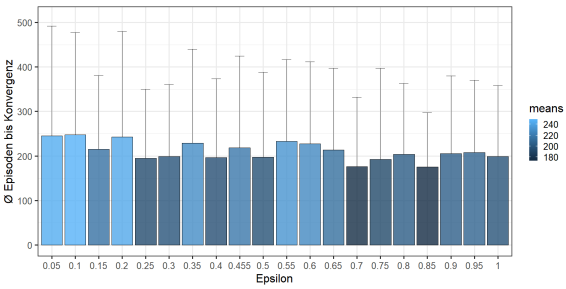

Abbildung 15: Q-Learning Z_1, B_2

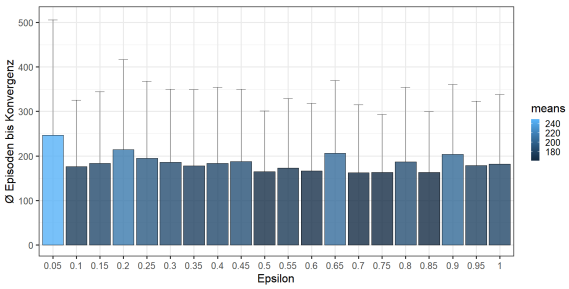

Abbildung 16: SARSA Z_1, B_2

Einfluss einer detaillierteren Belohnungsfunktion

Als Letztes wurde getestet, ob eine detaillierte Belohnungsfunktion die Konvergenzgeschwigkeit erhöhen. Hierzu wurde ebenfalls eine Belohnung von -1 an den Agenten verteilt, wenn dieser versucht im Sprung die Aktion *JUMP* auszuführen, da diese keinen Effekt hat. Die Annahme ist, dass das *Q-Learning* dadurch einen deutlich geringen Zustands- und Aktionsraum für die Auswahl der optimalen Entscheidungssequenz zur Verfügung hat, da es direkt die Aktion *JUMP* als suboptimal verwirft, sobald einmal die Aktion *JUMP* in einem Zustand mit „inJump“ ausgeführt wird.

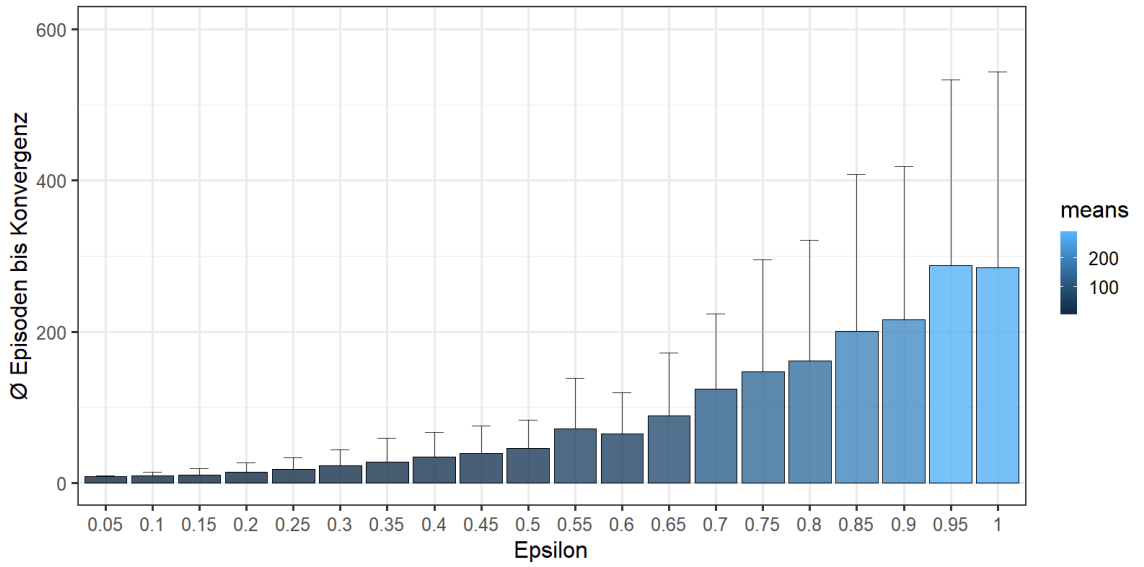

Abbildung 17: Q-Learning Z_2, B_4

Mit nur 8 durschnittlichen Episoden bis zu dem Erreichen des optimalen Verhaltens bei einer Standardabweichung von 1.6 ist diese Kombination von Lernalgorithmus, Zustands- und Belohnungsfunktionmodellierung mit Abstand die effizienteste.

4.4.2 Konvergenzverhalten *Advanced* Variante

Um den Aufwand zur Lösung des Jumping Dino Spiels zu vergrößern, wurde eine *Advanced* Variante in Kapitel 4.2.1 vorgestellt. In dieser Version bewegen sich die Hindernisse zufällig mit vier unterschiedlichen Geschwindigkeiten und erscheinen mit vier unterschiedlichen Abständen. Statt 116 Zustands-Aktions-Paaren, müssen nun 4146 Paare gespeichert und bewertet werden.

Betrachtet wird zunächst die *First-Visit* Variante der Monte-Carlo Methoden, die ausschließlich auf Basis von vollständig gesammelten Episoden lernen kann.

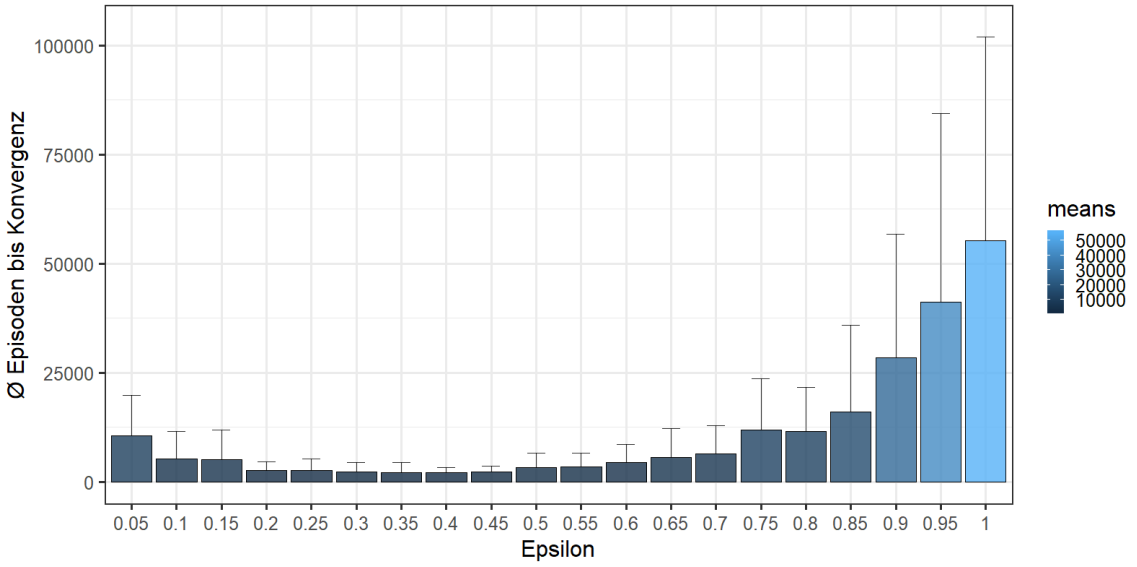

Abbildung 18: First-Visit Monte-Carlo Z_3, B_1

Im Vergleich zu der *Simple* Variante, bei der niedrige Werte für ϵ am langsamsten konvergierten, ähnelt die Verteilung der, der TD-Algorithmen, die mit höherer Zufallswahrscheinlichkeit ϵ deutlich mehr Episoden benötigten, um zu optimalen Verhalten zu gelangen. Durschnittlich benötigte der *First-Visit* Algorithmus 11155 Episoden und

lieferte bei $\epsilon = 0.4$ das beste Ergebnis mit 2079 Episoden bei einer Standardabweichung von 1190.

Das *Advanced* Problemszenario ist in sofern komplexer, da deutlich längere Entscheidungssequenzen bewertet und optimiert werden müssen. Lange Sequenzen wie in diesem Beispiel verdeutlichen die threoretischen Annahmen für Werte von ϵ. Ist ϵ zu niedrig gewählt, dann sinkt die Effizienz. In diesem konkreten Experiment äußern sich Werte kleiner gleich 0.15 negativ auf das Konvergenzverhalten. Niedrige Werte für ϵ bewirken, dass aktuell optimal erscheinende Entscheidungssequenzen nur langsam angepasst werden, um letztendlich verbesserte Sequenzen zu erzeugen. Andersherum sorgen hohe Werte für ϵ dafür, dass bisher optimale Entscheidungsketten frühzeitig durch eine suboptimal, zufällig gewählte Aktion unterbrochen werden und kein zusätzliches Lernen durch die aktuelle Episode erreicht werden kann. Dabei ist $\epsilon = 1$ ist der Extremfall, bei dem jegliche Kombinationen von Aktionsreihenfolgen zufällig erzeugt werden und lediglich die Bewertung dieser zu optimalen Verhalten führt. Für diesen Fall wird also nur das Vorhersageproblem gelöst und das Konzept der *Generalized Policy Iteration (GPI)* verworfen, weil keine Strategieverbesserung stattfinden kann, da sich die Strategie nicht an den Aktions-Nutzen orientiert, sondern ausschließlich willkürlich agiert.

Anwendung der TD-Algorithmen

Die Ergebnisse zu dem Konvergenzverhalten bei der *Simple* Variante des Jumping Dino Spiels haben gezeigt, dass die TD-Algorithmen *Q-Learning* und *SARSA* deutlich effizienter lernen als die MC-Methoden. Daher ist anzunehmen, dass sie auch in der *Advanced* Variante eine bessere Performance zeigen.

In der praktischen Anwendung der beiden TD-Algorithmen stellte sich jedoch heraus, dass beide nicht in der Lage sind, zu optimalen Verhalten zu konvergieren. Dabei ist es unerheblich, welche Variante der Belohungsmodellierung gewählt wird. Auf eine aufwendige Untersuchung, welche konkreten Ursachen für dieses Versagen verantwortlich sind, wurde aus zeitlichen Gründen verzichtet. Neben den unterschiedlichen Belohnungsfunktionen wurden auch zahlreiche Kombinationen von Diskontierungsfaktor γ

und Lernrate α getestet, ohne Erfolg.

4.4.3 Zusammenfassung

In diesem Kapitel wurden viele unterschiedliche Problemszenarien unter Verwendung verschiedener RL-Algorithmen untersucht. In der *Simple* Variante des Jumping Dino Spiels zeigten die *Temporal-Difference Learning* Algorithmen ein deutlich schnelleres Konvergenzverhalten. Vor allem bei der angepassten Belohnungsfunktion B_4, die wirkungslose Aktionen direkt bestraft, zeigte das *Q-Learning* seine Stärken.

Interessante Erkenntnisse konnten über die unterschiedliche Vorgehensweise zwischen den beiden MC-Methoden *First-Visit* und *Every-Visit* gesammelt werden. Nur die *Every-Visit* war in der Lage zu optimalen Verhalten zu konvergieren, ohne die Information zu erhalten, ob der Dino sich derzeit im Sprung befindet oder nicht.

Dass theoretisches Grundlagenwissen über die Arbeitsweise der beiden Algorithmengruppen Monte-Carlo und *Temporal-Difference* essentiell für eine Bewertung ist, welche Algorithmen für eine bestimmte Aufgabenstellung anwendbar sind, offenbart die Anwendung dieser auf die *Advanced* Variante. Bei klar definierten Episoden kann die Anwendung der MC-Methoden präferiert werden, da die Vorgehensweise bzw. die Berechnung der Aktions-Nutzen leichter zu durchdringen ist als bei dem TD-Learning, die das Konzept eines Terminalzustands nicht benutzen. Zudem sind die MC-Methoden in der Lage bei der *Advanced* Variante zu optimalen Verhalten zu konvergieren, die TD-Algorithmen jedoch nicht.

4.5 Ergebnisse AntGame

Im vorherigen Kapitel wurden die Ergebnisse für ein episodales Problem vorgestellt. Dieses Kapitel präsentiert nun die Ergebnisse für das kontinuierliche Lernszenario *AntGame*, bei dem kein Terminalzustand vorhanden ist und somit keine Episoden erzeugt werden können. Folglich werden alle Experimente mit einem *Temporal-Difference* Algorithmus, genauer dem *Q-Learning*, durchgeführt.

Zunächst werden die Auswirkungen von verschiedenen Kombinationen von Diskontie-

rungsfaktor und Lernrate gezeigt. Anschließend wird geklärt, ob *Q-Learning* in der Lage ist optimales Verhalten zu erlernen, d.h. direkte Wege zu Laufen und so effizient wie möglich Futter zu sammeln.

4.5.1 Auswirkung Diskontierungsfaktor und Lernrate

Im Vergleich zu den MC-Methoden haben zwei weitere Parameter Einfluss auf das Lernen bei dem *Q-Learning* bzw. den TD-Methoden. Zum einen der Diskontierungsfaktor γ, der bei MC-Methoden in der Theorie den Wert 1 hat und somit nicht explizit erwähnt wird. Die Auswirkungen von γ wurden in Kapitel **??** erläutert und werden in diesem Unterabschnitt praktisch untersucht. Zum anderen die Lernrate α, die in Kapitel **??** erwähnt worden ist und eine Gewichtung darstellt für die Anpassung alter Werte von $q(s, a)$ durch neue Schätzungen.

Methodik

Für die Untersuchungen für diesen Unterabschnitt wurde die detaillierte Belohungsfunktion B_3 verwendet, jedoch mit einer kleinen Modifikation. Anstatt einer Belohnung von $+1$ für das erfolgreiche Ablegen von Futter in das Nest der Ameise wurde eine Belohnung von $+40$ verteilt, um eine größere Differenz zwischen den durschnittlichen Belohnungen pro Zeitstempel im Verlauf zu erzeugen.

Alle Exprimente wurden mit dem Explorationsfaktor $\epsilon = 0.15$ durchgeführt.

Die Werte für die durschnittliche Belohnung pro Zeitstempel ergeben sich aus den Durschnitten für jeweils 1000 Zeitstempel.

Kombinationen Diskontierungsfaktor und Lernrate

Folgende Untersuchungen sollen explizit keine Aussage über die Fähigkeit für das Erreichen des optimalen Verhaltens treffen, sondern zeigen, wie sich Diskontierungsfaktor und Lernrate auf das Konvergenzverhalten des *Q-Learnings* an sich auswirken.

Wie eingangs erwähnt, gibt die Lernrate α an, wie schnell neue Gegebenheiten bzw.

Schätzungen vorhandene Aktions-Nutzen verändern. Eine hohe Lernrate sorgt somit dafür, dass der Algorithmus sich schnell an neue Eigenschaften der Umwelt angepasst und alte Lernfortschritte überschreibt. Da die Umwelt des AntGames stationär ist und sich im Verlauf der Interaktion nicht ändert, ist anzunehmen, dass eine sehr hohe Lernrate zu schnellerer Konvergenz führt.

Interessant ist zudem die Einwirkung von dem Diskontierungsfaktor γ. Mit Worten umschrieben sagt dieser aus, wie „weitsichtig“ ein Agent handeln soll. Im Fall des AntGames bekommt der Agent erst eine positive Belohnung, wenn er Futter aufsammelt bzw. auf das Nest ablegt. In der Zwischenzeit erhält er ausschließlich negative Belohnungen und muss somit die Fähigkeit haben, zu wissen, dass in der Zukunft, unter bestimmten Entscheidungsverhalten, eine postive Belohnung auf ihn wartet.

Der erste Diskontierungsfaktor, der getest wird, ist 0.5:

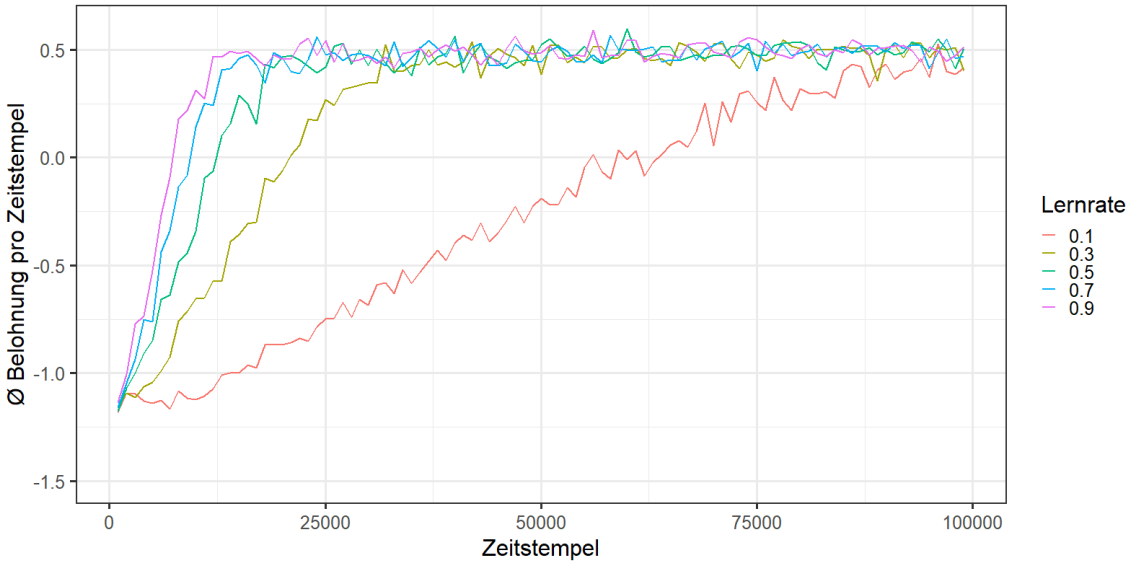

Abbildung 19: Durschnittliche Belohnung pro Zeitstempel bei $\gamma = 0.5$ und $\epsilon = 0.15$

Bezogen auf die Lernrate α konnte die ursprüngliche Annahme bestätigt werden. Alle Durchläufe konvergieren zwar auf eine durschnittliche Belohnung pro Zeitstempel von circa 0.5, jedoch wird diese Konvergenz mit steigender Lernrate schneller erreicht. Eine Lernrate von 0.9 benötigt rund 12500 Zeitstempel, wohingegen eine Lernrate

von 0.1 über 100000 Zeitstempel benötigt, um zu konvergieren. Durch ein weiteres Expriment, welches nicht Teil der Abbildung ist, konnte zudem bewiesen werden, dass sogar eine Lernrate von $\alpha = 1$ praktibel für dieses stationäre Problemszenario ist und am schnellsten konvergiert.

Das zweite Experiment wurde mit einem höheren Diskontierungsfaktor von $\gamma = 0.9$ durchgeführt:

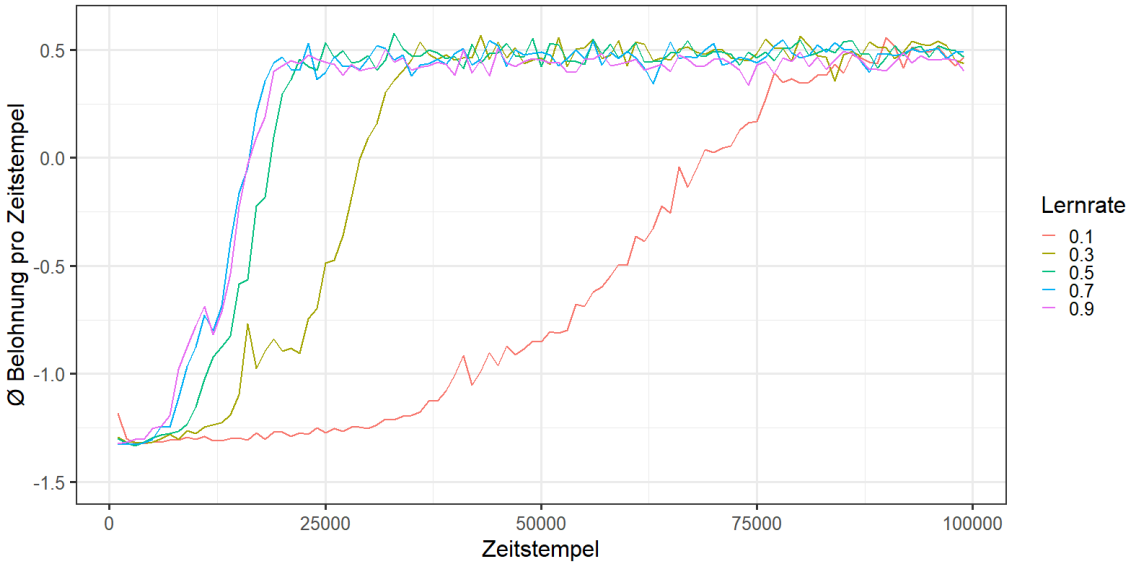

Abbildung 20: Durschnittliche Belohnung pro Zeitstempel bei $\gamma = 0.9$ und $\epsilon = 0.15$

Zu erkennen ist, dass sämtliche Durchläufe zu Beginn des Lernens eine deutlich schlechtere durschnittliche Belohnung pro Zeitstempel erhalten. Bei dem vorherigen Experiment in Abbildung **??**, erleben alle Durchläufe (mit Ausnahme des $\alpha = 0.9$ Durchlaufs) einen starken Anstieg direkt zu Beginn des Lernensprozesses. Durch die Erhöhung des Diskontierungsfaktors γ auf einen Wert von 0.9 agiert der Agent zu weitsichtig im Bezug auf dieses Problem. Der Agent betrachtet nicht den erwarteten Gewinn bis zu dem Ablegen des Futters und der daraus resultierenden postiven Belohnung, sondern weit darüber hinaus, wodurch erfolgreiche Entscheidungssequenzen zunächst schlechter erscheinen, weil negative Belohnungen nach dem Ablegen zu optimalen Entscheidungen der Vergangenheit und deren erwarteter Gewinn hinzu addiert werden. Anders ausgedrückt lernt und bewertet der Agent nicht Entscheidungsreihenfolgen,

um eine Futterquelle zu sammeln und auf dem Nest abzulegen, sondern Reihenfolgen, die beispielweise 10 Futtereinheiten auf optimalen Weg sammeln. Dadurch erhöht sich der Aufwand der Exploration und der Iterationen bis zu einer korrekten Bewertung der Aktions-Nutzen.

Diese Schlussfolgerung lässt sich zudem sehr gut durch einen noch höheren Wert für γ überprüfen. Wie in Kapitel **??** erläutert, darf γ nicht den Wert 1 für kontinuierliche Probleme annehmen, da sonst der Gewinn, also die Summe der erhaltenen Belohnungen unendlich ist. Erlaubt ist jedoch ein Wert sehr nah an 1, wie z.B. 0.99:

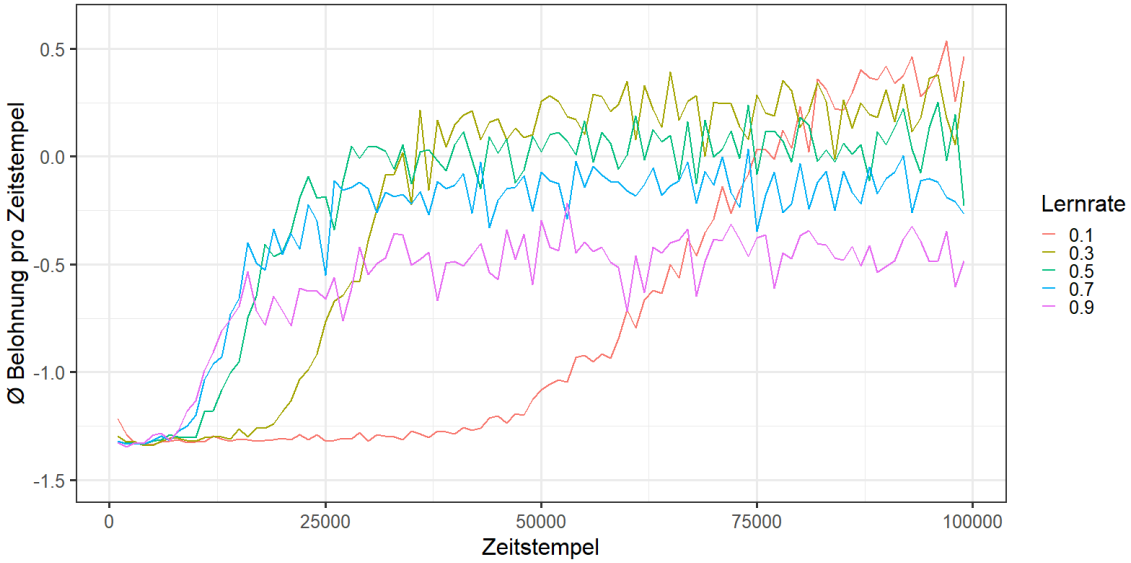

Abbildung 21: Durschnittliche Belohnung pro Zeitstempel bei $\gamma = 0.99$ und $\epsilon = 0.15$

Eine erneute Abschwächung der durschnittlich erhaltenen Belohnung pro Zeitstempel ist festzustellen, vor allem in Kombinationen mit einer sehr niedrigen Lernrate von 0.1. Ein positiver Lerneffekt kann unter diesen Bedingungen erst nach circa 48000 Zeitstempeln bemerkt werden. Auch die restlichen Durchläufe zeigen eine deutlich schlechtere Performance und große Schwankungen im Vergleich zu den beiden vorigen Experimenten. Keine Konfiguration schafft es innerhalb von 100000 Zeitstempeln auf den durschnittlichen Belohungswert von 0.5 zu konvergieren.

Es erscheint zudem, als ob höhere Lernraten zu einem niedrigeren lokalen Maximum

führen. Lokal deshalb, weil alle Konfiguration letztendlich zu dem gleichen Ergebnissen für die durschnittlichen Belohnungen konvergieren wie in den beiden Experimenten zuvor, jedoch erst nach Millionen von Zeitstempel. Erklären lassen sich die unterschiedlichen Maxima damit, dass hohe Lernraten dafür sorgen, dass schneller vermeintlich optimale Entscheidungssequenzen gelernt werden, die Futter zwar aufsammeln, aber nicht auf dem direkten Weg. Je geringer die Lernrate, desto mehr mögliche Reihenfolge bilden den Durschnitt für den erwarteten Gewinn, also auch mehr zufällig gewählte direkte Wege zum Futter und zurück.

4.5.2 Optimales Verhalten

Nachdem im vorigen Unterabschnitt alleine die Auswirkungen zweier Parameter untersucht worden sind, geht es in diesem Unterabschnitt um das Konvergenzverhalten zu dem optimalen Verhalten und der Fragestellung, ob das *Q-Learning* lernt einen Wegfindungsalgorithmus zu immitieren und die Ameise optimal zu steuern, d.h. ohne Umwege zu dem Futter zu leiten, um dieses anschließend in das Nest ablegen zu können.

Methodik

Optimales Verhalten der Ameise spiegelt sich insofern wider, das die Ameise immer die kürzeste Route wählt, keine unnötigen Aktionen ausführt und nicht gegen Hindernisse läuft. Es stellt sich zunächst die Frage, wie gemessen werden kann, ob der Agent ein solches Verhalten erzeugt.

Der erste Schritt besteht darin, zu ermitteln, wie viele Aktionen die Ameise von dem Startfeld aus ausführen muss, um optimal zu handeln. Erscheint eine Futtereinheit direkt neben dem Startfeld zu benötigt die Ameise z.B. vier Aktionen (*MOVE_LEFT, PICK_UP, MOVE_RIGHT, DROP_DOWN*). Die Anzahl der benötigten Aktionen wird nun für jedes Feld ausgerechnet:

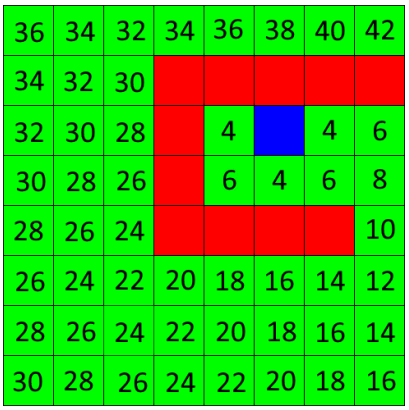

Abbildung 22: A figure

Daraufhin wird der Durschnitt für die benötigten Aktionen bzw. Zeitstempel für ein Futter berechnet. Dieser beträgt 22.92 Zeitstempel pro Futtereinheit. Während des Lernprozess wird jetzt der durschnittliche Aufwand pro Futtereinheit berechnet, indem konkret die benötigten Zeitstempel durch 1000 geteilt werden für jeweils 1000 gesammelte Futtereinheiten. Schwankt der Durschnitt des Agenten um den optimalen Durschnitt, dann ist optimales Verhalten erreicht worden.

Der Explorationsparamter ϵ wurde im Laufe des Lernens verringert, um letztendlich willkürliches Verhalten zu verhindern. Gestartet wird mit einem Wert $\epsilon = 0.2$. Für jede 1000 gesammelte Futtereinheit wird dieser Wert um 0.05 verringert, bis er schließlich bei 4000 gesammelten Einheiten 0 erreicht.

Da durch den vorigen Unterabschnitt gezeigt wurde, dass eine hohe Lernrate für dieses spezifische Problemszenario ausschließlich schnellere Konvergenzen bedeuten, wird die Lernrate $\alpha = 0.9$ gewählt.

Verhalten bei unterschiedlichen Diskontierungsfaktoren

Dass der Diskontierungsfaktor eine entscheidene Rolle für das Konvergenzverhalten spielt, haben vor allem die Experimente des vorherigen Unterabschnitts gezeigt. Zu hohe Werte für γ sorgen für eine deutlich langsamere Konvergenz, doch bewirkt diese auch optimal und kann der Agent zu „kurzsichtig“ sein, wenn γ zu niedrig gewählt wird?

Das nachfolgeden Expriment untersucht, ob unterschiedliche Diskontierungsfaktoren zu optimalen Verhalten führen oder nicht. Dabei ist das optimale Verhalten (22.92 Zeitstempel pro Futter) mit einer schwarz gestrichelten Linie markiert.

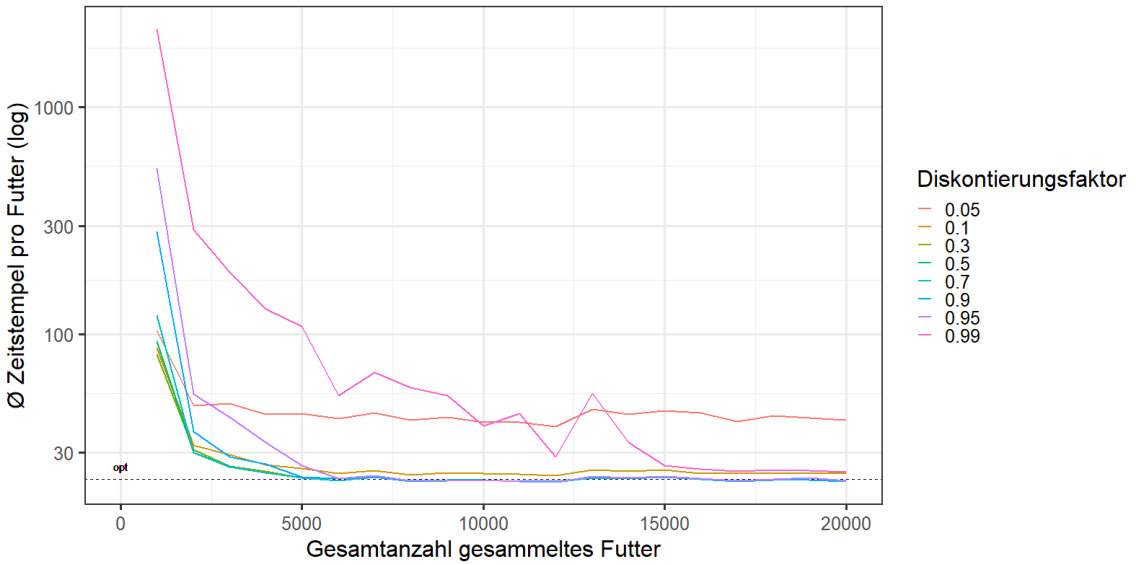

Abbildung 23: Optimales Verhalten bei unterschiedlichen Diskontierungsfaktoren

Ein Diskontierungsfaktor zwischen 0.3 und 0.9 sorgt für eine Konvergenz zu optimalem Verhalten nachdem der ϵ nach 4000 gesammelten Futtereinheit auf 0 gesetzt worden ist. Ein hoher Wert von $\gamma = 0.95$ erreicht erst nach rund 6000 Episoden optimales Verhalten, kann also noch nach dem Wegfallen der Exploration durch Anpassung der Aktions-Nutzen lernen. Auffällig ist, dass bei dem höchste Wert für γ, 0.99, deutlich mehr Zeitstempel benötigt für die ersten 1000 Futtereinheit, 2205 im Vergleich zu nur 81 bei $\gamma = 0.3$.

Die Abbildung **??** zeigt, dass der Durchlauf mit $\gamma = 0.99$ knapp mehr durschnittliche Zeitstempel benötigt als das optimale Verhalten. Dies ist zwar der Fall, gilt aber speziell durch die gewählte Schranken für die Verringerung von ϵ. Wird die letzte Senkung von 0.05 auf 0 auf eine Schranke von 20000 gesammelten Einheiten verringert, dann konvergiert auch $\gamma = 0.99$ zu optimalen Verhalten. Durch die „Weitsichtigkeit“ sind die Entscheidungsreihenfolge umfangreicher und eine längere Exploration muss stattfinden, um die Gewinne für das *bootstrapping* genauer zu schätzen.

Anders ist dies der Fall für die Werte 0.1 und 0.05 für γ. Bei $\gamma = 0.1$ pendelt sich der Agent auf durschnittlich 24.3 Zeitstempeln pro gesammeltes Futter ein, bei $\gamma = 0.05$ sind es circa 42 Zeitstempel und das Verhalten weicht folglich deutlich von dem optimalen ab. Grund für dieses Ergebnis ist, dass der Agent in der Tat zu „kurzsichtig“ geworden ist. Gespeicherte Aktions-Nutzen basieren auf geschätzen Gewinnen, deren Zusammensetzung nicht die notwendigen Belohnung der Zukunft beeinhaltet, die bei dem Ablegen des Futters in das Nest erhalten werden. Wird der Ameise nach dem Lernprozess bei ihrer Suche nach Futter zugeschaut, so ist dieser Effekt sehr sichtbar. Erscheint Futter in der Nähe des Nestes, dann agiert die Ameise optimale und läuft die kürzesten Wege. Futter, welches zu weit von dem Start entfernt erscheint, setzt jedoch lange erlernte Entscheidungsreihenfolge über viele Zeitstempel voraus. Da diese bei zu geringem Diskontierungsfaktor allerdings nicht berücksichtigt werden, bewegt sich die Ameise zufällig auf den Feldern um das Nest hin und her, bis sie zufällig in Reichwerte für das Futter gelangt, um erneut optimal zu laufen. Hat sie das Futter aufgenommen, dann wandert sie erneut willkürlich umher, bis das Nest in Reichweite ist.

Die Erkentnis, dass zu geringe Werte für γ optimales Verhalten verhindern, im Bezug auf diese konkrete Aufgabenstellung, lässt sich zudem durch die nachfolgede Abbildung erkennen. Gezeigt werden die total benötigten Zeitstempel bei fortlaufender Sammlung von Futter.

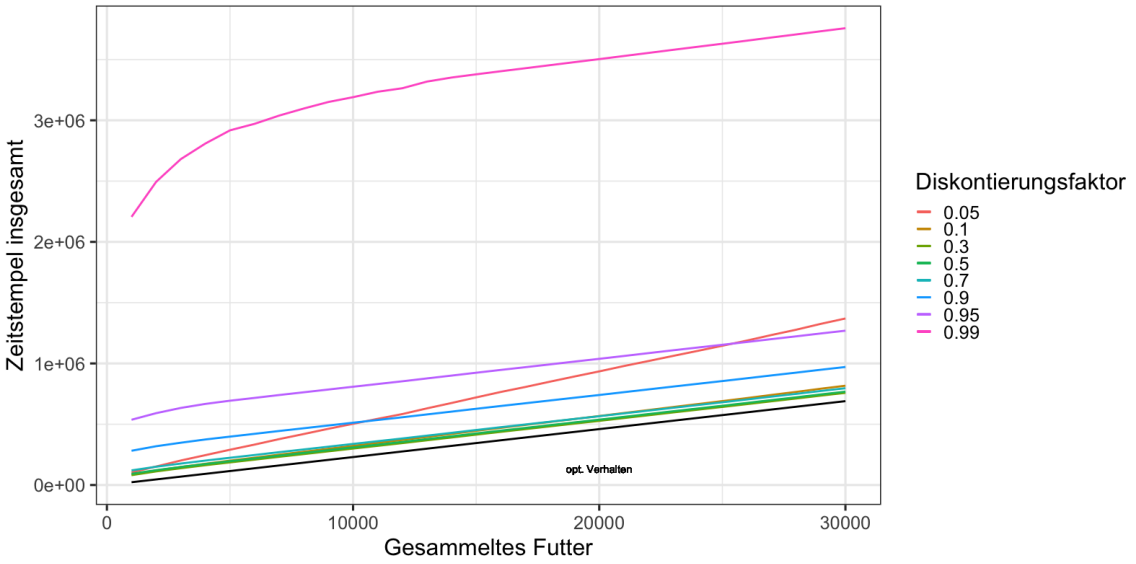

Abbildung 24: Total benötigte Zeitstempel bei fortlaufender Futtersammlung

Ist das optimale Verhalten gefunden, dann ist die Steigung jede Linie gleich und enstpricht 22.9 Zeitstempel pro gesammelten Futter. Folglich dürfte keine Linie eine andere nach erfolgreicher Konvergenz kreuzen. Dies ist jedoch bei $\gamma = 0.05$ und $\gamma = 0.1$ der Fall, da ihre Steigung 42 respektive 24.3 beträgt.

Zusätzlich soll diese Abbildung zeigen, wie viele Zeitstempel insgesamt benötigt werden, um eine Konvergenz zu erzielen. Sind es bei $\gamma = 0.3$ 186k Zeitstempel für 5000 gesammelte Futtereinheiten, ist der Aufwand bei $\gamma = 0.99$ mit 2.9M benötigten Zeitstempeln um mehr als den Faktor 15 größer.

4.5.3 Zusammenfassung

Das AntGame zeigt, dass der Diskontierungsfaktor stark von der gestellten Aufgabe abhängig ist. In der Theorie kann man immer Werte nah an 1 wählen z.b. 0.99. Doch wie gesehen, dauert der Lernprozess in diesem Fall deutlich länger, als eine geeignete Größe, die auf das Problem zugeschnitten ist. Größe Lernraten sorgen für eine schnellere Konvergenz. Lernraten an sich sind nützlich für sich verändernde Umwelten. Niedrige Lernraten passen die Q Werte dann sehr vorsichtig an die neuen Verhältnisse

an, da in dem Beispiel des AntGames jedoch eine stationäre Umwelt gegeben ist, sind höhrere Lernraten zu präverieren.

5 Fazit

6 Ausblick

Bellman Optimality Equation:

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