TT2 Problemstellung 1 : Zusammenfassung

Carsten Noetzel

12.05.2012

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Lernsituationen

1.1.1 Überwachtes Lernen - „supervised learning“

Ein Lehrer sagt was das richtige Ergebnis gewesen wäre.

1.1.2 Unüberwachtes Lernen - „unsupervised learning“

Das System bemerkt selbst, dass es unterschiedliche Eingangsklassen gibt.

1.1.3 (Ver-)Bestärkendes Lernen - „reinforcement learning“

Der Lehrer (Leben/Umgebung) belohnt oder bestraft.

2 Reinforcement Learning

2.1 Einführung

Beim Reinforcement Learning wird der Agent in einer Umgebung platziert, in der er agiert und aus einer Reihe von **Belohnungen/ Bestrafungen** lernt. Anders als beim überwachten Lernen, werden dem Agenten **keine Trainingsbeispiele** vorgegeben, der Agent lernt demnach nur aus seiner eigenen Erfahrung. In vielen Anwendungsbereichen ist es gar nicht möglich Trainingsbeispiele bereitzustellen, anhand derer ein Agent lernen kann (z.B. Schach), somit ist man gezwungen eine andere Form des Lernens anzuwenden.
Das Reinforcement (die Belohnung/Bestrafung) kann dabei entweder **direkt nach einer Aktion** oder **erst am Ende** erfolgen, in diesem Fall muss der Agent dann prüfen, welche der Aktionen am Wahrscheinlichsten die Ursache für das Ergebnis ist.

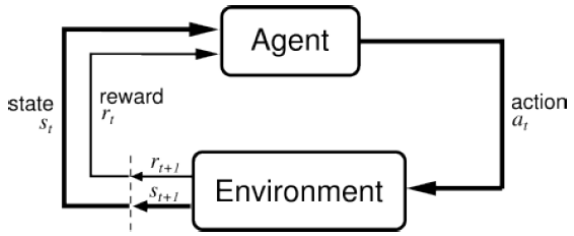

Abbildung 1: Agent in seiner Umgebung

Der Agent befindet sich zu jedem Zeitpunkt einem Zustand s_t und wählt eine Aktion a_t aus, die in der Umgebung ausgeführt wird. Der Agent gelangt daraufhin in den Folgezustand $s_{(t+1)}$ und erhält die Belohnung/Bestrafung $r_{(t+1)}$ von der Umgebung. Belohnung/Bestrafung und Folgezustand sind demnach Phänomene der Umgebung, die vom Agenten beobachtet werden (Modell der Umgebung).
Die **Umgebung** ist im Allgemeinen **nicht deterministisch** (die gleichen Aktionen im gleichen Zustand, können zu unterschiedlichen Folgezuständen führen) **aber stationär** (Wahrscheinlichkeiten für Folgezustände bei gegebener Aktion ändern sich nicht im Laufe der Zeit).

2.2 Policy - Strategie

Welche Aktion abhängig vom Zustand wählen?

2.2.1 stochastische Strategie

Für einen Zeitpunkt t gibt es eine Wahrscheinlichkeit $\pi_t(s, a)$ dafür, dass a die gewählte Aktion a_t sein wird falls s der aktuell vorliegende Zustand s_t sein sollte. Ziel des Agenten ist es seine Belohnungen über die Gesamtlaufzeit zu maximieren, also die optimale Strategie π^* zu lernen.

2.2.2 deterministische Strategie

Die deterministische Strategie mappt Zustände direkt auf Aktionen $\pi : S \to A$, wobei S die Menge der Zustände und A die Menge der Aktionen ist, mit $A(s)$ als mögliche Aktionen im Zustand s. (siehe Reflex Agent)

2.3 Belohnungen

2.3.1 episodische Aufgaben

Der Return ist die Summe der **Rewards ab dem Zeitpunkt** t, bei dem T ein abschließender Schritt wäre $R_t = r_{(t+1)} + r_{(t+2)} + r_{(t+3)} + ... + r_T$. Führt der Agent **episodische Aufgaben** aus, enden diese jeweils mit einem abschließenden Schritt. Hierbei wird zwischen der Menge der nicht-terminalen Zustände S und der Menge aller Zustände S^+ unterschieden.

2.3.2 kontinuierliche Aufgaben

Sind fortdauernde Aufgaben. Da nicht endende Aufgaben zu einer unendlich hohen Belohnung führen würden, muss der **Reward abgeschwächt** werden (**discounting**). Die Abschwächung erfolgt hierbei über eine Discount-Rate $0 \leq \gamma \leq 1$ die die Gesamtbelohnung begrenzt. Für den Reward ergibt sich somit:

$$R_t = r_{(t+1)} + \gamma * r_{(t+2)} + \gamma^2 * r_{(t+3)}) + ... = \sum_{k=0}^{\infty} \gamma^k * r_{(t+k+1)} \tag{1}$$

2.4 Agenten-Designs

2.4.1 Utility Based Agent

Der Agent lernt eine „Utility Function“ und nutzt diese um einen Zustand zu bewerten. Diese wird genutzt um auf Basis des aktuellen Zustands eine Aktion auszuwählen, die den größten Nutzen bringt. Hierzu benötigt der Agent aber ein Modell der Umwelt, um die Folgezustände der Aktionen bestimmen zu können.

2.4.2 Q-Learning

Diese Form von Agenten lernt eine „Action-Utility Function“, welche den erwarteten Nutzen einer Aktion in einem bestimmten Zustand bestimmt. Da die Aktionen verglichen werden ohne ihr

genaues Ergebnis zu kennen, benötigt der Agent kein Modell der Umwelt. Dies führt dazu, dass der Agent nicht in die Zukunft blicken kann, was ggfs. Auswirkungen auf die Lernfähigkeit des Agenten hat.

2.4.3 Reflex Agent

Der Agent lernt eine Strategie, welche Zustände direkt auf Aktionen mapt.

2.5 Arten

2.5.1 Passive Learning

Hierbei ist die Strategie des Agenten festgelegt und seine Aufgabe ist es des Nutzen der Zustände zu lernen. Dies kann auch das Erlernen eines Modells der Umgebung beinhalten.

2.5.2 Active Learning

Der Agent muss zum Nutzen einzelner Zustände auch noch lernen was zu tun ist.

2.6 Markow Decision Process (MDP)

2.6.1 Definition

Markow Entscheidungsprozesse sind ein **mathematisches Framework zur Modellierung von Entscheidungssituationen** in denen das Ergebnis teilweise zufällig ist und der Kontrolle eines Entscheiders unterliegt. MDPs werden genutzt um **Optimierungsprobleme** zu untersuchen die mittels **dynamischer Programmierung** und **Reinforcement Learning** gelöst werden.

Ein Markow Prozess ist ein 4-Tupel $(S, A, P(.,.).R(.,.))$, wobei

- S eine endliche Menge von Zuständen ist
- A eine endliche Menge von Aktionen ist
- $P_a(s, s') = P_r(s_{t+1} = s' | s_t = s, a_t = a)$ die Wahrscheinlichkeit das aus Aktion a im Zustand s zum Zeitpunkt t der Zustand s' zum Zeitpunkt $t+1$ resultiert (Übergangswahrscheinlichkeit durch Aktion a)
- $R_a(s, s')$ ist der Erwartungswert für die unmittelbare Belohnung, die durch den Übergang in den Zustand s' erhalten wird

Konkret sind MDPs **zeitdiskrete stochastische Prozesse**. In jedem Zeitschritt befindet sich der Prozess in einem Zustand s und der **Entscheider fällt zufällig eine Entscheidung** a, die im Zustand s ausführbar ist. Der Prozess wechselt daraufhin in den Zustand s' der dem Entscheider eine entsprechende Belohnung zurückgibt $R_a(s, s')$. Die Wahrscheinlichkeit, dass dabei in den Zustand s' gewechselt wird ist abhängig vom aktuellen Zustand und der gewählten Aktion a, die durch die Zustandsübergangsfunktion $P_a(s, s')$ beschrieben wird. Der Zustandsübergang ist demnach unabhängig von allen vorherigen Zuständen und Aktionen. Diese Gedächtnislosigkeit wird auch als **Markow-Eigenschaft** bezeichnet. Die **Umgebung liefert alle notwendigen Informationen für zukünftige Entscheidungen**.

MDPs sind eine **Erweiterung der Markow-Ketten**. Bei MDPs gibt es eine Auswahl an **Aktionen** (Auswahl) und **Belohnungen** (Motivation). Wenn nur eine Aktion für jeden Zustand existiert und alle Belohnungen gleich null sind, reduziert sich der Markow-Prozess zu einer Markow-Kette.

2.6.2 Markow Entscheidungsproblem

Das Kernproblem von MDPs ist es eine Policy (Strategie) für den Entscheider zu finden: ein Funktion π die die Aktion spezifiziert $\pi(s)$ die im Zustand s gewählt wird. Sobald ein MDP mit einer Policy kombiniert wird, werden die Aktionen für einzelne Zustände festgelegt, was zu einem Verhalten wie in einer Marow-Kette führt. Das Ziel ist es eine Policy π zu finden, die die Belohnungen maximiert, typischerweise ist das der abgeschwächte Reward über alle Zeitschritte.

$$\sum_{t=0}^{\infty} \gamma^t R_{at}(s_t, s_{t+1}) \tag{2}$$

Hierbei ist γ der Discount Faktor, wobei typischerweise gilt $\gamma = \frac{1}{1+r}$ mit r als Discountrate. Aufgrund der Markow-Eigenschaft, dass die Auswahl der Aktionen nur vom aktuellen Zustand abhängt, kann das Problem als Funktion beschrieben werden die nur von s abhängig ist, indem man für $at = \pi(s_t)$ einsetzt. Lösungsverfahren hierfür sind die **Policy bzw. die Value Iteration**.

2.6.3 MDP und Reinforcement Learning

Reinforcement Learning kann das Entscheidungsproblem lösen, ohne die Zustandsübergangsfunktionen $P_a(s, s')$ explizit spezifizieren zu müssen. Diese **Werte werden jedoch für die Value und Policy Iteration benötigt**.
Anstelle die Wahrscheinlichkeiten vorzugeben, werden diese beim Reinforcement Learning ermittelt, indem die **Simulation mehrfach neu gestartet** wird und immer zufällige Initiale Zustände gewählt werden.

2.7 Bewertungsfunktionen

Die Bewertungsfunktionen liefern ein **Maß für die Güte von Zuständen bzw. Aktionen unter Verfolgung einer bestimmten Strategie** und sind daher nützlich für die Auswahl von Aktionen. Die Funktionen sind dem Agenten **zunächst unbekannt** und müssen vom Agenten gelernt /geschätzt werden, sie werden auch als das Gedächtnis des Agenten bezeichnet.

2.7.1 V-Wert

Der V-Wert eines Zustands s ist der Erwartungswert (E) der aufsummierten Belohnungen, die, ausgehend von s unter einer bestimmten Strategie π erreicht werden. Die Funktion V^π heißt **Zustand-Wert-Funktion** (state-value-function) und V^* ist die optimale V-Funktion bei optimaler Strategie π^*.

$$V^\pi(s) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{(t+k+1)} | s_t = s\} \tag{3}$$

2.7.2 Q-Wert

Der Q-Wert eines Zustands s ist der Erwartungswert (E) der aufsummierten Belohnungen die, ausgehend von s unter Auswahl einer bestimmten Aktion a und der anschließenden Verfolgung einer Strategie π erreicht werden. Die Funktion Q^π heißt **Aktion-Wert-Funktion** (action-value-function) und Q^* ist die optimale Q-Funktion bei optimaler Strategie π^*.

$$Q^\pi(s,a) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{(t+k+1)} | s_t = s, a_t = a\} \tag{4}$$

Da die Funktion während des Lernes unbekannt ist, basiert die Erfahrung aus Paaren von (s,a) in der Form „Ich war im Zustand s und habe a ausgeführt und bin dadurch in s' gelandet“. Wenn man das Q-Wert Array hat und dieses durch Erfahrungen direkt aktualisiert, nennt man das **Q-Learning**.

3 Dynamic Programming

3.1 Definition

Unter dynamischer Programmierung versteht man eine Methode zum **algorithmischen Lösen von Optimierungsproblemen**. Der Begriff wurde von Richard Bellman eingeführt, weswegen auch häufig vom „Bellman Prinzip der dynamischen Programmierung gesprochen“ wird. **Voraussetzungen** für die dynamische Programmierung sind:

- Die Menge der Zustände ist bekannt und ist endlich
- Die Menge der möglichen Aktionen {a} ist bekannt und endlich
- Belohnungen für alle Zustände bekannt
- Folgezustände in Abhängigkeit von Ausgangszustand und Aktion bekannt

Dynamische Programmierung lässt sich dann erfolgreich einsetzen, wenn das Optimierungsproblem aus **vielen gleichartigen Teilproblemen** besteht und sich die optimale Lösung des Problems aus den optimalen Lösungen der Teilprobleme zusammensetzt. Hierfür werden zunächst die optimalen Lösungen der **kleinsten Teilprobleme direkt berechnet** und dann geeignet zu einer Lösung eines **nächstgrößeren Teilproblems zusammengesetzt**. **Teilergebnisse werden in einer Tabelle** gespeichert, um bei nachfolgenden Berechnungen gleichartiger Teilprobleme auf die Ergebnisse zurückgreifen zu können. Bei konsequentem Einsatz vermeidet die dynamische Programmierung kostspielige Rekursionen weil bekannte Teilergebnisse wiederverwendet werden. Zum Einsatz kommt die dynamische Programmierung in der Regelungstheorie und verwandten Gebieten und wird dort eingesetzt um beispielsweise Gleichungen herzuleiten (Hamilton-Jacobi-Bellman-Gleichung), deren Lösung den optimalen Wert ergibt. Dynamic Programming hat hauptsächlich einen theoretischen Stellenwert, da die Annahme der Markov-Eigenchaft (ein vollständiges Modell der Welt) nur schwer zu erfüllen ist und die Algorithmen rechenintensiv sind. Alle anderen Verfahren versuchen sozusagen, durch (deutlich) geringeren Rechenaufwand diesem Idealzustand anzunähern.

3.1.1 Bellman-Gleichungen ⇒ diskrete Zeit, dynamische Programmierung

Bellman-Gleichungen sind eine **notwendige Bedingung zur Ermittlung des optimalen Wertes** bei der dynamischen Programmierung. Ausgehend von einer **Initialen Entscheidung** wer-

den die Werte die sich aufgrund dieser ersten Entscheidung ergeben haben festgehalten, um die Werte der **nachfolgenden Entscheidungsprobleme** zu bestimmen. Dadurch wird das große **Optimierungsproblem in kleinere, einfachere Probleme unterteilt** wie es das **Optimalitätsprinzip von Bellman** vorschreibt (eine optimale Lösung setzt sich aus den optimalen Teillösungen zusammen).

Der Begriff Bellman-Gleichung bezieht sich normalerweise auf die dynamische Programmierung mit diskreter Zeit, wobei bei Optimierungsproblemen mit kontinuierlicher Zeit von Hamilton-Jacobi-Bellman-Gleichungen gesprochen wird.

3.1.2 Hamilton-Jacobi-Bellman Gleichungen ⇒ kontinuierliche Zeit, Regelungstheorie

HJBs sind **partielle Differentialgleichungen** zur Lösung von Optimierungsproblemen in Systemen mit **kontinuierlicher Zeit**. Die Lösung der HJB ergibt die „Value Function", welche die optimalen Kosten für ein dynamisches System mit einer assoziierten „Cost Function" darstellt. Lokal gelöst ist die HJB eine notwendige Bedingung für ein Optimum, wird die HJB jedoch über alle Zustände des Raums gelöst, ist sie die notwendige und hinreichende Bedingung für ein Optimum.

3.2 Evaluation und Improvement

3.2.1 Policy Evaluation (Strategie-Bewertung)

Bei der Policy Evaluation wird zunächst die „State-Value-Function" **einer willkürlichen für die Iteration fest vorgegebenen Strategie berechnet**. Hierbei ist $\pi(s, a)$ die Wahrscheinlichkeit die Aktion a im Zustand s unter Strategie π auszuwählen.

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} P^{a}_{ss'} [R^{a}_{ss'} + \gamma V^{\pi}(s')] \tag{5}$$

Für eine iterative Lösung gilt folgende Updateregel: Die Initiale Approximation V_0 wir zufällig ausgesucht und jede nachfolgende Approximation erhält man über die Bellman Gleichung für V^{π}.

$$V_{k+1}(s) = \sum_{a} \pi(s, a) \sum_{s'} P^{a}_{ss'} [R^{a}_{ss'} + \gamma V_k(s')] \tag{6}$$

Zur Erzeugung der Nachfolgeapproximation V_{k+1} von V_k wird in der **iterative policy evalutation** die gleiche Operation für jeden Zustand s ausgeführt: dieser ersetzt den alten Wert von s mit dem neuen, welcher auf Basis der Vorgänger von s und der erwartetem unmittelbaren Belohnungen berechnet wurde. Diese Form der Berechnung wird **full backup** genannt, da in jeder Iteration der berechnete Wert gespeichert wird, um im nächsten Schritt den neuen Approximationswert zu berechnen.

Algorithmisch würde es zwei Arrays geben, eines für die gespeicherten Werte $V_k(s)$ und eines für die neuen Werte $V_{k+1}(s)$. Dadurch können die neuen Werte nacheinander berechnet werden, ohne dass die alten Werte sich ändern. Natürlich könnte man auch auf einem Array arbeiten und die Werte „in place" aktualisieren, wodurch der Algorithmus schneller nach V^{π} konvergiert, da immer die aktuellsten Werte zur Berechnung verwendet werden.
Der Algorithmus bricht ab, wenn der Unterschied zwischen $V_k(s)$ und $V_{k+1}(s)$ klein genug ist.

Input π, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

Abbildung 2: Algorithmus zur Evaluation einer Policy

3.2.2 Policy Improvement (Strategie-Verbesserung)

Der Grund warum wir die „State-Value-Function“ für eine bestimmte Strategie berechnen, ist der dass sie uns helfen soll bessere Strategien zu finden. Nehmen wir an wir haben die „State-Value-Function“ V^π einer willkürlichen Strategie berechnet und möchten für einige Zustände s wissen, ob wir die Strategie ändern sollten um eine Aktion zu wählen für die gilt $a \neq \pi$. Durch den **Austausch einzelner Aktionen**, kann die Strategie ggfs. verbessert werden.

Wenn in einem Zustand s nicht der zugrundeliegenden Strategie π gefolgt wird, sondern eine einzelne Aktion $a \neq \pi$ gewählt wird (und danach weiterhin entsprechend π vorgegangen wird), ergibt sich folgender Wert für s:

$$Q^\pi(s,a) = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\} \tag{7}$$

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \tag{8}$$

Das Hauptkriterium ist, ob der durch den Austausch errechnete Wert größer oder kleiner als $V^\pi(s)$ ist. Wenn er größer ist, ist es besser einmal Aktion a im Zustand s auszuwählen und dann der Strategie π zu folgen, anstellen π die ganze Zeit zu verfolgen. Daraus folgend würde man erwarten, dass es immer besser ist a auszuwählen, wenn man sich im Zustand s befindet und dass die neue Strategie eine bessere wäre.

Die ist der Fall beim **policy improvement Theorem**, bei dem die oben genannte Annahme gilt und π und π' beliebige deterministische Strategien sind.

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \tag{9}$$

In diesen Fall muss die Strategie π' genauso gut oder besser als π sein und damit eine gleich große oder bessere Belohnung für alle Zustände einholen.

Optimale Bewertungsfunktionen
Zwei Strategien können durch ihre Bewertungsfunktionen vergleichen werden. Es gilt $\pi \geq \pi'$, wenn $V^\pi(s) \geq V^{\pi'}(s)$ für alle $s \in S$.

Die optimale Strategie verfügt über die optimale V-Funktion: $V^*(s) = max_\pi V^\pi(s)$, für alle $s \in S$

Außerdem über die optimale Q-Funktion: $Q^*(s,a) = max_\pi Q^\pi(s,a)$, für alle $s \in S$, $a \in A$

Q kann bezüglich V definiert werden: $Q^*(s,a) = E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\}$

Verallgemeinerung
Gegeben sind zwei Strategien π und π' und es gilt $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$ für alle $s \in S$
Das heißt: Es wird nur für einen Schritt nach π' vorgegangen, danach wieder nach π. Dann gilt für alle $s \in S$: $V^{\pi'}(s) \geq V^\pi(s)$!

Berechnung einer optimalen Strategie
Wiederholte Bewertung (Evaluation) einer gegebenen Strategie, dann Verbesserung (Improvement) $\Rightarrow$ Strategie-Iteration (**Policy Iteration**).

$$\pi_0 \xrightarrow{E} V^{\pi 0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi 1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^* \tag{10}$$

3.3 Policy Iteration

Bei der Policy Iteration wird eine vorhandene Strategie π bewertet (Evaluation) und verbessert (Improvement) um eine Strategie π' zu erhalten die besser ist. Dieses Schritte werden solange wiederholt, bis die Strategie stabil ist und sich die Werte in $\pi(s)$ nicht mehr verändern.

1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s') \right]$
   If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

Abbildung 3: Algorithmus zur Policy Iteration

3.4 Value Iteration

Betrachtet man die Policy Iteration stellt man fest, dass jeder Schritt die lange Bewertung (Evaluation) der Policy beinhaltet, in der eine iterative Berechnung über alle Zustände geschieht. Die iterativen Evaluation bricht erst ab, wenn es nach V^π konvergiert, doch so lange muss gar nicht gewartet werden. Die Policy Iteration, kann in vielen Fällen beschränkt werden, ohne die Konvergenz zu verlieren, welche durch die Policy Iteration garantiert wird. Ein Spezialfall ist das **Stoppen der Evaluation nach einem Durchgang** (alle Zustände wurden einmal gespeichert „backuped“), welcher als Value Iteration bezeichnet wird.

Hierbei wird die Backup-Operation mit dem Improvement der Policy kombiniert und man erhält die Formel:

$$V_{k+1}(s) = max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \tag{11}$$

Die Value Iteration kombiniert effektiv Evaluation und Improvement, der Algorithmus ist Abbildung 4 zu entnehmen.

Initialize V arbitrarily, e.g.,$V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

Abbildung 4: Algorithmus zur Value Iteration

3.5 Gemeinsamkeiten und Unterschiede von Policy und Value Iteration

Zuerst soll hier nochmal auf die Gemeinsamkeit von Policy und Value Iteration eingegangen werden. Da beides Verfahren der Dynamischen Programmierung sind, benötigen sie ein vollständiges Modell der Welt bzw. der Umgebung. Es müssen also die direkte Belohnung r und der Nachfolgezustand s_{t+1} (welcher durch eine Funktion $s_{t+1} = \delta(s, a)$ bestimmt wird) zur Durchführung der Policy Evaluation Phase (bzw. Berechnung der State-Value-Function V^π) für beiden Verfahren bekannt sein.

3.5.1 Konzeptionelle Unterschiede

Wie bereits erwähnt wird bei der Value Iteration entfällt das wiederholte optimieren der Strategie π bis zur Konvergenz der Strategie. Warum ist dies aber überhaupt möglich?

Hierzu muss das Konvergenzverhalten der State-Value-Function V_k und einer Policy π' betrachten. Dieses ist in der Abbildung 5 für eine Gridworld und eine Greedy Policy, welche es zu optimieren gilt, dargestellt. In der Abbildung wurde nach jedem Schritt in der Policy Evaluation die Strategie $\pi(s)$ für jedes $s \in S$ optimiert.
Hierbei fällt auf, das man bereits nach dem dritten Policy Evaluation Schritt eine optimale Strategy π^* erreicht hat. Da der Algorithmus für die Policy Iteration in der Policy Evaluation Phase (vgl. Abbildung 3) bis zur Konvergenz (also sehr oft) die State-Value-Function V berechnet, kann davon ausgegangen werden dass man bereits nach einer Policy Evaluation Phase die optimale Strategie π^* gefunden hat.
Auf Grund dieser Beobachtung kann die Policy Improvement Phase in der Value Iteration so optimiert werden, dass man für alle $s \in S$ nur noch die optimale Aktion a anhand der Zustands-Werte von s bestimmen muss. Dies wird im Algorithmus zur Value Iteration (vgl. Abbildung 4) mit der

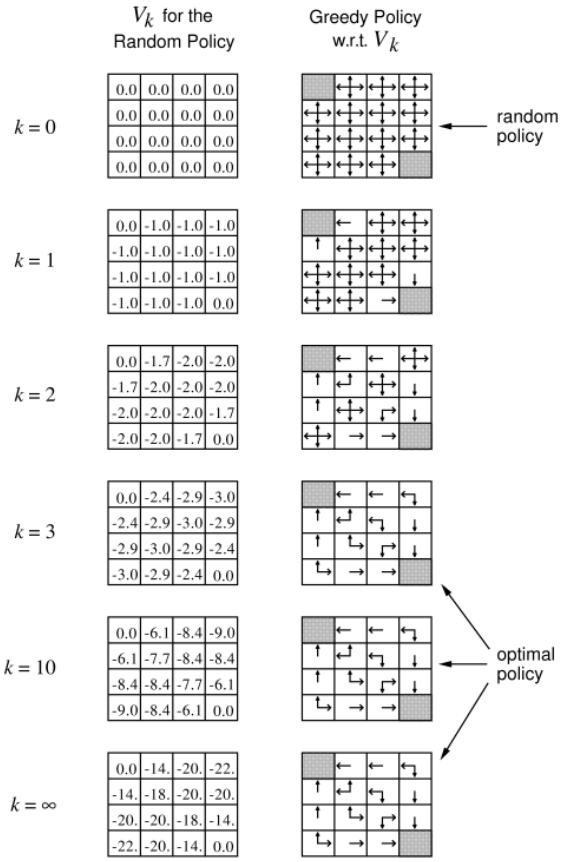

Abbildung 5: Zusammenhang zwischen der Konvergenz von V_k und der optimierten Policy $\pi^{'}$

Zeile $\pi(s) = argmax_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$ ausgedrückt.

Ein anderer Blickwinkel auf die Value Iteration führt über die Bellman-Optimalitäts-Gleichung $V^*(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$. Durch die Auswahl der optimalen Aktion $\max_a$ wird die Konvergenz in der Policy Evaluation Phase deutlich verbessert und es müssen weniger Iterationen (bessere Performance) gemacht werden.

Das Ausnutzen der beiden beschriebenen Verbesserungen überführen den Algorithmus der Policy Iteration hin zur Value Iteration.

3.6 Anwendung Policy und Value Iteration

In diesem Abschnitt soll anhand von Beispielen erklärt werden wie die beiden Algorithmen ihre Berechnung durchführen.

3.6.1 Beschreibung der Randbedingungen

- Zielzustand: unten rechts
- Aktionen: links, rechts, oben, unten
- Schrittkosten auf einen Nachbarzustand: -1
- Gegen Wand oder Barriere laufen: Position bleib unverändert und Kosten -2
- Discountfaktor: 0,8
- Umwelt ist deterministisch
- Initiale Situation: Zufallsstrategie, alle Werte auf 0

Es sind jeweils die nächsten beiden Iterationen der Verfahren zu berechnen.

3.6.2 Policy Iteration

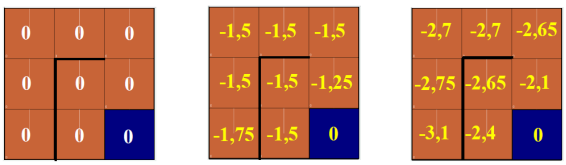

Abbildung 6: Ausgangssituation und die beiden folgenden Iterationen bei der Policy Iteration

Zur Berechnung der Zustands-Werte verwenden wir die folgende Formel aus der Policy Evaluation Phase:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \tag{12}$$

Im Folgendem wir die Berechnung der Werte für der ersten Iteration für einige Zustände beispielhaft durchgeführt. Hierzu werden die Zustände von links-oben nach rechts-unten und von Links nach Rechts durchnummeriert. Links-oben ist damit Zustand 1 und der Zielzustand ist Zustand

Nummer 9.

Berechnung Zustand 1:

$$s1^{'} = \frac{1}{4} * [(-2 + 0,8 * 0) + (-2 + 0,8 * 0) + (-1 + 0,8 * 0) + (-1 + 0,8 * 0) \tag{13}$$

$$s1^{'} = \frac{1}{4} * [-2 + -2 + -1 + -1] = -1,5 \tag{14}$$

Da bei der zugrundeliegenden Zufallsstrategie jede Aktion eine identische Wahrscheinlichkeit besitzt (alle $\frac{1}{4}$) kann die Wahrscheinlichkeit aus der Summe in der eckigen Klammer herausgezogen werden. Damit entspricht die $\frac{1}{4}$ dem Ausdruck $\sum_a \pi(s, a)$. Da es sich um eine deterministische Umwelt handelt, ist die Wahrscheinlichkeit für den Übergang von s nach $s^{'}$ ($P^a_{ss'}$) immer 1 und damit in der Berechnung nicht ersichtlich ($\frac{1}{4}$ würde mit 1 multipliziert werden).
Die Summe $\sum_a$ stellt das Auswerten aller Aktionen a. Jeder Ausdruck in den runden Klammern entspricht einer Aktion, die Reihenfolge in der vorgegangen wurde lautet: links, oben, rechts, unten.
Innerhalb der runden Klammer wird für jeder Aktion der direkte Reward $R^a_{ss'}$ und das Produkt aus Discountingfaktor und Wert des nächsten Zustands $\gamma V_k(s^{'})$ berechnet. Im Beispiel wird bei der Aktion „links“ ein Reward von -2 und und der Wert von $s1$, da wir mit der Aktion außerhalb der Umwelt landen und in $s1$ verbleiben, genommen. Ansonsten würde der Wert des Zustands $s^{'}$ genommen, dies ist z.B. der Fall für die Aktion „rechts“ im Zustand $s1$.

Für den Zustand 6 gestaltet sich die Berechnung wie folgt:

$$s6^{'} = \frac{1}{4} * [(-1 + 0,8 * 0) + (-1 + 0,8 * 0) + (-2 + 0,8 * 0) + (-1 + 0,8 * 0) \tag{15}$$

$$s6^{'} = \frac{1}{4} * [-1 + -1 + -2 + -1] = -1,25 \tag{16}$$

Für die zweite Iteration wird der Wert für den Zustand s6 mit den gerade berechneten Zustands-Werten ermittelt (full backup):

$$s6^{''} = \frac{1}{4} * [(-1 + 0,8 * -1,5) + (-1 + 0,8 * -1,5) + (-2 + 0,8 * -1,25) + (-1 + 0,8 * 0) \tag{17}$$

$$s6^{''} = \frac{1}{4} * [-2,2 + -2,2 + -3 + -1] = -2,1 \tag{18}$$

In der Policy Evaluation Phase würde man anschließen solange iterieren bis die Zustands-Werte konvergieren und dann mit der Policy Improvement Phase beginnen.
In ihr wird für alle Zustände $s \in S$ die optimale Aktion über $argmax_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s^{'})]$. Danach wird die verbesserte Policy $\pi(s)$ mit der vorherigen Policy b verglichen. Sollten sich noch Änderungen ergeben haben wird eine neue Iteration mit Policy Evaluation und Policy Improvement begonnen. Unterscheiden sich die beiden Policies nicht, so wurde die optimale Strategie gefunden.

Abschließen ein Beispiel zur Bestimmung der optimalen Aktion für einen Zustand. Zur Aktionsbestimmung wird die Gleichung $\pi(s) = argmax_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s^{'})]$ benutzt, welche unter allen Aktionen a im Zustand s diejenige Aktion mit dem größten Wert wählt. Hier am Beispiel

des Zustands $s6$ nach Iteration 2 durchgeführt:

$$\pi(s6) = argmax_{\{\text{links, oben, rechts, unten}\}} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \tag{19}$$

$$\pi(s6) = argmax\{-1 + 0,8 * -1, 8, -1 + 0,8 * -1, 8, -2 + 0,8 * -1, -1 + 0,8 * 0\} \tag{20}$$

$$\pi(s6) = argmax\{-2,44, -2,44, -2,8, -1\} \tag{21}$$

$$\pi(s6) = unten \tag{22}$$

Am Ergebnis erkennt man dass für den Zustand $s6$ bereits nach der zweiten Iteration die optimale Policy gefunden wurde. Diesen Schritt würde man in der Policy Improvement Phase für alle Zustände s durchführen und dann überprüfen ob sich Änderungen gegenüber der vorherigen Policy ergeben haben.

3.6.3 Value Iteration

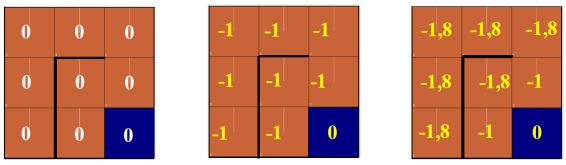

Abbildung 7: Ausgangssituation und die beiden folgenden Iterationen bei der Value Iteration

4 Monte Carlo Methoden

4.1 Grundidee

Bei den Monte Carlo Methoden besitzt man kein vollständiges Wissen über die Umgebung, sondern sammelt Erfahrungen durch tatsächliches oder simuliertes Interagieren mit der Umwelt.
Bei den Monte Carlo Methoden gibt es nur episodische Aufgaben, da die Belohnung sichergestellt sein muss. Die Abschätzung der Zustandswerte und Strategie-Änderungen geschieht erst am Ende einer Episode (Wert eines Zustandes = kumulierte Belohnung bis zum Ende der Episode). Am Ende wird der Durchschnitt über alle Episoden gebildet.

Vorteile gegenüber Dynamic Programming:

- Das optimale Verhalten kann direkt aus der Interaktion mit der Umgebung gelernt werden, ohne ein Modell der Umgebung vorhalten zu müssen.
- Sie können in Simulationen oder Beispiel-Modellen eingesetzt werden, man muss also kein komplettes Umgebungsmodell mit den Übergangswahrscheinlichkeiten erstellen.
- Es ist einfach und effizient sich mittels der Monte Carlo Methoden auf eine kleine Teilenge der Zustände zu fokussieren.

Trotz der Unterschiede zwischen Dynamic Programming und Monte Carlo Methoden, werden die wichtigsten Ideen vom Dynamic Programming übernommen. Es werden die gleichen Value-Functions berechnet und über die gleichen Methoden versucht, die optimale Funktion zu bestimmen. In der Evaluation werden V^π und Q^π für eine Policy π bestimmt und anschließen durch Improvement verbessert. Jeder dieser Schritte wird für die Monte Carlo Methoden übernommen, in dem nur beispielhafte Erfahrungen vorhanden sind, da die Umgebung unbekannt ist.

4.2 Abgrenzung: Monte-Carlo-Simulation

Verfahren aus der Stochastik in dem sehr häufig durchgeführte Zufallsexperimente die Basis darstellen. Ziel ist es analytisch nicht oder zu sehr aufwändig lösbare Probleme mit Hilfe der Wahrscheinlichkeitstheorie numerisch zu lösen. Die Grundlage bildet das Gesetz der Großen Zahlen, wobei computergenerierte Vorgänge den Prozess in ausreichend häufigen Zufallsereignissen simulieren können.

4.3 Policy Evaluation

4.3.1 Allgemeines

Wie bereits erwähnt ist der Wert eines Zustandes, der erwartete zukünftige kumulierte discounted reward, ausgehend vom betrachteten Zustand. Eine Möglichkeit diesen Wert aus Erfahrungen zu schätzen, ist es die erhaltenen Belohnungen zu mitteln, nachdem der Zustand besucht wurde. Je mehr Belohnungen für diesen Zustand beobachtet werden, desto näher sollte das Mittel zum erwarteten Wert konvergieren. Diese Idee verfolgen alle Monte Carlo Methoden.
Beide Verfahren First- und Every-visit MC konvergieren nach $V^{\pi}(s)$, wenn die Anzahl der Besuche ins unendliche geht (Gesetz der großen Zahlen).

4.3.2 Every-visit Monte Carlo

Die Every-visit MC Methode schätzt $V^{\pi}(s)$ über die Mittelwertbildung aller Belohnungen, die bei **jedem Besuch** vom Zustand s in einer Episode erhalten werden.

4.3.3 First-visit Monte Carlo

Innerhalb einer Episode wird der **erste Besuch** von s als first-visit bezeichnet und die First-visit MC Methode bildet den Mittelwert nur über die ersten Besuche von s innerhalb einer Episode.
Der Algorithmus wird in Abbildung 8 dargestellt. Es wird eine Episode für die Strategie π ausgeführt und für jeden Zustand s der Episode wird die Belohnung festgehalten, die dem ersten Besuch von s folgt. Danach werden die Rewards gemittelt und $V^{\pi}(s)$ zugewiesen.

4.4 Policy Improvement

4.4.1 Monte Carlo ES (Estimation of Action Values)

4.4.2 On-Policy Monte Carlo

On-Policy Verfahren versuchen die Strategie zu bewerten und verbessern die genutzt wird um Entscheidungen zu treffen.
Bei den den On-Policy Methoden gibt es nur eine Strategie, die über den nächsten Schritt entscheidet. Diese wird bewertet und verbessert. Hierbei besteht das **Control Problem** die optimale Strategie zu finden und das **Prediction Problem** die V-Werte oder Q-Werte für eine gegebene Strategie zu schätzen.

Initialize:
$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state-value function
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
(a) Generate an episode using π
(b) For each state s appearing in the episode:
$R \leftarrow$ return following the first occurrence of s
Append R to $Returns(s)$
$V(s) \leftarrow$ average($Returns(s)$)

Abbildung 8: First-visit MC Algorithmus zur Bestimmung von $V^{\pi}(s)$

4.4.3 Off-Policy Monte Carlo

Im Verglich zum On-Policy Verfahren mit dem charakteristischen Feature, dass der Wert einer Policy dadurch geschätzt wird, indem man sie zur Kontrolle verwendet, werden beim Off-Policy Verfahren diese beiden Funktionen getrennt.
Die Strategie die genutzt wird um Verhalten zu generieren wird **behaviour policy** genannt und ist evt. gar nicht abhängig von der Strategie welche bewertet (evaluated) und verbessert (improved) wird **estimation policy**. Vorteil dieser Trennung ist, dass die estimation Policy deterministisch sein kann, währen die behaviour policy weiterhin alle möglichen Aktionen ausprobieren kann.

5 Temporal-Difference Learning

5.1 Definition

Temporal Difference Learning ist eine **Vorhersagemethode**, welche hauptsächlich beim Reinforcement Learning eingesetzt wird und eine **Kombination aus Monte Carlo und Dynamic Programming** Ideen ist.
Es stellt eine Monte Carlo Methode dar, da es seine **Umgebung auf Basis einer Strategie erkundet** und nutzt Dynamic Programming Techniken indem es auf Basis der zuvor **gelernten Schätzungen**, seinen **aktuelle Schätzung approximiert**. Der Algorithmus ist dabei an das „temporal difference model“ tierischen Lernens angelehnt.

TD Learning vereint die Vorteile von Dynaminc Programming (Simulation des nächsten Schritts auf Basis des internen Modells) und Monte Carlo Methoden (es wird kein Modell benötigt). Jedoch besteht auch hier das **Control Problem** die optimale Strategie zu finden.

Als Vorhersagemethode nimmt Temporal Difference Learning es in Kauf, das **aufeinanderfolgende Vorhersagen häufig korrelieren**. Beim überwachten Lernen wird auf Basis vorhandener Beobachtungen gelernt und die Vorhersagen werden angepasst um bestimmte Beobachtungen genauer vorherzusagen zu können. Beim Temporal Difference Learning jedoch versucht man die Vorhersagen so anzupassen, dass diese genauere **Aussagen über Beobachtungen in der Zukunft** zulassen.

Mathematisch gesprochen wird bei beiden Varianten versucht eine Kostenfunktion im Hinblick auf den Fehler bei der Vorhersage einer Variable $E[z]$ zu optimieren. Beim Standardverfahren nehmen wir für $E[z] = z$ an, wobei z den aktuell beobachteten Wert darstellt. Beim Temporal Difference Learning wird hierfür ein Model verwendet, wobei z den totalen Gewinn darstellt und $E[z]$ über eine Bellman-Gleichung gegeben die das Ergebnis schätzt.

5.2 Vergleich TD Learning, DP und MC

Dynamic Programming betrachtet alle Zustände und benötigt ein Modell zur Bestimmung der Belohnungen und mit den Wahrscheinlichkeitsverteilungen für Folgezustände.
Monte Carlo Methoden und Temporal Difference Learning betrachten jeweils nur einen Pfad und benötigen kein Modell.
Bei Monte Carlo Methoden wird bis zum Ende einer Episode gewartet, bevor Informationen über die Belohnung erhalten werden, wobei beim Dynamic Programming und beim TD Learning nur ein Schritt gegangen werden muss (**Bootstrapping**).

5.3 On-Policy TD Control: Sarsa

Im ersten Schritt wird die „action-value-function“ $Q^\pi(s, a)$ für die aktuelle Strategie π und alle Zustände s und Aktionen a gelernt. Man erhält somit eine alternierende Sequenz aus Zuständen und Zustands-Aktions Paaren (state-action-pairs), die in Abbildung 9 dargestellt ist.

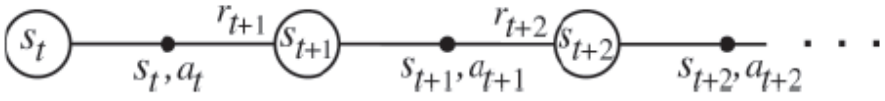

Abbildung 9: Sequenz aus states und state-action-pairs

Aus diesen Übergängen von state-action-pair zu state-action-pair werden die Belohnungen ermittelt, um den Wert der state-action-pairs zu bestimmen.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{23}$$

Der Wert α ist hierbei ein konstanter Parameter für die Schrittweite. Der Sarsa-Algorithmus ist ein On-Policy Control Algortihmus und wird in Abbildung 10 dargestellt.

5.4 Off-Policy TD Control: Q-Learning

Beim Off-Policy Verfahren approximiert die gelernte „action-value-function“ Q direkt dir optimale Funktion Q^* unabhängig davon welche Strategie verfolgt wird. Q-Learning ist ein Off-Policy TD Control Algorithmus und wird in Abbildung 11 dargestellt.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{24}$$

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
    $s \leftarrow s'$;
  until s is terminal

Abbildung 10: Sarsa: On Policy TD Control Algorithmus

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
    $s \leftarrow s'$;
  until s is terminal

Abbildung 11: Q Learning: Off Policy TD Control Algorithmus

5.5 Cliff-Anwendung

Die Cliff-Anwendung zeigt die Unterschiede zwischen Sarsa (On-Policy) und Q-Learning (Off-Policy) Methoden.
Der Agent soll vom Start zum Ziel gelangen und dabei der Klippe nicht zu nahe kommen. Der Agent kann sich nach oben, unten, links und rechts bewegen und wird mit einem Reward von -1 für alle Übergänge belohnt. Wenn er der Klippe jedoch zu nahe kommt, erhält er eine Belohnung von -100 und wird unmittelbar auf den Startpunkt zurück gesetzt (vgl. Abbildung 12).

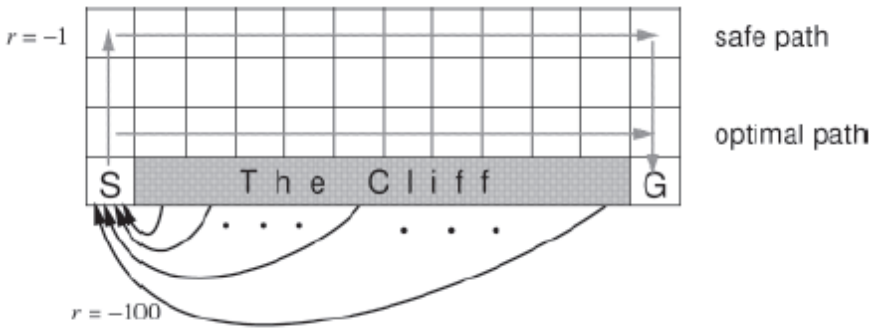

Abbildung 12: Cliff Anwendung

Die Ergebnisse sind Abbildung 13 zu entnehmen. Q Learning lernt die Werte für die optimale Strategie, wodurch der Agent nahe der Klippe entlangläuft und häufiger herunterfällt. Sarsa hin-

gegen berücksichtigt die Auswahl der Aktionen und lernt den längeren aber sichereren Pfad. Auch wenn Q Learning die optimale Strategie lernt, ist die Performance schlechter als die von Sarsa.

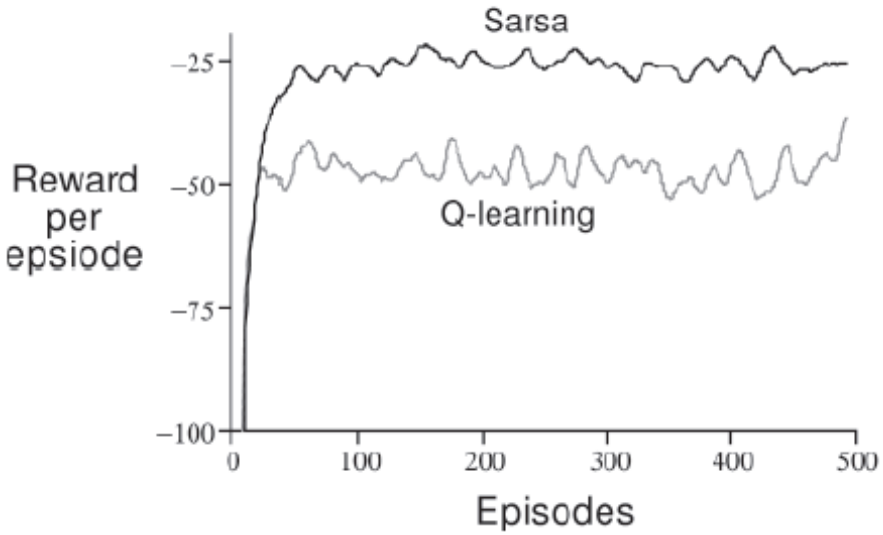

Abbildung 13: Vergleich Sarsa und Q-Learning

5.6 Trace Decay Factor

Der Trace Decay Factor λ ist ein „accumulating eligibility trace“ welcher angibt wie häufig ein Zustand besichtigt wurde (vgl. Abbildung 14). Wird ein Zustand weniger häufig besucht nimmt der Faktor über die Zeit hinweg ab.

5.7 Eligibility Traces

Werden in TD Learning Algorithmen verwendet und ist ein Konzept, um den in einem TD Schritt errechneten Fehler auch an vorhergehende Schritte weiterzureichen (vgl. Studienarbeit Markelic Kap 3.4.1).

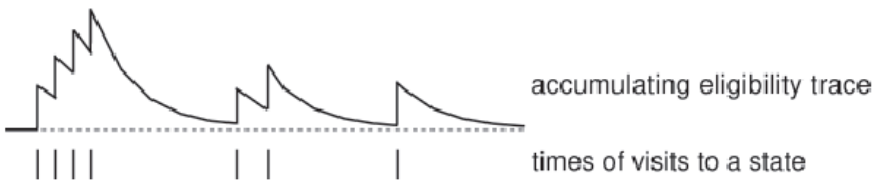

Abbildung 14: Trace Decay Factor