TT2 Problemstellung 1 : Zusammenfassung

Pascal Jäger, Stefan Münchow, Carsten Noetzel, Svend
Anjes Pahl, Milena Rötting, Oliver Steenbuck, Armin Steudte

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.

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).

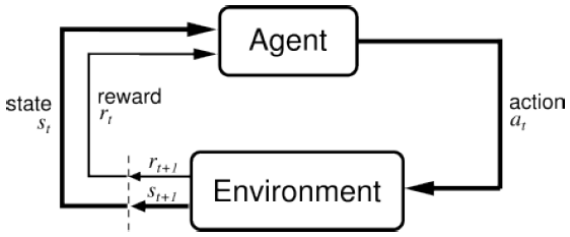

Abbildung 1: Agent in seiner Umgebung

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)

2.2.3 ε-greedy

Meistens wird die Aktion mit dem höchsten erwarteten Reward gewählt (greediest action), mit einer Wahrscheinlichkeit ε jedoch zufällig eine andere (unabhängig vom erwarteten Reward).

Wahrscheinlichkeitsvektor über die möglichen Aktionen in einem Zustand: $\vec{x} = (x_1, x_2, x_3, ..., x_n)$ mit n als der Anzahl möglicher Aktionen und x_i der Wahrscheinlichkeit, Aktion i zu wählen.

$$x_i = \begin{cases} (1-\varepsilon) + (\varepsilon/n), & \text{wenn } Q \text{ von } i \text{ am höchsten in } \vec{x} \text{ ist.} \\ \varepsilon/n, & \text{sonst.} \end{cases}$$

2.2.4 ε-soft

Meistens wird die Aktion mit dem höchsten erwarteten Reward gewählt, mit einer Wahrscheinlichkeit $1 - \varepsilon$ jedoch zufällig eine andere (unabhängig vom erwarteten Reward).

2.2.5 softmax

Jede Aktion wird anhand ihres zu erwarteten Rewards gewichtet. Es wird zufällig eine Aktion ausgewählt, allerdings mit Betrachtung des Gewichts, sodass die Aktion, die den schlechtesten Reward bringt auch am seltensten gewählt wird.

Dies unterbindet den Nachteil von ε-greedy bzw. ε-soft, dass, abgesehen von der besten Aktion, immer zufällig irgendeine Aktion gewählt wird. Bei diesen beiden Ansätzen könnte es sein, dass bei der zufälligen Wahl der Aktion immer die schlechteste gewählt wird.

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

Sieht das nur so aus oder kann ich mit diesem ganzen Wissen auch einen A^* machen ? OS

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** werden 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 $\pi^{'}$ 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:

$$V_{k+1}(s) = max_a \sum_{s'} P_{ss'}^{a} [R_{ss'}^{a} + \gamma V_k(s^{'})] \tag{11}$$

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

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

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

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, entfällt bei der Value Iteration das wiederholte Optimieren der Strategie π bis diese konvergiert. Aber warum kann man dies überhaupt wegfallen lassen?

Hierzu muss man 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, dass 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 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 Aktion mit der größten Belohnung $\max_a$ 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.

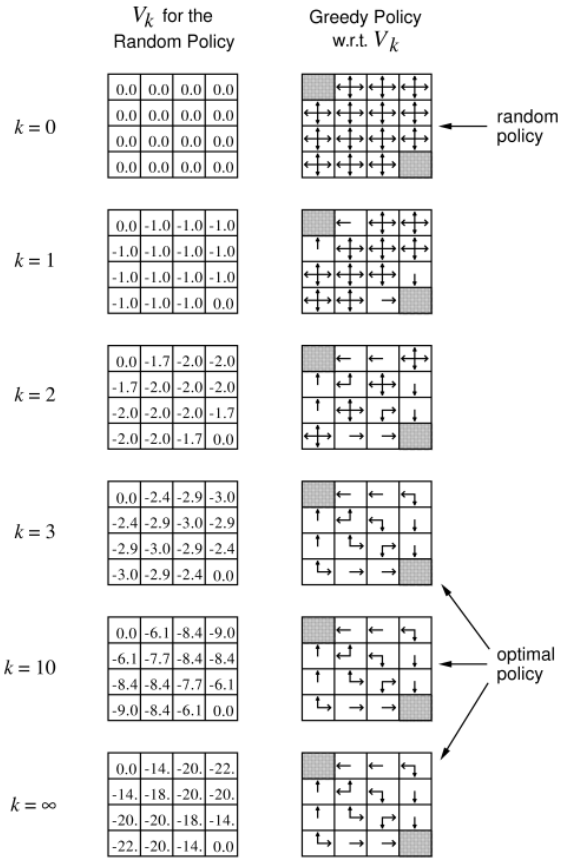

Abbildung 5: Zusammenhang zwischen der Konvergenz von V_k und der optimierten Policy $\pi^{'}$

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

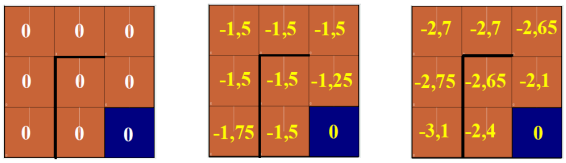

Abbildung 6: Ausgangssituation und die beiden folgenden Iterationen bei der Policy Iteration

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 dar. 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:

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

$$s6' = \frac{1}{4} * [-1 + -1 + -2 + -1] = -1,25 \quad (16)$$

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

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

$$s6'' = \frac{1}{4} * [-2,2 + -2,2 + -3 + -1] = -2,1 \quad (18)$$

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

Abschließend 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:

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

$$\pi(s6) = argmax\{-1 + 0,8 * -2,65; -1 + 0,8 * -2,65; -2 + 0,8 * -2,1; -1 + 0,8 * 0\} \quad (20)$$

$$\pi(s6) = argmax\{-3,12; -3,12; -3,68; -1\} \quad (21)$$

$$\pi(s6) = unten \quad (22)$$

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

Wie bereits im Abschnitt 3.5 erwähnt wurde, wird bei der Value Iteration in jedem Zustand die Aktion mit der größten Belohnung $V^*(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$ bestimmt und so eine

schnellere Konvergenz im Vergleich zur Policy Iteration erreicht.
In diesem Abschnitt wird dieses Verfahren, analog zu der in Abschnitt 3.6.2 beschriebenen Aufgabenstellung, beispielhaft für zwei aufeinander folgenden Iterationen erläutert. Die Ausgangssituation und die Ergebnisse sind in Abbildung 7 dargestellt. Zum Vergleich wird hier auch der selbe Zustand $s6$ wie bei der Value Iteration betrachtet.

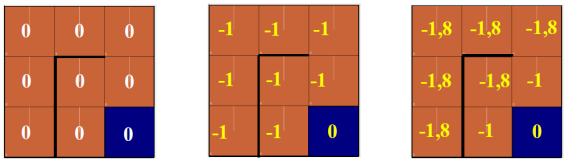

Abbildung 7: Ausgangssituation und die beiden folgenden Iterationen bei der Value Iteration

Zur Berechnung der Aktion mit der größten Belohnung gehen wir von der Formel $\underset{a}{max} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$ aus. Man betrachtet dabei in jeder Iteration alle Zustände und pro Zustand werden die direkten Belohnungen für alle Aktionen und die Zustandsbewertungen aus der vorherigen Iteration $V_k(s')$ betrachtet. Für die erste Iteration und Zustand $s6$ sieht das wie folgt aus:

$$\underset{a}{max} = \begin{cases} -1 + 0,8 * 0 & \text{links} \\ -1 + 0,8 * 0 & \text{oben} \\ -2 + 0,8 * 0 & \text{rechts} \\ -1 + 0,8 * 0 & \text{unten} \end{cases} \tag{23}$$

$$\underset{a}{max} = \begin{cases} -1 & \text{links} \\ -1 & \text{oben} \\ -2 & \text{rechts} \\ -1 & \text{unten} \end{cases} \tag{24}$$

$$\underset{a}{max} = \{link, oben, unten\} = -1 \tag{25}$$

Im Zustand $s6$ liefern die Aktionen $links, oben, unten$ die größte direkte Belohnung von -1 so dass $V_{k+1}(s) = -1$ ist. Für die zweite Iteration wird genau so vorgegangen:

$$\underset{a}{max} = \begin{cases} -1 + 0,8 * -1 & \text{links} \\ -1 + 0,8 * -1 & \text{oben} \\ -2 + 0,8 * -1 & \text{rechts} \\ -1 + 0,8 * 0 & \text{unten} \end{cases} \tag{26}$$

$$\underset{a}{max} = \begin{cases} -1,8 & \text{links} \\ -1,8 & \text{oben} \\ -2,8 & \text{rechts} \\ -1 & \text{unten} \end{cases} \tag{27}$$

$$\underset{a}{max} = \{unten\} = -1 \tag{28}$$

In diesem Fall ändert sich an dem für $V_{k+1}(s)$ berechneten Wert nichts, jedoch ist jetzt klar dass im Zustand $s6$ die Aktion *unten* die optimale Aktion mit der größten Belohnung ist. Dieses könnte auch nochmal mit der Formel $\pi(s6) = argmax_{\{\text{links, oben, rechts, unten}\}} \sum_{s'} P^a_{ss'}$ nach der Konvergenz der Zustands-Werte explizit berechnet werden.

Hier zum Vergleich, dass sich bei der Berechnung die Zustands-Werte auch ändern können, die beiden Iteration für den Zustand $s5$:

$$\max_{a} = \begin{cases} -2 + 0,8 * 0 & \text{links} \\ -2 + 0,8 * 0 & \text{oben} \\ -1 + 0,8 * 0 & \text{rechts} \\ -1 + 0,8 * 0 & \text{unten} \end{cases} \tag{29}$$

$$\max_{a} = \begin{cases} -2 & \text{links} \\ -2 & \text{oben} \\ -1 & \text{rechts} \\ -1 & \text{unten} \end{cases} \tag{30}$$

$$\max_{a} = \{rechts, unten\} = -1 \tag{31}$$

$$\max_{a} = \begin{cases} -2 + 0,8 * -1 & \text{links} \\ -2 + 0,8 * -1 & \text{oben} \\ -1 + 0,8 * -1 & \text{rechts} \\ -1 + 0,8 * -1 & \text{unten} \end{cases} \tag{32}$$

$$\max_{a} = \begin{cases} -2,8 & \text{links} \\ -2,8 & \text{oben} \\ -1,8 & \text{rechts} \\ -1,8 & \text{unten} \end{cases} \tag{33}$$

$$\max_{a} = \{rechts, unten\} = -1,8 \tag{34}$$

Auch in für diesen Zustand hat man nach der zweiten Iteration bereits die optimale Strategie gefunden. Im Zustand $s5$ ist es nämlich egal ob man *rechts* oder nach *unten* geht,so dass damit die optimalen Aktionen $rechts, unten$ sind.

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. Zudem 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 Teilmenge 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. Beim Dynamic Programming werden in der Evaluation 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 Schätzung der State-Values $V^\pi(s)$

Wie in den Abschnitten zuvor 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 man $V^\pi(s)$ 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 R festgehalten, die dem ersten Besuch von s folgt. R wird anschließend zu den aus vorherigen Episoden erhaltenen Rewards $Returns(s)$ addiert, woraufhin die Rewards gemittelt und $V^\pi(s)$ zugewiesen werden.

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 $V^\pi(s)$

4.3.4 Schätzung der Action-Values $Q^\pi(s, a)$

Beim Dynamic Programming haben wir ein komplettes Modell der Umwelt und können hinreichend anhand der State-values entscheiden, welche Policy die geeignetste ist, indem wir immer einen Schritt voraus schauen und die Aktion wählen, die uns zu der besten Kombination aus Reward und Folgezustand bringt.
Ohne dieses Modell benötigen wir zwingend die Schätzung der Aktionen um eine Policy bewerten zu können, deswegen ist eines der Hauptziele Q^* zu schätzen.

Das Problem hierbei ist alle möglichen $Q^\pi(s, a)$ zu schätzen. Die Methoden um einzelne $Q^\pi(s, a)$ Werte zu schätzen erfolgt genauso wie in den oben genannten First- und Every-visit Verfahren, wobei entweder das erste Auftreten des Zustandes s mit Auswahl der Aktion a oder jedes Auftreten der Kombination s und a über alle Episoden gemittelt wird. Auch hier konvergieren die Werte mit größer werdender Anzahl der Episoden.
Das Problem hierbei ist, dass möglicherweise nicht alle Kombinationen aus Zustand s und Aktion a besucht werden. **Wenn π eine deterministische Strategie ist, dann wird man nur jeweils eine Aktion pro Zustand beobachten** und die Erfahrung für andere Aktionen in dem entsprechenden Zustand wird sich nicht verbessern, da diese nicht betrachtet werden.
Dies ist ein großes Problem, da es der Zweck der Action-value-function $Q^\pi(s, a)$ ist, in einem Zustand die richtige Aktion auswählen zu können. Um die verschiedenen Aktionen jedoch vergleichen zu können, müssen mit **allen Aktionen Erfahrungen gesammelt** worden sein.

Dieses **„maintaining exploration“ Problem**, dass für die Nutzung der Action-values die kontinuierliche Exploration sichergestellt sein muss, kann dadurch gelöst werden, dass jeder erste Schritt der Episode an einem zufälligen State-Action-Pair startet und jedes State-Action-Pair eine Wahrscheinlichkeit ≥ 0 hat ausgewählt zu werden. Diese Annahme wird **Exploring starts** genannt.

4.4 Policy Improvement

4.4.1 Generalized Policy Iteration

Die Idee hinter der Generalized Policy Iteration (GPI) ist die gleiche wie beim Dynamic Programming, nur dass hier anstelle der State-Value-Function V die Action-Value-funtion Q verwendet wird.
Man hält jeweils eine approximierte Policy π und Action-Value-funtion Q vor. Die Action-Value-funtion wird immer wieder angepasst, sodass sie die Value-function der Policy besser approximiert und die Policy wird immer wieder auf Basis der Action-Value-funtion verbessert (vgl. Abbildung 9).

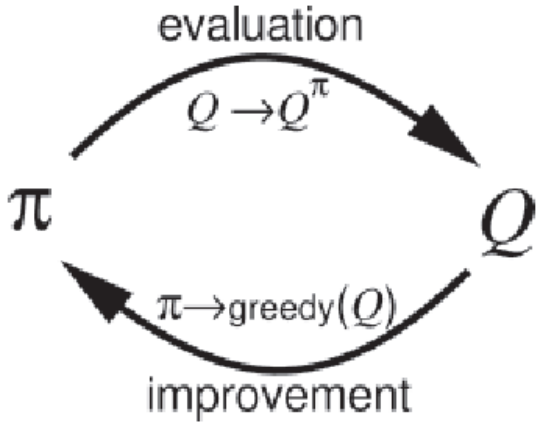

Abbildung 9: Generalized Policy Iteration

4.4.2 Monte Carlo ES (Exploring Starts)

Für die Monte Carlo Methoden ist es natürlich, dass die Bewertung (evaluation) und Verbesserung (improvement) zwischen den Episoden stattfindet. Nach jeder Episode werden die beobachteten Returns genutzt um die Strategie zu bewerten und danach wird die Strategie für alle besuchten Zustände in der Episode verbessert. Der Algorithmus hierzu wird in Abbildung 10 gezeigt.

Der Algorithmus startet damit, dass $Q(s, a)$ und die Policy π zufällig gewählt werden (exploring starts), sodass alle möglichen Kombinationen von s und a untersucht werden. Ähnlich wie beim First-visit MC in Abbildung 8, wird der Return des ersten Auftretens von Zustand s und Aktion a gespeichert und zu den bereits über alle Episoden beobachteten Belohnungen $Returns(s, a)$ addiert. Es folgt die Bildung des Mittelwertes für $Q(s, a)$ unabhängig davon welche Policy zum Zeitpunkt der Beobachtung aktiv war. Damit ist die Evaluation (b) beendet und es beginnt das Improvement (c) bei dem für einen Zustand s die Aktion in der Policy π hinterlegt wird, die den größten Wert $Q(s, a)$ hat.

4.4.3 On-Policy Monte Carlo

Beim zuvor vorgestellten Ansatz „MC mit Exploring Starts“ wurde mittels zufälliger Wahl verschiedener Anfangszustände s und Aktionen a versucht, alle möglichen Kombinationen abzudecken. Dies ist aber eine unrealistische Annahme die so in der Realität nicht erfüllt werden kann. Der einzige Weg um sicherzustellen, dass alle Aktion unendlich häufig getestet werden ist der, dass

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list

Repeat forever:
(a) Generate an episode using exploring starts and π
(b) For each pair s,a appearing in the episode:
$R \leftarrow$ return following the first occurrence of s,a
Append R to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

Abbildung 10: Monte Carlo Exploring Starts Algorithmus

der Agent diese auch weiterhin nutzt (nicht nur beim Start).
Es gibt zwei Ansätze die dies sicherstellen, die On-Policy und Off-Policy Verfahren, von denen hier zunächst das erste vorgestellt wird.

On-Policy Verfahren versuchen die Strategie zu bewerten und zu 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 **Policy ist generell soft** was bedeutet, dass $\pi(s,a) > 0$ für alle $s \in S$ und $a \in A(s)$ (alle Kombinationen aus Zustand und Aktion sind vorhanden). In dem hier vorgestellten Algorithmus unter Abbildung 11 kommt eine **ϵ-greedy Policy** zu Einsatz, die die meiste Zeit diejenige Aktion auswählt, die den größten erwarteten Gewinn verspricht und mit einer Wahrscheinlichkeit ϵ eine zufällige Aktion auswählt. Dadurch erhält man einen **Nichtdeterminismus bei der Auswahl von Aktionen**, wodurch alle Aktionen a für einen Zustand s gewählt werden können. ϵ-greedy Strategien sind dabei ein Spezialfall von **ϵ-soft Strategien** für die gilt $\pi(s,a) \geq \frac{\epsilon}{|A(s)|}$ für alle $s \in S$ und $a \in A(s)$ mit $\epsilon > 0$ (hierbei steht $\frac{\epsilon}{|A(s)|}$ für die Wahrscheinlichkeit aller Aktionen außer der günstigsten im Zustand s). Die günstigste Aktion erhält die Restwahrscheinlichkeit $1 - \epsilon + \frac{\epsilon}{|A(s)|}$.

Die Grundidee vom On-Policy Monte Carlo ist die gleiche wie bei der Generalized Policy Iteration (GPI). Es wird die First-visit MC Methode genutzt um die Action-value-function die Policy zu bestimmen, jedoch kann man nicht einfach die Policy auf Grundlage der Action-value-function verbessern, da dies dazu führen würde, dass nicht mehr alle Zustände exploriert würden. **Deswegen wird die Policy anstatt sie zu einer greedy Policy zu machen nur zu einer ϵ-greedy Policy gemacht.** Für jede ϵ-soft Policy π ist garaniert, dass jede ϵ-greedy Policy auf Basis von Q^π besser oder gleich π ist (sichergestellt durch das Policy Improvement Theorem).

4.4.4 Off-Policy Monte Carlo

Es sein angenommen, dass V^π und Q^π geschätzt werden sollen, aber nur Episoden der Policy π' vorhanden sind, wobei gilt $\pi' \neq \pi$. Das Lernen einer Value-function für eine Policy π auf Basis der Erfahrungen, die durch eine andere Policy π' gesammelt wurden nennt man Off-Policy Verfahren.

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s,a appearing in the episode:
$R \leftarrow$ return following the first occurrence of s,a
Append R to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:
$$\pi(s,a) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

Abbildung 11: ϵ-soft On-Policy Monte Carlo Algorithmus

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.
Der Algorithmus folgt dabei der behaviour policy und verbessert die estimation Policy. Um alle Möglichkeiten zu untersuchen muss, die **behaviour policy soft** sein. Abbildung 12 zeigt einen Off-Policy Monte Carlo Algorithmus.

Die behaviour policy π' ist eine willkürliche soft policy. Die estimation policy π ist greedy auf Basis von Q und eine Schätzung von Q^π.

5 Temporal-Difference Learning

5.1 Motivation

Dynamic Programming (siehe: 3) und Monte Carlo Methoden (siehe: 4) weisen Einschränkungen auf die im praktischen Einsatz die Größe der lösbaren Probleme stark einschränken. Dies sind im einzelnen:

DP Die Menge der Zustände ist bekannt und endlich

DP Die Menge der möglichen Aktionen {a} ist bekannt und endlich

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$N(s,a) \leftarrow 0$; Numerator and
$D(s,a) \leftarrow 0$; Denominator of $Q(s,a)$
$\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
(a) Select a policy π' and use it to generate an episode:
$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
(b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
(c) For each pair s, a appearing in the episode after τ:
$t \leftarrow$ the time of first occurrence (after τ) of s, a
$w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
$N(s,a) \leftarrow N(s,a) + wR_t$
$D(s,a) \leftarrow D(s,a) + w$
$Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
(d) For each $s \in \mathcal{S}$:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

Abbildung 12: Off-Policy Monte Carlo Algorithmus

DP Die Belohnungen für alle Zustände müßen bekannt sein

DP Die Folgezustände in Abhängigkeit von Ausgangszustand und Aktion müßen bekannt sein

DP Die Menge der Zustände muß bekannt und endlich sein

MC Das Problem muß episodisch sein

MC Wissen aus vorhergehenden Episoden wird nicht verwendet (Rechenaufwand)

Temporal Difference Learning verbindet Konzepte der Dynamischen Programmierung und der Monte Carlo Methoden um die Nachteile zu vermeiden und die Vorteile (s.u.) der jeweiligen Methode beizubehalten.

MC Zustandraum kann zur Laufzeit erforscht werden

DP Verteilung der Belohnungen auf die korrekten, die "verantwortlichen" Zustände

5.2 Arbeitsweise Umgangssprachlich

Der Gedank bei Temporal Difference Learning alle Erwartungswerte für rewards beliebig zu füllen und dann beim durchlaufen eine Pfades die auf diesem liegenden Werte anhand ihrer der Abweichung $real - erwartet$ anzugleichen.

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 13 dargestellt ist.

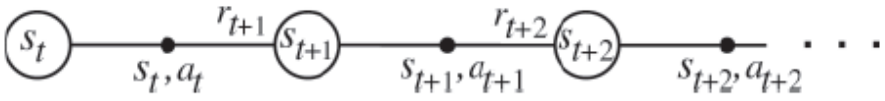

Abbildung 13: Sequenz aus states und state-action-pairs

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

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{35}$$

Der Wert α ist hierbei ein konstanter Parameter für die Schrittweite. Der Sarsa-Algorithmus ist ein On-Policy Control Algortihmus und wird in Abbildung 14 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 15 dargestellt.

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

Abbildung 14: Sarsa: On Policy TD Control Algorithmus

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{36}$$

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 15: 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 16).

Die Ergebnisse sind Abbildung 17 zu entnehmen. Q Learning lernt die Werte für die optimale Strategie, wodurch der Agent nahe der Klippe entlangläuft und häufiger herunterfällt. Sarsa hingegen 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.

check: Auf der cliff Abbildung 16 sieht der optimale weg aus wie ca. -14 in Abbildung 17 kommt keiner näher als ca. -25 ?

check: Oben steht das Q Learning den optimalen Pfad lernt, Q Lear-

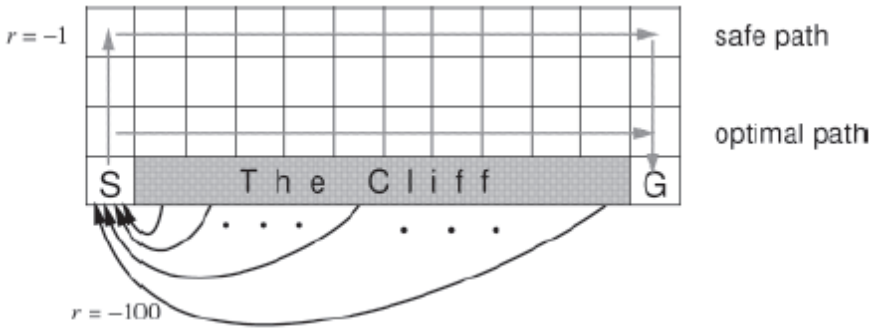

Abbildung 16: Cliff Anwendung

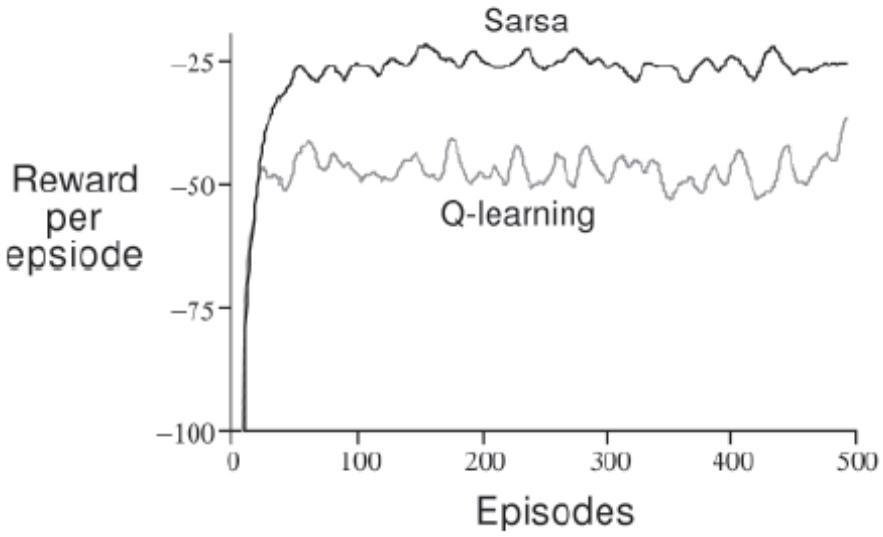

Abbildung 17: Vergleich Sarsa und Q-Learning

5.6 Trace Decay Factor

5.6.1 Zusammenfassung Umgangssprachlich

Der Trace decay faktor gibt an mit welcher Stärke eine in Zustand s_t beobachtete Abweichung der real gemessen Rewards von den erwarteten Rewards $V(s_t)$ in die erwarteten Rewards der Zustände über die zu s gelangt wurde eingerechnet wird. Hierzu wird ein Trace Decay Factor λ für jeden Zustand geführt der steigt wenn ein Zustand besucht wird und sonst für diesen abnimmt und in der Regel einen Multiplikator ≤ 1 darstellt. Mit λ multipliziert reduziert sich also der in s_t gefunden Fehler für die Zustände s_{t-n} (solche über die zu s_t gelangt wurde) in der Regel wird der Fehler umso mehr abgeschwächt umsoweiter s_{t-n} in der Vergangenheit liegt..

5.6.2 Referenzen/Bilder

Der Trace Decay Factor λ ist ein „accumulating eligibility trace“ welcher angibt wie häufig ein Zustand besichtigt wurde (vgl. Abbildung 18). 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).

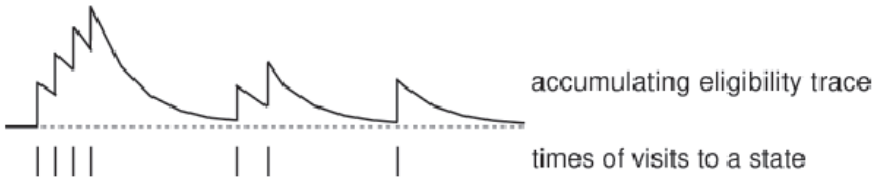

Abbildung 18: Trace Decay Factor

5.8 Vergleich der Verfahren

Drei drei vorher vorgestellten Verfahren dynamische Programmierung, Monte-Carlo-Methoden und Temporal Difference Learning werden in diesem Abschnitt nochmal im Hinblick auf ihre Vor- und Nachteile verglichen.

5.8.1 Dynamic Programming

Dynamische Programmierung berechnet optimale Strategien auf Basis eines vollständigen Modells der Umgebung. Die Zustände und Zustandsübergänge der Umgebung müssen dabei einen Markov-Entscheidungsprozess darstellen.

Voraussetzungen

- Die Menge der Umweltzustände ist bekannt und ist endlich

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

Vor- und Nachteile

+ Findet garantiert eine optimale Strategie in polynomialer Zeit
+ Funktioniert bei heutigen Computern mit Millionen von Zuständen
– Es wird ein vollständiges Modell der Umweltzustände, Aktionen und Belohnungen benötigt
– Hoher Rechenaufwand
– Nur für diskrete Zustandsräume verwendbar
– Sehr empfindlich gegenüber Verletzungen der Markov-Eigenschaft, da neue Schätzungen auf vorherigen Schätzungen basieren (*Bootstrapping*)

5.8.2 Monte-Carlo-Methoden

Monte Carlo Methoden betrachten jeweils nur eine Sequenz von Zustandsübergängen (eine *Experience*) und benötigt daher kein Modell.

Check: Kein Modell oder ein kleineres ? Wir erstellen doch beim TD eigentlich auch ein Modell, wie ist hier die Abgrenzung ? OS

Voraussetzungen

- Die zu lösende Aufgabe ist episodisch

Vor- und Nachteile

+ Kein Modell der Umgebung erforderlich. Stattdessen werden nur Episoden (Sequenzen aus Zuständen, Aktionen und Belohnungen) benötigt, die durch Interaktion mit der Umgebung generiert werden. Diese Episoden können durch einfache Modelle generiert werden, die häufig wesentlich einfacher zu finden sind, als vollständige Modelle der Umwelt mit expliziten Transitionswahrscheinlichkeiten
+ Robuster als DP gegenüber Verletzungen der Markov-Eigenschaft. Dies liegt daran, dass neue Schätzungen nicht auf vorherigen Schätzungen basieren (kein *Bootstrapping*)
+ Monte-Carlo-Methoden fokussieren sehr gut auf eine kleine Teilmenge der Umweltzustände
+ Garantierte (asymptotische) Konvergenz zur optimalen Strategie
– Nur für episodische Tasks geeignet, da die Belohnung immer erst am Ende einer Episode generiert wird
– Ausreichende Exploration ist bei deterministischen Strategien ein Problem, da keine alternativen Aktionen getestet werden. Dies kann z.B. durch den Start mit zufälligen Zustands-Aktions-Paaren (sog. *Exploring Starts*) verhindert werden

5.8.3 Temporal Difference Learning

Temporal Difference Learning vereint Ideen der dynamischen Programmierung und der Monte-Carlo-Methoden. Es verwendet *Bootstrapping* wie Monte-Carlo-Methoden, benötigt jedoch kein

Modell der Umwelt wie die dynamische Programmierung.

Voraussetzungen
Keine?

Ich sehe hier keine Vorraussetzungen - jemand von euch? SM

Vor- und Nachteile

+ Lernen direkt durch Erfahrung, kein Modell der Umwelt mit Zuständen, Zustandsübergängen und Belohnungen notwendig
+ Inkrementell wie dynamische Programmierung: Belohnungen nach jedem Schritt, nicht wie bei Monte-Carlo-Methoden erst nach einer Episode. Dadurch meist schneller, vor allem bei langen Episoden
+ Auch für nicht-episodische Aufgaben geeignet
+ Garantierte (asymptotische) Konvergenz zur optimalen Strategie. Konvergiert erfahrungsgemäß häufig schneller als Monte-Carlo-Methoden (einen formalen Beweis dafür gibt es allerdings nicht)
+ Einfachheit: Wenig Rechenaufwand, es werden nur einzelne Gleichungen benötigt, dadurch kleine Programme
− Empfindlich gegenüber Verletzungen der Markov-Eigenschaft, da aktuelle Schätzungen auf vorherigen Schätzungen basieren (*Bootstrapping*; Kann durch *Eligibility Traces* behoben werden)

*Todo list

- Sieht das nur so aus oder kann ich mit diesem ganzen Wissen auch einen A^* machen ? OS 8
- check: Auf der cliff Abbildung 16 sieht der optimale weg aus wie ca. -14 in Abbildung 17 kommt keiner näher als ca. -25 ? . . . 25
- check: Oben steht das Q Learning den optimalen Pfad lernt, Q Learning hat in Abbildung 17 aber anscheinend einen schlechteren Reward als Sarsa das den sicherreren Pfad lernen soll ? . . . 25
- Check: Kein Modell oder ein kleineres ? Wir erstellen doch beim TD eigentlich auch ein Modell, wie ist hier die Abgrenzung ? OS . . . 28
- Ich sehe hier keine Vorraussetzungen - jemand von euch? SM . . . 29