

# Reinforcement Learning (Machine Learning I) (1)

Definition: Mapping von Situationen auf Aktionen, Maximierung eines numerischen Belohnungssignals (Ziel)

Eigenschaft: RL betrachtet stets ein ganzes Problem (nie Subprobleme) eines zielgerichteten Agenten der mit unbekannter Umgebung interagiert

Subelemente: policy, reward function, value function, model of environment

$a_t \in A(s_t)$   
wobei  $A(s_t)$   
die Menge der  
Aktion verfügbar  
in State  $s_t$   
+ Mapping von  
States nach Wahrsch.,  
für das Wählen  
jeder möglichen  
Aktion  
 $\rightarrow$  die W. dass  
ein Agent eine Aktion  
auswählt

Policy: definiert das Verhalten eines Agenten zu beliebigem Zeitpunkt. Mapping zwischen beobachteten Zuständen der Umgebung zu Aktionen wenn in einer dieser Zustände (Psychologie: Stimulus Antwort)  
z.B. simple Funktion oder Lookup-Table  
 $\pi_t(s, a)$  ist die Wahrsch., dass  $a_t = a$  wenn  $s_t = s$

Reward function: das Ziel im RL-Problem. Sie mappt den beobachteten Zustand (oder State-Action-Paar) der Umgebung einer Nummer (Reward) zu (Indikator für intrinsische Attraktivität)  
 $\hookrightarrow$  Ziel eines RL-Agenten ist es alleinig den gesamten Reward den er erhält zu maximieren  
 $\rightarrow$  Action gewählt auf Basis einer Policy und wenig reward  $\Rightarrow$  Policy-Änderung, sodass andere Action gewählt

Value function: Reward function beschreibt eine gute Wahl einer Action in einem kurzfristigen Sinne. Value functions beschreiben dies langfristig. Der Value eines States bezeichnet den gesamten Reward den ein Agent erwarten kann, wenn er diesen State wählt. Sie beschreibt damit, welche rewards erwartet werden können bzw. wie hoch rewards in der Zukunft ausfallen können.

Aktionen werden eher auf Basis von values als auf rewards gewählt, weil dadurch States von höchsten Values folgen (und nicht von höchsten rewards)  $\rightarrow$  damit

Model: Abbild einer Umgebung, immittiert das Verhalten einer Umgebung. z.B., kann das Gegeben ein State und Action, das Model den resultierenden nächsten State und Reward schätzen

(Exploration-  
Exploitation-  
Dilemma)

Exploration ist profitabel, weil es ~~für die~~ Trainingsinformation die gewählten Aktionen bewahrt und diese Bewertung für das Training benutzt. Es wird dann ~~lehrte nicht~~ nicht vorgetragen, instruiert welche korrekten Aktionen

## Exploration-exploitation Dilemma

Trainingsdaten werden genutzt um gewählte Aktionen zu bewerten anstatt dem Lernern zu instruieren indem korrekte Aktionen zu instruieren

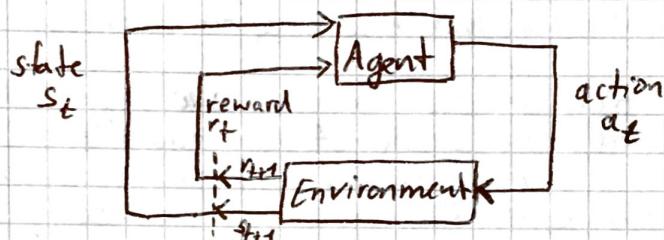
→ erfordert Exploration, weil durch reine Bewertung nicht entscheidbar ist, ob „gut“ eine Aktionswahl ist (also best/schlechte) (gibt nur Feedback wie gut gerade Action war - kein Vergleich/Verbesserung möglich ohne Exploration)

Ob gewählte Aktion best/schlechteste war  
→ Verbesserung nur durch Exploration,

$\epsilon$ -greedy Methode: mit Wahrscheinlichkeit  $\epsilon$ , wähle zufällig eine Aktion aus, die nicht die beste (also greedy) ist

mehr Rauschen/Varianz im Reward: wähle  $\epsilon$  größer  
Rauschen nul: wähle  $\epsilon = 0$  (full greedy)

## Environment Interface of an agent



## Policy

$\pi_t(s, a)$  ist die Wahrscheinlichkeit, dass  $a_t = a$  wenn  $s_t = s$  für  $a_t \in A(s_t)$ , wobei  $A(s_t)$  die Menge von actions ist, & verfügbar in state  $s_t$  und  $s_t \in S$  für  $S$  die Menge verfügbarer states

Zu jedem Zeitpunkt implementiert ein Agent ein Mapping von States zu Wahrscheinlichkeiten für die Selektion möglicher Actions.

## Reward Function (mit discount factor)

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

$\gamma$  Bewertung bzw. Gewichtung der Belohnung  $0 \leq \gamma \leq 1$

$< 1$ : Agent ist „kurzsichtig“, nur aktuelle Belohnungen wichtig und Agent befasst sich nur mit Maximierung direkter Belohnungen

$> 0$ : Agent wird mit Nähierung an  $\gamma = 1$  zunehmend „weltblickiger“, zukünftige Belohnungen fallen fließen stärker ein als  $\gamma$  nahe 0

$< 1$ : damit die Funktion konvergiert

Ein Zustand der erfolgreich alle relevanten Informationen beibehält, wird „Markov“ genannt (auch Markov-Eigenschaft)  
Formal:

$$\Pr \{ s_{t+1} = s' | r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0 \} \\ = \Pr \{ s_{t+1} = s' | r_{t+1} = r | s_t, a_t \}$$

\* die z.B. notwendig sind für nächste Entscheidung die dafür alle wichtigen Infos braucht (z.B. Damenhörspiel aber keine Historie)

## Markov Property

# Reinforcement Learning (Machine learning) (2)

## Markov Decision Process (MDP, deterministisch/stochastisch)

Je nachdem ob der Zustandsübergang  $S(s, a)$  deterministisch oder stochastisch ist, spricht man von "deterministic MDP" oder "non-deterministic MDP". Hier: stochastisch

### Transition Probability

charakteristisch - diskreter, zeit-stochastischer Zufallsprozess  
- State-Übergänge erfüllen Markov-Eigenschaft  
- modelliert Entscheidungsfindung

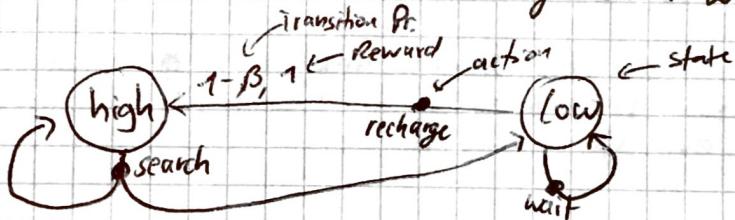
$$P_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

(die W. dass action a im state s zum Zeitpunkt t zum Zustand  $s'$  führt)

Da eine W-Verteilung wird der Erwartungswert eingesetzt für die Belohnung:

$$R_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

(der Erwartungswert der nächsten Belohnung wenn von state s über eine action a in state  $s'$  gewechselt wird)



### (state-)Value Function

(Bewertung eines States in welchem sich ein Agent befindet bzw. Aktionsbewertung zu einer gegebenen State)

-> hier Bewertung anhand zu erwartender zukünftiger Belohnungen

Einschub: policy  $\pi$  mappt für jeden State  $s \in S$  und Action  $a \in A(s)$  zur Wahrsch.  $\pi(s, a)$ , dass Action a umgesetzt wird im State s.

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

„Wert“ eines State s unter Anwendung von Policy  $\pi$

Der „Wert“ von State s unter einer Policy  $\pi$ , gekennzeichnet  $V^\pi(s)$ , ist der erwartete Ertrag wenn in s begonnen und  $\pi$  danach verfolgt wird.

$E_\pi\{\}$  beschreibt Erwartungswert unter Anwendung der Policy  $\pi$

## Action-) value Function

$$Q^{\pi}(s, a) = E_{\pi} \{ R_t | s_t = s, a_t = a \}$$

$$= E_{\pi} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}$$

„Wert“ der Wahl einer Action  $a$  in State  $s$  unter Anwendung einer Policy  $\pi$

## Unterschied $V^{\pi}$ und $Q^{\pi}$

$V^{\pi}$ : Wenn ein Agent policy  $\pi$  befolgt und einen Mittelwert pflegt für je einen State, dann konvergiert dieser Mittelwert gegen den State-Value  $V^{\pi}(s)$  mit zunehmender Anzahl, die dieser State „passiert“ wurde.

$Q^{\pi}$ : Wenn ein Agent verschiedene Mittelwerte für jede in einem State gewählte Action pflegt, dann konvergiert dieser Mittelwert genauso gegen den Action-Value  $Q^{\pi}(s, a)$ .

(→ Monte Carlo Methoden)

## Bellman equation für $V^{\pi}$

Beschreibt, dass der Value des Startzustands ( $s$ ) gleichbedeutend zum diskontierten Value des erwarteten nächsten State ( $s'$ ) ist, plus der erwartete Reward auf dem Weg dorthin.

$$V^{\pi}(s) = E_{\pi} \{ R_t | s_t = s \}$$

$$= E_{\pi} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\}$$

(Buch S. 70)

$$= E_{\pi} \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right\}$$

$$= \underbrace{\sum_a \pi(s, a)}_{\text{Policy, probab.}} \underbrace{\sum_{s'} P_{ss'} [R_{ss'} + \gamma V^{\pi}(s')]}_{\text{Transition that actions charm probabilities given } s}$$

given  $s$

Die Bellman equation mittelt über alle Actions, gewichtet mit ihren Wahrsch., dass sie gewählt werden (⇒ Reward pro action wird gewichtet)

=> rekursive Beziehung:

$$V^{\pi}(s) = x + V^{\pi}(s')$$

## Optimale Value Functions (\* = optimale Funktion)

$$V^* = \max_{\pi} V^{\pi}(s) \quad \text{und} \quad Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \quad \text{und} \quad V^*(s) = \max_{a'} Q^*(s, a')$$

$Q^*$  berechnet den erwarteten Rückgabewert für Anwendung action  $a$  unter state  $s$ , daher kann man auch schreiben (in Abhängigkeit von  $V$ ):

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

## Reinforcement Learning (Machine Learning) (3)

Weil ~~opt~~  $V^*$  optimale Value Function ist, kann man  $V^*$  ohne Policy  $\pi$  beschreiben  $\Rightarrow$  Bellman optimality equation.

Intuitiv: der Value eines States unter einer optimalen Policy ist gleich dem Erwartungswert für die beste Action ausgehend von diesem State:

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s, a) \\ &= \max_a \mathbb{E}_{\pi^*} \{ R_t | s_t = s, a_t = a \} \\ &= \max_a \sum_{s'} P_{ss'}^a [R_{ss'} + \gamma V^*(s')] \end{aligned}$$

$\Rightarrow$  jede Policy die „greedy“ ist in Bezug einer optimalen Value Funktion  $V^*$  ist eine optimale Policy

(S.78) Value Function muss „one-step-ahead“-Suche durchführen, um beste Action zu erhalten, während Action-Value Function (Q) dies nicht muss - sie sucht für jeden State  $s$  einfach die Action, die  $Q^*(s, a)$  maximiert (Q speichert effektiv gesehen die Ergebnisse aller „one-step-ahead“-Suchen für State-Action-Paare)

Finden einer optimalen Policy:

- 1) Lösen der Bellman Optimality Equation Gleichung  
→ selten genutzt, bzw. möglich in Praxis, weil
  - a) Dynamik der Umgebung nicht genau bekannt
  - b) Berechnen der Gleichung mit Computer zeitintensiv (z.B. Budget von  $10^{20}$  States)
  - c) Markov-Eigenschaft nicht gegeben

2) Approximation der Bellman Optimality Gleichung

- a) Heuristische Suchmethoden, z.B. A\*  
→ Idee: expandieren von  $\min_a \sum_{s'} P_{ss'}^a [R_{ss'} + \gamma V^*(s')]$  habe Wärsch.-Bäum bis Tiefe  $x$ , nutze andere Heuristik um  $V^*$  zu approximieren indem Blätter durchsucht werden

- all use some variation  
of generalized policy iteration  
(GPI), different in their  
approaches to prediction problem
- b) Dynamic Programming (↓ require complete env. model)
  - c) Monte-Carlo-Methode (↓ not suited for step-by-step increm. computation)
  - d) Temporal-Difference Learning (↓ analysis of TD complex)
- Vorteil RL vs. andere Methoden MDP zu lösen

Die On-Line-Natur von RL erlaubt es, optimale Policies zu approximieren, bei sodass mehr Aufwand in das Wählen guter Entscheidungen für häufig eintreffende States anstatt gelegt wird, anstatt viel Aufwand in wenig vorkommende States zu legen.

E-greedy: die meiste Zeit wählt eine Policy eine Action mit maximalen Erwartungswert, doch mit Wahrscheinlichkeit  $\epsilon$  wählt sie eine nicht-maximale Action aus (Exploration)

## Temporal-Difference Learning $\rightarrow$ modellfrei!

(Wie Monte Carlo Methoden) TD nutzt Erfahrung für das Lösen des „Prediction-Problems“  $\rightarrow$  gegeben Erfahrung nach Befolgen einer Policy  $\pi$ ,

TD updatest die Schätzung  $V$  von  $V^\pi$

### Update-Regel TD(0) (je Zeitschritt)

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

~~getrennt~~

Einschub: Update basierend auf Schätzung aller nächsten States  $\rightarrow$  „Bootstrapping“  
weil  $V^\pi(s_{t+1})$  unbekannt, nutzt stattdessen Schätzung für Anpassung  
 $\rightarrow$  Schätzung auf Basis von Schätzung (S. 137)

### Algorithmus TD(0)

Initialisiere  $V(s)$  zufällig,  $\pi$  zur <sup>zu</sup> bewertenden Policy  
Wiederhole (für jede Episode)

Initialisiere  $s$

Wiederhole (für jeden Schritt der Episode)

$a \leftarrow$  Aktion gegeben durch  $\pi$  für  $s$

Wende  $a$  an; beobachte Reward ( $r$ ) und nächster Zustand ( $s'$ )

$$V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$$

$s \leftarrow s'$

Bis  $s =$  Terminalzustand

„Temporal-Difference“ bezieht sich darauf, dass <sup>der</sup> Fehler beim Schätzen der Values proportional zur Änderung über die Zeit der Schätzung ist.  
Agent bekommt Update, um näher ans Ziel zu kommen.

### Unterschied zu Dynamic Programming + Monte Carlo Methode

$$\text{Monte Carlo: } V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$

Update von  $V(s_t)$  erst nach Ende der Episode (erst dann  $R_t$  bekannt)  
 $\rightarrow R_t$  wird „gesampled“  $\rightarrow$  d.h. es werden Actions gewählt und der Reward beobachtet  
 $\rightarrow$  kein Bootstrapping

DP:

Modell des environments ist bekannt, unbekannt ist aber die Policy  $\pi$ , damit auch  $V^\pi(s_{t+1})$ . Es wird nicht gesampled, weil Rewards durch das vorhandene Modell bekannt sind (Rewards müssen nicht entdeckt werden)  $\Rightarrow V(s) \leftarrow \dots V(s')$   
Aber Bootstrapping; denn  $V(s) \leftarrow \dots V(s')$

$\rightarrow$  TD(0) kombiniert Bootstrapping von DP mit Sampling von Monte Carlo

### Batch-Updating von TD(0)

Batch-Update-Phase ist relativ (siehe Kurvenzug verdeckt) zwar werden für jeden Zeitschritt  $t$  die Inkrementen ( $\alpha [\dots]$ ) berechnet über die Value Funktion  $V(s)$  wird nur einmal mit der Summe aller Inkrementen aktualisiert (nach Erreichen eines Terminal-States)  $\rightarrow$  Monte Carlo Sampling Methode ähnlich ( $R_t$ )

Bootstrapping: updating estimates on the basis of other estimates, e.g.  $V(s) \leftarrow \dots V(s_{t+1})$

Sampling: update looks ahead to a sample successor state (to observe Reward)  
e.g. Monte Carlo samples entire episode to estimate  $R$ , whereas

DP uses no samples because rewards and successor values known!

Die Algos implementieren nur langfristig eine „Schätzung“, indem sie  $V(s)$  approximieren. In jedem Zeitschritt wird eher ein Lookahead ausgeführt um Reward zu beobachten.

# Reinforcement Learning (Machine Learning I) (4)

## Sarsa - Algorithmus (temporal-difference)

- On-policy Lernalgorithmus (Update auf Basis von angewendeten Actions und Beobachtung der Ursache-Wirkung) Anwendung einer policy
- es wird statt State-Value-Funktion wie bei TD(0) eine Action-Value-Funktion gelernt, d.h.  $Q(s, a)$  statt  $V(s)$
- berücksichtigt werden also State-Action-Paare statt reine State-Werte (formell aber gleichbedeutend da Markovketten mit Reward beide sind)
- Update durch den Agenten nach jeder Interaktion (Transition zum nächsten State) mit der Umgebung, Update-Regel und Algorithmus:

Initialisiere  $Q(s, a)$  zufällig/beliebig

Wiederhole für jede Episode:

Initialisiere  $s$

Wähle  $a$  von  $s$  mithilfe Policy abgeleitet von  $Q$  (z.B.  $\epsilon$ -greedy)

Wiederhole (für jeden Schritt der Episode):

Wende  $a$  an, beobachte  $r$  und  $s'$

Wähle  $a'$  von  $s'$  mithilfe Policy abgeleitet von  $Q$  (z.B.  $\epsilon$ -greedy)

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

$S \leftarrow s'$ ;  $a \leftarrow a'$ ;

**SARSA**

solange  $s$  nicht Terminalzustand

nicht

## Q-Learning-Algorithmus (temporal-difference)

- Off-policy Lernalgorithmus (wegen max-Term; der Algorithmus wicht von der  $\epsilon$ -greedy Policy ab, indem er die greedy action  $a$  wählt geben  $s'$  und damit seine  $Q$ -Werte updatebt)
- Wie bei Sarsa wird auch hier Action-Value-Funktion eingesetzt ( $Q(s, a)$ ), nur approximiert  $Q$  direkt  $Q^*$  (optimale Fkt.) unabhängig der eingesetzten Policy (z.B.  $\epsilon$ -greedy) zur Auswahl  $a$  und  $s$ .  
→ eingesetzte Policy hat trotzdem Wirkung: bestimmt, welche State-Action Paare berücksichtigt und geupdated werden → erlaubt Exploration ( $\epsilon$ -greedy) und Exploitation (max über  $Q^*$ ) wird dazu genutzt  $Q^*$  zu approximieren
- Sarsa lernt action values relativ zu seiner verfolgten Policy, während Q-Learning dies relativ zur greedy policy tut, Algorithmus:
- Nach  $n$  Iterationen gilt:  $Q \leq Q^*$

Initialisiere  $Q(s, a)$  beliebig

Wiederhole für jede Episode:

Initialisiere  $s$

Wiederhole (für jeden Schritt der Episode):

Wähle  $a$  von  $s$  mithilfe Policy abgeleitet von  $Q$  (z.B.  $\epsilon$ -greedy)

Wende  $a$  an, beobachte  $r$  und  $s'$

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

$S \leftarrow s'$ ;

solange  $s$  nicht Terminal

**Vorteile:** konvergiert zur optimalen Policy schneller als Sarsa, gut bei off-line-Lern

**Nachteile:** schlechter als Sarsa beim On-line-Lerning (z.B. lernt Sarsa beim Cliff Walking die sicherere Policy → weniger Cliff-falls)

→  $\epsilon$ -Anpassung führt Sarsa aber auch zur optimalen Policy

→ Q-L. weniger anfällig für schlecht gewähltes  $\epsilon$

Verbesserung  $Q \leftarrow \dots$

In-line Learning  
training on incoming data  
spam detection learning  
on incoming emails

Off-line Learning  
training on static and available dataset  
(no time constraints also)

On-Line bei RL  
incremental  $\Delta V$  werden  
bei Updates sofort  
eingesetzt

Off-Line bei RL  
incremental  $\Delta V$  werden  
gesammelt und erst bei  
Episodenende für update  
verwendet

- Sarsa hatte hier  $a < a'$   
aber Q-L. nicht  
→  $\epsilon$ -greedy soll entscheiden  
(policy)  
welcher state als nächstes  
besucht wird

## Optimierungsmöglichkeit von Q-Learning

1) In jeder Lernepisode alle  $Q(s,a)$  vom Startzustand  $s$  zum Ziel

1) In jeder Lernepisode vom Startzustand  $s$  zum Ziel  $s_T$

- Propagiere Anpassungen „unkerer States“ ( $Q(s',a')$ ) hoch als „obere States“ ( $Q(s,a)$ ), notwendig: Pfad der Lernepisode muss Rewards speichern
- schnellere Konvergenz  $Q \rightarrow Q^*$  über Spektralabstand steigt

2) Q-Learning mit adaptivem Modell

- meisten Lernschritte am Modell  $\Delta Q$
- wenige Aktionen in realer Umgebung
- Anpassung des Modells

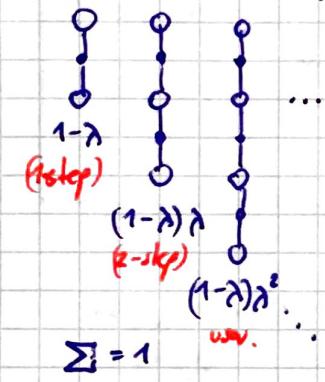
## Lernen von Aktionssequenzen

warum?

- bei langen Episoden kann erst spät gelernt werden
- nachfolgende Aktionen können schlecht sein

## Vorwärtsicht von TD( $\lambda$ )

TD( $\lambda$ )-Backup-Diagramm

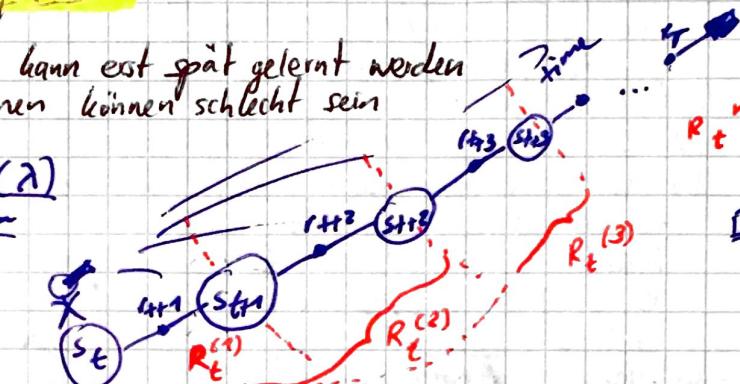


$\lambda$ -return Algo:

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

$$\Delta V_t(s_t) = \alpha [R_t^\lambda - V_t(s_t)]$$

Backup using  $\lambda$ -Return



- für jeden besuchten State werden zukünftige Rewards betrachtet u. kombiniert
- Update von Values mit:

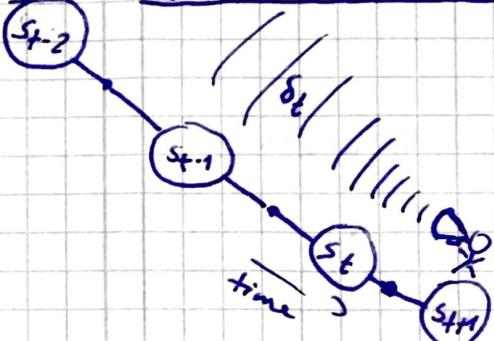
- a) nachfolgender Schätzung (1-step oder TD(0)) (wie bisher)
- b)  $n$  nachfolgenden Schätzungen ( $n$ -step)
- Parameter  $\lambda$  ist dazu zwischen 1-step ( $\lambda=0$ ) und Monte-Carlo-Methode ( $\lambda=1$ ) zu wechseln

-  $\lambda$  bestimmt wie schnell die exponentiellen Gewichte fallen und damit, wie weit in der Zukunft Rewards einfließen ( $\lambda^{n-1} R_t^{(n)}$ )

- TD( $\lambda$ ) bzw. Return nur möglich bei off-line Learning weil zukünftige Inputs unbekannt und bei off-line am Ende der Episode geupdated

# Reinforcement Learning (Machine Learning I) (5)

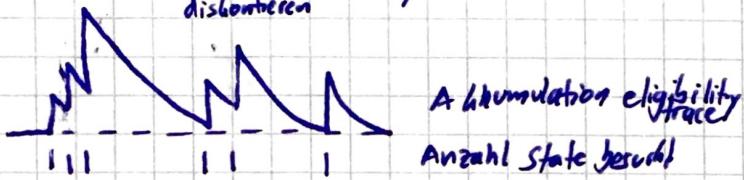
## Rückwärtssicht von TD( $\lambda$ )



- Voraussicht ist nicht direkt implementierbar, weil causal (Update benötigt Inputs aus der Zukunft)
- Rückwärtssicht ist causal und approximiert Voraussicht (On-Line) bzw. ist mathematisch gleich (Off-Line)
- „Speicher“ für jeden State: „eligibility trace“  
→ akkumulierender Trace, weil jedes Mal wenn wenn  $s \neq s_t$  State besucht, den Trace akkumuliert und wenn  $s = s_t$  wenn nicht besucht, Trace schrumpft

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) \\ \gamma \lambda e_{t-1}(s) + 1 \end{cases}$$

↑ trace decay  
diskontieren



„eligible“ = geeignet

## Algorithmus TD( $\lambda$ )

Initialisiere  $V(s)$  beliebig und  $e(s) = 0 \forall s \in S$   
Wiederhole für jede Episode:

Initialisiere  $s$

Wiederhole (für jeden Schritt der Episode):

$a \leftarrow$  Aktion gegeben von  $\pi$  für  $s$

Wende  $a$  an, beobachte  $r, s'$

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

$e(s) \leftarrow e(s) + 1$

Für alle  $s$ :

$V(s) \leftarrow V(s) + \alpha \delta e(s)$

$e(s) \leftarrow \gamma \lambda e(s)$

$s \leftarrow s'$

bis  $s$  Terminal ist

- Fehler  $\delta$  (temporal difference) wird an bisher passierte States zurückpropagiert
- Einfluss auf bisher passierte States wird anhand eligibility trace bestimmt (häufig besuchte States erhalten deutliches Update)
- wenn  $\lambda = 0$ , sind alle traces = 0  
→ ergibt TD(0), weil nur  $s_t$  und  $s_{t+1}$  einfließen
- wenn  $0 < \lambda < 1$ , aber mit zunehmend größerem  $\lambda$  mehr der bisherigen States geändert  
→ große Distanz = wenig Änderung

## Sarsa( $\lambda$ )

- Wie können eligibility traces nicht nur für Schätzung sondern auch Steuerung verwendet werden?

→ Action-Values statt State-Values lernen → Sarsa und eligibility traces kombiniert

- Trace nun von jedem State-Action Paar anstatt für jeden State  
↳  $Q(s, a)$  statt  $V(s)$  und  $e_t(s, a)$  statt  $e_t(s)$

$$e_t(s, a) = \begin{cases} \gamma \lambda e_{t-1}(s, a) + 1 & \text{wenn } s = s_t \text{ und } a = a_t \\ \gamma \lambda e_{t-1}(s, a) & \text{sonst} \end{cases}$$

Algo

## Sarsa ( $\lambda$ )

Initialisiere  $Q(s, a)$  beliebig und  $e(s, a) = 0 \quad \forall s, a$   
Wiederhole (für jede Episode)

Initialisiere  $s, a$

Wiederhole (für jeden Schritt der Episode):

Wende  $a$  an, beobachte  $r, s'$

Wähle  $a'$  von  $s'$  mit Policy abgeleitet von  $Q$  (z.B.  $\epsilon$ -greedy)

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

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

for all  $s, a$ :

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$$

$$e(s, a) \leftarrow \gamma \lambda e(s, a)$$

$$s \leftarrow s'; a \leftarrow a'$$

bis  $s$  Terminal ist

Eligibility Traces für Steveralgorithmen (wie Sarsa ( $\lambda$ ) oder  $Q(s, a)$ -Algos) erlauben höhere