Reinforcement Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 6

Deep Reinforcement Learning

TD & Monte-Carlo

Temporal-Difference (TD)

Unsere Quality Update Methode ist ein Beispiel von **Temporal-Difference (TD) lernen**,
da bei jedem Update die **Differenz** zwischen zwei Schätzungen verwendet wird:

QUALITY UPDATE

$q(a') = q(a \mid s) + [r + \gamma \max_a q(s',a) - q(a \mid s)]$

$q'(a \mid s) = q(a \mid s) + \alpha * [r + \gamma \mathbf{\max_{a'} q(s',a') - q(a \mid s)}]$
α learning rate: wie viel wollen wir unsere Schätzung updaten

Q-Learning (off-policy) : wende die Formel als pre-training an.

SARSA (on-policy) : wende die selbe Formel mehrfach pro episode (oder per replay) an:

$q\_target = r + \gamma \max_{a'} q(s',a')$
$q'(a) \mathrel{+}= \alpha * [\mathbf{q\_target - q(a \mid s)}]$

Temporal-Difference (TD) learning bezeichnet eine ganze Klasse von ähnlichen Ansätzen, in welchen in der Update Formel die zeitliche **Differenz zwischen zwei Schätzungen** vorkommt.

All Q-learning is TD learning, but not all TD learning is Q-learning.

Temporal-Difference (TD)

QUALITY UPDATE

best_reward := **$\max_{a'}$ q(s',a')**
target := **current_reward** + **γ** * best_reward
target = **r** + γ **$\max_{a'}$ q(s',a')**
estimate := q(a | s)

q'(a | s) = q(a | s) + **α** * [r + γ **$\max_{a'}$ q(s',a') - q(a | s)**]
q'(a | s) = q(a | s) + **α** * [**target - estimate**]
α learning rate: wie stark viel wollen wir unsere Schätzung updaten?

TD = Ein Zeitschritt (s, a) => (s', a')

SARSA = state action reward state action

Monte-Carlo

Monte-Carlo-Methoden lernen Wertfunktionen und optimale Policies aus Erfahrungen in Form von **Beispiel-Episoden**.

Replay = Trajektorie (ganzes Spiel)

Episoden Replay : kann man später (wieder)verenden (z.B. zum Trainieren) "*off* policy"

Bei **unvollständigem Modell** der Umgebung
(Simulierte) Erfahrung allein kann zu optimaler Policy führen!

In überraschend vielen Fällen ist es **einfach, Erfahrungen zu erzeugen**, die gemäß den gewünschten Wahrscheinlichkeitsverteilungen gezogen wurden, aber es ist meist *nicht möglich, die Verteilungen in expliziter Form* zu erhalten.

Frische Daten werden immer rarer mit jeder skalierung von **LLMs**, um so wichtiger, eigene Daten zu generieren.

Replay

Episoden Replay Buffer: speichert den gesamten Spielverlauf (Episode)

replay = [(s,a,r,s',a')] bei SARSA unabhängig

replay = [(s,a,r), (s',a',r')] bei Monte Carlo abhängig! (s' hängt von s ab)

replay = [(s,a,r), (s',a',r')] oder nennt sich 'episode' oder '**Trajektorie**'

replay = [(s,a,r,s')] mit next state je nach Algorithmus

Episoden Replay : kann man später (wieder)verwenden (z.B. zum Trainieren) "off policy"

Erinnert sehr stark an **batches** aus ML!

Sehr gut um Rauschen und Fehler in den Date '*auszugleichen*' (Regularisierung)

Replay vs Batch

replay = [(s,a,r), (s',a',r')] eine Episode oder

replay = [(s,a,r,s')] mit next state je nach Algorithmus

Erinnert sehr stark an **batches** aus ML!
Aber *Unterschied*:
batch = [(s,a,r)] **unabhängige** *samples*!

Kann man auch **verbinden**:
mehrere replays in ein batch: trainiere anhand von mehreren Spielverläufen.

Monte-Carlo

Monte-Carlo-Methoden lernen Wertfunktionen und optimale Policies aus Erfahrungen in Form von **Beispiel-Episoden**.

Bei unvollständigem Modell der Umgebung
(Simulierte) Erfahrung allein kann zu optimaler Policy führen!

In überraschend vielen Fällen ist es **einfach, Erfahrungen zu erzeugen**, die gemäß den gewünschten Wahrscheinlichkeitsverteilungen gezogen wurden, aber es ist meist *nicht möglich, die Verteilungen in expliziter Form* zu erhalten.

Dies verschafft ihnen gegenüber den DP-Methoden **Vorteile**:

- können verwendet werden, um optimales Verhalten direkt durch Interaktion mit der Umgebung zu erlernen,
- ohne ein Modell der Umgebungsdynamik zu benötigen.
- einfach und effizient, Monte-Carlo-Methoden auf eine kleine Teilmenge der Zustände zu konzentrieren.
    - Ein besonders interessanter Bereich kann genau bewertet werden, ohne die Kosten einer genauen Bewertung des gesamten Zustandsraums auf sich zu nehmen.
- können mit Simulationen oder Beispielmodellen eingesetzt werden.
    - Für überraschend viele Anwendungen ist es einfach, Beispiel-Episoden zu simulieren, obwohl es schwierig ist, die Art von explizitem Modell der Übergangswahrscheinlichkeiten zu erstellen, die von DP-Methoden verlangt wird.

Aspekte von Algorithmen

So viele Algorithmen und Namen, je nach Situation und Ansatz.

In der Praxis kann man oft wählen zwischen verschiedenen Modi:

model based y/n?
sampling y/n? (Episode play vs Bellman)
on policy y/n?
Bootstrapping y/n? Q(a)=0
Approximation given / exact / **tabellarisch** / linear / **deep**
Exploration **ε-greedy** / normed probs / softmax / …
quality-based / **v**alue-based / **p**olicy-based / actor-critic
episodenbasiert / Schrittweise (MC vs. TD)
deterministisch / stochastisch
Planning ja / nein simulierte Rollouts

	Uses samples	Requires model
Monte Carlo	✔	✘
TD (Q-learning)	✔	✘
Expected SARSA	✘ (for the update)	✔
Value Iteration	✘	✔
Model-based RL	✔ (for model learning) and ✘ (for planning)	✔

Off policy learning

Trainer policy

Ansatz:
Statt e-greedy kann man auch mit einer ganz anderen policy trainieren.
Mit gewisser W-Keit wird die trainer policy ausgewählt
Das ist dann ganz klar "**off policy**"
Motivation: Trainer policy ist nicht immer optimal, kann verbessert werden
Problem: es gibt keine **eigene** Exploration.
Andere policy liefert Daten die zu weit vom 'Kenntnisstand' entfernt sind.
Lösung: Mit e-greedy mischen.

"off policy":
gerade trainierte policy
≠ policy zum Daten sammeln

meist Zwischenlösung 'relativ off policy'

	Uses samples	Requires model
Monte Carlo	✔	✘
TD (Q-learning)	✔	✘
Expected SARSA	✘ (for the update)	✔
Value Iteration	✘	✔
Model-based RL	✔ (for model learning) and ✘ (for planning)	✔

Exploring Starts

"Full sweep"

Ein beliebter Ansatz bei Monte Carlo ist das System vor zu trainieren indem man von **allen denkbaren Startpositionen** beginnt, Episoden zu samplen.

Setzt **vollständiges wissen** von **p** also Bekanntheit unserer Umgebung voraus!

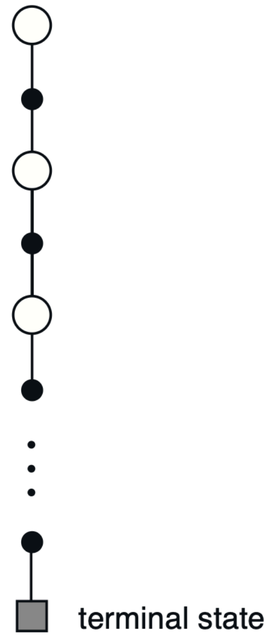
Ziel: vor dem durchspielen der Episoden schon **gute Schätzungen** erhalten.

Exploring Starts ist theoretisch **stärker als ε-greedy**, weil es garantiert, dass alle (s, a)-Paare beprobt werden können, während ε-greedy möglicherweise einige (s, a)-Paare nicht erreicht, wenn die Dynamik der Umgebung den Zugang einschränkt!

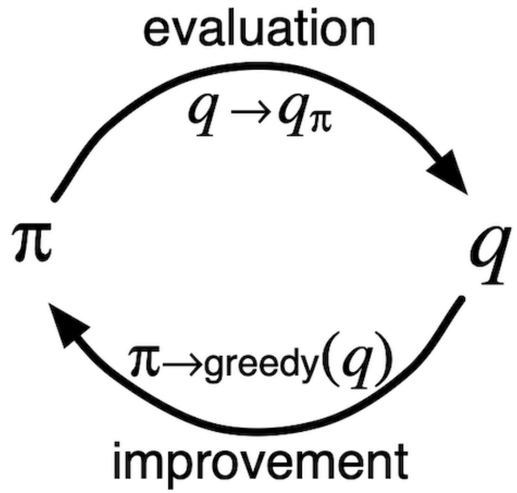
Monte-Carlo

Initialisiere, für alle s∈S, a∈A(s):
Q(s,a) ← 0 bootstrapping
π(s) ← random
Returns(s,a) ← []

while not done:
Wähle s0 ∈S and a0∈A(S0) so dass alle Paare probability >0
Generiere eine Episode beginnend von S0,A0, folge **π**
Für jedes Paar **s, a** das in der **Trajektorie** vorkommt:
G←Rückgabe nach dem ersten Auftreten von s,a
Append G in **Returns**(s,a)
Q(s,a) ←Mittelwert von Returns(s,a)
For each s in episode:
π(s) ←argmax a Q(s,a) "greedy" einfach immer beste Aktion wählen.



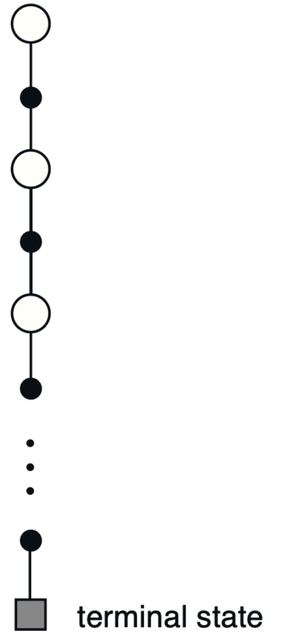
backup diagram for Monte Carlo estimation of v_π.



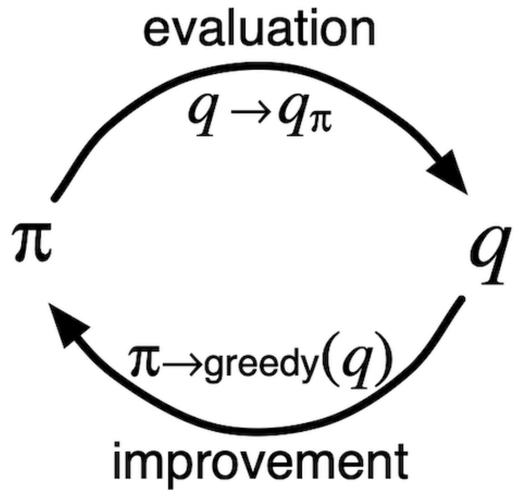
Monte-Carlo

Initialize, for all s∈S, a∈A(s):
  Q(s,a) ←arbitrary
  π(s) ←arbitrary
  Returns(s,a) ←empty list

Repeat forever:
  Choose S0 ∈S and A0 ∈A(S0) s.t. all pairs have probability >0
  Generate an episode starting from S0,A0, following π
  For each pair s,a appearing in the episode:
    G←return following the first occurrence of s,a
    Append G to Returns(s,a)
    Q(s,a) ←average(Returns(s,a))
  For each s in the episode:
    π(s) ←argmax a Q(s,a)



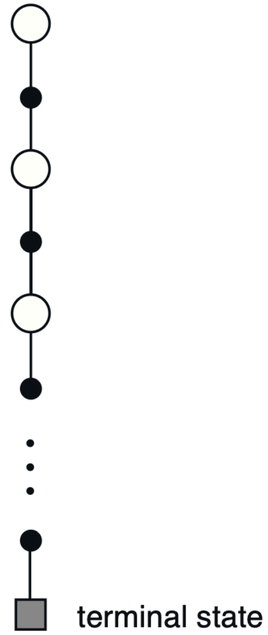
backup diagram for Monte Carlo estimation of v_π.



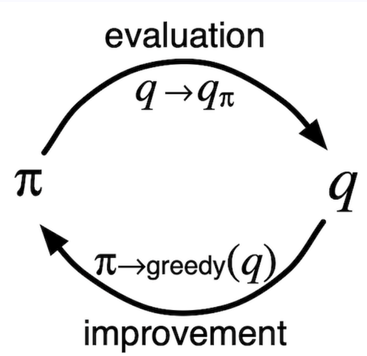
Monte-Carlo

Typischerweise wird Trajektorie rückwärts abgearbeitet:

```
visited = set()
G = 0
for t in reversed(range(len(episode_data))):
    state, action, reward = episode_data[t]
    G = reward + DISCOUNT_FACTOR * G   rekursiv
    if (state, action) not in visited:
        visited.add((state, action))
        Gains[state][action].append(G)
        Q[state, action] = np.mean(Gains[state][action])
        Policy[state] = np.argmax(Q[state])
```



backup diagram for Monte Carlo estimation of v_π.

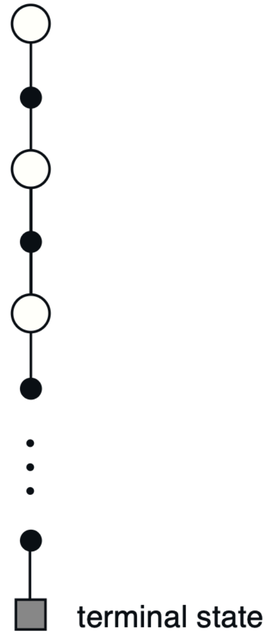


⚠ unsere Perspektive auf greedy reward ist nun (mit reverse) auf den Kopf gestellt: letzter Zug wird am wichtigsten bei kleinem DISCOUNT_FACTOR. Bei DISCOUNT_FACTOR 1 bleiben alle gleich viel wert

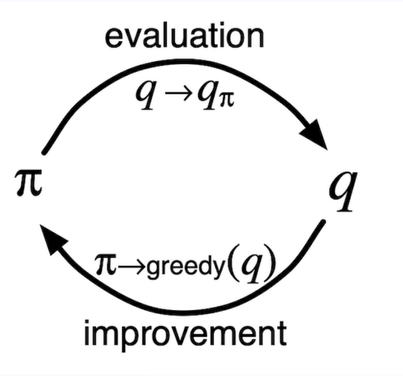
Monte-Carlo + NN

Typischerweise wird Trajektorie rückwärts abgearbeitet:

```
visited = set()
G = 0
for t in reversed(range(len(episode_data))):
    state, action, reward = episode_data[t]
    G = reward + DISCOUNT_FACTOR * G   rekursiv
    if (state, action) not in visited:
        visited.add((state, action))
        Gains[state][action].append(G)
        target = np.mean(Gains[state][action])
        Q(state, action) <- target   via adam MSE
        Policy[state] = np.argmax(Q[state])
```



backup diagram for Monte Carlo estimation of v_π.

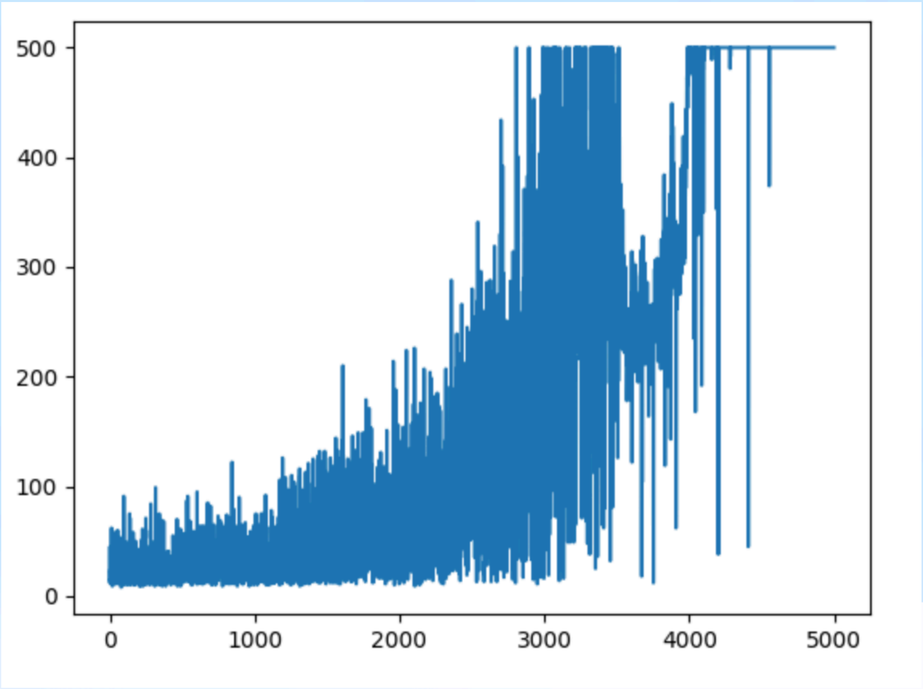


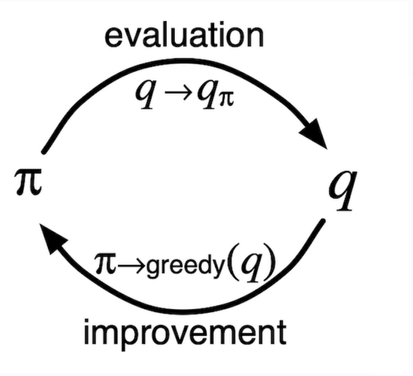
⚠ unsere Perspektive auf greedy reward ist nun (mit reverse)
auf den Kopf gestellt: letzter Zug wird am wichtigsten
bei kleinem DISCOUNT_FACTOR. Bei DISCOUNT_FACTOR 1 bleiben alle gleich viel wert

Monte-Carlo + NN

Wenige Zeilen Code =
Monte Carlo : mächtiger Algorithmus!

```
# Monte Carlo update learning
  Gain = 0
  for state, action, reward in reversed(episode):
    Gain = reward + gamma * Gain
    prediction = q_values(state)[action]
    # Train the model
    loss = (prediction - Gain)**2
```





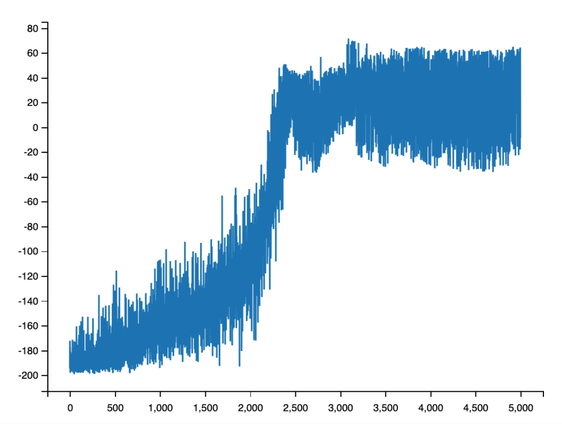
Monte-Carlo + custom rewards

Monte Carlo: Ist ein mächtiger Algorithmus, *wenn* er genügende 'dense' rewards erhält.
Bei seltenen **'sparse' rewards** müssen wir mit **reward shaping** etwas nachhelfen
oder auf verbesserte Algorithmen warten.

Beispiele von 'sparse' reward Problemen:

- Schach (Matt nach 100 Zügen?)
- Mountain Car (Flagge auf Berg wird schwer zufällig erreicht!)
- Taxi
- Viele Spiele (oft erst 'Belohnung' nach minutenlangem Spielen)
- **Frozen Lake** (geht bei kleinen Feldern gerade noch: Geschenk wird auch zufällig erreicht)

```python
def custom_reward(reward, next_state):
    position, velocity = next_state
    return reward + velocity*velocity*1000  # speed is all you need!
```



Pause

Verfahren	Typ	Updateformel	Erwartung / Sample	Politik	Funktionsapproximation
TD(0) (für V)	On-Policy	$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$	Sample	Gegeben	Möglich
SARSA	On-Policy	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$	Sample	Epsilon-greedy	Möglich
Expected SARSA	On-Policy	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \mathbb{E}_{a'} Q(s',a') - Q(s,a)]$	Erwartung	Soft/Epsilon	Möglich
Q-Learning	Off-Policy	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$	Sample	Greedy (target)	Möglich
DQN	Off-Policy	Verlust: $L = (Q_\theta(s,a) - [r + \gamma \max_{a'} Q_{\theta^-}(s',a')])^2$	Sample	Epsilon-greedy	Ja (Deep NN)

Aufgabe

Aufgaben:
a) pole_mc_monte_carlo.py erfolgreich **trainieren** (500 Schritte erreichen)
b) pole_mc_monte_carlo.py **verstehen** und mit pole_DQN_pure.py **vergleichen**
c) Wende unseren extrem erfolgreichen MC Algorithmus auf ein neues Problem an:

MountainCar-v0 (Tipp: gib extra **custom Belohnung** zum Finden der Fahne)
oder
Acrobot-v1
oder
Taxi-v3

oder

d) etwas *ganz* neues / anderes! (NICHT MuJoCo!)

Markov Decision Processes

Wenn das MDP Modell vollständig bekannt ist (d.h. **p**(s', r I s, a) ist explizit gegeben), dann:

- Kann eine dynamische Programmierung Lösung (ohne NN) wesentlich effizienter sein
- **Netze sind noch sinnvoll, wenn:**

1. **Zustandsraum zu groß für Tabellen**

Beispiel: visuelle Zustände (Pixelbilder) → keine tabellarische Repräsentation möglich. Netz dient dann als **Funktionsträger** für Werte- oder Policy-Funktionen.

2. **Zustandsraum kontinuierlich**

Bei kontinuierlichen Zustandsraum (S < R^n) sind Netze notwendig, selbst bei bekanntem $_p$, um Werte zu approximieren.

3. **Model-based RL mit Netz als Modell**

Wenn das MDP nicht gegeben ist, aber durch ein Netz gelernt werden soll → Netz als Approximation von p

4. **Planning mit Netz als Wertfunktion**

Selbst mit bekanntem Modell kann man Monte-Carlo Tree Search oder rollout-based planning (**Simulationen**) aufrufen, wobei das Netz **V(s)** approximiert.

Rollout-based Policy Improvement

- Man testet mehrere Aktionen ab einem Zustand $_s$.
- Für jede Aktion simuliert man einen Pfad der Länge $_d$.
- Der finale Zustand $_{s\_d}$ wird durch ein Netz **bewertet**, statt ihn weiter zu simulieren.

Grundprinzip von **AlphaZero! Schach + Go**

Wie gute Schach spieler: abwechselnd 'rechnen' (klassisch decision-trees) und Positionen 'schätzen' (über NN)

Markov Decision Processes

Wann macht es Sinn, trotz bekanntem **p** ein **Modell** zu lernen?

1. Funktional Approximation

Wenn p bekannt, aber tabellarisch nicht speicherbar ist (z.B. riesiger oder kontinuierlicher Raum), lernt man ein approximatives Modell:

→ nicht, weil p unbekannt ist, sondern weil p nicht speicher- oder (effizient) nutzbar ist in exakter Form.

2. **Transferlernen**

Man trainiert ein Modell auf **p**, um es in anderen MDPs mit **ähnlicher Dynamik** wiederzuverwenden.

Beispiel: ein Roboter lernt ein physikalisches Weltmodell, das auch auf neue Aufgaben übertragbar ist.

Braucht für Erfolg tiefe netze mit abstrakten 'latent' Schichten.

Finetuning: a) Alles weiter lernen b) bestimmte Schichten, oder c) ***LoRA*** **LOw Rank Adaptation**

statt aller Gewichte W neu zu trainieren, trainiert man **W** + m m modifizierer Matrix m = A*A'

3. Policy Gradient **Stabilisierung**

Model-based **Rollouts** (auch mit approximativem p) helfen, Variance im Policy Gradient zu reduzieren (z.B. Dyna-style algorithms).

4. **Sampling-Effizienz**

Ein approximatives Modell ermöglicht zusätzliche künstliche Rollouts, ohne echte Umweltinteraktion.

Transition Memory

Buchführung

Wie bei unserer klassischen Buchführung (z.B. SARSA) ist ein zentraler Aspekt vom Q-Learning der **replay buffer:**

1) Speichere alle **Zustandsübergänge** mit deren Belohnung p(s',r|s,a)
"Welche Belohnung und welcher Folgezustand bei Aktion a"
Jeder Schritt einzeln.

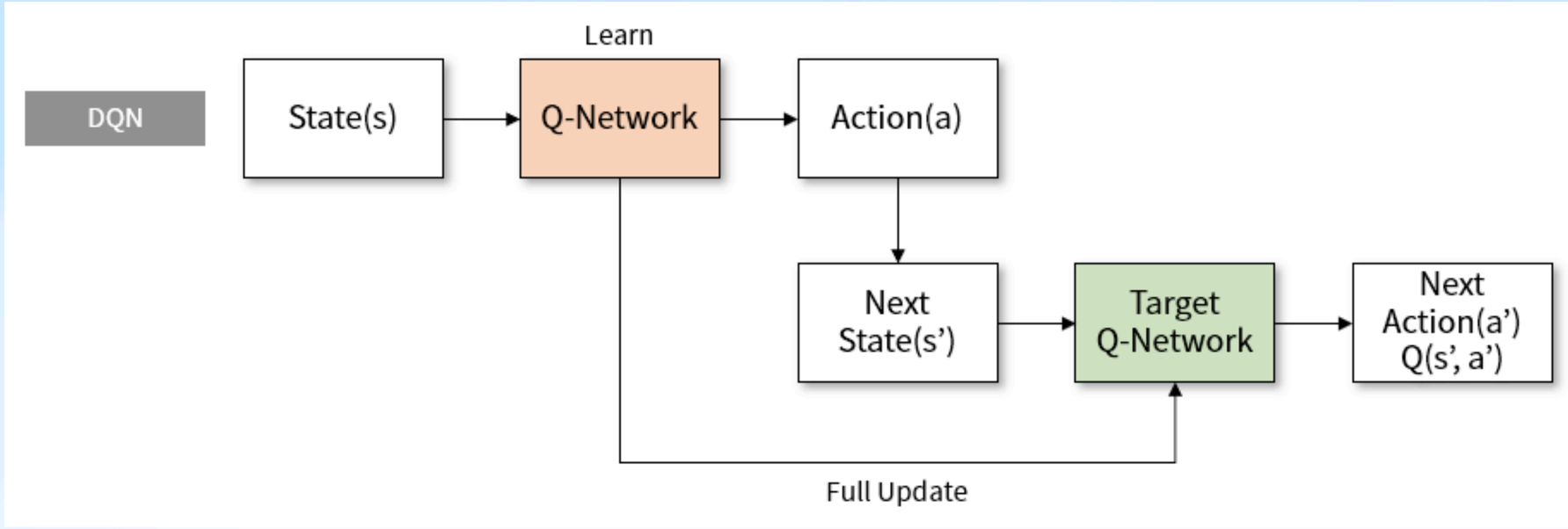
UND / ODER

2) Speichere alle kompletten **Spielverläufe** (Trajektorien/Pfade)
"Monte Carlo"

Target Network

Als Zwischenstrategie zwischen On-Policy und Off-Policy und zusätzlich zu der Batch kann man zwischen **zwei** verschiedenen Netzwerken hin- und herschalten, sodass die Policy nur alle paar Schritte upgedatet wird. Dieses so genannte **Target Network** erhöht die Stabilität beim lernen.

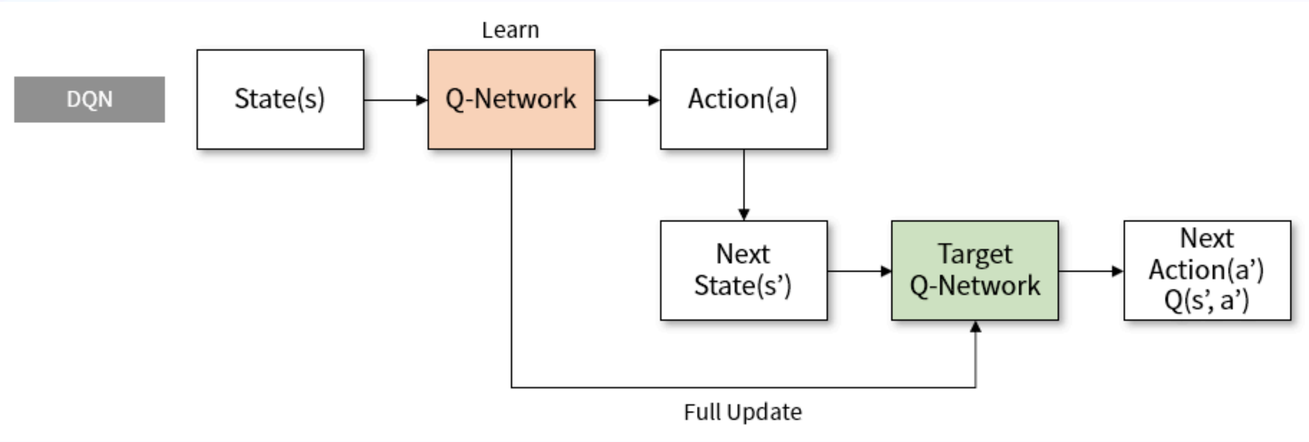
Man sampled das Q-Netzwerk und **glättet** so **stochastische** Varietät.



💡 double buffer beim Spielen

Target Network

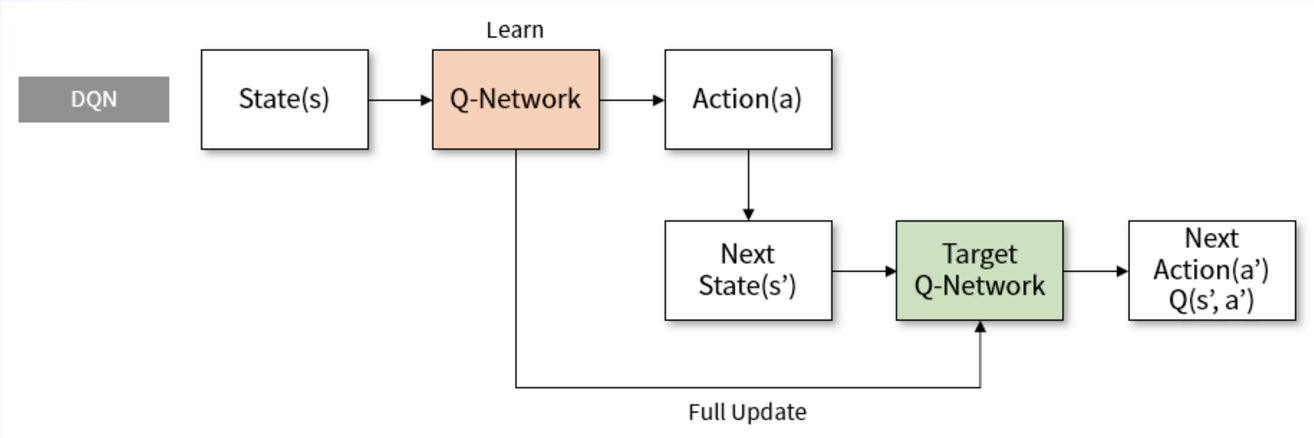
Kann Algorithmus etwas verbessern, aber viel wichtiger ist ein mächtiger Grundalgorithmus (z.B. Monte Carlo)



Target Network

Aufgabe:

a) **trainer policy** vom cart pole (julians Lösung)
b) target net irgendwo einbauen



Pause

Verfahren	Typ	Updateformel	Erwartung / Sample	Politik	Funktionsapproximation
TD(0) (für V)	On-Policy	$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$	Sample	Gegeben	Möglich
SARSA	On-Policy	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$	Sample	Epsilon-greedy	Möglich
Expected SARSA	On-Policy	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \mathbb{E}_{a'} Q(s',a') - Q(s,a)]$	Erwartung	Soft/Epsilon	Möglich
Q-Learning	Off-Policy	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$	Sample	Greedy (target)	Möglich
DQN	Off-Policy	Verlust: $L = (Q_\theta(s,a) - [r + \gamma \max_{a'} Q_{\theta^-}(s',a')])^2$	Sample	Epsilon-greedy	Ja (Deep NN)

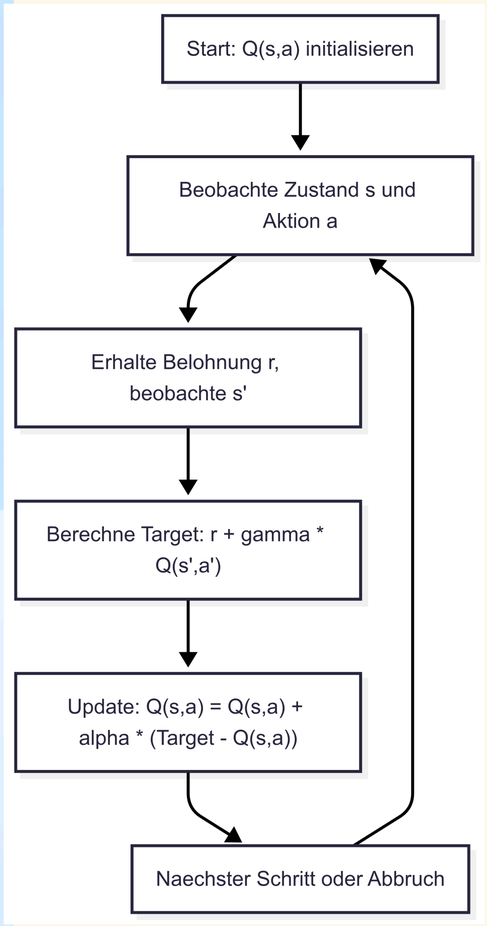
Übersicht bisherige Algorithmen

Methode	Lernmethode	Update-Formel	Rückfluss / Zielwert (target)	Zielfunktion J(θ)	Notiz
allgemein	DRL mit NN	$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\,\mathbf{target - Q(s,a)}\,]$	**target ≈ Gewinn**		
TD (allgemein)	Temporal-Difference, hier **off**-policy	$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\,(\mathbf{r_{t+1} + \gamma\, Q(s',a')) - Q(s,a)}\,]$	$r_{t+1} + \gamma\, Q(s',a')$	$E_{(\pi\theta)}[\, r_{t+1} + \gamma\, Q(s',a')\,]$	**ein Schritt**, Bootstrapping, Updates vor Episodenende, nur aktueller Schritt
SARSA	TD, **on**-policy	$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\,(\mathbf{r_{t+1} + \gamma\, Q(s',a')) - Q(s,a)}\,]$, a' aus aktueller Politik	$r_{t+1} + \gamma\, Q(s',a')$	$E_{(\pi\theta)}[\, r_{t+1} + \gamma\, Q(s',a')\,]$	Lernt tatsächlich gespielte Politik, stabil in stochastischen Umgebungen
Q-Learning	TD, off-policy	$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\,(\mathbf{r_{t+1} + \gamma\, max_{a'}\, Q(s',a')) - Q(s,a)}\,]$	$r_{t+1} + \gamma\, \max_{a'} Q(s',a')$	$E[\, r_{t+1} + \gamma\, \max_{a'} Q(s',a')\,]$	**max,** Findet optimale Politik unabhängig vom Verhalten
Monte-Carlo	MC-Sampling (on-/off-policy)	$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\, G_t - Q(s,a)\,]$	$G_t = r_t + \gamma\, G_{t+1}$	$E_{(\pi\theta)}[\, G_t\,]$	**G_t über ganze Episode (nach Spiel 'rückwärts')**
DQL/DQN	Q-Learning mit Neuronalem Netz	$\theta \leftarrow \theta + \boldsymbol{\alpha}\, \nabla\theta\, (\mathbf{r_{t+1} + \gamma\, max_{a'}\, Q^{-}(s',a') - Q(s,a)})^2$	y wie oben, Target-Netz θ^{-}	$E[\,(y - Q\theta(s,a))^2\,]$	Skaliert auf **kontinuierliche** Zustandsräume
Policy Gradient (PG)	Direkte Politikoptimierung	$\theta \leftarrow \theta + \alpha\, \nabla\theta \log \pi\theta(a \mid s) \cdot G_t$	G_t wie MC	$E_{(\pi\theta)}[\, G_t\,]$	
α	Lernrate	Man kann diskret und kontinuierlich mit 'tricks' austauschen:	Empfehlung: **MC!**		
γ	Diskontfaktor	kontinuierlich -> diskret buckets für Zustandsraum	Empfehlung: **PG** baselines		
π	Politik (Policy)	kontinuierlich -> diskret Normal sampling für Aktionsraum			
θ	Netzwerk- oder Politikparameter	Gewichte θ ∇θ Gradient von Gewichten in learning, **nur Notation**			
Q(s,a)	Quality / Aktionswertfunktion				
G_t	Gewinn/ Goal / Gain ab Zeit t				MC alternativ Ansicht $G_t = \Sigma_{k=0}\, \gamma^k\, r_{t+k+1}$
Qθ/πθ	θ theta steht für Gewichte im Q/π Netz				

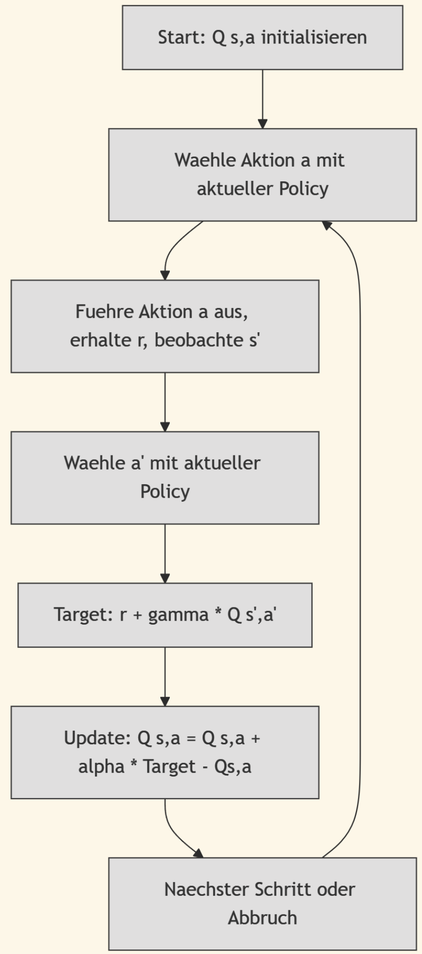
backup diagram for Monte Carlo

Übersicht bisherige Algorithmen

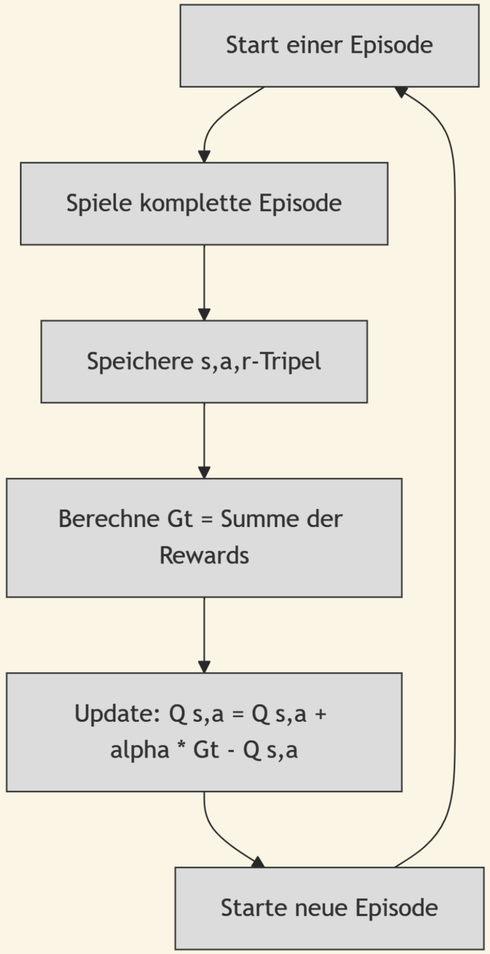
TD



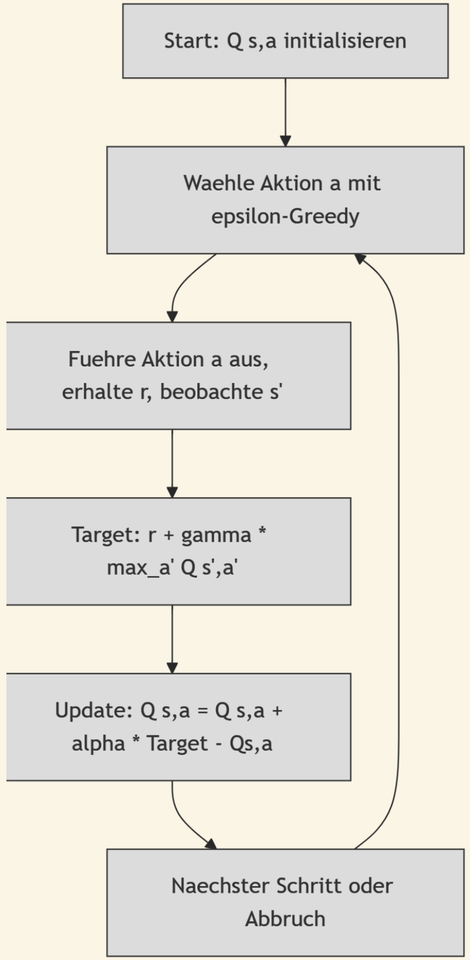
SARSA



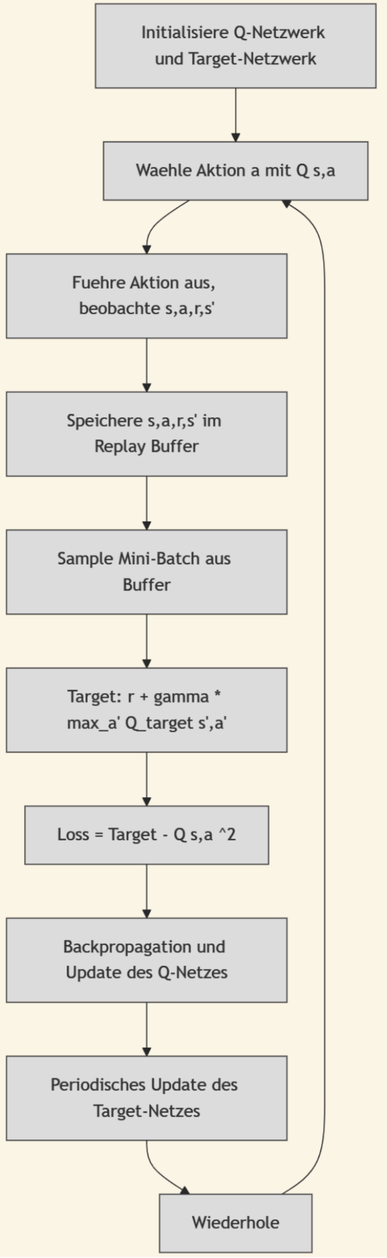
Monte Carlo



Q-Learning



Deep:



Aufgaben

Aufgaben Tag 6

Code GridBook2.py **verstehen**, **experimentieren**, vereinfachen wo möglich

a) VEREINFACHE code so weit es geht.
Reduziere beim entwickeln die Epochen auf 50 um zu sehen ob es prinzipiell trainiert.

b) schreibe **our_reward**(reward, state, …) und entferne alle -1 special cases ...

c) kann man das Netz trainieren ohne Positionen von goal, pit, wall mitzugeben?
erst mode='static'

d) baue neue **Feld**art ein, wo man **zufällig** herumgeworfen wird mit zufälligem Reward
p(s',r'|s,a)=random, unabhängig von Aktion auf dem Feld.

Vokabular

Deep Reinforcement Learning

Deep Q-Learning

Monte Carlo

Sampling

Replay <> batch

Trajektorie

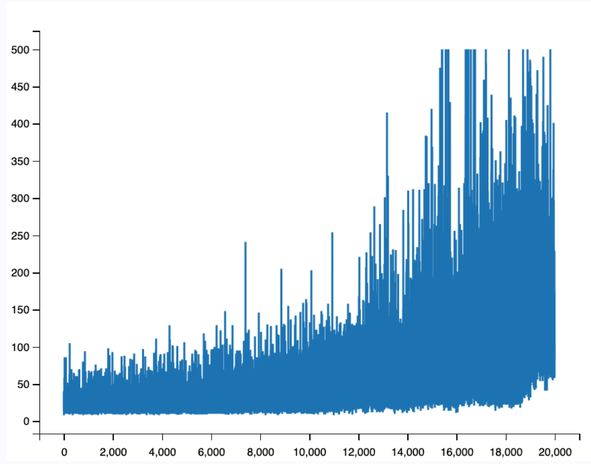
Episodic Sampling vs **Step** Sampling

Monte Carlo Sampling vs Temporal Difference Sampling

Monte Carlo vs Temporal Difference

MC vs TD

reverse gain



DQN convergenz

Deep Q-Learning DQN lernt recht langsam, katastrophales Vergessen, aber deutlich stabiler mit batch / replay: