Reinforcement Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-06 Dozent: Karsten Flügge

Themen des Tages

Tag 4

Exploration vs. Exploitation

Q-Learning: Definition, Algorithmus

Q-Learning: off policy
SARSA: on policy

Temporal-Difference Learning (TD)

Exploration vs. Exploitation

Exploration = Erkundung

Grundidee: um schnell optimale Pfade zu finden wollen wir am Anfang **alle** möglichen **Zustände und Aktionen** mindestens ein (paar) mal finden / ausprobieren. Das geht manchmal bei *endlichen* Problemen, bei kontinuierlichen versucht man Bereiche abzudecken, oder **zufällig** zu entdecken.

Novelty Neuheiten entdecken
Surprise Überrascht werden: Vorhersage ≠ Beobachtung

Exploration vs. Exploitation

Exploitation = Ausnutzung

Wenn man den **Wahren Wert** von Zuständen oder die **wahre Qualität** von Aktionen kennen würde könnte man als **policy stets die beste Aktion** auswählen!

Das nennt sich **greedy** exploitation (Ausnutzung).

Warum ist diese Strategie in der Regel nicht die beste?

- Risko
- niemals neue Situationen finden
- wir wissen in der Regel nie was wirklich der beste Zug ist
- "bester Zug" ist nur eine **Schätzung** (kann falsch sein)

Erstbeste Nicht die beste? Optimierbar? Zeit?

Exploration vs. Exploitation

Exploration = Erkundung

Generell am Anfang möglichst viel Erkundung,
dann auf Exploitation umschwenken.

Radikal: Einfach n steps Daten sammeln, systematisch oder zufällig

Prozentual: 10% der Actions zufällig "**e**psilon ≈ **E**rkundungsfaktor"

ε-greedy

Primitiver Ansatz (aber oft relativ effektiv) :

ε 'exploration Erkundungs-Wahrscheinlichkeit'

Mit **Wahrscheinlichkeit ε** immer wieder **zufällig eine Aktion** ausprobieren (meist mit **epsilon ε** bezeichnet).

```
# ε = 0.1 bedeutet 10% Zufall    ε = 1 = 100% Zufall
if random() < ε:     oder z.B. if step < 100:
    action = random(len(actions))  # gleichverteilt zufällige Aktion
else:
    action = best_action(state)     # argmax(q_values)
```

Das macht vor allem dann Sinn, wenn wir **nicht wissen,** ob die Umgebung uns **immer den selben reward** für eine Aktion gibt!

Extrem: Am Anfang ε = 1 Am Ende ε = 0
ε = 1 - step / total_steps einfach linear runter skalieren!

ε in anderen policies

ε 'explorations Wahrscheinlichkeit'

Mit **Wahrscheinlichkeit ε** immer wieder **zufällig eine Aktion** ausprobieren (meist mit **epsilon ε** bezeichnet).

das wird nicht nur bei ε verwendet, sondern fast in allen policies

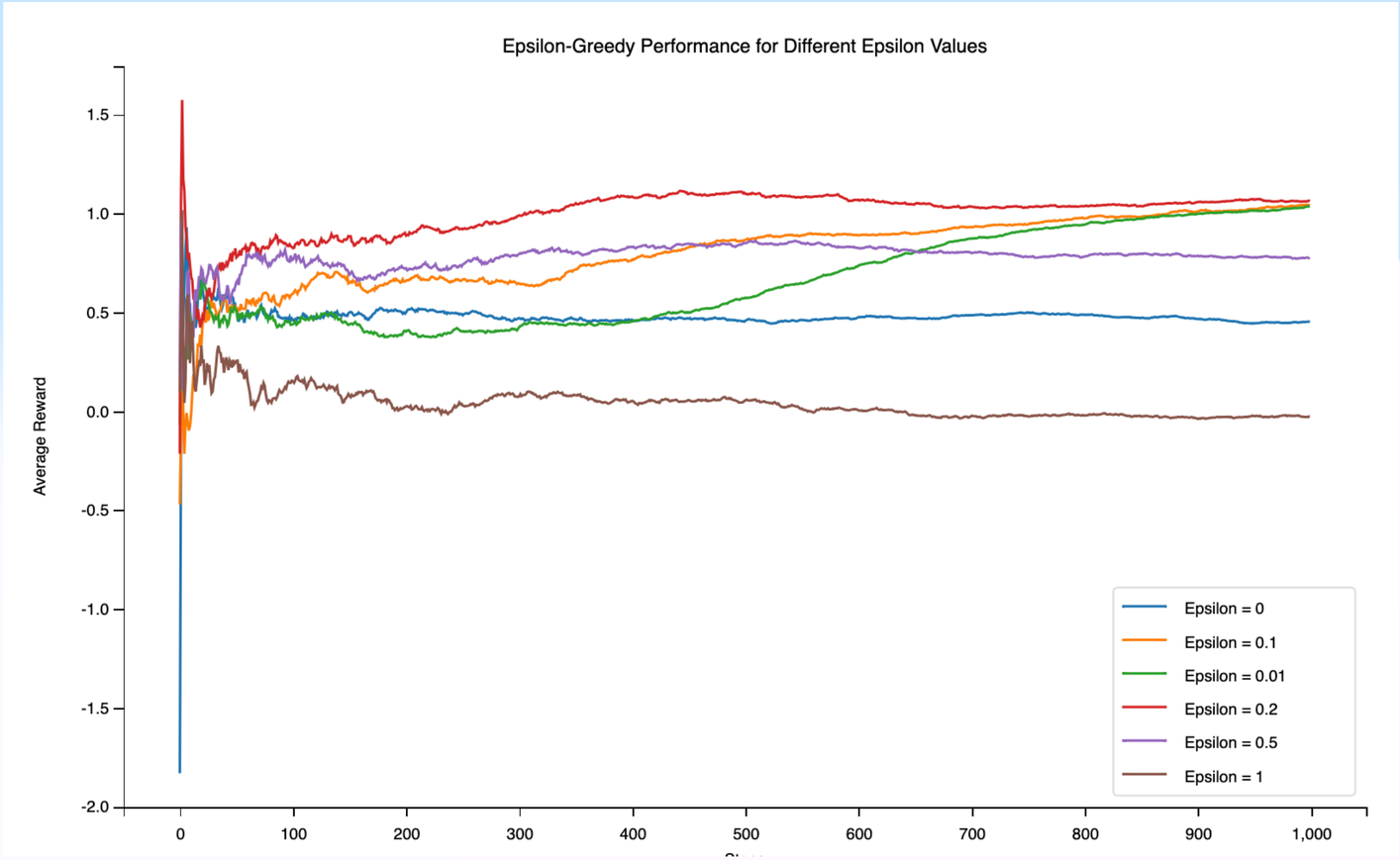
```
if random() <= ε:
    action = random(len(actions))  # gleichverteilt
else:
    action = my_policy(state) # z.B.:
    # action = softmax_action(state) # Zufall anhand der Q-Werte
```

Das macht vor allem dann Sinn, wenn wir **nicht wissen,** ob die Umgebung uns **immer den selben reward** für eine Aktion gibt!

ε in anderen policies

ε fest setzen ist primitiv aber funktioniert für ≈ .01 'ganz gut' (je nach problem)

Besser: **ε** mit der Zeit abfallen lassen.



Exploration vs. Exploitation

Systematische Exploration

Novelty

"Besucherzahlen" merken:

Wie oft haben wir State / Action schon erreicht / ausgeführt.

state_count = {} # als hash map

und / oder

actions_count = {} # key = (state, action) pair

Exploration vs. Exploitation

Exploration

Novelty

Distanz zum Anfang: nach wie vielen Schritten haben wir state erreicht.

state_distance = {}

Überblick?

Warum nicht immer?
Environment 'komplex'

- mehr**dimensional**
- **diskret vs kontinuierlich**
- Trotzdem anwenden über Töpfe / Intervalle / Range 'diskretisieren'

Exploration vs. Exploitation

Exploration Ausblick: **Entropie, ... maß von Unordnung / Unsicherheit.**

Machen wir in Woche 2/3

Exploration vs. Exploitation

Gefahr: Zu frühes 'gieriges' festlegen auf eine Policy: es könnte noch viel bessere Pfade geben.

Exploration vs. Exploitation

Vertiefung **Exploration** vs. Exploitation

Kapitel 3.2 im Buch

Grundidee wir wollen für **möglichst alle Zustände** und **Aktionen** einschätzen können welchen **Wert** sie haben. Wenn wir uns in unserer Policy zu schnell auf den besten Wert konzentrieren kann es sein dass wir einen besseren übersehen.

Daher sollten **alle** möglichen **Varianten** mindestens ein (paar) mal ausprobiert werden.

einfacher Fall bereits mit **ε-greedy** kennen gelernt

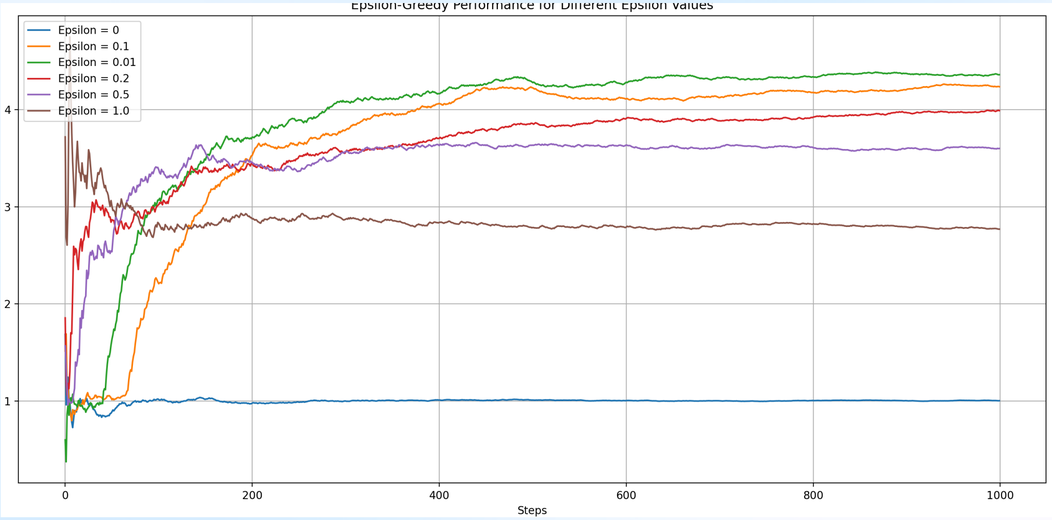
der gelernte **Policy Iteration** Algorithmus v<->π ist "**full sweep**" weil alle Zustände und Aktionen betrachtet wurden. Dem gegenüber steht das heutige Thema "**sampling**".

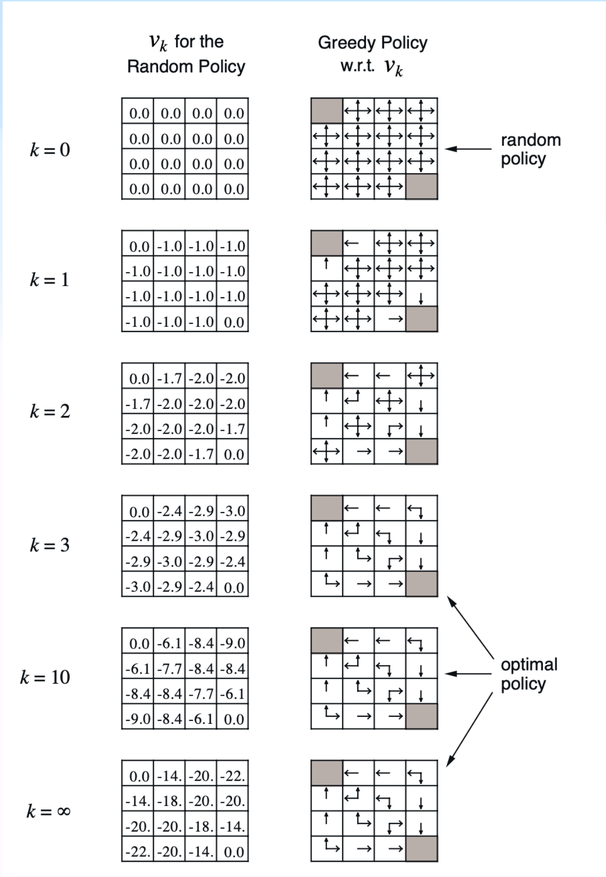
Exploration vs. Exploitation

Exploring Start

der gelernte **Policy Iteration** Algorithmus v<->π ist "**full sweep**" weil alle Zustände und Aktionen betrachtet wurden. Dem gegenüber steht das heutige Thema "**sampling**".

z.B. GridExploringStart.py



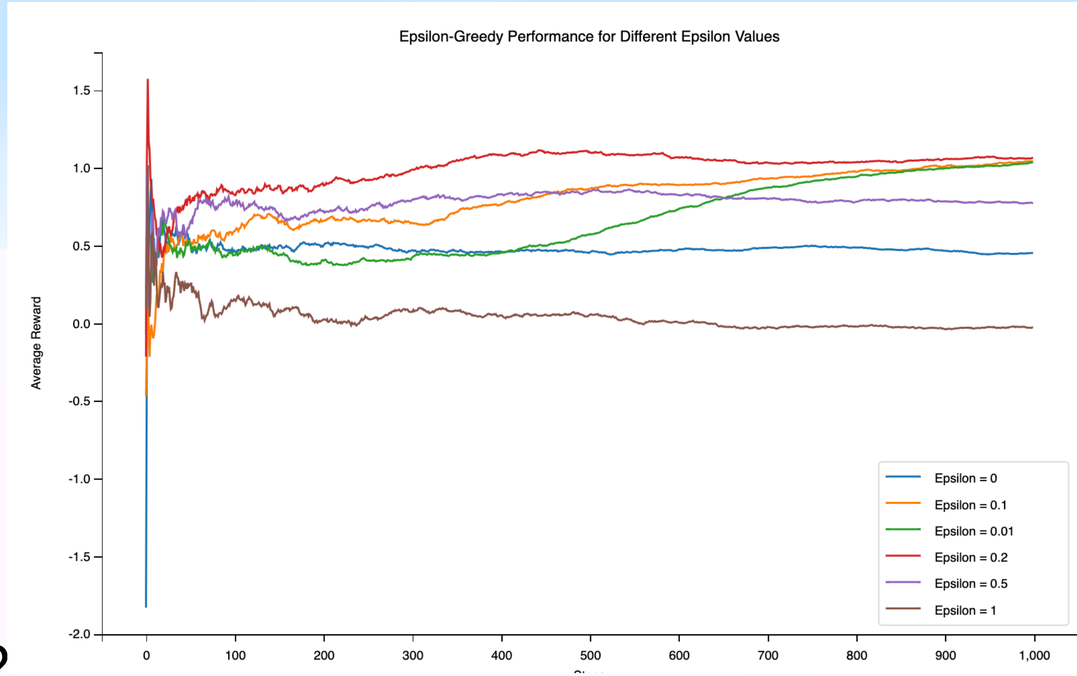


Aufgaben

Extra:

- plotten von verschiedenen Kurven für verschiedene epsilon

- Welches (feste) epsilon liefert den höchsten Gewinn am Ende?
- reproduzierbar? 0.1? warum (nicht)?
- kann epsilon=0 besser sein als andere
  - wann/warum (nicht)?
- kann epsilon=1 besser sein als andere π
  - wann/warum (nicht)?
- (warum) ist abfallendes epsilon besser
    - als konstantes?
- Extra: Vergleich verschiedener epsilon decays
- Kann man allgemein sagen ein decay ist besser?

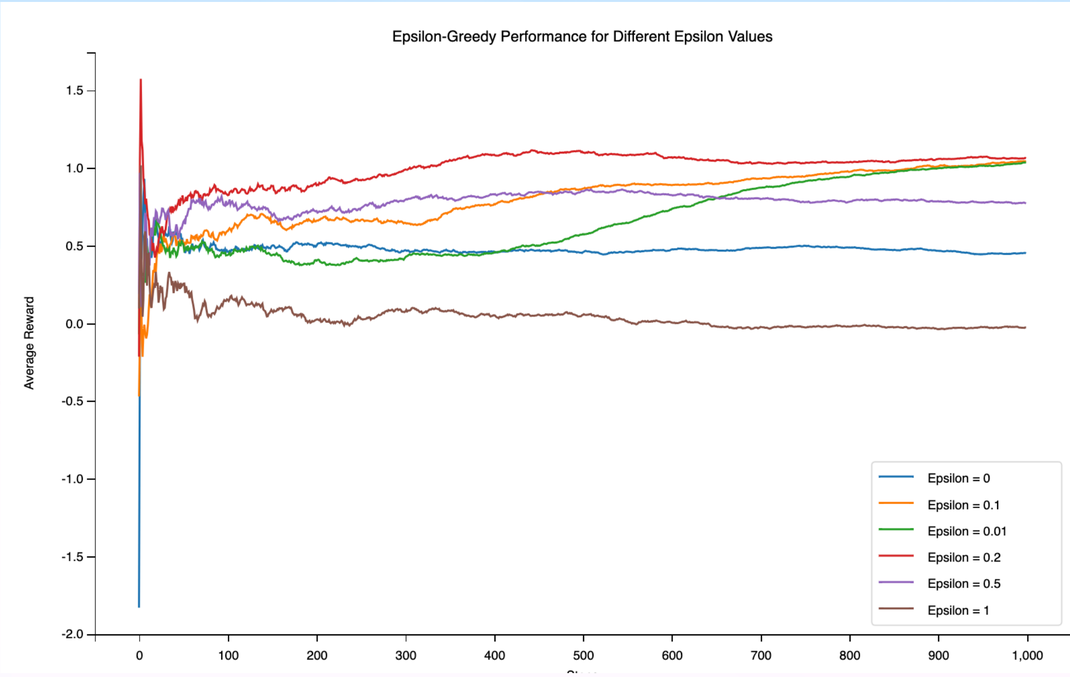


Aufgaben

Extra:

- plotten von verschiedenen Kurven für verschiedene epsilon

- Welches (feste) epsilon liefert den höchsten Gewinn am Ende?
- reproduzierbar? 0.1? warum (nicht)?
- kann epsilon=0 besser sein als andere
  - wann/warum (nicht)?
  - ja, wenn greed policy zufällig richtigen Automaten wählt
- kann epsilon=1 besser sein als andere π
  - wann/warum (nicht)?
  - idr nicht, nur wenn statt greedy 'dumme policy' immer 0 nimmt
- (warum) ist abfallendes epsilon besser
    - als konstantes?
    - idr ja, irgendwann haben wir *genug gelernt*
- Extra: Vergleich **verschiedener epsilon decays**
- Kann man allgemein sagen ein decay ist besser?



Better random

Bisher hatten wir bei ε-exploration streng geteilt zwischen **deterministisch** (greedy) und **gleichverteiltem Zufall**. Gibt es was zwischen gleichverteilt und 'immer best'

```
if random() < ε:
   action = random(len(actions))  # gleichverteilt
else:
   action = best_action(state)     # argmax(q_values)
```

Besser:

Auswahl der nächsten Aktion **zufällig gemäß** dessen **Qualität**

```
def q_weighted_sample(q_values):
  probs = |q_values| / np.sum(|q_values|) # lineare Normalisierung ∑probs=1
  return np.random.choice(len(probs), p=probs)
```

Lerneffekt? Schlechte Schätzung => kaum Verbesserung?
Am Anfang alle q_values 0 => ERROR (oder sogar später)

Better random

Auswahl der nächsten Aktion **zufällig gemäß** dessen **Qualität**

```
def q_weighted_sample(q_values):
  probs = |q_values| / np.sum(|q_values|) # lineare Normalisierung ∑probs=1
  return np.random.choice(len(probs), p=probs)
```

Lerneffekt? Schlechte Schätzung => kaum Verbesserung?
Am Anfang alle q_values 0 => ERROR (oder sogar später)

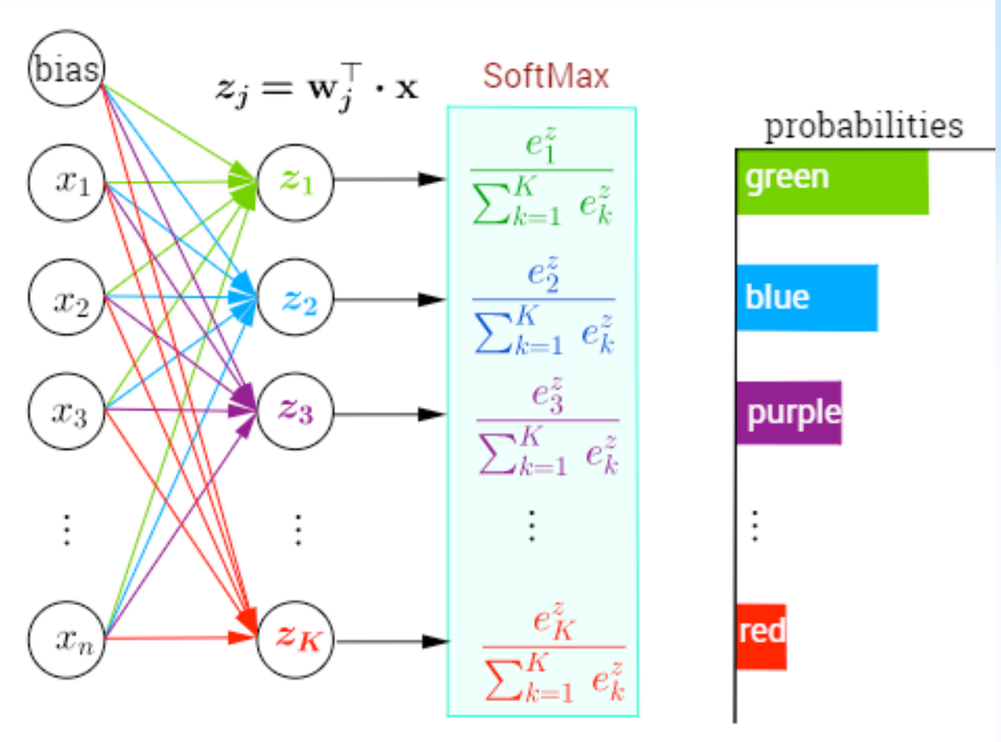
Ausweg: exponieren!

statt
prob = |q_values| / ∑|q_values|
prob = e^|q_values| / ∑e^|q_values| um 0 zu Vermeiden

Softmax

Auswahl der **nächsten Aktion** gemäß **skalierter Qualität**

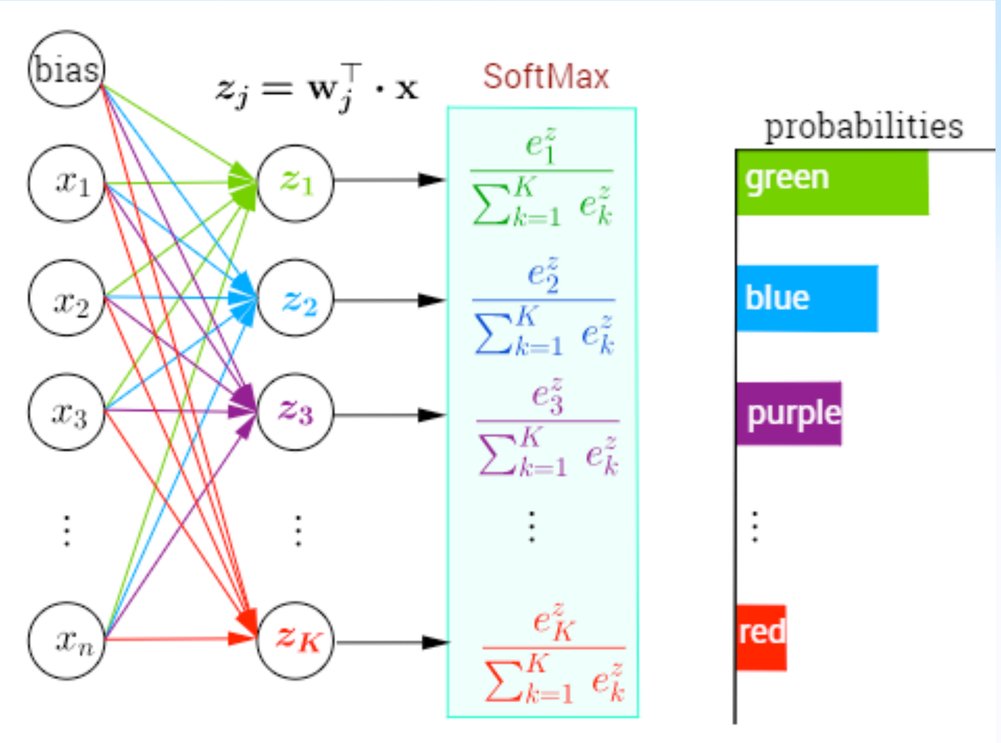
```
def q_softmax_sample(q_values): # action as index 0…n
  probs = np.exp(q_values) / np.sum(np.exp(q_values))
  return np.random.choice(len(probs), p=probs) # wähle zufällig gemäß Verteilung
```

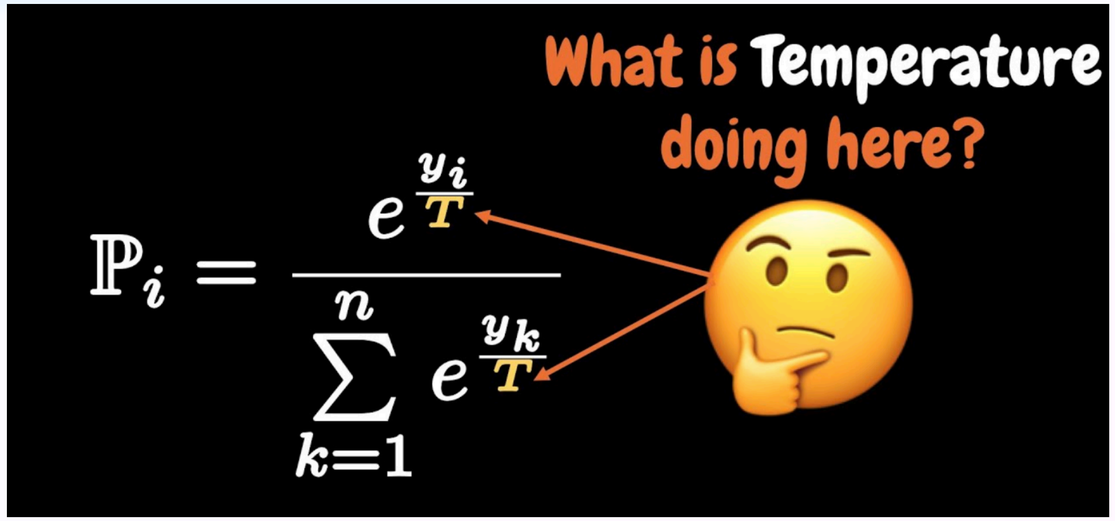


Softmax

Temperatur T
extra Parameter mit dem man steuern kann wie stark das maximum abgeschwächt / verstärkt wird.

```
def q_softmax_sample(q_values): # action as index 0…n
  probs = np.exp(q_values/T) / np.sum(np.exp(q_values/T))
  return np.random.choice(len(probs), p=probs) # wähle zufällig gemäß Verteilung
```





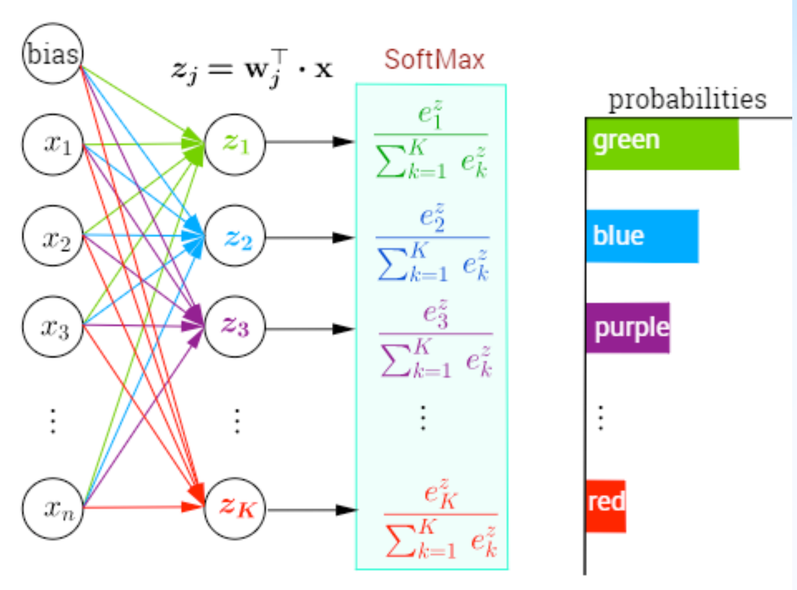
Softmax

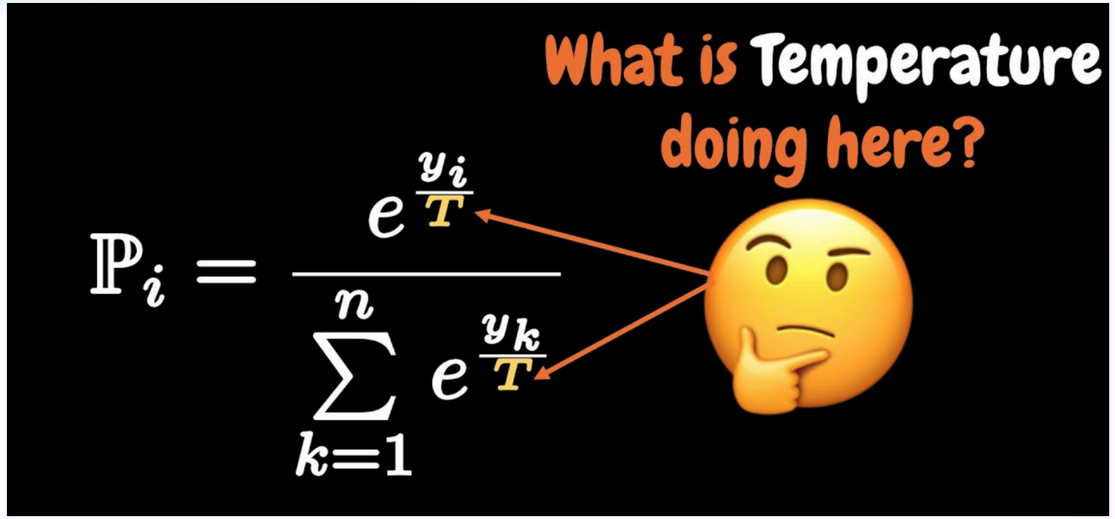
Softmax Ziel: alle Werte zwischen 0 und 1 quetschen, damit wir es als probabilty deuten können.
Summe aller Aktionen / Aktivierungen soll 1 sein.

was würde passieren wenn alle Werte null sind?

```
probabilties = |q_values| / np.sum(|q_values|) # lineare Normalisierung ∑probs=1
```

Deswegen wird bei Softmax noch e Exponent angewendet!
Für z0=0 (z1=**1** z2=**2**) dann sollte softmax(z0) immer noch nahe 0 sein
softmax(z0) = exp(0)/∑… = 1 / (1 + e^1 + e^2) ≈ 0.01
softmax(z1) = exp(1)/∑… = e / (1 + e^1 + e^2) ≈ 0.24
softmax(z2) = exp(2)/∑… = e^2 / (1 + e^1 + e^2) ≈ 0.66





Exploration via Softmax

Zufällige Auswahl der nächsten Aktion gemäß skalierter **Qualität**

```
def q_softmax_sample(q_values):
  probs = np.exp(q_values/T) / np.sum(np.exp(q_values/T))
  return np.random.choice(len(probs), p=probs)
```

Exploration

Temperatur

```
T ≈ 0.0 # Temperatur ≈0: deterministisch max
T = 1.0 # Temperatur 1: moderate Exploration
T = 100 # Temperatur ∞: uniform random
```

T / **τ (tau)** ist die Temperatur

- **Hohe T → hohe Entropie** (Exploration)
- **Niedrige T → niedrige Entropie** (Exploitation)

In Neuronalen Netzen haben wir gesehen dass
softmax … gibts auch bei LLMs

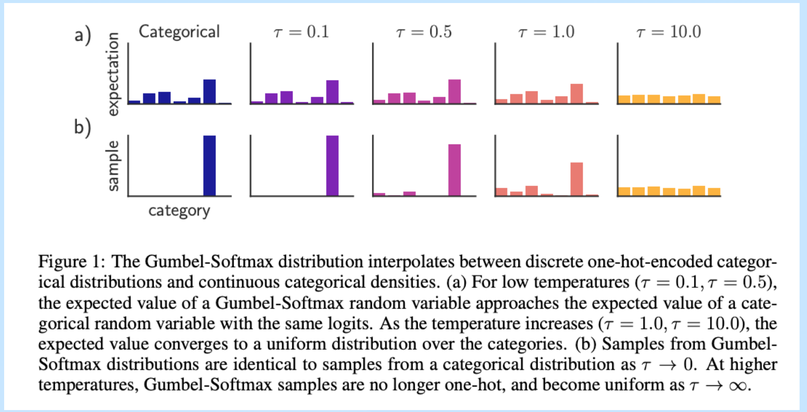
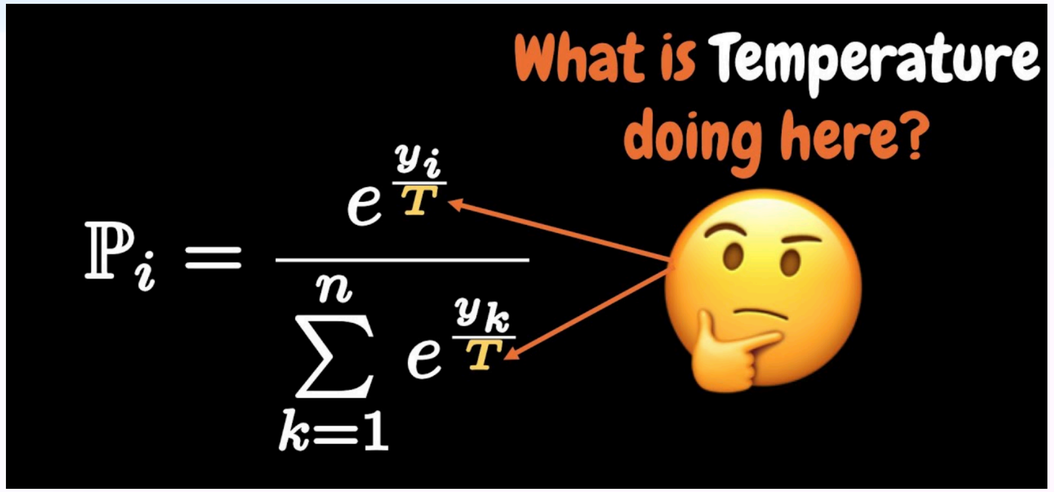


Figure 1: The Gumbel-Softmax distribution interpolates between discrete one-hot-encoded categorical distributions and continuous categorical densities. (a) For low temperatures ($\tau = 0.1, \tau = 0.5$), the expected value of a Gumbel-Softmax random variable approaches the expected value of a categorical random variable with the same logits. As the temperature increases ($\tau = 1.0, \tau = 10.0$), the expected value converges to a uniform distribution over the categories. (b) Samples from Gumbel-Softmax distributions are identical to samples from a categorical distribution as $\tau \to 0$. At higher temperatures, Gumbel-Softmax samples are no longer one-hot, and become uniform as $\tau \to \infty$.



Softmax Exploration

`Vertiefung Exploration vs. Exploitation`

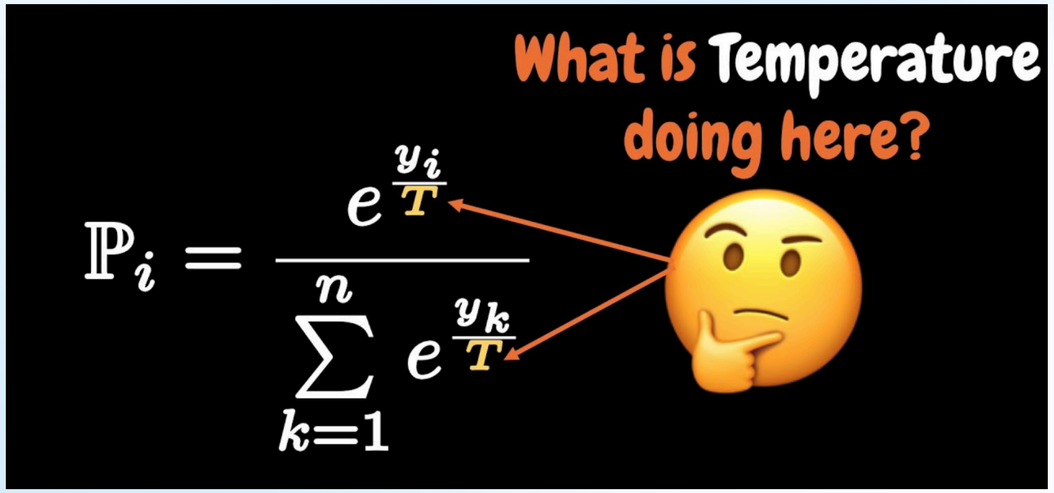
Kapitel 3.2 im Buch

Temperature

Ist bei T>0 sichergestellt, dass alle Aktionen einmal ausprobiert werden?

Im Grenzfall **ja**: keine W-keit ist wirklich 0.

Praktisch eventuell **nein**, wenn auf 0 abgerundet wird oder wir zu früh abbrechen



Banditen

Hello World of RL:

Umwelt: Casino, n-Automaten
Aktion: wähle eine Automat (a=x : Automat x)
Reward: unbekannt: Automaten haben irgendeine Zufallsverteilung

"probing/sampling/ausprobieren"

Policy: Alle Automaten mehrfach ausprobieren, mitzählen was der durchschnittliche Gewinn ist! Dann können wir greedy den besten Auswählen.

Erfolg messen mit **moving average reward**

epsilon decay

epsilon mit der Zeit runter setzen

Ansätze:

- **Linear interpolieren:**

```
epsilon = 1 am anfang
epsilon = 0 am ende
epsilon = 1 - step / number_steps

epsilon = 1 - 2*step / number_steps  # schon nach der Hälfte auf 0
```

- **Exponentieller Abfall**

```
epsilon = epsilon * decay_rate (<0)
Gefahr: epsilon zu schnell / langsam auf 0
```

- **Schrittweise**

- **Andere**

≈ learning rate decay

Systematic Trial Exploration

- **"Jede Aktion in jedem State einmal ausprobieren"**

- **Warum nicht immer?**

1) **zu viele Zustände / Aktionen** (z.B. Schach 10^160)
2) **zu lange pro Aktion Testen** (Medical Trials, ...)
3) **unendlich viele** (continuous)

Gegenmaßnahmen bei continuous: Blöcke/Intervalle

sampling => back to epsilon exploration

Wiederholung

Wiederholung

Gute Zustände "**V**alue"
Gute Aktionen "**Q**uality"

Was heißt gut intuitiv?
Was heißt gut präzise?

Wiederholung

Gute Aktionen "**Q**uality"

Quality über **mittleren reward**
(aufsummieren und Teilen $Q \approx \sum r/n$)

Kette von rewards, im laufe des Spiels rewards sammeln: **Gewinn G** $\approx \sum r$ (gain)

Q ≈ E(G) Qualität erwarteter mittlerer Gewinn
E(G) kennen wir nicht genau, sondern muessen wir **schätzen**

Wiederholung

Q(a) ≈ E(G) Qualität erwarteter mittlerer Gewinn
E(G) kennen wir nicht genau, sondern muessen wir **schätzen**

Q(a) <- r + E[G'] geschätzte ZUKÜNFTIGE Gewinne r konkreter Reward
Verschiedene Schätzungen, was man in Zukunft bekommt
Schätzungen verbessern mit dem was wir haben

Erweiterte Konzepte

Erweiterte (Hilfs)Konzepte

- **Episode:** "Spiel" Ablauf einer Folge von Aktionen und Zuständen ≈ **Trajektorie** $\tau=(s_0,a_0, \ldots a_t,s_t)$
- **Gain** Gewinn **g:** (Erwartete) *Summe der Belohnungen*, bis zum Ende des Spieles
- **Value Function v**: Erwartete zukünftige Belohnung ab einem **Zustand** ≈
- **Qualität (Quality) q**: Erwartete zukünftige Belohnung für eine **Aktion** in einem Zustand.

Value bewertet eine Position

Quality bewertet einen Zug!

Man kann in **guten** Positionen **schlechte** Züge machen und
in **schlechten** Positionen **optimale** Züge also das 'bestmögliche'.
Ob ein Zug zielführend war entscheidet sich oft erst am Ende des Spiels "Bauernopfer".

Gewinne Gewinne Gewinne

"Gewinn ≈ Summe Belohnungen" (erwartet oder real)

Gewinn Pro Spiel
Mittlerer Gewinn Pro Spiel gezählt **=>**
Erwarteter Gewinn $\mathbb{E}[G_t]$ abstrakt
Geschätzter erwarteter **Gewinn**

Diskont Gewinn: kumulierte, diskontierte Belohnung ab einem bestimmten Zeitpunkt *t*
$G_t = \sum_k \gamma^k r_{k+1}$ t = Aktueller-Zustand k=Zug# ab jetzt (**r ab jetzt** "lokal")

Im code **wird alles ganz einfach**

```
gain = 0
for each step:
   gain = reward + gamma * gain
```

Q-Learning

Q-Learning ist ein Reinforcement-Learning-Algorithmus, der einen Agenten darauf trainiert, seinen möglichen **Aktionen** in Abhängigkeit vom aktuellen Zustand **Werte zuzuweisen.**

Q-Learning

Q-Learning ist die **zentrale Idee** von modernem **RL**

Heute setzen wir sie **manuell** um mit einer eigenen kleinen **Tabelle.** Aber die selbe Idee wird bis zum Ende des Kurses auch mit neuronalen Netzen in modernen Algorithmen verwendet.

Q-Learning

Intuitive Grundidee:

Buchführung:
Welche **Aktion** führte zu welcher **Belohnung**?

Erst selbst probieren,
dann Theorie,
dann (nochmal) Theorie intuitiv umsetzen

Einfaches Bellmann

Vereinfacht wenn determisistisch

$\mathbf{v(s)} = \sum_a \boldsymbol{\pi}(a \mid s)\ [\ r + \gamma \cdot v(\mathbf{s}')\ ]$

Noch einfacher, ein Schritt und eine Aktion zur Zeit
$\mathbf{q(a, s)} = \ r + \gamma \cdot \text{argmax}\ q_a(\mathbf{s}')$

bootstrapping: alles **0**

```
Q(a):= r + E[G']

delta = (r + G') - Q(a)   möglichst 0!

delta misst den Fehler unserer Schätzung
```

=> Korrektur der Schätzung

Q-Learning

Q-Lernen ist eine Methode des bestärkenden Lernens. Beim bestärkenden Lernen führt ein Agent Aktionen in einer Umgebung aus und erhält dafür Belohnungen. Der Agent lernt eine Strategie, mit der er langfristig möglichst viele Belohnungen erhält. Beim Q-Lernen werden **Q-Werte geschätzt.** Ein Q-Wert steht für den langfristig erwarteten Wert einer Aktion. Q-Lernen ist eine Form des Temporal Difference Learnings, die Methode verbessert Schätzungen direkt nach dem Ausführen einer Aktion. Die **Verbesserung** erfolgt auf Basis der gerade erhaltenen **Belohnung und** der geschätzten zukünftig zu **erwartenden Belohnung.**

Tabellarische Methoden

Heute: Q-Learning, SARSA

SARSA

State
Action
Reward
State'
Action

Q-Learning

Sehr ähnlich zur vorher betrachteten **value iteration**, wo wir abwechselnd *π und v* verbessert haben, können wir **π und q** verbessern mit **quality iteration**.

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

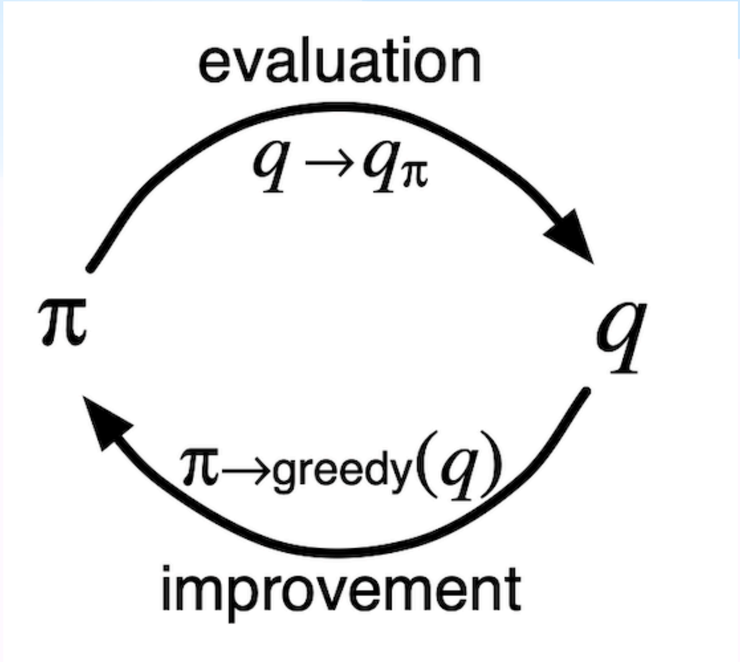
$q_{k+1}(a') = \Sigma\ p * [r + \gamma \max_a q(s',a)]$

$\pi_k(s) = \text{arg-max}_a\ q(a \mid s)$ "greedy with respect to q"

Daher bezeichnet man diese Algorithmen auch als

Q-Learning

quality learning =
action value improvement =
state-action-pair optimization



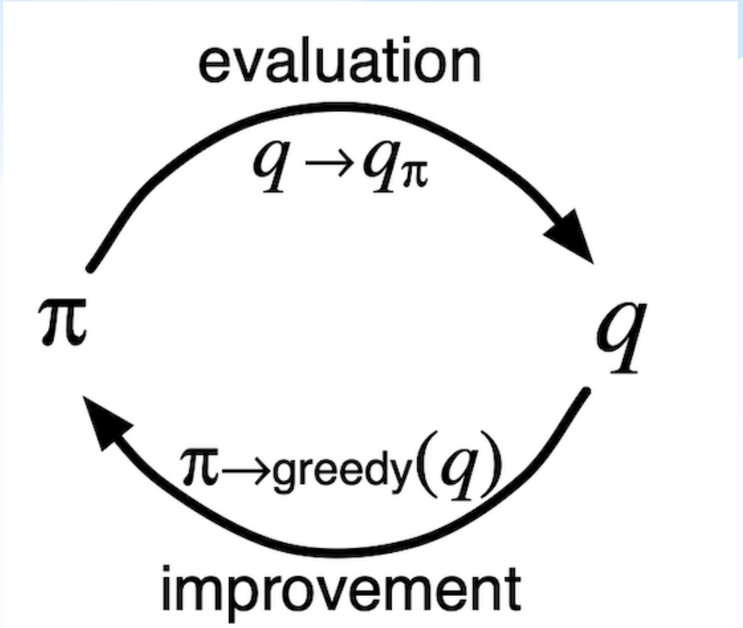
Q-Learning

Sehr ähnlich zur vorher betrachteten **policy iteration**,
wo wir zum verbessern von *π und v* die **Bellmann Gleichung** verwendeten,
können wir bei Monte-Carlo **abwechselnd π und q verbessert:**

QUALITY EQUILIBRIUM
$q^*(s,a) = \sum p(s',r|s,a) * [r + \gamma \cdot \max_{a'} q^*(s',a')]$
Theoretisches Gleichgewicht einer optimalen Quality

QUALITY ITERATION
$\mathbf{q_{k+1}(s,a) = \sum\sum(s',r)\ p(s',r\ |\ s,a) * [r + \gamma\ \max_{a'}\ q_k(s',a')]}$
$\mathbf{q_{k+1}(s,a) = \sum_a\ p * [r + \gamma\ \max_{a'}\ q_k(s',a')]}$ vereinfacht, wenn s' deterministisch
Praktischer Ansatz um die Qualität iterativ zu schätzen

QUALITY UPDATE
$\mathbf{q'(a\ |\ s) = q(a\ |\ s) + \alpha * [\ r + \gamma\ \max_{a'}\ q_k(s',a') - q(a\ |\ s)\ ]}$
Ähnlicher Ansatz um die Qualität iterativ zu schätzen



Aufgabe: Zeige dass QI Spezialfall von QU ist wenn p (Regeln!) deterministisch. (für welches alpha?)

Temporal Difference

Idee:
wie stark weicht unsere **Schätzung** vom **Gemessenen** ab?
delta = (r + E[G']) - Q(a) möglichst 0!

das nennt sich <u>temporal difference</u>
*Schritt*weise Differenz der Schätzung
Korrektur-Term, gewichtet mit alpha

Q(a) += alpha * (r + E[G'] - Q(a))

alpha zweite learning Rate (gamma im Gewinn)
alpha: wie stark sollen neue Erfahrungen altes Wissen überschreiben?
gamma: wie wichtig sind zukünftige Belohnungen

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' | s) = q(a | s) + [r + γ $\max_a$ q(s',a) - q(a | s)] # zu greedy policy

q'(a | s) = q(a | s) + ***α*** * [r + γ **$\max_a$' q(s',a') - q(a | s)**]
α learning rate: wie viel wollen wir unsere Schätzung updaten
mit der Differenz von dem was wir **geschätzt** haben zu dem was wir **gemessen** haben

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 + γ **$\max_a$'q(s',a')**
q'(a) += ***α*** * [**q_target - q(a | 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

target := r + γ **$\max_{a'}$ q(s',a')** **≈ E(G) ≈ erwarteter Gewinn 'gain'**
best_reward := **$\max_{a'}$ q(s',a')**
target := current_reward + γ * best_reward
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 viel wollen wir unsere Schätzung updaten

Wenn Schätzung 'estimate' übereinstimmt mit neuer Messung 'target',
dann müssen wir q nicht updaten!

Wenn Schätzung 'estimate' etwas zu niedrig ist, dann wird q nach oben korrigiert
Wenn Schätzung 'estimate' etwas zu hoch ist, dann wird q nach unten korrigiert

Temporal-Difference (TD)

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

QUALITY UPDATE

q'(a | s) = q(a | s) + ***α*** * [target **- estimate**]

α learning rate: wie viel wollen wir unsere Schätzung updaten

Verschieden Algorithmen mit verschiedenen targets.
target ist immer ein **Schätzwert** (Zahl)
target = r + gamma * $\max_a$ q(a) ≈ Durchschnitts-Gewinn
target = "Advantage" "Vorteil unserer Aktion gegen andere"

Temporal-Difference Learning

"die zentrale neue Idee beim RheinForcement Learning"

Kombination von **Monte Carlo (replay)** und dynamic programming (DP) (Q-values):
man betrachtet **nur einen Schritt**, welcher deterministisch (oder via samples statt Erwartungswerte) erfolgt.
Dadurch fällt das p in der Formel weg:

Die Bellmann Optimalitätsbedingung bei TD wird radikal einfacher:

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

Sample-based Temporal Difference Learning **bootstraps** from its own estimate:

Im ersten Schritt sind alle Q 0

Im zweiten Schritt sind alle Q = directer **r**eward

Im dritten Schritt kommen dann die vergangenen Schätzungen mit ins Spiel

Temporal-Difference Learning

	Monte Carlo	TD (sample-based)	TD (non-sample-based)
Uses bootstrapping?	✘ No (waits for full return)	✔ Yes (uses estimated next value)	✔ Yes (uses expected next value)
Uses samples?	✔ Yes (sampled returns)	✔ Yes (sampled transitions)	✘ No (uses model expectation)
Requires model?	✘ No	✘ No	✔ Yes
Update target	$G = \sum_{t=0}^{T} \gamma^t r_t$	$r + \gamma \hat{V}(s')$	$\mathbb{E}_{s',a'}[r + \gamma V(s')]$

On Policy / Off Policy

On-Policy Learning:
Die Lernmethode nutzt **Daten**, die mit der aktuellen Policy (Strategie) **erzeugt** wurden.
Das bedeutet: dieselbe **Policy** wird zum **Handeln** und zum Lernen verwendet.
Beispiel: **SARSA** (State-Action-Reward-State-Action)
Achtung: premature Optimization? Im Extremfall noch während eines Spieles

Off-Policy Learning:
Lernmethode nutzt Daten, die von einer **anderen Policy** erzeugt wurden
– typischerweise einer explorativen oder **älteren** Policy.
Lernen erfolgt anhand einer Ziel-Policy (**target policy**),
während eine Verhaltens-Policy (**behavior policy**) die **Daten generiert**.
Beispiel: **Q-Learning**

Fließender Übergang, oft wird die Policy verzögert Updated: pro **batch**, nach mehreren Episoden …
Auch klassisches off-Policy wird zum latenten on-policy, wenn man den Trainingslauf mehrfach wiederholt.
Fließender Übergang von off policy zu supervised learning

Q-Learning vs SARSA

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\text{target} - Q(s_t, a_t)\right]$$

target bei Q-Learning: r + ɣ*max Q(s',a') bestes r über alle actions
target bei SARSA: r + ɣ*Q(s',a') für konkretes a' der policy

```python
def update_q_values(state, action_idx, reward, next_sta
    y, x = state
    ny, nx = next_state
    best_next_reward = np.max(Q[ny, nx])
    target = reward + ɣ * best_next_reward # off policy
    Q[y, x, action_idx] += ALPHA * (target - Q[y, x, action_idx])
```

```python
def update_q_values(state, action_idx, reward, next_state):
    y, x = state
    ny, nx = next_state
    policy_next_reward = Q[ny, nx, ACTIONS[next_state]]
    target = reward + ɣ * policy_next_reward # on policy
    Q[y, x, action_idx] += ALPHA * (target - Q[y, x, action_idx])
```

Aspect	Q-Learning	SARSA
Policy	Off-policy (greedy target)	On-policy (uses actual next action)
Target	$\max_{a'} Q(s', a')$	$Q(s', a')$ (for chosen a')
Behavior	Learns optimal policy	Learns behavior policy

Temporal-Difference Learning

6.4. SARSA: ON-POLICY TD CONTROL

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
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

Figure 6.9: Sarsa: An on-policy TD control algorithm.

Garantierte Konvergenz

Garantierte Konvergenz von Q-Learning unter folgenden Bedingungen (Watkins & Dayan, 1992):

- **endlicher** Zustands- und Aktionsraum,
- jede Aktion wird (theoretisch) **unendlich** oft gewählt (exploration),
- Lernrate **α** erfüllt Bedingungen: a) $\sum \alpha_i = \infty$ b) $\sum \alpha_i^2 < \infty$ bei theoretischer unendlicher Ausführung**

  α_i dynamisch Lernrate z.b. **1/i**

Dann Konvergenz gegen die optimale Q-Funktion Q* (bei Q-Learning), bzw. gegen Qπ bei SARSA

α_n = 1/n

$\sum$ 1/n = 1/2+1/3+1/4 + … = ∞ a) OK

$\sum$(1/n)^2 = 1 + 1/4 + 1/9 + 1/16 + … = π/6 endlich! b) OK

$\sum$1/(2^n) = 1 + 1/2 + 1/4 + 1/8 + … = 2 Veranschaulichung einer unendlichen Summe, die endlich ist.

Q-Learning mit **NN** ist **nicht konvergent** in der Theorie (Baird's counterexample).

💡 Ein Vorteil sich nahe an der Theorie zu halten: Konvergenz '**wahrscheinlicher**'.

Garantierte Konvergenz

Erkenntnis: alpha mit der Zeit abfallen lassen (scheduler)

Pause

Q-Learning, SARSA, Monto-Carlo

Q-Learning, SARSA, Monto-Carlo

sind sehr ähnliche Arten um unsere Quality zu schätzen!

Q-Learning vs SARSA: E(G')≈max Q vs E(G')≈Q(a',s') nach policy

Q-Learning vs Monte Carlo $E(G') \approx \sum_k \gamma^k r_{t+k}$

Schritt versus Episode

Q-Learning ist ein "**TD** Algorithmus" bei dem **ein Schritt** genügt zum lernen

Dem gegenüber stehen "**MC** Algorithmen" bei dem erst nach **ein**em **Spiel** gelernt wird.

Temporal Difference vs Monte Carlo

Es werden bei Monte Carlo alle **Schritte gemeinsam bewertet,**
was immer zu **deutlich besseren Ergebnissen** führt!

Erinnert an **batch (Haufen von Trainingsdaten gemeinsam)**
Warum war batch gut? 'Stabilisierung' Durchschnitt vermeidet **Ausreisser.**

replay ≈ batch

Monte-Carlo

Kontrast zu globaler Ansicht von gestern (policy evaluation + policy improvement = policy iteration) global:
Kontrast zu **Temporal Difference (ein Schritt)**

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

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. (env.P)

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 / Trajektorie / Pfad / Rollout / Experiment / Run)

replay = [(s,a,r)] (Reihenfolge wichtig!) oder

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

Episoden Replay : kann man später (wieder)verenden (z.B. zum Trainieren)

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

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

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.

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!

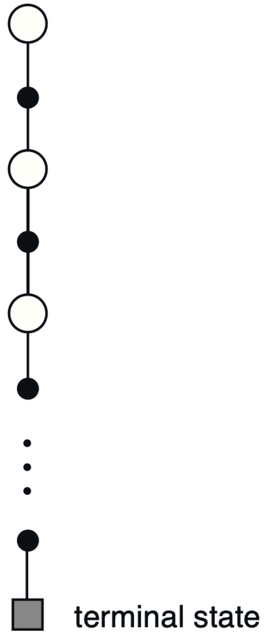
Ziel: vor dem durchspielen der Episoden schon **gute Schätzungen** erhalten.

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!

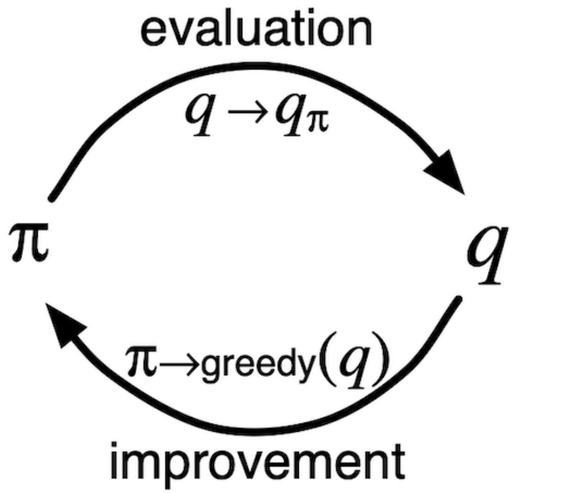
Monte-Carlo

Initialisiere, für alle s∈S, a∈A(s):
  Q(s,a) ← 0
  π(s) ←0
  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 Episode 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)



backup diagram for Monte Carlo estimation of v_π.



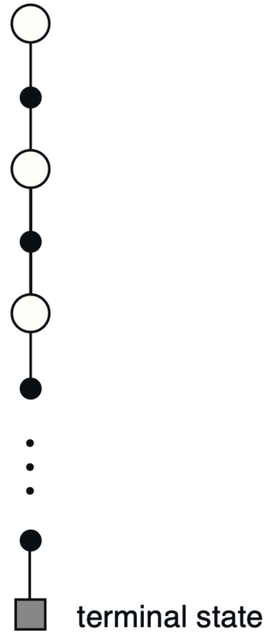
Monte-Carlo

*Monte Carlo bezeichnet sowohl einen konkreten **Algorithmus***
*als auch eine ganze **Klasse von Algorithmen** welche alle die selbe Idee teilen:*
Update von Werten** anhand des über das **gesamte Spiel summierten Gewinnes!

```
# Monte Carlo update
def mc_update(episode_history):
    G = 0 # gain Gewinn
    for state, action, reward in reversed(episode_history):
        G = reward + gamma * G
        q_table[state][action] += alpha * (G - q_table[state][action])
```

Man zählt bei Monte Carlo die Gewinne rückwärts vom Sieg aus zusammen.
Das ist nur aus *Effizienzgründen*, aber ändert an der Mathematik nichts!

O(n) statt O(n^2) quadratisch



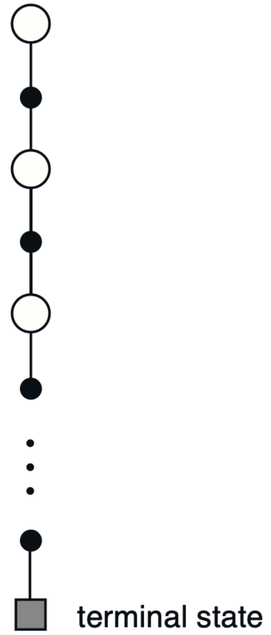
backup diagram for Monte Carlo estimation of v_π.

Monte-Carlo

Genereller Ansatz:

Monte-Carlo
Update nach jedem Spiel
Spiel durchspielen
Pfad (Trajektorie der Episode) aufzeichnen (replay(buffer))
Nach dem Spiel Schätzungen anpassen (Q,V)

replay (buffer) : ein Spiel oder mehrere (**batch**)



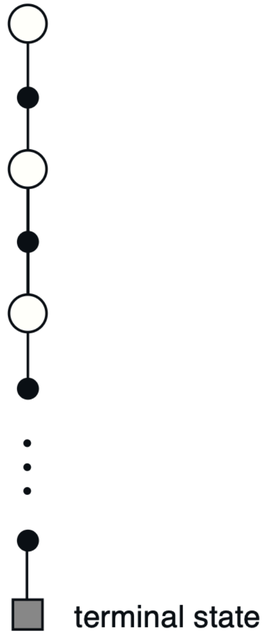
backup diagram for Monte Carlo estimation of v_π.

Kontrast: TD temporal difference
step-wise: Update für/nach jedem Schritt im Spiel
geht auch per batch /replay (steps)

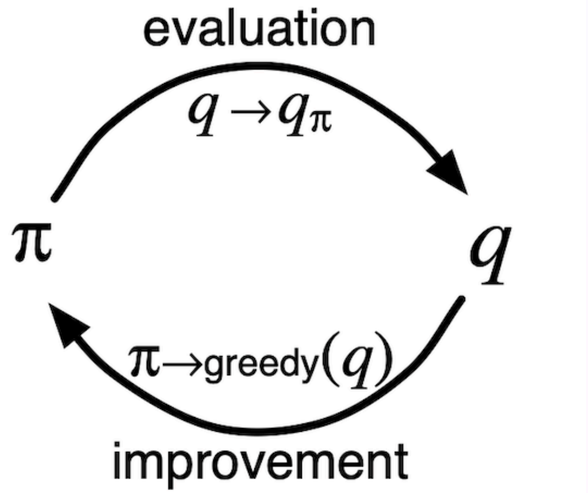
Monte-Carlo

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.



SARSA on policy

💡 Idee als Pseudocode

solange nicht abgeschlossen:

beobachte Zustand **s**

wähle Aktion **a** gemäß Policy **π**(**s**)

führe Aktion a aus

beobachte neue Belohnung **r** und neuen Zustand **s'**

aktualisiere Policy π basierend auf (s, a, **r**, s') < SARSA on policy

State–**a**ction–**r**eward–**s**tate–**a**ction (SARSA) oder

State–action–reward–state–**A**KTUALISIERUNG

SARSA on policy

```python
def sarsa(state, action, next_state, reward):
  # 💡 SARSA State–action–reward–state–action
  # Wende unsere Update Formel mehrfach LIVE "on policy" pro Episode an
  # neue Schätzung für Wert einer Aktion wird gemischt je nach Lernrate alpha und discountfaktor γ
  next_action = policy(next_state)  # sample actual action taken
  next_q = q_values[(next_state, next_action)]
  current_q = q_values[(state, action)]
  # q(a,s) = qk + a * [r + γ maxa q(s', a') - qk(a,s)]
  q_values[(state, action)] = current_q + alpha * (reward + γ * next_q - current_q)
```

Sobald wir **einen** neuen reward erhalten wird unsere Schätzung upgedated

Spielfigur sammelt on policy Schritt für Schritt Informationen.

Monte Carlo update

```python
# Monte Carlo update
def mc_update(episode_history):
    G = 0 # gain Gewinn
    for state, action, reward in reversed(episode_history):
        G = reward + gamma * G
        q_table[state][action] += alpha * (G - q_table[state][action])
```

MC gamma schaut Spiel rückwärts vom **Ziel** aus an:
was unmittelbar zum Schachmatt geführt hat wird am höchsten belohnt.

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?
on policy y/n?
Bootstrapping y/n? 0
Approximation given / exact / tabellarisch / linear / deep
Exploration ε-greedy / normed probs / softmax / …
quality-based / value-based / policy-based / actor-critic
episodenbasiert / Schrittweise (MC vs. TD)
deterministisch / stochastische policy
diskret / kontinuierlich
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)	✔

Zusammenfassung

Damit ist das theoretische Grundwerk gelegt um uns ab nächster Woche vollständig auf praktische Lösungen von RL Problemen dank neuronalen Netzen zu widmen.

Wir **verlassen** damit den Pfad von **Sutton** und der **klassischen Theorie** und **fokussieren uns** auf das Buch **ZAI & BROWN** sowie später aktuellen **Papers.**

Pause

Aufgabe

Copy Paste pole-q_learning-mc.py anwenden auf irgend ein Problem der Welt.

Idealerweise **diskretes gymnasium.**

https://gymnasium.farama.org/environments/toy_text/blackjack/

Gelernte Konzepte

Erweiterte Konzepte

Erweiterte (Hilfs)Konzepte

- **Episode:** "Spiel" Ablauf einer Folge von Aktionen und Zuständen ≈ **Trajektorie** $\tau=(s_0,a_0, \ldots a_t,s_t)$
- **Gain** Gewinn **g:** (Erwartete) *Summe der Belohnungen*, bis zum Ende des Spieles
- **Value Function v**: Erwartete zukünftige Belohnung ab einem **Zustand** ≈
- **Qualität (Quality) q**: Erwartete zukünftige Belohnung für eine **Aktion** in einem Zustand.

Value bewertet eine Position

Quality bewertet einen Zug!

Man kann in **guten** Positionen **schlechte** Züge machen und
in **schlechten** Positionen **optimale** Züge also das 'bestmögliche'.
Ob ein Zug zielführend war entscheidet sich oft erst am Ende des Spiels "Bauernopfer".

Wiederholung

Drei Ansätze für E[G']:

Q(a,s) += alpha * (r + E[G'] - Q(a))
alpha "learning rate" später Adam Optimierer

1.) maximaler zukünftiger Gewinn
E[G'] ≈ max Q(a',s') "klassisches **Q-Learning**"
"greedy" immer der 'besten' Aktion folgen!

2.) nach policy Q(a',s') rekursiv
E[G'] ≈ Q(a',s') "SARSA"

3.) erwartete zukünftige Gewinne ganz anders schätzen

Model

Ein **Modell** ahmt das Verhalten der Umgebung nach oder erlaubt allgemeiner gesagt Rückschlüsse darauf, wie sich die Umgebung verhalten wird. Zum Beispiel könnte das Modell bei gegebenem Zustand und gegebener Aktion den resultierenden nächsten *Zustand* und die nächste *Belohnung* **vorhersagen.**

Dank Deep Learning bilden sich Modelle **automatisch** aus d.h. die Muster welche unsere neuronalen Netze 'erkennen' spiegeln irgendwann idealerweise die Muster der realen Umgebung wieder!

Modellbasierte Methoden verwenden **Planung**, im Gegensatz zu einfacheren modellfreien Methoden, die explizit durch Versuch und Irrtum lernen.

In Chapter 9 we explore reinforcement learning systems that simultaneously learn by trial and error, learn a model of the environment, and use the model for planning.

Aufgaben

Aufgaben Tag 4

0) banditen e-greedy (ohne GPT!)

a) Q-Learning

1) Hyperparameter Grid: wie lange muss man vortrainieren um bei einer Feldgröße von n*n zu positivem Gewinn zu kommen? (Experimentell mit GridExploringStart.py)
   Antwort 1!? warum? hängt von Hyperparametern ab?
2) maximales n? warum?

b) Exploration vs. Exploitation
   1) Q-Updates
   statt wie in GridExploringStart.py alle q-values im Vorfeld zu berechnen machen wir dies nun ad-hoc in jeder Episode (für alle besuchten states/actions).
   Das benötigt vorher ein Anpassen der policy, so dass wir alle actions ausprobieren:
   2) e-greedy
   3) q-greedy
   4) softmax q-greedy

c) Kopiere GridQLearning.py und ersetze q-values durch v-values (Vereinfachung!)

Aufgaben

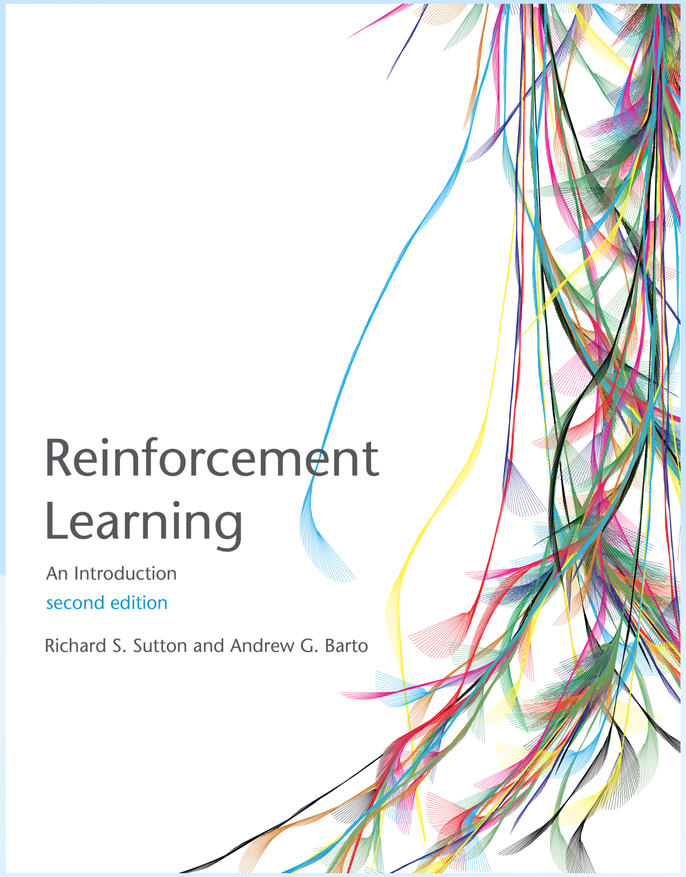
a) Q-Learning

nimm eine der beiden Q-Learning Vorlagen und wende sie auf ein neues Problem an. Idealerweise 'plug-and-play' einfach anderes gymnasium Problem einsetzen. (am besten gleich mc!)

Empfehlung: Sutton

6 Temporal-Difference Learning 119 - 138

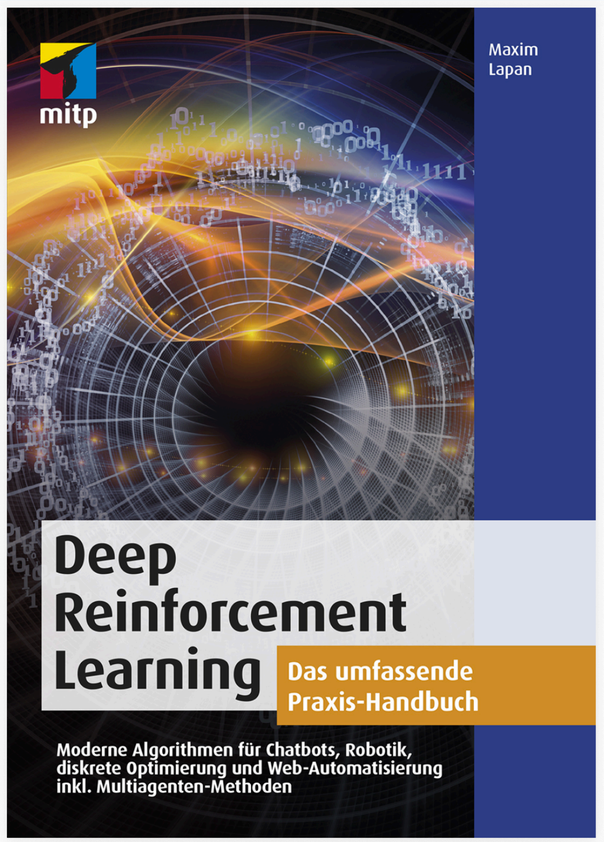
Buch



Sutton: bis Seite **138** (gegebenenfalls nur überfliegen!)

https://www.coursera.org/specializations/reinforcement-learning
https://github.com/ivanbelenky/RL Sutton and Barto book implementation

Buch



Lapan Deep-Reinforcement-Learning: **bis Seite 66 + 120-138** (gegebenenfalls nur überfliegen)

Vokabular

Exploration
Exploitation
Softmax
Temperature

Q-Learning
Monte Carlo
sampling
probing
(episodic/batch) **replay**

Full Sweep
Exploring Starts

TD: Temporal-Difference Learning (Quality Updates)