Reinforcement Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-06 Dozent: Karsten Flügge

Themen des Tages

Tag 6

Deep Reinforcement Learning

Deep Q-Learning

Deep Reinforcement Learning

Es lassen sich alle Hilfskonzepte und Hilfsfunktionen welche wir bis jetzt kennen gelernt haben mit **neuronalen Netz trainieren**:

- **Value** Funktionen,
- **Quality** Funktionen oder
- **Policy** direkt

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

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

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

α learning rate: wie viel wollen wir unsere Schätzung updaten

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

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

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

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 a,s => a',s'

Monte-Carlo

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.

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',r')] oder nennt sich 'episode' oder '**Trajektorie**'

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

Episoden Replay : kann man später (wieder)verenden (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.

Replay Queue

Bei **TD** sind die Schritte unabhängig, daher kann man replay mit mehreren Spielen füllen, bis replay buffer 'voll' ist.

replay.append((s,a,r,s'))

replay queue: z.b. 10000 Einträge

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?
on policy y/n?
Bootstrapping y/n?
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 / 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)	✔

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!

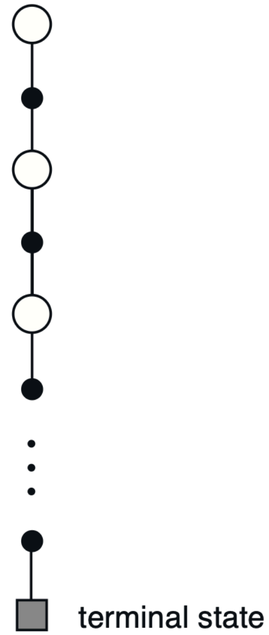
Ziel: vor dem durchspielen der Episoden schon **gute Schätzungen** erhalten.

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!

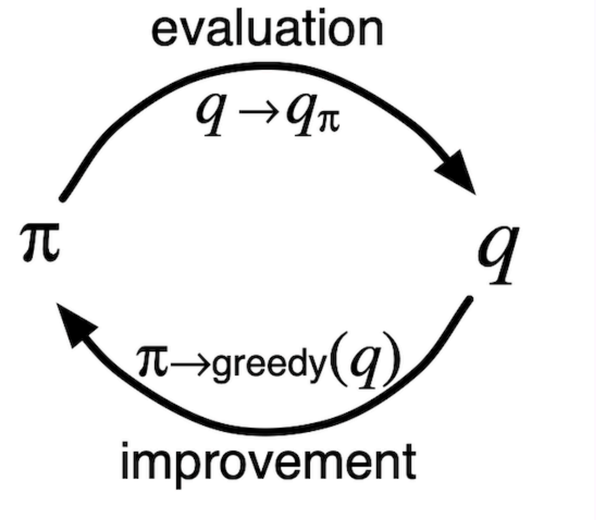
Monte-Carlo

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.



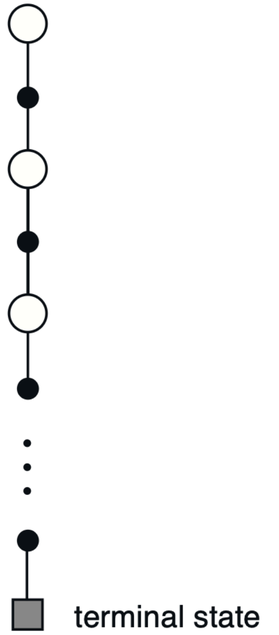
backup diagram for Monte Carlo estimation of v_π.



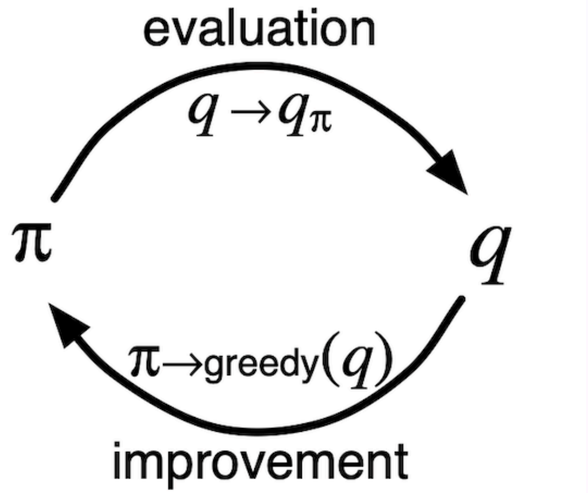
Monte-Carlo

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)



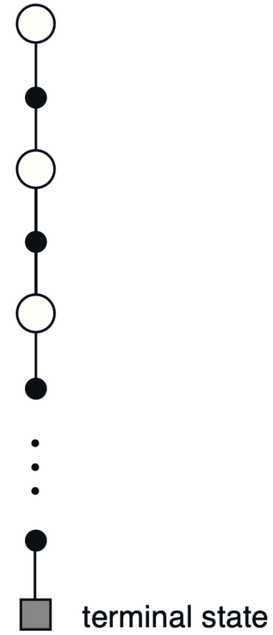
backup diagram for Monte Carlo estimation of v_π.



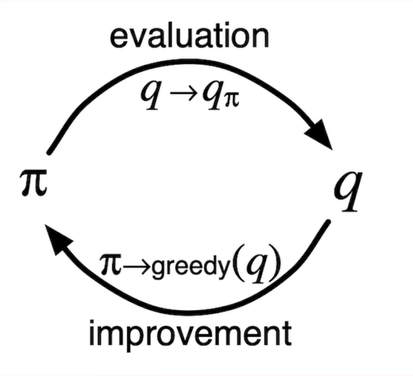
Monte-Carlo

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.

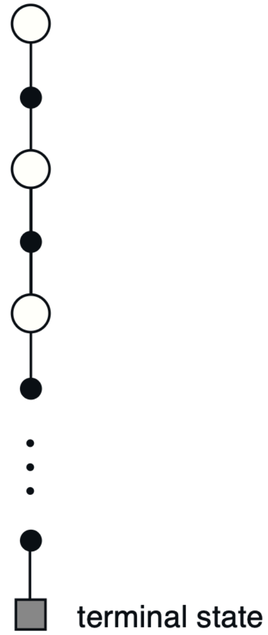


⚠️ unsere Perspektive auf greedy reward ist nun (mit reverse) auf den Kopf gestellt: letzter Zug wird am wichtigsten bei kleinem DISCOUNT_FACTOR. Bei DISCOUNT_FACTOR 1 bleiben alle gleich viel wert

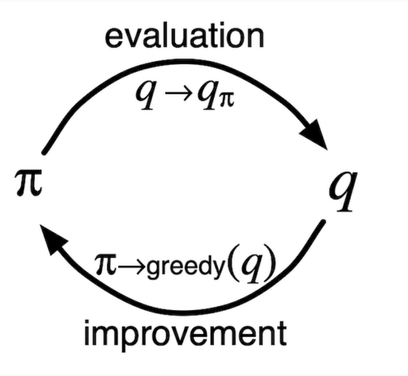
Monte-Carlo + NN

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.

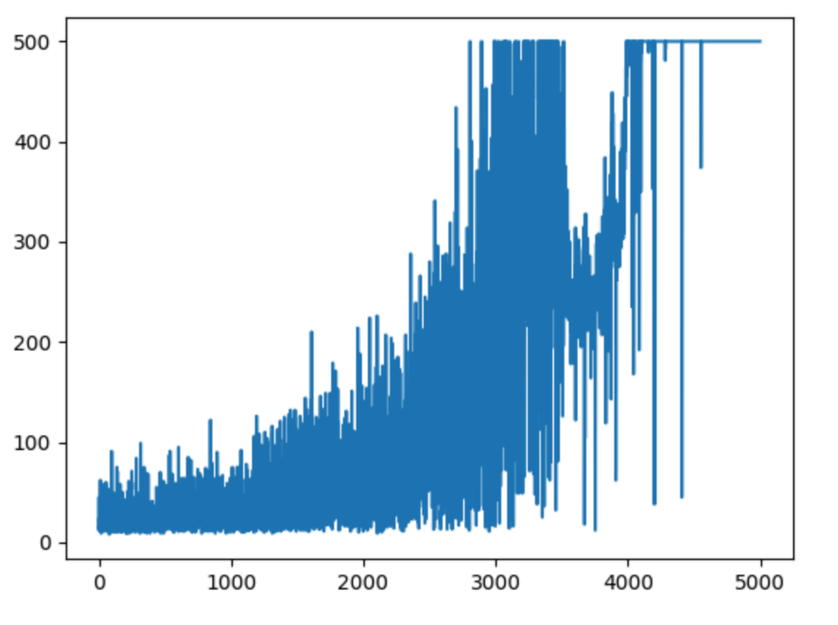


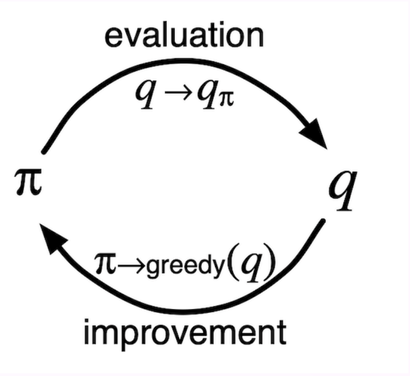
⚠️ unsere Perspektive auf greedy reward ist nun (mit reverse)
auf den Kopf gestellt: letzter Zug wird am wichtigsten
bei kleinem DISCOUNT_FACTOR. Bei DISCOUNT_FACTOR 1 bleiben alle gleich viel wert

Monte-Carlo + NN

Wenige Zeilen Code
Monte Carlo : mächtiger Algorithmus!

```python
# 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
```



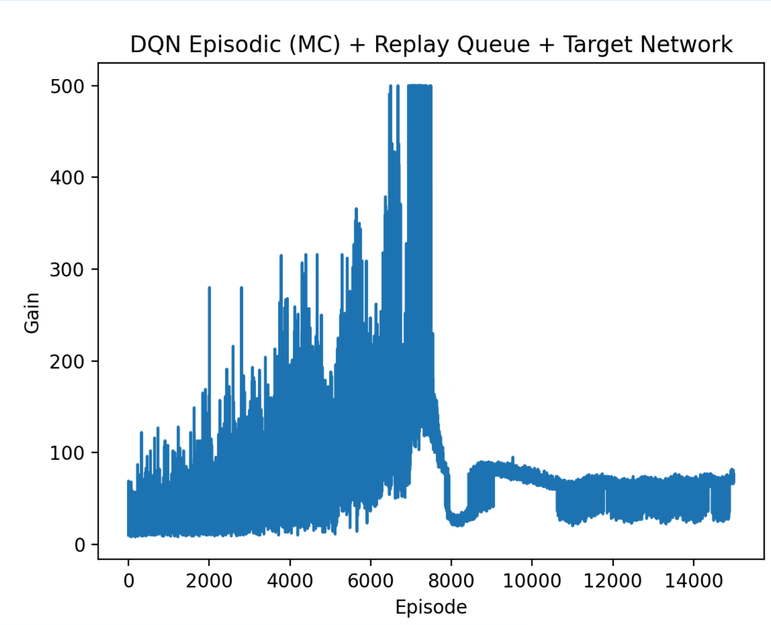


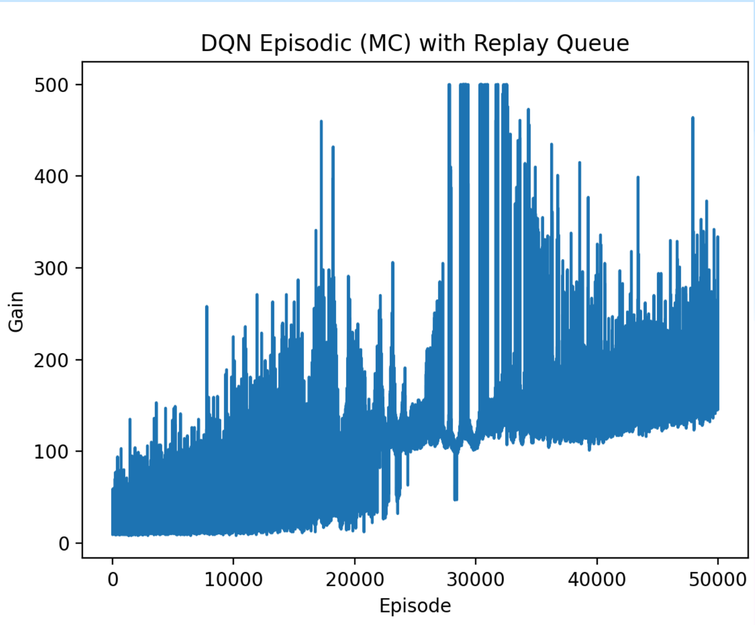

```
gamma = 0.99
lr = 0.001 # learning rate
```

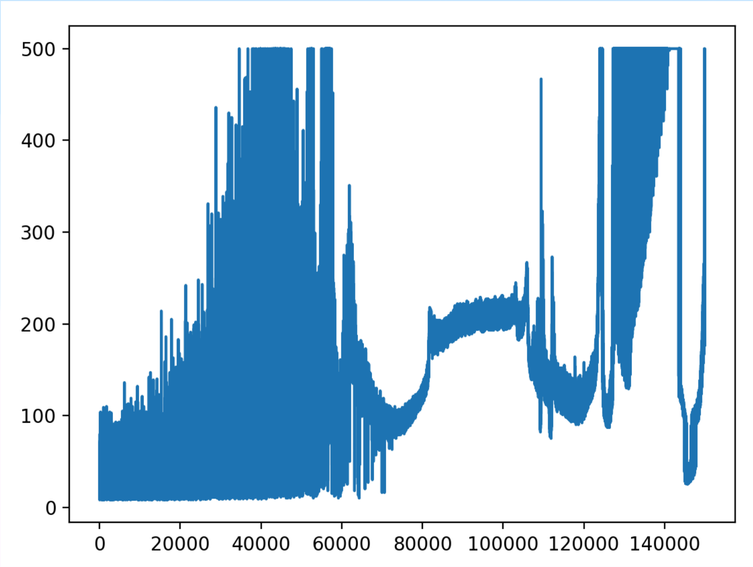
Monte-Carlo + NN

Alle Tricks helfen nicht? Was war problem vor pole_DQN_episodic.py ?
Early Stopping um Katastrophales Vergessen zu vermeiden? Ja, aber gibt besseres (morgen)

```
with torch.no_grad(): # bei Evaluierung keine Gradienten!
```







Ziel: Einfacher Algorithmus der ohne Tricks/Verkomplizierungen gut funktioniert!

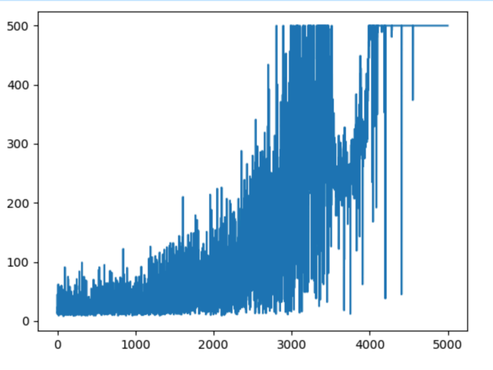
Pause

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)

Laden / Speichern

```
torch.save(q_value_net.state_dict(), save_file) # nur Gewichte ODER
torch.save(q_value_net, save_file) # komplettes model

passend beim Laden:

q_value_net.load_state_dict(torch.load(save_file, weights_only=True))
oder  q_value_net = torch.load(save_file) # whole model
```

Aufgabe



0) Erfolg reproduzieren von pole_DQN_**episodic**.py , dann:

Wende unseren extrem erfolgreichen episodic(MC) Algorithmus auf ein neues Problem an:

Acrobot-v1
oder
Taxi-v3
oder
MountainCar-v0 (schwierig) nachhelfen mit eigenem reward (z.B. speed / distance)

oder

etwas *ganz* anderes!

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. "Temporal Difference"

UND / ODER

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

Grundalgorithmus DQN

Grundalgorithmus DQN eher zum Verständnis, nicht für die Praxis:

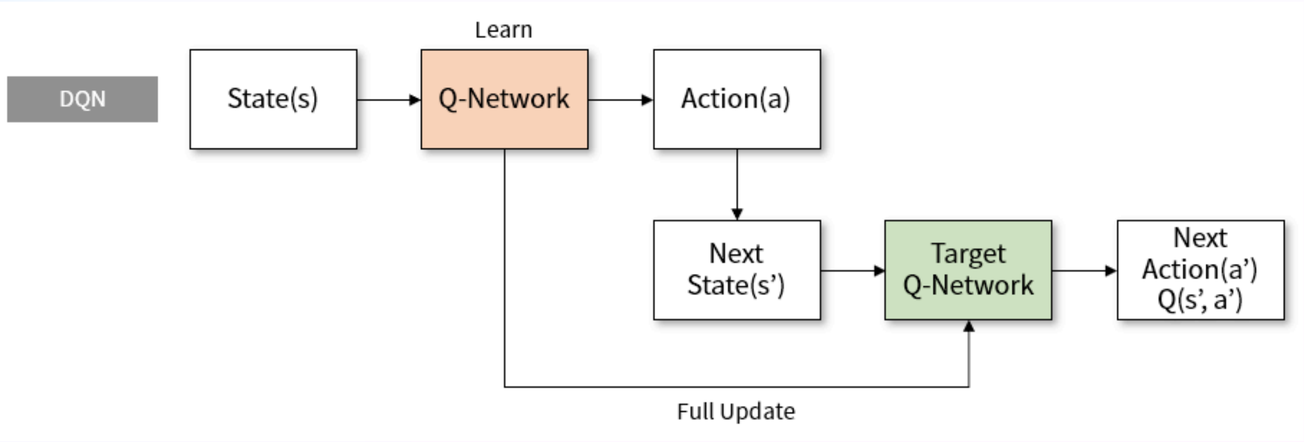
DQN Funktioniert oft nur mit Finetuning und laaaangen Episoden (100000?) oder mit "Tricks" wollen wir vermeiden

- Episodische "Monte Carlo" Variante ist einfacher und funktioniert deutlich besser & schneller (5000 Episoden pole)

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.

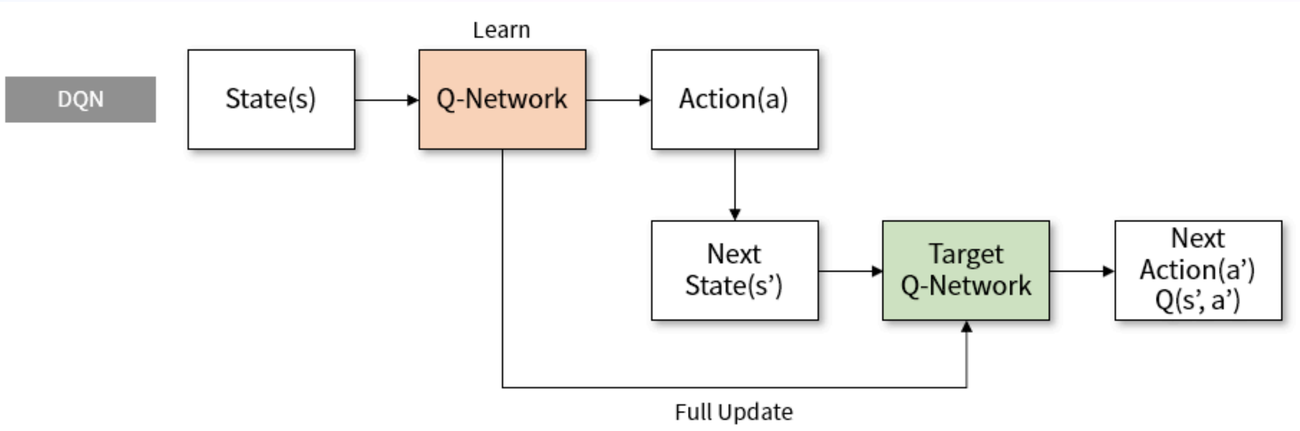


💡 double buffer beim Spielen

Target Network

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

Zwei identische Netze: Eines sammelt die Informationen, das andere wendet sie an.
Update: Gradient verzögert per "update" aufs Q-Netz geschrieben.



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)

DDPG

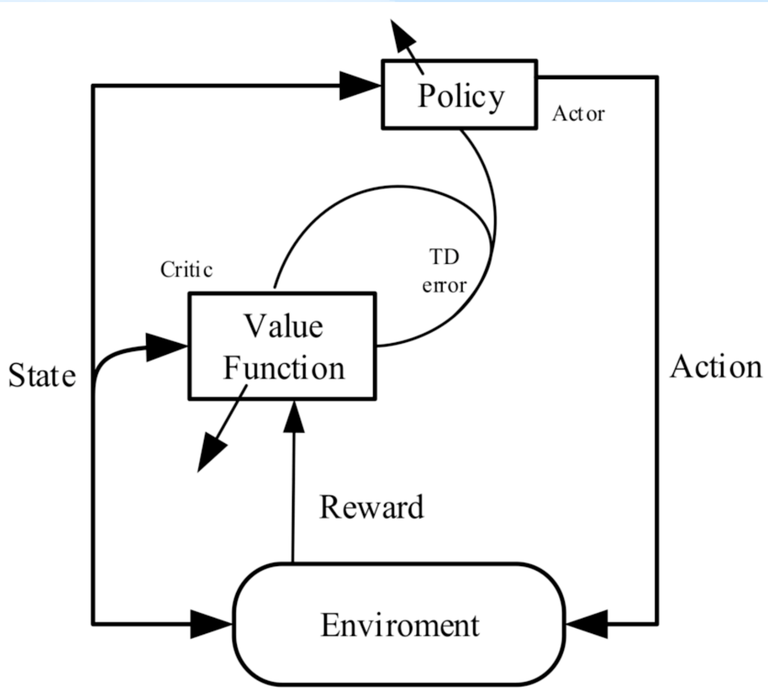
Deep **Deterministic** **Policy Gradient** Methode

Lillicrap et al., 2016

Deep NNs (actor + critic)
Stabilisation tricks: **Replay buffer** + **target networks**
Exploration mechanism: Built-in action **noise** (OU) !

Scales to high-dimensional continuous control

Foundation for modern algos such as **SAC, TD3**



DDPG

Deep Deterministic Policy Gradient as **Foundation** for modern algos:

2018 **TD3** (Twin-Delayed DDPG) ① Twin critics with min-value target, ② actor updated every 2-3 critic steps, ③ Gaussian smoothing on target actions, Cures Q-overestimation, stabilizes training

2018 **D4PG** (Distributed **Di**stributional DDPG) Categorical critic (Z-distribution), N-step returns, distributed actors, PER; Higher throughput and better value estimates

2018 **SAC** (**Soft Actor-Critic**) Stochastic policy with entropy term, double critics; retains off-policy replay Robust exploration, state-of-the-art sample efficiency

2017 **DDPGfD** (DDPG from Demonstrations) **Dual replay** buffers (demo + agent), PER weighting Solves sparse-reward robotics tasks

2017 → MADDPG Centralized critic with all agents' actions, decentralized deterministic actors Mixed cooperative-competitive multi-agent control

2021 TD3 + BC **Behavior-cloning** loss weighted by advantage, feature normalization Strong minimalist baseline for offline RL

2021 DrQ-v2 Heavy image augmentation, n-step returns, DDPG backbone Pixel-based continuous control on one GPU

2020–24 TQC / VI-TD3 / VI-DDPG Quantile truncation or value-improved targets applied to TD3/DDPG critics Sharper value distributions, faster convergence

2024 **P-DDPG** (Prioritized + Param-noise) PER on transitions, parameter-space exploration noise More robust continuous-space agents

2025 DATD3 Depth-wise-attention encoder for observation histories built on TD3 Handles partial observability without RNNs

2025 DBOP-DDPG Swarm-style dung-beetle optimizer;) for actor pre-updates, tri-level priority replay Escapes local optima, accelerates convergence

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an **algorithm** which uses off-policy data and the Bellman equation to learn the Q-function, and uses the **Q-function to learn the policy**.

policy π via best action $a^*(s) = \arg\max_a Q^*(s,a).$

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) \max_{a'} Q_\phi(s',a') \right) \right)^2 \right]$$

d: done 1/0

DDPG can be thought of as being *Deep Q-learning* for *continuous* action spaces.

https://spinningup.openai.com/en/latest/algorithms/ddpg.html

DDPG

Deep Deterministic Policy Gradient.

```
Initialisiere Actor-Netzwerk μ(s|θ^μ) und Critic-Netzwerk Q(s,a|θ^Q)
Initialisiere Zielnetzwerke μ' und Q' mit gleichen Parametern
Initialisiere Replay-Buffer R

für jede Episode:
    Setze Umgebung zurück, initialisiere Startzustand s
    für jeden Zeitschritt:
        Wähle Aktion a = μ(s|θ^μ) + Noise (z.B. Ornstein-Uhlenbeck)
        Führe a aus, beobachte Belohnung r und neuen Zustand s'
        Speichere (s,a,r,s') in R
        Ziehe zufälliges Mini-Batch aus R
        Berechne Zielwert y = r + γ * Q'(s', μ'(s'|θ^μ') | θ^Q')
        Update Critic durch Minimierung von (Q(s,a) - y)²
        Update Actor mit Policy Gradient: ∇θ^μ J ≈ ∇a Q(s,a|θ^Q) ∇θ^μ μ(s|θ^μ)
        Soft Update der Zielnetzwerke:
            θ' ← τθ + (1−τ)θ'
```

DDPG

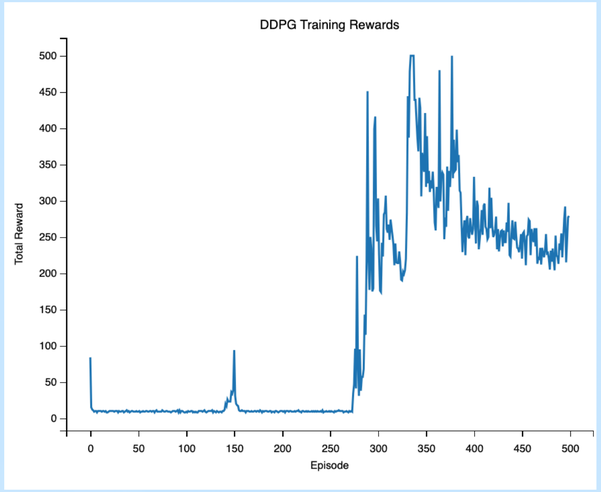
Deep Deterministic Policy Gradient.

Off-Policy, model-free Algorithmus zur Lösung von kontinuierlichen Steuerungsproblemen.

- Kombination aus:
  - Actor-Critic Architektur
  - Deterministic Policy Gradient Methode
- Verwendung von Replay Buffer und Target Networks
- Ursprünglich von DeepMind eingeführt (Lillicrap et al., 2015).

DDPG

```
def DDPGStep(s_t, a_t, r_t, s_t+1, done):
    a_t+1 = target_actor(s_t+1).detach()

    # Q-Wert der nächsten State-Action-Paarung aus target critic
    Q_t+1 = target_critic(s_t+1, a_t+1).detach()

    # Zielwert für critic (Bellman-Backup)
    y_t = r_t if done else r_t + γ * Q_t+1

    # Q(s_t, a_t) aus aktuellem critic
    Q_sa = critic(s_t, a_t)

    # Critic Loss (MSE)
    critic_loss = (Q_sa - y_t)²

    # Actor Loss: ∇a Q(s, a) |_{a=μ(s)}
    a_pred = actor(s_t)
    actor_loss = -critic(s_t, a_pred)

    # Gradient descent Schritte
    actorθ ← actorθ - α_actor · ∇ actor_loss
    criticθ ← criticθ - α_critic · ∇ critic_loss

    # Soft Update der Target-Netze
    for θ, θ_target in zip([actorθ, criticθ], [actorθ_target, criticθ_target]):
        θ_target ← τ · θ + (1 - τ) · θ_target
```



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

Trajektorie

Episodic Sampling vs **Step** Sampling =

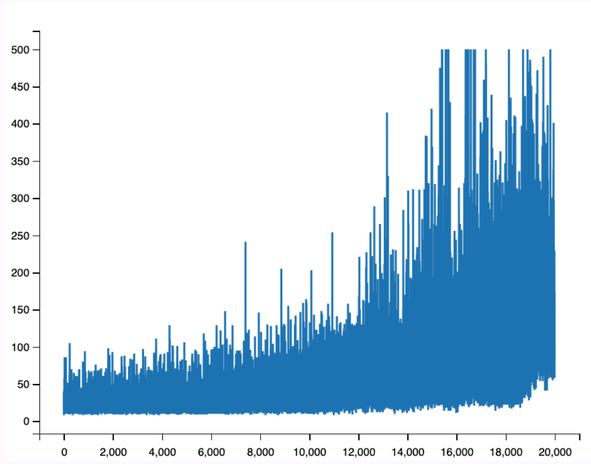
Monte Carlo Sampling vs Temporal Difference Sampling =

Monte Carlo vs Temporal Difference =

MC vs TD

reverse gain

Target Network (Kopie)



DQN convergenz

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