Reinforcement Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

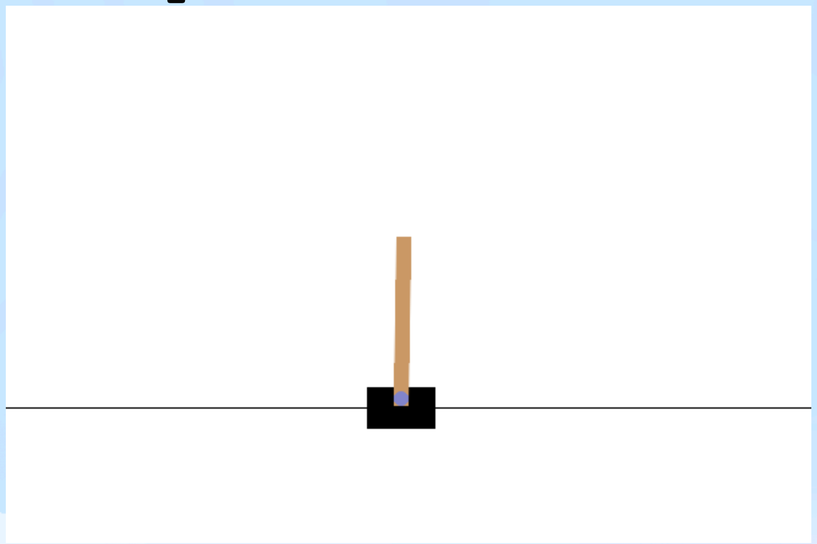
Tag 5

Deep Reinforcement Learning

Deep Q-Learning

Stab balancieren

cart pole



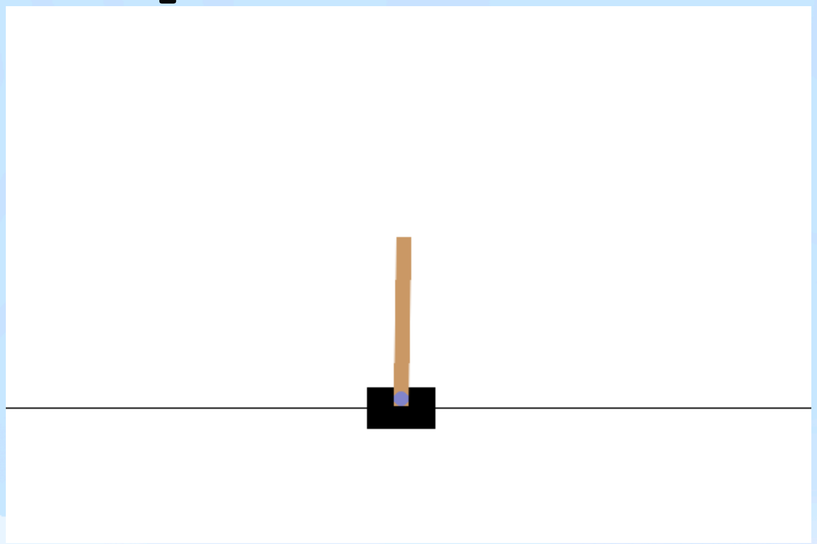
ein Wagen kann nach links oder rechts beschleunigt werden,
um einen Stab zu balancieren.

Wenn der Stab umkippt (Winkel>12/24°) ist Spiel verloren.

https://gymnasium.farama.org/environments/classic_control/cart_pole/

Stab balancieren

cart pole



Recht schwer hier regelbasiert gute Balance zu finden (ausser für Julian;)
=> hier bringt automatisches Lernen echte Vorteile

Können wir Q-Learning von gestern verwenden?
Jein, **Tabellenbasierte verfahren brauchen endliche Zustandsmenge.**
Hier gibt es **kontinuierliche** Winkel etc, aber
man *könnte* es 'diskretisieren' (machen wir aber nicht)
Heisst, man *könnte* Zustandsraum in 'buckets' Eimer/Bereiche einteilen.

Deep Reinforcement Learning

Deep Reinforcement Learning (Deep RL) ist der **Oberbegriff** für eine ganzen Klasse von Ansätzen bei welchen man **neuronale Netze** verwendet um Aspekte des RL abzubilden.

"**Deep**" ist ein Vorweggriff auf tiefe Netzwerk aber man verwendet den Begriff auch für flache Beispiel Netzwerke.

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!

Deep Reinforcement Learning

Riesen Vorteile

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

- **continuous** states
- **Generalisierung**

Deep Reinforcement Learning

Beispiel Algorithmen für die Hauptklassen von Deep Reinforcement Learning

- **Value**
  TD(λ) (Temporal Difference Learning) (eher akademisch) $V(s) \leftarrow V(s) + \alpha (r + \gamma * V(s') - V(s))$
  Monte Carlo Value Iteration
- **Quality**
  - DQN (Deep Q-Network)
  - Dueling DQN (Value & **Advantage**)

- **Policy**
  - Policy Gradient (Dienstag)
  - REINFORCE (Monte-Carlo Policy Gradient)
  - DDPG (Deep Deterministic PG) (Mittwoch)
  - TRPO (Trust Region) <
  - PPO (Proximal Policy Optimization) (Woche 3)

- **Actor-Critic (AC)** (**Mittwoch**)
  - Kombiniert beide Ansätze (Value & Policy)

Orthogonale Klassen / Algorithmen (meist aber mit Deep RL):

Deep Reinforcement Learning

- **Actor-Critic (AC)**
  - Kombiniert beide Ansätze
  - A2C / A3C (Advantage Actor-Critic, synchron/asynchron)

Orthogonale Klassen / Algorithmen (meist aber mit Deep RL):

- **Model**-based RL (Donnerstag) lernt p zu schätzen (MBPO, PETS, Dreamer, MuZero) (Donnerstag)
- **Multi-Agent** Reinforcement Learning (MARL) (kooperativ, kompetitiv oder gemischt) (Freitag)
- **Inverse** Reinforcement Learning (beobachte π, schätze reward) "Absicht" (3te / nächste Woche)
- **Meta**-RL – „Learning to Learn“ (3te / nächste Woche)
- Alpha**Zero** Kombination aus MCTS (Monte Carlo Tree Search) + Policy/Value-Netzwerk
- GAIL (GAN) … …

Deep Reinforcement Learning

- **Value** Networks

TD(λ) (Temporal Difference Learning) **V(s)** ← **V(s)** + α (r + γ * V(s') - V(s)) eher akademisch

Monte Carlo Value Iteration

V(s) als **Hilfsgröße** in A2C oder PPO

Generalized Advantage Estimation (GAE)

⚠️ **Warum sind reine V(s)-Methoden im Deep RL selten?**

Grund	Erklärung
Aktionswahl	Mit $V(s)$ alleine lässt sich **nicht direkt** eine Aktion wählen.
Q(s,a) ist feiner	Bietet direkten Bezug zu Aktionen → besser für Entscheidungsfindung
Value-only ist Policy-abhängig	$V(s)$ hängt stark von der aktuellen Policy ab → instabil beim Lernen

Deep Q-Learning

Deep **Q**-Learning

Teilgebiet vom Deep Reinforcement Learning bei welchem man die **Quality** Funktion mit einem neuronalen Netzwerk schätzt.

Ziel ist es, das Netzwerk die Bellmann Optimalitätsbedingung trainieren zu lassen, also die Differenz (loss) zwischen rechter und linker Seite zu minimieren:

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

Heisst: unsere **Schätzung** ist dann **gut**,

wenn sie die **aktuelle Belohnung r** plus die **beste Belohnung** der **Nachbarschaft** annähert (mal discount).

Deep Q-Learning

Training:

Loss minimieren,
target = r + g*max(Q') "das sei unser guter Schätzwert"

heisst rechte und linke Seite sollen identisch werden:

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

Q = prediction, was unser Netz grad ausspuckt
T = target = r + g*max(Q') 'label', was wir grad berechnen
train(net, Q, T) => **minimiere loss**

z.B. mit Adam und L2 = (**Q** - (r + g*max(Q'))^2 "Mean Square Error" MSE ridge

Deep Q-Learning

Training **Algorithmus:**

Definiere Schätznetzwerk **Q z.B. 3 Layer**
Input = (s,a) meistens flach zB Gridworld (x,y,a) statt ((x,y),a)
Output = ein Wert **: Quality oder alle Action-Qualities um s**

```
def policy(state): # action
  return argmax_a(Q(state,action))

def update(net,action,state,reward, next_state)
  next_q_values = q_network(next_state, all_actions)
  target = r + g * tf.max(next_q_values) # beste nächste quality
  train(net, activation, target)
```

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

```
Training Loop:
  Starte Episode
  state=game.reset()
  While not terminal(state): # solange Spiel läuft
    action = policy(state)
    reward, next_state = environment.do(action)
    update(Q,action,state,reward,next_state) # oder gesammelt per replay-buffer
    state = next_state
```

Deep Q-Learning

Unser Q-Netzwerk

Deep Q-Learning DQL

Deep Q-Network DQN

input: **states, action**
hidden: frei
output: **quality** **q(a|s)** eine einzige Zahl

Oder alle Q-Werte gleichzeitig berechnen:

input: **states**
hidden: frei
output: action-**qualities** [q(a0),q(a1),...] alle gleichzeitig
^^ *besser*! **balancierter, parallelisierbar**

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

Deep Q-Learning

Verbesserter (echter Q-Learning) **Algorithmus:**

```
def policy(state): # action
  return argmax_a(Q(state,action))

def update(net,action,state,reward, next_state)
  batch = random.sample(replay, batch_size)
  next_q_values = model(batch) # tensor!
  target = r + g * tf.max(next_q_values)
  train(net, activation, target)

replay = []
Training Loop:
  Starte Episode
  state=game.reset()
  While not terminal(state) and not max_moves:
    action = policy(state)
    reward, next_state = game.do(action)
    replay += [state,action,reward,next_state]
    state = next_state

  update(Q) #
```

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

Deep Q-Learning

DQN Deep Q-Netzwerk: liefert Q werte zu state-**action pairs**

```
input = ( state-dimension , action-dimension)
layers = …
output = ( q-wert )
```

Gängige alternative:

```
input = ( state-dimension)
layers = …
output = ( action-dimension ) # q-Werte
```

alle Aktion gleichzeitig bewerten!

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

Deep Q-Learning

Algorithmus:

Erste Schätzung: alle Qualities ≈0 ("bootstrapping")
Folge Schätzungen werden besser, weil wir echtes r mit reinbringen

```
train(activation, target) # Differenz implizit im Framework je nach optimizer

activation = Letze Schicht vom Netzwerk
target = r + g*max(Q')
```

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

Train wie immer

```
optimizer = Adam
loss = MSE

def train():
```

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!**

Wie gute Schach spieler: abwechselnd 'rechnen' und 'schätzen'

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.

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

Wichtigster 'trick' beim ML: *batch/replay!*

Buchführung & zusammenfassen von Daten

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

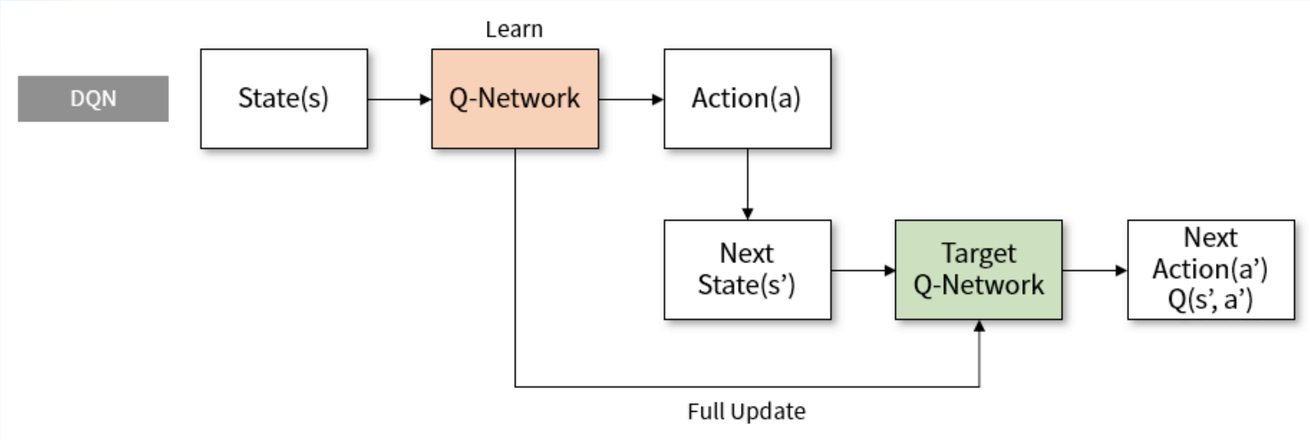
UND / ODER

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

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)

Aufgaben

Aufgaben Tag 5

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'

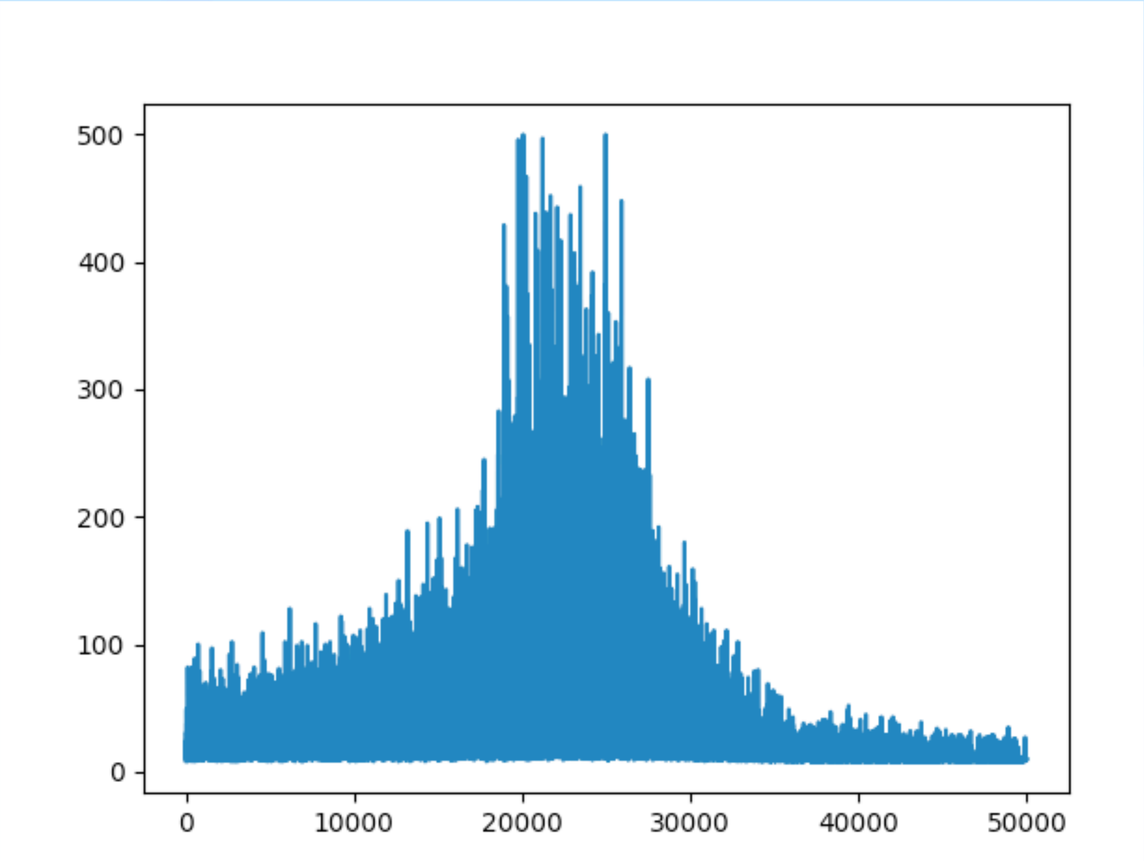
d) baue neue **Feld**art ein, wo man **zufällig** herumgeworfen wird mit zufälligem Reward
p(s',r'|s,a)=random, unabhängig von Aktion auf dem Feld.

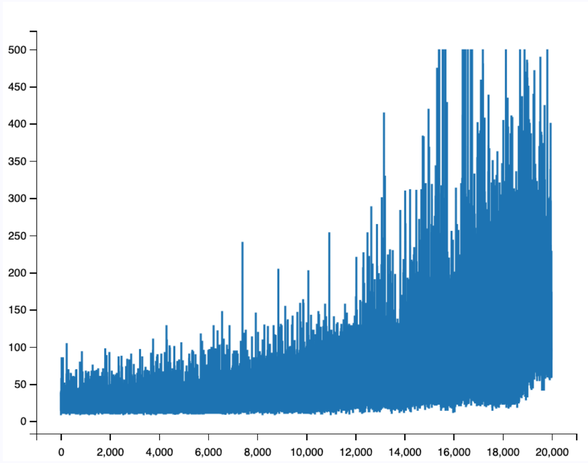
Vokabular

Deep Reinforcement Learning

Deep Q-Learning

Catastrophic forgetting





DQN convergenz

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