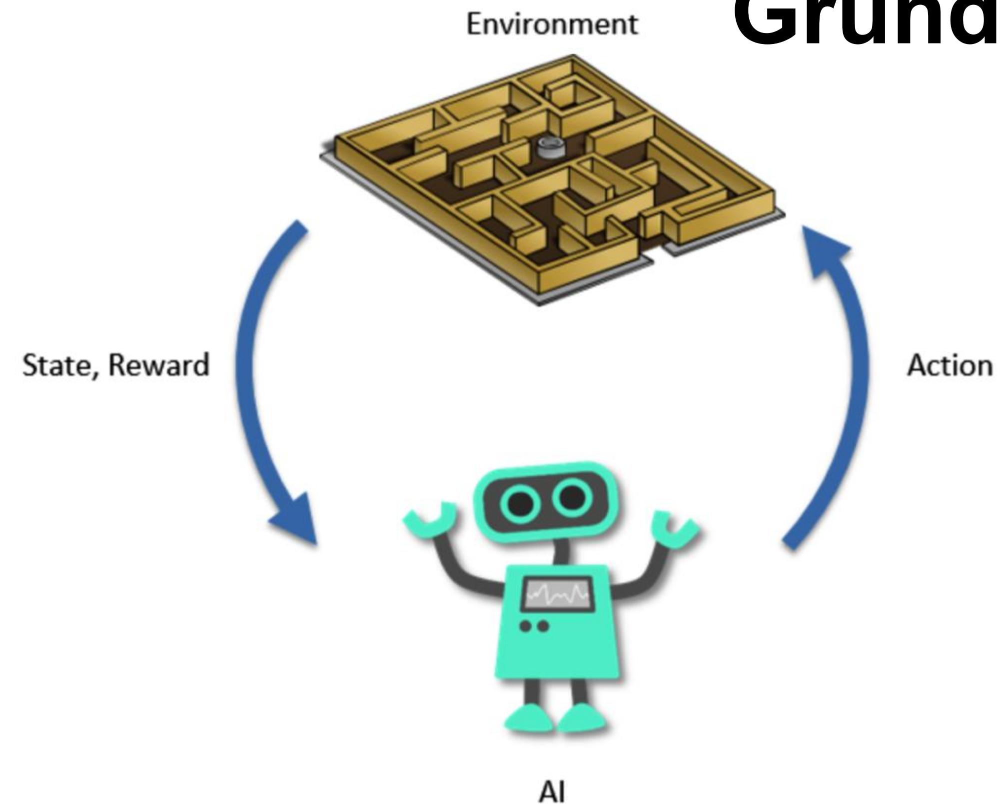
Grundlagen der Künstlichen Intelligenz

Wintersemester 25/26



Vorlesung 14:
Reinforcement Learning

Prof. Dr. Gerhard Neumann

Karlsruher Institut für Technologie

Tagesordnung für heute...

Reinforcement Learning:

- Recap: MDPs und Optimalität
- TD-Learning
- Q-Learning

Approximation der Value Funktion:

- Generalization
- Approximate Q-Learning
- Deep Q-Networks (DQN)

Folien: Basierend auf Sergey Levine, UC Berkeley und RL Bootcamp, UC Berkeley

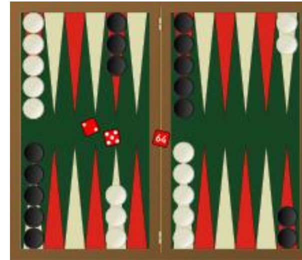
Reinforcement Learning (RL)

Lerne, eine gute Abfolge von Aktionen zu wählen

- Wiederholte Interaktion mit der Welt (wie bei einem MDP)
- Es gibt ein Optimalitätskriterium (wie bei einem MDP)
- Kein (oder nur begrenztes) Wissen darüber, wie die Welt funktioniert

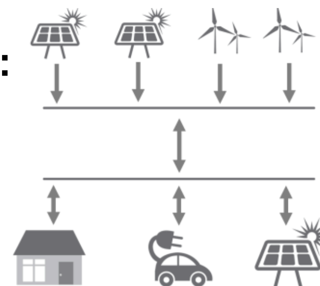
Beispiele für Reinforcement Learning

Brettspiele: Backgammon und Go Weltmeister

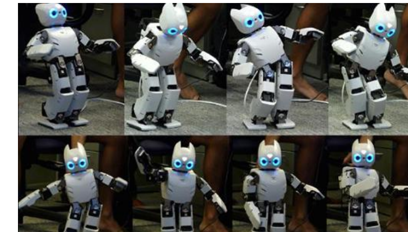


Computer-Spiele

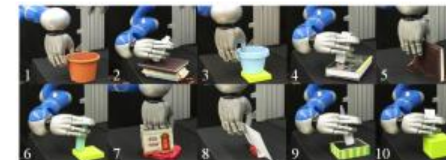
Steuerung von Kraftwerken:



Gehen mit Humanoiden:

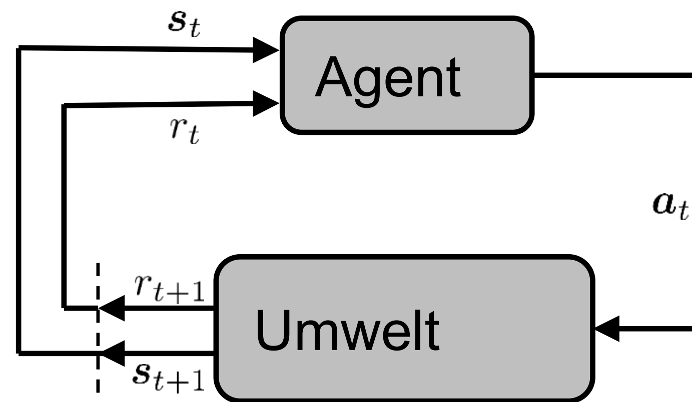


Manipulation von Objekten:



Fine-tuning für LLMs:

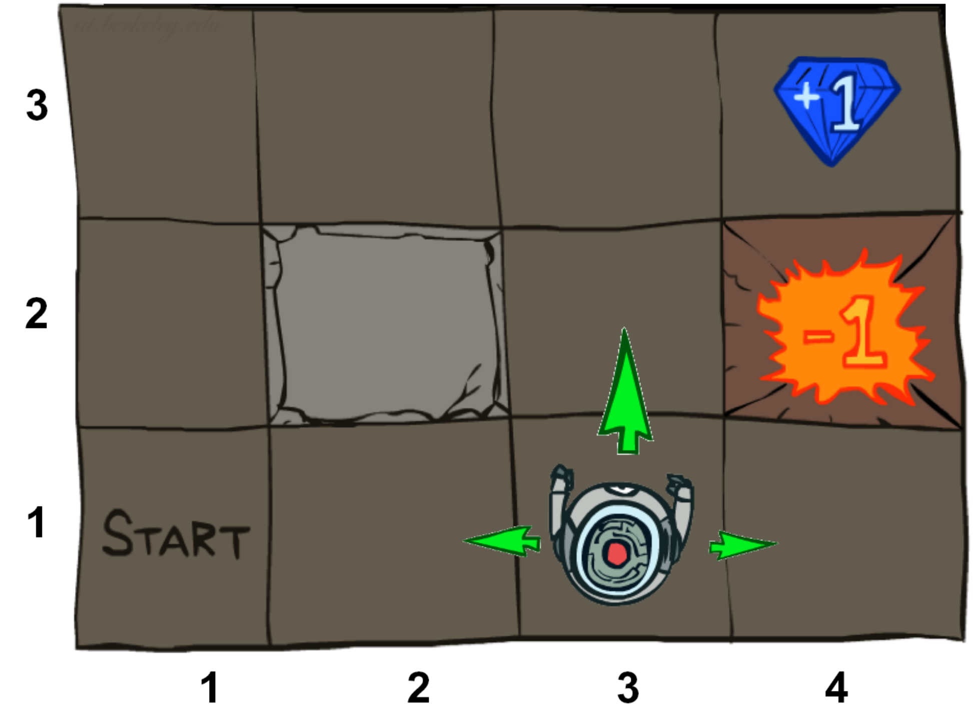
Formale Definition



Markov-Entscheidungsprozesse

Ein MDP ist definiert durch:

- Eine Menge von Zuständen (States) $s \in S$
- Eine Menge von Aktionen $a \in A$
- Eine Transitionsverteilung $p(s'|\, s, a)$
  - Wahrscheinlichkeit, dass a von s nach s' führt
  - Wird auch das "Modell" oder die "Dynamik" genannt
- Eine Belohnungsfunktion $r(s, a)$
- Eine Verteilung $\mu_0(\boldsymbol{s})$ über den Startzustand
- Eine optionale Liste von Endzuständen



Ziel des Agenten

Zielsetzung: Suche nach einer Policy, die den erwarteten kumulativen zukünftigen Reward (auch **Return** genannt) maximiert

$$J_t = \mathbb{E}_\pi \Bigg[\underbrace{\sum_{k=0}^{\infty} \gamma^k r_{t+k}}_{\text{return } G_t} \Bigg]$$

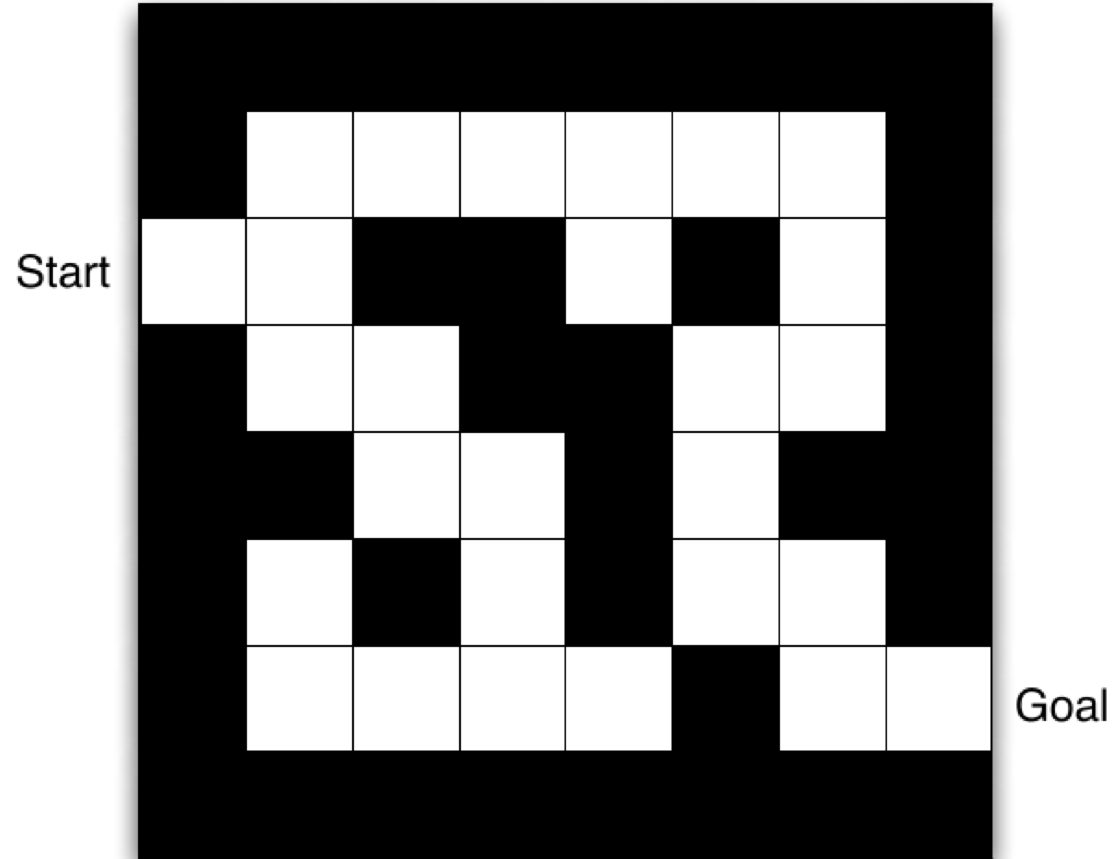
- Discount-Faktor: $0 \leq \gamma < 1$
- Trade-off zwischen der Optimierung der langfristigen ($\gamma \rightarrow 1$) und kurzfristigen ($\gamma \rightarrow 0$) Rewards

Components of a RL Agent

A RL agent consists of at least one of the following components:

- **Policy:** Behaviour of the Agent
- **Value-Function:** Evaluation-function for states and actions

Example



- **Rewards:** -1 per time step
- **States:** (x,y) Position of the agent
- **Actions:** N, S, W, O

Policy

Start

Goal

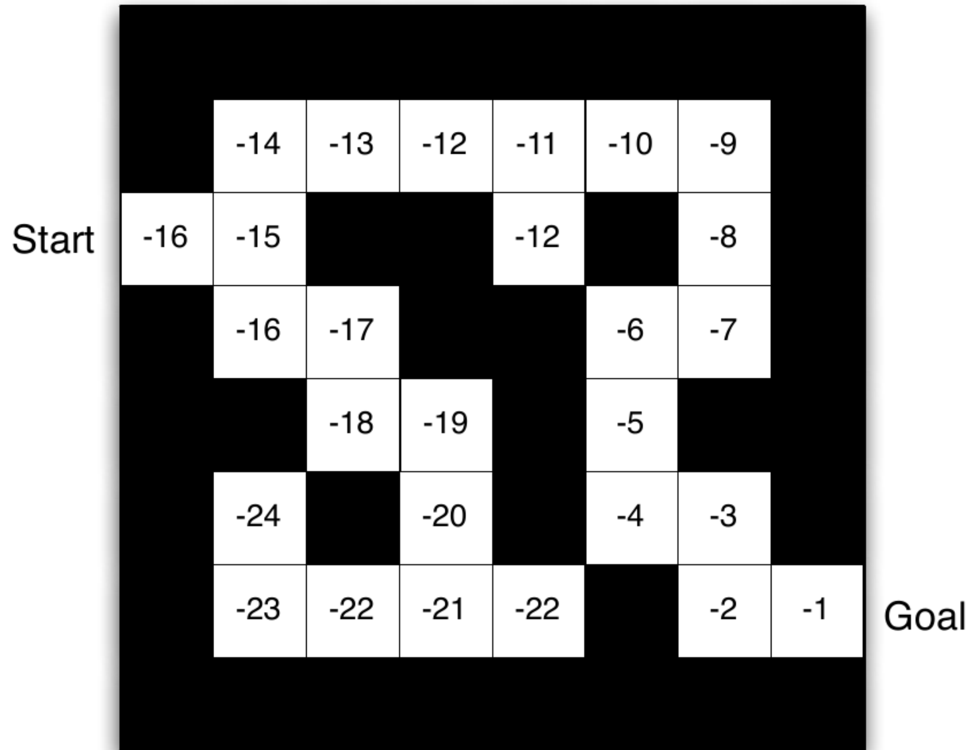
- Represents the behaviour of the agent
- Selects action given state
- **Deterministic:**

$$\boldsymbol{a} = \pi(\boldsymbol{s})$$

- **Stochastic:**

$$\pi(\boldsymbol{a}|\boldsymbol{s})$$

Value Functions



- **V-Function:** Expected future reward if agent is in state $\boldsymbol{s}$ and follows policy π

$$V^{\pi}(\boldsymbol{s}) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t \middle| \boldsymbol{s}_0 = \boldsymbol{s}\right]$$

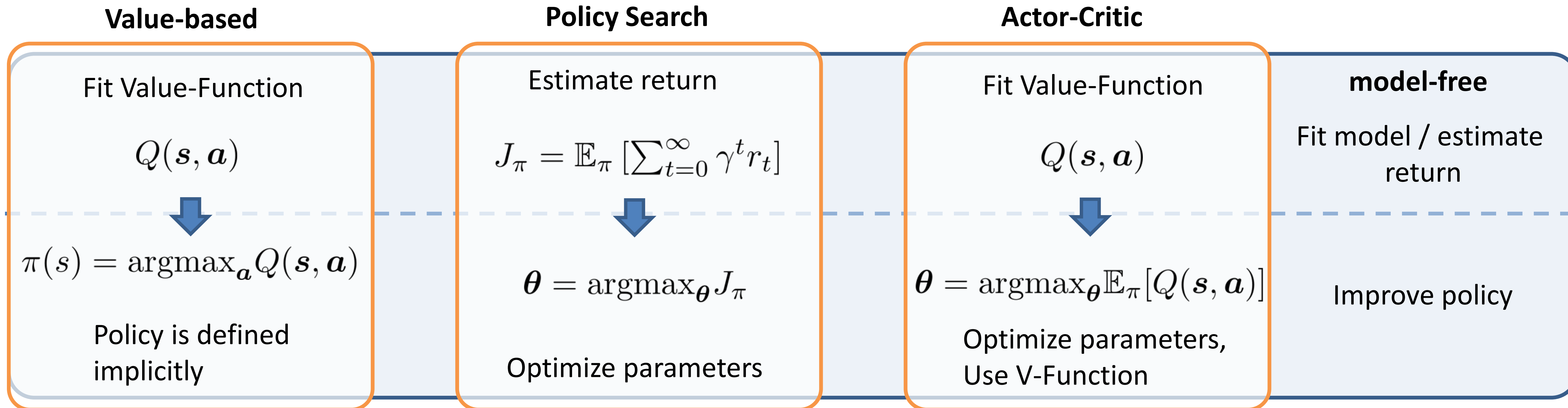
  - **Quality metric for states**

- **Q-Function:** Expected future reward if agent in state $\boldsymbol{s}$, chooses action $\boldsymbol{a}$ and follows policy π afterwards

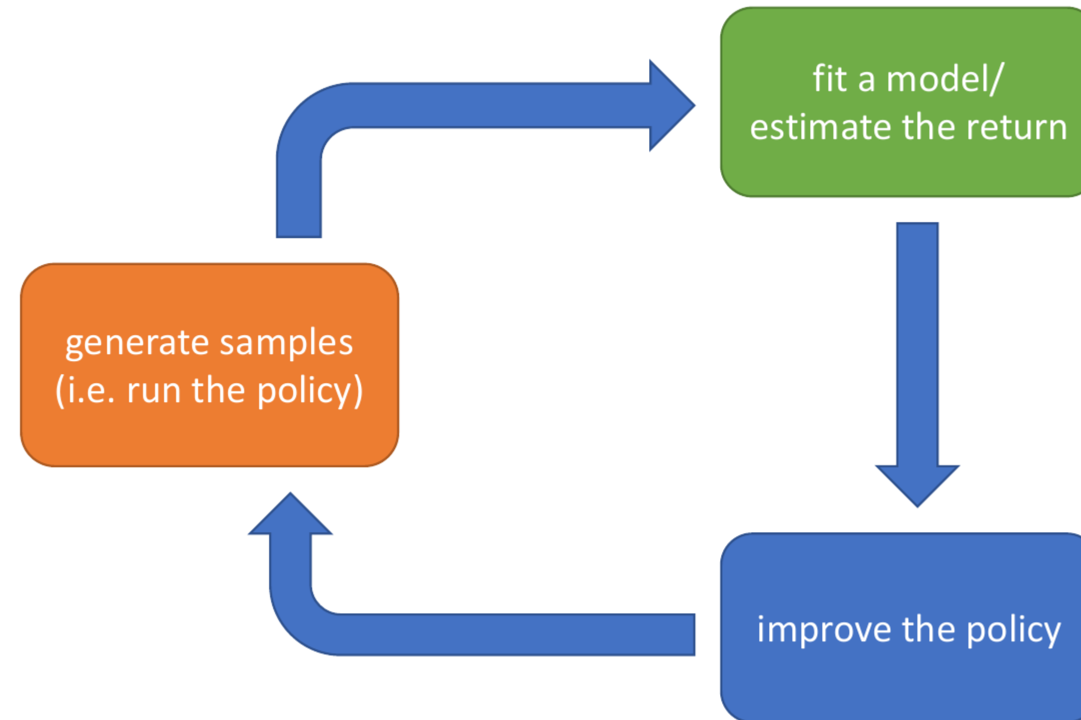
$$Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t \middle| \boldsymbol{s}_0 = \boldsymbol{s}, \boldsymbol{a}_0 = \boldsymbol{a}\right]$$

  - **Quality metric for state-action pairs**

Taxonomy of RL Algorithms



The anatonomy of RL algorithms



Why so many methods?

Different tradeoffs:

- Sample efficiency
- Stability & ease of use

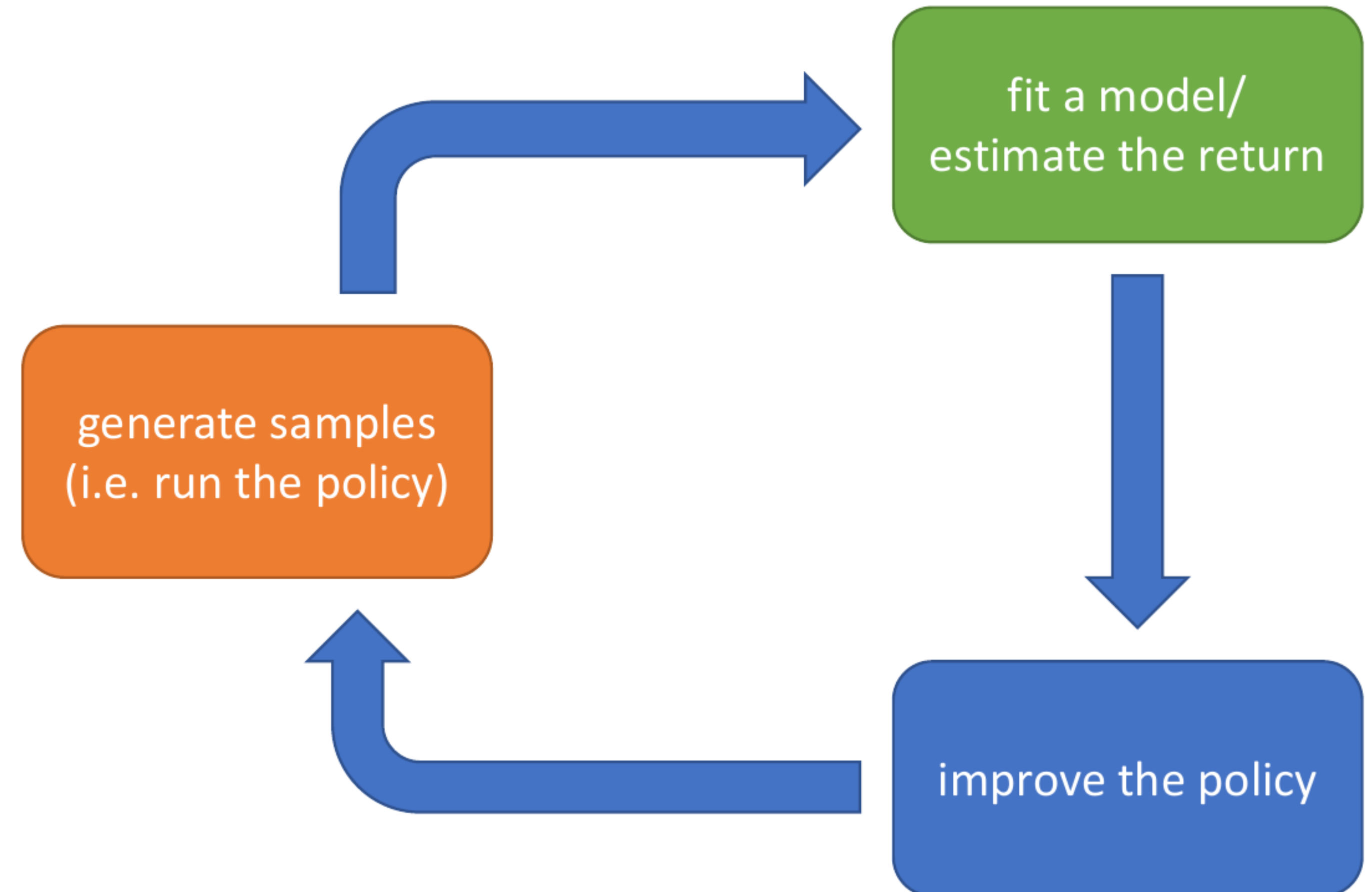
Different assumptions:

- Stochastic or deterministic?
- Continuous or discrete?
- Episodic or infinite horizon?

Different things are easy or hard in different settings

- Easier to represent the policy?
- Easier to represent the model?

We will only cover basic value-based methods in this lecture



Value-basierendes RL -
Der tabellarische Fall

Value-Iteration

Value-Iteration Algorithmus

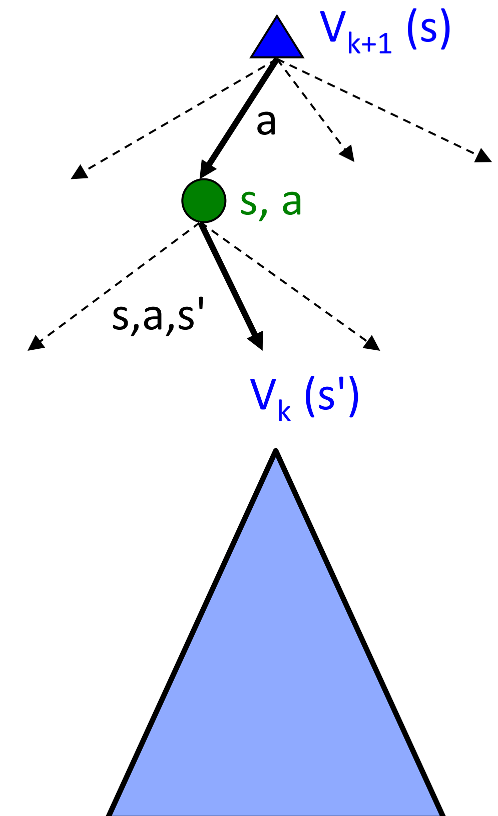
Init: $V_0^\pi(s) = 0$

Repeat

Bei gegebenem Vektor der Values V_{k-1} (s), Berechnung von V_k (s) (für alle Zustände):

$$V_k^*(s) = \max_a \left(r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_{k-1}^*(s') \right)$$

Until Convergence



Komplexität der einzelnen Iterationen: $O(S^2 A)$

Recap: Value-Iteration

Value-Iteration Algorithmus

Init: $V_0^{\pi}(s) = 0$

Repeat

Bei gegebenem Vektor der Values V_{k-1} (s), Berechnung von V_k (s) (für alle Zustände):

$$V_k^*(s) = \max_a \left(r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_{k-1}^*(s') \right)$$

Until Convergence

Exakte Methoden:

- **Value-Iteration**
- **Policy-Iteration**
  (nicht durchgenommen)

Beschränkungen:

- Aktualisierungen erfordern Dynamikmodell
- Iteration über alle Zustände + Aktionen erfordert kleine diskrete Probleme

Lösung:

- Auf Stichproben basierende Annäherungen
- Approximation der Value-Funktionen

Stichprobenbasierte Policy-Evaluation

Wir wollen die Value-Funktion schätzen

$$V^\pi(\boldsymbol{s}) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| \boldsymbol{s}_0 = \boldsymbol{s}\right]$$

- Erwartete zukünftige Belohnung, wenn der Agent sich im Zustand ***s*** befindet und der Policy π folgt
- Mit Hilfe von Sample-Trajektorien (kein Modell P(s'|s,a) verfügbar)

$$\boldsymbol{\tau} = (\boldsymbol{s}_0, \boldsymbol{a}_0, r_0, \boldsymbol{s}_1, \boldsymbol{a}_1, r_1, \dots)$$

2 Optionen:

- Monte Carlo (MC)-Schätzungen
- Lernen mittels Temporal Difference

Monte-Carlo-Schätzung (MC)

Die Values sind nicht bekannt, daher müssen wir Erwartungswerte schätzen

Monte-Carlo-Schätzung:

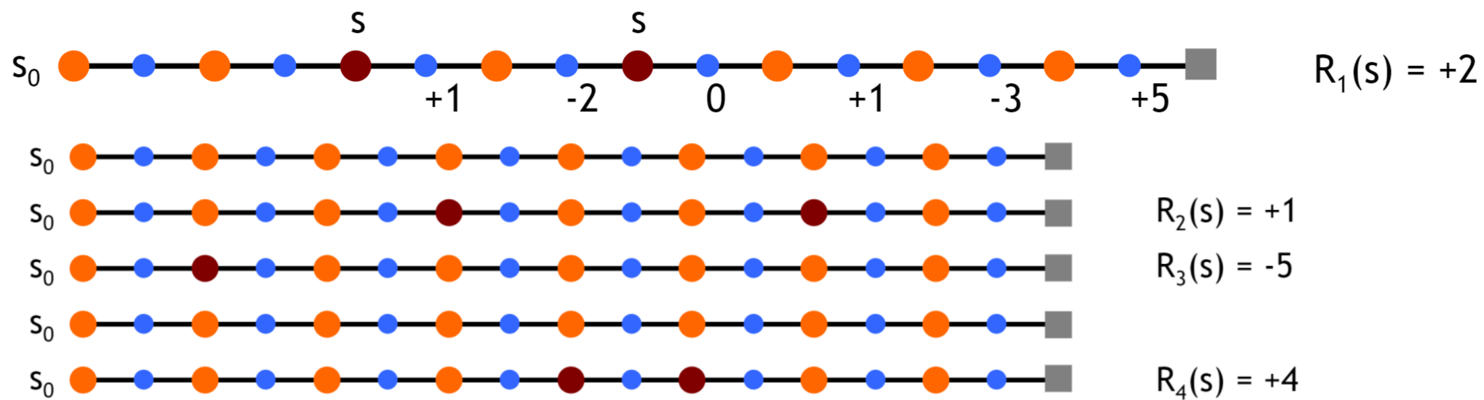
- Wir können einen Erwartungswert immer durch Stichproben approximieren

$$\mathbb{E}_{p(x)}\left[f(x)\right] = \int p(x)f(x)dx \approx \frac{1}{N}\sum_{x_i \sim p(x)} f(x_i)$$

- Auch Sample-Average genannt

Monte-Carlo-Schätzungen

First-Visit Monte Carlo: Mittlere Return nach dem ersten Visits des Zustands ***s***



Besser: Berechnung des dynamischen Mittelwerts

$$V^\pi(s_t) \leftarrow (1-\alpha)V^\pi(s_t) + \alpha R_t$$

Limitierungen:

- Returns sind sehr rauschbehaftet!
  - Stochastizität aufgrund von Policy und Dynamik
  - Summe vieler Schritte, jeder Schritt verursacht Rauschen
- Braucht viele Stichproben!

Sample-basiertes dynamisches Programmieren

- **Policy-Evaluation für die aktuelle Policy π :**

$$V_k^\pi(\boldsymbol{s}) = \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}) \left(r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}'} P(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a}) V_{k-1}^\pi(\boldsymbol{s}') \right)$$
$$= \mathbb{E}_\pi \left[r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \mathbb{E}_P \left[V_{k-1}^\pi(\boldsymbol{s}') \right] \right]$$

- **Die Erwartungswerte können durch Stichproben angenähert werden!**
  - Ziehe Aktion: $\boldsymbol{a}_t \sim \pi_i(\cdot|\boldsymbol{s}_t)$, Ziehe den nächsten Zustand: $\boldsymbol{s}_{t+1} \sim P(\cdot|\boldsymbol{s}_t, \boldsymbol{a}_t)$
  - Berechnung des neuen Targets für den aktuellen Zeitschritt t: $y_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma V^\pi(\boldsymbol{s}_{t+1})$
  - Zielwert durch gleitenden Durchschnitt einbeziehen: $V^\pi(\boldsymbol{s}_t) \leftarrow (1-\alpha) V^\pi(\boldsymbol{s}_t) + \alpha y_t$
    - α ... Lernrate

Lernen mit Temporal Differences (TD)

Der daraus resultierende Algorithmus wird als temporal difference (TD) learning bezeichnet:

$$
\begin{aligned}
V^\pi(\boldsymbol{s}_t) &\leftarrow (1-\alpha)V^\pi(\boldsymbol{s}_t) + \alpha\Big(r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma V^\pi(\boldsymbol{s}_{t+1})\Big) \\
&= V^\pi(\boldsymbol{s}_t) + \alpha\Big(\underbrace{r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma V^\pi(\boldsymbol{s}_{t+1}) - V^\pi(\boldsymbol{s}_t)}_{\delta_t}\Big)
\end{aligned}
$$

δ_t ... wird als temporal difference bezeichnet

Kombination von Monte Carlo (MC) Schätzung und dynamischer Programmierung

- Es wird nur eine Transition verwendet, um die Targets zu erzeugen...
- ... im Gegensatz zu vielen Transitions (ganze Trajektorie) in MC
- Weniger Noise, effizientere Schätzung

TD-Learning

TD-Learning

Init: für alle Zustände s, $V_0^\pi(s) = 0$

For k = 1,2,... until convergence

Ziehe Aktion a und nächsten Zustand s'

falls s' **ein** Endzustand **ist**:

$$\delta = r(\boldsymbol{s}, \boldsymbol{a}) - V^\pi(\boldsymbol{s})$$

Ziehe s' als neuen Ausgangszustand

sonst:

$$\delta = r(\boldsymbol{s}, \boldsymbol{a}) + \gamma V^\pi(\boldsymbol{s}') - V^\pi(\boldsymbol{s})$$

$$V^\pi(\boldsymbol{s}) \leftarrow V^\pi(\boldsymbol{s}) + \alpha\delta$$

$$\boldsymbol{s} \leftarrow \boldsymbol{s}'$$

Anmerkungen

Die V-Funktion kann nicht direkt zur Verbesserung der Policy verwendet werden.

- kein Modell für 1-Step Prediction

Aber: Wir können die gleichen Aktualisierungen für die Q-Funktion vornehmen

- kein Modell erforderlich

Funktioniert nur für diskrete Zustände!

Sample-basierende Q-Value Iteration

Q-Value Iteration:

$$Q_k^*(s,a) = \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q_{k-1}^*(s',a') \right)$$

Dynamischer Mittel:

- Targets $$y_t = r(s_t, a_t) + \gamma \max_{a'} Q^*(s_{t+1}, a')$$
- Aktualisierung: $$\begin{aligned} Q(\boldsymbol{s}_t, \boldsymbol{a}_t) &\leftarrow (1-\alpha) Q(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha \left(r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a}') \right) \\ &= Q(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha \underbrace{\left(r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \max_{\boldsymbol{a}'} \gamma Q(\boldsymbol{s}_{t+1}, \boldsymbol{a}') - Q(\boldsymbol{s}_t, \boldsymbol{a}_t) \right)}_{\text{temporal difference } \delta_t} \end{aligned}$$

Exploration

Trade-Off zwischen Exploration und Exploitation:

- Durch die Verwendung der "greedy Policy" könnten wir bessere Aktionen verpassen, bei denen der Q-Value noch falsch geschätzt wird
- Wir **müssen** also noch **explorieren,** d.h. Aktionen wählen, die suboptimal zu sein scheinen!
- Wann soll man explorieren, wann exploiten? Eines der schwierigsten Probleme im RL!

Explorationpolicies (diskrete Aktionen)

- **Epsilon-Greedy-Policy:**
  - Zufällige Aktion mit Wahrscheinlichkeit ϵ wählen

$$\pi(\boldsymbol{a}|\boldsymbol{s}) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}|, & \text{if } \boldsymbol{a} = \text{argmax}_{\boldsymbol{a}'} Q^\pi(\boldsymbol{s}, \boldsymbol{a}') \\ \epsilon/|\mathcal{A}, & \text{otherwise} \end{cases}$$

- **Soft-Max-Policy:**
  - Höherer Q-Value, höhere Wahrscheinlichkeit

$$\pi(a|s) = \frac{\exp\left(Q(s,a)/\beta\right)}{\sum_{a'} \exp\left(Q(s,a')/\beta\right)} \qquad \beta \ldots \text{ temperature}$$

(Tabellarisches) Q-learning

Q-Learning Algorithmus

Init: $Q(s,a) = 0$ für alle Zustände s und Aktionen a

For k = 1,2,... until convergence

Ziehe Aktion a mit Exploration-Policy, und nächsten Zustand s'

wenn s' **ein** Endzustand **ist**:

$$\delta = r(\boldsymbol{s},\boldsymbol{a}) - Q(\boldsymbol{s},\boldsymbol{a})$$

Ziehe s' als neuer Ausgangszustand

sonst:

$$\delta = r(\boldsymbol{s},\boldsymbol{a}) + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}',\boldsymbol{a}') - Q(\boldsymbol{s},\boldsymbol{a})$$

$$Q(\boldsymbol{s},\boldsymbol{a}) \leftarrow Q(\boldsymbol{s},\boldsymbol{a}) + \alpha\delta$$

$$\boldsymbol{s} \leftarrow \boldsymbol{s}'$$

Anmerkungen

Q-Learning ist ein Beispiel für TD-Learning

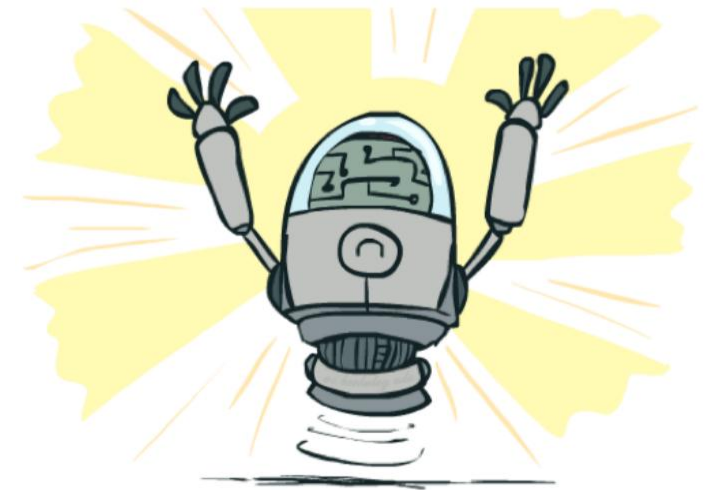
Q-learning ist einer der bekanntesten und am häufigsten verwendeten RL-Algorithmen

Q-Learning: Eigenschaften

Erstaunliches Ergebnis: Q-Learning konvergiert zu einer optimalen Strategie - selbst wenn man (aufgrund von Exploration) suboptimal agiert!

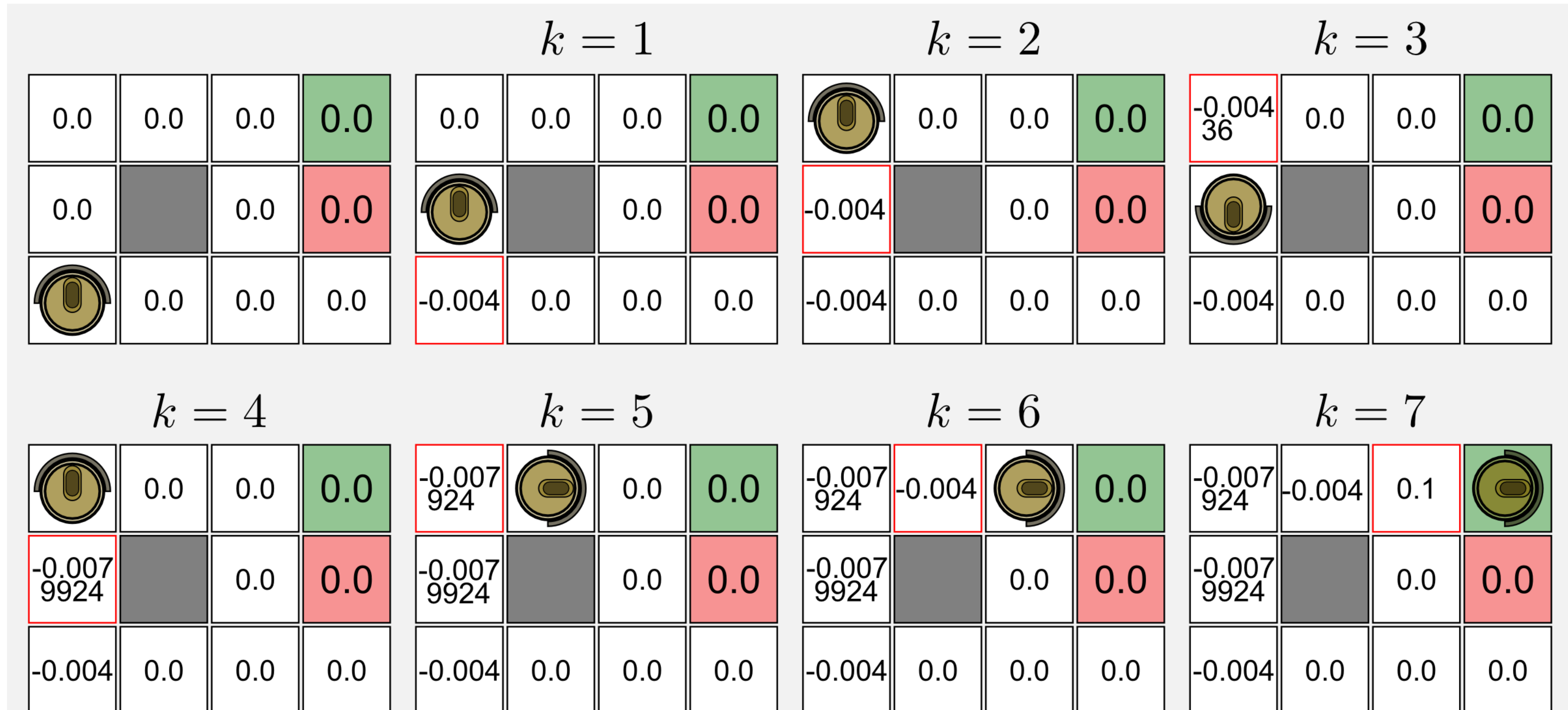
Aber:

- Es muss genug exploriert werden ...
- jedes Zustands-Aktionspaar muss unendlich oft besucht werden (laut Theorie)



Beispiel: Gridworld

- Discountfactor: 0.9, Lernrate: 0.1, “Living-Reward”: -0.04



Zusammenfassung

Wir können V- und Q-Funktionen abschätzen, indem wir nur gesehene Transitionen verwenden

i. Monte-Carlo-Schätzungen (hohe Varianz)
ii. TD-Learning und Q-Learning sind stichprobenbasierte Versionen von dynamischen Programmieren
   – Beide verwenden den Temporal Difference Fehler zur Aktualisierung der Values
   – Q-Learning konvergiert zur optimalen Policy (Dies gilt nur für den tabellarischen Fall!)
- Kein Modell erforderlich!

Unterschied zum dynamischen Programmieren?

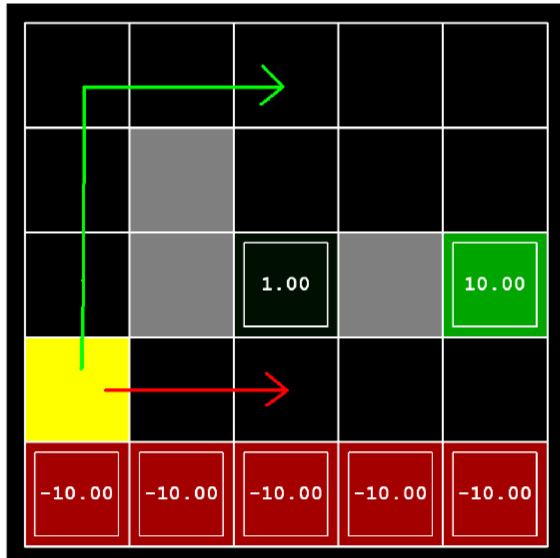
- Wir müssen explorieren!
- Wir müssen dafür sorgen, dass wir jedes Zustands-Aktions Paar ausreichend oft besuchen!

Value-basierendes RL -
Approximation der Valuefunktion

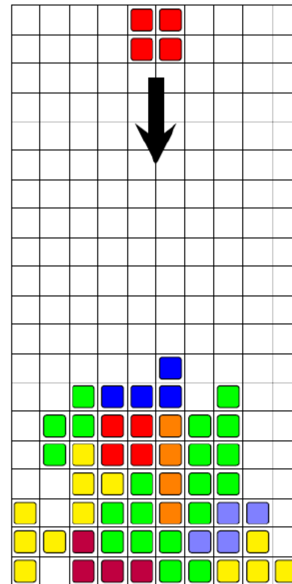
KIT

Können tabellarische Methoden skaliert werden?

- **Diskrete Systeme**



Gridworld
10^1

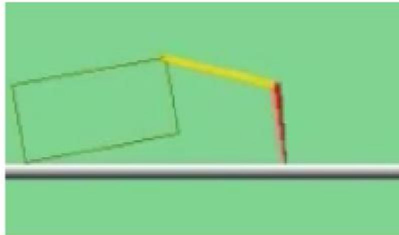


Tetris
10^60

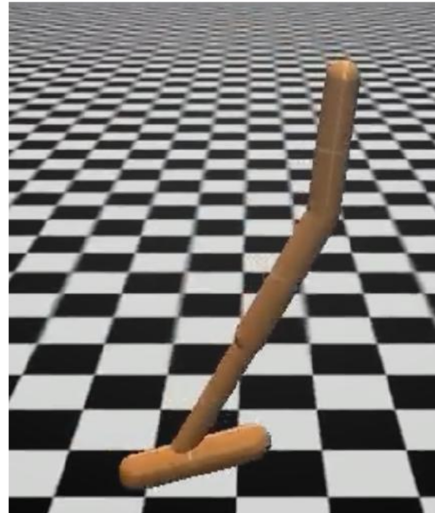
Atari
10^308 (ram) 10^16992 (pixels)

Können tabellarische Methoden skaliert werden?

- **Kontinuierliche Umgebungen (grobe Diskretisierung)**



Crawler
10^2



Hopper
10^10

Humanoid
10^100

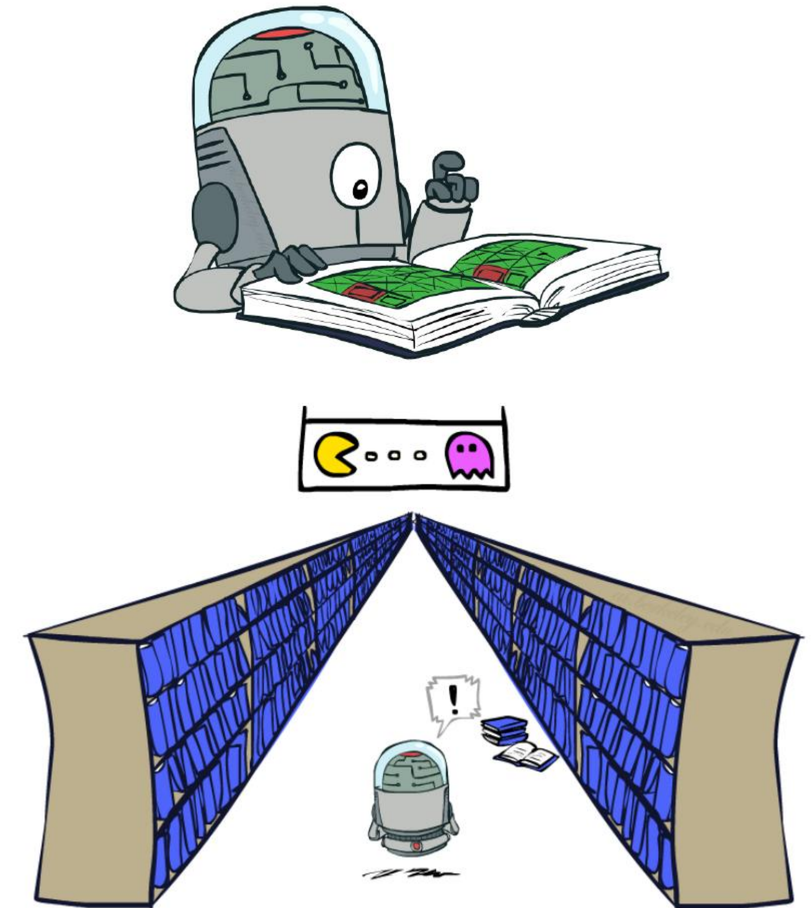
Generalisierung über Zustände

Basic Q-Learning führt eine Tabelle mit allen Q-Werten:

- In realistischen Situationen können wir unmöglich über jeden einzelnen Zustand Bescheid wissen!
- Zu viele Zustände, um sie alle im Training zu besuchen
- Zu viele Zustände, um die Q-Tabellen überhaupt zu speichern

Stattdessen wollen wir verallgemeinern:

- Lernen aus Erfahrung an einer kleine Anzahl von Trainings-Zuständen
- diese Erfahrung auf neue, ähnliche Situationen verallgemeinern
- Dies ist ein Grundgedanke des maschinellen Lernens



Value-Funktion Approximation

Anstelle einer Tabelle haben wir eine parametrisierte Q- (oder V-) Funktion:

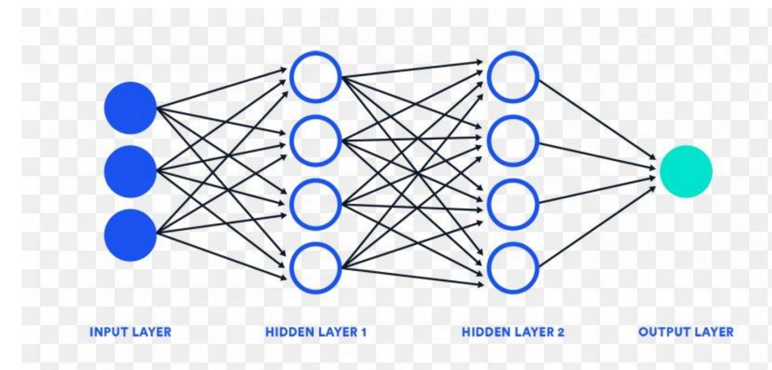
- **Kann eine lineare Funktion von Features sein**

Features oder Basisfunktionen

$$Q_{\boldsymbol{\beta}}(\boldsymbol{s}, \boldsymbol{a}) = \phi_1(\boldsymbol{s}, \boldsymbol{a})\beta_1 + \phi_2(\boldsymbol{s}, \boldsymbol{a})\beta_2 + \cdots = \sum_{d=1}^{D} \phi_d(\boldsymbol{s}, \boldsymbol{a})\beta_d$$

- **Oder ein komplexes neuronales Netz** $Q_{\boldsymbol{\beta}}(\boldsymbol{s}, \boldsymbol{a})$

Parameter der Q-Funktion



Beispiel für Features: Tetris

Zustand:

- naive Brettkonfiguration + Form der fallenden Figur
- etwa 10^{60} Zustände!

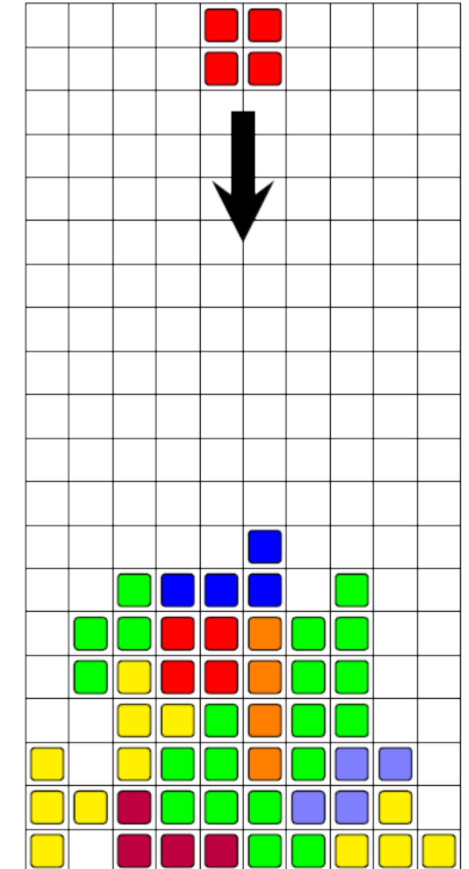
Aktion:

- Rotation und Translation, die auf das fallende Stück angewendet werden

22 Features alias Basisfunktionen

- Zehn Basisfunktionen, 0, . . . , 9, bilden die Höhe h[k] der einzelnen Spalten ab.
- Neun Basisfunktionen, 10, . . . , 18, bilden die absolute Differenz zwischen den Höhen aufeinanderfolgender Spalten ab: |h[k+1] - h[k]|, k = 1, . . . , 9.
- Basisfunktion 19, maximale Säulenhöhe: $\max_k$ h[k]
- Basisfunktion, 20: Anzahl der "Löcher" im Brett abbildet.
- Basisfunktion, 21: Konstant 1

$$V_{\boldsymbol{\beta}}(\boldsymbol{s}) = \sum_{d=1}^{D} \phi_d(\boldsymbol{s})\beta_d$$



Approximate Q-Learning

Tabellarisches Q-Learning:

9.41	9.51	9.61	9.70	9.80
9.32		9.70	9.80	9.90
9.41		1.00		10.00
9.51	9.61	9.70	9.80	9.90
-10.00	-10.00	-10.00	-10.00	-10.00

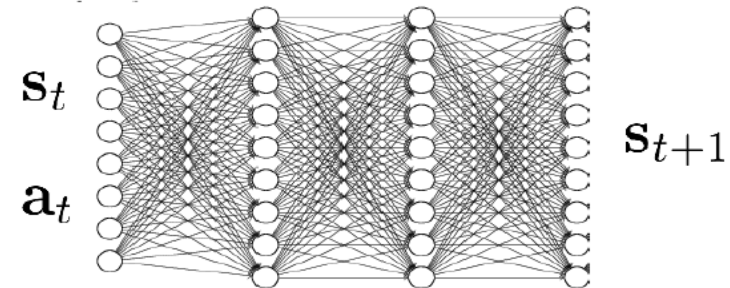
- Berechne Targets

$$y_t = r(s_t, a_t) + \gamma \max_{a'} Q^*(s_{t+1}, a')$$

- Q-Value mit dynamischen Mittel aktualisieren:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) Q(s_t, a_t) + \alpha y_t$$

Approximate Q-Learning:



- Berechne Targets:

$$y_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}')$$

- Quadratischer Fehler:

$$L_t = \left(Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}) - y_t \right)^2$$

- Q-Funktion mit Gradientenabstieg aktualisieren

$$\boldsymbol{\beta}_{\text{new}} = \boldsymbol{\beta} - \alpha \frac{dL_t}{d\boldsymbol{\beta}}$$

Q-Learning mit "Gradient Decent"

Q-Funktion mit Gradientenabstieg aktualisieren

$$\boldsymbol{\beta}_{\text{new}} = \boldsymbol{\beta} - \alpha \nabla_{\boldsymbol{\beta}} L_t = \boldsymbol{\beta} - \alpha \frac{d}{d\boldsymbol{\beta}} \left(Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - y_t \right)^2$$

$$= \boldsymbol{\beta} - 2\alpha \left(Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - y_t \right) \frac{d}{d\boldsymbol{\beta}} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}_t)$$

Steigung der Q-Funktion

Targets einsetzen, um die folgende Regel zu erhalten:

$$\boldsymbol{\beta}_{\text{new}} = \boldsymbol{\beta} + \alpha \left(r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}') - Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}_t) \right) \frac{d}{d\boldsymbol{\beta}} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}_t)$$

$$= \boldsymbol{\beta} + \alpha \delta_t \frac{d}{d\boldsymbol{\beta}} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_t, \boldsymbol{a}_t)$$

Temporal Difference (TD) Error

Approximate Q-Lernen:

- Multipliziere den Gradienten der Q-Funktion mit dem Error und verwende Gradient zur Aktualisierung des Parametervektors

Approximate Q-Learning

Q-Learning-Algorithmus

Init: β **z.B. mit kleinem Noise**

For k = 1,2,... until convergence

Ziehe Aktion a mit Explorationspolicy und nächsten Zustand s'

Berechnen Sie den TD-Error:

$$\delta = r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}}(\boldsymbol{s}', \boldsymbol{a}') - Q_{\boldsymbol{\beta}}(\boldsymbol{s}, \boldsymbol{a})$$

Aktualisierung des Parametervektors

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \alpha \delta \frac{d}{d\boldsymbol{\beta}} Q_{\boldsymbol{\beta}}(\boldsymbol{s}, \boldsymbol{a})$$

$$\boldsymbol{s} \leftarrow \boldsymbol{s}'$$

Anmerkungen

Der Einfachheit halber haben wir die Endzustände vernachlässigt

Funktioniert es? NEIN!

Q-Learning mit neuronalen Netzen ist dafür bekannt, dass es sehr leicht divergiert!

- Es können sehr einfache MDPs konstruiert werden, bei denen dies der Fall ist
- Selbst wenn die optimale Value-Funktion dargestellt werden könnte durch $Q_{\boldsymbol{\beta}}(s, a)$

Okay, warum ist das so? Gradientenabstieg konvergiert doch immer, richtig?

- Ehm... Es ist eigentlich kein echter Gradientenabstieg ☹

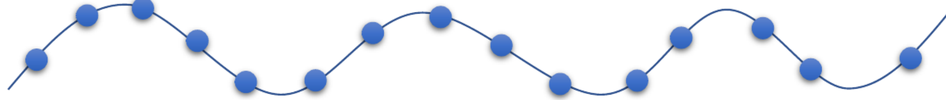
2 Hauptprobleme:

- sequentielle Zustände sind stark korreliert

- der Targets ändern sich ständig

Replay-Buffer

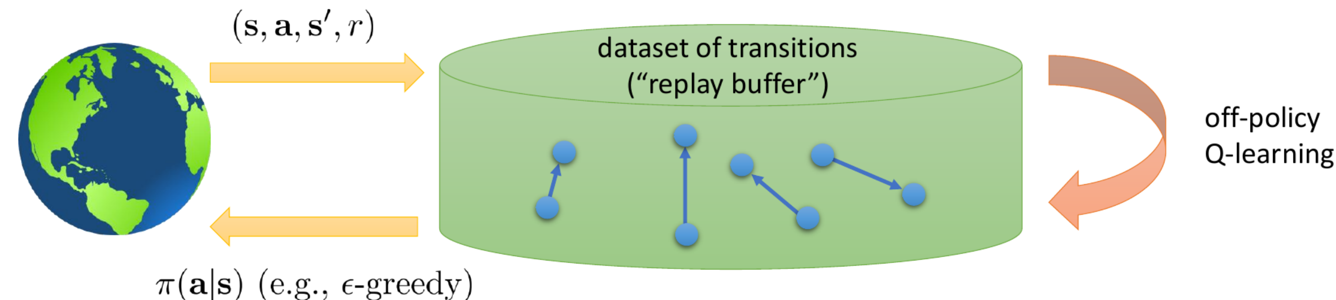
Problem: Sequentielle Zustände sind stark korreliert



- Die Q-Funktion wird viele Male für ähnliche Zustände aktualisiert....
- ... während es die Q-Values für andere Zustände vergisst (Catastrophic Forgetting genannt)

Lösung: Q-Learning mit Replay-Buffern

- Wiederholung von Erfahrungen (Lin, 1993).
- Wurde früher “blos” für eine bessere Dateneffizienz verwendet
  - Aber essentiell für RL mit DNNs



Approximate Q-Learning mit Replay Buffern

Q-Learning mit Replay Buffer

Init: $\boldsymbol{\beta}$ **z.B. mit kleinem Noise**

For k = 1,2,... until convergence

Ziehe Aktion a mit Explorationspolicy und nächsten Zustand s'

Füge Transition $(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')$ zu Buffer B hinzu

Ziehe Mini-Batch aus dem Buffer: $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}_i')\}_{i=1\ldots K}$

Aktualisierung des Parametervektors

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \alpha \sum_{i=1}^{K} \left(r_i + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i', \boldsymbol{a}') - Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i, \boldsymbol{a}_i) \right) \frac{d}{d\boldsymbol{\beta}} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i, \boldsymbol{a}_i)$$

Übergang zum nächsten Zustand: $\boldsymbol{s} \leftarrow \boldsymbol{s}'$

Anmerkungen

Häufige Variante:

- Buffer vor der Aktualisierung mit weiteren Übergängen auffüllen

Macht die Datenverteilung stationärer:

- Vermeidet Catastrophic Forgetting

Verbessert auch die Dateneffizienz erheblich

- Nennt sich "off-policy" update, da Daten verwendet werden die nicht von der momentanen Policy kommen

Target-Networks

Problem: Die Targets ändern sich ständig

- Durch die Änderung von $\boldsymbol{\beta}$, ändern wir auch die Zielwerte y_t
- Dies ist besonders wahrscheinlich, da der nächste Zustand s' typischerweise sehr ähnlich zu s ist
- Wir machen eigentlich *keinen* Gradientenabstieg

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{K} \left(Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i, \boldsymbol{a}_i) - \left(r_i + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}}(\boldsymbol{s}'_i, \boldsymbol{a}') \right) \right)^2$$

Hier wird kein Gradient berechnet!

Lösung: Verwende einen älteren Satz von Gewichten $\boldsymbol{\beta}'$, um die Targets zu berechnen (Targetnet):

- Verhindert, dass sich die Targets zu schnell ändern.

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{K} \left(Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i, \boldsymbol{a}_i) - \left(r_i + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}'}(\boldsymbol{s}'_i, \boldsymbol{a}') \right) \right)^2$$

Die Targets ändern sich nicht!

Der Deep Q-Networks (DQN) Algorithmus

Q-Learning mit Replay Buffer

Init: $\boldsymbol{\beta}$ **z.B. mit kleinem Noise**

For k = 1,2,... until convergence

Alle N Schritte, Target-Network speichern $\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta}$

Ziehe Aktion a mit Explorationspolicy und nächsten Zustand s'

Füge Transition $(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')$ zu Buffer B hinzu

Ziehe Mini-Batch aus dem Buffer: $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}_i')\}_{i=1\dots K}$

Aktualisierung des Parametervektors

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \alpha \sum_{i=1}^{K} \left(r_i + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{\beta}'}(\boldsymbol{s}_i', \boldsymbol{a}') - Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i, \boldsymbol{a}_i) \right) \frac{d}{d\boldsymbol{\beta}} Q_{\boldsymbol{\beta}}(\boldsymbol{s}_i, \boldsymbol{a}_i)$$

Übergang zum nächsten Zustand: $\boldsymbol{s} \leftarrow \boldsymbol{s}'$

Anmerkungen

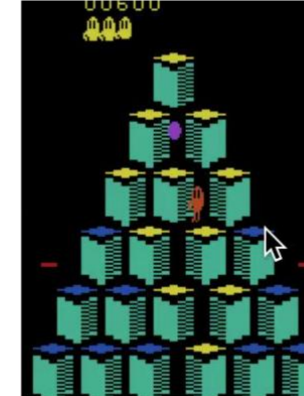
- Für ein gegebenes Targetnetwork sind die DQN-Updates genau wie Regression
- Dennoch gibt es keinen Beweis für Konvergenz, wenn wir die Targets ändern, und es könnte sogar zu Divergenz kommen.
- In der Praxis funktioniert das oft gut

DQN auf Atari

Pong

Enduro

Beamrider


Q*bert

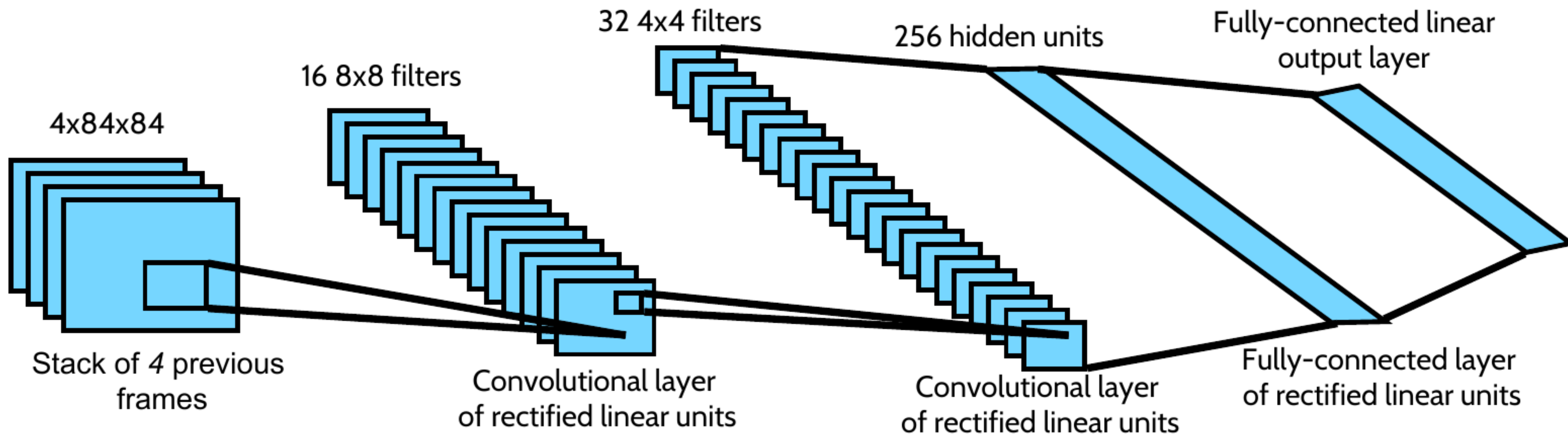
- 49 ATARI 2600-Spiele.
- Von Pixeln zu Aktionen.
- Die Veränderung der Punktzahl ist die Belohnung.
- Gleicher Algorithmus.
- Gleicher Funktionsapproximator, mit 3M freien Parametern.
- Gleiche Hyperparameter.
- In 29 von 49 Spielen, annähernd dem menschlichen Niveau

"Human-Level Control Through Deep Reinforcement Learning", Mnih, Kavukcuoglu, Silver et al. (2015)

Atari-Netzwerkarchitektur

Architektur des CNNs:

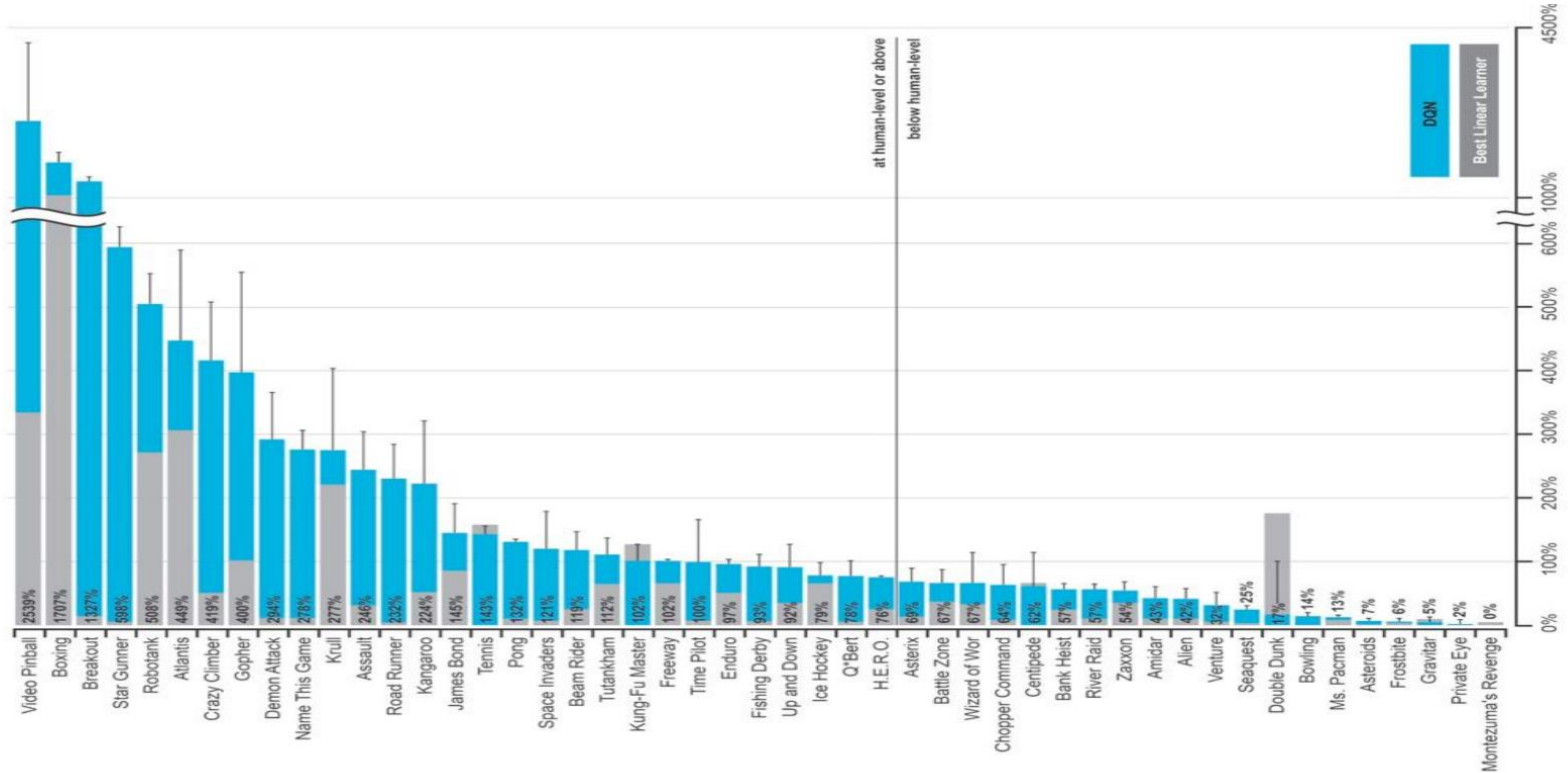
- Historie der Bilder als Eingabe.
- Eine Ausgabe pro Aktion – erwarteter Return für diese Aktion Q(s, a).



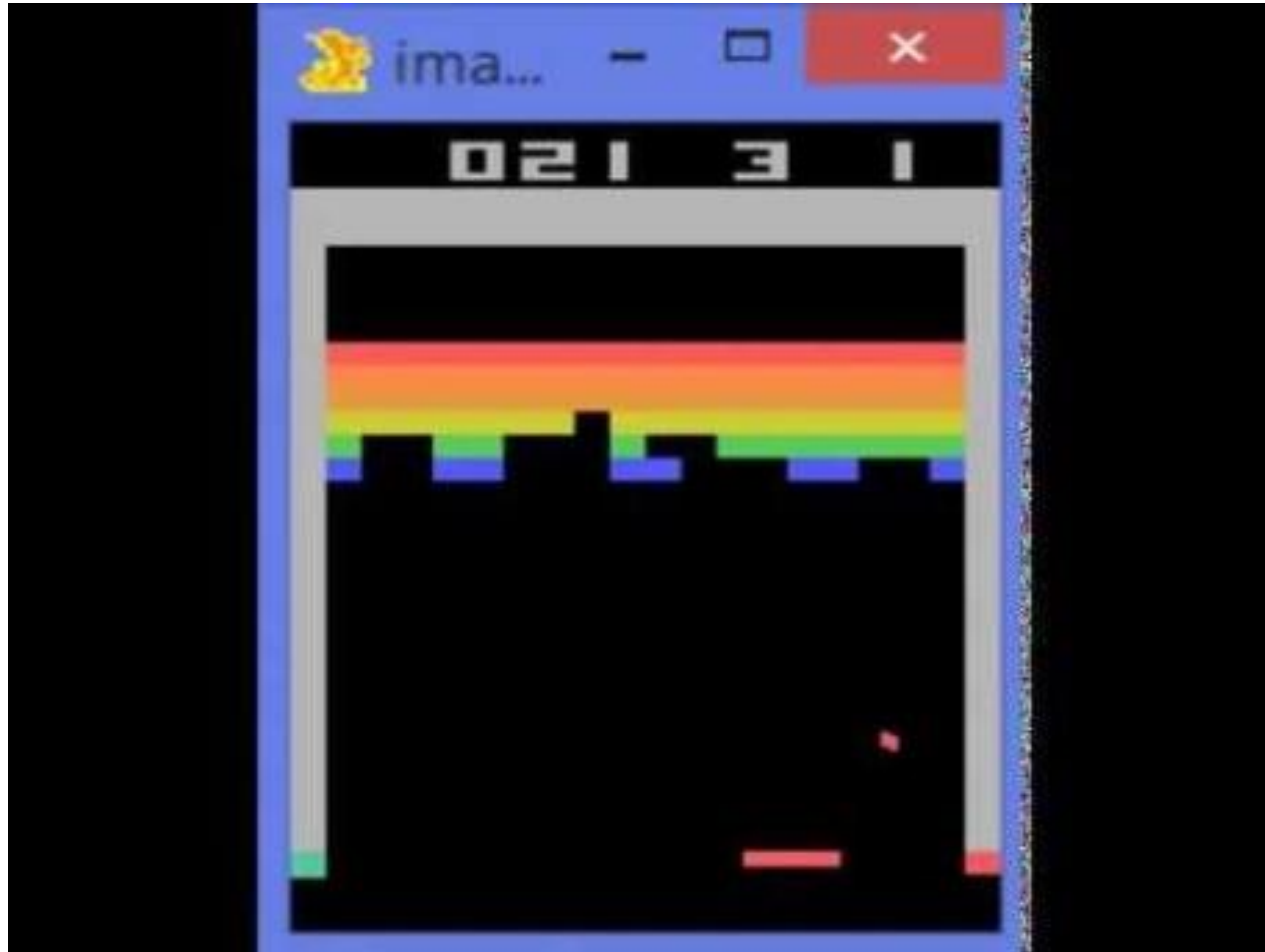
Stabilitätsanalyse

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

Atari-Ergebnisse



Atari-Video

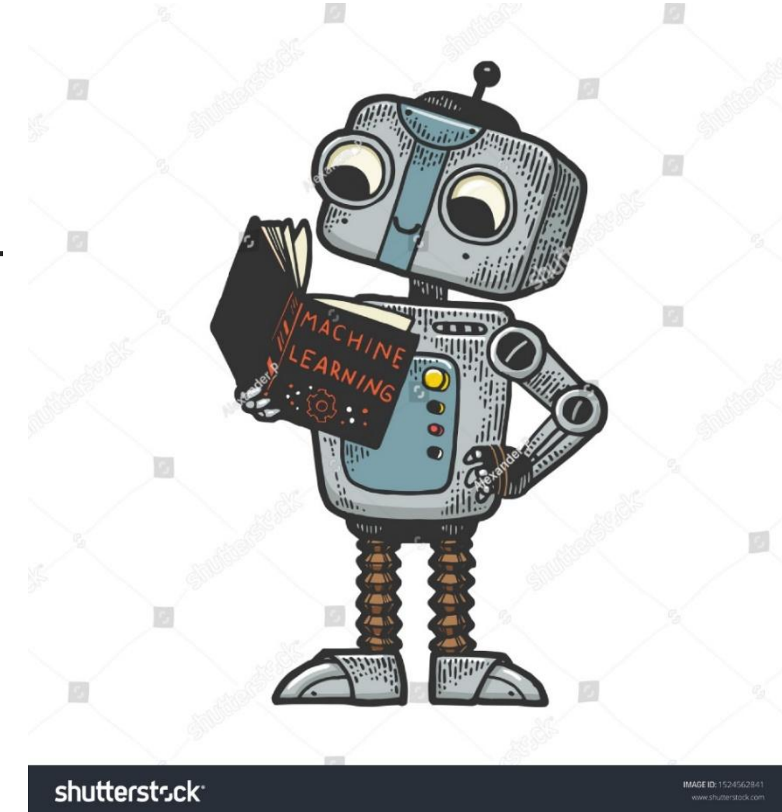


Applications of Robot Learning

Robot Learning

Ist sehr schwierig, weil...

- Hochdimensionale Beobachtungen (Kameras, Punktwolken)
- Hochdimensionale kontinuierliche Aktionen
- Daten sind teuer! Wir können nicht Millionen von Episoden laufen lassen...
- Simulatoren werden immer besser... sind aber noch immer ganz so weit
- Belohnung oft schwer zu definieren
- Exploration könnte den Roboter zerstören



Robot Learning ist viel Imitation Learning

… aber RL wird immer wichtiger!

Laufen, springen, Hinternisse…

Getting more crazy…

- Unitree G1 (HKUST…)

Boston Dynamics:

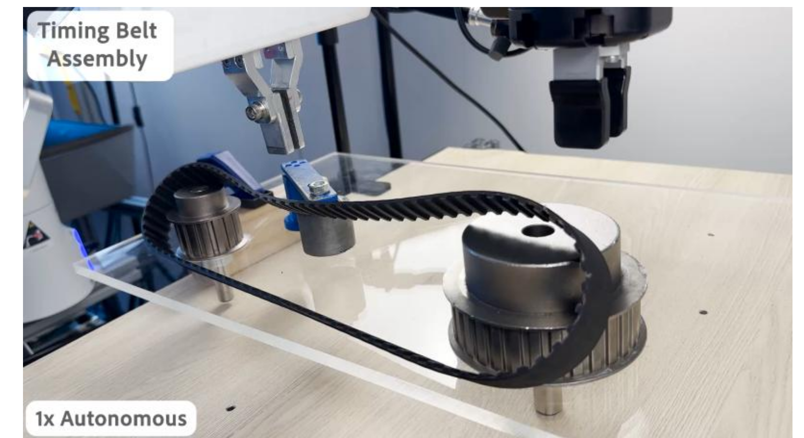
Learning Soccer

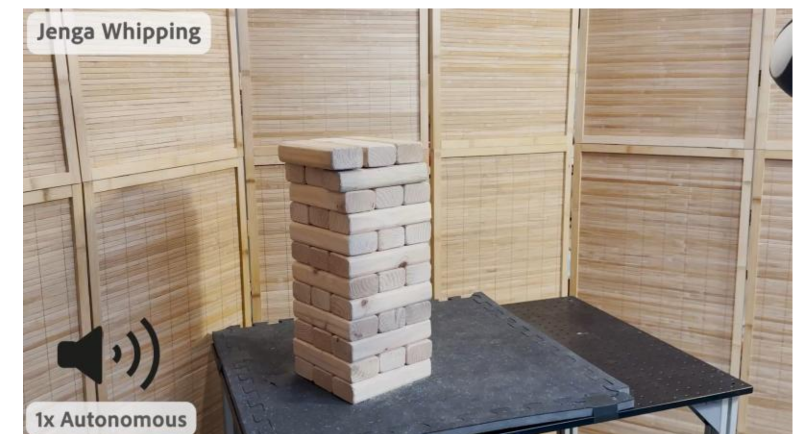
- 2 Player Soccer
- Learned in Simulation with Domain Pertubation
- Egocentric Vision

[Learning Robot Soccer from Egocentric Vision with Deep Reinforcement Learning](#)
Dhruva Tirumala, et. al. , CoRL 2024

Industrial Assembly Tasks

- Combination with human demonstrations
- Human can provide online corrections



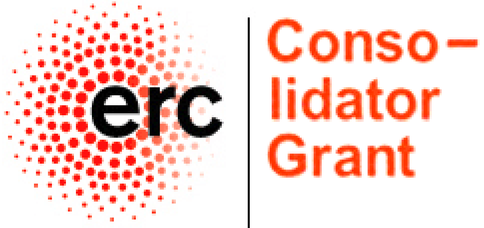


Precise and Dexterous Robotic Manipulation via Human-in-the-Loop Reinforcement Learning
Jianlan Luo, Charles Xu, Jeffrey Wu, Sergey Levine, arxiv 2024

Manipulation…

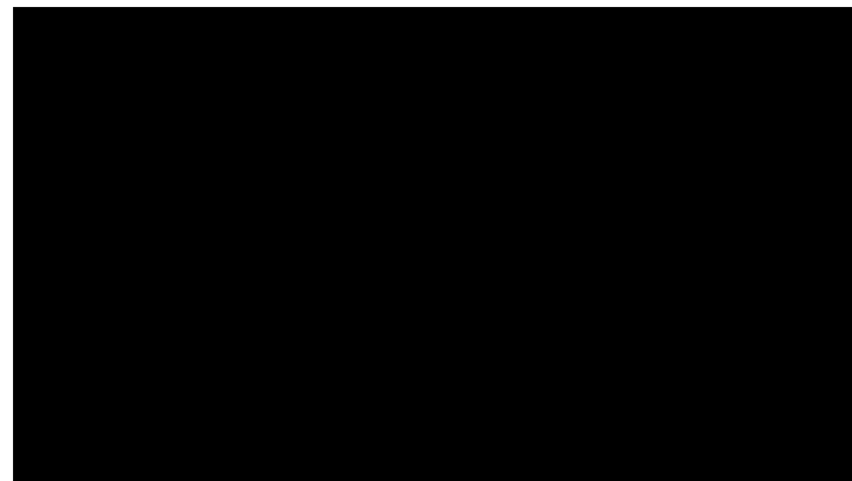
Large Scale Imitation Learning…

Unser Lab: Autonome Lernende Roboter

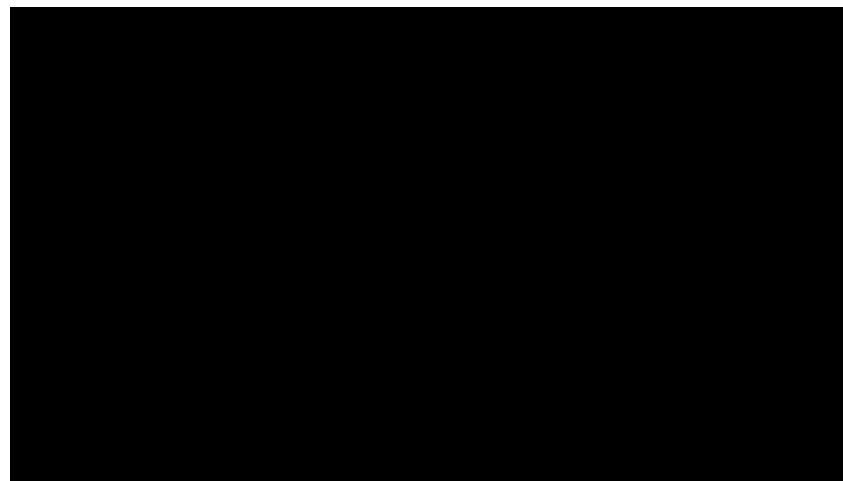


SMARTI3: Scalable Manipulation Learning through AR-enhanced Teleoperation enabling Intuitive Interactive Instructions

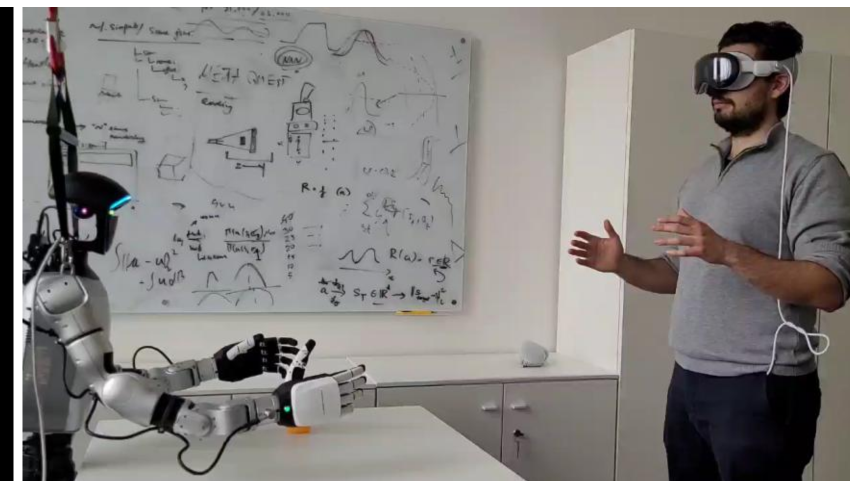
Our newest team member…



Robots playing Lego



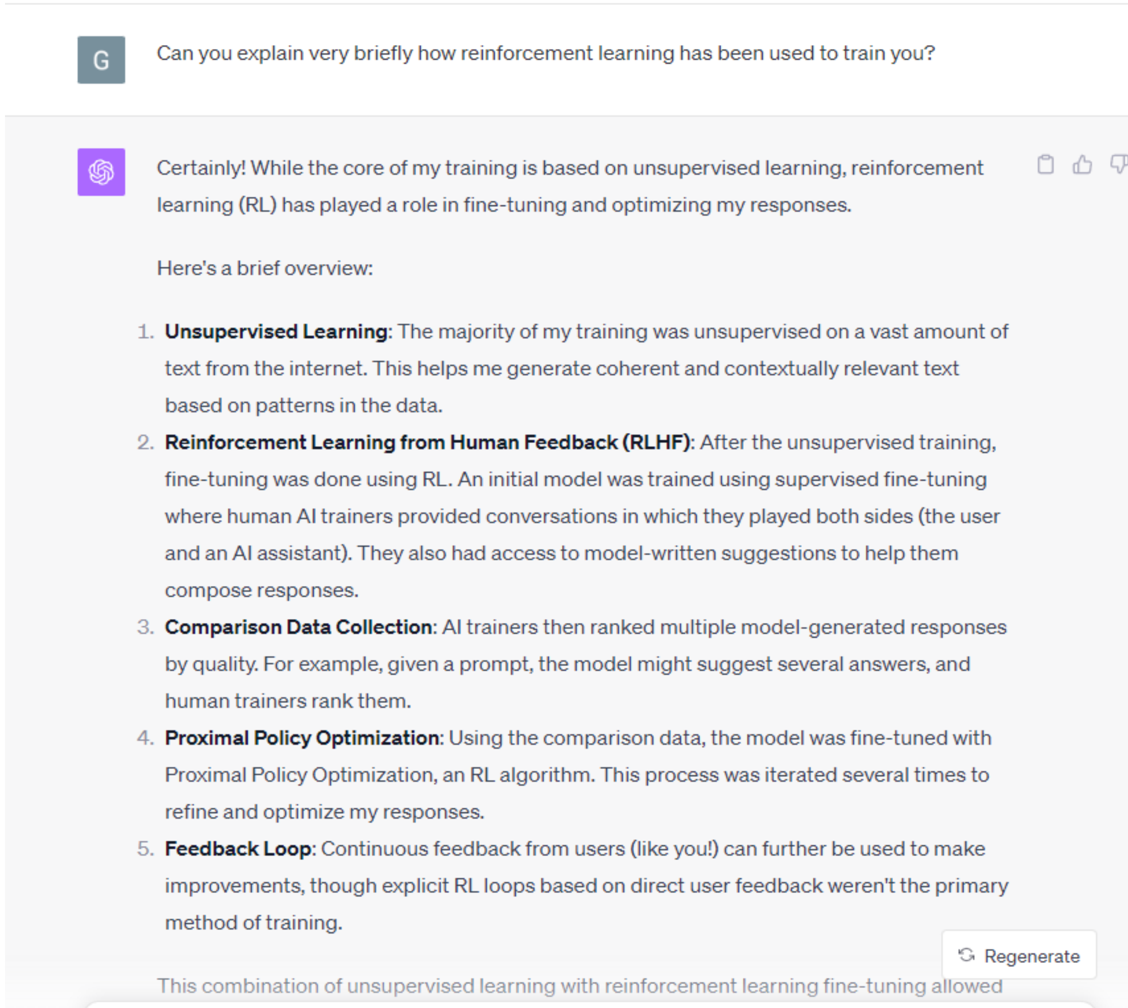
Do beautiful Origami



Interesse an Forschungsprojekten oder einer Abschlussarbeit zu diesen Themen?

- Schauen Sie sich https://alr.iar.kit.edu/81.php an.

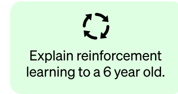
RL success stories: chatGPT



Step 1

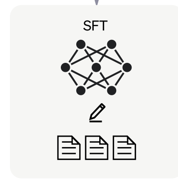
Collect demonstration data and train a supervised policy.

A prompt is sampled from our prompt dataset.



A labeler demonstrates the desired output behavior.

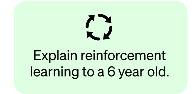
This data is used to fine-tune GPT-3.5 with supervised learning.

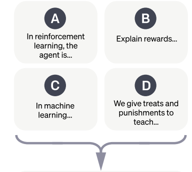


Step 2

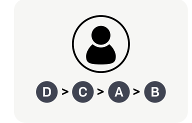
Collect comparison data and train a reward model.

A prompt and several model outputs are sampled.

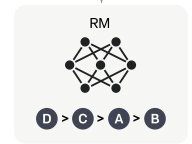




A labeler ranks the outputs from best to worst.



This data is used to train our reward model.

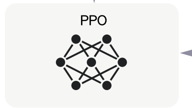


Step 3

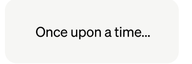
Optimize a policy against the reward model using the PPO reinforcement learning algorithm.

A new prompt is sampled from the dataset.

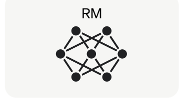
The PPO model is initialized from the supervised policy.



The policy generates an output.



The reward model calculates a reward for the output.



The reward is used to update the policy using PPO.

Also work from our lab:

[36] Becker, Freymuth, ..., Neumann: Trust Regions improve Reinforcement Learning for Large Language Models," **ICLR (2026), Oral**.

Zusammenfassung

Value-Basierende Methoden liefern in vielen Bereichen die besten RL Ergebnisse

- Kann mit nicht off-policy Daten verwendet werden (z. B. Replay Buffer)
- Eher daten effizient
- Ermöglicht das Lernen sehr komplexer, deep Q-Functions
- Wenn sie funktionieren, funktionieren sie sehr gut.

Aber:

- Niemand weiß wirklich, warum sie mit DNNs funktionieren (keine Konvergenzbeweise)
- Nach wie vor schwer zu tunen, Stabilitätsprobleme nicht vollständig beseitigt
- Fehler bei der Approximation der Q-Funktion können die Qualität der Policy einschränken
- Die meisten Erfolge in der Robotik kommen von onpolicy Algorithmen (policy gradient)

Selbsttest

Was Sie jetzt wissen sollten:

- Was wir mit Monte-Carlo-Schätzungen meinen
- Warum TD-Lernen als eine Kombination aus MC und dynamischer Programmierung angesehen werden kann
- Was ist der TD-Fehler?
- Wie Q-Learning funktioniert
- Warum brauchen wir Value Funktion Approximation und wie sieht diese aus?
- Warum ist Q-Learning kein eigentlicher Gradientenanstieg?
- Wie kann man Q-Learning für tiefe neuronale Netze verbessern?

Evaluierung

- https://onlineumfrage.kit.edu/evasys/public/online/index/index?online_php=&p=GHT58&ONLINEID=596510153992868915855621735249698450636813