Praktikum: Selbstlernende Systeme

Policy Gradient

11.12.2019

Universität Augsburg
Institut für Informatik
Lehrstuhl für Organic Computing

Gliederung

ORGANIC COMPUTING

1. Policy Gradient vs. DQN

2. Policy Search

3. Quellen

Policy Gradient vs. DQN

value-based reinforcement learning

ORGANIC COMPUTING

Um zu entscheiden welche Aktion in einem Zustand gewählt wird, orientiert man sich am **höchsten Q-Wert** (Maximum der zukünftigen Rewards in jedem Zustand)

⇓

Die policy existiert nur anhand dieser **Aktion-Wert Abschätzungen**

policy based reinforcement learning

ORGANIC COMPUTING

- kein lernen der Value Function $\rightarrow$ erwartete Summe zukünftiger Rewards gegeben State und Aktion
- direktes lernen der policy $\rightarrow$ Mappt einen Zustand auf ein Aktion
- Optimieren der Policy Function π - ohne Value Function
- Direktes Parametrisieren von π

deterministic policy

ORGANIC COMPUTING

- mappt einen Zustand direkt auf eine Aktion

$$\pi(s) = a$$

- benutzt in deterministischen Environments
  - gewählte Aktionen bestimmen das Ergebnis
  - keine Unsicherheit
  - Bsp.: Schach

stochastic policy

ORGANIC COMPUTING

- gibt eine Zufallsverteilung über alle Aktionen zurück

$$\pi(s) = P(a_t|s_t)$$

- wenn man Aktion a wählt, dann führt man diese Aktion nur mit einer gewissen Wahrscheinlichkeit aus
- benutzt in unsicherem Environment
  - Pratially Observable Markov Decision Process (POMDP)
  - Bsp.: Roboter

Vorteile

ORGANIC COMPUTING

Deep Q Learning funktioniert gut! Warum policy-based Reinforcement Learning?

Vorteile: Konvergenz

ORGANIC COMPUTING

- bessere Konvergenzeigenschaften
- Value-based:
  - können während dem Training stark oszillieren
  - eine kleine Veränderung der Vorhersage (Aktion Werte) kann zu einer vergleichsweise dramatischen Veränderung bei der Auswahl der Aktion führen.
- policy gradient:
  - folgen des Gradienten um die besten Parameter zu finden $\rightarrow$ smoothes Update der policy in jedem Schritt
  - garantierte Konvergenz in lokalem Maximum (worst case) oder gloablem Maximum (best case)

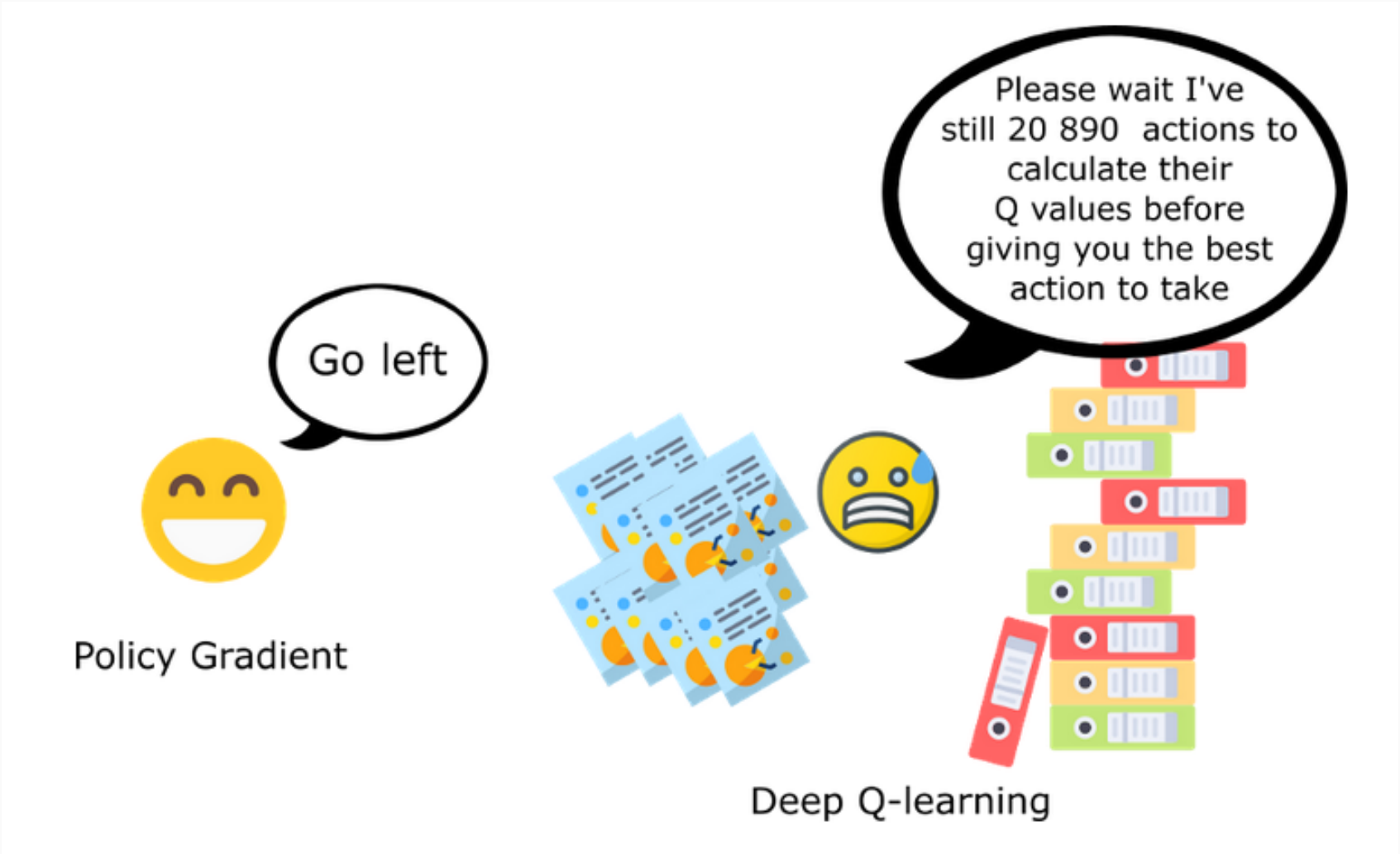
Konvergenz(2)

Vorteile: hoch-dimensionaler Aktionsraum

ORGANIC COMPUTING

- effizienter in hoch-dimensionalen Aktionsräumen oder bei kontinuierlichen Aktionen.
- Value-based:
  - für jeden Zeitschritt wird jeder möglichen Aktion ein Wert zugeordnet
  - curse of dimensionality
  - unendlich viele Aktionen? $\rightarrow$ selbstfahrendes Auto
- policy based:
  - direktes anpassen der Parameter
  - Verstehen was das Maximum ist anstatt es jeden Schritt direkt zu berechnen

hoch-dimensionaler Aktionsraum(2)



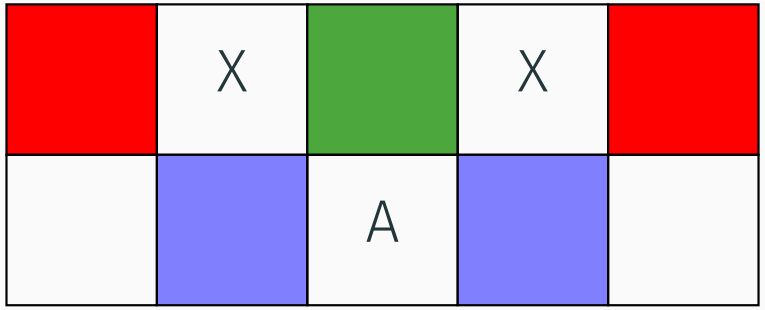
Vorteile: stochastische Policies

ORGANIC COMPUTING

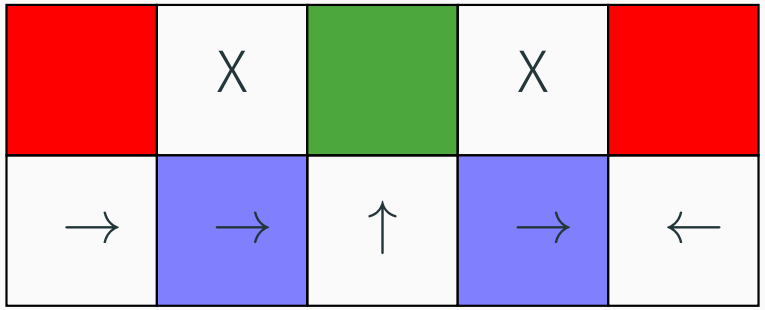
- Policy Gradient kann stochastische Policies erlernen.
- kein Exploration/Exploitation Trade-off mehr
- Output: Wahrscheinlichkeitsverteilung über die Aktionen
- $\rightarrow$ der Situationsraum wird exploriert ohne immer die gleiche Aktion zu wählen
- kein problem of perceptual aliasing: Zustände sind gleich benötigen aber unterschiedliche Aktionen

stochastische Policies(2)

blaue Felder sind aliased states



deterministische policy:



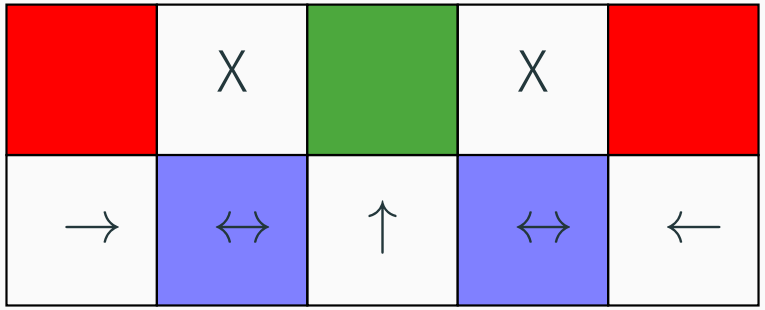
Problem: Agent bleibt stecken

stochastische Policies(3)

ORGANIC COMPUTING

stochastische policy:

die Policy wird sich zufällig nach rechts oder nach links bewegen $\rightarrow$ bleibt nicht stecken



$$\pi = (\textit{wall UP and DOWN} \mid \textit{Go LEFT}) = 0.5$$

$$\pi = (\textit{wall UP and DOWN} \mid \textit{Go RIGHT}) = 0.5$$

Nachteile

Ein großer Nachteil: Konvergiert oft zu einem lokalen Maximum anstatt zum globalen Maximum.

Konvergiert langsam Schritt für Schritt $\rightarrow$ kann länger dauern als bei DQN

Policy Search

Policy als Optimierungsproblem

ORGANIC COMPUTING

- policy π hat Parameter θ und gibt eine Wahrscheinlichkeitsverteilung der Aktionen aus:

$$\pi_\theta(a|s) = P[a|s]$$

- Wann ist policy gut? $\rightarrow$ policy als Optimierungsproblem:

$$J(\theta) = E_{\pi\theta}\left[\sum \gamma r\right]$$

- 2 Schritte:
  - messen der Qualitat von π mit einer policy score function $J(\theta)$
  - mit policy gradient ascent die besten Parameter θ finden

Schritt 1: Policy Score Function

ORGANIC COMPUTING

3 unterschiedliche Methoden - abhängig von Environment und Zielen

1. episodisches Environment:
   - Startwert $\rightarrow$ Berechnen des mittleren Rewards vom ersten Schritt G_1
   - kumulierter diskontierter Reward der gesamten Episode

$$J_1(\theta) = E_\pi[G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots] = E_\pi(V(s_1))$$

   - Wenn ich immer in s_1 starte, was ist dann der gesamte Reward den ich von diesem Zustand bis zum Endzustand bekomme?
   - Policy die G_1 maximiert = optimale Policy

Beispiel: Breakout

ORGANIC COMPUTING

- Spiel beginnt immer im gleichen Zustand
- berechnen des Scores mittels $J_1(\theta) \rightarrow$ Anzahl der getroffenen Bricks

Policy Score Function (2)

ORGANIC COMPUTING

2. kontinuierliches Environment:
   - durchschnittlicher Wert $\rightarrow$ kein fixer Startzustand
   - jeder State Value wird mit der Wahrscheinlichkeit gewichtet mit der er auftritt

$$J_{avgV}(\theta) = E_{\pi}(V(s)) = \sum d(s)V(s)$$

$$where\ d(s) = \frac{N(s)}{\sum_{s'} N(s')}$$

   - $N(s)$ - Auftreten von Zustand s
   - $\sum_{s'} N(s')$ - Auftreten aller Zustände

Policy Score Function (3)

ORGANIC COMPUTING

3. Avg Reward pro Zeitschritt:
   - größter Reward in jedem Zeitschritt

$$J_{avgR}(\theta) = E_\pi(r) = \sum_s d(s) \sum_a \pi_\theta(s,a) R_s^a$$

   - $\sum_s d(s)$ - Wahrscheinlichkeit in s zu sein
   - $\sum_a \pi_\theta(s,a)$ - Wahrscheinlichkeit Aktion a zu wählen
   - R_s^a - sofortiger Reward

Schritt 2: Policy Gradient Ascent

ORGANIC COMPUTING

- maximieren von $J(\theta) \rightarrow$ Gradient Ascent auf Policy Parametern
- Gradient Ascent $=$ Inverse von Gradient Descent (steilster Anstieg)
- kein Gradient Desent weil wir nichts minimieren wollen (Fehler Function vs. Score Function)
- Finde den Gradienten, der die aktuelle Policy in Richtung der größten Steigung updatet und iteriere:
  1. Policy: π_θ
  2. Objective function: $J(\theta)$
  3. Gradient: $\nabla_\theta J(\theta)$
  4. Update: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Policy Gradient Ascent(2)

ORGANIC COMPUTING

Ziel: finde besten Parameter θ^* der den Score maximiert:

$$\theta^* = \underset{\theta}{\text{argmax}} \underbrace{E_{\pi\theta}\left[\sum_t R(s_t, a_t)\right]}_{J(\theta)}$$

Score Function:

$$J(\theta) = \underbrace{E_\pi}_{\substack{\textit{expexted} \\ \textit{given} \\ \textit{policy}}} [R(\underbrace{\tau}_{\substack{s_0,a_0,r_1, \\ s_1,a_1,r_2,\ldots}})]$$

expected future reward

$\Rightarrow$ gesamte Summe aller erwarteten Rewards geben policy π

Policy Gradient Ascent(3)

ORGANIC COMPUTING

Differenzieren der Score Function:

$$J_1(\theta) = V_{\pi\theta}(s_1) = E_{\pi\theta}[v_1] = \underbrace{\sum_{s \in S} d(s)}_{\textit{State distribution}} \underbrace{\sum_{a \in A} \pi_\theta(s,a)}_{\textit{Action distribution}} R_s^a$$

Problem:

- Policy Parameter beeinflussen wie Aktionen gewählt werden $\rightarrow$ Reward & welchen Zustand betreten wir wie oft
- Performance abhängig von Aktionsauswahl & Verteilung von zugehörigen Zuständen $\rightarrow$ Herausforderung: Änderungen finden, die Verbesserung sicherstellen
- Einfluss der Parameter auf Zustandsverteilung? $\rightarrow$ Environment Function unbekannt
- Wie bestimmt man den Gradienten im Bezug auf π , wenn er von der Unbekannten Beziehung zwischen einer Änderung auf die Zustandsverteilung abhängt?

Policy Gradient Ascent(4)

Lösung: Policy Gradient Theorem

$\rightarrow$ liefert analytischen Ausdruck für Gradienten ∇ von $J(\theta)$ im Bezug auf π der kein Differenzierung der Zustandsverteilung enthält

$$J(\theta) = E_\pi\left[R(\tau)\right]$$

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau)$$

$$= \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau)$$

Likelihood ratio trick:

$$\pi(\tau;\theta)\frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)} \qquad \nabla \log x = \frac{\nabla x}{x}$$

$$= \sum_\tau \pi(\tau;\theta)\nabla_\theta(\log \pi(\tau;\theta)) R(\tau)$$

$$\Downarrow$$

$$\nabla_\theta J(\theta) = E_\pi\left[\nabla_\theta(\log \underbrace{\pi(\tau|\theta)}_{\textit{Policy function}}) \underbrace{R(\tau)}_{\textit{Score function}}\right]$$

Policy Gradient Ascent(5)

ORGANIC COMPUTING

- Policy Gradient: $E_\pi\left[\nabla_\theta(\log \pi(s,a,\theta))R(\tau)\right]$
- Update Regel: $\Delta\theta = \alpha \times \nabla_\theta(\log \pi(s,a,\theta))R(\tau)$
- $R(\tau)$ ist ein skalarer Wert:
  - gesamter diskontierter zukünftiger Reward
  - Wenn $R(\tau)$ groß ist $\rightarrow$ im Durchschnitt werden Aktionen gewählt die zu einem hohen Reward führen, Wahrscheinlichkeit der gesehen Aktionen $\uparrow$
  - Wenn $R(\tau)$ klein ist $\rightarrow$ Wahrscheinlichkeit der gesehen Aktionen $\downarrow$

REINFORCE: Monte Carlo Policy Gradient

ORGANIC COMPUTING

Algorithm 1: REINFORCE: Monte-Carlo Policy Gradient Control (episodic) for π

Initialize θ

for *each episode* **do**

  Generate an epsiode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following π_θ

  for *t=0 to T-1* **do**

    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

    $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \log \pi(A_t | S_t, \theta)$

REINFORCE in keras

- $G_t = r_t + \gamma * r_{t+1} + \gamma^2 * r_{t+2} + ...$
- keras minimiert seine Fehler Funktion mittels Gradient Descent
- **Trick:** negatives Vorzeichen $\rightarrow$ Gradient Ascent
- $J(\theta)$ als Fehlerfunktion:

$$loss_t = -\log \pi(A_t, S_t) * G_t$$

$$loss = \frac{\sum_t loss_t}{T}$$

Loss Funktionen in keras

ORGANIC COMPUTING

```
1  from keras.layers import Input, Dense
2  from keras.models import Model
3  from keras.optimizers import RMSProp
4  import keras.backend as K
5
6  inL = Input(self.state_size)
7  h1 = Dense(16, activation='relu')(inL)
8  out = Dense(self.action_size, activation='softmax')(h1)
9  self.model = Model(input=inL, output=out)
10 self._build_train()
11
12 def _build_train():
13   # placholder Tensor for target values
14   target = K.placeholder()
15   prediction = self.model.output
16   #compute error
17   error = K.mean(K.sqrt(1 + K.square(prediction - target)) - 1, axis=-1)
18   # create optimizer
19   opimizer = RMSProp(lr=self.learningrate)
20   update = optimizer.get_updates(loss=error, params=self.model.trainable_weights)
21   # create fit function with in- and outputs
22   self.fit = K.function(inputs=[self.model.input, target], outputs=[error], updates=
         update)
23
24 #train model
25 err = self.fit([states, targets])
```

Softmax

ORGANIC COMPUTING

- normalisierte Exponentialfunktion
- “quetscht” einen Vektor der Dimension K in einen Vektor dem Dimension K und dem Wertebereich $(0, 1)$
- Komponenten summieren sich zu 1 auf
- Wahrscheinlichkeitsverteilung über K unterschiedliche Ereignisse

Quellen

Bilder

- https://medium.freecodecamp.org/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f