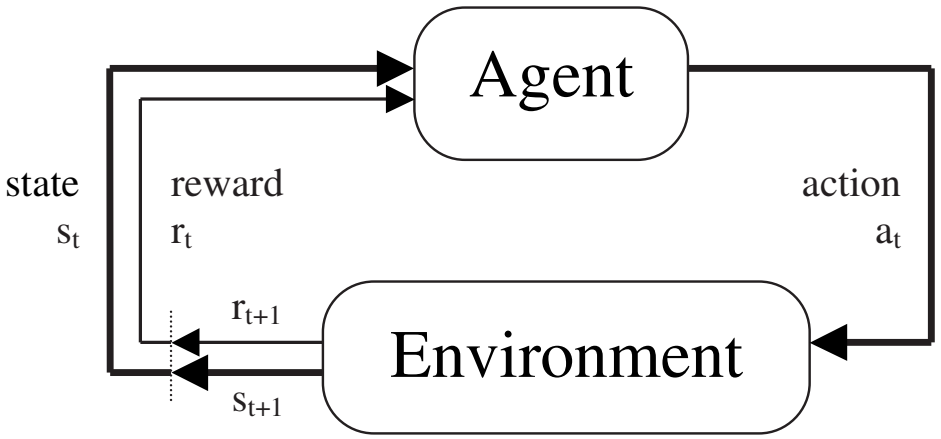
Reinforcement Learning

1. Allgemein Reinforcement Learning
2. Neuronales Netz als Bewertungsfunktion
3. Neuronales Netz als Reinforcement Learning

Allgemein Reinforcement Learning

Unterschied zu anderen Lernverfahren:



- bewertet ausgeführte Aktionen
- liefert keine richtigen Aktionen
- sondern nur eine Zahl
- läuft in Zeitschritten ab

Reinforcement Learning System besteht aus 4 Teilen:

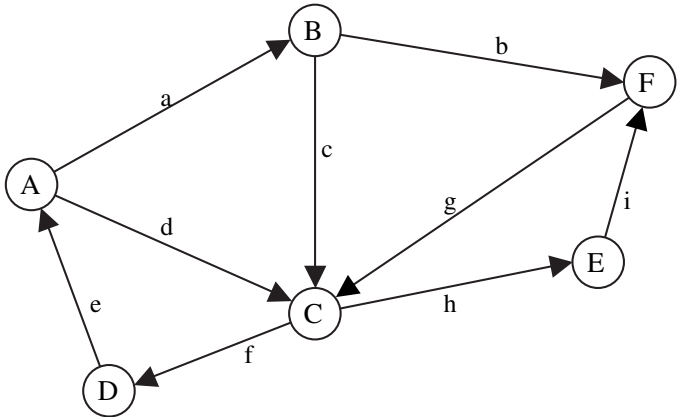
Policy: Definiert die Verhaltensweise des Agenten (vgl. stimulus-response rules) (Zustand → Aktion).

Reward function: Liefert die Belohnung die der Agent bekommt (Zustand-Aktions-Paar → Zahl).

Value function: Gibt an, wie günstig eine Aktion langfristig ist (Zustand-Aktions-Paar → Zahl).

(Model): Modell der Umgebung, von Zustands-Aktions-Paar wird neuer Zustand und Belohnung vorhergesagt.

Grundlegende Funktionsweise



$\forall a \in \mathit{Aktion}$:

$Q^*(a)$ = fester Wert (oder Verteilung)

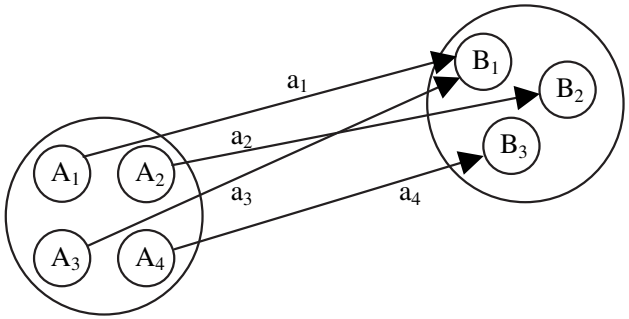
$Q_0(a)$ = initialer Wert

$Q_t(a)$ = abgeschätzer Wert zu Zeit t

1. eine Aktion (a) auswählen (nach einer Strategie)
2. Belohnung (r_t) und neuen Zustand von Umgebung bekommen
3. $Q_t(a)$ anpassen (bessere Schätzung für die Belohnung)

$\Rightarrow$ $Q_t(a) \longrightarrow Q^*(a)$ wenn $t \to \infty$

Abstraktion von Zuständen



- Nicht alle Informationen über die Umwelt stehen zur Verfügung
- Notwendige Abstraktion wegen Datenmenge

$\Rightarrow$ Kann einige Zustände nicht voneinander unterscheiden

Deshalb wird $Q^*(a)$ auch als Verteilung angegeben (als Annäherung zur Wirklichkeit)

Value function anpassen ($Q_t(a) \longrightarrow Q^*(a)$)

$Q_t(a)$ enthält den erwarteten Gewinn von Aktion a zur Zeit t.

alter Wert

$$Q_t(a) = \overbrace{Q_{t-1}(a)} + \alpha \underbrace{[r_t - Q_{t-1}]}$$

Fehlerabschätzung

Fehlerabschätzung gibt die Richtung an mit der sich der Wert ändern soll.
$\alpha (\in [0; 1])$ gibt die Geschwindigkeit an mit der sich der Wert ändern soll.

Anpassen durch das arithmetisches Mittel

Der einfachste Weg, aber umständlich zu berechnen:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a} = \frac{1}{k_a} \sum_{i=1}^{k_a} r_i$$

Daher umformen zu allgemeiner Gleichung:

$$Q_{k+1} = \frac{1}{k+1} \sum_{i=1}^{k+1} r_i = \frac{1}{k+1} [r_{k+1} + \sum_{i=1}^{k} r_i]$$

$$= \frac{1}{k+1} [r_{k+1} + kQ_k + Q_k - Q_k] = \frac{1}{k+1} [r_{k+1} + (k+1)Q_k - Q_k]$$

$$= Q_k + \frac{1}{k+1} [r_{k+1} - Q_k]$$

Einbeziehung von zukünftigen Belohnungen

Die Belohnung sollte auf lange Sicht maximiert werden

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$$

Formel nur bei in Episoden unterteilten Problemen verwendbar (wegen ∞).

$\Rightarrow$ Geringere Gewichtung für zukünftige Belohnungen

$$R_t = r_{t+1} + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad \text{(wobei } 0 \leq \gamma \leq 1\text{)}$$

$\Rightarrow$ Einführung eines Horizonts: $k \leq -\log_2 \gamma$

$$R_t = r_{t+1} + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{-\log_2 \gamma} = \sum_{k=0}^{-\log_2 \gamma} \gamma^k r_{t+k+1}$$

Exploration vs. Exploitation

Exploration

- Agent wählt eine beliebige Aktion aus.

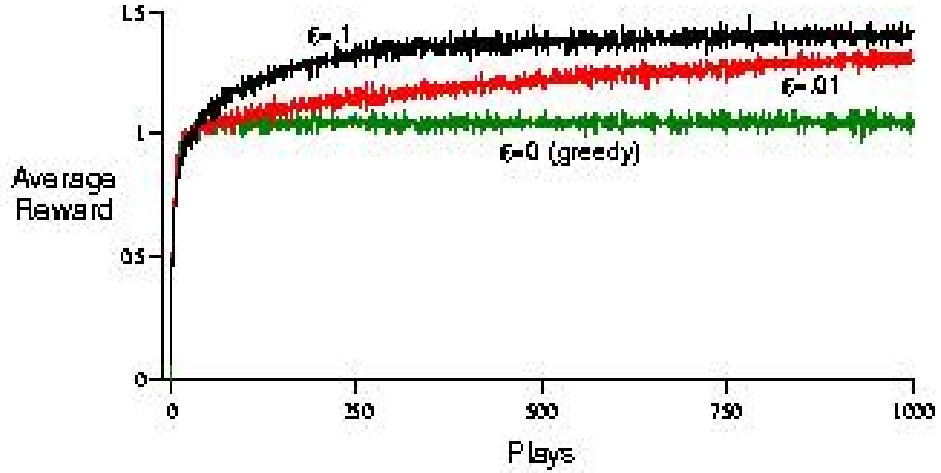
⇒ maximiert die Belohnung auf lange Sicht

Exploitation

- Agent wählt die Aktion (greedy) mit dem höchsten geschätzen Wert.

⇒ maximiert die erwartete Belohnung auf kurze Sicht

!Die beiden Möglichkeiten schließen sich aus!



Methoden um diesen Konflikt zu beseitigen

ϵ-greedy:
Mit einer niedrigen Wahrscheinlichkeit ϵ wird zufällig eine Aktion ausgesucht, sonst wird greedy Aktion genommen.

Softmax:
Aktionen werden mit der Wahrscheinlichkeit

$$\frac{e^{Q_t(a)/T}}{\sum_{b=1}^{n} e^{Q_t(b)/T}} \text{ (Boltzmann Verteilung)}$$

gewählt. Die Temperatur T wird dabei in Abhängigkeit von t immer kleiner.

Markov Property

Umgebung, deren Zustands-Signal alle relevanten Informationen (vergangene und momentane) liefert, ist Markov (oder hat die *Markov property*)

Häufig mehr als die momentan aufgetretenen Aspekte der Umgebung, aber nie mehr als alle vergangenen Aspekte.

Reaktion der Umgebung auf eine Aktion:
Formale Definition (am allgemeingültigsten):

$$P[s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0]$$

Wenn Markov Property gilt:

$$P[s_{t+1} = s', r_{t+1} = r | s_t, a_t]$$

Markov Decision Process

Reinforcement Learning Problem, das die *Markov property* erfüllt heißt *MDP*.
(Mit endlichem Zustands- und Aktionsraum: *finit MDP*)

Wichtigsten Aspekte eines finit MDP werden beschrieben durch:

1. Wahrscheinlichkeit für Folgezustand:

$$\mathcal{P}^{\pi}_{ss'} = P[s_{t+1} = s' | s_t = s, a_t = a]$$

2. Belohnung für Folgezustand:

$$\mathcal{R}^{\pi}_{ss'} = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$$

Nur Verteilung der erwarteten Belohnung geht verloren

Value Function bei finit MDP

state-value function for policy π:

$$V^\pi(s) = E_\pi[R_t|s_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s]$$

action-value function for policy π:

$$Q^\pi(s,a) = E_\pi[R_t|s_t = s, a_t = a] = E_\pi[\sum_{t=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a]$$

Diese Funktionen V^π und Q^π können aus Erfahrung abgeschätzt oder berechnet werden.

Berechnung der Value Function bei finit MDP

$$V^{\pi}(s) = E_{\pi}[R_t|s_t = s]$$

$$= E_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s]$$

$$= E_{\pi}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s]$$

$$= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^{\pi}_{ss'}[\mathcal{R}^{\pi}_{ss'} + \gamma E_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s']]$$

$$= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^{\pi}_{ss'}[\mathcal{R}^{\pi}_{ss'} + \gamma V^{\pi}(s')]$$

Neuronales Netz als Bewertungsfunktion

Notwendigkeit bei großen Problemen:

- Die meisten Zustände werden nicht besucht
- Nicht alle Zustände (mit Werten) können gespeichert werden

$\Rightarrow$ Der einzige Weg etwas zu lernen ist die Generalisierung (von bekannten auf unbekannte Zustände schließen)

Die Art der Generalisierung, die man dafür benötigt, ist die *function approximation* aus dem Bereich des *supervised learning*.

Annäherung der Funktion - Allgemein

V^π soll durch V_t (bestimmt durch $\overrightarrow{\theta_t}$) angenähert werden.

$\overrightarrow{\theta_t}$: Kann alles sein (z.B. Verbindungsgewichte des Netzes)

Während der Interaktion werden so genannte „backups“ in der Form

$$s \rightarrow v \text{ (abgeschätzer Wert soll mehr wie } v \text{ sein)}$$

gewonnen.

function approximation:

Input: “backups“

Output: Funktion mit gewünschtem Ein-/Ausgabe-Verhalten

Performance Measure

Aber wie optimiert man das Ergebnis bei der Approximation?

Man bildet den mittleren quadratischen Fehler (Mean-squared error)

$$MSE(\overrightarrow{\theta_t}) = \sum_{s \in S} P(s)[V^{\pi}(s) - V_t(s)]^2$$

($P(s)$: Verteilung gewichtet die Fehler von Zuständen)

und versucht diesen global zu optimieren

$$MSE(\overrightarrow{\theta^*}) < MSE(\overrightarrow{\theta})$$

wobei $\overrightarrow{\theta^*}$ der optimale Parameter für die Funktion ist.

Methoden für die Approximation

Diese Methoden aus dem Bereich *supervised lerning from examples* können verwendet werden:

- Entscheidungsbäume
- Multivariate Regression
- Neuronales Netz
- ...

Die Aufzählung soll deutlich machen, daß die Generalisierung durch viele verschiedene Methoden erreicht werden kann.

Probleme der Annäherung bei Neuronale Netze

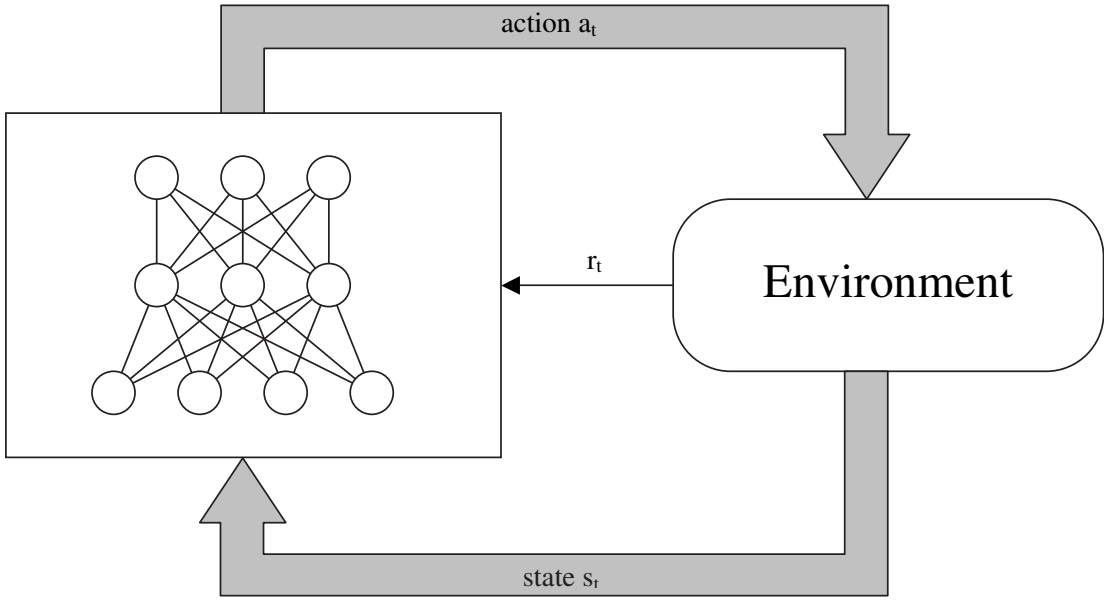
Annahme der Netze:

- Feste Trainingsbeispiele, die mehrmals gelernt werden.

Tatsächlich benötigt:

- Online Lernen (während der Interaktion mit der Umgebung)
- Funktion, die effizient von zunehmenden Daten lernt

Neuronales Netz als Reinforcement Learning



Vereinfachung von RF:

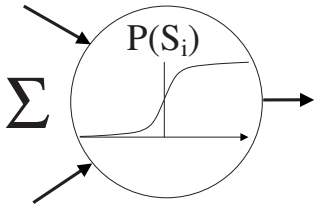
- Signal nur 0 und 1
- Nur einfache Problemklasse

Langsamer als *supervised learning*

Grundlegende Möglichkeit

Neuronale Netze sind deterministisch und liefern daher keine zufälligen Ausgaben, Reinforcement Learning ist aber auf Exploration angewiesen (kein *teacher*).

Stochastic Units:



$$P(S_i = \pm 1) = g(\pm h_i) = \frac{1}{1 + e^{\mp 2h_i/T}} \qquad (\text{wobei } h_i = \sum_j w_{ij} V_j)$$

Die Ausgangschicht des Netzes besteht aus stochastic units

Lernen des RF-Netzes

Die nötigen Pattern werden (mit Hilfe von r) selber erstellt:

$$\xi_i^\mu = \begin{cases} S_i^\mu & falls\ r^\mu = +1 \\ -S_i^\mu & falls\ r^\mu = 0 \end{cases}$$

Daraus wird der Fehler berechnet:

$$\delta_i^\mu = \xi_i^\mu - \langle S_i^\mu \rangle \qquad \text{(Pattern - Erwartungswert)}$$

Damit können die Gewichte der Ausgangsneuronen geändert werden:

$$\Delta w_{ij} = \eta(r^\mu)\delta_i^\mu V_i^\mu$$

Restliches Netz wird per Backpropagation von δ_i^μ trainiert

Ausblick für diese Möglichkeit

- Lernen konvergiert, dann werden die Stochastic Units fast deterministisch

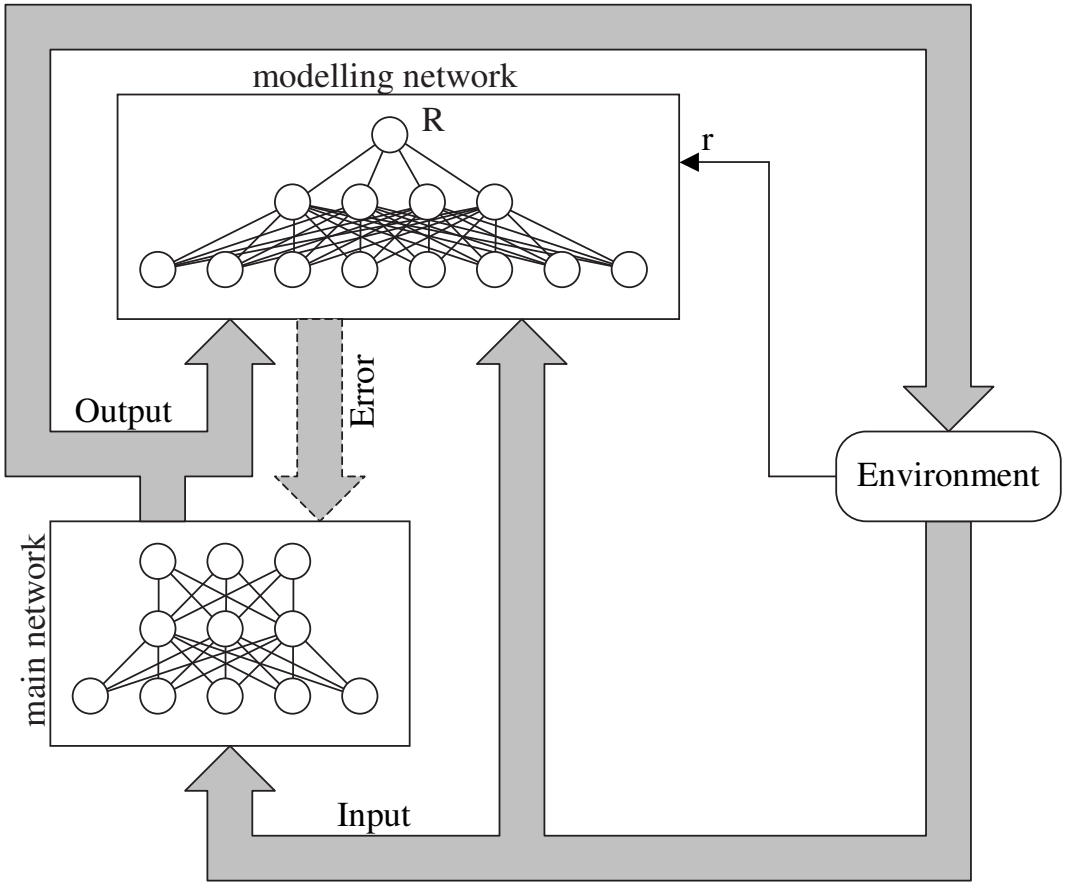
$$\langle S_i^\mu \rangle \to \pm 1 \quad \text{(Erwartungswert geht gegen } \pm 1\text{)}$$

- Variation für kontinuierliche Reinforcement Signale($0 \leq r \leq 1$):

$$\Delta w_{ij} = \eta(r^\mu)(r^\mu[S_i^\mu - \langle S_i^\mu \rangle] + (1 - r^\mu)[-S_i^\mu - \langle S_i^\mu \rangle])V_j^\mu$$

- Keinen allgemein gültigen Konvergenzbeweis
  Nur für ein paar Spezialfälle (z.B. nur ein Ausgangsneuron)

Erweiterung durch ein Modell



main network:

- Generiert die Ausgabe (gleiches Netz wie vorher)

modelling network:

- Bekommt die gleichen Informationen wie Umgebung
- Soll r duplizieren

$\Rightarrow$ Modell ist gut, wenn $R \approx r$

Vorgehensweise

1. Modelling network trainieren ($R \rightarrow r$)
2. Modelling network verwenden um main network zu trainieren

   Ziel: R maximieren

   Lernen: Backpropagation durch beide Netze (Gewichte nur im main ändern)

$\Rightarrow$ Reinforcement Problem zu 2 *supervised learning*-Teilen mit bekannten Zielen umgewandelt.