Die Q-Funktion

Man definiert die Q-Funktion wie folgt:

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

Damit läßt sich π^* schreiben als

$$\pi^*(s) = arg\max_a Q(s,a)$$

D.h.: Die optimale Policy kann erlernt werden, indem Q gelernt wird, auch wenn r und δ nicht bekannt sind.

Q-Lernalgorithmus - I

Q und V^* stehen in enger Beziehung zueinander:

$$V^*(s) = \max_{a'} Q(s, a')$$

Damit läßt sich die Definition von $Q(s, a)$ umschreiben:

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

Diese rekursive Definition von Q bildet die Basis für einen Algorithmus, der Q iterativ approximiert.

Q-Lernalgorithmus - II

Im folgenden sei $\hat{Q}$ der aktuelle Schätzwert für Q. s' sei der neue Zustand nach Ausführung der gewählten Handlung und r sei der dabei erzielte Reward.

Dann ergibt sich aus der rekursiven Definition von Q die Iterationsvorschrift wie folgt:

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

$\Rightarrow$:

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

Q-Lernalgorithmus - III

Damit lautet der Algorithmus:

1. Initialisierte alle Tabelleneinträge von $\hat{Q}$ zu 0.

2. Ermittle aktuellen Zustand s.

3. Loop

   - Wähle Handlung a und führe sie aus.
   - Erhalte Reward r.
   - Ermittle neuen Zustand s'.
   - $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$
   - $s \leftarrow s'$.

   Endloop

Konvergenz des Q-Lernens

Satz: Es seien folgende Bedingungen erfüllt:

- $|r(s,a)| < \infty, \forall s, a$
- $0 \leq \gamma < 1$
- jedes (s,a)-Paar wird unendlich oft besucht

Dann konvergiert $\hat{Q}$ gegen die richtige Q-Funktion.

Kontinuierliche Systeme

Die Q-Funktion sehr großer oder kontinuierlicher Zustandsräume läßt sich nicht durch eine explizite Tabelle darstellen.

Man verwendet stattdessen einen Funktionsapproximations-Algorithmus, z.B. ein Neuronales Netz oder B-Splines.

Die Ausgaben des Q-Learning-Algorithmus dienen dem Neuronalen Netz als Trainingsbeispiele.

Konvergenz ist dann aber nicht mehr garantiert!

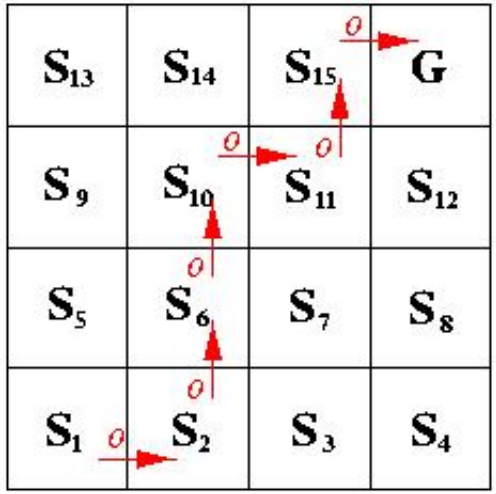
Beispiel: GridWorld - I

gegeben: $m \times n$-Grid

- $S = \{(x, y) | x \in \{1, \cdots, m\}, y \in \{1, \cdots, n\}\}$
- $A = \{up, down, left, right\}$
- $r(s, a) = \begin{cases} 100, \text{falls } \delta(s, a) = \text{Goalstate} \\ 0, sonst. \end{cases}$
- $\delta(s, a)$ gibt den Folgezustand entsprechend der durch a gegebenen Bewegungsrichtung an.

Beispiel: GridWorld - II

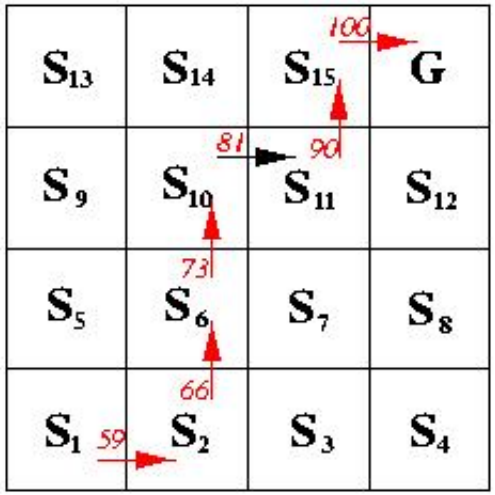
Beispiel für einen Pfad durch einen Zustandsraum:



Die Zahlen an den Pfeilen geben die momentanen Werte von $\hat{Q}$ an.

Beispiel: GridWorld - III

Entwicklung der $\hat{Q}$-Werte:



$$\hat{Q}(S_{11}, up) = \quad r + \gamma \max_{a'} \hat{Q}(s', a') = 0 + 0.9 * 100 = 91$$

$$\hat{Q}(S_{10}, right) = \quad 0 + 0.9 * 91 = 81$$

.
.
.

Ad-hoc Explorationsstrategie

- Zu ausgiebiges Erforschen bedeutet, daß der Agent auch nach langem Lernen noch ziellos im meist sehr großen Zustandsraum umherwandert. Dadurch werden auch Bereiche intensiv untersucht, die für die Lösung der Aufgabe gar nicht relevant sind.
- Zu frühes Ausbeuten der gelernten Approximation der Q-Funktion bewirkt möglicherweise, daß sich ein suboptimaler, d.h. längerer Pfad durch den Zustandsraum, der zufällig zuerst gefunden wurde, etabliert und die optimale Lösung nicht mehr gefunden wird.

Es existieren:

- "Greedy strategies"
- "randomized strategies"
- "interval-based techniques"

Beschleunigung des Lernens

Ad-hoc Techniques for

- Experience Replay
- Backstep Punishment
- Reward Distance Reduction
- Lerner-Kaskade

Experience Replay - I

Ein Pfad durch den Zustandsraum gilt als beendet, sobald G erreicht wurde.

Nun sei angenommen, im Zuge des Q-Learning werde dieser Pfad wiederholt durchlaufen.

Oftmals sind echt neue Lernschritte sehr viel kostenintensiver und/oder zeitaufwendiger als interne Wiederholungen bereits gespeicherter Lernschritte. Aus den genannten Gründen bietet es sich an, die Lernschritte abzuspeichern und intern zu wiederholen. Das Verfahren wird **Experience Replay** genannt.

Eine **Erfahrung** e (engl.: *experience*) ist ein Tupel

$$e = (s, s', a, r)$$

mit $s, s' \in S$, $a \in A$, $r \in \mathbb{R}$. e repräsentiert einen Lernschritt, d.h. einen Zustandsübergang, wobei s der Ausgangszustand, s' der Zielzustand, a die Aktion, die zu dem Zustandsübergang führte, und r das dabei erhaltene Reinforcement-Signal ist.

Experience Replay - II

Ein **Lernpfad** ist dann eine Serie $e_1 \ldots e_{L_k}$ von Erfahrungen (L_k ist die Länge des k-ten Lernpfades).

Der ER-Algorithmus:

```
for k = 1 to N
    for i = L_k to 1
        update(e_i aus Serie k)
    end for
end for
```

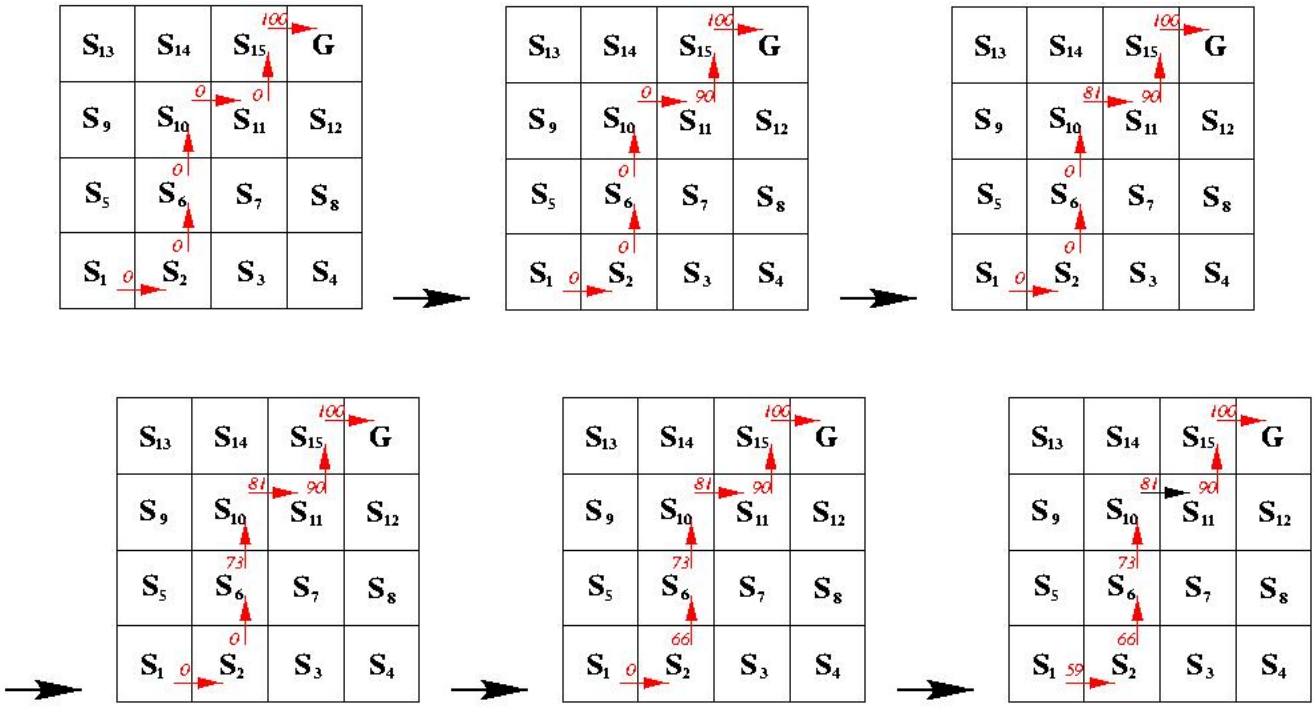
Experience Replay - III

Vorteile:

- Interne Wiederholungen gespeicherter Lernschritte verursachen meist weit weniger Kosten als echt neue Lernschritte.
- Intern läßt sich ein Lernpfad in umgekehrter Reihenfolge durchlaufen, was die Informationsausbreitung beschleunigt.
- Wenn sich Lernpfade kreuzen, können sie sozusagen „voneinander lernen“, d.h. Information austauschen. ER macht diesen Austausch unabhängig von der Reihenfolge, in der die Lernpfade erstmals ausgeführt wurden.

Experience Replay - Beispiel

Entwicklung der $\hat{Q}$-Werte bei wiederholtem Durchlaufen eines Lernpfades:



Backstep Punishment

Eine Explorationsstrategie wird benötigt, die für eine etwas „geradlinigere“ Bewegung des Agenten durch den Zustandsraum sorgt.

Dazu müssen vor allem Rückschritte vermieden werden.

Eine sinnvollere Methode scheint es zu sein, für den Fall, daß der Agent einen Rückschritt auswählt, ihn diesen ausführen zu lassen, aber ein *künstliches, negatives Reinforcement-Signal* zu generieren.

Kompromiß zwischen “Sackgasse-Vermeidung” und “schnellem Lernen”.

Beim zielorientierten Lernen könnte eine entsprechend erweiterte Reward-Funktion wie folgt aussehen:

$$r_{\mathsf{BP}} = \begin{cases} 100 & \text{falls Übergang in einen Zielzustand erfolgte} \\ -1 & \text{falls ein Rückschritt gemacht wurde} \\ 0 & \text{sonst} \end{cases}$$

Reward Distance Reduction

Die Reward-Funktion kann eine intelligentere Bewertung der Aktionen des Agenten vornehmen. Dies setzt aber Kenntnisse über die Struktur des Zustandsraumes voraus.

Wenn man zusätzlich die Kodierung des Zielzustandes kennt, dann kann es eine gute Strategie sein, die euklidische Distanz zwischen dem aktuellen Zustand und dem Zielzustand zu verringern.

Man kann die Rewardfunktion so erweitern, daß Aktionen, die die euklidische Distanz zum Zielzustand verringern, belohnt werden (**reward distance reduction**, **RDR**):

$$r_{\text{RDR}} = \begin{cases} 100 & \text{falls } \vec{s}' = \vec{s}_g \\ 50 & \text{falls } |\vec{s}' - \vec{s}_g| < |\vec{s} - \vec{s}_g| \\ 0 & \text{sonst} \end{cases}$$

wobei $\vec{s}$, $\vec{s}'$ und $\vec{s}_g$ die den Ausgangszustand, den Endzustand und den Zielzustand kodierenden Vektoren sind.

Lerner-Kaskade - I

Es ist für die Genauigkeit der Positionierung entscheidend, wie fein der Zustandsraum eingeteilt ist.

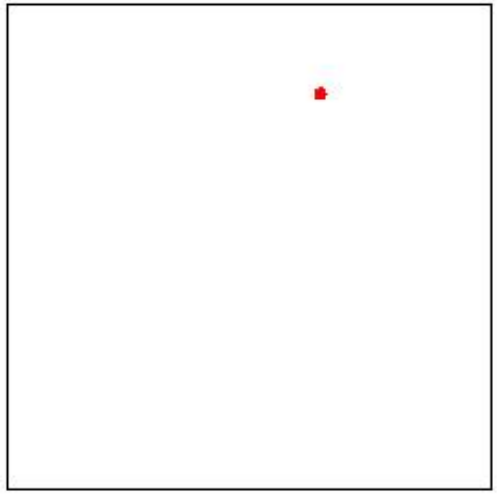
Andererseits steigt mit wachsender Feinheit der Diskretisierung auch die Anzahl der Zustände und damit der Aufwand für das Lernen.

Man muß sich vor dem Lernen für eine Diskretisierung entscheiden und dabei abwägen zwischen Lernaufwand und Genauigkeit der Positionierung.

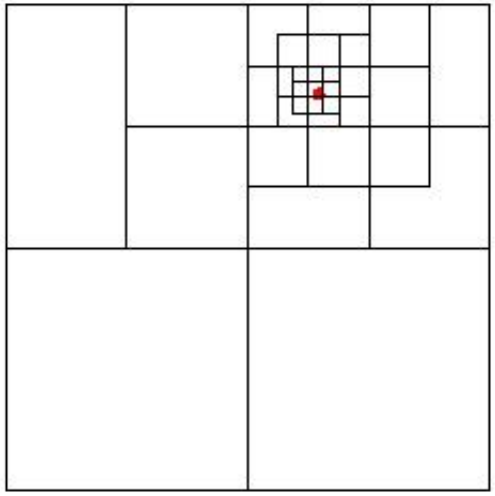
Lerner-Kaskade - Variable Diskretisierung

Ein Beispiel-Zustandsraum a) ohne Diskretisierung b) mit variabler Diskretisierung

a)



b)

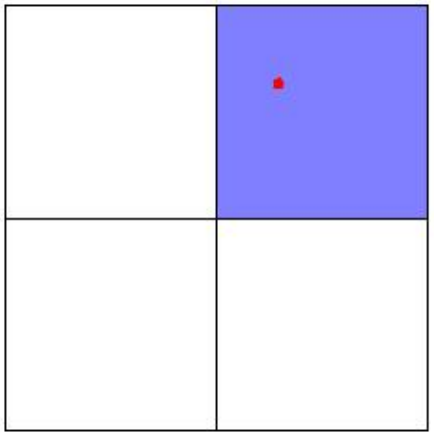


Dies setzt aber Kenntnisse über die Struktur des Zustandsraumes voraus.

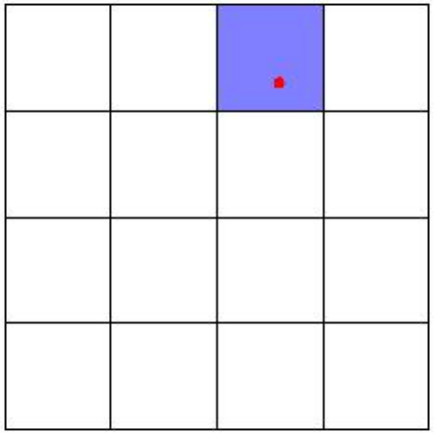
Lerner-Kaskade - n-stufige Lerner-Kaskade

Einteilungen des Zustandsraumes einer vierstufigen Lerner-Kaskade:

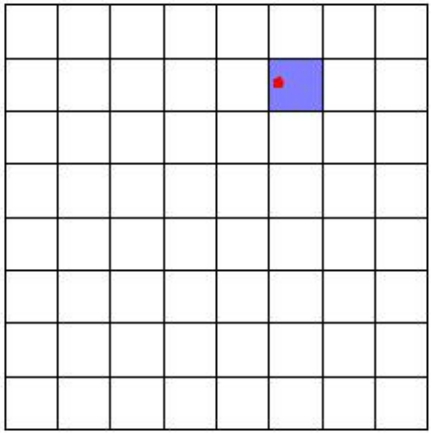
1. Lerner:



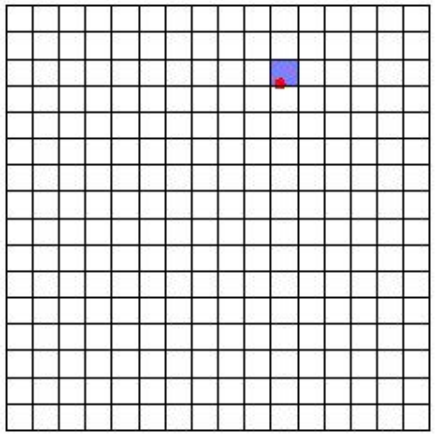
2. Lerner:



3. Lerner:



4. Lerner:



Q-Lernen - offene Fragen

- oft nicht vorhanden: Markov-Annahme, Beobachbarkeit
- kontinuierliche Zustand-Aktion-Räume
- Generalisierung über Zustand und Aktion
- Kompromiß zwischen “Exploration” und “Exploitation”
- Generalisierung der automatischen Bewertung (Kredit-Vergabe)

Dynamische Programmierung für Modellbasiertes Lernen

Wenn die Dynamik der Umwelt bekannt ist, kann die Klasse der Dynamischen Programmierung verwendet werden.

Bei der Dynamischen Programmierung handelt es sich um eine Sammlung von Ansätzen, bei der ein perfektes Modell des MDP's vorhanden ist. Um die optimale Policy zu berechen, werden die Bellmann Gleichungen in Update Regeln eingebettet, die dann die gewünschte Werte-Funktion (V) approximieren.

Drei Schritte:

1. Policy Evaluation

2. Policy Improvement

3. Policy Iteration

Policy Evaluation - I

Unter $Policy\ Evaluation$ versteht man das Problem, aus einer willkürlichen festen Policy π die Werte-Funktion V^π zu berechnen. Ausgehend von der Bellmann-Gleichung kann man eine Update-Regel entwerfen, in der approximierte Werte-Funktionen $V_0, V_1, V_2, V_3, \ldots$ berechnet werden:

$$V_{k+1}(s) \quad = \quad \sum_a \pi(s,a) \sum_{s'} [R^a_{ss'} + \gamma V_k(s')] \qquad \forall s \in S$$

Es wird ausgehend von der k-ten Näherung V_k für jeden Zustand s sukzessiv die k+1-te Näherung V_{k+1} berechnet, mit anderen Worten wird der alte Funktionswert von s durch den neuen Wert ersetzt, der mit den alten Werten gemäß der Iterationsregel berechnet wird.

Es kann gezeigt werden, daß die Sequenz der iterierten Werte-Funktionen $\{V_k\}$ gegen V^π konvergiert, falls $k \to \infty$.

Policy Evaluation - II

Die Initialwerte-Funktion V_0 kann dabei willkürlich gewählt werden, lediglich die Terminal-Zustände, falls eine episodische Aufgabe vorliegt, müssen den Wert 0 erhalten.

Gegeben sei die zu evaluierende Policy π
Initialisiere $V(s) := 0 \quad \forall s \in S^+$
Wiederhole
    $\delta := 0$
    Für jedes $s \in S$ tue
        $V_{k+1}(s) := \sum_a \pi(s,a) \sum_{s'} [R^a_{ss'} + \gamma V_k(s')]$
        $\delta := \max(\delta, |V_{k+1}(s) - V_k(s)|)$
bis $\delta < \Theta$
Man erhält $V \approx V^\pi$

Policy Improvement

Man betrachtet also die Aktions-Werte-Funktion $Q^\pi(s, a')$, bei der in Zustand s Aktion a' ausgewählt wird, und dann die Policy π verfolgt wird:

$$Q^\pi(s, a') = \sum_{s'} [R_{ss'}^{a'} + \gamma V^\pi(s')]$$

Man sucht in jedem Zustand die Aktion, die die Aktions-Werte-Funktion maximiert. Somit wird eine greedy Policy π' für eine gegebene Werte-Funktion V^π generiert:

$$\begin{aligned} \pi'(s, a') &= 1 \quad \wedge \\ a' &= \arg\max_a Q^\pi(s, a) \\ &= \arg\max_a \sum_{s'} [R_{ss'}^{a} + \gamma V^\pi(s')] \end{aligned}$$

Policy Iteration - I

Verwendet man abwechselnd Policy Improvement und Policy Evaluation, so heißt das, dass mit einer festen Werte-Funktion V^π die Policy π verbessert wird und dann für die bessere Policy π' die dazugehörige Werte-Funktion $V^{\pi'}$ berechnet wird.

Man wendet wieder Policy Improvement an, um eine noch bessere Policy π'' zu bekommen, und so weiter . . . :

$$\pi_0 \xrightarrow{\text{PE}} V^{\pi_0} \xrightarrow{\text{PI}} \pi_1 \xrightarrow{\text{PE}} V^{\pi_1} \xrightarrow{\text{PI}} \pi_2 \xrightarrow{\text{PE}} V^{\pi_2} \cdots \xrightarrow{\text{PI}} \pi^* \xrightarrow{\text{PE}} V^*$$

Hierbei steht $\xrightarrow{\text{PI}}$ für das Anwenden des Policy Improvement und analog $\xrightarrow{\text{PE}}$ für Policy Evaluation.

Policy Iteration - II

Der Algorithmus für die Policy Iteration:

Initialisiere zufällig $V(s) \in \mathcal{R}$ und $\pi(s) \in A(s)\ \forall s \in S$

L (Policy Evaluation)

Wiederhole

$\delta := 0$

Für jedes $s \in S$ tue

$V_{k+1}(s) := \sum_{s'}[R_{ss'}^{\pi(s)} + \gamma V_k(s')]$

$\delta := \max(\delta, |V_{k+1}(s) - V_k(s)|)$

bis $\delta < \Theta$ (mit $\Theta > 0$ und klein)

(Policy Improvement)
POLICY_STABIL := TRUE
Für jedes $s \in S$ tue
    $b := \pi(s)$
    $\pi(s) := \arg\max_a \sum_{s'} [R^a_{ss'} + \gamma V(s')]$
    wenn $b \neq \pi(s)$ dann POLICY_STABIL := FALSE
wenn POLICY_STABIL = TRUE, dann stoppe,
sonst weiter bei Marke L

Bahnplanung mit der Policy Iteration - I

Zum finden des Pfades zwischen Start- und Zielkonfiguration wird die Policy Iteration benutzt.

Folgende Größen müßen dementsprechend beim vorliegenden Planungsproblem definiert werden:

- Als Zustandsmenge S dient der diskrete Konfigurationsraum, dh. jedes mögliche Gelenkwinkeltupel $(\theta_1, \theta_2, \ldots, \theta_{Dim})$ ist genau ein Zustand der Menge S mit Ausnahme der Zielkonfiguration

- Die Menge $A(s)$ aller möglichen Aktionen für einen Zustand s sind die Übergänge zu Nachbarkonfigurationen im K-Raum, also für einen oder mehrere Gelenkwinkel θ_i die Änderung um $\pm Dis$. Dabei sind nur Aktionen a in $A(s)$ enthalten, die nicht zu K-Hindernissen führen, dh. die nicht mit Hindernissen kollidieren, und nicht die Grenzen des K-Raumes verletzen

Bahnplanung mit der Policy Iteration - II

- Für die Rewardfunktion $R^a_{ss'}$ gilt:

  $R^a_{ss'} =$ **Reward_nicht_im_Ziel** $\forall s \in S, a \in A(s), s' \in S$ und
  $R^a_{ss'} =$ **Reward_im_Ziel** $\forall s \in S, a \in A(s), s' = s_t$.

  Also nur für das Erreichen des Zielzustandes wird ein anderer Rewardwert geliefert. Für alle anderen Zustände gibt es einen einheitlichen Betrag

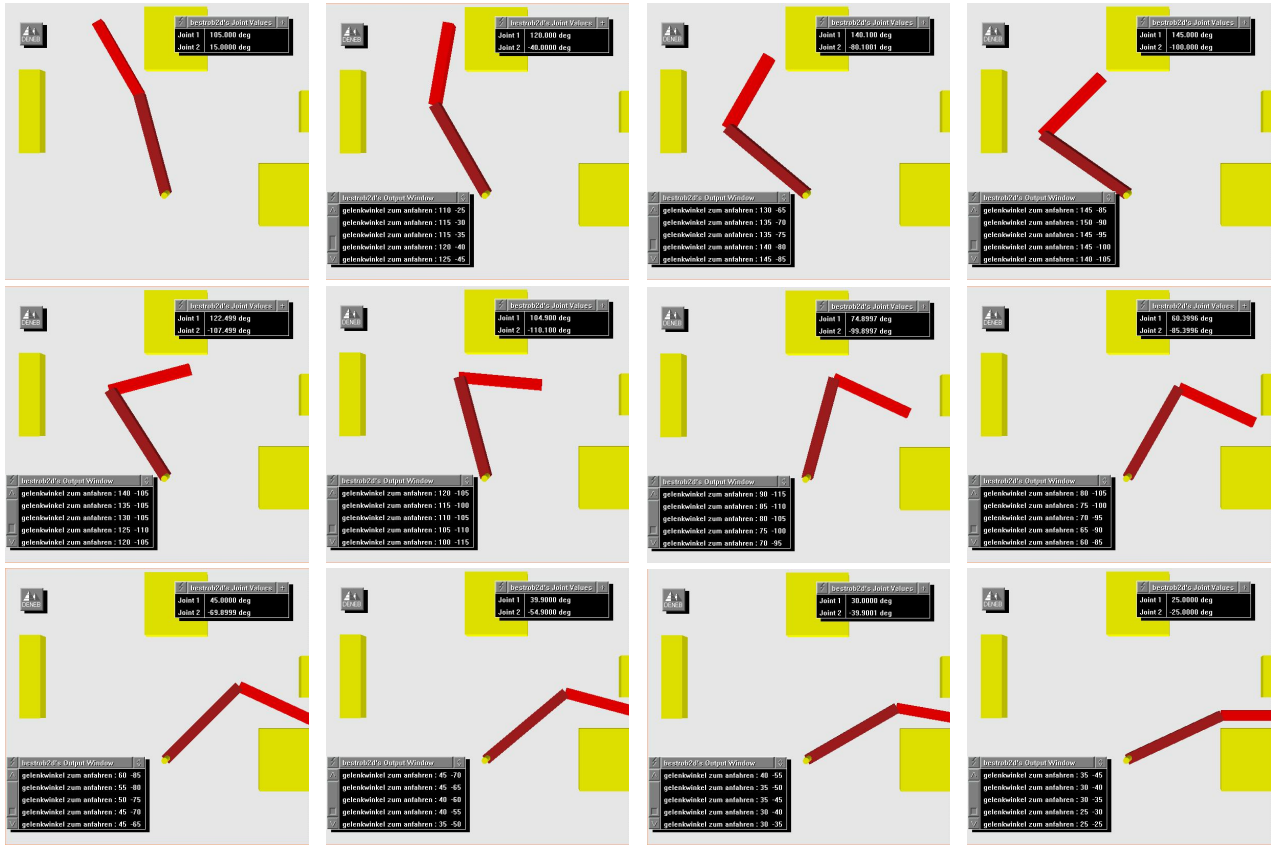
- Die Policy $\pi(s, a)$ sei ebenfalls deterministisch, dh. es gibt genau ein a mit $\pi(s, a) = 1$

- Eine Schranke Θ, bei der die Policy Evaluation beendet wird, muss gewählt werden

- Für das Problem ist das Infinite-Horizon Discounted Model am sinnvollsten, deshalb muss γ entsprechend gesetzt werden

2-gelenkiger Roboter

Die gefundene Bewegungssequenz des 2-gelenkigen Roboters (von links nach rechts, von oben nach unten):

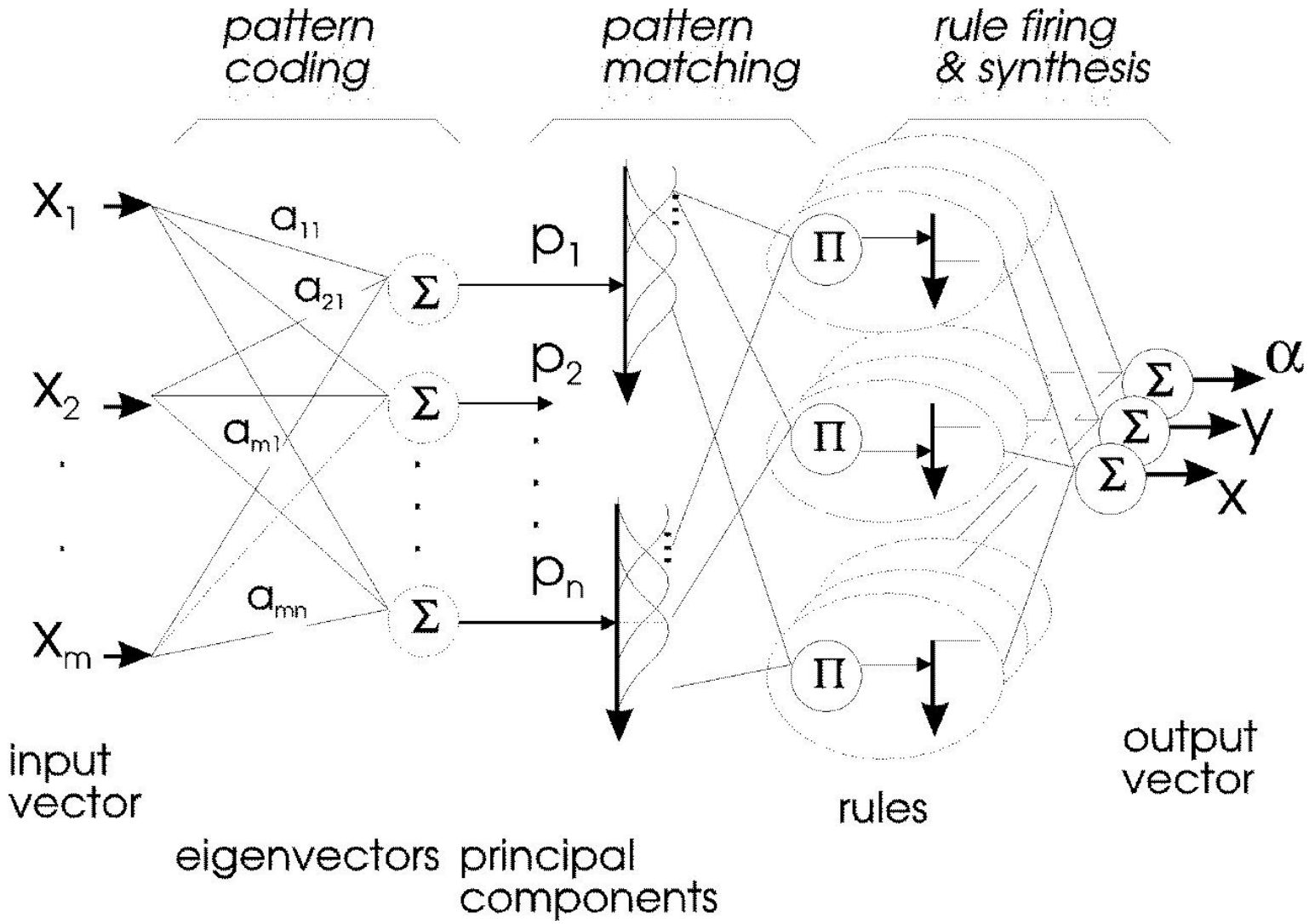


Combining Dimension Reduction with the B-Spline Model

Eigenvectors can be partitioned by covering them with linguistic terms.

Linguistic terms are defined for input variables with B-spline basis functions and for output variables with singletons.

Such a combination a linear, feed-forward neural network and a B-spline controller is referred as a neuro-fuzzy model.



The task based mapping can be interpreted as a neuro-fuzzy model.

Output-relevant features

As outlined above, PCA is a suitable approach for dimension reduction.

With the first n dimensions of the eigenspace, the original image can be reconstructed to a pre-defined resolution.

Since the magnitude of the eigenvalue corresponds to the variability of a random variable, problems may occur with input variables whose variance is low but that are nevertheless significant for controlling the process.

In such situations, with pure PCA applied to the input data set, a large number of eigenvectors are needed to represent control input variables in an appropriate way.

A solution to this problem is to use a set of vectors that directly correlate input and output space, instead of using the eigenvectors of the input data. Features that should affect the output are called $Output\ Related\ Features$ (ORF).

Based on a single-layer feed-forward perceptron network, the ORFs can be extracted through training with the $Hebbian\ learning\ rules$.

Assume that the training data are denoted by x_j $(j = 1, \ldots, k)$. If one ORF weight vector is trained which is denoted by $\vec{a}$, then the network output P is:

$$P = \sum a_j x_j = \vec{a}^T \vec{x} = \vec{x}^T \vec{a}. \tag{1}$$

Unlike PCA, which maximizes the variance of the input data along the weight vector (eigenvector), the learning rule for the ORF weight vectors is to minimize the direct error, i.e. the difference between the desired and real values of the output.

Obviously, this requires both the input x and the desired output Y_S (in our case the absolute position of the robot in a given coordinate system) to be available.

Then, one element a_j of the weight vector $\vec{a}$ can be modified as follows:

$$\Delta a_j = \eta(Y_S - P)x_j \tag{2}$$

where η is the learning rate.

To calculate more than one ORF weight vector, denoted by $\vec{a_i}, (i = 1, 2, \dots)$, we begin the computation begins with the first ORF weight vector (i=1) using Equation (2).

For calculating further $\vec{a_i}(i > 1)$, all the input data are projected onto the last ORF vectors, i.e. $\vec{a}_1, \dots, \vec{a}_{i-1}$, through which the components of the input vector, lying parallel to the ORF vector, are calculated.

These components are subtracted from the input. The element a_{ij} of the vector $\vec{a_i}$ can be then adapted by:

$$\Delta a_{ij} = \eta \left(Y_S - P_i\right) \left(x_j - \sum_{k=1}^{i-1} P_k a_{kj} \right). \tag{3}$$

Unlike the eigenvectors the ORF weight vectors are not orthogonal. Therefore, they cannot be used for reconstructing the original data unambiguously.

However, for a supervised learning system, ORFs are more efficient than principal components (alone) because they take into account the input-output relation.

When modeling a complex non-linear system, the benefit of finding the ORFs is to determine a small number of the most significant features and to isolate them through a linear transformation.

The rest of the task, i.e. modeling the non-linear part, is performed by a fuzzy controller.