Hochschule Offenburg
University of Applied Sciences

Künstliche Intelligenz
8. Reinforcement Learning

Prof. Dr. Klaus Dorer

Übersicht

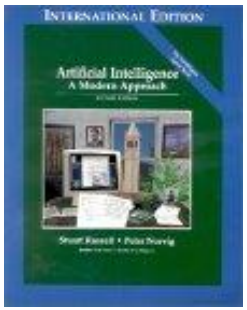
- Reinforcement Learning
  - Nutzenbasierte Agenten
    - Passiver Agent
    - Aktiver Agent
  - Exploration vs Exploitation
  - Q-Learning Agent

Ziele

- Funktionsweise von Reinforcement Learning verstehen
- Nutzenfunktion berechnen können
- Anwendungsmöglichkeiten einschätzen können

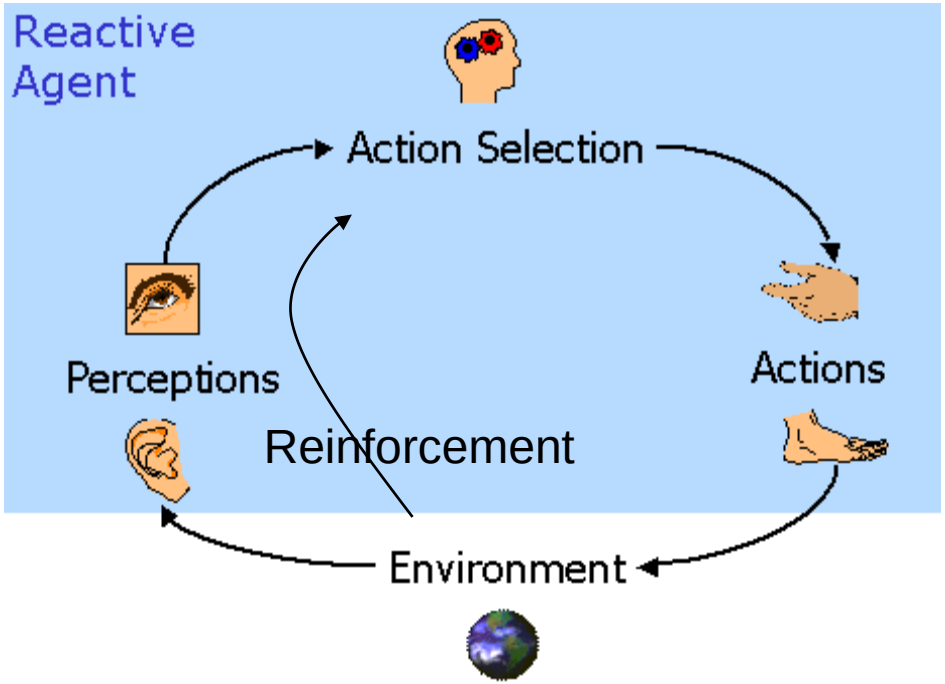
Quellen

- Russel, Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, ISBN 0137903952, 2002.
- www.wikipedia.de



Reinforcement Learning

- Agenten in einer nicht-deterministischen Umgebung
  - Nehmen die Umgebung wahr
  - Handeln und ändern damit die Umgebung
  - Erhalten positive oder negative Rückmeldung von der Umgebung (Reinforcement)

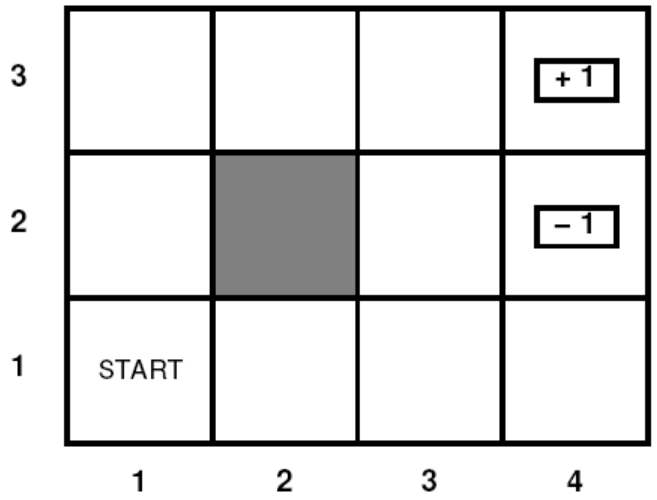


Reinforcement Learning

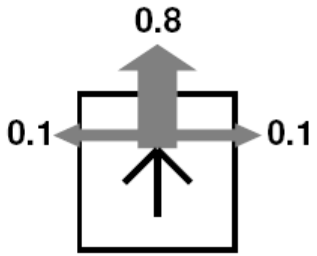
- Lernen aufgrund von positiven oder negativen Rückmeldungen
  - $o_1\ a_1\ r_2\ o_2\ a_2\ r_3\ o_3\ a_3$
  - Es muss einen ‘Trainer’ geben, der lobt oder tadelt
- Ziel
  - Langfristigen Nutzen maximieren
- Das Lernen ist schwierig, weil
  - der Agent keine Information erhält, welches die beste Aktion ist
  - Rückmeldungen zeitlich verzögert sein können und somit kein direkter Zusammenhang von Aktion zu Nutzen ersichtlich ist (temporal credit assignment problem)
- Ein reinforcement kann als Wahrnehmung modelliert werden, muss aber fest als solches verdrahtet sein
- Funktioniert prinzipiell ohne Vorwissen über die Umgebung

Nutzenbasierter Agent

- Benötigt ein Modell der Umgebung
- Beispiel
  - 4*3 Welt mit einem guten (+1) und einem schlechten (-1) Zustand
  - jeder andere Zustand hat den Nutzen -0.04
  - Aktionen Nord, Ost, Süd, West
    - sind nicht-deterministisch (b)
    - Beim Fahren gegen eine Wand bleibt man im selben Zustand



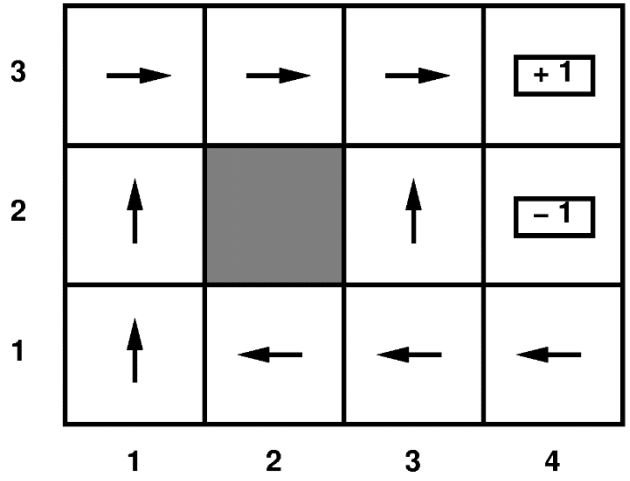
(a)



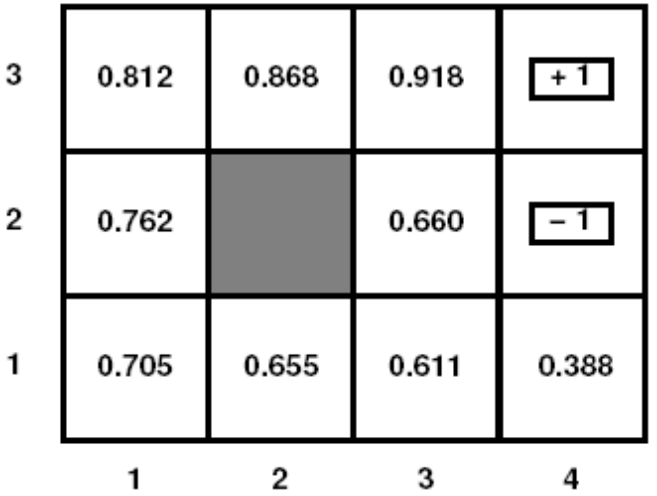
(b)

Nutzenbasierter Passiver Agent

- Strategie steht fest



- Ziel: Lernen wie gut die Strategie ist, also lernen der Nutzenfunktion



Begriffe

- Sequenz (Epoche)
  - Folge von Zuständen vom Startzustand bis zum Erreichen eines Endzustands
  - $(1,1)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(2,3)\rightarrow(3,3)\rightarrow(4,3)$
  - $(1,1)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(2,3)\rightarrow(3,3)\rightarrow(4,3)$
- Verstärkung (reward, reinforcement)
  - positive oder negative Rückmeldung
  - $(1,1)_{-0.04}\rightarrow(1,2)_{-0.04}\rightarrow(1,3)_{-0.04}\rightarrow(2,3)_{-0.04}\rightarrow(3,3)_{-0.04}\rightarrow(4,3)_{+1}$
- Zukünftiger Nutzen (reward-to-go, rtg)
  - Summe der Verstärkungen vom aktuellen bis zu einem Endzustand
  - $(1,1)_{0.72}\rightarrow(1,2)_{0.76}\rightarrow(1,3)_{0.80}\rightarrow(1,2)_{0.84}\rightarrow(1,3)_{0.88}\rightarrow(2,3)_{0.92}\rightarrow(3,3)_{0.96}\rightarrow(4,3)_{1}$

Nutzenbasierter Passiver Agent

```
public IOperator decide(IProblemState currentState)
{
      // add new perception to the list of perceptions
      percepts.add(currentState);

      // update the value function
      valueFunctionUpdateStrategy.update(currentState, percepts, lastAction);

      // end of episode?
      if (currentState.checkGoalState()) {
          percepts.clear();
      }

      lastAction = decideOnAction(currentState);
      return lastAction;
}
```

- Entscheidung basierend auf fixer Strategie π
- Interessant ist die update(..) Funktion
  - Least Mean Squares (LMS)
  - Adaptive Dynamic Programming (ADP)
  - Temporal Difference Learning (TD)

Least Mean Squares (LMS)

- Aus der adaptive control theory (Widrow und Hoff, 1960)
- Annahme: beobachteter zukünftiger Nutzen gibt direkten Hinweis auf tatsächlichen erwarteten zukünftigen Nutzen
- Am Ende einer Sequenz kann der zukünftige Nutzen und der erwartete zukünftige Nutzen als laufender Durchschnitt berechnet werden $U_i = \frac{U_i \cdot n + Rtg_i}{n+1}$
- Induktives Lernen der Nutzenfunktion aus Beispielen
- Schwäche: LMS macht keinen Gebrauch von der Tatsache, dass der Nutzen von Zuständen nicht unabhängig voneinander ist

```
public void update(IProblemState currentState, List<IProblemState> percepts, IOperator lastAction)
{
  if (currentState.checkGoalState()) {
    float rewardToGo = 0.0f;
    for (int i = percepts.size() - 1; i >= 0; i--) {
      IProblemState state = percepts.get(i);
      // calculate the reward to go
      rewardToGo += state.getReinforcement();

      // calculate the running average
      state.onExploration();
      int count = state.getExplorationCount();
      state.setUtility((state.getUtility() * (count - 1) + rewardToGo) / count);
} } }
```

Least Mean Squares (LMS)

- Beispiel
  - $U_i = \frac{U_i \cdot n + Rtg_i}{n+1}$

Bisheriger Nutzen U	0.5	0.6	0.8	0.9	0.9	1.0
Anzahl Besuche n	60	40	30	20	15	10
Sequenz	(1,1)	(1,2)	(1,3)	(2,3)	(3,3)	(4,3)
Reinforcement r	-0.04	-0.04	-0.04	-0.04	-0.04	1.0
Reward-to-go						
Neuer Nutzen (LMS)						

Adaptive Dynamic Programming

- Annahmen:
  - Modell der Übergangswahrscheinlichkeiten M_{ij} ist bekannt
  - Alle Rewards R(i) sind bekannt
- Die Nutzenwerte sind die Lösung des Gleichungssystems

$$U_i = R_i + \sum_j M_{ij} U_j$$

- Kann durch dynamische Programmierung gelöst werden
  - Value Iteration
  - Policy Iteration (ein einzelner Schritt value determination)
- Schwäche: nicht effizient berechenbar für große Zustandsräume
  - Backgammon ca 10^{50} Gleichungen in 10^{50} Unbekannten

Temporal Difference Learning

- Nutzt die beobachteten Zustandsübergänge um lokal die Nutzenwerte anzupassen
- $U_i = U_i + \alpha(N_i) \cdot (R_i + U_j\text{-}U_i)$
- Anstatt alle Folgezustände zu berücksichtigen wird nur der tatsächliche Folgezustand der Sequenz betrachtet
- Wenig wahrscheinliche Zustandsübergänge werden entsprechend selten in Sequenzen auftreten
- Sinkt die Lernrate mit zunehmender Anzahl Explorationen N_i, konvergiert U_i auf den tatsächlichen Wert
- Benötigt kein Modell der Umgebung

Temporal Difference Learning

- Beispiel
  - $U_i = U_i + \alpha(N_i) \cdot (R_i + U_j\text{-}U_i)$

Bisheriger Nutzen	0.5	0.6	0.8	0.9	0.9	1.0
Zustand	(1,1)	(1,2)	(1,3)	(2,3)	(3,3)	(4,3)
Aktion	Nord	Nord	Nord	Ost	Ost	
Reinforcement	-0.04	-0.04	-0.04	-0.04	-0.04	1.0
Neuer Nutzen (α=0.5)						

Temporal Difference Learning

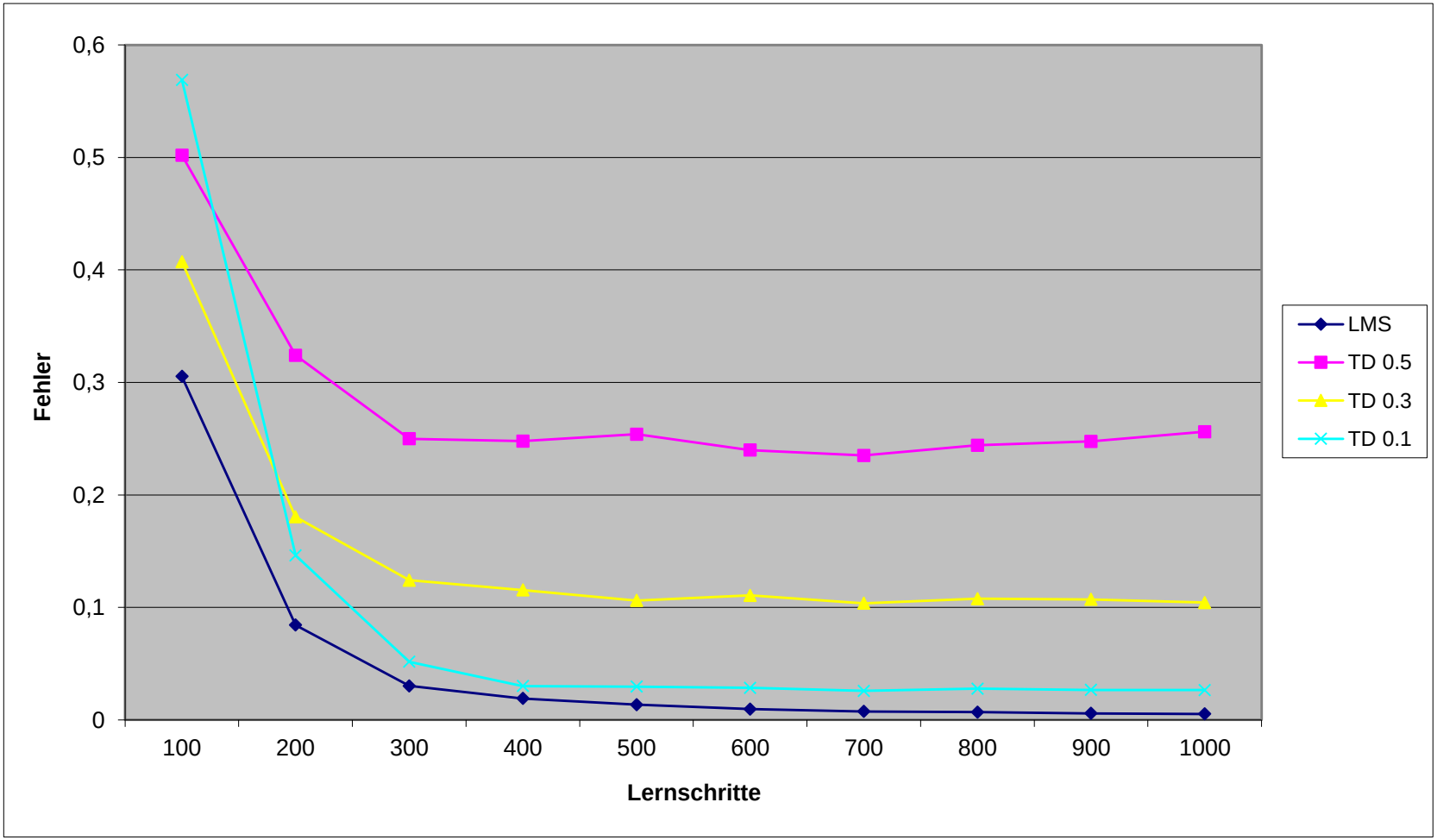
- Umsetzung

```
public void update(IProblemState currentState, List<IProblemState> percepts, IOperator lastAction)
{
  if (currentState.checkGoalState()) {
    // for goal states we can simply calculate the running average
    currentState.onExploration();
    int count = currentState.getExplorationCount();
    currentState.setUtility((currentState.getUtility()*(count-1)+currentState.getReinforcement())/count);
  }

  if (percepts.size() > 1) {
    // we have a state transition so apply TD update rule
    IProblemState penultimateState = percepts.get(percepts.size() - 2);
    penultimateState.onExploration();
    float learnrate = learnrateStrategy.getAlpha(penultimateState.getExplorationCount());
    float utility = penultimateState.getUtility();
    float reinforcement = penultimateState.getReinforcement();
    // td update rule
    penultimateState.setUtility(utility + learnrate *
                                (reinforcement + currentState.getUtility() - utility));
  }
}
```

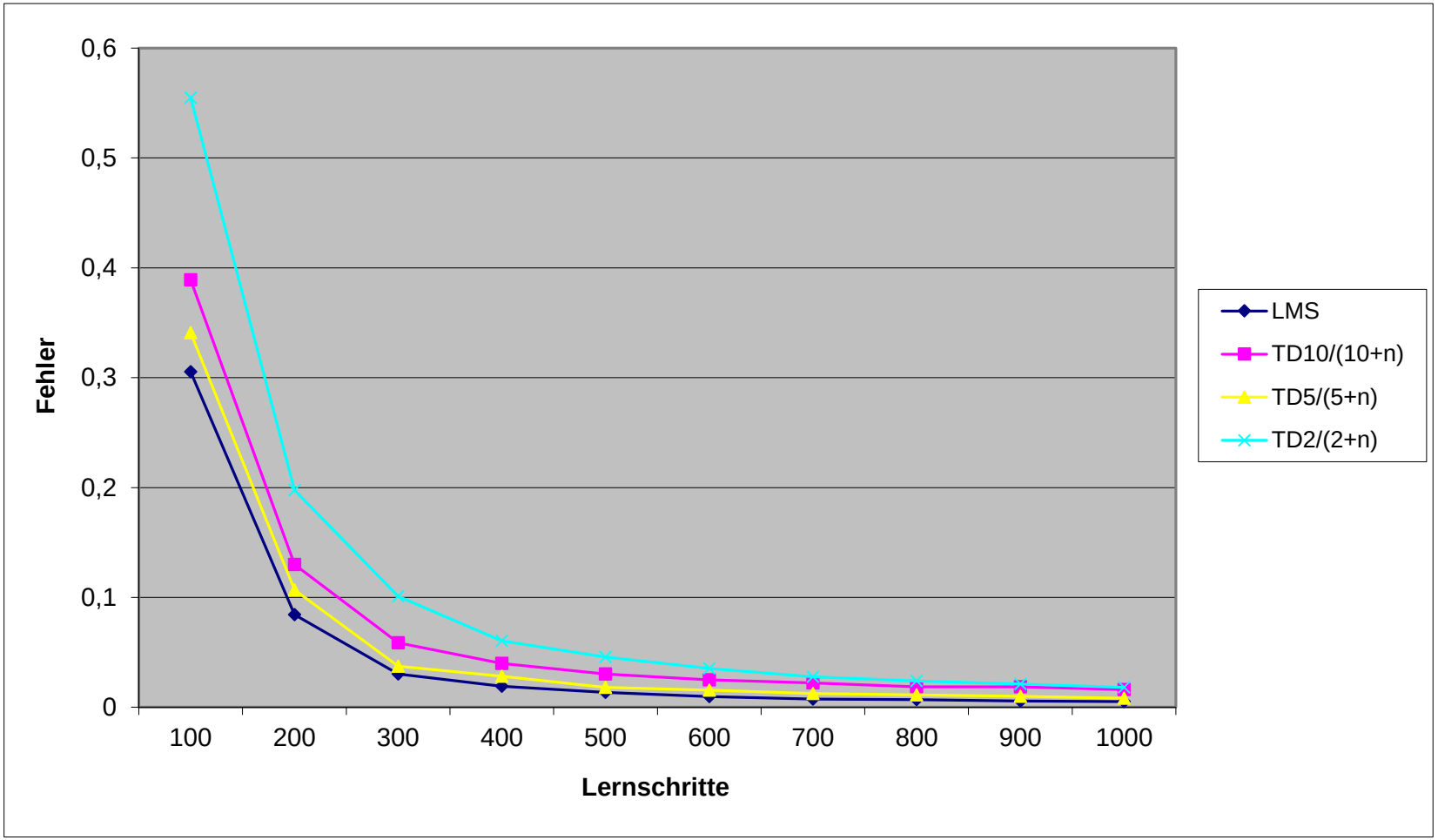
Vergleich

- LMS versus TD mit verschiedenen Lernraten



Vergleich

- LMS versus TD mit dynamischer Lernrate



Nutzenbasierter Aktiver Agent

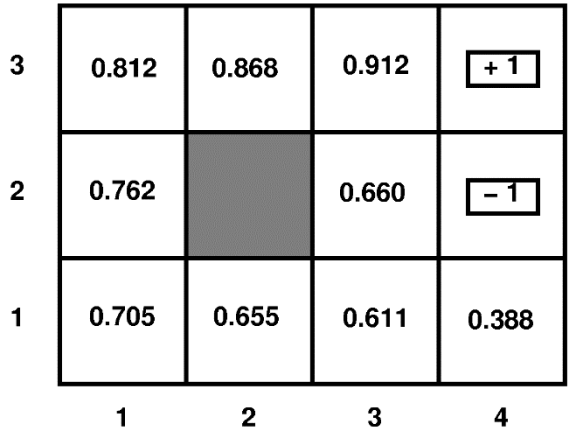
```
protected IOperator decide(IProblemState currentState)
{
  // learn model
  TransitionModel transitionModel = model.get(currentState);
  if (transitionModel == null) {
    transitionModel = new TransitionModel();
    model.put(currentState, transitionModel);
  }

  // determine best action
  List<IOperator> actions = currentState.getOperators();
  float bestExpectedUtility = Float.NEGATIVE_INFINITY;
  IOperator bestAction = actions.get(0);
  for (IOperator action : actions) {
    // check all possible successor states
    float expectedUtility = transitionModel.getExpectedUtility(action);
    if (expectedUtility > bestExpectedUtility) {
      bestAction = action;
      bestExpectedUtility = expectedUtility;
    }
  }
  return bestAction;
}
```

- Der passive Agent hatte für jeden Zustand eine vorgegebene Aktion
- Der aktive Agent muss sich für eine Aktion entscheiden
- Modell der Umgebung muss die Wahrscheinlichkeiten eines Zustandsübergangs abhängig von der Aktion M_{ij}^{a} beinhalten

Nutzenbasierter Aktiver Agent

- Der Agent wird versuchen seinen Nutzen zu maximieren
- Expected Utility $eu_{a,i} = \sum_j M_{ij}^a U_j$
- Nutzen eines Zustands $U_i = R_i + \max_a \sum_j M_{ij}^a U_j$
- Beispiel: Agent auf (3,1)
  - $eu_{Nord,(3,1)}$ =0.8*0.660 + 0.1*0.388 + 0.1*0.655 = 0.632
  - $eu_{Ost,(3,1)}$ =0.8*0.388 + 0.1*0.611 + 0.1*0.660 = 0.438
  - $eu_{Süd,(3,1)}$ =0.8*0.611 + 0.1*0.655 + 0.1*0.388 = 0.593
  - $eu_{West,(3,1)}$ =0.8*0.655 + 0.1*0.660 + 0.1*0.611 = 0.651
  - max_a -> West
  - U(3,1) = -0.04 + 0.651 = 0.611

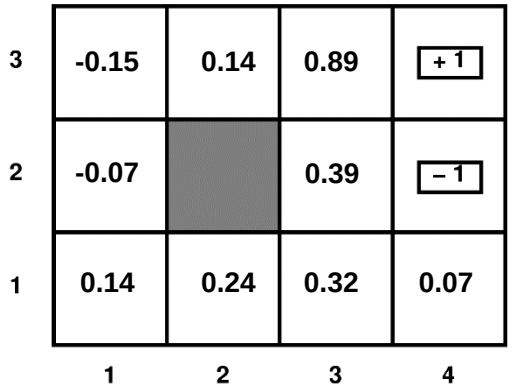


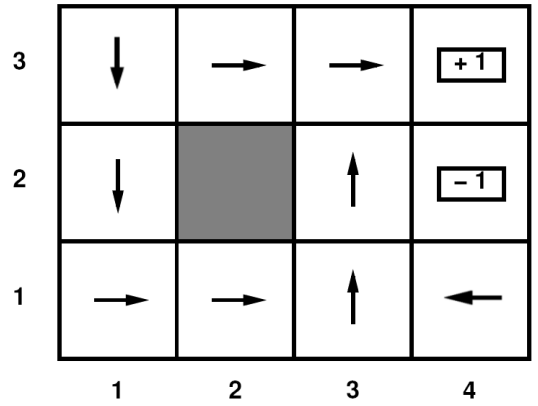
Exploration

- Aktionen haben 2 Effekte
  - Sie sammeln Belohnungen innerhalb der aktuellen Sequenz
  - Sie beeinflussen die Wahrnehmungen des Agenten, also das Lernen selbst und die Aussicht, in zukünftigen Sequenzen Belohnung zu erhalten
- Tradeoff zwischen
  - sofortigem Nutzen (gierig)
  - langzeitigem Nutzen (explorativ)
- Je länger ein Agent Zeit hat, desto explorativer sollte er sein, je kürzer er Zeit hat, desto gieriger

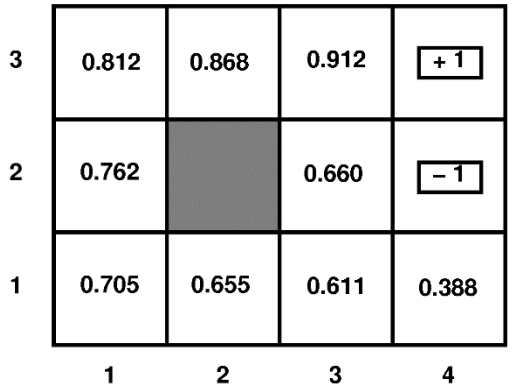
Domäne für den aktiven Agenten

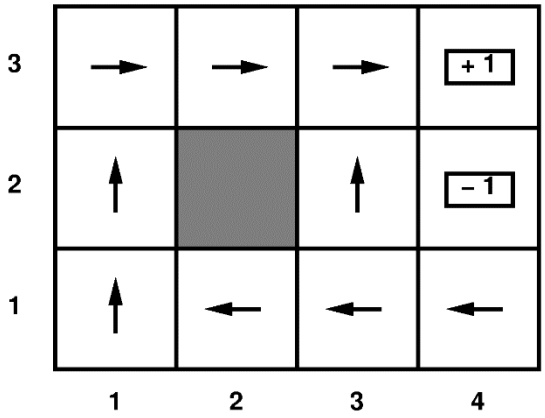
- Gieriger Agent





- Explorativer Agent





Nutzen der Zustände

Strategie

Gierig versus Explorativ

- Gieriger Agent
  - findet vielleicht den ‘unteren’ Weg
  - von da ab wird er diesen benutzen
  - Gesamtnutzen ist hoch, aber nicht optimal
  - Fehler in der Nutzenfunktion ist groß
- Explorativer Agent
  - wählt die Schritte zufällig
  - Nutzenfunktion konvergiert zu den tatsächlichen Werten
  - Gesamtnutzen ist niedriger
- Besser wäre, wenn der Agent explorativer ist, solange er die Umgebung schlecht kennt und gierig, wenn er sie gut genug kennt

Lernen der Aktion-Nutzen Funktion

- Q-Learning
- Q(a,i) ist der Nutzen im Zustand i Aktion a auszuführen
- Zusammenhang zwischen Q-Werten und Nutzen Werten

$$U_i = \max_a Q_i^a$$

- Man benötigt kein Modell der Umgebung (einfacher)
- Q-Werte können direkt aus Reinforcements gelernt werden
- Keine Vorausschau möglich
- Für korrekte Q-Werte muss gelten

$$Q_i^a = R_i + \sum_j M_{ij}^a \max_{a'} Q_j^{a'}$$

Temporal Difference Q-Learning

- Update Regel für TD Q-Learning
  - $Q_i^a = Q_i^a + \alpha(N_i) \cdot (R_i + \max_{a'} Q_j^{a'} - Q_i^a)$

$$f(u,n) = \begin{cases} R^+ \text{ if } n < N_e \\ u \text{ otherwise} \end{cases}$$

```
public void update(IProblemState currentState, List<IProblemState> percepts, IOperator lastAction)
{
  if (currentState.checkGoalState()) {
    // for goal states we can simply calculate the running average
    float reinforcement = currentState.getReinforcement();
    IOperator bestAction = qTable.getBestAction(currentState);
    float oldUtility = qTable.getUtility(currentState, bestAction);
    float count = qTable.getExplorationCount(currentState, bestAction);
    float newUtility = (oldUtility * count + reinforcement) / (count + 1);
    qTable.update(currentState, bestAction, newUtility);
  }

  if (percepts.size() > 1) {
    // we have a state transition so apply TD update rule
    IProblemState penultimateState = percepts.get(percepts.size() - 2);
    float learnrate = learnrateStrategy.getAlpha(qTable.getExplorationCount(penultimateState, lastAction));
    float oldUtility = qTable.getUtility(penultimateState, lastAction);
    float successorUtility = qTable.getBestUtility(currentState);
    float reinforcement = penultimateState.getReinforcement();
    float newUtility = oldUtility + learnrate * (reinforcement + successorUtility - oldUtility);
    qTable.update(penultimateState, lastAction, newUtility);
  }
}
```

Temporal Difference Q-Learning

- Beispiel
  - $Q_i^a = Q_i^a + \alpha(N_i) \cdot (R_i + \max_{a'} Q_j^{a'} - Q_i^a)$

Sequenz		(1,1)	(1,2)	(1,3)	(2,3)	(3,3)	(4,3)
Aktion		Nord	Nord	Ost	Ost	Ost	
Bisher gelernte Nutzenwerte	Nord	0.5	0.6	0.7	0.75	0.8	1.0
	Ost	0.35	0.5	0.8	0.9	0.9	1.0
	Süd	0.3	0.4	0.5	0.75	0.2	1.0
	West	0.4	0.5	0.6	0.6	0.7	1.0
Reinforcement		-0.04	-0.04	-0.04	-0.04	-0.04	1.0
Neuer Nutzen (α=0.5)		0.53					

Generalisierung bei RL

- Bisher sind wir davon ausgegangen, dass alle Funktionen, die der Agent gelernt hat (U, M, R, Q) in Tabellen gespeichert sind (explizite Repräsentation)
- Geht nur bei sehr kleinen Umgebungen
- Für größere Umgebungen benötigt man eine implizite Repräsentation dieser Funktionen
- Beispiel Schach
  - Ca 10^{120} Zustände (explizite Repräsentation)
  - $U(i) = w_1 f_1(i) + w_2 f_2(i) + \ldots + w_n f_n(i)$ (implizite Repräsentation)
  - Für n=10 kann man schon gut Schach spielen
- Mögliche Repräsentationen
  - Neuronale Netzwerke
  - Entscheidungsbäume (real valued)

Zusammenfassung

- Zustand-Nutzen Funktion kann durch verspätetes Reinforcement gelernt werden
- Q-Learning lernt die Aktion-Nutzen Funktion
- Funktioniert langsam und am besten mit einer Simulation

- Fürs Leben
  - Aktive Agenten sind erfolgreicher als passive