Das Learning Classifier System XCS

Andreas Bernauer

18.7.07

1 Einführung

Learning Classifier Systeme (LCS) sollen sowohl *Klassifizierungsprobleme* als auch allgemein *Reinforcement-Probleme* lösen[1]. LCS lösen diese Probleme, indem sie ein Menge von Regeln evolvieren. Das Erzeugen neuer Regeln hängt zum einen von einem Kritiker ab, der die Güte der Regeln bewertet (Reinforcement), zum anderen von einem genetische Algorithmus, der die Regeln evolviert. Die Lösung des LCS soll möglichst *genau* und möglichst *generell* sein. Möglichst genau bedeutet, dass das LCS möglichst viele Probleminstanzen korrekt klassifizieren soll. Möglichst generell bedeutet, dass die gefundene Lösung möglichst viele noch nicht bekannte Probleminstanzen korrekt klassifiziert.

Ein **Klassifizierungsproblem** besteht aus *Probleminstanzen* $s \in S$, die einer Klasse $a \in A$ angehören (in LCS-Sprechweise einer *Aktion*). In Machine-Learning-Sprechweise heißt s oft Featurevektor (engl. *feature vector*) und a Konzeptklasse (engl. *concept class*). Das *Zielkonzept* gibt die gesuchte Abbildung von S auf A an. Das Zielkonzept ist Element des Konzeptraums, also aller möglichen Abbildungen von S nach A. Das LCS soll das Zielkonzept lernen, d.h. es soll für jede Probleminstanz s_i die zugehörige Klasse a_i angeben können.

Das Standard-Klassifizierungsproblem in der LCS-Forschung sind **Boolesche Funktionen**. Im Allgemeinen ist eine Boolesche Funktion ein Abbildung $f : S = \{0,1\}^l \mapsto A = \{0,1\}$ aus dem Raum $S = \{0,1\}^l$ der Bitstrings der Länge l in die Klassenmenge $A = \{0,1\}$. Die am häufigsten verwendete Boolesche Funktion bei der Untersuchung von LCS ist der Multiplexer, da LCS dabei anderen Machine-Learning-Verfahren überlegen [10] ist. Beim Multiplexer geben die ersten k Bit des Bitstrings an, welcher der darauf folgenden 2^k Bits die Klassenzugehörigkeit angibt. Bei einem 6-Multiplexer ist zum Beispiel $f(100010) = 1$: die ersten beiden Bits (01)geben an, dass von den darauf folgenden Bits (beginnend mit der Stelle 0) das Bit an der Stelle 2 die Klasse angibt. Weitere Beispiele sind $f(000111) = 0$ und $f(110101) = 1$. Schreibt man die disjunktive Normalform des 6-Multiplexers auf,

$$6\mathrm{MP}(x_5, x_4, x_3, x_2, x_1, x_0) = \neg x_5 \neg x_4 x_3 \vee \neg x_5 x_4 x_2 \vee x_5 \neg x_4 x_1 \vee x_5 x_4 x_0 \quad (1)$$

so erkennt man, dass sich die Teilterme nicht überlappen – eine wichtige Eigenschaft für den Erfolg von LCS.

Bei einem **Reinforcement-Problem** ist das Feedback für das LCS, anders als beim Klassifizierungsproblem, nicht eine Klassenzugehörigkeit, sondern eine Belohnung, die angibt wie gut die gewählte Aktion (oder Klasse) war. Außerdem können Probleminstanzen voneinander abhängig sein, so dass Probleminstanzen von vorhergehenden Prombleminstanzen und gewählten Aktion abhängen. Zwar sind Reinforcement-Probleme damit schwieriger zu lösen als Klassifizierungsprobleme,

[1] Diese Einführung sowie weite Teile dieses Artikels sind dem sehr gut gelungenen Buch von Martin V. Butz [4] entnommen.

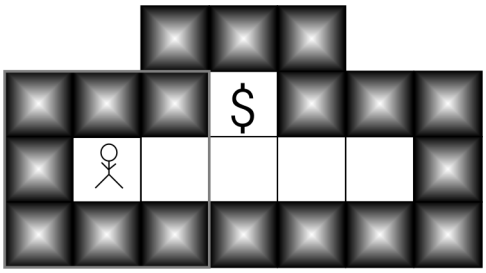

Abbildung 1: Das Animat-Problem. Der dick hervorgehebene Bereich um das Animat ist sein Wahrnehmungs- und Aktionsradius.

sie entsprechen jedoch mehr natürlich vorkommenden Problemen, wie zum Beispiel dem häufig gewählten *Animat*-Problem.

Beim **Animat-Problem** ist ein Reinforcement-Problem, bei dem sich ein so genanntes „Animat“ in einem (einfachen) Labyrinth befindet und dort nach einem Schatz suchen soll (Abbildung 1). Das Animat kann nur seine direkte Umgebung wahrnehmen, also die acht ihn umgebenden Felder (Probleminstanz), die entweder leer sind oder eine Wand. Anhand der Wahrnehmung soll das Animat entscheiden, in welche Richtung es einen Schritt machen möchte (Aktion). Das LCS soll nun die Abbildung von Wahrnehmung zu Aktion lernen, also lernen, bei welcher Wahrnehmung das Animat welchen Schritt tun muss, um dem Schatz näher zu kommen oder zu erreichen.

Klassifizierungsprobleme lassen sich als Reinforcement-Probleme formulieren, indem zum Beispiel die Belohnung bei falscher Klassifizierung 0 und bei richtiger Klassifizierung 1000 ist. Diese Art von Reinforcement-Problemen heißen *single-step Reinforcement-Probleme*, da sich die Lösung in einem Schritt ergibt. Daneben gibt es *multi-step Reinforcement-Probleme*, bei denen das LCS die Belohnung erst nach mehreren Schritten erhält und Probleminstanzen voneinander und den gewählten Aktionen abhängen. Wenn im Zusammenhang von LCS also von Klassifizierungsproblemen gesprochen wird, handelt es sich tatsächlich um single-step Reinforcement-Probleme. Die Boolesche Funktionen sind Beispiele für single-step Reinforcement-Probleme: die Klassifizierung bzw. Belohnung steht für jede Probleminstanz unabhängig von zuvor gesehenen Probleminstanzen oder Aktionen sofort fest. Das Animat ist ein Beispiel für ein multi-step Reinforcement-Problem, da die Probleminstanzen davon abhängen, welche Aktion zuvor gewählt wurde und die Belohnung (der Schatz) erst am Ende einer Reihe von Aktionen ausgegeben wird.

Das Animat ist auch ein Beispiel für eine Klasse von multi-step Reinforcement-Problemen, die bei LCS eine große Rolle spielen: die Markov-Entscheidungsprobleme (MDP, engl. *Markov decision process*). Ein MDP besteht aus

- einer Menge von möglichen Sensoreingängen (Zuständen, Wahrnehmungen) $s \in S$,
- einer Menge von möglichen Aktionen $a \in A$,
- einer Übergangsfunktion $f : S \times A \mapsto \Pi(S)$, wobei $\Pi(S)$ eine Wahrscheinlichkeitsverteilung über alle möglichen Folgezustände ist, und
- einer Belohnungsfunktion $R : S \times A \times S \mapsto \Re$.

Die Übergangsfunktion gibt den Folgezustand in Abhängigkeit vom aktuellen Zustand und der gewählten Aktion an. Die Belohnungsfunktion gibt die Belohnung an, die sich aus dem gerade vollführten Übergang ergibt. Das MDP erfüllt die Markov-Eigenschaft, da zukünftige Zustände nur vom aktuellen Zustand abhängen.

Zum Lösen von Reinforcement-Problemen setzen LCS eine Variante des Reinforcement-Lernverfahrens *Q-learning* ein. Q-learning lernt Zustand-Aktion-Paare,

d.h. es lernt, welche Belohnung es zu erwarten hat, wenn es im Zustand s_i Aktion a_i durchführt. Die erwartete Belohnung heißt *Q-Wert*. Auf lange Sicht lernt Q-Learning dadurch das Zielkonzept, also die gesucht Abbildung vom Zustandsraum (bzw. Raum der Probleminstanzen) in den Aktionsraum. Wendet das Lernverfahren im Zustand s die Aktion a an, erreicht es dadurch Zustand s' und erhält dafür die Belohnung r, so ändert sich der Q-Werte $Q(s, a)$ für das Zustand-Aktion-Paar (s, a) wie folgt:

$$Q(s,a) \leftarrow Q(s,a) + \beta(r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)) \tag{2}$$

Dabei gibt β die Lernrate an und γ den Einfluss früherer Belohnungen, die das Lernverfahren beim Erreichen von s' erhalten hatte. Ist β groß, so misst Q-learning neuen Informationen mehr Bedeutung zu als alten, wodurch sich die Q-Werte schnell ändern können; ist β klein, so misst Q-learning hingegen neuen Informationen weniger Bedeutung zu als alten, wodurch sich die Q-Werte nur langsam ändern. Oft nimmt die Lernrate β mit der Dauer des Lernens ab (wie zum Beispiel vom *Simulated Annealing* her bekannt). Analog gilt für γ: je größer γ ist, desto mehr Gewicht misst Q-learning früheren Belohnungen zu.

Wenn alle Zustand-Aktion-Paare unendlich oft ausgeführt werden und die Lernrate β entsprechend abnimmt, kann gezeigt werden, dass Q-learning zu den optimalen Q-Werten konvergiert. Sind die optimalen Werte gelernt, so ist die optimale Aktion im Zustand s

$$\pi^*(s) = \arg\max_a Q(s,a) \tag{3}$$

also die Aktion, welche die größte Belohnung erwarten lässt (den größten Q-Wert besitzt). Um sicherzustellen, dass Q-learning den gesamten Zustand-Aktion-Raum abdeckt, wählt Q-learning während des Lernens mit einer Wahrscheinlichkeit p eine zufällige Aktion:

$$\pi(s) = \begin{cases} \arg\max_a Q(s,a) & \text{mit Wahrsch. } 1-p \\ \text{random}(A) & \text{sonst} \end{cases} \tag{4}$$

LCS verwenden außerdem einen **genetischen Algorithmus**, der aus der Menge der bekannten Regeln neuer Regeln erstellt. Im weiteren wird davon ausgegangen, dass der Leser mit genetischen Algorithmen (Evaluation, Selektion, Mutation, Rekombination, Deletion) in Grundzügen vertraut ist. Darüber hinaus verweise ich auf die einschlägige Literatur wie [7] oder dem aktuellen Buch von Goldberg [8].

2 Ein einfaches Learning Classifier System: LCS1

Abbildung 2 zeigt ein typisches LCS, das so genannte LCS1 [4]. Das LCS hält eine Menge von Regeln in einer Population vor. Die Regeln bestehen aus einer Bedingung, einer Aktion und einer (erwarteten) Belohnung bzw. einem Stärkemaß. Die Regeln heißen auch *Classifier*, notiert mit cl. Die Bedingung $cl.C$ ist in der Regel binär kodiert (um mit einem genetischen Algorithmus neue Regeln erzeugen zu können), wobei '#' sowohl auf '0' als auch auf '1' passt. Die *Spezifität* $\sigma(cl)$ eines Classifiers ist das Verhältnis von '0' und '1' zu '#' in seiner Bedingung und gibt somit an, wie viel ein Classifier vom Zustandsraum abdeckt. Ein Classifier mit k '0' und '1' hat also Spezifität von $\frac{k}{l}$.

Das LCS wählt zu jeder Probleminstanz s (in der Abbildung '10010010'), die Regeln aus der Population aus, deren Bedingung auf die Probleminstanz (Wahrnehmung durch den Detektor) passen. Diese Regeln bilden das *Match-Set* $[M]$, das

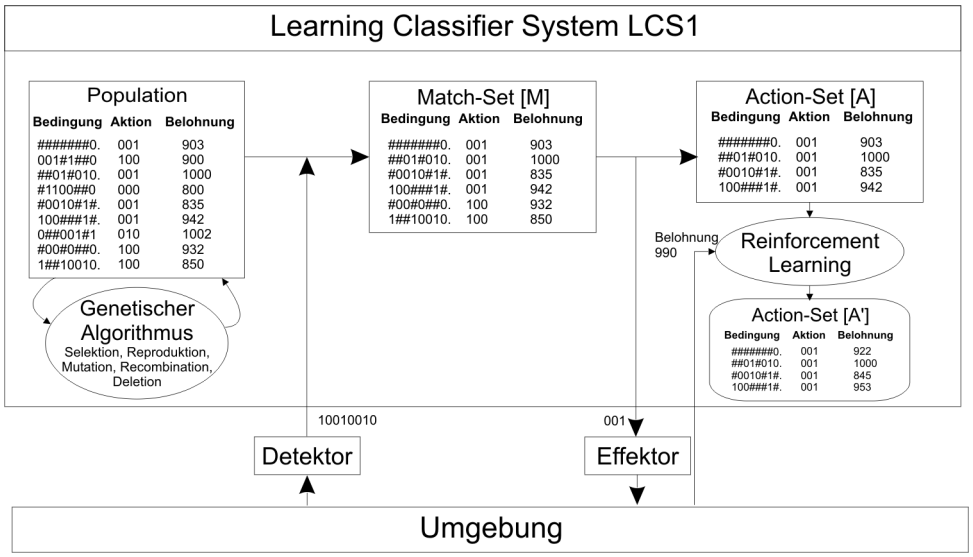

Abbildung 2: Der Aufbau eines typischen Learning Classifier System, LCS1. Details siehe Text. Nach [3]

die Menge von Aktionen vorgibt, aus denen das LCS eine wählen wird nach der Vorschrift

$$\pi_{\mathrm{LCS1}}(s) = \begin{cases} \arg\max_a \frac{\sum_{[M]|_a} cl.R}{\|[M]_|a\|} & \text{mit Wahrsch. } 1-p \\ \mathrm{random(A)} & \text{otherwise} \end{cases} \tag{5}$$

wobei $[M]|_a = \{cl \in [M] \,|\, cl.A = a\}$ die Menge aller Classifier ist, welche die Aktion a vorschlagen. Hat sich das LCS für die Aktion $a' = \pi_{\mathrm{LCS1}}(s)$ entschieden (in der Abbildung '001'), so bildet es das Action-Set $[A] = [M]|_{a'}$, das aus allen Classifiern des Match-Sets $[M]$ besteht, welche die Aktion a' vorschlagen.

Nachdem das LCS1 für die Aktion a' die Belohnung r erhalten hat (in der Abbildung $r = 900$), und sich für die darauf folgende Probleminstanz s^+ das Match-Set $[M]^+$ gebildet hat, aktualisiert das LCS die Vorhersage der erwarteten Belohnung in $[A]$ nach der angepassten Q-learning Gleichung

$$R \leftarrow R + \beta \left(r + \gamma \cdot \max_a \frac{\sum_{[M]^+|_a} cl.R}{\|[M]^+|_a\|} - R \right) \tag{6}$$

welche die maximale, zukünftige Belohnung aus dem Durchschnitt aller teilnehmenden Classifier schätzt. Im Unterschied zum Q-learning verwendet das LCS statt einem Classifier eine Menge von Classifiern, um die zukünftige Belohnung abzuschätzen. Wären alle Classifier spezifisch (d.h. $\sigma(cl) = 1 \; \forall cl$), würde LCS1 Q-learning durchführen.

Das LCS1 erzeugt neue Regeln nach zwei Verfahren: *Covering* und *genetischer Algorithmus*. Das Covering kommt zum Einsatz, wenn kein Classifier auf die Probleminstanz passt und kommt damit vor allem zu Beginn des Lernens vor. Beim Covering wird ein Classifier erzeugt, dessen Bedingung bis auf einer zufälligen Zahl an Stellen, die auf '#' gesetzt werden, genau der Probleminstanz entspricht und der eine zufällige Aktion besitzt. Der genetische Algorithmus (GA) ist in der einfachsten Form ein *steady-state* GA, der entsprechend der erwarteten Belohnung einer Regeln zwei Regeln aus der Population wählt, mutiert, rekombiniert und wieder in die Population einfügt, wobei zwei andere Regeln zufällig entsprechend der Inversen

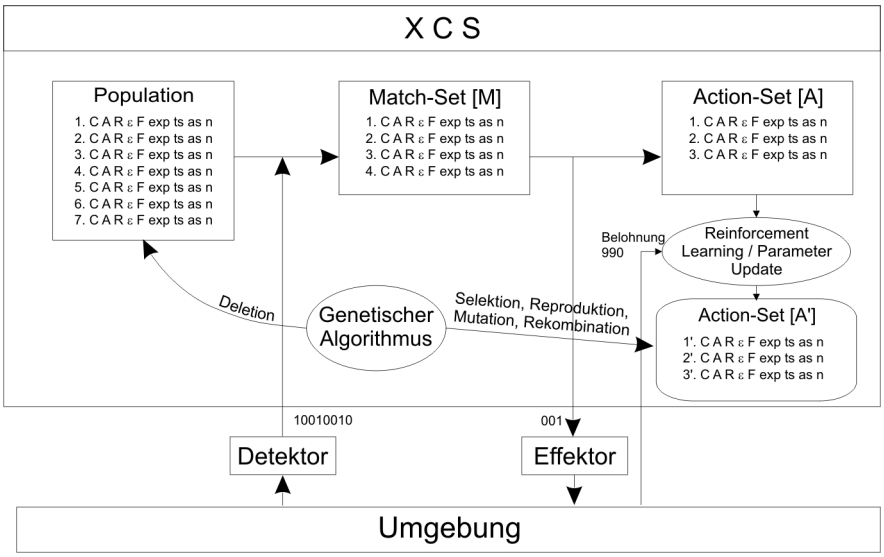

Abbildung 3: Lernen im XCS. Nach [3]

.

der erwarteten Belohnung entfernt werden. Für einen erfolgreicheren GA sei auf [3] verwiesen.

3 Das XCS Classifier System

Das XCS Classifier System wurde erstmals von Wilson [19] vorgeschlagen und stellt einen Meilenstein in der Entwicklung der Learning Classifier Systeme dar. Es unterscheidet sich von anderen LCS in hauptsächlich in folgender Hinsicht (vgl. auch Abbildung 3).

- Die Fitness einer Regel im genetischen Algorithmus hängt nicht von der Stärke der Regel ab, sondern von der Genauigkeit der Vorhersage der Belohnung. Damit können auch Regeln in der Population verbleiben, die zwar absolut gesehen wenig Belohnung bringen, diese jedoch exakt vorhersagen können.
- Der genetische Algorithmus arbeitet nur auf Basis der Match-Sets bzw. Action-Sets, löscht Regeln jedoch in der Gesamtpopulation. Dies ermöglicht es dem XCS auch in Nischen zu lernen, da immer nur die für eine Situation zutreffenden Regeln zur Produktion neuer Regeln herangezogen werden.

Als Ergebnis stellte Wilson [19] fest, dass der XCS dazu neigt, die Abbildung $(C, A) \Rightarrow P$ von der Menge der Paare von Bedingungen (C) und Aktionen (A) zu der Menge der Belohnungen (P) vollständig und genau zu lernen. Außerdem tendiert XCS dazu, die Regeln so generell wie möglich zu halten. Diese Ergebnisse wurde in einem späteren, detaillierteren Artikel von Kovacs [11] bestätigt. Damit erfüllt das XCS die zwei wichtigsten Eigenschaften, die an LCS gestellt werden.

Formal besteht ein XCS-Classifier aus fünf Hauptbestandteilen sowie einigen weiteren. Die fünf Hauptbestandteile sind:

- Die *Bedingung* c, die angibt, wann der Classifier anwendbar ist.
- Die *Aktion* a, welche die vorgeschlagene Aktion angibt.
- Die *Vorhersage* r der Belohnung, wenn C gilt und A angewendet wird.

- Der *Vohersagefehler* ϵ, der den durchschnittlichen, absoluten Fehler der Vorhersage R von der tatsächlichen Belohnung angibt.
- Die *Fitness* f, welche die Genauigkeit (den inversen Fehler) des Classifiers relativ zu anderen, ihn überlappenden Classifier angibt.

Die Bedingung c, die Aktion a und die Vorhersage r der Belohnung haben dieselbe Syntax und Bedeutung wie beim LCS1. Hinzugekommen sind die Vorhersagefehler ϵ, die der XCS dazu verwendet, Classifier zu bevorzugen, die möglichst genaue Belohnungsvorhersagen machen können und damit ausgezeichnete Spezialisten in ihrer Nische sind, und die Fitness f, welche der genetische Algorithmus verwendet, um nur Classifier gegeneinander antreten zu lassen, die auch echte Konkurrenten sind.

Neben den fünf Hauptbestandteilen gibt es folgende weitere Bestandteile des XCS-Classifiers:

- Die Schätzung der *Action-Set-Größe as*, die den laufenden Durchschnitt der Action-Set-Größe angibt, in denen der Classifier war.
- Ein *Zeitstempel ts* (engl. *timestamp*) der angibt, wann der Classifier das letzte Mal in einer GA-Runde konkurrieren musste.
- Der *Erfahrungswert exp* (engl, *experience*), der angibt, wie oft die Parameter des Classifiers geändert wurden.
- Die *Numerosität num*, welche die Anzahl an identischen Micro-Classifiern angibt, die dieser Classifier repräsentiert. Damit kann ein Classifier mehrere identische Classifier darstellen.

Die Aktualisierung der Parameter beim XCS-Verfahren verläuft ähnlich zum Verfahren beim LCS1, mit den folgenden Änderungen:

- Nach dem Bilden des Match-Sets $[M]$, kann das XCS einen *Vorhersagevektor* $P(A)$ bilden, der die Belohnung für jede mögliche Aktion angibt:

$$P(a) = \frac{\sum_{[M]|_a} cl.r \cdot cl.f}{\sum_{[M]|_a} cl.f} \tag{7}$$

  $P(a)$ gibt damit den mit der Fitness gewichteten Durchschnitt aller vorhergesagten Belohnung in $[M]$ wieder, die als Aktion a vorschlagen. Das XCS verwendet den Vorhersagevektor, um eine geeignete Aktion zu wählen, wobei es verschiedene Modi für diese Wahl gibt. Während der Lernphase wählt das XCS die Aktion normalerweise zufällig, während der Anwendungsphase wählt es $a_{\max} = \arg\max_a P(a)$.

- Das XCS aktualisiert seine Parameter normalerweise in der folgenden Reihenfolge: Vorhersagefehler ϵ, Vorhersage r und Fitness f. Sei $P^{t+1}(A)$ der Vorhersagevektor der nächsten Runde, dann sei $Q = \max_{a \in A} P^{t+1}(a)$.

- Der Vohersagefehler ϵ eines jeden Classifier cl in $[A]$ ergibt sich damit zu:

$$cl.\epsilon \leftarrow cl.\epsilon + \beta(|\rho - R| - cl.\epsilon) \tag{8}$$

  wobei in Klassifizierungsproblemen $\rho = cl.r$ und in Reinforcement-Problemen $\rho = cl.r + \gamma Q$ ist. β und γ haben die selbe Bedeutung wie beim LCS1.

- Die Vorhersage r der Belohnung berechnet sich mit:

$$cl.r \leftarrow cl.r + \beta(\rho - R) \tag{9}$$

XCS führt damit ein Q-learning durch, das jedoch auf allen Classifiern in $[A]$ basiert.

- Die Aktualisierung des Fitnesswertes f eines jeden Classifiers in $[A]$ hängt von seiner relativen Genauigkeit κ' ab, die von dem Vohersagefehler ϵ wie folgt ableitet:

$$cl.\kappa = \begin{cases} 1 & \text{falls } cl.\epsilon < \epsilon_0 \\ \alpha \left(\frac{cl.\epsilon}{\epsilon_0}\right)^{-\nu} & \text{sonst} \end{cases} \tag{10}$$

$$cl.\kappa' = \frac{cl.\kappa \cdot cl.num}{\sum_{cl' \in [A]} cl'.\kappa \cdot cl'.num} \tag{11}$$

  Hier misst $cl.\kappa$ die momentane absolute Genauigkeit des Classifiers und verwendet dabei eine Potenzfunktion mit Exponenten ν, um Classifier mit kleinen Fehlern zu bevorzugen, d.h. Classifier, deren Fehler unter ϵ_0 sinkt, werden als genau betrachtet.

  Die Aktualisierung des Fitnesswertes erfolgt mit

$$cl.f \leftarrow cl.f + \beta(cl.kappa' - F) \tag{12}$$

- Die Action-Set-Größe as wird nach dem gleichen β-Schema aktualisiert:

$$cl.as \leftarrow cl.as + \beta(\|[A]\| - cl.as) \tag{13}$$

- Jedes Mal, wenn die Parameter eines Classifiers aktualisiert werden, wird sein Erfahrungswert $cl.exp$ um eins erhöht.

- Die Parameter $cl.r$, $cl.\epsilon$ und $cl.as$ werden nur aktualisiert, wenn der Erfahrungswert eines Classifiers kleiner $1/\beta$ ist (*moyenne adaptive modifiée*, [18]).

Auch der GA besitzt Änderungen zum Verfahren im LCS1:

- Der GA operiert auf dem aktuellen Action-Set $[A]$, jedoch nur, wenn seit der letzten Anwendung des GAs die Zeit θ_{GA} verstrichen ist.

- Der GA wählt zwei Eltern-Classifier mit der Wahrscheinlichkeit

$$p_s(cl) = \frac{cl.f}{\sum_{cl' \in [A]} cl'.f} \tag{14}$$

  Die Eltern verbleiben in $[A]$ und konkurrieren damit mit ihren Nachkommen.

- Der GA gewinnt die Nachkommen aus den Eltern p durch Crossover. Die Parameter der Nachkommen initialisiert es wie folgt. Der GA mutiert die Bedingung entweder beliebig oder so, dass die Bedingung immer noch zur aktuellen Wahrnehmung s passt (Nischen-Mutation). $cl.r = R$, $cl.\epsilon = \text{avg}_{cl \in [A]}(cl.\epsilon)$, , $cl.f = 0,1p.f$, $cl.as = p.as$ und $cl.exp = cl.num = 1$.

- Bevor der GA die Kinder in die Population einfügt, prüft er, ob die Kinder von einem hinreichen genauen ($\epsilon < \epsilon_0$) und ausreichend erfahrenen ($exp > \theta_{\text{sub}}$) Classifier in $[A]$ überdeckt wird. In diesem Fall fügt der GA das Kind nicht hinzu und erhöht die Numerosität des überdeckenden Classifiers.

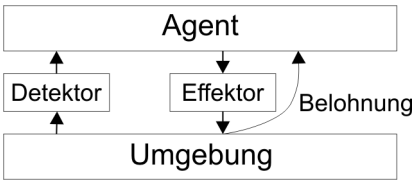

Abbildung 4: Typischer Aufbau bei einem Reinforcement-Problem.

- Der GA fügt die Kinder der Population hinzu. Überschreitet die Population eine gewisse Maximalgröße N, so löscht der GA Regeln mit einer Wahrscheinlichkeit, die proportional zu $cl.as$ ist. Wenn ein Classifier ziemlich erfahren ist ($cl.exp > \theta_{\text{del}}$), seine Fitness $cl.f$ jedoch deutlich schlechter als der Durchschnitt $\bar{f}$ in der Gesamtpopulation ist ($cl.f < \delta\bar{f}$), so erhöht sich seine Wahrscheinlichkeit, dass der GA ihn aus der Population entfernt, um $\bar{f}/f$.

Detaillierte Angaben zum Algorithmus XCS finden sich in [2]. Das Buch [3] gibt darüber hinaus Informationen über theoretische Grundlagen.

4 Ohne Formeln: Was macht ein LCS?

Learning Classifier Systeme (LCS) sind eine Klasse von maschinellen Lernverfahren, die Reinforcement-Verfahren und genetische Algorithmen verbinden. Sie werden seit Ende der 1980er Jahre erforscht. John H. Holland, einer der führenden Wissenschaftler in diesem Bereich, nimmt dabei folgende Umgebung für das LCS an, die typisch für Reinforcement-Verfahren ist (vgl. auch Abbildung 4):

- Das LCS besitzt eine Menge von *Detektoren* (wie z.B. die Sehsinneszellen in der Retina), mit dem es die Umgebung wahrnimmt.
- Die Informationsverarbeitung findet mit standardisierten *Nachrichtenpaketen* statt. Das LCS verarbeitet diese Nachrichtenpakete und erzeugt daraus neue Nachrichtenpakete.
- Das LCS kann eine Reihe von *Effektoren* ansteuern, welche die Umwelt verändern.
- Das LCS erhält von der Umwelt eine Belohnung, die angibt, wie „gut“ es (re)agiert hat (*feedback*).

Nach Holland geht ein Classifier System vor allem drei Probleme von maschinellen Lernverfahren an [9]: Interaktion und Koordination von Regeln, Zuweisung von Credits und Finden neuer Regeln.

Interaktion und Koordination von Regeln Ein maschinelles Lernverfahren kann unmöglich jegliche Kombination von Wahrnehmungen im Vorfeld kennen (z.B. „ein roter VW an der Straßenseite mit einem platten Reifen“). Wenn das Lernverfahren jedoch Basisblöcke kennt (z.B. „roter VW“, „Straßenseite“ und „platter Reifen“), kann es damit die Situation relativ leicht erfassen.

Ein Regel-basiertes Lernverfahren kann also eine Menge neuer Situation erfassen, wenn es auf die Situation mit Hilfe einer Kombination von Basisblöcken oder Regeln reagieren kann. Durch die Vielzahl an möglichen Kombinationen kann das Regel-basierte Lernverfahren viele verschiedene Situationen erfassen. Da günstige Basisblöcke oft und in zahlreichen Situationen vorkommen, kann das Lernverfahren sie oft und ausgiebig testen und validieren. Das Problem besteht jedoch darin, wie

das Lernverfahren die Interaktion und Koordination einer großen Zahl von gleichzeitig aktiven Regeln bewerkstelligt.

Ein LCS geht das Problem der Interaktion und Koordination von Regeln dadurch an, dass es die Aktion einer Regeln darauf beschränkt, eine Nachricht zu versenden. Die Nachrichten können *tags* besitzen, die sich leicht zu ihrer Adressierung und zur Koordination von Aktionen verwenden lassen. Das LCS ist damit nicht mehr als ein System zur Verarbeitung einer aktuellen Liste von Nachrichten. Da die Nachrichten einfach nur Regeln aktivieren, vermeidet das LCS Probleme mit der Widerspruchsfreiheit der Regeln: es gibt einfach nur eine Liste von aktiven Regeln.

Zuweisung von Credits Ein großes Problem bei Regel-basierten Lernverfahren stellt die Frage dar, welche Regeln nun zum großen Erfolg des Lernverfahrens beigetragen hat. Dies ist insbesondere schwierig, wenn erst eine Reihe von Aktionen nötig sind, um ein positives Resultat zu erhalten (z.B. das Opfern eines Bauerns im Schach).

In realen Umgebungen ist es nicht möglich, alle möglichen Aktion-Reihenfolgen zu testen (wie im Dynamic Programming oder Q-learning). LCS versuchen das Problem dadurch zu lösen, indem sie eine Marktsituation aufstellen: jede Regel agiert als Mittelsmann (Broker) einer Kette von Regeln von der aktuellen Situation und zu einem (möglicherweise positivem) Ergebnis. In jedem Glied wettet jede Regeln, deren Bedingungen erfüllt sind, darauf, dass sie aktiviert wird. Wenn die Regel aktiviert wird, zahlt sie ihren Wetteinsatz an die aktiven Regeln vor ihr in der Kette (Lieferanten), die es ihr ermöglicht haben, aktiv zu werden. Die aktivierte Regel darf nun die Wetteinsätze ihrer aktivierten Nachfolger (Konsumenten) erhalten und bildet somit Kapital (*Stärke der Regel*). Der letzte Konsument ist die Umgebung, die dem LCS eine Belohnung für ihre Aktionen gibt.

Finden neuer Regeln Das Finden neuer Regeln stellt für Regel-basierte Lernverfahren ein großes, noch ziemlich unergründetes Problem dar. Klar ist, dass schlechte Regeln ersetzt werden müssen. Unklar ist jedoch, durch was sie ersetzt werden sollen. Zufällige Regeln sind nur für kleine Probleme hilfreich. Bei großen Problemen liegt der Schlüssel zum Erfolg darin, die angesammelte Erfahrung zu nutzen und daraus plausible Hypothesen bzw. Regeln über die Situation abzuleiten (ohne offensichtliche Widersprüche einzuführen).

LCS erzeugen neue Regeln mittels genetische Algorithmen und nutzen deren Fähigkeit, vorhandene Basisblöcken neu zu kombinieren. Der genetische Algorithmus arbeitet dabei auf beide Arten von Basisblöcken, die in LCS vorkommen: die Basisblöcke, aus denen Regeln aufgebaut sind (Bedingungen und Aktionen), und die „Basisblöcke“ von Regeln, die zu einer Aktion geführt haben. Die Stärke einer Regel geht dabei als Fitnesswertes in den genetischen Algorithmus ein und wirkt damit auf beiden Basisblock-Ebenen.

Jeder der beschriebenen Mechanismen sollen es dem LCS ermöglichen, sich ständig an seine Umgebung anzupassen und dabei Schritt für Schritt den Anforderungen der Umgebung gewachsen zu sein. Das LCS versucht dabei laufend *exploration* (Aufnahme neuer Information) und *exploitation* (vorhanden Informationen und Fähigkeit effizient nutzen) zu balancieren.

4.1 Schwierigkeiten beim LCS

Bei der Verwendung von LCS treten einige Schwierigkeiten auf, die im Folgenden kurz angesprochen werden (und aus [16] entnommen sind). Das Lernverfahren XCS geht die meisten dieser Schwierigkeiten an.

Einige Schwierigkeiten kommen daher, dass LCS meist als Verfahren beschrieben wird (so auch hier) wohingegen es sich eher um eine Herangehensweise handelt. Bei der Verwendung des LCS muss man sich dann festlegen, wie man die Herangehensweise tatsächlich umsetzt. Unter anderem sieht man sich folgenden Fragen gegenüber:

- Was ist eine gute Repräsentation der Umwelt innerhalb des LCS?
- Wie teilt man die Belohnung unter den Regeln auf?
- Wie kann man die interne Nachrichtenverarbeitung so ändern, dass das LCS in Umgebungen ohne Markov-Eigenschaft[2] lernen kann?

Anschließend, wenn man das LCS dann umsetzt, erkennt man einige der folgenden Erscheinungen, die allesamt einen negativen Einfluss auf den Lernprozess haben:

- Manchmal können ein paar starke Regeln viele schwache Regeln verdrängen, obwohl diese ebenfalls zur guten Performance des LCS beitragen.
- Dieses Problem ist oft mit dem Problem der Verteilung der Belohnung unter den Regeln verknüpft, etwa wem in der Regel-Kette soll das LCS wie viel von der Belohnung zuweisen?
- Durch das Problem der Verteilung der Belohnung kommt es vor, dass sehr generell gehaltene Regeln entstehen (Regeln mit vielen '#'), die andere, speziellere Regeln übermäßig stark verdrängen.
- Der Versuch, durch die Anpassung der internen Nachrichtenverarbeitung das Lernen in Umgebungen ohne Markov-Eigenschaft zu ermöglichen, scheint diese Probleme nur weiter zu verstärken und führte bisher zu parasitären Regeln. Parasitäre Regeln erhalten zwar regelmäßig eine Belohnung, weil sie sich in die Regel-Kette einbauen, senken jedoch die Leistung des Systems.

5 Ausblick

In [21] erwähnt Wilson mögliche Richtungen für zukünftige Forschung im Bereich XCS ab dem Jahr 2000, von denen ich hier einige aufgreife:

- Die Syntax der Bedingungen so erweitern, dass XCS auch Zustände mit kontinuierlichen, ordinalen oder gemischten Werten erfassen kann. Die Syntax sollte auch Regelmäßigkeiten in der Umgebung leichter abbilden können.

  Hierzu gab es in der Zwischenzeit schon zwei Arbeiten. In [20] und [17] beschreiben Wilson und Stone et al., wie das XCS mit reellen Zustandswerten umgehen kann. In [4] beschreiben Butz et al. wie die Regeln Hyperellipsoide im Zustandsraum abdecken können, statt Hyperkubi, wie das bisher der Fall war.

- Um ein besseres Modell der Umgebung zu erhalten, das XCS lernen zu lassen, den nächsten Zustand vorherzusagen. Ich finde das einen interessanten Ansatz. Allerdings sind mir hierzu keine Arbeiten bekannt.

- Herausfiltern von Rauschen jeglicher Art, um reale Datensätze lernen zu können. Hierzu gibt es zahlreiche Artikel, welche die Anwendbarkeit des XCS im Data Mining beschreiben. Einer der ersten ist von Wilson selbst [22]. Dort berichtet er von der erfolgreichen von XCS auf synthetische Datensätze und auf

[2] In Umgebungen, welche die Markov-Eigenschaft besitzen, hängt die optimale Aktion nur von der aktuellen Situation ab, wie zum Beispiel im Schach. In Umgebungen, die keine Markov-Eigenschaft besitzen, hängt die optimale Aktion auch von vorhergehenden Zuständen ab, wie zum Beispiel beim Skat.

den Wisconsin-Breast-Cancer-Datensatz. Bei letzterem war es möglich, anhand der Regeln für Menschen verständliche Zusammenhänge im Wisconsin-Breast-Cancer-Datensatz herauszufinden. In [6] beschreiben Dam et al. ein verteiltes XCS und vergleichen es mit dem (normalen) zentralisiertem XCS bei verrauschten Daten. Die Ergebnisse hier sind leider nicht so deutlich wie bei [22].

- Andere Reinforcement-Learning Methoden außer Q-learning versuchen. Hierzu sind mir keine Arbeiten bekannt.
- Entwicklung einer Theorie für die genetische Suche, wie sie XCS anwendet. Das Buch von Butz [3] ist eine erste Zusammenfassung der Theorie zum XCS, die auch die Suche von XCS theoretisch untersucht.

Hinzu kommen Forschungsbereiche, zu denen bisher nichts bekannt ist und in denen ich in der Zwischenzeit auch keine Fortschritte feststellen konnte:

- Die Aktionen kontinuierliche Werte verwenden lassen, etwa „um 34 Grad drehen“.
- Kontinuierliche statt diskrete Zeitschritte verwenden.
- Das Verhältnis von XCS zu anderen Classifier-Systemen untersuchen.

Zu dem letzten Punkt gibt es zwar einige Arbeiten, die verschiedene Classifier-Systeme vergleichen, etwa [12] und [1]. Allerdings sind diese Vergleiche eher punktuell und bieten kein allgemeines Verständnis für die Beziehung von XCS zu anderen Classifier-Systemen.

Literatur

[1] E. Bernadó, X. Llorà, and J. M. Garrell. XCS and GALE: A Comparative Study of Two Learning Classifier Systems on Data Mining. In Lanzi et al. [14], pages 115–132.

[2] M. Butz and S. W. Wilson. An Algorithmic Description of XCS. In Lanzi et al. [14], pages 253–272.

[3] M. V. Butz. *Rule-Based Evolutionary Online Learning Systems*. Springer-Verlag, Berlin, 2006.

[4] M. V. Butz, P. L. Lanzi, and S. W. Wilson. Hyper-ellipsoidal conditions in XCS: rotation, linear approximation, and solution structure. In Cattolico [5], pages 1457–1464.

[5] M. Cattolico, editor. *Genetic and Evolutionary Computation Conference (GECCO 2006)*. ACM, July 2006.

[6] H. H. Dam, H. A. Abbass, and C. Lokan. DXCS: an XCS system for distributed data mining. In Rothlauf [15], pages 1883–1890.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[8] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[9] J. H. Holland, L. B. Booker, M. Colombetti, M. Dorigo, D. E. Goldberg, S. Forrest, R. L. Riolo, R. E. Smith, P. L. Lanzi, W. Stolzmann, and S. W. Wilson. What Is a Learning Classifier System? In Lanzi et al. [13], pages 3–32.

[10] K. A. D. Jong and W. M. Spears. Learning Concept Classification Rules using Genetic Algorithms. In *Proceedings of the Twelfth International Conference on Artificial Intelligence (IJCAI-91)*, volume 2, pages 651–657, 1991.

[11] T. Kovacs. XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions. In Roy, Chawdhry, and Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 59–68. Springer-Verlag, London, 1997.

[12] T. Kovacs. Strength or Accuracy? Fitness Calculation in Learning Classifier Systems. In Lanzi et al. [13], pages 143–160.

[13] P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors. *Learning Classifier Systems: From Foundations to Applications*, volume 1813 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2000.

[14] P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors. *Advances in Learning Classifer Systems*. Number 2321 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001.

[15] F. Rothlauf, editor. *Genetic and Evolutionary Computation Conference (GECCO 2005)*. ACM, June 2005.

[16] R. E. Smith, B. A. Dike, B. Ravichandran, A. El-Fallah, and R. K. Mehra. The Fighter Aircraft LCS: A Case of Different LCS Goals and Techniques. In Lanzi et al. [13], pages 283–300.

[17] C. Stone and L. Bull. For real! XCS with continuous-valued inputs. *Evol. Comput.*, 11(3):299–336, 2003.

[18] G. Venturini. Adaptation in dynamic environments through a minimal probability of exploration. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: from animals to animats 3 (SAB94)*, pages 371–379, Cambridge, MA, USA, 1994. MIT Press.

[19] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[20] S. W. Wilson. Get Real! XCS with Continuous-Valued Inputs. In Lanzi et al. [13], pages 209–222.

[21] S. W. Wilson. State of XCS Classifier System Research. In Lanzi et al. [13], pages 63–82.

[22] S. W. Wilson. Mining Oblique Data with XCS. In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Advances in Learning Classifer Systems*, number 1996 in Lecture Notes in Artificial Intelligence, pages 158–174, Berlin, September 2001. Springer-Verlag.