ENCYCLOPEDIA OF COGNITIVE SCIENCE

2000

Learning Rules and Productions

Machine Learning # Cognitive Modelling # Production Rules # Generalization # Concept learning

Taatgen, Niels

Niels A. Taatgen

University of Groningen Netherlands

[Definition Describes the algorithms and mechanisms involved in learning new rules (or productions) from examples and existing rules in both Machine Learning and Cognitive Modelling. Machine Learning is focussed on finding a rule set that efficiently characterizes a concept, while cognitive modelling tries to understand human learning.

Introduction

Rules are popular means of knowledge representation used in several different domains of cognitive science. Not only are they a powerful form of representation, they differ from many other types of representation in the sense that they incorporate both the knowledge itself, and the way to use this knowledge.

Different types of rules

Rules are small units of knowledge that, although they work in concert with other rules, are relatively independent, as opposed to a line of code in an arbitrary programming language. The attractive property of this independence is that knowledge can be build up incrementally. The focus of this article will be on two types of rules, logical rules and production rules.

Logical rules are a subclass of first-order logic expressions, often Horn clauses. A Horn clause is an expression of the form:

$(L_1 \text{ and } L_2 \text{ and } ... \text{ and } L_n) \rightarrow H$

In this expression, L_1 to L_n and H are all literals. Horn clauses closely resemble production rules, which have the following form:

IF condition_1 and condition_2 and ... and condition_n THEN action_1, action_2 ... action_m

For the present discussion, we will largely ignore the difference between the two, but the reader must be aware that they have different properties. As an example, production rules are generally used left-to-right: once the conditions of the rule are satisfied, the action can be carried out. Horn clauses on the other hand, when used in a Prolog context, are used the other way around: in order to satisfy some predicate, a rule is selected that has the predicate as a conclusion.

Goals of learning rules

Before examining mechanisms for rule learning, it is useful to characterize the context in which rules are learned, and the goals of rule learning. Two broad fields can be distinguished: Machine Learning and Cognitive Modelling.

Machine Learning

The main focus of rule learning in machine learning is to learn rules that characterize concepts. The goal of learning is to find a set of rules that can decide whether or not a particular example is an instance of a certain concept. A concept can correspond to a natural category like a bird, a dog or a chair, or concepts like "paper accepted at the cognitive science conference". Suppose we want to characterize this latter category, papers accepted at a certain conference. The final part of the decision process is whether or not to accept a paper given the judgments of the reviewers. Table 1 gives an example of judgments.

	Relevance A	Technical A	Overall A	Relevance B	Technical B	Overall B	Accept?
1	Good	Fair	Good	Fair	Fair	Good	Yes
2	Good	Good	Good	Fair	Poor	Poor	No
3	Good	Good	Good	Fair	Poor	Fair	Yes
4	Fair	Poor	Fair	Good	Good	Good	Yes
5	Good	Fair	Poor	Good	Good	Fair	No

Table 1. Five judgments in the conference example

The goal of rule learning is to find a rule or set of rules that characterizes the concept of "accepted paper". Given the five examples we may come up with the following rules:

$(\text{Overall A} = \text{Good}) \wedge (\text{Overall B} \neq \text{Poor}) \rightarrow \text{Accept} = \text{yes}$

$(\text{Overall B} = \text{Good}) \wedge (\text{Overall A} \neq \text{Poor}) \rightarrow \text{Accept} = \text{yes}$

The assumption in this and later examples will be that if Accept is not set to "yes" by some rule, Accept will be "no". Although this may look like a very plausible

characterization of the five examples, it is by no means the only one. It may be too general, as it does not take into account other attributes than Overall A and Overall B.

The following three rules also characterize the five examples:

(Relevance A = Good) ∧ (Technical A = Fair) ∧ (Overall A = Good) ∧
(Relevance B = Fair) ∧ (Technical B =Fair) ∧ (Overall B = Good) → Accept = yes

(Relevance A = Good) ∧ (Technical A = Good) ∧ (Overall A = Good) ∧
(Relevance B = Fair) ∧ (Technical B =Poor) ∧ (Overall B = Fair) → Accept = yes

(Relevance A = Fair) ∧ (Technical A = Poor) ∧ (Overall A = Fair) ∧
(Relevance B = Good) ∧ (Technical B = Good) ∧ (Overall B = Good) → Accept = yes

This set of rules, though technically correct, is much less satisfactory, because it just lists examples 1, 3 and 4, and is therefore too specific. The goal of a good rule learning algorithm is to find the right set of rules for a certain set of positive and negative examples.

The general procedure in machine learning is that the learning algorithm is trained on a set of examples of which the answer is provided. After learning, the rule set that has been developed is tested on new set of examples, the test set. The quality of the algorithm is judged by the efficiency of the algorithm, and the score on the test set.

Cognitive Modelling

If rules are considered as not merely convenient representations, but as atomic components of human knowledge (e.g., Anderson & Lebiere, 1998) then such a theory of human knowledge has to include mechanisms to learn these rules. Whereas the goal of Machine Learning is to find a rule set that characterizes some concept, the focus in cognitive modelling is more on the learning process than the learning outcome. A cognitive modeller tries to produce a computer simulation that mimics human learning as closely as possible. Cognitive modelling approaches that use rules have to answer the question how these rules are learned, and what the effects of rule learning are on performance.

An issue in cognitive modelling is how task-specific rules can be learned on the basis of general rule knowledge on the one hand and instruction and experience on the other hand. Anderson (1987) uses the example of learning to program in Lisp. When novices have to learn a new skill like programming, they rely not only on general instruction but also on examples that can be used as templates.

In an experiment, participants were given a template on how to define Lisp functions, and an example:

```
(DEFUN <function name>
        (<parameter 1><parameter 2> ... <parameter n>)
        <process description>)

(DEFUN F-TO-C (TEMP)
        (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
```

They were then given the assignment to write a Lisp definition of a new function FIRST that returns the first element of a list. Many participants came up with the following definition:

```
(DEFUN FIRST (LIST1)
       (CAR (LIST1)))
```

They had produced this definition by using both the general template and the example, but had wrongly generalized the (DIFFERENCE TEMP 32) part to produce (LIST1) in the answer. The parentheses are present because DIFFERENCE is itself a function call. No parentheses are needed in the case of LIST1 as this is one of the parameters. The correct solution is:

```
(DEFUN FIRST (LIST1)
       (CAR LIST1)
```

Anderson's rule learning system, which we will examine in detail later on, produced the following two rules in a simulation of the acquisition knowledge in this task:

```
IF      the goal is to write a function of one variable
THEN    write (DEFUN function (variable)
        and set as a subgoal to code the relations calculated by this function
        and then write )

IF      the goal is to code an argument
        and that argument corresponds to a variable of the function
THEN    write the variable name
```

Note that these functions are generalizations of both examples, but that the first rule is a specialization of the general template, as it only applies to functions of one variable.

In the Lisp example, general knowledge and a single example are used to find the solution to a new example. Except for the template and the example, general strategies like analogy are assumed in the model. Almost as a by-product rules are learned. Although this setting is different from the Machine Learning perspective, there is a strong resemblance: rules are learned on the basis of examples, in this case with some domain knowledge. In cognitive modelling the focus is also on errors and speed, so the cognitive model also has to make the same errors people do, and show the same increase in performance due to practice.

Algorithms for concept learning

Single hypothesis learning

An early rule learning algorithm was developed by Winston (1970). Many variants have been produced, but the basic idea is very simple. A single rule or rule set is maintained, and this rule is adjusted as new examples arrive.

When a new example is presented, it is first checked whether the rule is already consistent with the example. When the example is inconsistent, we have to update the rule. If the example is a positive example, we have to generalize the rule, so that it

will include the new example. In the case of a negative example, the rule has to be specialized to exclude the new example.

Consider a simplification of the conference acceptance problem in Table 2.

	reviewer A	reviewer B	Accept?
1	Good	Good	Yes
2	Good	Poor	No
3	Good	Fair	Yes
4	Fair	Good	Yes
5	Poor	Fair	No

Table 2. Simplification of the conference example

A possible rule based on the first example is:

$A = \text{Good} \rightarrow \text{Accept} = \text{yes}$ (1)

The second, negative example is inconsistent with this rule, so specialization is needed, for example by adding a condition:

$A = \text{Good} \wedge B \neq \text{Poor} \rightarrow \text{Accept} = \text{yes}$ (2)

The third example is consistent with this rule, so no further modification is necessary. The fourth example, however, again requires a generalization of the rule. This may be achieved by dropping a condition, resulting in:

$B \neq \text{Poor} \rightarrow \text{Accept} = \text{yes}$ (3)

Example five is again inconsistent with the rule, requiring a final specialization:

$B \neq \text{Poor} \wedge A \neq \text{Poor} \rightarrow \text{Accept} = \text{yes}$ (4)

A problem with this approach is that with each generalization or specialization we have to make sure all the previous examples are still consistent with the current rule. Also, finding a new hypothesis can become very hard, as there are many possible generalizations and specializations, and they are not all as easy to derive from the current rule.

Version Space

In the example two methods were used to update the rule: generalization and specialization. Rules can be ordered with respect generality: rule *p* is said to be more general than rule *q* when all instances that are included in the concept by rule *q* are also included by rule *p*. This ordering is a partial order. The most general rule is the rule that includes all instances of the concept, and the most specific rule is the rule that includes no instances at all. All other rules are in between these extremes. This view on hypotheses offers a handle for a more systematic approach than the one offered by single hypothesis learning. Figure 1 shows part of the hypothesis space for the conference problem.

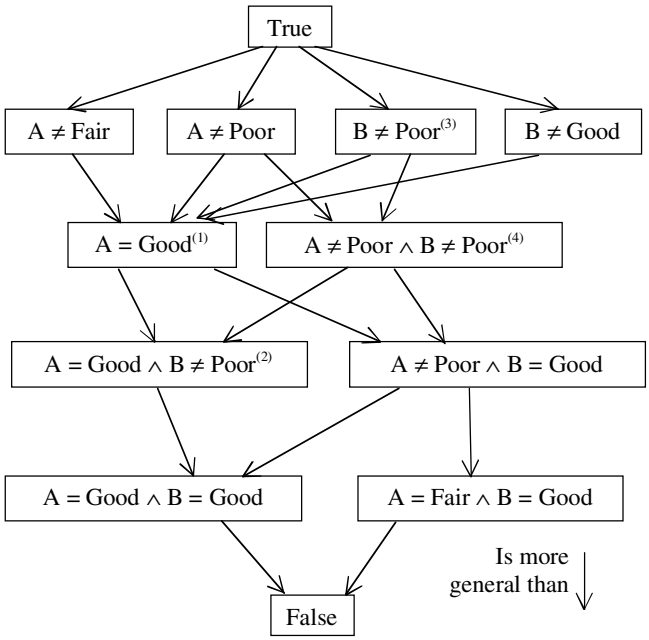

Figure 1. Part of the hypothesis space of the conference problem

The most general hypothesis is that all papers are accepted (True), and the most specialized hypothesis is that none are accepted (False). All other hypotheses are ordered in between. The numbers in the picture refer to the four hypotheses we had in the previous example: each time a positive example is not included in the current hypothesis, we have to generalize and move up in hypothesis space, and each time a negative example is not excluded we have to move down. The version space, the space of all plausible versions of the concept, is the subset of all possible hypotheses that are still consistent with the examples we have seen up to now.

Version Space Learning

Instead of maintaining a single hypothesis about the target concept, an alternative is to represent all hypotheses that are still consistent with the present set of examples. The naive version would be to list all the hypotheses, but fortunately this is not necessary. Due to the fact that hypotheses are partially ordered by the more-general-than relation, it is only necessary to represent the set of the most general hypotheses that are still consistent with all examples, and the set of the most specific hypotheses that are still consistent with all examples. These two sets, usually designated *G* and *S*, respectively, are often called boundary sets, as everything more general than *G* or more special than *S* is not consistent with the examples, but everything in between is. The algorithm that maintains these boundary sets is known as version space learning (Mitchell, 1977). *G* is initialized to {True}, and *S* to {False}. For each example, *G* and *S* are updated along the following lines:

- For positive examples:
    - Remove all members of *G* that do not include the example.

- For each member of *S* that does not include the example, replace it by all immediate generalizations of that member that do include the example, and are specializations of some member of *G*.

- For negative examples:
  - Remove all members of *S* that do not exclude the example.
  - For each member of *G* that does include the example, replace it by all immediate specializations of that member that do not include the example, and are generalizations of some member of *S*.

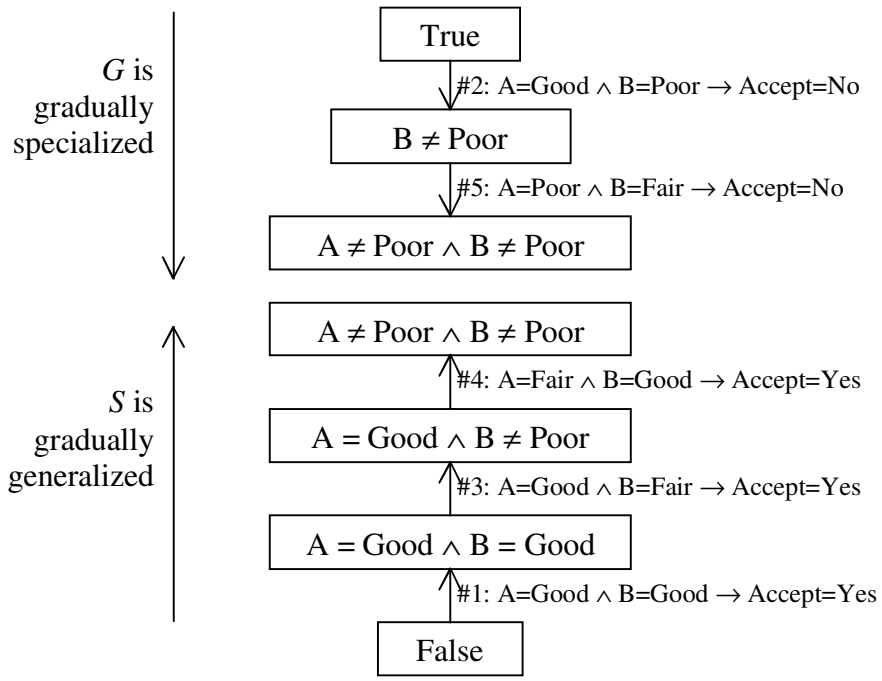

Figure 2. Version space learning operating on the conference example.

Figure 2 shows how version space learning would handle the conference acceptance example. The algorithm starts out with *G* set to {True} and *S* set to {False}. The first example is a positive example: A=Good ∧ B=Good. This means that False is no longer the most specific hypothesis, so it is generalized to A=Good ∧ B=Good. The second example is a negative example, which means that the True hypothesis needs specialization. Although there are two specializations of True consistent with the second example (A ≠ Good, B ≠ Poor), only B ≠ Poor is also a generalization of a member of *S*. The process continues until, after example 5, both *G* and *S* have converged to the hypothesis A ≠ Poor ∧ B ≠ Poor. Note that in this example both *G* and *S* only contain one hypothesis at a time, this is not true in general.

A disadvantage of version space learning is that it cannot handle noise: it relies on the fact that all examples are correct. Another problem is that it cannot handle disjunction very well. In our example we have only used a limited form of disjunction (in the sense that B ≠ Poor means B = Good ∨ B = Fair). The case A = Fair ∧ B = Fair is classified as a positive example, although it has never been presented as an example,

and it would be perfectly plausible if it would have been a negative example (in the case the criterion is "At least one Good and no Poors").

Other concept learning algorithms

Except for single hypothesis learning and version space learning many other algorithms have been developed for concept learning. Decision tree learning, for example, learns a decision tree for a concept. Learning is however not incremental, because the algorithm learns the whole tree on the basis of a set of examples. As a decision tree is not really a rule-based representation, it is beyond the scope of this article. Another family of algorithms, often shared under the heading of inductive logic programming, infers sets of rules to characterize concepts instead of single rules. The advantage of using a set of rules is that single rules only cover part of the concept, so there is no problem of overly specific rules. Inductive logic programming algorithms also operate on the set of examples as a whole.

Learning with Domain Knowledge

A property of the algorithms discussed in the previous paragraph is that they operate on the basis of examples only. In addition to examples there may be other knowledge that may guide the learning process, for example common sense knowledge, or a complete set of domain knowledge. An algorithm that does include the use of domain knowledge is Explanation Based Learning (e.g., DeJong, 1981).

Explanation Based learning

The assumption of Explanation Based Learning (EBL) is that we have a complete set of knowledge, the domain knowledge, that is in principle enough to make decisions, but may be computationally intractable. The goal of EBL is, given a proof for a single example, to derive new rules of intermediate computational complexity that are generalizations of the example, but specializations of the domain theory. Suppose we have the following rule set to make conference decisions:

$\text{reviewer}(X) \wedge \text{review}(X, \text{poor}) \rightarrow \text{negative}$
$\text{reviewer}(Y) \wedge \text{review}(Y, \text{good}) \rightarrow \text{positive}$
$\text{negative} \rightarrow \text{decide}(\text{no})$
$\text{positive} \wedge \neg\text{negative} \rightarrow \text{decide}(\text{yes})$

Given a specific example, for instance $\text{reviewer}(a) \wedge \text{reviewer}(b) \wedge \text{review}(a, \text{good}) \wedge \text{review}(b, \text{good})$ a general problem solver like Prolog can derive $\text{decide}(\text{yes})$. EBL now uses the proof (or explanation, hence the name) of $\text{decide}(\text{yes})$ to generate a new rule that is a generalization of the example but a specialization of the domain theory:

$\neg\text{review}(a,\text{poor}) \wedge \text{review}(b,\text{good}) \rightarrow \text{decide}(\text{yes})$

This newly learned rule does not contribute anything new, as all the knowledge is already contained in the domain knowledge. However, if the domain knowledge itself is very inefficient to use, new efficient rules may effectively extend the capabilities of the system by allowing new proofs that were previously computationally unachievable. Take the example of mathematics: the natural numbers can be defined

by a small set of axioms, but mathematicians use many derived rules to solve actual problems.

In EBL new rules are added to the rule set, so in that sense it differs from algorithms discussed earlier where learned rules replaced old rules. This introduces a new problem: adding rules to the system may improve its performance because rules are tailored to certain often-occurring situations, but may also decrease performance if they are never used. This *utility problem* is an issue different from correctness: knowledge that is true may still be undesirable because it is useless. Possible solutions to the utility problem are to develop procedures that estimate the cost/benefit properties of a rule in advance, or to just introduce them into the system and keep track of how they fare (e.g., Minton, 1988).

Learning rules in cognitive models

Rule learning in cognitive modelling has aspects in common with both the purely inductive algorithms like version space learning and deductive algorithms like explanation based learning. Human learners have a large store of background knowledge, strategic knowledge and domain knowledge, but still have to make generalizations from examples that are not fully deducible from the domain knowledge. A general view in cognitive modelling is that humans have a set of weak methods that, when supplied with some background knowledge and a particular case to work on, will produce the desired performance, and are also the basis for learning. Weak methods are problem solving strategies that are independent of the particular problem, and are generally applicable. Examples of these strategies are: means-ends analysis, forward-checking search, analogy, etc.

In the Lisp learning example in the introduction the weak method of analogy was used to generate a new Lisp program on the basis of an example and some background knowledge on Lisp. During this process some new rules were learned, and the mechanisms cognitive modellers use to achieve this rule learning show a resemblance to explanation based learning. The difference is that learning and problem solving happen at the same time, and domain knowledge is often incomplete.

Chunking in Soar

Newell and Rosenbloom (1981) proposed a rule mechanism called *chunking* that became an important component of the Soar cognitive architecture (Newell, 1990). Within Soar, learning rules is tied to impasses and subgoaling. Whenever Soar reaches a state in which it runs into some impasse (no applicable rules, an irresolvable choice between operators, etc.), it automatically creates the subgoal to resolve this impasse. When the subgoal is successfully completed and the impasse is resolved, a specialized rule is learned that summarizes all the processing required to achieve that subgoal. If Soar later encounters a similar impasse, it no longer needs a separate subgoal to process it. Instead it can use the learned rule to solve it in a single step.

Although Soar's basic set of methods encompasses several weak methods, the method reported most often is forward checking. Suppose we have the blocks-world problem in Figure 3.

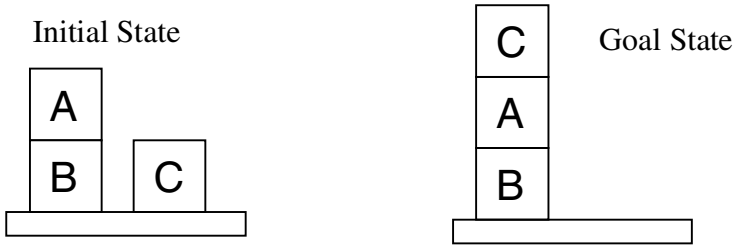

Figure 3. Example blocks-world problem

If Soar were to solve this problem, it would discover that there are two possible operators in the initial state: "Move block A to C" and "Move block C to A". If Soar would have no additional knowledge on how to choose between these operators (called search-control knowledge), it would run into a so-called tie impasse. To resolve this impasse, Soar would create a subgoal to resolve this impasse by evaluating these two operators. Evaluation is can be done by "mental simulation": what would happen if each of the tied operators is applied? In this mental simulation Soar would quickly discover that "Move C to A" is the best operator as it immediately accomplishes the goal. So the subgoal created by the impasse comes back with the resolution "Move C to A is best", and in the main goal this operator is applied.

Learning occurs directly after the subgoal is resolved. The rule that is created has as a condition the circumstances in which the impasse occurred, and as action the result of the subgoal, in this case the "best-preference" for the operator that accomplishes the goal:

IF the problem-space is a simple-blocks-world
and the state has block X and block Y clear,
and block X is on the table,
and the goal state has block X on block Y,
THEN make a best preference for the operator that move block X onto block Y

The consequence of this rule is that whenever Soar encounters the situation described in the rule, an impasse no longer occurs and an operator is chosen immediately.

A problem with Soar's rule learning mechanism is that there is no solution to the utility problem: Soar can produce over-specific rules that are expensive to match and has no way to get rid of them once learned.

Production learning in ACT

Anderson (1983) proposed a set of four rule-learning mechanisms that were specified to work in the ACT* cognitive architecture. An aspect of ACT* is that it not only has a memory for rules (procedural memory), but also a memory for facts (declarative memory). The following four mechanisms were used to learn new rules:

1. Proceduralization. If a rule accesses declarative memory, and uses the knowledge from declarative memory in its action, then learn a new rule that is identical to the old rule but has the retrieved knowledge instantiated into the rule, eliminating the declarative retrieval.

2. Composition. Collapse a sequence of rules into a single rule that performs all the actions of the individual rules of the sequence.

3. Generalization. If there are two rules that are similar, create a generalized version of these rules. This can be done by removing conditions or by substituting constants by variables.

4. Discrimination. If a rules is successful in one situation but not in another, add conditions to the rule to make a more restrictive version that only applies in successful situations.

In the Lisp-learning example proceduralization and composition are used to learn the two rules to write lisp functions on the basis of the weak method of analogy.

Despite the relative success of the mechanisms in ACT*, they were too unconstrained, and were able to produce too many invalid rules and too many rules with poor utility. In the latest version of ACT (ACT-R 5.0), a constrained version of the four mechanisms is introduced. This single mechanism combines the proceduralization and composition mechanisms from ACT*. In ACT-R, the expressive power of production rules is more constrained when compared to earlier versions of ACT and when compared to Soar. Each production rule is only allowed a single access to declarative memory. Also, this access is composed of two steps. First, a rule has to issue a request to declarative memory for a certain fact as part of its action side, and a second rule can match the retrieved fact on its condition side. Suppose we want to add three numbers. In the older ACT and Soar systems, this would require only a single rule. In ACT-R, we need three rules to accomplish this:

Rule 1:	Rule 2:	Rule 3:
IF the goal is to add three numbers THEN send a retrieval request to declarative memory for the sum of the first two numbers	IF the goal is to add three numbers AND the sum of the first two numbers is retrieved THEN send a retrieval request to declarative memory for the sum of the currently retrieved sum and the third number	IF the goal is to add three numbers AND the sum of the first two numbers and the third number is retrieved THEN the answer is the retrieved sum

Using one of the older composition mechanisms, one general rule could be learned out of these three rules to do three-number additions in one step. From a cognitive perspective this is not desirable, as people generally cannot do these types of additions in one step, although they have ample experience with them. Also, in ACT-R it would be no longer possible, as only one retrieval from declarative memory is allowed in each rule.

Rule learning in ACT-R is aimed at composing two rules that fire in sequence into one new rule, while maintaining the constraint that only one retrieval from declarative memory is allowed. This is done by eliminating a retrieval request in the first rule and a retrieval condition in the second rule. The fact that has been retrieved is filled in in the combined action of the new rule. Suppose that in the example above, the three

numbers that are added are 1, 2 and 3, then this would produce two new rules, a combination of rule 1 and 2, and a combination of rule 2 and 3. Each of these two new rules can be combined with one of the original rules to learn a rule that combines all three rules:

Rule 1 & 2:	Rule 2 & 3:	Rule 1 & 2 & 3:
IF the goal is to add 1, 2 and a third number THEN send a retrieval request to declarative memory for the sum of 3 and the third number	IF the goal is to add three numbers and the third number is 3 AND the sum of the first two numbers is retrieved and is equal to 3 THEN the answer is 6	IF the goal is to add 1, 2 and 3 THEN the answer is 6

Compared to the original rules these rules are very specialized: they work only for certain numbers. The implication is that people will only learn specific rules for very common additions: it is likely that one immediately knows the sum of 1, 2 and 3, but not of 9, 3 and 4.

Although the example is about addition, production learning can also be used to transform general production rules into task-specific rules, and to incorporate previous experiences into rules. The utility problem is handled in two ways: by constraining the size of the rules, rules with many conditions are impossible. Rules that are learned are introduced only gradually in the system, ensuring a relatively slow but safe proceduralization.

Summary

Rule learning is an fundamental issue in both machine learning and cognitive modelling. This article has focussed on incremental learning mechanisms, in which a current hypothesis is continuously updated on the basis of examples, involving both processes of generalization and specialization. Learning can be with or without domain or background knowledge. If no background knowledge is provided, in algorithms like single-hypothesis learning and version-space learning, learning is purely inductive. Explanation-based learning on the other hand deduces its knowledge from the domain theory, guided by examples.

Learning in cognitive modelling often employs a mixture of inductive and deductive methods, as background knowledge is often available but almost never complete. The most common method employed is to instantiate general problem solving methods with specific knowledge and examples, producing task-specific rules.

References

Anderson, J.R. (1983). *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.

Anderson, J.R. (1987). Skill Acquisition: Compilation of Weak-Method Problem Solutions. *Psychological Review, 94*, 192-210.

Anderson, J.R., & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Erlbaum.

DeJong, G. (1981). Generalizations based on explanations. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 67-70).

Minton, S. (1988). *Learning search control knowledge: An explanation-based approach*. Boston, MA: Kluwer.

Mitchell, T.M. (1977). Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on AI* (pp. 305-310). Cambridge, MA: MIT Press.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Newell, A., & Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1-55). Hillsdale, NJ: Erlbaum.

Winston, P.H. (1970). *Learning structural descriptions from examples*. Ph.D. dissertation. MIT Technical Report AI-TR-231.

Bibliography

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review, 89*, 369-406.

Mitchell, T.M. (1997). *Machine Learning*. New York: McGraw-Hill.