**Chapter 7: Learning**

**What is Learning?**

Learning is an important area in AI, perhaps more so than planning.

* Problems are hard -- harder than planning.
* Recognised Solutions are not as common as planning.
* A goal of AI is to enable computers that can be taught rather than programmed.

*Learning* is a an area of AI that focusses on processes of self-improvement.

Information processes that improve their performance or enlarge their knowledge bases are said to *learn*.

*Why is it hard?*

* Intelligence implies that an organism or machine must be able to adapt to new situations.
* It must be able to learn to do new things.
* This requires knowledge acquisition, inference, updating/refinement of knowledge base, acquisition of heuristics, applying faster searches, *etc.*

### How can we learn?

Many approaches have been taken to attempt to provide a machine with learning capabilities. This is because learning tasks cover a wide range of phenomena.

Listed below are a few examples of how one may learn. We will look at these in detail shortly

**Skill refinement**

-- one can learn by practicing, e.g playing the piano.

**Knowledge acquisition**

-- one can learn by experience and by storing the experience in a knowledge base. One basic example of this type is rote learning.

**Taking advice**

-- Similar to rote learning although the knowledge that is input may need to be transformed (or operationalised) in order to be used effectively.

**Problem Solving**

-- if we solve a problem one may learn from this experience. The next time we see a similar problem we can solve it more efficiently. This does not usually involve gathering new knowledge but may involve reorganisation of data or remembering how to achieve to solution.

**Induction**

-- One can learn from examples. Humans often classify things in the world without knowing explicit rules. Usually involves a teacher or trainer to aid the classification.

**Discovery**

-- Here one learns knowledge without the aid of a teacher.

**Analogy**

-- If a system can recognize similarities in information already stored then it may be able to transfer some knowledge to improve to solution of the task in hand.

**Rote Learning**

Rote Learning is basically *memorisation*.

* Saving knowledge so it can be used again.
* Retrieval is the only problem.
* No repeated computation, inference or query is necessary.

A simple example of rote learning is *caching*

* Store computed values (or large piece of data)
* Recall this information when required by computation.
* Significant time savings can be achieved.
* Many AI programs (as well as more general ones) have used caching very effectively.

Memorisation is a key necessity for learning:

* It is a basic necessity for any intelligent program -- is it a separate learning process?
* Memorisation can be a complex subject -- how best to store knowledge?

Samuel's Checkers program employed rote learning (it also used parameter adjustment)

* A minimax search was used to explore the game tree.
* Time constraints do not permit complete searches.
* It *records* board positions and scores at search ends.
* Now if the same board position arises later in the game the stored value can be recalled and the end effect is that more deeper searched have occurred.

Rote learning is basically a simple process. However it does illustrate some issues that are relevant to more complex learning issues.

**Organisation**

-- access of the stored value must be faster than it would be to recomputed

. Methods such as hashing, indexing and sorting can be employed to enable this.

*E.g* Samuel's program indexed board positions by noting the number of pieces.

**Generalisation**

-- The number of potentially stored objects can be very large. We may need to generalise some information to make the problem manageable.

*E.g* Samuel's program stored game positions only for white to move. Also rotations along diagonals are combined.

**Stability of the Environment**

-- Rote learning is not very effective in a rapidly changing environment. If the environment does change then we must detect and record exactly what has changed -- *the frame problem*.

**Learning by Taking Advice**

The idea of advice taking in AI based learning was proposed as early as 1958 (McCarthy). However very few attempts were made in creating such systems until the late 1970s. Expert systems providing a major impetus in this area.

There are two basic approaches to advice taking:

* Take high level, abstract advice and convert it into rules that can guide performance elements of the system. *Automate all aspects of advice taking*
* *Develop sophisticated tools* such as knowledge base editors and debugging. These are used to aid an expert to translate his expertise into detailed rules. Here the expert is an *integral* part of the learning system. Such tools are important in *expert systems* area of AI.

### Automated Advice Taking

The following steps summarise this method:

**Request**

-- This can be simple question asking about general advice or more complicated by identifying shortcomings in the knowledge base and asking for a remedy.

**Interpret**

-- Translate the advice into an internal representation.

**Operationalise**

-- Translated advice may still not be usable so this stage seeks to provide a representation that can be used by the performance element.

**Integrate**

-- When knowledge is added to the knowledge base care must be taken so that bad side-effects are avoided.

E.g. Introduction of redundancy and contradictions.

**Evaluate**

-- The system must assess the new knowledge for errors, contradictions etc.

The steps can be iterated.

### Knowledge Base Maintenance

Instead of automating the five steps above, many researchers have instead assembled tools that aid the development and maintenance of the knowledge base.

Many have concentrated on:

* Providing intelligent editors and flexible representation languages for integrating new knowledge.
* Providing debugging tools for evaluating, finding contradictions and redundancy in the existing knowledge base.

EMYCIN is an example of such a system.

### Example Learning System - FOO

**Learning the game of hearts**

FOO (First Operational Operationaliser) tries to convert high level advice (principles, problems, methods) into effective executable (LISP) procedures.

Hearts:

* Game played as a series of tricks.
* One player - who has the lead - plays a card.
* Other players follow in turn and play a card.
  + The player must follow suit.
  + If he cannot he play any of his cards.
* The player who plays the highest value card wins the trick and the lead.
* The winning player takes the cards played in the trick.
* The aim is to avoid taking points. Each heart counts as one point the queen of spades is worth 13 points.
* The winner is the person that after all tricks have been played has the lowest points score.

Hearts is a game of partial information with no known algorithm for winning.

Although the possible situations are numerous general advice can be given such as:

* Avoid taking points.
* Do not lead a high card in suit in which an opponent is void.
* If an opponent has the queen of spades try to flush it.

In order to receive advice a human must convert into a FOO representation (LISP clause)

(avoid (take-points me) (trick))

FOO operationalises the advice by translating it into expressions it can use in the game. It can UNFOLD avoid and then trick to give:

(achieve (not (during

(scenario

(each p1 (players) (play-card p1))

(take-trick (trick-winner)))

(take-points me))))

However the advice is still not operational since it depends on the outcome of trick which is generally not known. Therefore FOO uses case analysis (on the during expression) to determine which steps could case one to take points. Step 1 is ruled out and step 2's take-points is UNFOLDED:

(achieve (not (exists c1 (cards-played)

(exists c2 (point-cards)

(during (take (trick-winner) c1)

(take me c2))))))

FOO now has to decide: Under what conditions does (take me c2) occur during (take (trick-winner) c1).

A technique, called partial matching, hypothesises that points will be taken if me = trick-winner and c2 = c1. We can reduce our expression to:

(achieve (not (and (have-points(card-played))

(= (trick-winner) me ))))

This not quite enough a this means Do not win trick that has points. We do not know who the trick-winner is, also we have not said anything about how to play in a trick that has point led in the suit. After a few more steps to achieve this FOO comes up with:

(achieve (>= (and (in-suit-led(card-of me))

(possible (trick-has-points)))

(low(card-of me)))

FOO had an initial knowledge base that was made up of:

* basic domain concepts such as trick, hand, deck suits, avoid, win etc.
* Rules and behavioural constraints -- general rules of the game.
* Heuristics as to how to UNFOLD.

FOO has 2 basic shortcomings:

* It lacks a control structure that could apply operationalisation automatically.
* It is specific to hearts and similar tasks.

## Learning by Problem Solving

There are three basic methods in which a system can learn from its own experiences.

### Learning by Parameter Adjustment

Many programs rely on an evaluation procedure to summarise the state of search etc. Game playing programs provide many examples of this.

However, many programs have a static evaluation function.

In learning a slight modification of the formulation of the evaluation of the problem is required.

Here the problem has an evaluation function that is represented as a polynomial of the form such as:

displaymath2280

The *t* terms a values of features and the *c* terms are weights.

In designing programs it is often difficult to decide on the exact value to give each weight initially.

So the basic idea of idea of parameter adjustment is to:

* Start with some estimate of the correct weight settings.
* Modify the weight in the program on the basis of accumulated experiences.
* Features that appear to be good predictors will have their weights increased and bad ones will be decreased.

Samuel's Checkers programs employed 16 such features at any one time chosen from a pool of 38.

### Learning by Macro Operators

The basic idea here is similar to Rote Learning:

Avoid expensive recomputation

Macro-operators can be used to group a whole series of actions into one.

For example: Making dinner can be described a lay the table, cook dinner, serve dinner. We could treat laying the table as on action even though it involves a sequence of actions.

The STRIPS problem-solving employed macro-operators in it's learning phase.

Consider a blocks world example in which ON(C,B) and ON(A,TABLE) are true.

STRIPS can achieve ON(A,B) in four steps:

UNSTACK(C,B), PUTDOWN(C), PICKUP(A), STACK(A,B)

STRIPS now builds a macro-operator MACROP with preconditions ON(C,B), ON(A,TABLE), postconditions ON(A,B), ON(C,TABLE) and the four steps as its body.

MACROP can now be used in future operation.

But it is not very general. The above can be easily generalised with variables used in place of the blocks.

However generalisation is not always that easy