**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 recognise 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 which will be discussed shortly).

* 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 recompute it. 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*.

### Store v Compute

Rote Learning must not decrease the efficiency of the system.

We be must able to decide whether it is worth storing the value in the first place.

Consider the case of multiplication -- it is quicker to recompute the product of two numbers rather than store a large multiplication table.

How can we decide?

**Cost-benefit analysis**

-- Decide when the information is first available whether it should be stored. An analysis could weigh up amount of storage required, cost of computation, likelihood of recall.

**Selective forgetting**

-- here we allow the information to be stored initially and decide later if we retain it. Clearly the frequency of reuse is a good measure. We could tag an object with its time of last use. If the cache memory is full and we wish to add a new item we remove the least recently used object. Variations could include some form of cost-benefit analysis to decide if the object should be removed.

**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

### 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.

### 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

### Learning by Chunking

Chunking involves similar ideas to Macro Operators and originates from psychological ideas on memory and problem solving.

The computational basis is in production systems (studied earlier).

SOAR is a system that use production rules to represent its knowledge. It also employs chunking to learn from experience.

**Basic Outline of SOAR's Method**

* SOAR solves problems it fires productions these are stored in long term memory.
* Some firings turn out to be more useful than others.
* When SOAR detects are useful sequence of firings, it creates chunks.
* A chunk is essentially a large production that does the work of an entire sequence of smaller ones.
* Chunks may be generalised before storing.

**Inductive Learning**

This involves the process of *learning by example* -- where a system tries to induce a general rule from a set of observed instances.

This involves classification -- assigning, to a particular input, the name of a class to which it belongs. Classification is important to many problem solving tasks.

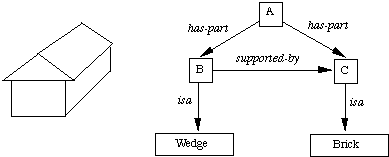
A learning system has to be capable of evolving its own class descriptions:

* Initial class definitions may not be adequate.
* The world may not be well understood or rapidly changing.

The task of constructing class definitions is called *induction* or *concept learning*

### A Blocks World Learning Example -- Winston (1975)

* The goal is to construct representation of the definitions of concepts in this domain.
* Concepts such a house - brick (rectangular block) with a wedge (triangular block) suitably placed on top of it, tent - 2 wedges touching side by side, or an arch - two non-touching bricks supporting a third wedge or brick, were learned.
* The idea of near miss objects -- similar to actual instances was introduced.
* Input was a line drawing of a blocks world structure.
* Input processed (see VISION Sections later) to produce a semantic net representation of the structural description of the object (Fig. [27](http://www.cs.cf.ac.uk/Dave/AI2/node145.html#fighousenet))

**Fig.**[**27**](http://www.cs.cf.ac.uk/Dave/AI2/node145.html#fighousenet)**House object and semantic net**

* Links in network include left-of, right-of, does-not-marry, supported-by, has-part, and isa.
* The marry relation is important -- two objects with a common touching edge are said to marry. Marrying is assumed unless does-not-marry stated.

There are three basic steps to the problem of concept formulation:

1. Select one know instance of the concept. Call this the concept definition.
2. Examine definitions of other known instance of the concept. Generalise the definition to include them.
3. Examine descriptions of near misses. Restrict the definition to exclude these.

Both steps 2 and 3 rely on comparison and both similarities and differences need to be identified.

### Version Spaces

Structural concept learning systems are not without their problems.

The biggest problem is that the teacher must guide the system through carefully chosen sequences of examples.

In Winston's program the order of the process is important since new links are added as and when now knowledge is gathered.

The concept of version spaces aims is insensitive to order of the example presented.

To do this instead of evolving a single concept description a set of possible descriptions are maintained. As new examples are presented the set evolves as a process of new instances and near misses.

We will assume that each slot in a version space description is made up of a set of predicates that do not negate other predicates in the set -- positive literals.

Indeed we can represent a description as a frame bases representation with several slots or indeed use a more general representation. For the sake of simplifying the discussion we will keep to simple representations.

If we keep to the above definition the Mitchell's candidate elimination algorithm is the best known algorithm.

Let us look at an example where we are presented with a number of playing cards and we need to learn if the card is odd and black.

We already know things like red, black, spade, club, even card, odd card etc.

So the tex2html_wrap_inline8374 is red card, an even card and a heart.

This illustrates on of the keys to the version space method specificity:

* Conjunctive concepts in the domain can be partially ordered by specificity.
* In this Cards example the concept black is less specific than odd black or spade.
* odd black and spade are incomparable since neither is more (or less) specific.
* Black is more specific than any card, any 8 or any odd card

The training set consist of a collection of cards and for each we are told whether or not it is in the target set -- odd black

The training set is dealt with incrementally and a list of most and least specific concepts consistent with training instances are maintained.

Let us see how can learn from a sample input set:

* Initially the most specific concept consistent with the data is the empty set. The least specific concept is the set of all cards.
* Let the tex2html_wrap_inline8376 be the first card in the sample set. We are told that this is odd black.
* So the most specific concept is tex2html_wrap_inline8376 alone the least is still all our cards.
* Next card tex2html_wrap_inline8380: we need to modify our most specific concept to indicate the generalisation of the set something like ``odd and black cards''. Least remains unchanged.
* Next card tex2html_wrap_inline8382: Now we can modify the least specific set to exclude the tex2html_wrap_inline8382. As more exclusion are added we will generalise this to all black cards and all odd cards.
* NOTE that negative instances cause least specific concepts to become more specific and positive instances similarly affect the most specific.
* If the two sets become the same set then the result is guaranteed and the target concept is met.

**The Candidate Elimination Algorithm**

Let us now formally describe the algorithm.

Let *G* be the set of most general concepts. Let *S* be the set of most specific concepts.

Assume: We have a common representation language and we a given a set of negative and positive training examples.

Aim: A concept description that is consistent with all the positive and none of the negative examples.

Algorithm:

* Initialise *G* to contain one element -- the null description, all features are variables.
* Initialise *S* to contain one element the first positive example.
* **Repeat**
  + Input the next training example
  + If a positive example -- first remove from *G* any descriptions that do not cover the example. Then update *S* to contain the most specific set of descriptions in the version space that cover the example and the current element set of *S*. I.e. Generalise the elements of *S* as little as possible so that they cover the new training example.
  + If a negative example -- first remove from *S* any descriptions that cover the example. Then update *G* to contain the most general set of descriptions in the version space that do not cover the example. I.e. Specialise the elements of *S* as little as possible so that negative examples are no longer covered in *G*'s elements.

**until** *S* and *G* are both singleton sets.

* If *S* and *G* are identical output their value.
* *S* and *G* are different then training sets were inconsistent.

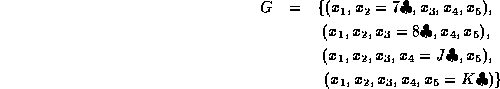
Let us now look at the problem of learning the concept of a flush in poker where all five cards are of the same suit.

Let the first example be positive: tex2html_wrap_inline8422

Then we set   
eqnarray2378

No the second example is negative: tex2html_wrap_inline8424

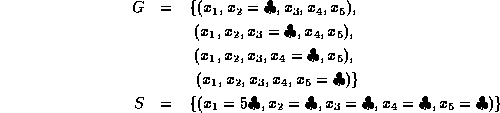
We must specialise *G* (only to current set):



*S* is unaffected.

Our third example is positive: tex2html_wrap_inline8430

Firstly remove inconsistencies from *G* and then generalise *S*:



Our fourth example is also positive: tex2html_wrap_inline8436

Once more remove inconsistencies from *G* and then generalise *S*:

eqnarray2434

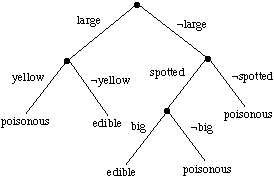
* We can continue generalising and specialising
* We have taken a few big jumps in the flow of specialising/generalising in this example. Many more training steps usually required to reach this conclusion.
* It might be hard to spot trend of same suit etc.

### Decision Trees

Quinlan in his ID3 system (986) introduced the idea of decision trees.

ID3 is a program that can build trees automatically from given positive and negative instances.

Basically each leaf of a decision tree asserts a positive or negative concept. To classify a particular input we start at the top and follow assertions down until we reach an answer (Fig [28](http://www.cs.cf.ac.uk/Dave/AI2/node147.html#figdectree))



**Fig.**[**28**](http://www.cs.cf.ac.uk/Dave/AI2/node147.html#figdectree)**Edible Mushroom decision tree**

**Building decision trees**

* ID3 uses an iterative method.
* Simple trees preferred as more accurate classification is afforded.
* A random choice of samples from training set chosen for initial assembly of tree -- the window subset.
* Other training examples used to test tree.
* If all examples classified correctly stop.
* Otherwise add a number of training examples to window and start again.

**Adding new nodes**

When assembling the tree we need to choose when to add a new node:

* Some attributes will yield more information than others.
* Adding a new node might be useless in the overall classification process.
* Sometimes attributes will separate training instances into subsets whose members share a common label. Here branching can be terminates and a leaf node assigned for the whole subset.

Decision tree advantages:

* Quicker than version spaces when concept space is large.
* Disjunction easier.

Disadvantages:

* Representation not natural to humans -- a decision tree may find it hard to explain its classification.
* **Explanation Based Learning (EBL)**
* Humans appear to learn quite a lot from one example.
* Basic idea: Use results from one examples problem solving effort next time around.
* An EBL accepts 4 kinds of input:
* **A training example**
* -- what the learning *sees* in the world.
* **A goal concept**
* -- a high level description of what the program is supposed to learn.
* **A operational criterion**
* -- a description of which concepts are usable.
* **A domain theory**
* -- a set of rules that describe relationships between objects and actions in a domain.
* From this EBL computes a generalisation of the training example that is sufficient not only to describe the goal concept but also satisfies the operational criterion.
* This has two steps:
* **Explanation**
* -- the domain theory is used to prune away all unimportant aspects of the training example with respect to the goal concept.
* **Generalisation**
* -- the explanation is generalised as far possible while still describing the goal concept.

### EBL example

Goal: To get to Brecon -- a picturesque welsh market town famous for its mountains (beacons) and its Jazz festival.

The training data is:

near(Cardiff, Brecon),

airport(Cardiff)

The Domain Knowledge is:

    near(x,y) tex2html_wrap_inline7782 holds(loc(x),s) tex2html_wrap_inline7156 holds(loc(y), result(drive(x,y),s))

   airport(z) tex2html_wrap_inline7156 loc(z), result(fly(z),s)))

In this case operational criterion is: We must express concept definition in pure description language syntax.

Our goal can expressed as follows:

holds(loc(Brecon),s) -- find some situation *s* for this holds.

We can prove this holds with *s* defined by:

result(drive(Cardiff,Brecon),

result(fly(Cardiff), s')))

We can fly to Cardiff and then drive to Brecon.

If we analyse the proof (say with an ATMS). We can learn a few general rules from it.

Since Brecon appears in query and binding we could abstract it to give:

holds(loc(x),drive(Cardiff,x),

result(fly(Cardiff), s')))

but this not quite right - we cannot get everywhere by flying to Cardiff.

Since Brecon appears in the database when we abstract things we must explicitly record the use of the fact:

near(Cardiff,x) tex2html_wrap_inline7156 holds(loc(x),drive(Cardiff,x), result(fly(Cardiff), s')))

This states if x is near Cardiff we can get to it by flying to Cardiff and then driving. We have learnt this general rule.

We could also abstract out Cardiff instead of Brecon to get:

near(Brecon,x) tex2html_wrap_inline7782 airport(x) tex2html_wrap_inline7156 holds(loc(Brecon), result(drive(x,Brecon),   
result(fly(x),s')))

This states we can get top Brecon by flying to another nearby airport and driving from there.

We could add airport(Swansea) and get an alternative means of travel plan.

Finally we could actually abstract out both Brecon and Cardiff to get a general plan:

near(x,y) tex2html_wrap_inline7782 airport(y) tex2html_wrap_inline7156 holds(loc(y), result(drive(x,y),result(fly(x),s')))

## Analogy

Analogy involves a complicated mapping between what might appear to be two dissimilar concepts.

   Bill is built like a large outdoor brick lavatory.

   He was like putty in her hands

Humans quickly recognise the abstractions involved and understand the meaning.

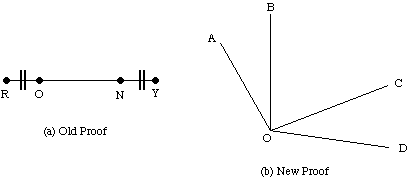
There are two methods of analogical problem methods studied in AI.

### Transformational Analogy

Look for a similar solution and copy it to the new situation making suitable substitutions where appropriate.

E.g. Geometry.

If you know about lengths of line segments and a proof that certain lines are equal (Fig. [29](http://www.cs.cf.ac.uk/Dave/AI2/node154.html#figline)) then we can make similar assertions about angles.



**Fig.**[**29**](http://www.cs.cf.ac.uk/Dave/AI2/node154.html#figline)**Transformational Analogy Example**

* We know that lines *RO* = *NY* and angles *AOB* = *COD*
* We have seen that *RO* + *ON* = *ON* + *NY* - additive rule.
* So we can say that angles *AOB* + *BOC* = *BOC* + *COD*
* So by a transitive rule line *RN* = *OY*
* So similarly angle *AOC* = *BOD*

Carbonell (1983) describes a T-space method to transform old solutions into new ones.

* Whole solutions are viewed as states in a problem space -- the T-space.
* T-operators prescribe methods of transforming existing solution states into new ones.
* Reasoning by analogy becomes a search in T-space -- means-end analysis.

### Derivational Analogy

Transformational analogy does not look at how the problem was solved -- it only looks at the final solution.

The history of the problem solution - the steps involved - are often relevant.

Carbonell (1986) showed that derivational analogy is a necessary component in the transfer of skills in complex domains:

* In translating Pascal code to LISP -- line by line translation is no use. You will have to reuse the major structural and control decisions.
* One way to do this is to replay a previous derivation and modify it when necessary.
* If initial steps and assumptions are still valid copy them across.
* Otherwise alternatives need to found -- best first search fashion.