MODULE 5

****

SYLLABUS

Module 5 (6Hrs 15%Marks)

Learning Concepts: Version space search. Back propagation learning. Social and emergent models of learning-genetic algorithm, classifier systems and genetic programming

****

Learning

One of the most often heard criticism of AI is that machine cannot be called intelligent until they are able to learn new things and to adapt to new situations, rather than simply doing as they are told to do. There can be question that the ability to adapt to new surroundings and to solve new problems is an important characteristics of intelligent entities.

Learning covers a wide range of phenomena. At one end of the spectrum is skill refinement. People get better at many task simply by practicing. At the other end of the specturum lies knowledge acquisition. Knowledge is generally through experience.

Knowledge acquisition activities:

1. Role learning-Simple storing of computed information
2. Advice from others
3. Problem solving experience
4. Learning from examples

Rote Learning

When a computer stores a piece of data, it is performing primitive form of learning . After all this act of storage presumably allow the program to perform better in future.

In **Data Caching**, we store computed values so that we do not have to recompute them later. When computation is more expensive than recall, this strategy can save a significant amount of time. Caching has been used in AI programs to produce some surprising performance improvements. Such caching is known as rote learning.

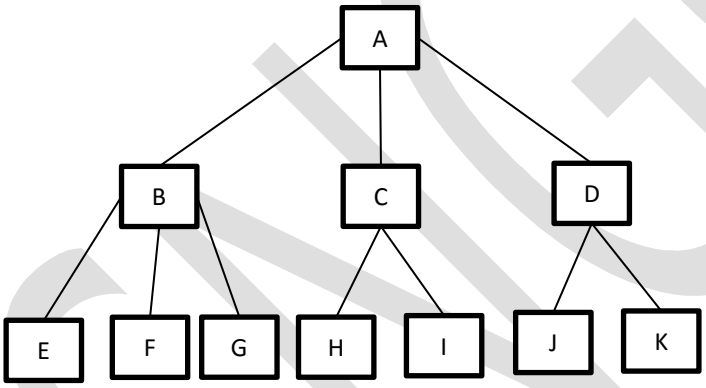
Samuel's Checker Program

This program learned to play checker well enough to beat its creator. It exploited two kind of learning:

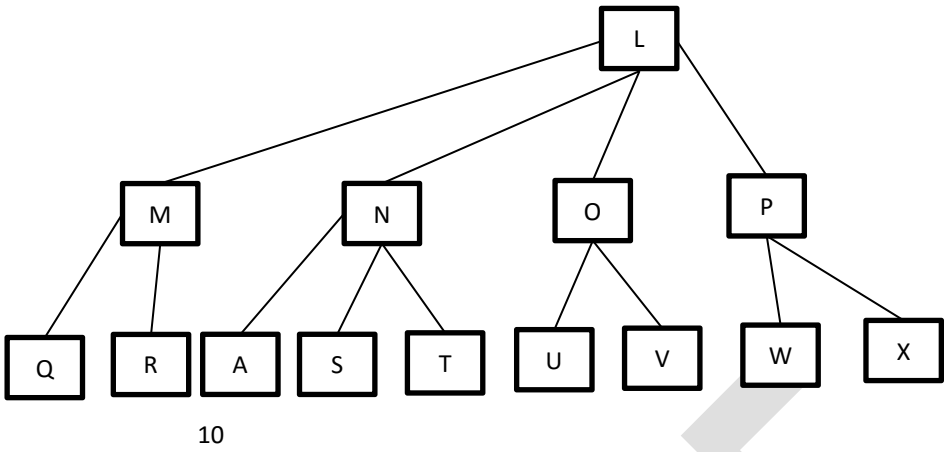
1. Rote learning and

2. Parameter adjustments

Samuel's program used the minmax procedure to explore checker game trees. As with all such programs time constraints permitted to search only a few levels in the tree. When it could search no deeper, it applied the static evaluation function to board positions and use that score to continue its search of game tree. When it finished searching the tree and propagating the values backward, it had a score for the position represented by the root of the tree. It could choose the best move and make it. But it also recorded the board position at the root of the tree and the backed up score that had just been computed for it. But it also recorded the board position at the root of the tree and backed up score that had just been computed for it.



Scored value of A:10



Storing

Rote learning of this sort is very simple. It does not appear to involve any sophisticated problem solving capabilities. But even it show some capabilities which are.

- **Organized storage of information:** It is faster to use a stored value than it would be to re-compute it, there must be a way to access the stored values quickly. Eg, indexing positions.
- **Generalization:** the number of distinct objects that might potentially be stored can be very large. To keep the number of stored objects down to a manageable level, some kind of generalization is necessary.

Learning by taking advice

A computer can do very little without a program for it to run. When a programmer writes a series of instructions into a computer, a fundamental kind of learning taking place.

The programmer is sort of a teacher. Computer is sort of a student. After being programmed the computer is now able to do something it previously could not. Executing programs may not be such a simple matter. Suppose the program is written in high level language like LISP. Some interpreter or compiler must intervene to change the teacher's instruction into code that machine can execute directly.

Learning the game of hearts

FOO(First Operational Operationaliser) tries to convert high level advice into effective executable procedure

Hearts

- It is a game played as series of tricks.
- One player who has the lead plays a card
- Other players follow in turn and plays a card
  - The player must follow suit
  - If he cannot he play any of his card
- The player who plays the highest value card wins the trick and the lead.
- Winning player takes the card played in the trick
- The aim is to avoid taking points. Each heart counts as one point. The queen of spade is worth 13 points.
- The winner is the person that after all tricks have been played and 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 higher card in suit in which an opponent is void
- If an opponent has the queen of spade try to flush it.

A human user first translate the advice from english to a representation that FOO can understand.

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

```
(avoid(take_points me)(trick))
```

- Avoid taking points.

FOO opertionalises the advice by translating it into expression it can use in the game. One strategy FOO can follow is to UNFOLD an expression by replacing some term by its definition.

By unfolding the definition of avoid FOO comes up with

```
(acheive(not(during(trick)(takepoints me))))
```

FOO considers the advice to apply to the player called “me”.

Next, FOO UNFOLDS the definition of trick

```
(acheive (not (during
         ( scenario
            (each pl(player) (play_card pl))        --->(1)
            ( take-trick (trick-winner) ) )     --->(2)
         (take-point me) ) ) )
```

In other words the player should avoid taking points during the scenario consisting of

(1) player playing cards and

(2) One player taking the trick

FOO then uses case analysis to determine which step could cause one to take points. It rules out step (1) on the basis that it knows of no intersection of the concept take-poinnt and play-card. But step (2) could affect taking points, so FOO UNFOLDS the definition of take points.

```
(acheive (not (there-exists c1(card-played)
         (there-exists c2(point-cards)
            (during (take (trick-winner) c1)
               (take me c2))))))
```

This advice says that player should avoid taking point-cards during the process of trick-winner taking the trick.

The question for FOO now is under what condition does (take me c2) occur during

```
(take (trick-winner) c1)?
```

By using a technique called partial match, FOO hypothesized that points will be taken if

```
me=trick_winner and c2=c1
```

It transforms the advice into:

```
(acheive (not (and (have-points(card-played))
         (=(trick-winner) me))))
```

This means “Do not win a trick that has points”.

```
(acheive (>=(and (in-suit-led(card-of me))
                 (possible(trick-has-points)))
                  (low (card-of me)))
```

When playing a card that is the same suit as the card that was played first, if the trick possibly contains points, then play a low card. At last FOO has translated the rather vague advice “avoid taking points” into a specific useable heuristic. FOO is able to play a better game of hearts after receiving this advice.

A human can watch FOO play, detect new mistakes correct them through yet more advice, such as “Play high cards when it is safe to do so”

The ability to operationalize knowledge is critical for systems that learn from a teacher’s advice. It is also an important component of explanation-based learning.

Learning in Problem-Solving

Can a program get better without the aid of the programmer? It can by generalizing from its own experiences.

1. Learning by Parameter Adjustment

Many programs rely on an evaluation procedure that combines information from several sources into a single summary statistics.

Game playing programs do this in their static evaluation functions in which a variety of factors, such as piece advantage and mobility, are combined into a single score reflecting the desirability of a particular board position. Pattern classification programs often combine several features to determine the correct category into which a given stimulus should be placed. In designing such programs, it is often difficult to know a priori how much weight should be attached to each feature being used. One way of finding the correct weight is to begin with some estimate of the correct setting and then to let the program modify the settings on the basis of its experience.

Features that appear to be good predictors of overall success will have their weights increased, while those that do not will have their weights decreased.

Eg. Samuel's checkers program exploited this kind of learning in addition to the rote learning. As its static evaluation function, the program used a polynomial of the form

$C_1t_1 + C_2t_2 + \ldots + C_{16}t_{16}$

The term t are the values of the sixteen features that contribute to the evaluation. The term C are coefficients(weights) that are attached to each of these values. As learning progresses, the C values will change.

The most important question in the design of a learning program based on parameter adjustment is:

1. When should the value of a coefficient be increased and when should it be decreased?
2. How much the value should be changed?

The simple answer to the first question is that coefficients of term that predicted the final outcome accurately should be increased, while the coefficients of poor predictors should be decreased.

The program does not get any concrete feedback from individual moves. It does not find out for sure until the end of the game whether it has won. But many moves have contributed to that final outcome. Even if the program wins, it may have made some bad moves along the way. The problem of appropriately assigning responsibility to each of the step that led to a single outcome is known as the **credit assignment problem**.

Samuel's program exploits one technique, albeit imperfect for solving this problem. Assume that the initial values chosen for the coefficient are good enough that the total evaluation function produces values that are fairly reasonable measures of the correct score even if they are not as accurate as we hope to get them. Then this evaluation function can be used to provide feedback to itself. Move sequences that lead to positions with higher values can be considered good.

Because of the Limitation of this approach, however, Samuel's program did two other things:

- When the program was in learning mode, it played against another copy of itself. Only one of the copies altered its scoring function during the game the other reminder fixed. At the end of the game old one was retained. If however

this happened very many times then some drastic change was made to the function in an attempt to get the process going in a more profitable direction.

- Periodically one term in the scoring function was eliminated and replaced by another. This was possible because although the program used only sixteen features at any one time, it actually knew about 38.

This process of learning by successive modification to weights of terms in scoring function has many limitations:

- Mostly arising out of lack of exploitation of any knowledge about structure of the problem with which it is dealing and Logical relationship among components
- Learning is a variety of hill climbing, it suffer of hill climbing programs

2. Learning with Macro Operators

The idea is to avoid expensive recomputation. For eg, suppose you are faced with the problem of getting to the downtown postoffice. Your solution may involve

1. getting in your car
2. Starting it
3. Driving along a certain route

Substantial planning may go into choosing the appropriate route, but you need not plan about how to go about starting your car. You are free to choose START_CAR as an atomic action, even though it really consist of several actions:

1. sitting down
2. Adjusting the mirror
3. Inserting key
4. Turning the key

Sequence of actions that can be treated as a whole are called MACRO-OPERATORS. Macro-Operators were used in early problem solving system STRIPS.

In artificial intelligence, **STRIPS** (**St**anford **R**esearch **I**nstitute **P**roblem **S**olver) is an automated planner developed by Richard Fikes and Nils Nilsson in 1971 at SRI International.The same name was later used to refer to the formal language of the inputs to this planner. This language is the base for most of the languages for expressing automated planning problem instances in use today; such languages are commonly known as action languages. This article only describes the language, not the planner.

STRIPS has a learning component. After each problem solving episode, **the learning component takes the computed plan and stores** it away as macro-operator or MACROP.

A MACROP is just like a regular operator except it consist of a sequence of actions, not just a single one.

A MACROP ‘s precondition are the initial conditions of the problem just solved, and its postconditions correspond to the goal just achieved. In its simplest form the caching of previously computed plan is rote learning.

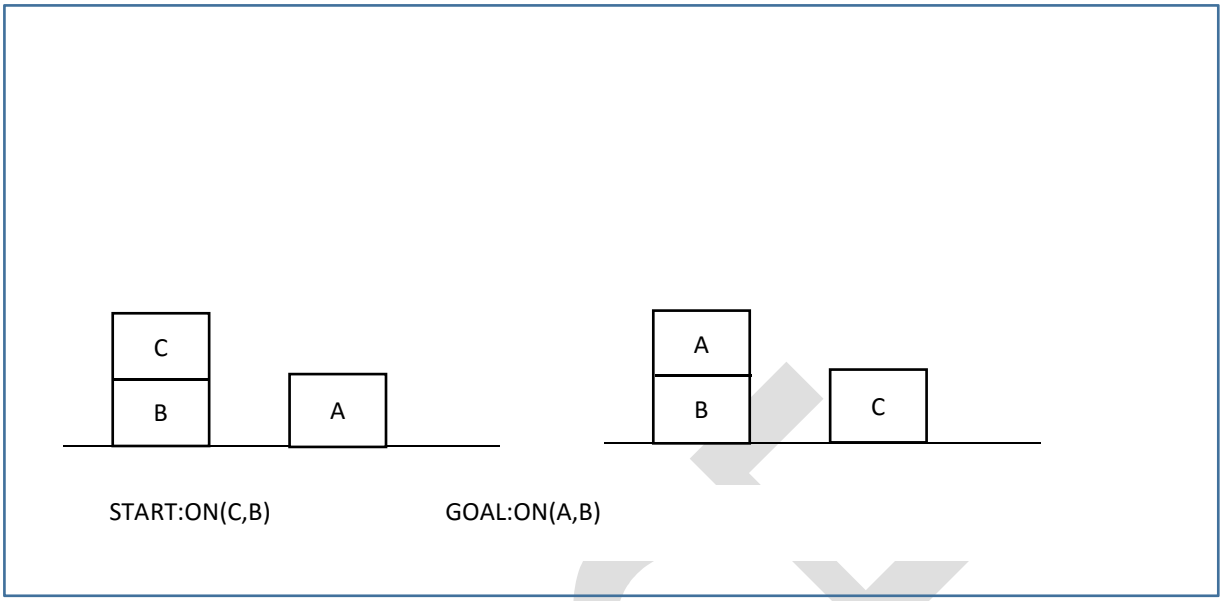
Example

There is a robot arm to manipulate blocks. The actions are

1. **UNSTACK(A,B):** pickup block A from its current position on block B. The arm must be empty and block A must have no blocks on top of it.
2. STACK(A,B): Place block A on block B. The arm must already be holding and the surface of must be clear.
3. PICKUP(A): Pick up block A from the table and hold it. The arm must be empty and there must be nothing on top of block A
4. PUTDOWN(A): Put block A down on the table. The arm must have been holding block A.

Result Predicates

1. ON(A,B): Block A is on Block B
2. ONTABLE(A): Block A on table
3. CLEAR(A): There is nothing on top of block A
4. HOLDING(A): The arm is holding block A
5. ARMEMPTY: The arm is holding nothing



Suppose we are given an initial blocks world situation in which ON(C,B) and ON(A, Table) are both true.

STRIPS can achieve the goal ON(A,B) by devising a plan with four steps:

1. UNSTACK(C,B)

2. PUTDOWN(C)

3. PICKUP(A)

4. STACK(A,B)

STRIPS now builds a MACROP with precondition ON(C, B), ON(A, Table) and postcondition ON(C, Table), ON(A, B)

The body of the MACROP consist of the four steps. In future planing, STRIPS is free to use this complex macro-operator just as it would use any other operator.

But rarely will STRIPS see the exact same problem again twice. New problem will differ from prvious problem twice. New problem will differ from previous problems. We would still like the problem solver to make efficient use of knowledge it gained from its previous experiences.

By generalizing MACROPs before storing them, STRIPS is able to accomplish this. The simplest idea for genaralization is to replace all of the contents in macro_operator by variables. Instead of storing the MACROP, STRIPS can generalize the plan to consist of the steps

1. UNSTACK(x1,x2) // C=x1, B=x2, A=x3

2. PUTDOWN(x1)

3. PICKUP(x3)

4. STACK(x3, x2)

This plan can then be stored with

- Preconditions: ON(x1,x2), ON(x3, Table) and
- Postconditions: ON(x1, Table), ON(x2, x3)

Such MACROP can now apply in a variety of situations.

Sometimes constants must retain their specific values. SUppose our domain included an operator called STACK_ON_B(x), with preconditions that both x and B be clear and with postcondition ON(x,B)

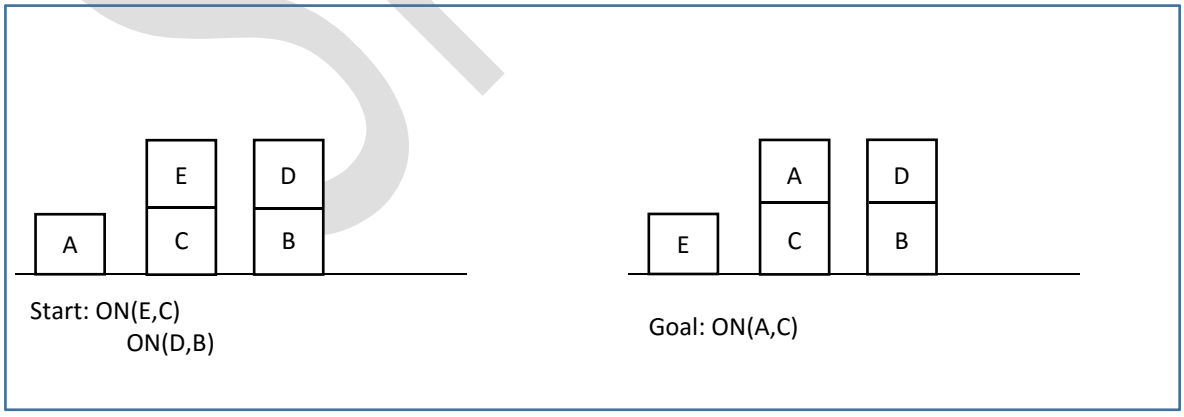
STRIPS might come up with the plan UNSTACK(C,B), PUTDOWN(C), STACK_ON_B(A). Let's generalize this plan and store it as a MACROP. The precondition becomes ON(x3,x2) and postcondition becomes ON(x1,x2) and plan itself becomes:

UNSTACK(x3,x2)

PUTDOWN(x3)

STACK_ON_B(x1)

Some different problem



The generalized MACROP we just stored seems well suited to solving this problem if we let x1=A , x2=C and x3=E. Its preconditions are staisfied, so we construct the plan UNSTACK(E,C)

PUTDOWN(E)

STACK_ON_B(A)

But this plan does not work. The problem is that the postcondition of MACROP is overgeneralized. This operation is only useful for stacking blocks onto B, which is not what we need in this new example. In this case, the difficulty will be discovered when the last step is attempted. Although we cleared C, which is where we wanted to put A, we failed to clear B, which is where we wanted to put A, we failed to clear B, which is were the MACROP is going to try to put it. Since B is not clear, STACK_ON_B cannot be executed. If B had happened to be clear, the MACROP would have executed to completion, but it would not have accomplished the start goal.

STRIPS uses a more complex generalization procedure. First all constants are replaced by variables. Then for each operator in the parameterized plan, STRIPS revaluates its preconditions.

More recent works on macro-operators appear in Korf. It turns out that the set of problems for which macro-operators are critical are exactly those problems with nonserializable subgoals will necessarily interface with the previous solution to anoher subgoal.

For eg, Consider 8-puzzle problem

Once aprogram has correctly placed the first four tiles it is difficult to place the fifth tile without disturbing the first four. Because disturbing previously solved subgoals is detected as a bad thing by heuristic scoring functions, it is strongly resisted. Hence we need domain specific knowledge.

We can learn the domain specific knowledge we need in the form of macro operators. Thus macro-operators can be viewed as a weak method for learning. In the 8-puzzle, we might have a macro- a complex, prestored sequence of operators for placing the fifth tile without disturbing any of the first and file externally.

Learning by Chunking

Chunking is a process similar in flavor to macro-operators. The idea of chuncking comes from psychological literature on memory and problem solving. Computational basis is production system.

SOAR system use of control knowledge. SOAR also exploit chunking so that its performance can increase with experience. The designer of SOAR hypothesis that chunking is a universal learning method, it account for all types of learning in intelligent systems.

SOAR is a general architecture for building intelligent systems. SOAR is based on a set of specific, cognitively motivated hypothesis about the structure of human problem solving.

In SOAR system

1. Long term memory is stored as a set of production or rules

2. Short term memory or working memory is a buffer that is affected by perception and serves as a storage area for facts deducted by rules in long term memory. Working memory is analogous to state description in problem solving

3. All problem solving activity takes place as state space traversal. There are several classes of problem solving activities including reasoning about which states to explore, wich rules to apply in a given situation and what effects those rules will have.

4. All intermediate and final results of problem solving are remembered for future references.

SOAR solves problems by firing productions which are stored in long term memory. Some of those firing turns out to be more useful than others. When SOAR detects a useful sequence of production firing, it creates a chunk which is esstentially a large production that does the work of an entire sequence of smaller ones. Chuncks are generalized before they are used.

In solving 8-puzzle for eg, SOAR learns how to place a given tile without permenantly disturbing the previously placed tiles. Given the way that the SONAR learns, several chunks may encode a single macro operator and chunck may prarticipate in a number of macro sequences. Chunk are generally applicable towards any goal state. This contrast with macro tables, which are structured towards searching a particular goal state from any initial state.

Chucking emphasis how learning can occur during problem solving while macro tables are usually built during preprocessing stage. As a result, SONAR is able to learn within trials as well as across trails. Chuncks learned during the initial stages of solving a problem are applicable in later stages of the same problem solving episoded. After a solution is found the chunks remains in memory ready for use in next problem.

The price that SOAR pays for this generality and flexibility is speed. Chuncking is inadequate for duplicating the contents of large directly computed macro-operator tables.

Learning From Example: Induction

Classification is the process of assigining to a particular input, the name of the class to which it belongs. Classification is an important component of many problem solving tasks. In its simplest form, it is presented as a straightforward recognition task.

Before classification can be done, **the classes it will use must be defined**. This can be done in a variety of ways, including:

1. Isolate a set of features that are relevant to the task domain. **Define each class by weighted sum of values of these features.** Each class is then defined by a **scoring function** that looks very similar to the scoring functions often used in other situations.

Functions that has form

$C_1t_1 + C_2t_2 + C_3t_3 + \ldots$

t: value of relavent parameter

c: weight

Negative weight can be used to indicate features whose presence usually constitutes negative evidence for a give class. For eg, if the task is weather prediction, the parameters can be such measure as rainfall and location of cold fronts. Different functions can be written to combine these parameters to predict sunny, cloudy, rainy or snowy weather.

2. Isolate a set of features that are relevant to the task domain. Define each class as a structure component of those features.

For eg; if the task is to identify animals, the body of each type of animal can be stored as a stucture with various features representing such things as color, length of neck and feathers.

The idea of producing a classification program that can evolve its own class definition is appealing. This task of constructing class definition is called concept learning or induction.

The technique used for this task must depend on the way that classes are decribed. If classes are decribed by scoring function then concept learning can be done using Parameter Adjustment. However if we want to define classes structurally some other technique for learning class definition is necessary, three such techniques are

1. Winston's Learning Program
2. Verson Spaces
3. Decision Trees

Winston's Learning Program

Winston descibes an early structural concept learning program. This program operated in a simple blocks world domain. Its goal was to construct representation of definition of concepts in the block domain.

For example it learned the concept House, Tent and Arch. The figure shows an example of a near miss for each concept. Near miss is an object that is not an instance of concept in the question but is very similar to the instances.

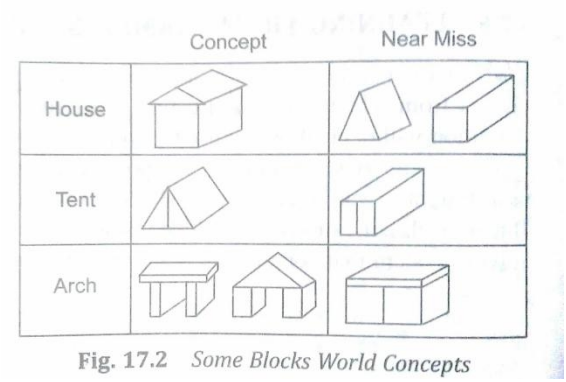
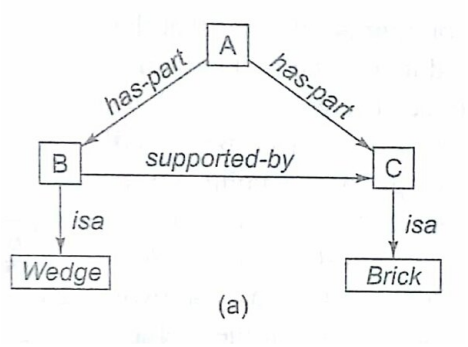


Fig. 17.2 *Some Blocks World Concepts*

A never miss is an object that is not an instance of the concept in question but that is similar to such instances.

The program started with a line of drawing of blocks world structure and constructs a sematic net representation of structural description of the objects. This structural descripion was then provided as input to learning program. The below figure shows the structural description for the House:



Node A represents the entire structure, which is composed of two parts Node B(a Wedge) and node C(a Brick).

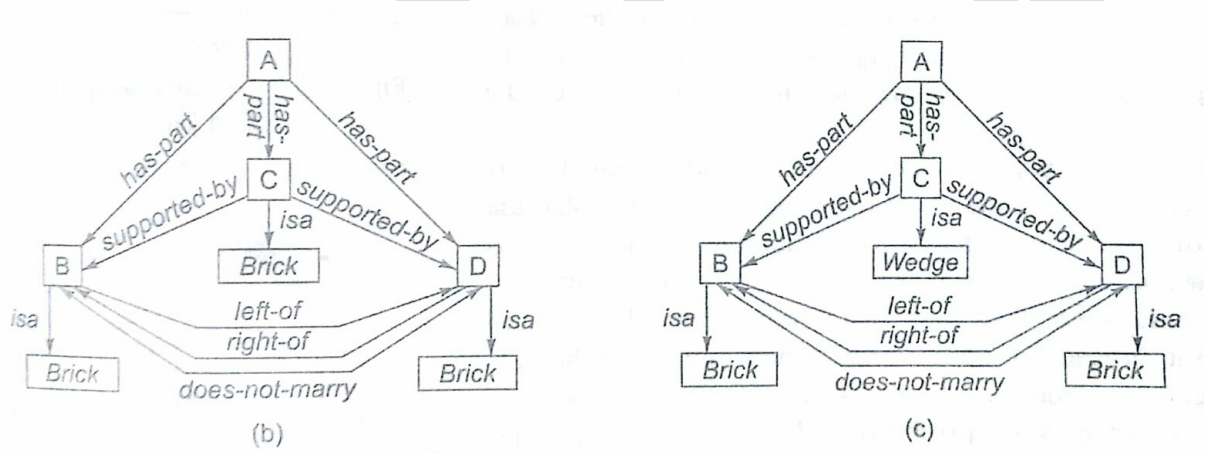


Fig. 17.3 *Structural Descriptions*

These are the structural definitions of Arch. These definitions are identical except the types of object on the top: one is a brick while other is a wedge. The two supporting objects are not only by left-of and right-of but also by a does-not-marry link. Which says that the two objects do not marry. Two objects marry if they have faces that touch and they have a common edge.

Winston's Program

1. Begin with a structural description of one known instance of the concept. Call that description the concept definition

2. Examine description of other known instances of the concept. Generalize the definition to include them

3. Examine descriptions of near miss of the concept. Restrict the definition to exclude these.

Version Space Search

Version space search illustrates the implementation of inductive learning as search through a concept space.

Generalization operators and concept space

Generalization and specialization are most common type of operations for defining a concept space. The primary generalization operation used in machine learning are:

1. Replace constants with variables

Color(ball,red)

Generalizes to

Color(X, red)

2. Dropping conditions from a conjuctive expression.

Shape(X, round) ^ size(x,small) ^ color(x,red)

Generalizes to

Shape(X, round) ^ color(X, red)

3. Adding a disjunct to an expression

Shape(X, round) ^ size(x,small) ^ color(x,red)

Generalize to

Shape(X, round) ^ size(X, small) ^ color(X, red) v color(X, blue)

Version Spaces

The goal is the same: to produce a description that is constant with all positive examples but no negative example in the training set. Version space work by

maintaining a set of possible descriptions and evolving that set as a new example and near misses are present.

Consider a frame representing an individual car

Car023

Orgin: Japan

Manufacturer: Honda

Color: Blue

Decade: 1970

Type : Economy

Now suppose that each slot may contain only the discrete values.

origin	∈	{*Japan, USA, Britain, Germany, Italy*}
manufacturer	∈	{*Honda, Toyota, Ford, Chrysler, Jaguar, BMW, Fiat*}
color	∈	{*Blue, Green, Red, White*}
decade	∈	{1950, 1960, 1970, 1980, 1990, 2000}
type	∈	{*Economy, Luxury, Sports*}

Fig. 17.8 *Representation Language for Cars*

The choice of features and value is called **bais** of learning system. By being embeded in a particular program and by using particular representation, every learning system is baised, because it learns some things more easily than others. In this example bais is fairly simple- eg, we can learn concepts that have to do with car manufacturers but not owners. In more complex system, the bais is less obivous. A clear statement of bias of a learning system is very important to its evaluation.

Concept descriptions as well as training example can be stated in terms of these slots and values. For example the concept “Japanese Economy Car” can be represented as

Orgin: Japan

Manufacturer: x1

Color: x2

Decade: x3

Type : Economy

The names x1, x2, x3 are variables. Their value is not relevant. Now the learning problem is: Given a representation language and given a positive and negative training example how can we produce a concept description that is consistent with all training examples?

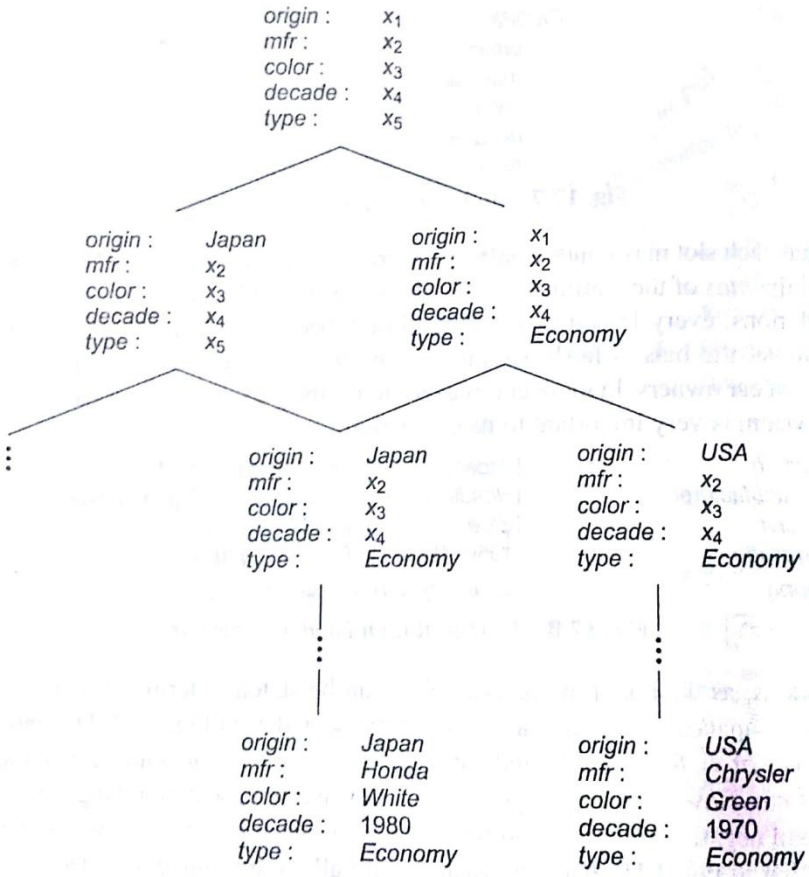


Fig. 17.10 *Partial Ordering of Concepts Specified by the Representation Language*

The representation language defines a partial ordering of the descriptions. A portion of that partial ordering is shown in above figure.

The entire partial ordering is called the concept space, it is as follows

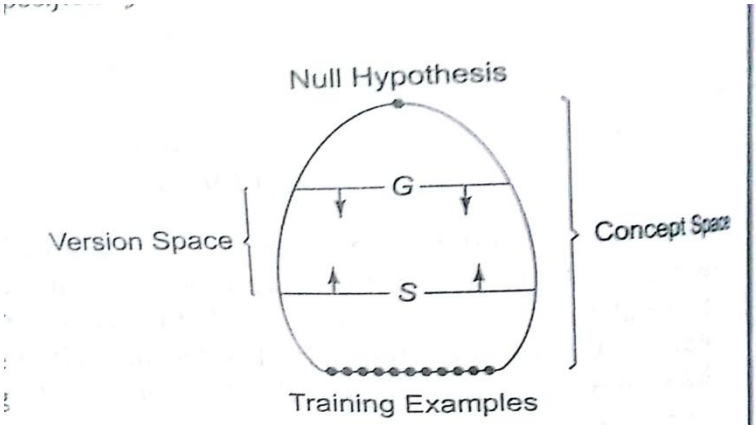


Fig. 17.11 *Concept and Version Spaces*

At the top of the concept space is the null description consisting only of variables and at bottom are all the possible training instances, which contain no variables. Before we receive any training examples, we know that the target concept lies somewhere in the concept space. For example, if every possible description is an instance of intended concept, then the null description is the concept definition since it matches everything. On the other hand if target concept includes only a single example then one of the description at bottom of the concept space is the desired concept definition.

The version space is simply a set of description so initial idea is to keep an explicit list of those description. Unfortunetly the number of descriptions in the context space is exponential in number of features and values. But Version Space has a consise representation. It consist of two subset of concept space:

1. Subset G contain the most general descriptions consistent with training example

2. Subset S contains most specific descriptions consistent with training examples.

Version space is the set of all descriptions that lie between some element of G and some element of S in partial order of concept space.

This representation of version space is not only effective for storage but also for modification. Intutively each time we receive a positive training example we want to make the S set more general. Negative training example serves to make the G more specific. If the S and G set coverage, our range of hypothesIs will narrow to a single concept description. The algorithm for narrowing the version space is called **candidate elemination algorithm**.

Candidate Elimination Algorithm

Given: A representation language and set of positive and negative examples expressed in that language

Compute: A concept decription that is consstent with all the positive examples and none of the negative examples

1. Initalize G to contain one element: the null description(all features are variables)
2. Initialize S to contain one element: the first positive example
3. Accept a new training example:

   If it is a positive example, first remove from G any descriptions that do not cover the example. Then update the S set to contain the most specific set of descriptions in the version space that cover the example and the current example of the S set.ie generalize the elements of S as little as possible so that they cover the new training example.

   If it is a negative example first remove from S any descriptions that cover the example. Then, update the G set to contain the most general set of descriptions in the version space that don not cover the example. Ie, specialize the element of G as little as possible so that negative example is no longer covered by any element of G

4. if S and G are both singleton sets, then if they are identical, output their value and halt. If they are both singleton sets but they are different then the training cases were inconsistent. Output this result and halt. Otherwise go to step 3.

In mathematics, a **singleton**, also known as a unit **set**, is a **set** with exactly one element. For example, the **set** {0} is a **singleton**.

Eg., Suppose we want to learn the concept of "Japanese Economy Car" from the examples

(1)

Orgin: Japan

Manufacturer: Honda

Color: Blue

Decade: 1980

Type : Economy

(+)

S and G both start out as singleton sets:

G <= null description

S <= first positive training example

G={(x1,x2,x3,x4,x5)}

S={(Japan, Honda,Blue,1980,Economy)}

Now we are ready to process the second example

(2)

Orgin: Japan

Manufacturer: Toyota

Color: Green

Decade: 1970

Type : Sports

(-)

The G set must be specialized in such a way that the negative example is no longer in version space. Specialization invloves replacing variables with constants.

G={ (x1,Honda,x3,x4,x5), (x1,x2,Blue,x4,x5), (x1,x2,x3,1980,x5), (x1,x2,x3,x4,Economy) }

The set S is uneffected by the negative example.

Now we come to the third example, a positive one. The first order of business is to remove from the G set an descriptions that are inconsistent with positive example

(3)

Orgin: Japan

Manufacturer: Toyota

Color: Blue

Decade: 1990

Type : Economy

(-)

G={ (x1,x2,Blue,x4,x5), (x1,x2,x3,x4,Economy) }

We must now generalize the S set to include the new example. This involves replacing constants with variables. Here is new set S set.

S={ (Japan,x2,Blue,x4,Economy)}

S => Japanese blue economy car

G=> either blue car or economy car.

(4)

Orgin: USA

Manufacturer: Chrysler

Color: Red

Decade: 1980

Type : Economy

(-)

The set S is uneffected, but the G set must be specialized to avoid coving the new example

G=((Japan,x2,Blue,x4,x5), (Japan,x2,x3,x4,Economy) }

We now know that the car must be japanese because all of the the description in the version space contains Japan as orgin. It could be the case that our target concept is "not chrystler" but we will ignore this posiblilty because our representation language is not powerful enough to express negation and disjunction.

(5)

Orgin: Japan

Manufacturer: Honda

Color: White

Decade: 1980

Type : Economy

(+)

G={(Japan,x2,x3,x4,Economy)}

S={(Japan,x2,x3,x4,Economy)}

G and S are both singletons so the algotithm has converted on target concept. No more examples are needed.

Things to Note

1. Least_Commitment Algorithm: the version space is prunned as little as possible at each step

2. A algorithm exhaustive, breadth first search technique through version space

3. In our simple representation language the set S always contains exactly one element, because any two positive examplw have exactly one generalization. Other representation lanuages may not share this property.

Deficiencies

1. Large space requirements of the exhaustive, breath first search

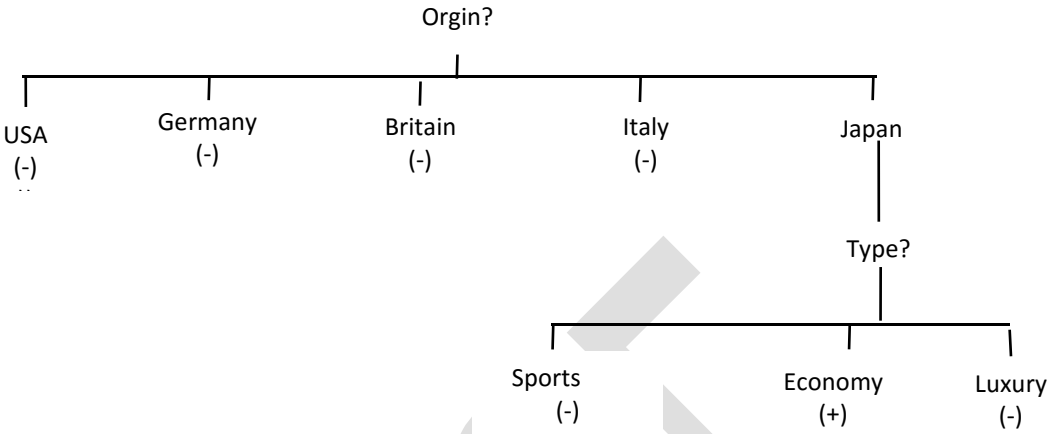
2. Inconsistent data also called noise can cause the candidate elimination algorithm to prune the target concept from version space prematurely.

3. Learning of disjuctive concepts: eg, suppose we wanted to learn the concept of "European Car" which in our representation either a German, British or Italian Car. Given such a generalization, a negative instance(Japanese Car) will cause an inconsistency of the type mentioned above.

Decision Trees

A third approach to concept learning is the induction of decision trees as exemplified by the ID3 program.

ID3 uses a tree representation for conceptssuch as shown in below figure:



To classify a particular input we start at top of the tree and answer questions until we reach a leaf, where classification is stored. The above figure represents the concept "Japanese Economy car".

ID3 is a program that build decision trees automatically given positive and negative instances of a concept.

ID3 uses an iterative method to build up decision trees, preferring simple trees over complex ones, on the theory that simple trees are more acurate classifier of future inputs. It begins by choosing a random subset of the training examples. This subset is called window. The algorithm builds a decision tree that correctly classifies all examples in window. The tree is then tested on the training example outside the window. If all the examples are classified correctly, the algorithm halts. Otherwise, it adds a number of training examples to the window and the process repeats.

At a given point in the tree, some attribues will yeild more information than others. For example, testing the attribute color is useless if the color of a car does not help us to classify it correctly. Ideally an attribute will seperate training instances into subsets whose members share common label. In that case branching is terminated and leaf nodes are labeled.

There are many variations on the basic algorithm. For example, when we add a test that has more than two branches, it is possible that one branch has no corresponding training instances. In that case we can either leave the node unlabeled or we can

attempt to guess a label based on statistical properties of the set of instances being tested at that point in the tree.

Noisy input is another issue. One way of handling noisy input is to avoid building new branches if information gained is slight. We do not want to overcomplicate the tree to account for isolated noisy instances.

Another source of uncertainity is that attribute values may be unknown. For example patient's medical records may be incomplete. One solution to guess the correct branch to take; another solution is to build special unknown branches at each node guring learning.

Advantages

- When concept space is very large, decision tree learning algorithm run more quickly than their version space.
- Also disjunction is more straightforward

Drawback

- Large complex decision tree can be difficult for humans to understand and so a decision tree system may have a hard time explaining the reason for its classifications.

Backpropagation

Backpropagation, short for "backward propagation of errors," is an algorithm for supervised learning of artificial neural networks using gradient descent. Given an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights.

The "backwards" part of the name is from the fact that calculation of the gradient proceeds backwards through the network, with the **gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last.** Partial computations of the gradient from one layer are reused in the computation of the gradient for the previous layer. This backwards flow of the error information allows for efficient computation of the gradient at each layer versus the naive approach of calculating the gradient of each layer separately.

Backpropagation's popularity has experienced a recent resurgence given the widespread adoption of deep neural networks for image recognition and speech recognition. It is considered an efficient algorithm, and modern implementations take advantage of specialized GPUs to further improve performance.

Formal Definition

backpropagation requires three things:

DEFINITION

1) **Dataset** consisting of input-output pairs $(\vec{x_i}, \vec{y_i})$, where $\vec{x_i}$ is the input and $\vec{y_i}$ is the desired output of the network on input $\vec{x_i}$. The set of input-output pairs of size N is denoted $X = \left\{(\vec{x_1}, \vec{y_1}), \ldots, (\vec{x_N}, \vec{y_N})\right\}$.

2) A **feedforward neural network**, as formally defined in the article concerning feedforward neural networks, whose parameters are collectively denoted θ. In backpropagation, the parameters of primary interest are w_{ij}^k, the weight between node j in layer l_k and node i in layer l_{k-1}, and b_i^k, the bias for node i in layer l_k. There are no connections between nodes in the same layer and layers are fully connected.

3) An **error function**, $E(X, \theta)$, which defines the error between the desired output $\vec{y_i}$ and the calculated output $\hat{\vec{y_i}}$ of the neural network on input $\vec{x_i}$ for a set of input-output pairs $(\vec{x_i}, \vec{y_i}) \in X$ and a particular value of the parameters θ.

Training a neural network with gradient descent requires the calculation of the gradient of the error function $E(X, \theta)$ with respect to the weights w_{ij}^k and biases b_i^k. Then, according to the learning rate α, each iteration of gradient descent updates the weights and biases (collectively denoted θ) according to

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta^t)}{\partial \theta},$$

where θ^t denotes the parameters of the neural network at iteration t in gradient descent.

Backpropagation is the central mechanism by which neural networks learn. It is the messenger telling the network whether or not the net made a mistake when it made a prediction.

To propagate is to transmit something (light, sound, motion or information) in a particular direction or through a particular medium. When we discuss

backpropagation in deep learning, we are talking about the transmission of information, and that information relates to the error produced by the neural network when they make a guess about data.

Untrained neural network models are like new-born babies: They are created ignorant of the world, and it is only through exposure to the world, experiencing it, that their ignorance is slowly revised. Algorithms experience the world through data. So by training a neural network on a relevant dataset, we seek to decrease its ignorance. The way we measure progress is by monitoring the error produced by the network.

The knowledge of a neural network with regard to the world is captured by its weights, the parameters that alter input data as its signal flows through the neural network towards the net's final layer that will make a decision about that input. Those decisions are often wrong, because the parameters transforming the signal into a decision are poorly calibrated; they haven't learned enough yet. A data instance flowing through a network's parameters toward the prediction at the end is forward propagation. Once that prediction is made, its distance from the ground truth (error) can be measured.

So the parameters of the neural network have a relationship with the error the net produces, and when the parameters change, the error does, too. We change the parameters using an optimization algorithm called gradient descent, which is useful for finding the minimum of a function. We are seeking to minimize the error, which is also known as the loss function or the objective function.

Backpropagation

A neural network propagates the signal of the input data forward through its parameters towards the moment of decision, and then backpropagates information about the error through the network so that it can alter the parameters one step at a time.

You could compare a neural network to a large piece of artillery that is attempting to strike a distant object with a shell. When the neural network makes a guess about an instance of data, it fires, a cloud of dust rises on the horizon, and the gunner tries to make out where the shell struck, and how far it was from the target. That distance from the target is the measure of error. The measure of error is then applied to the angle of and direction of the gun (parameters), before it takes another shot.

Backpropagation takes the error associated with a wrong guess by a neural network, and uses that error to adjust the neural network's parameters in the direction of less error. How does it know the direction of less error?

Gradient Descent

A gradient is a slope whose angle we can measure. Like all slopes, it can be expressed as a relationship between two variables: "y over x", or rise over run. In this case, the y is the error produced by the neural network, and x is the parameter of the neural network. The parameter has a relationship to the error, and by changing the parameter, we can increase or decrease the error. So the gradient tells us the change we can expect in y with regard to x.

To obtain this information, we must use differential calculus, which enables us to measure instantaneous rates of change, which in this case is the tangent of a changing slope expressed the relationship of the parameter to the neural network's error.

Obviously, a neural network has many parameters, so what we're really measuring are the partial derivatives of each parameter's contribution to the total change in error.

What's more, neural networks have parameters that process the input data sequentially, one after another. Therefore, backpropagation establishes the relationship between the neural network's error and the parameters of the net's last layer; then it establishes the relationship between the parameters of the neural net's last layer those the parameters of the second-to-last layer, and so forth, in an application of the chain rule of calculus.

It is of interest to note that backpropagation in artificial neural networks has echoes in the functioning of biological neurons, which respond to rewards such as dopamine to reinforce how they fire; i.e. how they interpret the world. Dopaminergenic behavior tends to strengthen the ties between the neurons involved, and helps those ties last longer.

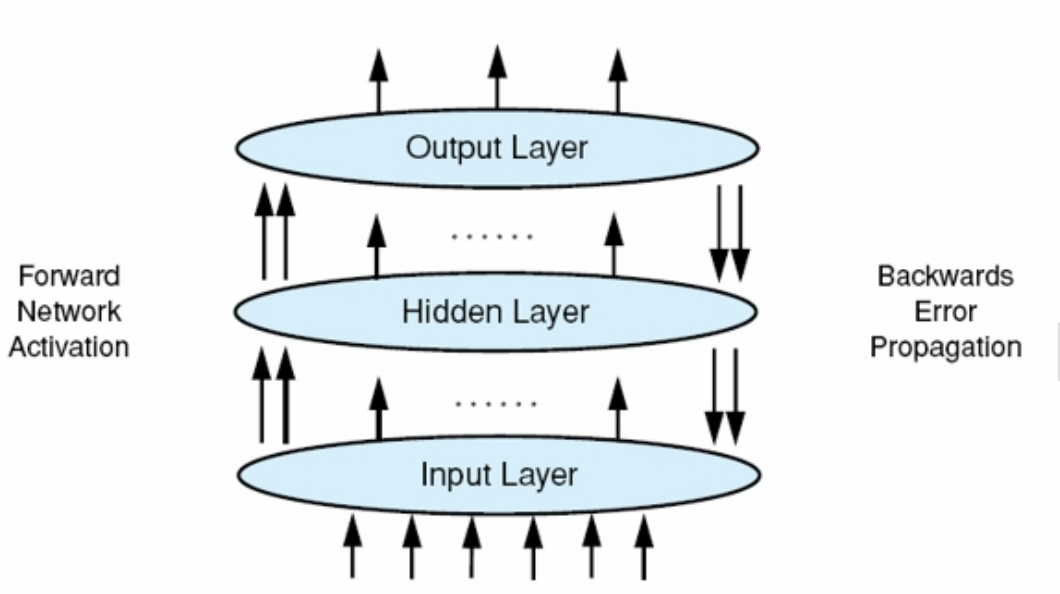
Backpropagation learning

Single layer perceptron networks are limited as to the classifications that they can perform. Addition of multiple layers can overcome many of these limitations

The neurons in a multilayer network, as in Figure 11.9, are connected in layers, with units in layer n passing their activations only to neurons in layer n + 1. Multilayer

signal processing means that errors deep in the network can spread and evolve in complex, unanticipated ways through successive layers Thus, the analysis of the source of error at the output layer is complex. Backpropagation provides an algorithm for apportioning blame and adjusting weights accordingly.

Fig 11.9 Backpropagation in a connectionist network having a hidden layer.



The approach taken by the backpropagation algorithm is to start at the output layer and propagate error backwards through the hidden layers. When we analyzed learning with the delta rule, we saw that all the information needed to update the weights on a neuron was local to that neuron, except for the amount of error. For output nodes, this is easily computed as the difference between the desired and actual output values. For nodes in the hidden layers, it is considerably more difficult to determine the error for which a node is responsible

The activation function for backpropagation is usually the logistic function:

$f(net) = 1/(1 + e^{-\lambda * net})$, where $net = \Sigma x_i w_i$

This function is used for four reasons:

First, it has the sigmoid shape.

Second, as a continuous function, it has a derivative everywhere.

Third, since the value of the derivative is greatest where the sigmoidal function is changing most rapidly, the assignment of the most error is attributed to those nodes whose activation was least certain.

Finally, the derivative is easily computed by a subtraction and multiplication:

$$f'(net) = (1/(1 + e^{-\lambda * net})) = \lambda(f(net) * (1 - f(net))).$$

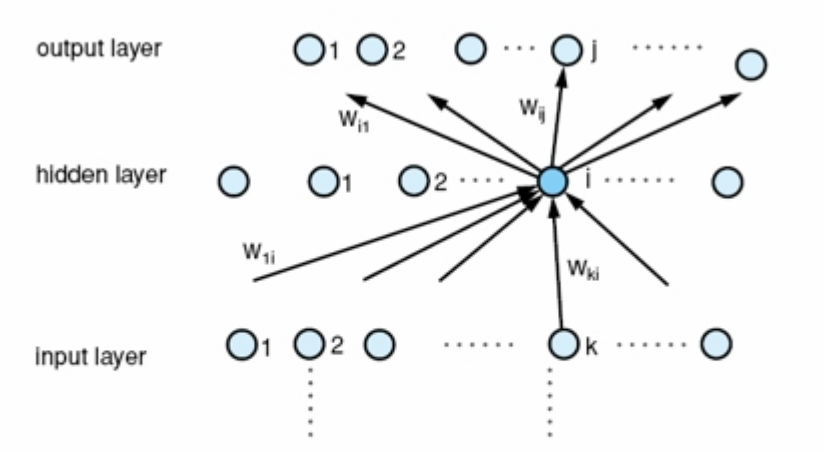
Backpropagation training uses the generalized delta rule. This uses the gradient descent approach. For nodes in the hidden layer we look at their contribution to the error at the output layer.

The formulas for computing the adjustment of the weight wki on the path from the kth to the ith node in backpropagation training are:

1) $\Delta wki = -c(di - Oi) * Oi (1 - Oi) xk$, for nodes on the output layer, and

2) $\Delta wki = -c * Oi (1 - Oi) \Sigma j (-deltaj * wij) xk$, for nodes on hidden layers.

In 2), j is the index of the nodes in the next layer to which i's signals fan out and: Refer page 467-472, AI by Luger



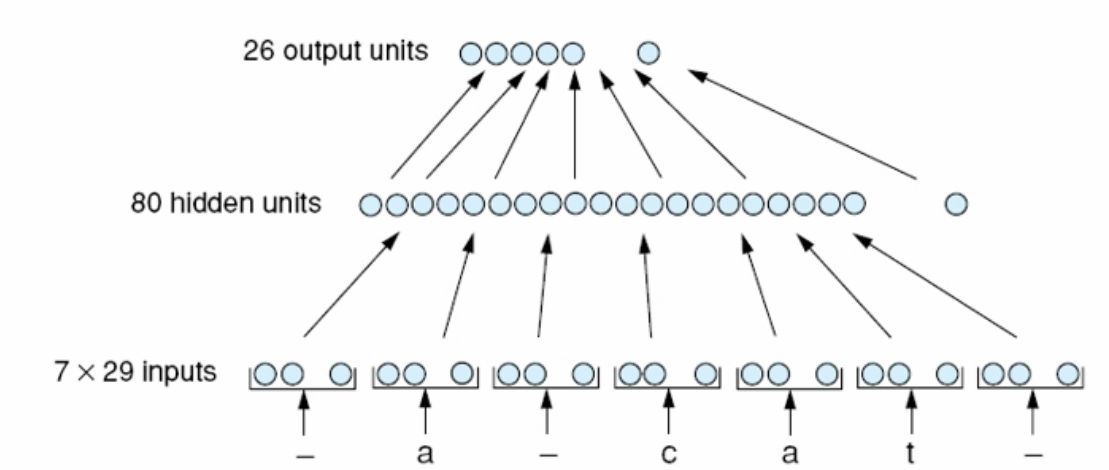
$\sum_j -delta_j * w_{ij}$ is the total contribution of node i to the error at the output. Our derivation gives the adjustment for w_{ki}.

Backpropagation example :nettalk

NETtalk learned to pronounce English text. This can be a difficult task since English pronunciation is highly irregular. NETtalk learned to read a string of text and return a phoneme and an associated stress for each letter in the string.

A phoneme is the basic unit of sound in a language; the stress is the relative loudness of that sound. Because the pronunciation of a single letter depends upon its context and the letters around it, NETtalk was given a seven character window. As the text moves through this window, NETtalk returns a phoneme/stress pair for each letter.

Fig 11.11 The network topology of NETtalk.



The Figure 11.11 shows the architecture of NETtalk. The network consists of three layers of units. The input units correspond to the seven character window on the text. Each position in the window is represented by 29 input units, one for each letter of the alphabet, and 3 for punctuation and spaces. The letter in each position activates the corresponding unit. The output units encode phonemes using 21 different features of human articulation. The remaining five units encoded stress and syllable boundaries. NETtalk has 80 hidden units, 26 output values, and 18,629 connections.

NETtalk is trained by giving it a seven character window and letting it attempt to pronounce the middle character. Comparing its attempted pronunciation to the correct pronunciation, it adjusts its weights using backpropagation. NETtalk, like many backpropagation networks, has fewer neurons in the hidden layer than in the input layer.

This means that since fewer nodes on the hidden layer are used to encode the information in the training patterns, some form of abstraction is taking place. The shorter encoding implies that different patterns on the input layer can be mapped into identical patterns at the hidden layer. This reduction is a generalization.

Backpropagation example :exclusive-or

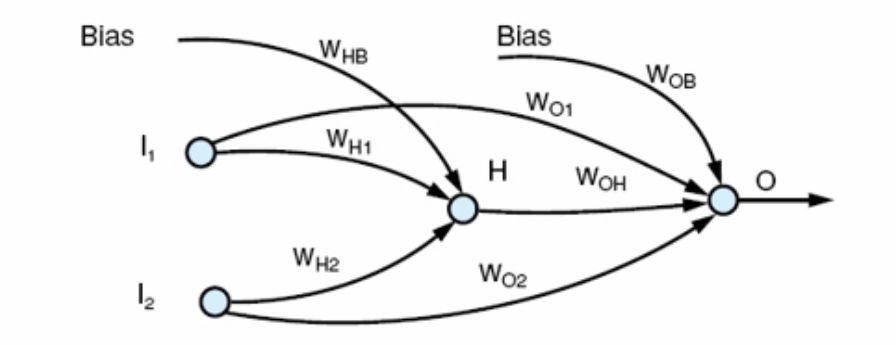


Fig 11.12 A backpropagation net to solve the exclusive-or problem. The W_{ij} are the weights and H is the hidden node.

Social and emergent models of learning

Emergent models of learning simulate nature's most elegant and powerful form of adaptation: the evolution of plant and animal life forms. Charles Darwin saw "...no limit to this power of slowly and beautifully adapting each form to the most complex relations of life...". Through this simple process of introducing variations into successive generations and selectively eliminating less fit individuals, adaptations of increasing capability and diversity emerge in a population. Evolution and emergence occur in populations of embodied individuals, whose actions affect others and that, in turn, are affected by others. Thus, selective pressures come not only from the outside environment, but also from inter actions between members of a population. An ecosystem has many members, each with roles and skills appropriate to their own survival, but more importantly, whose cumulative behavior shapes and is shaped by the rest of the population.

When the genetic algorithm is used for problem solving, it has three distinct stages:

first, the individual potential solutions of the problem domain are encoded into representations that support the necessary variation and selection operations; often, these representations are as simple as bit strings. In the second stage, mating and mutation algorithms, analogous to the sexual activity of biological life forms, produce a new generation of individuals that recombine features of their parents. Finally, a

fitness function judges which individuals are the “best” life forms, that is, most appropriate for the eventual solution of the problem. These individuals are favored in survival and reproduction, shaping the next generation of potential solutions. Eventually, a generation of individuals will be interpreted back to the original problem domain as solutions for the problem. Genetic algorithms are also applied to more complex representations, including production rules, to evolve rule sets adapted to interacting with an environment. For example,genetic programming combines and mutates fragments of computer code in an attempt to evolve a program for solving problems such as capturing the invariants in sets of data.

Genetic Algorithm

Like neural networks, genetic algorithms are based on a biological metaphor: They view learning as a competition among a population of evolving candidate problem solutions. A “fitness” function evaluates each solution to decide whether it will contribute to the next generation of solutions. Then, through operations analogous to gene transfer in sexual reproduction, the algorithm creates a new population of candidate solutions.

Let P(t) define a population of candidate solutions, , at time t:

We now present a general form of the genetic algorithm:

```
procedure genetic algorithm;

    begin
        set time t:= 0;
        initialize the population P(t);
        while the termination condition is not met do
            begin
                evaluate fitness of each member of the population P(t);
                select members from population P(t) based on fitness;
                produce the offspring of these pairs using genetic operators;
                replace, based on fitness, candidates of P(t), with these offspring;
                set time t := t +1
            end
    end.
```

This algorithm articulates the basic framework of genetic learning; specific implementations of the algorithm instantiate that framework in different ways. What percentage of the population is retained? What percentage mate and produce offspring?

How often and to whom are the genetic operators applied? The procedure "replace the weakest candidates of P(t)" may be implemented in a simple fashion, by eliminating fixed percentage of the weakest candidates. More sophisticated approaches may order a population by fitness and then associate a probability measure for elimination with each member, where the probability of elimination is an inverse function of its fitness. Then the replacement algorithm uses this measure as a factor in selecting candidates to eliminate. Although the probability of elimination would be very low for the fittest members of the society, there is a chance that even the best individuals could be removed. The advantage of this scheme is that it may save some individuals whose overall fitness is poor but that include some component that may contribute to a more powerful solution. This replacement algorithm has many names, including Monte Carlo, fitness proportionate selection, and roulette wheel.

The genetic algorithm initializes P(0) to a population of candidate patterns. Typically, initial populations are selected randomly. Evaluation of candidate solutions assumes a fitness function, $f(x_i^t)$ that returns a measure of the candidate's fitness at time t. A common measure of a candidate's fitness tests it on a set of training instances and returns the percentage of correct classifications.

Using such a fitness function, an evaluation assigns each candidate solution the value:

$$f(x_i^t)/m(P, t)$$

where m(P,t) is the average fitness over all members of the population. It is also common for the fitness measure to change across time periods, thus fitness could be a function of the stage of the overall problem solution.

After evaluating each candidate, the algorithm selects pairs for recombination. Recombination uses genetic operators to produce new solutions that combine components of their parents. As with natural evolution, the fitness of a candidate determines the extent to which it reproduces, with those candidates having the highest evaluations being given a greater probability of reproducing.

Selection is often probabilistic, where weaker members are given a smaller likelihood of reproducing, but are not eliminated outright. That some less fit candidates survive is important since they can still contain some essential component of a solution, for instance part of a bit pattern, and reproduction may extract this component.

There are a number of genetic operators that produce offspring having features of their parents; the most common of these is crossover. Crossover takes two candidate solutions and divides them, swapping components to produce two new candidates.

Figure 12.1 illustrates crossover on bit string patterns of length 8.

The operator splits them in the middle and forms two children whose initial segment comes from one parent and whose tail comes from the other. Note that splitting the candidate solution in the middle is an arbitrary choice. This split may be at any point in the representation, and indeed, this splitting point may be randomly adjusted or changed during the solution process

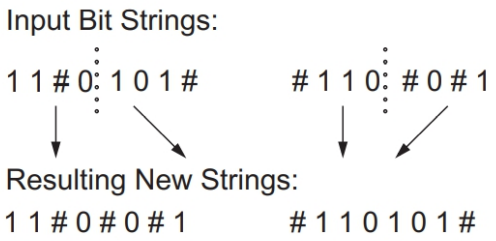


Figure 12.1 Use of crossover on two bit strings of length eight. # is don t care.

Mutation is another important genetic operator. Mutation takes a single candidate and randomly changes some aspect of it. For example, mutation may randomly select a bit in the pattern and change it, switching a 1 to a 0 or #.

Mutation is important in that the initial population may exclude an essential component of a solution. In our example, if no member of the initial population has a 1 in the first position, then crossover, because it preserves the first four bits of the parent to be the first four bits of the child, cannot produce an offspring that does. Mutation would be needed to change the values of these bits.

Other genetic operators, for example inversion, could also accomplish this task The genetic algorithm continues until some termination requirement is met, such as having one or more candidate solutions whose fitness exceeds some threshold.

CLASSIFIER SYSTEMS

Early research in genetic algorithms focused almost exclusively on low-level representations, such as strings of {0, 1, #}.

Holland (1986) developed a problem-solving architecture called classifier systems that applies genetic learning to rules in a production system. A classifier system (Figure 12.3) includes the familiar elements of a production system:

- production rules (here called classifiers)
- working memory
- input sensors (or decoders)
- and outputs (or effectors).

Unusual features of a classifier system include the use of competitive bidding for conflict resolution, genetic algorithms for learning, and the bucket brigade algorithm to assign credit and blame to rules during learning.

Feedback from the outside environment provides a means of evaluating the fitness of candidate classifiers, as required in genetic learning.

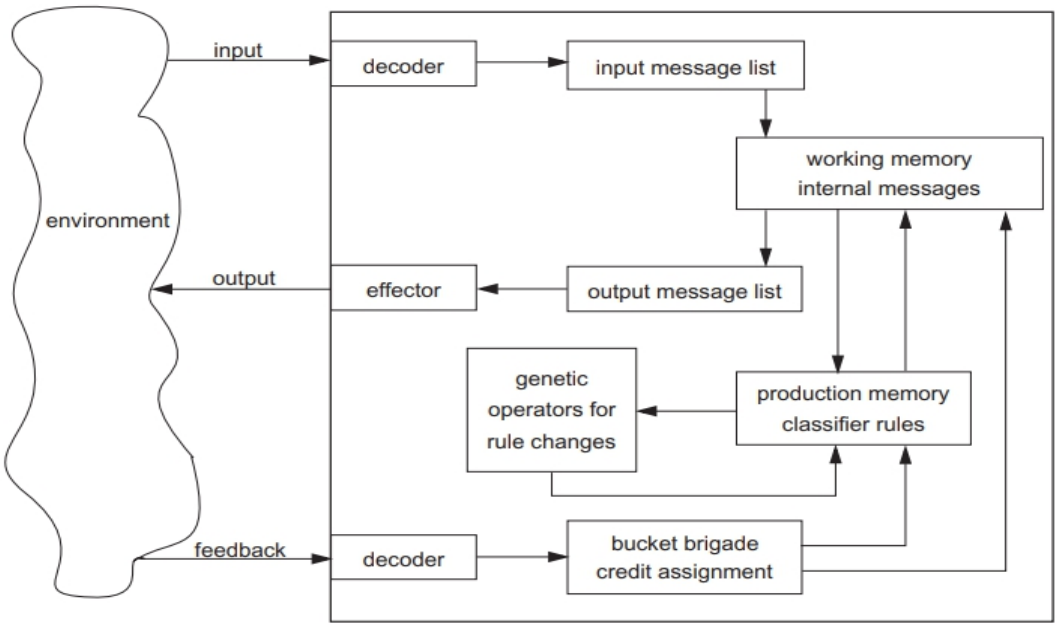


Figure 12.3 A classifier system interacting with the environment, adapted from Holland (1986).

The classifier system in the figure 12.3 has the following major components

1. Detectors of input messages from the environment.
2. Detectors of feedback messages from the environment.
3. Effectors translating results of rule applications back to the environment.
4. A production rule set made up of a population of classifiers. Each classifier has an associated fitness measure.
5. A working memory for the classifier rules. This memory integrates the results of production rule firing with input information.
6. A set of genetic operators for production rule modification.
7. A system for giving credit to rules involved in producing successful actions.

In problem solving, the classifier performs as a traditional production system. The environment sends a message, perhaps a move in a game, to the classifier system's detectors. This event is decoded and placed as a pattern on the internal message list, the working memory for the production system. These messages, in the normal action of data-driven production system, match the condition patterns of the classifier rules. The selection of the "strongest activated classifiers" is determined by a bidding scheme

A bid is a function of both the accumulated fitness of the classifier and the quality of the match between the input stimulus and its condition pattern. The classifiers with the closest match add messages (the action of the fired rules) to working memory. The revised message list may send messages to the effectors which act upon the environment or activate new classifier rules as the production system processing continues.

Classifier systems implement a form of reinforcement learning. Based on feedback from a teacher or fitness evaluation function, the learner computes the fitness of a population of candidate rules and adapts this population using a variation of genetic learning. Classifier systems learn in two ways.

First, there is a reward system that adjusts the fitness measures of the classifier rules, rewarding successful rule firings and penalizing errors. The credit assignment algorithm passes part of the reward or penalty back to any classifier rules that have contributed to the final rule firing. This distribution of differential rewards across

interacting classifiers, as well as those that enabled their firing, is often implemented in a bucket brigade algorithm.

The bucket brigade algorithm addresses the problem of assigning credit or blame in situations where the system's output may be the product of a sequence of rule firings. The bucket brigade algorithm allocates both credit and blame across a sequence of rule applications according to measures of each rule's contribution to the final conclusion.

The second form of learning modifies the rules themselves using genetic operators such as mutation and crossover. This allows the most successful rules to survive and combine to make new classifiers, while unsuccessful rule classifiers disappear.

Each classifier rule consists of three components: First, the rule's condition matches data in the working memory in the typical production system sense. In learning, genetic operators can modify both the conditions and the actions of the production rules. The second component of the rule, the action, can have the effect of changing the internal message list (the production memory).

Finally, each rule has a fitness measure. This parameter is changed, both by successful as well as by unsuccessful activity. This measure is originally assigned to each rule on its creation by the genetic operators; for example, it may be set as the average fitness of its two parents.

A simple example illustrates the interactions of these components of a classifier system. Assume that a set of objects to be classified are defined by six attributes (conditions c1, c2, ..., c6), and suppose that each of these attributes can have five different values. Although the possible values of each attribute are different (for example, the value of c3 might be color, while c5 might describe the weather) we will, without loss of generality, give each attribute an integer value from {1, 2, ..., 5}. Suppose the conditions of these rules place their matching object in one of four classes: A1, A2, A3, A4.

Based on these constraints, each classifier will have the form:

$$(c1\ c2\ c3\ c4\ c5\ c6) \rightarrow Ai$$

where i = 1, 2, 3, 4.

where each ci in the condition pattern denotes the value {1, 2, ..., 5} of the ith attribute of the condition.

Usually, conditions can also assign a value of # or "don't care" to an attribute. Ai denotes the classification, A1, A2, A3, or A4.

Table 12.2 presents a set of classifiers. Note that different condition patterns can have the same classification, as in rules 1 and 2, or the same patterns, as in rules 3 and 5, can lead to different classifications.

Condition (Attributes)		Action (Classification)	Rule Number
(1 # # # 1 #)	→	A1	1
(2 # # 3 # #)	→	A1	2
(# # 4 3 # #)	→	A2	3
(1 # # # # #)	→	A2	4
(# # 4 3 # #)	→	A3	5
etc.			

Table 12.2 A set of condition → action classifiers to be learned.

Consider the classifier system's performance in learning the classification A1.

That is, we will ignore the other classifications, and assign condition patterns a value of 1 or 0 depending on whether or not they support classification A1.

Note that there is no loss of generality in this simplification; it may be extended to problems of learning more than one classification by using a vector to indicate the classifications that match a particular condition pattern.

For example, the classifiers of Table 12.2 may be summarized by A1A2A3A4

(1 # # # 1 #) → (1 0 0 0)

(2 # # 3 # #) → (1 0 0 0)

(1 # # # # #) → (0 1 0 0)

(# # 4 3 # #) → (0 1 1 0)

In this example, the last of these summaries indicates that the condition attributes support classification rules A2 and A3 and not A1 or A4. By replacing the 0 or 1

assignment with these vectors, the learning algorithm can evaluate the performance of a rule across multiple classifications.

In this example, we will use the rules in Table 12.2 to indicate the correct classifications; essentially, they will function as teachers or evaluators of the fitness of rules in the learning system.

As with most genetic learners, we begin with a random population of rules. Each condition pattern is also assigned a strength, or fitness, parameter (a real number between 0.0, no strength, and 1.0, full strength. This strength parameter, s, is computed from the fitness of each rule's parents, and measures its historical fitness.

At each learning cycle, the rules attempt to classify the inputs and are then ranked by the teacher or fitness metric.

For example, assume that at some cycle, the classifier has the following population of candidate classification rules, where the conclusion of 1 indicates that the pattern led to a correct classification and 0 that it did not:

(# # # 2 1 #) → 1 s = 0.6

(# # 3 # # 5) → 0 s = 0.5

(2 1 # # # #) → 1 s = 0.4

(# 4 # # # 2) → 0 s = 0.23

GENETIC PROGRAMMING

Koza has suggested that a successful computer program might evolve through successive applications of genetic operators. In genetic programming, the structures being adapted are hierarchically organized segments of computer programs. The learning algorithm maintains a population of candidate programs. The fitness of a program will be measured by its ability to solve a set of tasks programs are modified by applying crossover and mutation to program subtrees. Genetic programming searches a space of computer programs of varying size and complexitythe search space is the space of all possible computer programs composed of functions and terminal symbols appropriate to the problem domain. This search is random, largely blind and effective

Genetic programming starts with an initial population of randomly generated programs made up of appropriate program pieces. These pieces, suitable for a problem domain, may consist of standard arithmetic operations, other related programming operations, and mathematical functions, as well as logical and domain-specific functions. Program components include data items of the usual types: Boolean, integer, floating point, vector, symbolic, or multiple-valued.

After initialization, thousands of computer programs are genetically bred. The production of new programs comes with application of genetic operators. Crossover, mutation, and other breeding algorithms must be customized for the production of computer programs.

The fitness of each new program is then determined by seeing how well it performs in a particular problem environment. The nature of the fitness measure will vary according to the problem domain. Any program that does well on this fitness task will survive to help produce the children of the next generation.

To summarize, genetic programming includes six components, many very similar to the requirements for GAs:

1. A set of structures that undergo transformation by genetic operators.
2. A set of initial structures suited to a problem domain.
3. A fitness measure, again domain dependent, to evaluate structures.
4. A set of genetic operators to transform structures.
5. Parameters and state descriptions that describe members of each generation.
6. A set of termination conditions.

Genetic programming manipulates hierarchically organized program modules Lisp was (and still remains) the primary representation for the programming language components. Koza represents program segments as Lisp symbol expressions, or s-expressions.

Genetic operators manipulate s-expressions. operators map tree structures of s-expressions, (Lisp program segments), into new trees, (new Lisp program segments).

Genetic programming will construct useful programs, given that the atomic pieces and evaluable predicates of the problem domain are available first analyze what terminals are required for units in its solution as well as what functions are necessary to produce these terminals.

To initialize the structures for adaptation by genetic operators, we must create two sets: F, the set of functions and T, the set of terminal values required for the domain. F can be as simple as {+, *, −, /} or may require more complex functions such as sin(X), cos(X), or functions for matrix operations. T may be the integers, reals, matrices, or more complex expressions. The symbols in T must be closed under the functions defined in F.

Next, a population of initial "programs" is generated by randomly selecting elements from the union of sets F and T. For example, if we begin by selecting an element of T, we have a degenerate tree of a single root node. when we start with an element from F, say +, we get a root node of a tree with two potential children. Suppose the initializer next selects * (with two potential children) from F, as the first child, and then terminal 6 from T as the second child. Another random selection might yield the terminal 8, and then the function + from F. Assume it concludes by selecting 5 and 7 from T

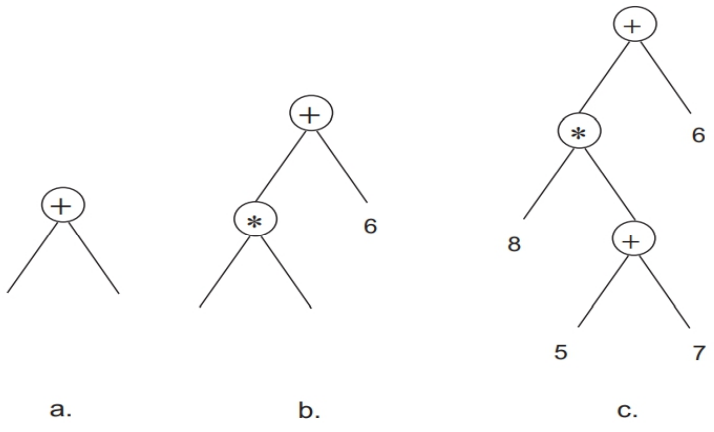


Figure 12.4 The random generation of a program to initialize. The circled nodes are from the set of functions.

The program we have randomly produced is represented in Figure 12.4. Figure 12.4a gives the tree after the first selection of +, 15.4b after selecting the terminal 6, and

15.4c the final program. A population of similar programs is created to initialize the genetic programming process. Sets of constraints, such as the maximum depth for programs to evolve, can help prune this population.

Next, we require a fitness measure for populations of programs. The fitness measure is problem domain dependent and usually consists of a set of tasks the evolved programs must address. The fitness measure itself is a function of how well each program does on these tasks.

A simple raw fitness score would add the differences between what the program produced and the results that the actual task from the problem domain required. Thus, raw fitness could be seen as the sum of errors across a set of tasks. Other fitness measures are also possible. Normalized fitness divides raw fitness by the total sum of possible errors and thus puts all fitness measures within the range of 0 to 1.

Normalization can have an advantage when trying to select from a large population of programs. A fitness measure can also include an adjustment for the size of the program, for example, to reward smaller programs.

Genetic operators on programs include both transformations on a tree itself as well as the exchange of structures between trees. Koza (1992) describes the primary transformations as reproduction and crossover. Reproduction simply selects programs from the present generation and copies them (unchanged) into the next generation.

Crossover exchanges subtrees between the trees representing two programs. For example, suppose we are working with the two parent programs of Figure 12.5, and that the random points indicated by | in parent a and parent b are selected for crossover. The resulting children are shown in Figure 12.6.

Crossover can also be used to transform a single parent, by interchanging two subtrees from that parent. Two identical parents can create different children with randomly selected crossover points. The root of a program can also be selected as a crossover point.

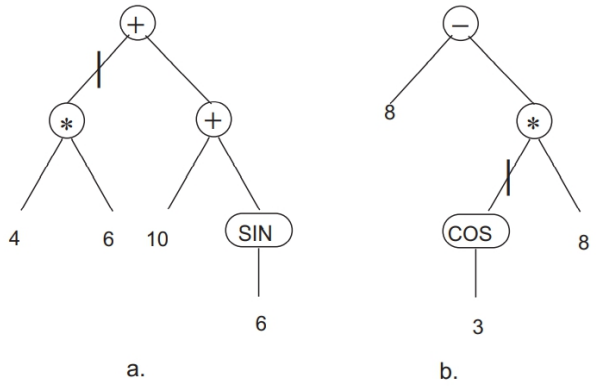


Figure 12.5 Two programs, selected on fitness for crossover. Point | from a and b are randomly selected for crossover.

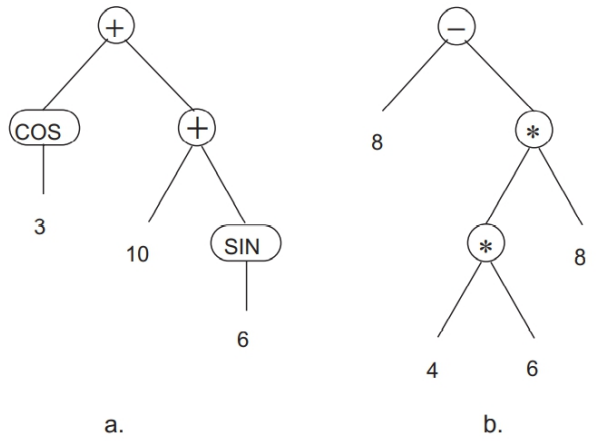


Figure 12.6 The child programs produced by cross-over of the points in Figure 12.5.

There are a number of secondary, and much less used, genetic transforms of program trees. These include mutation, which simply introduces random changes in the structures of a program. For example, replacing a terminal value with another value or a function subtree. The permutation transform, similar to the inversion operator on strings, also works on single programs, exchanging terminal symbols, or subtrees, for example.

The state of the solution is reflected by the current generation of programs. There is no record keeping for backtrack or any other method for skipping around the fitness landscape. The genetic programming paradigm parallels nature in that the evolution of new programs is a continuing process. Nonetheless, lacking infinite time and

computation, termination conditions are set. These are usually a function both of program fitness and computational resources.