[Unit 5: Machine Learning]

Artificial Intelligence (CSC 355)

Bal Krishna Subedi

Central Department of Computer Science & Information Technology
Tribhuvan University

What is Learning?

“Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task (or tasks drawn from the same population) more effectively the next time.” --Herbert Simon

"Learning is constructing or modifying representations of what is being experienced." --Ryszard Michalski

"Learning is making useful changes in our minds." --Marvin Minsky

Types of Learning:

The strategies for learning can be classified according to the amount of inference the system has to perform on its training data. In increasing order we have

1. **Rote learning** – the new knowledge is implanted directly with no inference at all, e.g. simple memorisation of past events, or a knowledge engineer’s direct programming of rules elicited from a human expert into an expert system.

2. **Supervised learning** – the system is supplied with a set of training examples consisting of inputs and corresponding outputs, and is required to discover the relation or mapping between then, e.g. as a series of rules, or a neural network.

3. **Unsupervised learning** – the system is supplied with a set of training examples consisting only of inputs and is required to discover for itself what appropriate outputs should be, e.g. a *Kohonen Network* or *Self Organizing Map*.

Early expert systems relied on rote learning, but for modern AI systems we are generally interested in the supervised learning of various levels of rules.

The need for Learning:

As with many other types of AI system, it is much more efficient to give the system enough knowledge to get it started, and then leave it to learn the rest for itself. We may even end up with a system that learns to be better than a human expert.

The ***general learning approach*** is to generate potential improvements, test them, and discard those which do not work. Naturally, there are many ways we might generate the potential improvements, and many ways we can test their usefulness. At one extreme, there are model driven (top-down) generators of potential improvements, guided by an understanding of how the problem domain works. At the other, there are data driven (bottom-up) generators, guided by patterns in some set of training data.

Machine Learning:

As regards machines, we might say, very broadly, that a machine learns whenever it changes its structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance improves. Some of these changes, such as the addition of a record to a data base, fall comfortably within the province of other disciplines and are not necessarily better understood for being called learning. But, for example, when the performance of a speech-recognition machine improves after hearing several samples of a person's speech, we feel quite justified in that case saying that the machine has learned.

Machine learning usually refers to the changes in systems that perform tasks associated with artificial intelligence (AI). Such tasks involve recognition, diagnosis, planning, robot control, prediction, etc. The changes might be either enhancements to already performing systems or synthesis of new systems.

Learning through Examples: (A type of Concept learning)

Concept learning also refers to a learning task in which a human or machine learner is trained to classify objects by being shown a set of example objects along with their class labels. The learner will simplify what has been observed in an example. This simplified version of what has been learned will then be applied to future examples. Concept learning ranges in simplicity and complexity because learning takes place over many areas. When a concept is more difficult, it will be less likely that the learner will be able to simplify, and therefore they will be less likely to learn. This learning by example consists of the idea of **version space**.

A **version space** is a hierarchical representation of knowledge that enables you to keep track of all the useful information supplied by a sequence of learning examples without remembering any of the examples.

The **version space method** is a concept learning process accomplished by managing multiple models within a version space.

Version Space Characteristics

In settings where there is a generality-ordering on hypotheses, it is possible to represent the version space by two sets of hypotheses: (1) the **most specific** consistent hypotheses and (2) the **most general** consistent hypotheses, where "consistent" indicates agreement with observed data.

The most specific hypotheses (i.e., the specific boundary **SB**) are the hypotheses that cover the observed positive training examples, and as little of the remaining feature space as possible. These are hypotheses which if reduced any further would *exclude* a *positive* training example, and hence become inconsistent. These minimal hypotheses essentially constitute a (pessimistic) claim that the true concept is defined just by the *positive* data already observed: Thus, if a novel (never-before-seen) data point is observed, it should be assumed to be negative. (I.e., if data has not previously been ruled in, then it's ruled out.)

The most general hypotheses (i.e., the general boundary **GB**) are those which cover the observed positive training examples, but also cover as much of the remaining feature space without including any negative training examples. These are hypotheses which if enlarged any further would *include* a *negative* training example, and hence become inconsistent.

Tentative heuristics are represented using version spaces. A version space represents all the alternative plausible **descriptions** of a heuristic. A plausible description is one that is applicable to all known positive examples and no known negative example.

A version space description consists of two complementary trees:

1. One that contains nodes connected to overly **general** models, and
2. One that contains nodes connected to overly **specific** models.

Node values/attributes are **discrete**.

<u>Fundamental Assumptions</u>

1. The data is correct; there are no erroneous instances.
2. A correct description is a conjunction of some of the attributes with values.

<u>Diagrammatical Guidelines</u>

There is a **generalization** tree and a **specialization** tree.

Each **node** is connected to a **model**.

Nodes in the generalization tree are connected to a model that matches everything in its subtree.

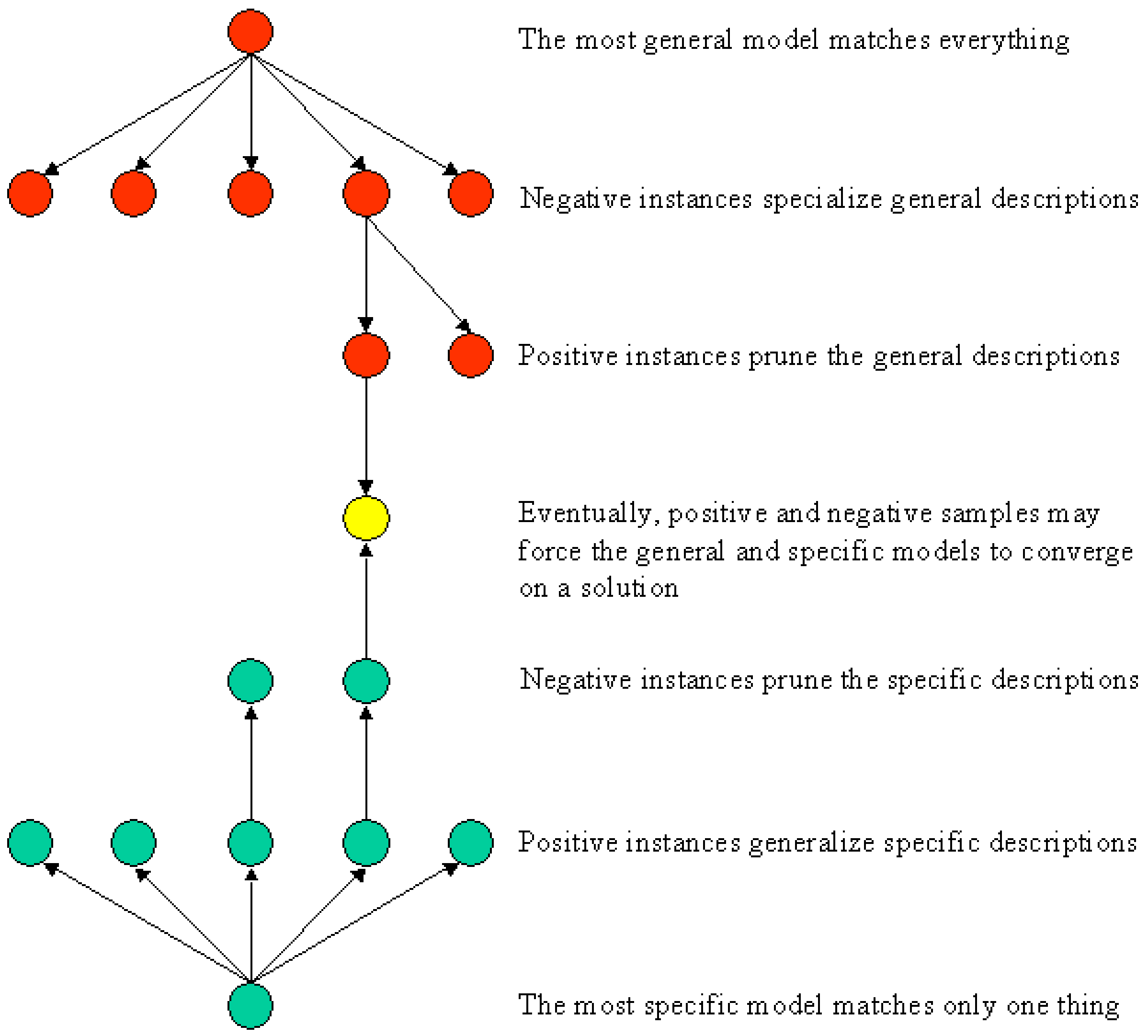
Nodes in the specialization tree are connected to a model that matches only one thing in its subtree.

Links between nodes and their models denote

- generalization relations in a generalization tree, and
- specialization relations in a specialization tree.

<u>Diagram of a Version Space</u>

In the diagram below, the specialization tree is colored **red**, and the generalization tree is colored **green**.


The most general model matches everything
Negative instances specialize general descriptions
Positive instances prune the general descriptions
Eventually, positive and negative samples may force the general and specific models to converge on a solution
Negative instances prune the specific descriptions
Positive instances generalize specific descriptions
The most specific model matches only one thing

<u>Generalization and Specialization Leads to Version Space Convergence</u>

The key idea in version space learning is that specialization of the general models and generalization of the specific models may ultimately lead to just one correct model that matches all observed positive examples and does not match any negative examples.

That is, each time a negative example is used to specialilize the general models, those specific models that match the negative example are eliminated and each time a positive example is used to generalize the specific models, those general models that fail to match the positive example are eliminated. Eventually, the positive and negative examples may be such that only one general model and one identical specific model survive.

Candidate Elimination Algorithm:

The version space method handles positive and negative examples symmetrically.

Given:

- A representation language.
- A set of positive and negative examples expressed in that language.

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

Method:

- Initialize G, the set of maximally general hypotheses, to contain one element: the null description (all features are variables).
- Initialize S, the set of maximally specific hypotheses, to contain one element: the first positive example.
- Accept a new training example.
    - If the example is **positive**:
        1. Generalize all the specific models to match the positive example, but ensure the following:
            - The new specific models involve minimal changes.
            - Each new specific model is a specialization of some general model.
            - No new specific model is a generalization of some other specific model.
        2. Prune away all the general models that fail to match the positive example.
    - If the example is **negative**:
        1. Specialize all general models to prevent match with the negative example, but ensure the following:
            - The new general models involve minimal changes.
            - Each new general model is a generalization of some specific model.
            - No new general model is a specialization of some other general model.
        2. Prune away all the specific models that match the negative example.
    - If S and G are both singleton sets, then:
        - if they are identical, output their value and halt.
        - if they are different, the training cases were inconsistent. Output this result and halt.
        - else continue accepting new training examples.

The algorithm stops when:

1. It runs out of data.
2. The number of hypotheses remaining is:
    - 0 - no consistent description for the data in the language.
    - 1 - answer (version space converges).
    - 2^+ - all descriptions in the language are implicitly included.

Problem 1:

Learning the concept of "Japanese Economy Car"

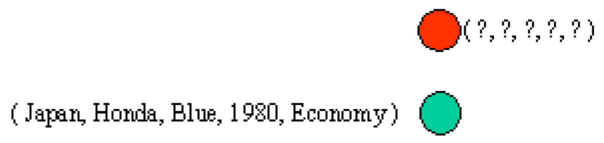
Features: (Country of Origin, Manufacturer, Color, Decade, Type)

Origin	Manufacturer	Color	Decade	Type	Example Type
Japan	Honda	Blue	1980	Economy	Positive
Japan	Toyota	Green	1970	Sports	Negative
Japan	Toyota	Blue	1990	Economy	Positive
USA	Chrysler	Red	1980	Economy	Negative
Japan	Honda	White	1980	Economy	Positive

Solution:

1. **Positive Example**: (Japan, Honda, Blue, 1980, Economy)

Initialize G to a singleton set that includes everything.	G = { (?, ?, ?, ?, ?) }
Initialize S to a singleton set that includes the first positive example.	S = { (Japan, Honda, Blue, 1980, Economy) }



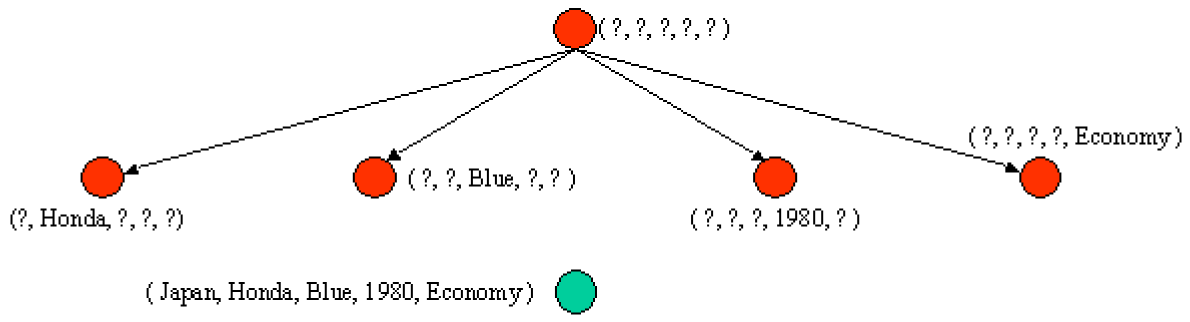
These models represent the most general and the most specific heuristics one might learn. The actual heuristic to be learned, "Japanese Economy Car", probably lies between them somewhere within the version space.

2. **Negative Example**: (Japan, Toyota, Green, 1970, Sports)

Specialize G to exclude the negative example.

G = { (?, Honda, ?, ?, ?),
(?, ?, Blue, ?, ?),
(?, ?, ?, 1980, ?),
(?, ?, ?, ?, Economy) }

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



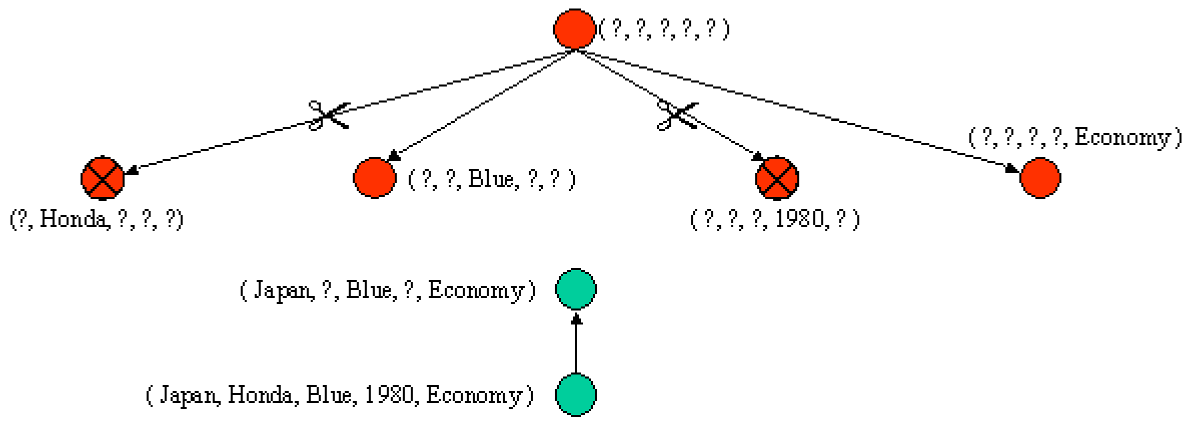
Refinement occurs by generalizing S or specializing G, until the heuristic hopefully converges to one that works well.

3. **Positive Example**: (Japan, Toyota, Blue, 1990, Economy)

Prune G to exclude descriptions inconsistent with the positive example. (Prune = ✂)
Generalize S to include the positive example.

G = { (?, ?, Blue, ?, ?),
(?, ?, ?, ?, Economy) }

S = { (Japan, ?, Blue, ?, Economy) }

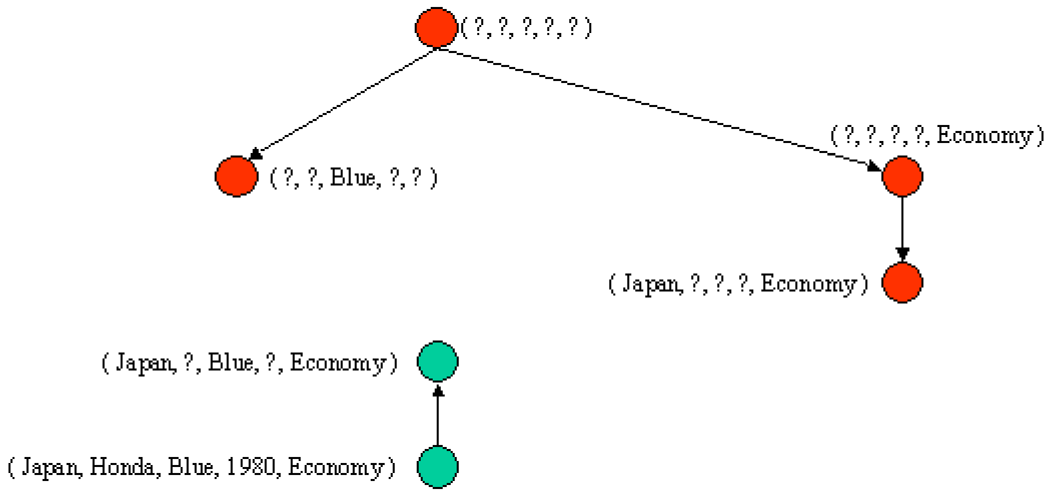


4. **Negative Example**: (USA, Chrysler, Red, 1980, Economy)

Specialize G to exclude the negative example (but stay consistent with S)

G = { (?, ?, Blue, ?, ?),
(Japan, ?, ?, ?, Economy) }

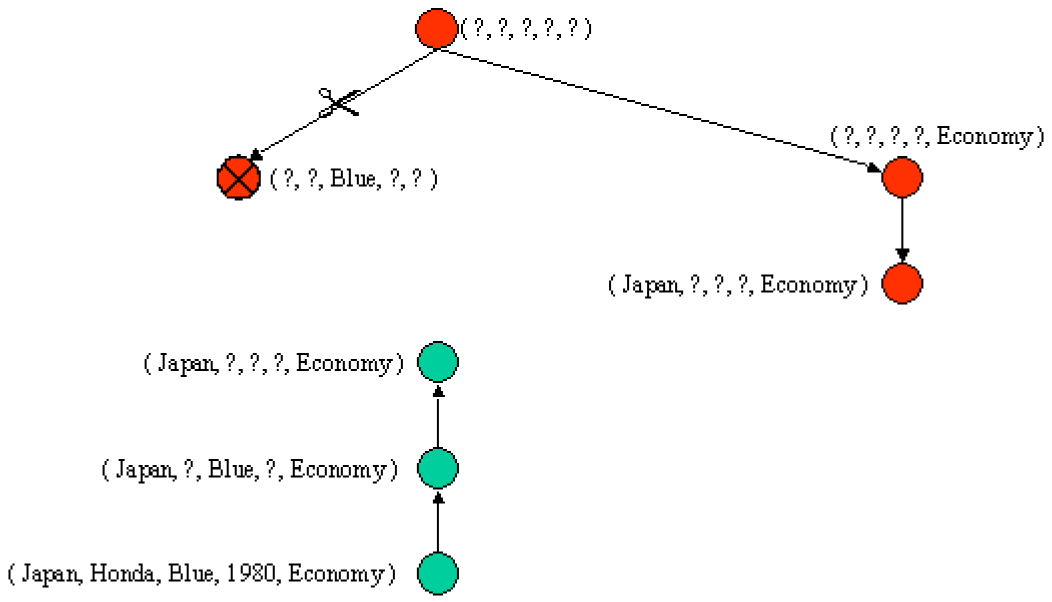
S = { (Japan, ?, Blue, ?, Economy) }


(?, ?, ?, ?, ?)
(?, ?, Blue, ?, ?)
(?, ?, ?, ?, Economy)
(Japan, ?, ?, ?, Economy)
(Japan, ?, Blue, ?, Economy)
(Japan, Honda, Blue, 1980, Economy)

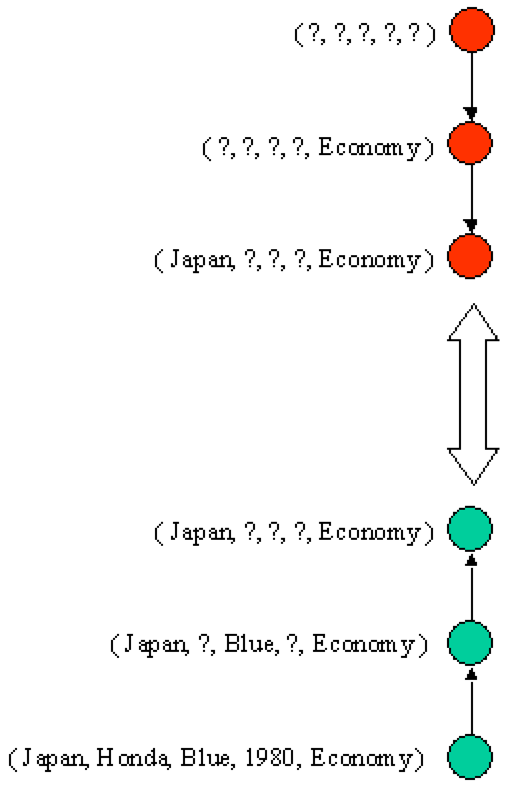
5. **Positive Example**: (Japan, Honda, White, 1980, Economy)

Prune G to exclude descriptions inconsistent with positive example.
Generalize S to include positive example.

G = { (Japan, ?, ?, ?, Economy) }
S = { (Japan, ?, ?, ?, Economy) }


(?, ?, ?, ?, ?)
(?, ?, Blue, ?, ?)
(?, ?, ?, ?, Economy)
(Japan, ?, ?, ?, Economy)
(Japan, ?, ?, ?, Economy)
(Japan, ?, Blue, ?, Economy)
(Japan, Honda, Blue, 1980, Economy)

G and S are singleton sets and S = G.
Converged.
No more data, so algorithm stops.


(?, ?, ?, ?, ?)
(?, ?, ?, ?, Economy)
(Japan, ?, ?, ?, Economy)
(Japan, ?, ?, ?, Economy)
(Japan, ?, Blue, ?, Economy)
(Japan, Honda, Blue, 1980, Economy)

Explanation Based Machine Learning:

Explanation-based learning (EBL) is a form of machine learning that exploits a very strong, or even perfect, domain theory to make generalizations or form concepts from training examples. This is a type of *analytic* learning. The advantage of explanation-based learning is that, as a deductive mechanism, it requires only a single training example (inductive learning methods often require many training examples)

An Explanation-based Learning (**EBL**) system accepts an example (i.e. a training example) and explains what it learns from the example. The **EBL** system takes only the relevant aspects of the training.

EBL accepts four inputs:

A training example : what the learning *sees* in the world. (specific facts that rule out some possible hypotheses)

A goal concept : a high level description of what the program is supposed to learn. (the set of all possible conclusions)

A operational criterion : a description of which concepts are usable. (criteria for determining which features in the domain are efficiently recognizable, e.g. which features are directly detectable using sensors)

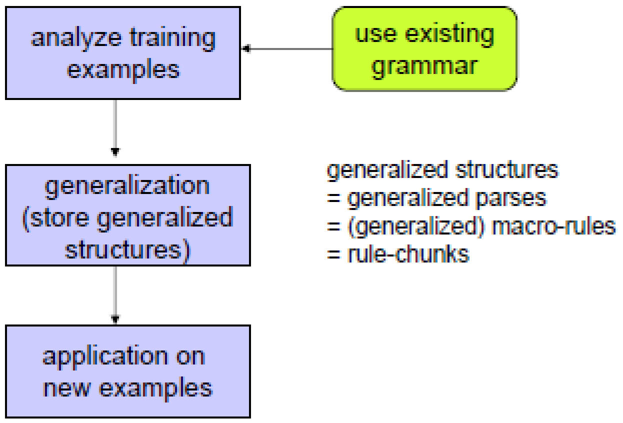
A domain theory : a set of rules that describe relationships between objects and actions in a domain. (axioms about a domain of interest)

From this EBL computes a generalization 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.

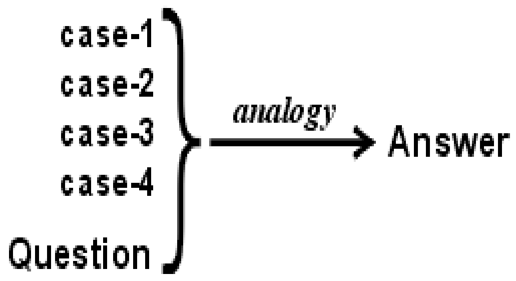
Generalisation: the explanation is generalized as far possible while still describing the goal concept


analyze training examples
use existing grammar
generalization (store generalized structures)
generalized structures
= generalized parses
= (generalized) macro-rules
= rule-chunks
application on new examples

An example of EBL using a perfect domain theory is a program that learns to play chess by being shown examples. A specific chess position that contains an important feature, say, "Forced loss of black queen in two moves," includes many irrelevant features, such as the specific scattering of pawns on the board. EBL can take a single training example and determine what the relevant features are in order to form a generalization.

<u>Learning by Analogy:</u>

Reasoning by analogy generally involves abstracting details from a a particular set of problems and resolving structural similarities between previously distinct problems. Analogical reasoning refers to this process of recognition and then applying the solution from the known problem to the new problem. Such a technique is often identified as *case-based reasoning*. Analogical learning generally involves developing a set of mappings between features of two instances.



The question in above figure represents some known aspects of a new case, which has unknown aspects to be determined. In deduction, the known aspects are compared (by a version of structure mapping called *unification*) with the premises of some implication. Then the unknown aspects, which answer the question, are derived from the conclusion of the implication. In analogy, the known aspects of the new case are compared with the corresponding aspects of the older cases. The case that gives the best match may be assumed as the best source of evidence for estimating the unknown aspects of the new case. The other cases show alternative possibilities for those unknown aspects; the closer the agreement among the alternatives, the stronger the evidence for the conclusion.

1. **Retrieve:** Given a target problem, retrieve cases from memory that are relevant to solving it. A case consists of a problem, its solution, and, typically, annotations about how the solution was derived. For example, suppose Fred wants to prepare blueberry pancakes. Being a novice cook, the most relevant experience he can recall is one in which he successfully made plain pancakes. The procedure he followed for making the plain pancakes, together with justifications for decisions made along the way, constitutes Fred's retrieved case.
2. **Reuse:** Map the solution from the previous case to the target problem. This may involve adapting the solution as needed to fit the new situation. In the pancake example, Fred must adapt his retrieved solution to include the addition of blueberries.
3. **Revise:** Having mapped the previous solution to the target situation, test the new solution in the real world (or a simulation) and, if necessary, revise. Suppose Fred adapted his pancake solution by adding blueberries to the batter. After mixing, he discovers that the batter has turned blue – an undesired effect. This suggests the following revision: delay the addition of blueberries until after the batter has been ladled into the pan.
4. **Retain:** After the solution has been successfully adapted to the target problem, store the resulting experience as a new case in memory. Fred, accordingly, records his newfound procedure for making blueberry pancakes, thereby enriching his set of stored experiences, and better preparing him for future pancake-making demands.

Transformational Analogy:

Suppose you are asked to prove a theorem in plane geometry. You might look for a previous theorem that is very similar and copy its proof, making substitutions when necessary. The idea is to transform a solution to a previous problem in to solution for the current problem. The following figure shows this process,

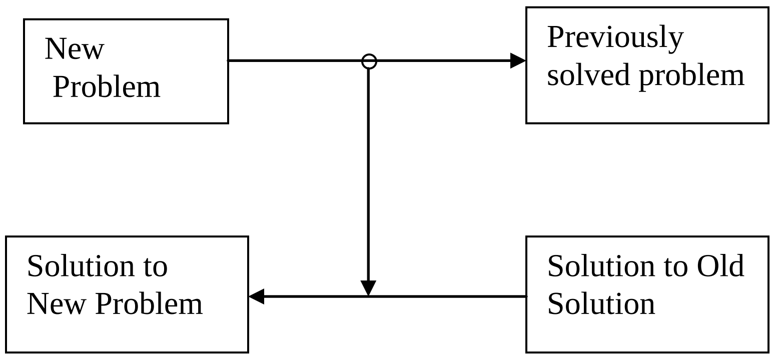


Fig: Transformational Analogy

Derivational Analogy:

Notice that transformational analogy does not look at how the old problem was solved, it only looks at the final solution. Often the twists and turns involved in solving an old problem are relevant to solving a new problem. The detailed history of problem solving episode is called derivation, Analogical reasoning that takes these histories into account is called derivational analogy.

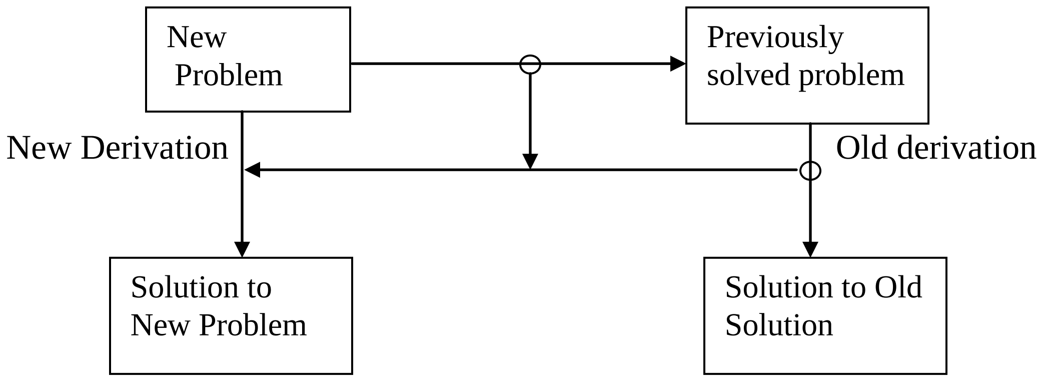


Fig: Derivational Analogy

Refer Book:- E. Rich, K. Knight, S. B. Nair, Tata MacGraw Hill (Pages 371-372)

Learning by Simulating Evolution:

Refer Book:- P. H. Winston, Artificial Intelligence, Addison Wesley. (Around page 220)

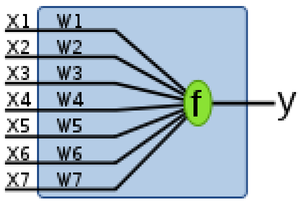
Learning by Training Perceptron:

Below is an example of a learning algorithm for a single-layer (no hidden-layer) perceptron. **For multilayer perceptrons, more complicated algorithms such as backpropagation must be used**. Or, methods such as the delta rule can be used if the function is non-linear and differentiable, although the one below will work as well.

The learning algorithm we demonstrate is the same across all the output neurons, therefore everything that follows is applied to a single neuron in isolation. We first define some variables:

- $x(j)$ denotes the j-th item in the n-dimensional input vector
- $w(j)$ denotes the j-th item in the weight vector
- $f(x)$ denotes the output from the neuron when presented with input x
- α is a constant where $0 < \alpha \leq 1$(learning rate)

Assume for the convenience that the bias term b is zero. An extra dimension $n + 1$ can be added to the input vectors x with $x(n + 1) = 1$, in which case $w(n + 1)$ replaces the bias term.



the appropriate weights are applied to the inputs, and the resulting weighted sum passed to a function which produces the output y

Let $D_m = \{(x_1, y_1), \ldots, (x_m, y_m)\}$be training set of m training examples, where x_i is the input vector to the perceptron and y_i is the desired output value of the perceptron for that input vector.

Learning algorithm steps:

1. Initialize weights and threshold.

- Set $w_i(t)$, $(1 \leq i \leq m)$ to be the weight i at time t, and ø to be the threshold value in the output node.
- Set w(0) to be -ø,the bias, and x(0) to be always 1.
- Set $w_i(1)$ to small random values, thus initialising the weights and threshold.

2. Present input and desired output

- Present input $x_0 = 1$ and $x_1, x_2, ..., x_m$ and desired output d(t)

3. Calculate the actual output

- $y(t) = fh[w_0(t) + w_1(t)x_1(t) + w_2(t)x_2(t) + ... + w_m(t)x_m(t)]$

4. Adapts weights

- $w_i(t + 1) = w_i(t) + \alpha[d(t) - y(t)]x_i(t)$, for $0 \leq i \leq m$.

Steps 3 and 4 are repeated until the iteration error is less than a user-specified error threshold or a predetermined number of iterations have been completed.