

MACHINE LEARNING

B. TECH (CSE)

3rd YEAR – 2nd SEM

UNIT-IV

Genetic Algorithms

Learning Sets of Rules

Reinforcement Learning

NOTES MATERIAL

RAVIKRISHNA B

DEPARTMENT OF CSE

VIGNAN INSTITUTE OF TECHNOLOGY & SCIENCE

DESHMUKHI

UNIT-IV :: GENETIC ALGORITHMS

Motivation behind Genetic Algorithms

Let's start with the famous quote by Charles Darwin:

“Survival for the Fittest”

--It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.

Entire concept of a genetic algorithm is based on the above line.

Genetic Algorithms – Motivation:

Write short notes on Dynamic programming:

Dynamic Programming (DP) is one of the techniques available to solve self-learning problems. It is widely used in areas such as operations research, economics and automatic control systems, among others. Artificial intelligence is the core application of DP since it mostly deals with learning information from a highly uncertain environment.

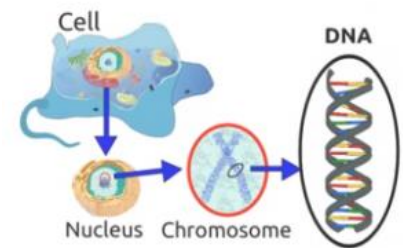
Biological Inspiration:

Cells are the basic building block of all living things.

Therefore, in each cell, there is the same set of chromosomes. Chromosome are basically the strings of DNA.

Traditionally, these chromosomes are represented in binary as strings of 0's and 1's.

A chromosome consists of genes, commonly referred as blocks of DNA, where each gene encodes a specific trait, for example hair color or eye color.



What is a Genetic Algorithm?

- Genetic Algorithm (GA) is a search based optimization technique based on the principles of Genetics and Natural Selection. It is frequently used to find optimal or near optimal solutions to difficult problems which otherwise would take a lifetime to solve.
- A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.
- It is frequently used to solve optimization problems, in research, and in machine learning.

How & when they developed?

- GAs were developed by John Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.
- In GAs, we have a **pool or a population of possible solutions** to the given problem.

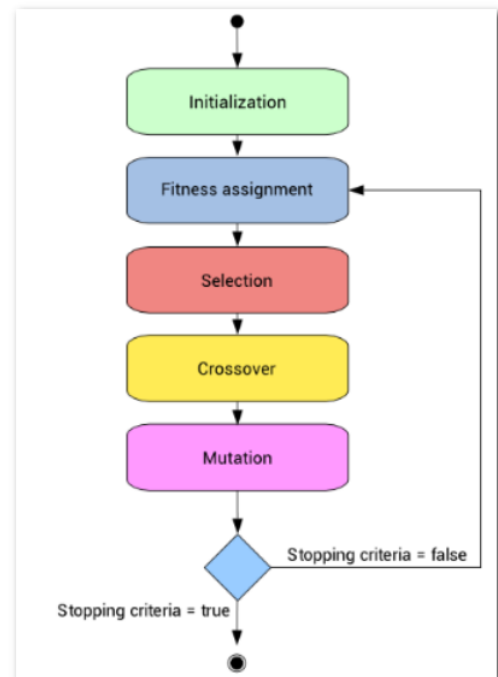
These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations.

- Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

How genetic algorithm actually works:

- Firstly, we defined our initial population as our countrymen.
- We defined a function to classify whether is a person is good or bad.
- Then we select good people for mating to produce their off-springs (progenitor: a biologically related ancestor)
- And finally, these off-springs replace the bad people from the population and this process repeats.
-

This is how genetic algorithm actually works, which basically tries to mimic the human evolution to some extent.



In GAs, we have a pool or a population of possible solutions to the given problem.

- These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations.
- Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more “fitter” individuals. This is in line with the Darwinian Theory of “Survival of the Fittest”.
- In this way we keep “evolving” better individuals or solutions over generations, till we reach a stopping criterion.
- Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

Advantages of Gas

(different factors behind the importance for genetic algorithms)

GAs have various advantages which have made them immensely popular. These include –

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of “good” solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Benefits of GA ‘s:

Genetic algorithms differ from traditional search and optimization methods in four significant points:

- Genetic algorithms search parallel from a population of points. Therefore, it has the ability to avoid being trapped in local optimal solution like traditional methods, which search from a single point.
- Genetic algorithms use probabilistic selection rules, not deterministic ones.
- Genetic algorithms work on the Chromosome, which is encoded version of potential solutions’ parameters, rather the parameters themselves.
- Genetic algorithms use fitness score, which is obtained from objective functions, without other derivative or auxiliary information

Limitations of GA ‘s:

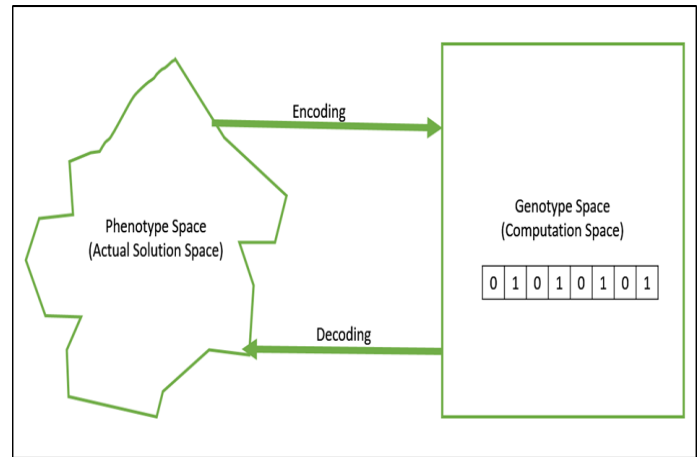
Like any technique, GAs also suffer from a few limitations. These include –

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

Genotype – Genotype is the population in the computation space. In the computation space, the solutions are represented in a way which can be easily understood and manipulated using a computing system.

Phenotype – Phenotype is the population in the actual real world solution space in which solutions are represented in a way they are represented in real world situations.

Decoding and Encoding – For simple problems, the phenotype and genotype spaces are the same. However, in most of the cases, the phenotype and genotype spaces are different. Decoding is a process of transforming a solution from the genotype to the phenotype space, while encoding is a process of transforming from the phenotype to genotype space. Decoding should be fast as it is carried out repeatedly in a GA during the fitness value calculation.



Fitness Function – A fitness function simply defined is a function which takes the solution as input and produces the suitability of the solution as the output. In some cases, the fitness function and the objective function may be the same, while in others it might be different based on the problem.

Genetic Operators – These alter the genetic composition of the offspring. These include crossover, mutation, selection, etc.

Basic Terminology:

Before beginning a discussion on Genetic Algorithms, it is essential to be familiar with some basic terminology which will be used throughout this tutorial.

- **Population** – It is a subset of all the possible (encoded) solutions to the given problem. The population for a GA is analogous to the population for human beings except that instead of human beings, we have Candidate Solutions representing human beings.
- **Chromosomes** – A chromosome is one such solution to the given problem.
- **Gene** – A gene is one element position of a chromosome.
- **Allele** – It is the value a gene takes for a particular chromosome.



Genetic Algorithms - Population

Population is a subset of solutions in the current generation. It can also be defined as a set of chromosomes. There are several things to be kept in mind when dealing with GA population –

- The diversity of the population should be maintained otherwise it might lead to premature convergence.

- The population size should not be kept very large as it can cause a GA to slow down, while a smaller population might not be enough for a good mating pool. Therefore, an optimal population size needs to be decided by trial and error.

The population is usually defined as a two-dimensional array of – **size population, size x, chromosome size**.

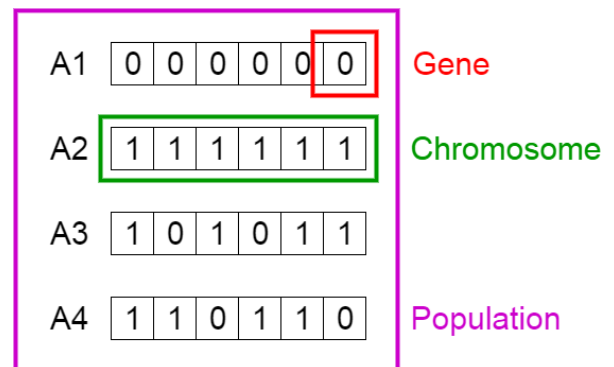
Five phases are considered in a genetic algorithm.

1. **Initial population**
2. **Fitness function**
3. **Selection**
4. **Crossover**
5. **Mutation**

Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution).



In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

Population Initialization

There are two primary methods to initialize a population in a GA.

They are –

Random Initialization – Populate the initial population with completely random solutions.

Heuristic initialization – Populate the initial population using a known heuristic for the problem.

- It has been observed that the entire population should not be initialized using a heuristic, as it can result in the population having similar solutions and very little diversity.
- It has been experimentally observed that the random solutions are the ones to drive the population to optimality.

Fitness Function

Fitness Function (also known as the Evaluation Function) evaluates how close a given solution is to the optimum solution of the desired problem. It determines how fit a solution is.

Why we use Fitness Functions?

In genetic algorithms, each solution is generally represented as a string of binary numbers, known as a chromosome. We have to test these solutions and come up with the best set of solutions to solve a given problem. Each solution, therefore, needs to be awarded a score, to indicate how close it came to meeting the overall specification of the desired solution. This score is generated by applying the fitness function to the test, or results obtained from the tested solution.

Selection

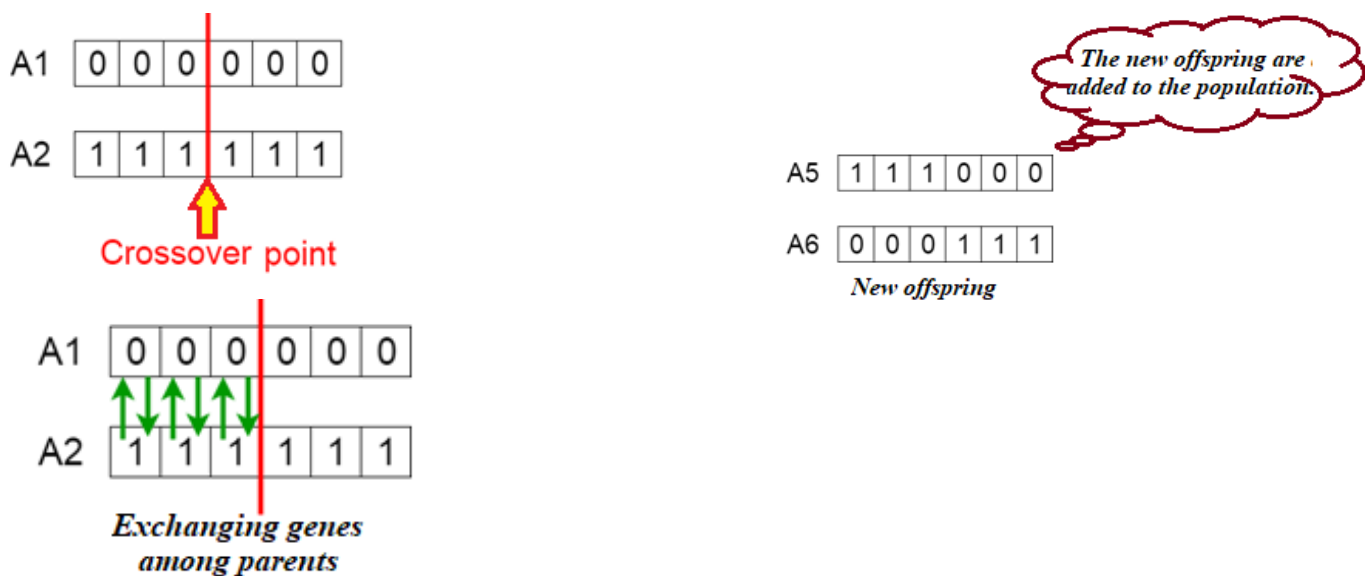
The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.

Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.

For example, consider the crossover point to be 3 as shown below.



Crossover point:

Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.

Mutation

- In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.
- Mutation occurs to maintain diversity within the population and prevent premature convergence.

Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

Pseudocode:

1. START
2. Generate the initial population
3. Compute fitness
4. REPEAT
5. Selection
6. Crossover
7. Mutation
8. Compute fitness
9. UNTIL population has converged
10. STOP

Application/Example/Case-study:

Here, to make things easier, let us understand it by the famous Knapsack problem. Let's suppose we are going to spend a month in the wilderness. Only thing you are carrying is the backpack which can hold a maximum weight of 30 kg. Now you have different survival items, each having its own "Survival Points" (which are given for each item in the table). So, your objective is maximizing the survival points.

Here is the table giving details about each item.

Let's say, you are going to spend a month in the wilderness. Only thing you are carrying is the backpack which can hold a maximum weight of 30 kg. Now you have different survival items, each having its own "Survival Points"

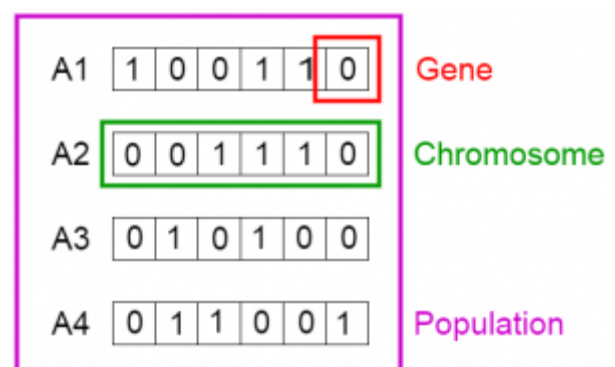
ITEM	WEIGHT	SURVIVAL POINTS
SLEEPING BAG	15	15
ROPE	3	7
POCKET KNIFE	2	10
TORCH	5	5
BOTTLE	9	8
GLUCOSE	20	17

(which are given for each item in the table). So, your objective is maximize the survival points. Here is the table giving details about each item.

1. Initialization

To solve this problem using genetic algorithm, our first step would be defining our population. So our population will contain individuals, each having their own set of chromosomes.

We know that, chromosomes are binary strings, where for this problem 1 would mean that the following item is taken and 0 meaning that it is dropped.



This set of chromosome is considered as our initial population.

2. Fitness Function:

Let us calculate fitness points for our first two chromosomes.

For A1 chromosome [100110],

ITEMS	WEIGHT	SURVIVAL POINTS
Sleeping bag	15	15
Torch	5	5
Bottle	9	8
TOTAL	29	28

Similarly for A2 chromosome [001110],

ITEMS	WEIGHT	SURVIVAL POINTS
Pocket Knife	2	10
Torch	5	5
Bottle	9	8
TOTAL	16	23

So, for this problem, our chromosome will be considered as more fit when it contains more survival points.

Therefore chromosome 1 is more fit than chromosome 2.

3 Selection:

Now, we can select fit chromosomes from our population which can mate and create their off-springs.

General thought is that we should select the fit chromosomes and allow them to produce off-springs. But that would lead to chromosomes that are more close to one another in a few next generation, and therefore less diversity.

Therefore, we generally use Roulette Wheel Selection method.

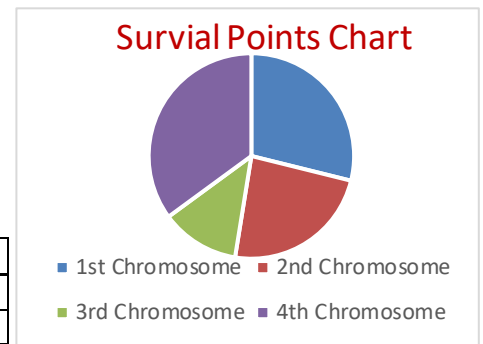
Don't be afraid of name, just take a look at the image below.



I suppose we all have seen this, either in real or in movies. So, let's build our roulette wheel.

Consider a wheel, and let's divide that into m divisions, where m is the number of chromosomes in our populations. The area occupied by each chromosome will be proportional to its fitness value.

	Survival Points	Percentage
Chromosome1	28	28.9% (28/97)
Chromosome2	23	23.7%
Chromosome3	12	12.4%
Chromosome4	34	35.1%
	TOTAL: 97 (TOTAL SURVIVAL PNTS)	



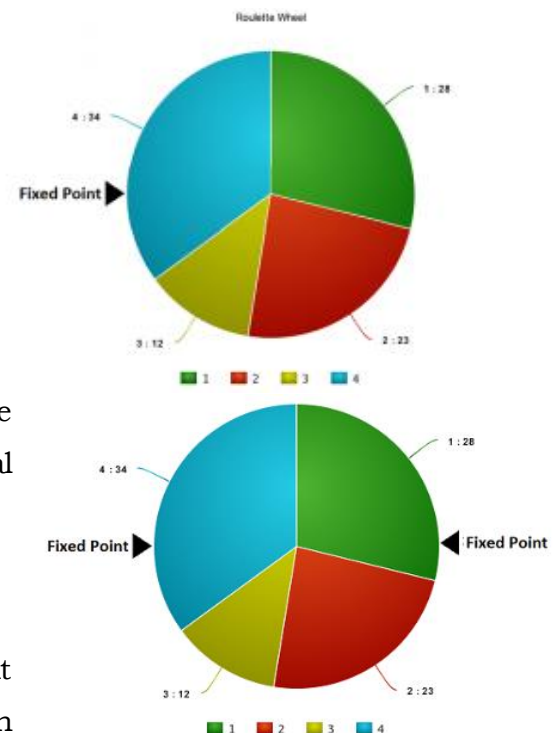
Based on these values, let us

create our roulette wheel.

So, now this wheel is rotated and the region of wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.

Sometimes we mark two fixed point as shown in the figure below.

So, in this method we can get both our parents in one go. This method is known as Stochastic Universal Selection method.



Crossover

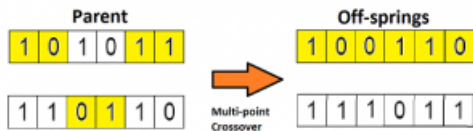
So in this previous step, we have selected parent chromosomes that will produce off-springs. So in biological terms, crossover is nothing but reproduction.

So let us find the crossover of chromosome 1 and 4, which were selected in the previous step. Take a look at the image below.



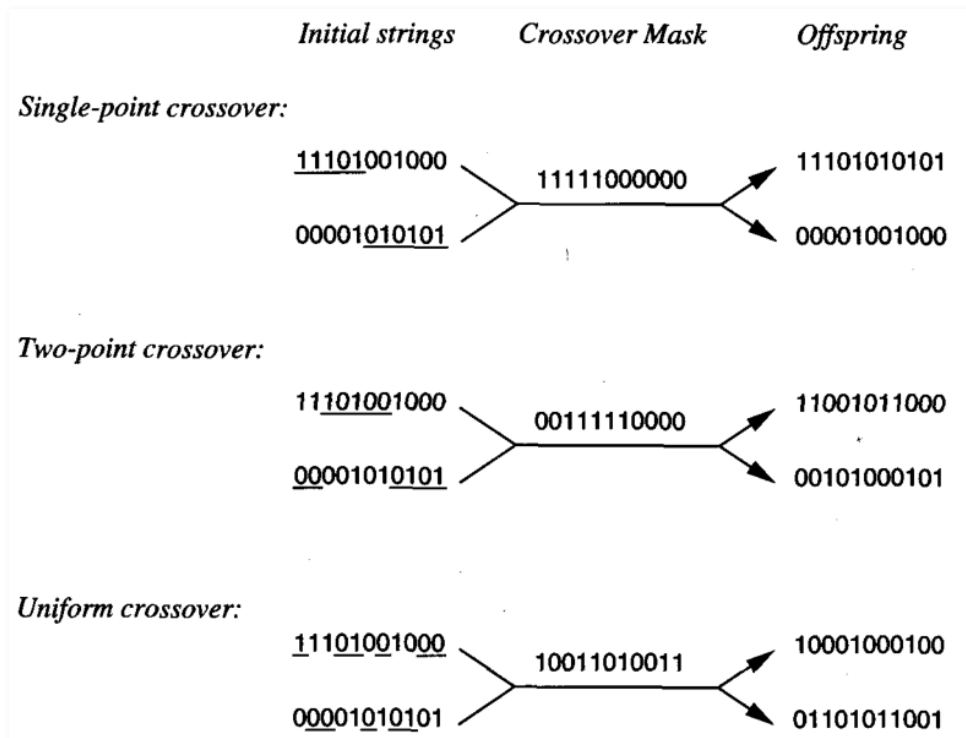
This is the most basic form of crossover, known as one point crossover. Here we select a random crossover point and the tails of both the chromosomes are swapped to produce a new off-springs.

If you take two crossover point, then it will called as multi point crossover which is as shown below.



crossover operator:

The crossover operator produces two new offspring from two parent strings, by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the crossover mask. To illustrate, consider the single-point crossover operator at the top of Table.



5 Mutation

Now if you think in the biological sense, are the children produced have the same traits as their parents? The answer is NO. During their growth, there is some change in the genes of children which makes them different from its parents.

This process is known as mutation, which may be defined as a random tweak in the chromosome, which also promotes the idea of diversity in the population.

A simple method of mutation is shown in the image below.



So the entire process is summarize as shown in the figure.



The off-springs thus produced are again validated using our fitness function, and if considered fit then will replace the less fit chromosomes from the population.

But the question is how we will get to know that we have reached our best possible solution? So basically, there are different termination conditions, which are listed below:

1. There is no improvement in the population for over x iterations.
2. We have already predefined an absolute number of generations for our algorithm.
3. When our fitness function has reached a predefined value.

Source: <https://www.analyticavidhya.com/blog/2017/07/introduction-to-genetic-algorithm/>

Fitness Function and Selection:

- The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. Often other criteria may be included as well, such as the complexity or generality of the rule. More generally, when the bit-string hypothesis is interpreted as a complex procedure, the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.
- In our prototypical GA shown, the probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population. This method is sometimes called fitness proportionate selection, or roulette wheel selection. Other methods for using fitness to select hypotheses have also been proposed.

Prototypical Genetic Algorithm:

GA(Fitness, Fitness_threshold, p , r , m)

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitness_threshold: A threshold specifying the termination criterion.

p : The number of hypotheses to be included in the population.

r : The fraction of the population to be replaced by Crossover at each step.

m : The mutation rate.

- **Initialize population:** $P \leftarrow$ Generate p hypotheses at random

- **Evaluate:** For each h in P , compute $Fitness(h)$

- **While** $[\max_h Fitness(h)] < Fitness_threshold$ **do**

Create a new generation, P_s :

1. **Select:** Probabilistically select $(1 - r)p$ members of P to add to P_s . The probability $Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^p Fitness(h_j)}$$

2. **Crossover:** Probabilistically select $\frac{rp}{2}$ pairs of hypotheses from P , according to $Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s .

3. **Mutate:** Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.

4. **Update:** $P \leftarrow P_s$.

5. **Evaluate:** for each h in P , compute $Fitness(h)$

- **Return** the hypothesis from P that has the highest fitness.

Hypothesis space search:

- A population containing p hypotheses is maintained. On each iteration, the successor population P_s is formed by probabilistically selecting current hypotheses according to their fitness and by adding new hypotheses.
- New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered.
- Typical crossover and mutation operators are defined in a subsequent table.
- The inputs to this algorithm include the fitness function for ranking candidate hypotheses, a threshold defining an acceptable level of fitness for terminating the algorithm, the size of the population to be maintained, and parameters that determine how successor populations are to be generated: the fraction of the population to be replaced at each generation and the mutation rate.
- Notice in this algorithm each iteration through the main loop produces a new generation of hypotheses based on the current population. First, a certain number of hypotheses from the current population are selected for inclusion in the next generation. These are selected probabilistically, where the probability of selecting hypothesis h_i is given by

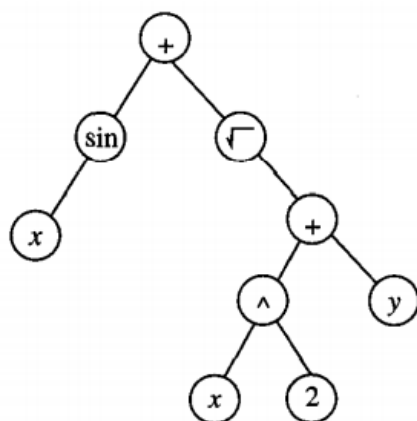
$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^p \text{Fitness}(h_j)} = \frac{f(h_i)}{n * \bar{f}(t)}$$

- Thus, the probability that a hypothesis will be selected is proportional to its own fitness and is inversely proportional to the fitness of the other competing hypotheses in the current population.
- Once these members of the current generation have been selected for inclusion in the next generation population, additional members are generated using a crossover operation.
- Crossover, defined in detail in the next section, takes two parent hypotheses from the current generation and creates two offspring hypotheses by recombining portions of both parents. The parent hypotheses are chosen probabilistically from the current population, again using the probability function given by Equation.
- After new members have been created by this crossover operation, the new generation population now contains the desired number of members.

- At this point, a certain fraction m of these members are chosen at random, and random mutations are performed to alter these members.
- This GA algorithm thus performs a randomized, parallel beam search for hypotheses that perform well according to the fitness function. In the following subsections, we describe in more detail the representation of hypotheses and genetic operators used in this algorithm.

GENETIC PROGRAMMING

- Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings. Koza (1992) describes the basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP.
- Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes.
- For example, Figure 9.1 illustrates this tree representation for the function $\sin(x) + \sqrt{x^2 + y}$. To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., \sin , \cos , J , $+$, $-$, exponential), as well as the terminals (e.g., x , y , constants such as 2). The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives.
- As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and mutation. The fitness of a given individual program in the population is typically determined by executing the program on a set of training data.



Crossover operations are performed by replacing a randomly chosen subtree of one parent.

Fig: Tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

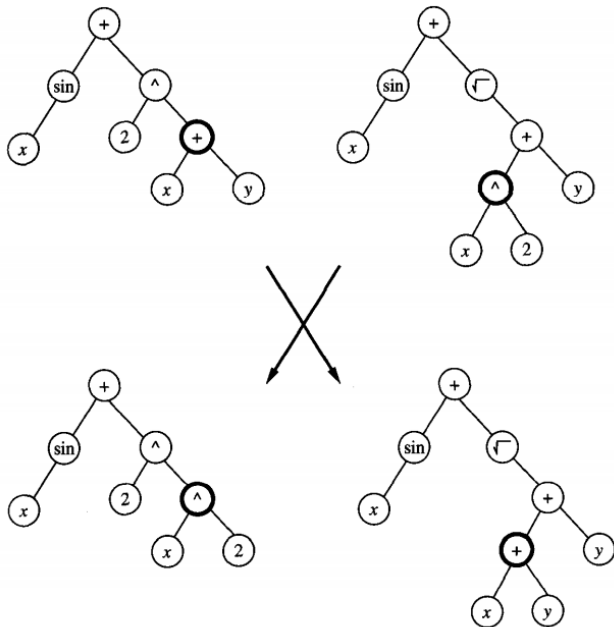


Fig: Crossover operation applied to two parent program trees (top).

Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children's trees (bottom).

Models of evolution and Learning:

- In many natural systems, individual organisms learn to adapt significantly during their lifetime. At the same time, biological and social processes allow their species to adapt over a time frame of many generations.
- Larnarck was a scientist who, in the late nineteenth century, proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime.
- In particular, he proposed that experiences of a single organism directly affected the genetic makeup of their offspring: If an individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait.
- This is an attractive conjecture, because it would presumably allow for more efficient evolutionary progress than a generate-and-test process (like that of GAS and GPs) that ignores the experience gained during an individual's lifetime.
- Baldwin Effect: Although Lamarckian evolution is not an accepted model of biological evolution, other mechanisms have been suggested by which individual learning can alter the course of evolution. One such mechanism is called the Baldwin effect, after J. M. Baldwin (1896), who first suggested the idea. The Baldwin effect is based on the following observations:
 - If a species is evolving in a changing environment, there will be evolutionary pressure to favor individuals with the capability to learn during their lifetime. For example, if a new predator appears in the environment, then individuals capable of learning to avoid the predator will be more successful than individuals who cannot learn.

- In effect, the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness.
- In contrast, non-learning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.
- Those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits. As a result, these individuals can support a more diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code.
- This more diverse gene pool can, in turn, support more rapid evolutionary adaptation. Thus, the ability of individuals to learn can have an indirect accelerating effect on the rate of evolutionary adaptation for the entire population.

Parallelizing Genetic Algorithms:

- GAs' are naturally suited to parallel implementation, and a number of approaches to parallelization have been explored. Coarse grain approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called demes.
- Each deme is assigned to a different computational node, and a standard GA search is performed at each node.
- Communication and cross-fertilization between demes occur on a less frequent basis than within demes.
- Transfer between demes occurs by a migration process, in which individuals from one deme are copied or transferred to other demes. This process is modeled after the kind of cross-fertilization that might occur between physically separated subpopulations of biological species.
- One benefit of such approaches is that it reduces the crowding problem often encountered in nonparallel GAS, in which the system falls into a local optimum due to the early appearance of a genotype that comes to dominate the entire population.
- In contrast to coarse-grained parallel implementations of GAS, fine-grained implementations typically assign one processor per individual in the population. Recombination then takes place among neighboring individuals.

LEARNING SETS OF RULES

Introduction:

- One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules.
- In this chapter we explore algorithms for learning such sets of rules.
- One important special case involves learning sets of rules containing variables, called first-order Horn clauses.
- Sets of first-order Horn clauses can be interpreted as programs in the logic programming language PROLOG, learning them is often called inductive logic programming (ILP).
- One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules. One important special case involves learning sets of rules containing variables, called first-order Horn clauses.
- Because sets of first-order Horn clauses can be interpreted as programs in the logic programming language PROLOG, learning them is often called inductive logic programming (ILP).
- As an example of first-order rule sets, consider the following two rules that jointly describe the target concept Ancestor. Here we use the predicate $Parent(x, y)$ to indicate that y is the mother or father of x, and the predicate $Ancestor(x, y)$ to indicate that y is an ancestor of x related by an arbitrary number of family generations.

• *IF Parent (x, y) THEN Ancestor(x,y)*

• *IF Parent(x, z) ^ Ancestor(z, y) THEN Ancestor(x, y)*

- these two rules compactly describe a recursive function that would be very difficult to represent using a decision tree or other propositional representation. One way to see the representational power of first-order rules is to consider the general purpose programming language PROLOG. In PROLOG, programs are sets of first-order rules such as the two shown above (rules of this form are also called Horn clauses).

SEQUENTIAL COVERING ALGORITHMS

- we consider a family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process. Such algorithms are called sequential covering algorithms.
- To elaborate, imagine we have a subroutine LEARN-ONE-RULE that accepts a set of

positive and negative training examples as input, then outputs a single rule that covers many of the positive examples and few of the negative examples.

- Given this LEARN-ONE-RULE subroutine for learning a single rule, one obvious approach to learning a set of rules is to invoke LEARN-ONE-RULE on all the available training examples, remove any positive examples covered by the rule it learns, then invoke it again to learn a second rule based on the remaining training examples.
- This procedure can be iterated as many times as desired to learn a disjunctive set of rules that together cover any desired fraction of the positive examples. This is called a sequential covering algorithm because it sequentially learns a set of rules that together cover the full set of positive examples.
- The final set of rules can then be sorted so that more accurate rules will be considered first when a new instance must be classified.
- This sequential covering algorithm is one of the most widespread approaches to learning disjunctive sets of rules.
- It reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned. Because it performs a greedy search, formulating a sequence of rules without backtracking, it is not guaranteed to find the smallest or best set of rules that cover the training examples.

SEQUENTIAL-COVERING(*Target_attribute, Attributes, Examples, Threshold*)

- *Learned_rules* $\leftarrow \{\}$
- *Rule* \leftarrow LEARN-ONE-RULE(*Target_attribute, Attributes, Examples*)
- while PERFORMANCE(*Rule, Examples*) > *Threshold*, do
 - *Learned_rules* \leftarrow *Learned_rules* + *Rule*
 - *Examples* \leftarrow *Examples* – {examples correctly classified by *Rule*}
 - *Rule* \leftarrow LEARN-ONE-RULE(*Target_attribute, Attributes, Examples*)
- *Learned_rules* \leftarrow sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

- we design LEARN-ONE-RULE to meet the needs of the sequential covering algorithm. We require an algorithm that can formulate a single rule with high accuracy, but that need not cover all of the positive examples.

Exploring algorithm:

The Sequential Learning algorithm takes care of to some extent, the low coverage problem in the Learn-One-Rule algorithm covering all the rules in a sequential manner.

The algorithm involves a set of 'ordered rules' or 'list of decisions' to be made.

Step 1 – create an empty decision list, 'R'.

Step 2 – 'Learn-One-Rule' Algorithm

It extracts the best rule for a particular class 'y', where a rule is defined as:

General Form of Rule

$r_i : (\text{Condition}_1, \text{condition}_2, \dots, \text{condition}_i) \rightarrow y_i$

In the beginning,

Step 2.a – if all training examples \in class 'y', then it's classified as **positive example**.

Step 2.b – else if all training examples \notin class 'y', then it's classified as **negative example**.

Step 3 – The rule becomes '**desirable**' when it covers a majority of the positive examples.

Step 4 – When this rule is obtained, delete all the training data associated with that rule.

(i.e. when the rule is applied to the dataset, it covers most of the training data, and has to be removed)

Step 5 – The new rule is added to the bottom of decision list, 'R'. (Fig.3)

Below, is a visual representation describing the working of the algorithm.

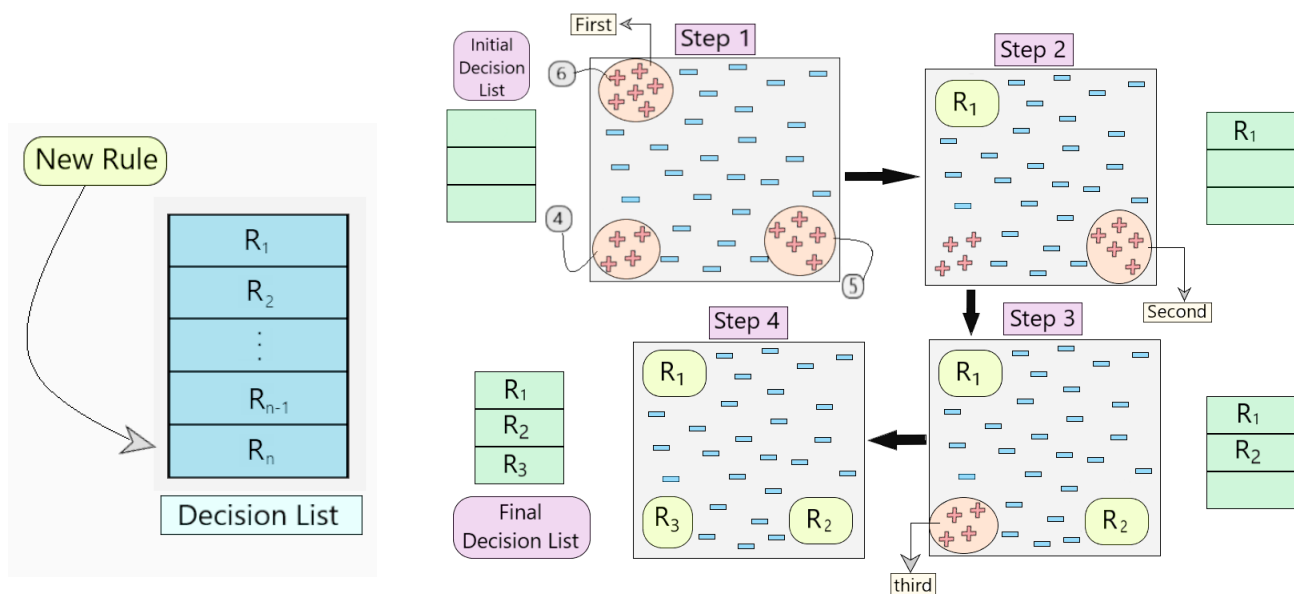


Fig : Decision List 'R'

Fig : Visual Representation of working of the algorithm

- Let us understand step by step how the algorithm is working in the example shown in Fig.
- First, we created an empty decision list. During Step 1, we see that there are three sets of positive examples present in the dataset. So, as per the algorithm, we consider the one with maximum no of positive example.

- Once we cover these 6 positive examples, we get our first rule R_1 , which is then pushed into the decision list and those positive examples are removed from the dataset.
- Now, we take the next majority of positive examples and follow the same process until we get rule R_2 .
- In the end, we obtain our final decision list with all the desirable rules.

Sequential Learning is a powerful algorithm for generating rule-based classifiers in Machine Learning. It uses 'Learn-One-Rule' algorithm as its base to learn a sequence of disjunctive rules. For doubts/queries regarding the algorithm, comment below.

Rule-Based Classifier – Machine Learning

Rule-based classifiers are just another type of classifier which makes the class decision depending by using various “if..else” rules. These rules are easily interpretable and thus these classifiers are generally used to generate descriptive models. The condition used with “if” is called the **antecedent** and the predicted class of each rule is called the **consequent**.

Properties of rule-based classifiers:

- **Coverage:** The percentage of records which satisfy the antecedent conditions of a particular rule.
- The rules generated by the rule-based classifiers are generally not mutually exclusive, i.e. many rules can cover the same record.
- The rules generated by the rule-based classifiers may not be exhaustive, i.e. there may be some records which are not covered by any of the rules.
- The decision boundaries created by them is linear, but these can be much more complex than the decision tree because the many rules are triggered for the same record.

An obvious question, which comes into the mind after knowing that the rules are not mutually exclusive is that how would the class be decided in case different rules with different consequent cover the record.

There are two solutions to the above problem:

- Either rules can be **ordered**, i.e. the class corresponding to the highest priority rule triggered is taken as the final class.
- Otherwise, we can assign **votes** for each class depending on some their weights, i.e. the rules remain unordered.

LEARN-ONE-RULE ALGORITHM

Learn-One-Rule:

This method is used in the sequential learning algorithm for learning the rules. It returns a single rule that covers at least some examples. However, what makes it really powerful is its ability to create relations among the attributes given, hence covering a larger hypothesis space.

Learn-One-Rule Algorithm

The Learn-One-Rule algorithm follows a greedy searching paradigm where it searches for the rules with high accuracy but its coverage is very low. It classifies all the positive examples for a particular instance. It returns a single rule that covers some examples.

LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*, *k*)

Returns a single rule that covers some of the Examples. Conducts a general-to-specific greedy beam search for the best rule, guided by the PERFORMANCE metric.

- Initialize *Best_hypothesis* to the most general hypothesis \emptyset
- Initialize *Candidate_hypotheses* to the set {*Best_hypothesis*}
- While *Candidate_hypotheses* is not empty, Do
 1. Generate the next more specific *candidate_hypotheses*
 - *All_constraints* \leftarrow the set of all constraints of the form ($a = v$), where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
 - *New_candidate_hypotheses* \leftarrow
 - for each h in *Candidate_hypotheses*,
 - for each c in *All_constraints*,
 - create a specialization of h by adding the constraint c
 - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
 2. Update *Best_hypothesis*
 - For all h in *New_candidate_hypotheses* do
 - If ($\text{PERFORMANCE}(h, \text{Examples}, \text{Target_attribute}) > \text{PERFORMANCE}(\text{Best_hypothesis}, \text{Examples}, \text{Target_attribute})$)
 - Then *Best_hypothesis* $\leftarrow h$
 3. Update *Candidate_hypotheses*
 - *Candidate_hypotheses* \leftarrow the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form

“IF *Best_hypothesis* THEN *prediction*”

where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h , *Examples*, *Target_attribute*)

- *h_examples* \leftarrow the subset of *Examples* that match h
 - return $-\text{Entropy}(h_examples)$, where entropy is with respect to *Target_attribute*
-

Simpler Form of**Learn-One-Rule (target_attribute, attributes, examples, k):**

Pos = positive examples

Neg = negative examples

best-hypothesis = the most general hypothesis

candidate-hypothesis = {best-hypothesis}

while candidate-hypothesis:

//Generate the next more specific candidate-hypothesis

constraints_list = all constraints in the form "attribute=value"

new-candidate-hypothesis = all specializations of candidate-

hypothesis by adding all-constraints

remove all duplicates/inconsistent hypothesis from new-candidate-

hypothesis.

*//Update best-hypothesis*best_hypothesis = $\text{argmax}_{(h \in CH_s)} \text{Performance}(h, \text{examples}, \text{target_attribute})$ *//Update candidate-hypothesis*candidate-hypothesis = the k best from new-candidate-hypothesis
according to Performance.prediction = most frequent value of target_attribute from examples that
match best-hypothesis

IF best_hypothesis:

return prediction

It involves a PERFORMANCE method that calculates the performance of each candidate hypothesis.
(i.e. how well the hypothesis matches the given set of examples in the training data.

Performance (NewRule, h):

h-examples = the set of rules that match h

return (h-examples)

It starts with the most general rule precondition, then greedily adds the variable that most improves performance measured over the training examples.

For example:

IF Mother (y, x) and Female(y),
THEN Daughter (x, y).

Here, any person can be associated
with the variables x and y

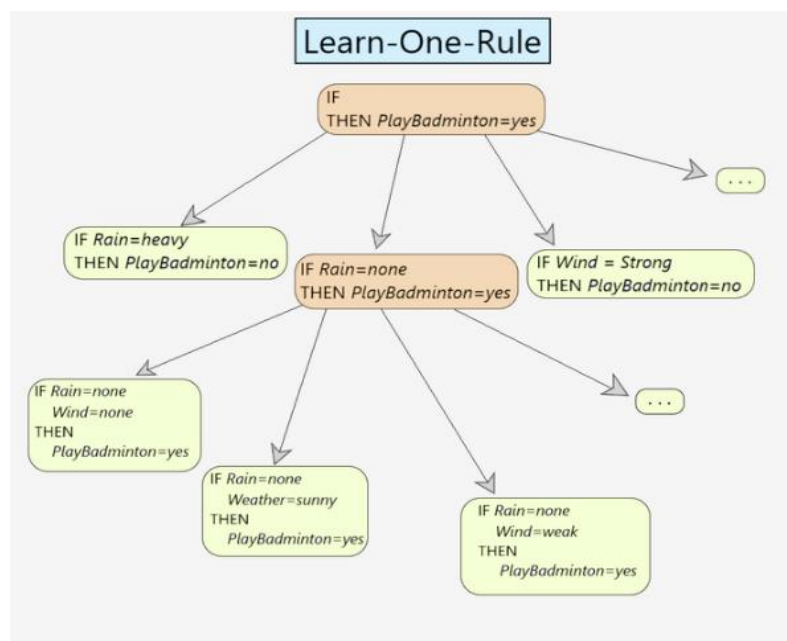


Figure:

Learn-One-Rule Example

Learn-One-Rule Example

Let us understand the working of the algorithm using an example:

Day	Weather	Temp	Wind	Rain	PlayBadminton
D1	Sunny	Hot	Weak	Heavy	No
D2	Sunny	Hot	Strong	Heavy	No
D3	Overcast	Hot	Weak	Heavy	No
D4	Snowy	Cold	Weak	Light/None	Yes
D5	Snowy	Cold	Weak	Light/None	Yes
D6	Snowy	Cold	Strong	Light/None	Yes
D7	Overcast	Mild	Strong	Heavy	No
D8	Sunny	Hot	Weak	Light/None	Yes

Step 1 - best_hypothesis = IF h THEN PlayBadminton(x) = Yes

Step 2 - candidate-hypothesis = {best-hypothesis}

Step 3 - constraints_list = {Weather(x)=Sunny, Temp(x)=Hot, Wind(x)=Weak,}

Step 4 - new-candidate-hypothesis = {IF Weather=Sunny THEN PlayBadminton=YES, IF Weather=Overcast THEN PlayBadminton=YES, ...}

Step 5 - best-hypothesis = IF Weather=Sunny THEN PlayBadminton=YES

Step 6 - candidate-hypothesis = {IF Weather=Sunny THEN PlayBadminton=YES, IF Weather=Sunny THEN PlayBadminton=YES...}

Step 7 - Go to Step 2 and keep doing it till the best-hypothesis is obtained.

- Sequential Learning Algorithm uses this algorithm, improving on it and increasing the coverage of the hypothesis space. It can be modified to accept an argument that specifies the target value of interest.

LEARNING RULE SETS: SUMMARY:

- The SEQUENTIAL-COVERING algorithm described above and the decision tree learning algorithms, suggest a variety of possible methods for learning sets of rules.
- First, sequential covering algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples. In contrast, decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. We might, therefore, call algorithms such as ID3 simultaneous covering algorithms, in contrast to sequential covering algorithms. The key difference occurs in the choice made at the most primitive step in the search. At each search step ID3 chooses among alternative attributes by comparing the partitions of the data they generate.
- To learn a set of n rules, each containing k attribute-value tests in their preconditions, sequential covering algorithms will perform n. k primitive search steps, making an

independent decision to select each precondition of each rule. In contrast, simultaneous covering algorithms will make many fewer independent choices, because each choice of a decision node in the decision tree corresponds to choosing the precondition for the multiple rules associated with that node. In other words, if the decision node tests an attribute that has m possible values, the choice of the decision node corresponds to choosing a precondition for each of the m corresponding rules.

- A second dimension along which approaches vary is the direction of the search in LEARN-ONE-RULE. In the algorithm described above, the search is from general to specific hypotheses.
- One advantage of general to specific search in this case is that there is a single maximally general hypothesis from which to begin the search, whereas there are very many specific hypotheses in most hypothesis spaces (i.e., one for each possible instance). Given many maximally specific hypotheses, it is unclear which to select as the starting point of the search.
- A third dimension is whether the LEARN-ONE-RULE search is a generate then test search through the syntactically legal hypotheses, as it is in our suggested implementation, or whether it is example-driven so that individual training examples constrain the generation of hypotheses. Prototypical example-driven search algorithms include the FIND-S and CANDIDATE-ELIMINATION algorithms.
- A fourth dimension is whether and how rules are post-pruned. As in decision tree learning, it is possible for LEARN-ONE-RULE to formulate rules that perform very well on the training data, but less well on subsequent data. As in decision tree learning, one way to address this issue is to post-prune each rule after it is learned from the training data.
- A final dimension is the particular definition of rule PERFORMANCE used to guide the search in LEARN-ONE-RULE. Various evaluation functions have been used.
- Some common evaluation functions include:
- **Relative frequency:** Let n denote the number of examples the rule matches and let n_c denote the number of these that it classifies correctly. The relative frequency estimate of rule performance is n_c/n .
- **m-estimate of accuracy:** This accuracy estimate is biased toward the default accuracy expected of the rule. It is often preferred when data is scarce and the rule must be evaluated based on few examples.
- As above, let n and n_c denote the number of examples matched and correctly predicted by the rule. Let p be the prior probability that a randomly drawn example from the entire data set will have the classification assigned by the rule (e.g., if 12 out of 100 examples have the value predicted by the rule, then $p = .12$). Finally, let m be the weight, or equivalent number of examples for weighting this prior p . The m-estimate of rule accuracy is

$$\frac{n_c + mp}{n + m}$$

- Entropy. This is the measure used by the PERFORMANCE subroutine in the algorithm. Let S be the set of examples that match the rule preconditions. Entropy measures the uniformity of the target function values for this set of examples.
- We take the negative of the entropy so that better rules will have higher scores.

$$-Entropy(S) = \sum_{i=1}^c p_i \log_2 p_i$$

LEARNING FIRST-ORDER RULES

- we consider learning rules that contain variables-in particular, learning first-order Horn theories. Our motivation for considering such rules is that they are much more expressive than propositional rules.
- Inductive learning of first-order rules or theories is often referred to as inductive logic programming (or LP for short), because this process can be viewed as automatically inferring PROLOG programs from examples.
- PROLOG is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses.

First-Order Horn Clauses:

To see the advantages of first-order representations over propositional (variable-free) representations, consider the task of learning the simple target concept Daughter (x, y), defined over pairs of people x and y. The value of Daughter(x, y) is True when x is the daughter of y, and False otherwise. Suppose each person in the data is described by the attributes Name, Mother, Father, Male, Female. Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute Daughter. For example, the following is a positive example in which Sharon is the daughter of Bob:

$(Name_1 = Sharon, \quad Mother_1 = Louise, \quad Father_1 = Bob,$
 $Male_1 = False, \quad Female_1 = True,$
 $Name_2 = Bob, \quad Mother_2 = Nora, \quad Father_2 = Victor,$
 $Male_2 = True, \quad Female_2 = False, \quad Daughter_{1,2} = True)$

where the subscript on each attribute name indicates which of the two persons is being described. Now if we were to collect a number of such training examples for the target concept $Daughter_{1,2}$ and provide them to a propositional rule learner such as CN2 or C4.5, the result would be a collection of very specific rules such as

IF $(Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True)$
 THEN $Daughter_{1,2} = True$

Although it is correct, this rule is so specific that it will rarely, if ever, be useful in classifying future pairs of people. The problem is that propositional representations offer no general way to describe the essential *relations* among the values of the attributes. In contrast, a program using first-order representations could learn the following general rule:

IF $Father(y, x) \wedge Female(y),$ THEN $Daughter(x, y)$

where x and y are variables that can be bound to any person.

First-order Horn clauses may also refer to variables in the preconditions that do not occur in the postconditions. For example, one rule for *GrandDaughter* might be

IF $Father(y, z) \wedge Mother(z, x) \wedge Female(y)$
 THEN $GrandDaughter(x, y)$

Note the variable z in this rule, which refers to the father of y , is not present in the rule postconditions. Whenever such a variable occurs only in the preconditions, it is assumed to be existentially quantified; that is, the rule preconditions are satisfied as long as there exists at least one binding of the variable that satisfies the corresponding literal.

Before getting into the FOIL Algorithm, let us understand the meaning of first-order rules and the various terminologies involved in it.

FIRST-ORDER LOGIC:

All expressions in first-order logic are composed of the following attributes:

constants — e.g. tyler, 23, a

variables — e.g. A, B, C

predicate symbols — e.g. male, father (True or False values only)

function symbols — e.g. age (can take on any constant as a value)

connectives — e.g. \wedge , \vee , \neg , \rightarrow , \leftarrow

quantifiers — e.g. \forall , \exists

Term: It can be defined as any constant, variable or function applied to any term.
e.g. age(bob)

Terminology:

Before moving on to algorithms for learning sets of Horn clauses, let us learn some basic terminology from formal logic. All expressions are composed of constants (e.g., Bob, Louise), variables (e.g., x, y), predicate symbols (e.g., Married, Greater-Than), and function symbols (e.g., age).

The difference between predicates and functions is that predicates take on values of True or False, whereas functions may take on any constant as their value.

We will use lowercase symbols for variables and capitalized symbols for constants. Also, we will use lowercase for functions and capitalized symbols for predicates.

From these symbols, we build up expressions as follows:

A term is any constant, any variable, or any function applied to any term (e.g., Bob, x, age(Bob)). A literal is any predicate or its negation applied to any term (e.g., Married (Bob, Louise), -Greater-Than(age(Sue), 20)). If a literal contains a negation (\neg) symbol, we call it a negative literal, otherwise a positive literal.

A clause is any disjunction of literals, where all variables are assumed to be universally quantified.

A Horn clause is a clause containing at most one positive literal, such as

$$H \vee \neg L_1 \vee \dots \vee \neg L_n$$

where H is the positive literal, and $\neg L_1 \dots \neg L_n$ are negative literals. Because of the equalities $(B \vee \neg A) = (B \leftarrow A)$ and $\neg(A \wedge B) = (\neg A \vee \neg B)$, the above Horn clause can alternatively be written in the form

$$H \leftarrow (L_1 \wedge \dots \wedge L_n)$$

which is equivalent to the following, using our earlier rule notation

$$\text{IF } L_1 \wedge \dots \wedge L_n, \text{ THEN } H$$

Whatever the notation, the Horn clause preconditions $L_1 \wedge \dots \wedge L_n$ are called the clause body or, alternatively, the clause antecedents. The literal H that forms the postcondition is called the clause head or, alternatively, the clause consequent.

LEARNING SETS OF FIRST-ORDER RULES: FOIL

- A variety of algorithms has been proposed for learning first-order rules, or Horn clauses. Now Let us consider a program called FOIL (Quinlan 1990) that employs an approach very similar to the SEQUENTIAL-COVERING and LEARN-ONE

RULE algorithms.

- In fact, the FOIL program is the natural extension of these earlier algorithms to first-order representations. Formally, the hypotheses learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause with two exceptions.
- First, the rules learned by FOIL are more restricted than general Horn clauses, because the literals are not permitted to contain function symbols (this reduces the complexity of the hypothesis space search).
- Second, FOIL rules are more expressive than Horn clauses, because the literals appearing in the body of the rule may be negated. FOIL has been applied to a variety of problem domains. For example, it has been demonstrated to learn a recursive definition of the QUICKSORT algorithm and to learn to discriminate legal from illegal chess positions.

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* \leftarrow those *Examples* for which the *Target_predicate* is *True*
 - *Neg* \leftarrow those *Examples* for which the *Target_predicate* is *False*
 - *Learned_rules* $\leftarrow \{\}$
 - while *Pos*, do
 - Learn a NewRule*
 - *NewRule* \leftarrow the rule that predicts *Target_predicate* with no preconditions
 - *NewRuleNeg* \leftarrow *Neg*
 - while *NewRuleNeg*, do
 - Add a new literal to specialize NewRule*
 - *Candidate_literals* \leftarrow generate candidate new literals for *NewRule*, based on *Predicates*
 - *Best_literal* $\leftarrow \underset{L \in \text{Candidate_literals}}{\operatorname{argmax}} \operatorname{Foil_Gain}(L, \text{NewRule})$
 - add *Best_literal* to preconditions of *NewRule*
 - *NewRuleNeg* \leftarrow subset of *NewRuleNeg* that satisfies *NewRule* preconditions
 - *Learned_rules* \leftarrow *Learned_rules* + *NewRule*
 - *Pos* \leftarrow *Pos* – {members of *Pos* covered by *NewRule*}
 - Return *Learned_rules*
-

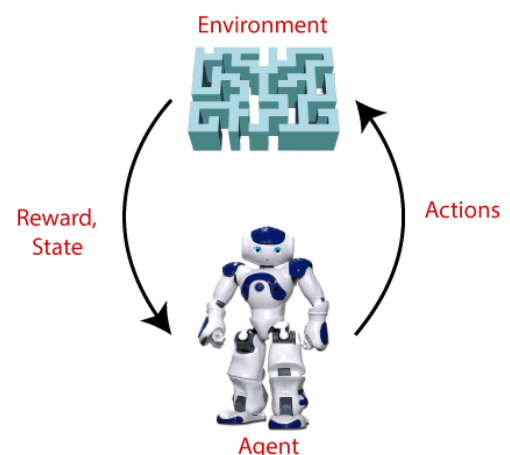
- The outer loop corresponds to a variant of the SEQUENTIAL-COVERING algorithm discussed earlier; that is, it learns new rules one at a time, removing the positive examples covered by the latest rule before attempting to learn the next rule.
- The inner loop corresponds to a variant of our earlier LEARN-ONE-RULE algorithm, extended to accommodate first-order rules.
- Note also there are a few minor differences between FOIL and these earlier algorithms. In particular, FOIL seeks only rules that predict when the target literal is True, whereas our earlier algorithm would seek both rules that predict when it is True and rules that predict when it is False.

- Also, FOIL performs a simple hill climbing search rather than a beam search (equivalently, it uses a beam of width one). The hypothesis space search performed by FOIL is best understood by viewing it hierarchically. Each iteration through FOIL'S outer loop adds a new rule to its disjunctive hypothesis, Learned rules.

REINFORCEMENT LEARNING:

What is Reinforcement Learning?

- Reinforcement Learning is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions. For each good action, the agent gets positive feedback, and for each bad action, the agent gets negative feedback or penalty.
- In Reinforcement Learning, the agent learns automatically using feedbacks without any labeled data, unlike supervised learning.
- Since there is no labeled data, so the agent is bound to learn by its experience only.
- RL solves a specific type of problem where decision making is sequential, and the goal is long-term, such as **game-playing, robotics**, etc.
- The agent interacts with the environment and explores it by itself. The primary goal of an agent in reinforcement learning is to improve the performance by getting the maximum positive rewards.
- The agent learns with the process of hit and trial, and based on the experience, it learns to perform the task in a better way. Hence, we can say that **"Reinforcement learning is a type of machine learning method where an intelligent agent (computer program) interacts with the environment and learns to act within that."** How a Robotic dog learns the movement of his arms is an example of Reinforcement learning.
- It is a core part of Artificial intelligence, and all AI agent works on the concept of reinforcement learning. Here we do not need to pre-program the agent, as it learns from its own experience without any human intervention.
- **Example:** Suppose there is an AI agent present within a maze environment, and his goal is to find the diamond. The agent interacts with the environment by performing some actions, and based on those actions, the state of the agent gets changed, and it also receives a reward or penalty as feedback.
- The agent continues doing these three things (**take action, change state/remain in the same state, and get feedback**), and by doing these actions, he learns and explores the environment.
- The agent learns that what actions lead to positive feedback or rewards and what actions lead to negative feedback penalty. As a positive reward, the agent gets a positive point, and as a penalty, it gets a negative point.



Terms used in Reinforcement Learning

- **Agent():** An entity that can perceive/explore the environment and act upon it.
- **Environment():** A situation in which an agent is present or surrounded by. In RL, we assume the stochastic environment, which means it is random in nature.
- **Action():** Actions are the moves taken by an agent within the environment.
- **State():** State is a situation returned by the environment after each action taken by the agent.
- **Reward():** A feedback returned to the agent from the environment to evaluate the action of the agent.
- **Policy():** Policy is a strategy applied by the agent for the next action based on the current state.
- **Value():** It is expected long-term return with the discount factor and opposite to the short-term reward.
- **Q-value():** It is mostly similar to the value, but it takes one additional parameter as a current action (a).

Key Features of Reinforcement Learning

- In RL, the agent is not instructed about the environment and what actions need to be taken.
- It is based on the hit and trial process.
- The agent takes the next action and changes states according to the feedback of the previous action.
- The agent may get a delayed reward.
- The environment is stochastic, and the agent needs to explore it to reach to get the maximum positive rewards.

Approaches to implement Reinforcement Learning

There are mainly three ways to implement reinforcement-learning in ML, which are:

1. Value-based:

The value-based approach is about to find the optimal value function, which is the maximum value at a state under any policy. Therefore, the agent expects the long-term return at any state(s) under policy π .

2. Policy-based:

Policy-based approach is to find the optimal policy for the maximum future rewards without using the value function. In this approach, the agent tries to apply such a policy

that the action performed in each step helps to maximize the future reward. The policy-based approach has mainly two types of policy:

- **Deterministic:** The same action is produced by the policy (π) at any state.
 - **Stochastic:** In this policy, probability determines the produced action.
3. **Model-based:** In the model-based approach, a virtual model is created for the environment, and the agent explores that environment to learn it. There is no particular solution or algorithm for this approach because the model representation is different for each environment.

Elements of Reinforcement Learning

There are four main elements of Reinforcement Learning, which are given below:

1. Policy
2. Reward Signal
3. Value Function
4. Model of the environment
5. **2) Reward Signal:** The goal of reinforcement learning is defined by the reward signal. At each state, the environment sends an immediate signal to the learning agent, and this signal is known as a **reward signal**. These rewards are given according to the good and bad actions taken by the agent. The agent's main objective is to maximize the total number of rewards for good actions. The reward signal can change the policy, such as if an action selected by the agent leads to low reward, then the policy may change to select other actions in the future.
6. **3) Value Function:** The value function gives information about how good the situation and action are and how much reward an agent can expect. A reward indicates the **immediate signal for each good and bad action**, whereas a value function specifies **the good state and action for the future**. The value function depends on the reward, as, without reward, there could be no value. The goal of estimating values is to achieve more rewards.
7. **4) Model:** The last element of reinforcement learning is the model, which mimics the behaviour of the environment. With the help of the model, one can make inferences about how the environment will behave. Such as, if a state and an action are given, then a model can predict the next state and reward.
8. The model is used for planning, which means it provides a way to take a course of action by considering all future situations before actually experiencing those situations. The approaches for solving the RL problems **with the help of the**

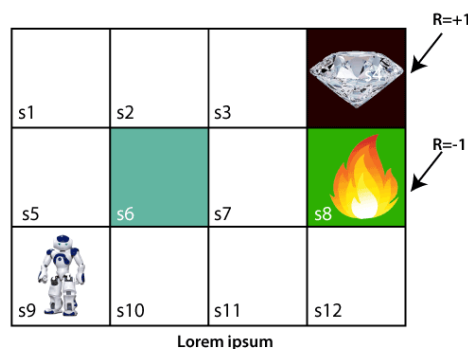
model are termed as the **model-based approach**. Comparatively, an approach **without using a model** is called a **model-free approach**.

How does Reinforcement Learning Work?

To understand the working process of the RL, we need to consider two main things:

- **Environment:** It can be anything such as a room, maze, football ground, etc.
- **Agent:** An intelligent agent such as AI robot.

Let's take an example of a maze environment that the agent needs to explore. Consider the below image:

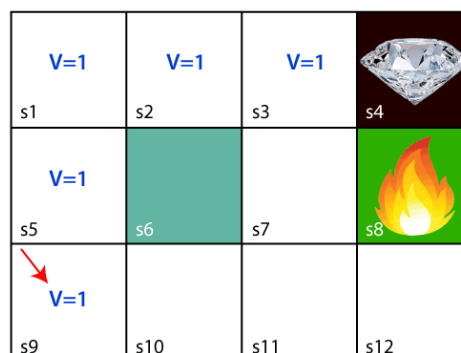


In the above image, the agent is at the very first block of the maze. The maze is consisting of an S_6 block, which is a **wall**, S_8 a **fire pit**, and S_4 a **diamond block**.

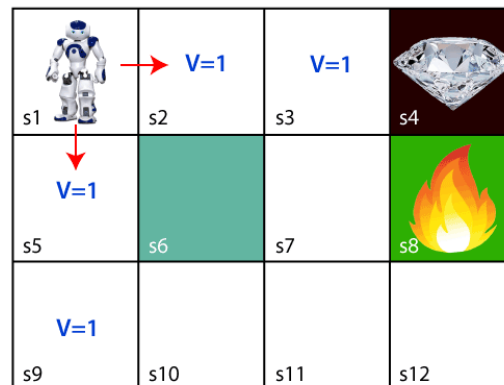
The agent cannot cross the S_6 block, as it is a solid wall. If the agent reaches the S_4 block, then get the **+1 reward**; if it reaches the fire pit, then gets **-1 reward point**. It can take four actions: **move up, move down, move left, and move right**.

The agent can take any path to reach to the final point, but he needs to make it in possible fewer steps. Suppose the agent considers the path **S9-S5-S1-S2-S3**, so he will get the +1-reward point.

The agent will try to remember the preceding steps that it has taken to reach the final step. To memorize the steps, it assigns 1 value to each previous step. Consider the below step:



Now, the agent has successfully stored the previous steps assigning the 1 value to each previous block. But what will the agent do if he starts moving from the block, which has 1 value block on both sides? Consider the below diagram:



It will be a difficult condition for the agent whether he should go up or down as each block has the same value. So, the above approach is not suitable for the agent to reach the destination. Hence to solve the problem, we will use the **Bellman equation**, which is the main concept behind reinforcement learning.

The Bellman Equation

The Bellman equation was introduced by the Mathematician **Richard Ernest Bellman** in the year **1953**, and hence it is called as a Bellman equation. It is associated with dynamic programming and used to calculate the values of a decision problem at a certain point by including the values of previous states.

It is a way of calculating the value functions in dynamic programming or environment that leads to modern reinforcement learning.

The key-elements used in Bellman equations are:

- Action performed by the agent is referred to as "a"
- State occurred by performing the action is "s."
- The reward/feedback obtained for each good and bad action is "R."
- A discount factor is Gamma " γ ."

The Bellman equation can be written as:

$$V(s) = \max [R(s,a) + \gamma V(s')]$$

Where,

$V(s)$ = value calculated at a particular point.

$R(s,a)$ = Reward at a particular state s by performing an action.

γ = Discount factor

$V(s')$ = The value at the previous state.

In the above equation, we are taking the max of the complete values because the agent tries to find the optimal solution always.

So now, using the Bellman equation, we will find value at each state of the given environment. We will start from the block, which is next to the target block.

For 1st block:

$V(s_3) = \max [R(s,a) + \gamma V(s')]$, here $V(s') = 0$ because there is no further state to move.

$$V(s_3) = \max[R(s,a)] \Rightarrow V(s_3) = \max[1] \Rightarrow \mathbf{V(s_3) = 1.}$$

For 2nd block:

$V(s_2) = \max [R(s,a) + \gamma V(s')]$, here $\gamma = 0.9$ (lets), $V(s') = 1$, and $R(s, a) = 0$, because there is no reward at this state.

$$V(s_2) = \max[0.9(1)] \Rightarrow V(s_2) = \max[0.9] \Rightarrow \mathbf{V(s_2) = 0.9}$$

For 3rd block:

$V(s_1) = \max [R(s,a) + \gamma V(s')]$, here $\gamma = 0.9$ (lets), $V(s') = 0.9$, and $R(s, a) = 0$, because there is no reward at this state also.

$$V(s_1) = \max[0.9(0.9)] \Rightarrow V(s_1) = \max[0.81] \Rightarrow \mathbf{V(s_1) = 0.81}$$

For 4th block:

$V(s_5) = \max [R(s,a) + \gamma V(s')]$, here $\gamma = 0.9$ (lets), $V(s') = 0.81$, and $R(s, a) = 0$, because there is no reward at this state also.





$$V(s_5) = \max[0.9(0.81)] \Rightarrow V(s_5) = \max[0.73] \Rightarrow \mathbf{V(s_5) = 0.73}$$

For 5th block:





$V(s_9) = \max [R(s,a) + \gamma V(s')]$, here $\gamma = 0.9$ (lets), $V(s') = 0.73$, and $R(s, a) = 0$, because there is no reward at this state also.

$$V(s_9) = \max[0.9(0.73)] \Rightarrow V(s_9) = \max[0.66] \Rightarrow \mathbf{V(s_9) = 0.66}$$


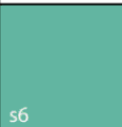

Consider the below image:

V=0.81 s1	V=0.9 s2	V=1 s3	 s4
V=0.73 s5	 s6	s7	 s8
 V=0.66 s9	s10	s11	s12

Now, we will move further to the 6th block, and here agent may change the route because it always tries to find the optimal path. So now, let's consider from the block next to the fire pit.

V=0.81 s1	V=0.9 s2	V=1 s3	 s4
V=0.73 s5		 s7	 s8
V=0.66 s9	s10	s11	s12

Now, the agent has three options to move; if he moves to the blue box, then he will feel a bump if he moves to the fire pit, then he will get the -1 reward. But here we are taking only positive rewards, so for this, he will move to upwards only. The complete block values will be calculated using this formula. Consider the below image:

V=0.81 s1	V=0.9 s2	V=1 s3	 s4
V=0.73 s5		V=0.9 s7	 s8
V=0.66 s9	V=0.73 s10	V=0.81 s11	V=0.73 s12

Types of Reinforcement learning

There are mainly two types of reinforcement learning, which are:

- **Positive Reinforcement**
- **Negative Reinforcement**

Positive Reinforcement:

The positive reinforcement learning means adding something to increase the tendency that expected behavior would occur again. It impacts positively on the behavior of the agent and increases the strength of the behavior.

This type of reinforcement can sustain the changes for a long time, but too much positive reinforcement may lead to an overload of states that can reduce the consequences.

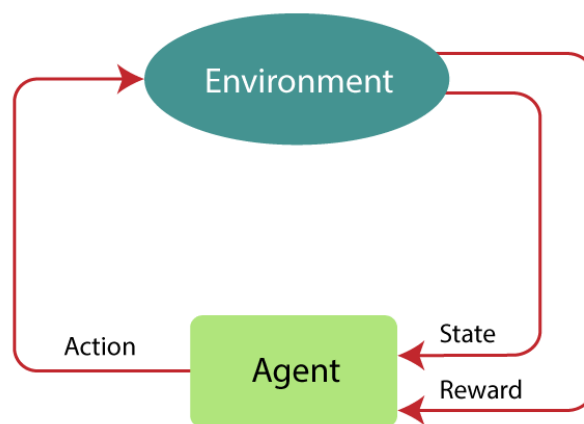
Negative Reinforcement:

The negative reinforcement learning is opposite to the positive reinforcement as it increases the tendency that the specific behavior will occur again by avoiding the negative condition.

It can be more effective than the positive reinforcement depending on situation and behavior, but it provides reinforcement only to meet minimum behavior.

Markov Decision Process

Markov Decision Process or MDP, is used to **formalize the reinforcement learning problems**. If the environment is completely observable, then its dynamic can be modeled as a **Markov Process**. In MDP, the agent constantly interacts with the environment and performs actions; at each action, the environment responds and generates a new state.



MDP is used to describe the environment for the RL, and almost all the RL problem can be formalized using MDP.

MDP contains a tuple of four elements (S, A, P_a , R_a):

- A set of finite States S
- A set of finite Actions A
- Rewards received after transitioning from state S to state S', due to action a.
- Probability P_a .

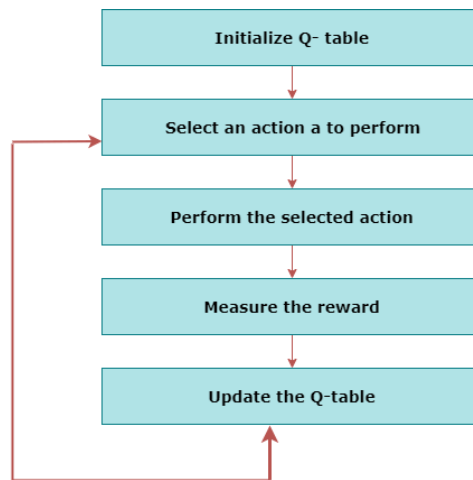
MDP uses **Markov property**, and to better understand the MDP, we need to learn about it.

Reinforcement Learning Algorithms – Q-Learning:

Reinforcement learning algorithms are mainly used in AI applications and gaming applications. The main used algorithms are:

- **Q-Learning:**

- Q-learning is an **Off policy RL algorithm**, which is used for the temporal difference Learning. The temporal difference learning methods are the way of comparing temporally successive predictions.
- It learns the value function $Q(S, a)$, which means how good to take action "**a**" at a particular state "**s**."
- The below flowchart explains the working of Q- learning:



- **State Action Reward State action (SARSA):**

- SARSA stands for **State Action Reward State action**, which is an **on-policy** temporal difference learning method. The on-policy control method selects the action for each state while learning using a specific policy.
- The goal of SARSA is to calculate the **$Q \pi(s, a)$ for the selected current policy π and all pairs of $(s-a)$** .
- The main difference between Q-learning and SARSA algorithms is that **unlike Q-learning, the maximum reward for the next state is not required for updating the Q-value in the table**.
- In SARSA, new action and reward are selected using the same policy, which has determined the original action.
- The SARSA is named because it uses the quintuple **$Q(s, a, r, s', a')$** . Where,
 - s: original state**
 - a: Original action**
 - r: reward observed while following the states**
 - s' and a': New state, action pair.**

- **Deep Q Neural Network (DQN):**

- As the name suggests, DQN is a **Q-learning using Neural networks**.
- For a big state space environment, it will be a challenging and complex task to define and update a Q-table.
- To solve such an issue, we can use a DQN algorithm. Where, instead of defining a Q-table, neural network approximates the Q-values for each action and state.

Temporal Difference (TD)

Temporal difference is an agent learning from an environment through episodes with no prior knowledge of the environment. This means temporal difference takes a model-free or unsupervised learning approach. You can consider it learning from trial and error.

we'll discuss 3 algorithms: TD(0), TD(1) and TD(λ).

1. **Gamma (γ):** the discount rate. A value between 0 and 1. The higher the value the less you are discounting.
2. **Lambda (λ):** the credit assignment variable. A value between 0 and 1. The higher the value the more credit you can assign to further back states and actions.
3. **Alpha (α):** the learning rate. How much of the error should we accept and therefore adjust our estimates towards. A value between 0 and 1. A higher value adjusts aggressively, accepting more of the error while a smaller one adjusts conservatively but may make more conservative moves towards the actual values.
4. **Delta (δ):** a change or difference in value.

Confidence Intervals:

One common way to describe the uncertainty associated with an estimate is to give an interval within which the true value is expected to fall, along with the probability with which it is expected to fall into this interval. Such estimates are called confidence interval estimates.

Definition: An N% confidence interval for some parameter p is an interval that is expected with probability N% to contain p.

Difference between Reinforcement Learning and Supervised Learning:

The Reinforcement Learning and Supervised Learning both are the part of machine learning, but both types of learnings are far opposite to each other. The RL agents interact with the environment, explore it, take action, and get rewarded. Whereas supervised learning algorithms learn from the labeled dataset and, on the basis of the training, predict the output.

The difference table between RL and Supervised learning is given below:

Reinforcement Learning	Supervised Learning
RL works by interacting with the environment.	Supervised learning works on the existing dataset.

The RL algorithm works like the human brain works when making some decisions.	Supervised Learning works as when a human learns things in the supervision of a guide.
There is no labeled dataset is present	The labeled dataset is present.
No previous training is provided to the learning agent.	Training is provided to the algorithm so that it can predict the output.
RL helps to take decisions sequentially.	In Supervised learning, decisions are made when input is given.

Reinforcement Learning Applications

1. Robotics:

1. RL is used in **Robot navigation, Robo-soccer, walking, juggling**, etc.

2. Control:

1. RL can be used for **adaptive control** such as Factory processes, admission control in telecommunication, and Helicopter pilot is an example of reinforcement learning.

3. Game Playing:

1. RL can be used in **Game playing** such as tic-tac-toe, chess, etc.

4. Chemistry:

1. RL can be used for optimizing the chemical reactions.

5. Business:

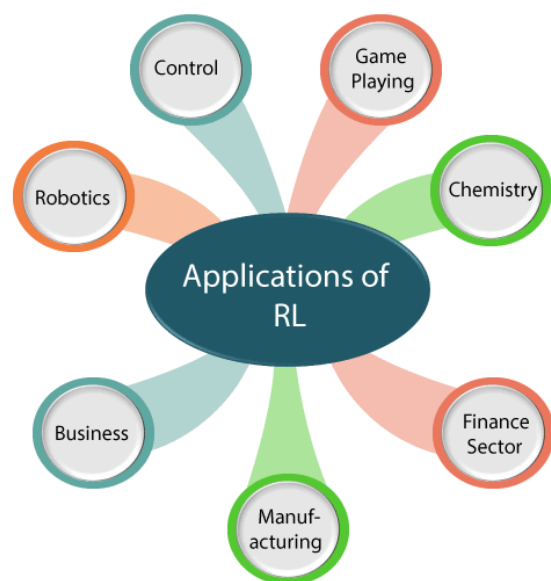
1. RL is now used for business strategy planning.

6. Manufacturing:

1. In various automobile manufacturing companies, the robots use deep reinforcement learning to pick goods and put them in some containers.

7. Finance Sector:

1. The RL is currently used in the finance sector for evaluating trading strategies.



Source:

<https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56>

<https://www.geeksforgeeks.org/what-is-reinforcement-learning/>

<https://www.javatpoint.com/reinforcement-learning#Q-Learning>