MODULE 4

BAYESIAN LEARNING

Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data

INTRODUCTION

Bayesian learning methods are relevant to study of machine learning for two different reasons.

1. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems
2. The second reason is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Features of Bayesian Learning Methods

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example
- Prior knowledge can be combined with observed data to determine the final probability of a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions
- New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.
- Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

Practical difficulty in applying Bayesian methods

1. One practical difficulty in applying Bayesian methods is that they typically require initial knowledge of many probabilities. When these probabilities are not known in advance they are often estimated based on background knowledge, previously available data, and assumptions about the form of the underlying distributions.
2. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case. In certain specialized situations, this computational cost can be significantly reduced.

BAYES THEOREM

Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

Notations

- P(h) prior probability of h, reflects any background knowledge about the chance that h is correct
- P(D) prior probability of D, probability that D will be observed
- P(D|h) probability of observing D given a world in which h holds
- P(h|D) posterior probability of h, reflects confidence that h holds after D has been observed

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability P(h|D), from the prior probability P(h), together with P(D) and P(D|h).

Bayes Theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- P(h|D) increases with P(h) and with P(D|h) according to Bayes theorem.
- P(h|D) decreases as P(D) increases, because the more probable it is that D will be observed independent of h, the less evidence D provides in support of h.

Maximum a Posteriori (MAP) Hypothesis

- In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis h ∈ H given the observed data D. Any such maximally probable hypothesis is called a maximum a posteriori (MAP) hypothesis.
- Bayes theorem to calculate the posterior probability of each candidate hypothesis is h_{MAP} is a MAP hypothesis provided

$$\begin{aligned} h_{MAP} &= \underset{h \in H}{argmax}\ P(h|D) \\ &= \underset{h \in H}{argmax}\ \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{argmax}\ P(D|h)P(h) \end{aligned}$$

- P(D) can be dropped, because it is a constant independent of h

Maximum Likelihood (ML) Hypothesis

- In some cases, it is assumed that every hypothesis in H is equally probable a priori ($P(h_i) = P(h_j)$ for all h_i and h_j in H).
- In this case the below equation can be simplified and need only consider the term P(D|h) to find the most probable hypothesis.

$$h_{MAP} = \underset{h \in H}{argmax}\ P(D|h)P(h)$$

the equation can be simplified

$$h_{ML} = \underset{h \in H}{argmax}\ P(D|h)$$

P(D|h) is often called the likelihood of the data D given h, and any hypothesis that maximizes P(D|h) is called a maximum likelihood (ML) hypothesis

Example

- Consider a medical diagnosis problem in which there are two alternative hypotheses: (1) that the patient has particular form of cancer, and (2) that the patient does not. The available data is from a particular laboratory test with two possible outcomes: + (positive) and - (negative).

- We have prior knowledge that over the entire population of people only .008 have this disease. Furthermore, the lab test is only an imperfect indicator of the disease.
- The test returns a correct positive result in only 98% of the cases in which the disease is actually present and a correct negative result in only 97% of the cases in which the disease is not present. In other cases, the test returns the opposite result.
- The above situation can be summarized by the following probabilities:

$$P(cancer) = .008 \qquad P(\neg cancer) = 0.992$$
$$P(\oplus|cancer) = .98 \qquad P(\ominus|cancer) = .02$$
$$P(\oplus|\neg cancer) = .03 \qquad P(\ominus|\neg cancer) = .97$$

Suppose a new patient is observed for whom the lab test returns a positive (+) result. Should we diagnose the patient as having cancer or not?

$$P(\oplus|cancer)P(cancer) = (.98).008 = .0078$$
$$P(\oplus|\neg cancer)P(\neg cancer) = (.03).992 = .0298$$
$$\Rightarrow h_{MAP} = \neg cancer$$

The exact posterior probabilities can also be determined by normalizing the above quantities so that they sum to 1

$$P(cancer|\oplus) = \frac{0.0078}{0.0078 + 0.0298} = 0.21$$

$$P(\neg cancer|\oplus) = \frac{0.0298}{0.0078 + 0.0298} = 0.79$$

Basic formulas for calculating probabilities are summarized in Table

---

- ***Product rule***: probability $P(A \wedge B)$ of a conjunction of two events A and B

$$P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$$

- ***Sum rule***: probability of a disjunction of two events A and B

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

- ***Bayes theorem***: the posterior probability $P(h|D)$ of h given D

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- ***Theorem of total probability***: if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$, then

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

BAYES THEOREM AND CONCEPT LEARNING

What is the relationship between Bayes theorem and the problem of concept learning?

Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, and can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable.

Brute-Force Bayes Concept Learning

Consider the concept learning problem

- Assume the learner considers some finite hypothesis space H defined over the instance space X, in which the task is to learn some target concept $c : X \rightarrow \{0,1\}$.
- Learner is given some sequence of training examples $((x_1, d_1) \ldots (x_m, d_m))$ where x_i is some instance from X and where d_i is the target value of x_i (i.e., $d_i = c(x_i)$).
- The sequence of target values are written as $D = (d_1 \ldots d_m)$.

We can design a straightforward concept learning algorithm to output the maximum a posteriori hypothesis, based on Bayes theorem, as follows:

BRUTE-FORCE MAP LEARNING algorithm:

1. For each hypothesis h in H, calculate the posterior probability

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

2. Output the hypothesis h_{MAP} with the highest posterior probability

$$h_{MAP} = \underset{h \in H}{argmax}\ P(h|D)$$

In order specify a learning problem for the BRUTE-FORCE MAP LEARNING algorithm we must specify what values are to be used for P(h) and for P(D|h) ?

Let's choose P(h) and for P(D|h) to be consistent with the following assumptions:

- The training data D is noise free (i.e., $d_i = c(x_i)$)
- The target concept c is contained in the hypothesis space H
- Do not have a priori reason to believe that any hypothesis is more probable than any other.

What values should we specify for P(h)?

- Given no prior knowledge that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis h in H.
- Assume the target concept is contained in H and require that these prior probabilities sum to 1.

$$P(h) = \frac{1}{|H|} \text{ for all } h \in H$$

What choice shall we make for P(D|h)?

- P(D|h) is the probability of observing the target values D = (d_1 . . .d_m) for the fixed set of instances (x_1 . . . x_m), given a world in which hypothesis h holds
- Since we assume noise-free training data, the probability of observing classification d_i given h is just 1 if $d_i = h(x_i)$ and 0 if $d_i \neq h(x_i)$. Therefore,

$$P(D|h) = \begin{cases} 1 & \text{if } d_i = h(x_i) \text{ for all } d_i \in D \\ 0 & \text{otherwise} \end{cases}$$

Given these choices for P(h) and for P(D|h) we now have a fully-defined problem for the above BRUTE-FORCE MAP LEARNING algorithm.

Recalling Bayes theorem, we have

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Consider the case where h is inconsistent with the training data D

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0$$

The posterior probability of a hypothesis inconsistent with D is zero

Consider the case where h is consistent with D

$$P(h|D) = \frac{1 \cdot \frac{1}{|H|}}{P(D)} = \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} = \frac{1}{|VS_{H,D}|}$$

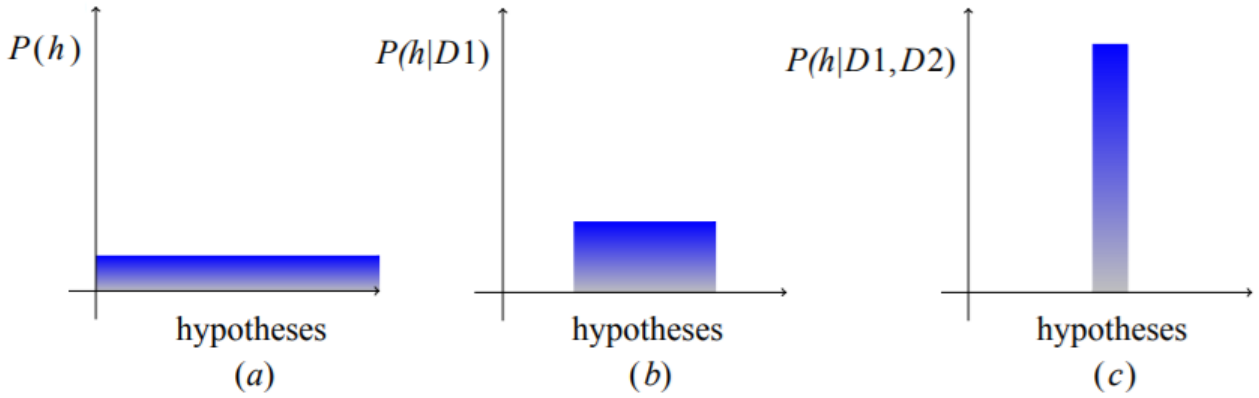
Where, $VS_{H,D}$ is the subset of hypotheses from H that are consistent with D

To summarize, Bayes theorem implies that the posterior probability P(h|D) under our assumed P(h) and P(D|h) is

$$P(D|h) = \begin{cases} \frac{1}{|V S_{H,D}|} & \text{if h is consistent with D} \\ 0 & \text{otherwise} \end{cases}$$

The Evolution of Probabilities Associated with Hypotheses

- Figure (a) all hypotheses have the same probability.
- Figures (b) and (c), As training data accumulates, the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to 1 is shared equally among the remaining consistent hypotheses.



MAP Hypotheses and Consistent Learners

- A learning algorithm is a consistent learner if it outputs a hypothesis that commits zero errors over the training examples.
- Every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability distribution over H ($P(h_i) = P(h_j)$ for all i, j), and deterministic, noise free training data (P(D|h) =1 if D and h are consistent, and 0 otherwise).

Example:

- FIND-S outputs a consistent hypothesis, it will output a MAP hypothesis under the probability distributions P(h) and P(D|h) defined above.
- Are there other probability distributions for P(h) and P(D|h) under which FIND-S outputs MAP hypotheses? Yes.
- Because FIND-S outputs a maximally specific hypothesis from the version space, its output hypothesis will be a MAP hypothesis relative to any prior probability distribution that favours more specific hypotheses.

Note

- Bayesian framework is a way to characterize the behaviour of learning algorithms
- By identifying probability distributions P(h) and P(D|h) under which the output is a optimal hypothesis, implicit assumptions of the algorithm can be characterized (Inductive Bias)
- Inductive inference is modelled by an equivalent probabilistic reasoning system based on Bayes theorem

MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

Consider the problem of learning a *continuous-valued target function* such as neural network learning, linear regression, and polynomial curve fitting

A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a *maximum likelihood (ML) hypothesis*

- Learner L considers an instance space X and a hypothesis space H consisting of some class of real-valued functions defined over X, i.e., ($\forall$ h $\in$ H)[h : X $\rightarrow$ R] and training examples of the form $<x_i,d_i>$
- The problem faced by L is to learn an unknown target function f : X $\rightarrow$ R
- A set of m training examples is provided, where the target value of each example is corrupted by random noise drawn according to a Normal probability distribution with zero mean ($d_i = f(x_i) + e_i$)
- Each training example is a pair of the form (x_i ,d_i) where $d_i = f(x_i) + e_i$.
  - Here $f(x_i)$ is the noise-free value of the target function and e_i is a random variable representing the noise.
  - It is assumed that the values of the e_i are drawn independently and that they are distributed according to a Normal distribution with zero mean.
- The task of the learner is to *output a maximum likelihood hypothesis* or a *MAP hypothesis assuming all hypotheses are equally probable a priori*.

Using the definition of h_{ML} we have

$$h_{ML} = \underset{h \in H}{argmax}\ p(D|h)$$

Assuming training examples are mutually independent given h, we can write P(D|h) as the product of the various (d_i|h)

$$h_{ML} = \underset{h \in H}{argmax} \prod_{i=1}^{m} p(d_i|h)$$

Given the noise e_i obeys a Normal distribution with zero mean and unknown variance σ^2 , each d_i must also obey a Normal distribution around the true targetvalue $f(x_i)$. Because we are writing the expression for P(D|h), we assume h is the correct description of f.
Hence, $\mu = f(x_i) = h(x_i)$

$$h_{ML} = \underset{h \in H}{\text{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i-\mu)^2}$$

$$h_{ML} = \underset{h \in H}{argmax} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - h(x_i))^2}$$

Maximize the less complicated logarithm, which is justified because of the monotonicity of function p

$$h_{ML} = \underset{h \in H}{argmax} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

The first term in this expression is a constant independent of h, and can therefore be discarded, yielding

$$h_{ML} = \underset{h \in H}{argmax} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Maximizing this negative quantity is equivalent to minimizing the corresponding positive quantity

$$h_{ML} = \underset{h \in H}{argmin} \sum_{i=1}^{m} \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Finally, discard constants that are independent of h.

$$h_{ML} = \underset{h \in H}{argmin} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

Thus, above equation shows that the maximum likelihood hypothesis h_{ML} is the one that minimizes the sum of the squared errors between the observed training values d_i and the hypothesis predictions $h(x_i)$

Note:

Why is it reasonable to choose the Normal distribution to characterize noise?

- Good approximation of many types of noise in physical systems
- Central Limit Theorem shows that the sum of a sufficiently large number of independent, identically distributed random variables itself obeys a Normal distribution

Only noise in the target value is considered, not in the attributes describing the instances themselves

MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

- Consider the setting in which we wish to learn a nondeterministic (probabilistic) function f : X → {0, 1}, which has two discrete output values.
- We want a function approximator whose output is the probability that f(x) = 1. In other words, learn the target function f ` : X → [0, 1] such that f ` (x) = P(f(x) = 1)

How can we learn f ` using a neural network?

- Use of brute force way would be to first collect the observed frequencies of 1's and 0's for each possible value of x and to then train the neural network to output the target frequency for each x.

What criterion should we optimize in order to find a maximum likelihood hypothesis for f' in this setting?

- First obtain an expression for P(D|h)
- Assume the training data D is of the form D = {(x_1, d_1) . . . (x_m, d_m)}, where d_i is the observed 0 or 1 value for f (x_i).
- Both x_i and d_i as random variables, and assuming that each training example is drawn independently, we can write P(D|h) as

$$P(D \mid h) = \prod_{i=1}^{m} P(x_i, d_i \mid h) \qquad \text{equ (1)}$$

Applying the product rule

$$P(D \mid h) = \prod_{i=1}^{m} P(d_i \mid h, x_i) P(x_i) \qquad \text{equ (2)}$$

The probability $P(d_i|h, x_i)$

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases} \qquad \text{equ (3)}$$

Re-express it in a more mathematically manipulable form, as

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \qquad \text{equ (4)}$$

Equation (4) to substitute for P(di |h, xi) in Equation (5) to obtain

$$P(D|h) = \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i) \qquad \text{equ (5)}$$

We write an expression for the maximum likelihood hypothesis

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i)$$

The last term is a constant independent of h, so it can be dropped

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} \qquad \text{equ (6)}$$

It easier to work with the log of the likelihood, yielding

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)) \qquad \text{equ (7)}$$

Equation (7) describes the quantity that must be maximized in order to obtain the maximum likelihood hypothesis in our current problem setting

Gradient Search to Maximize Likelihood in a Neural Net

- Derive a weight-training rule for neural network learning that seeks to maximize G(h,D) using gradient ascent
- The gradient of G(h,D) is given by the vector of partial derivatives of G(h,D) with respect to the various network weights that define the hypothesis h represented by the learned network
- In this case, the partial derivative of G(h, D) with respect to weight w_{jk} from input k to unit j is

$$\begin{aligned} \frac{\partial G(h,D)}{\partial w_{jk}} &= \sum_{i=1}^{m} \frac{\partial G(h, D)}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^{m} \frac{\partial (d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)))}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^{m} \frac{d_i - h(x_i)}{h(x_i)(1 - h(x_i))} \frac{\partial h(x_i)}{\partial w_{jk}} \qquad \text{equ (1)} \end{aligned}$$

- Suppose our neural network is constructed from a single layer of sigmoid units. Then,

$$\frac{\partial h(x_i)}{\partial w_{jk}} = \sigma'(x_i) x_{ijk} = h(x_i)(1 - h(x_i)) x_{ijk}$$

where x_{ijk} is the k^{th} input to unit j for the i^{th} training example, and d(x) is the derivative of the sigmoid squashing function.

- Finally, substituting this expression into Equation (1), we obtain a simple expression for the derivatives that constitute the gradient

$$\frac{\partial G(h, D)}{\partial w_{jk}} = \sum_{i=1}^{m} (d_i - h(x_i))\ x_{ijk}$$

Because we seek to maximize rather than minimize P(D|h), we perform gradient ascent rather than gradient descent search. On each iteration of the search the weight vector is adjusted in the direction of the gradient, using the weight update rule

$$w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$$

Where,

$$\Delta w_{jk} = \eta \sum_{i=1}^{m} (d_i - h(x_i))\ x_{ijk} \qquad \text{equ (2)}$$

Where, η is a small positive constant that determines the step size of the i gradient ascent search

MINIMUM DESCRIPTION LENGTH PRINCIPLE

- A Bayesian perspective on Occam's razor
- Motivated by interpreting the definition of h_{MAP} in the light of basic concepts from information theory.

$$h_{MAP} = \underset{h \in H}{argmax}\ P(D|h)P(h)$$

which can be equivalently expressed in terms of maximizing the $\log_2$

$$h_{MAP} = \underset{h \in H}{argmax}\ \log_2 P(D|h) + \log_2 P(h)$$

or alternatively, minimizing the negative of this quantity

$$h_{MAP} = \underset{h \in H}{argmin}\ -\log_2 P(D|h) - \log_2 P(h) \qquad \text{equ (1)}$$

This equation (1) can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data

- $-\log_2 P(h)$: the description length of h under the optimal encoding for the hypothesis space H, $L_{C_H}(h) = -\log_2 P(h)$, where C_H is the optimal code for hypothesis space H.
- $-\log_2 P(D \mid h)$: the description length of the training data D given hypothesis h, under the optimal encoding from the hypothesis space H: $L_{C_H}(D|h) = -\log_2 P(D|\ h)$, where $C_{D|h}$ is the optimal code for describing data D assuming that both the sender and receiver know the hypothesis h.
- Rewrite Equation (1) to show that h_{MAP} is the hypothesis h that minimizes the sum given by the description length of the hypothesis plus the description length of the data given the hypothesis.

$$h_{MAP} = \underset{h \in H}{argmin}\ L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

Where, C_H and $C_{D|h}$ are the optimal encodings for H and for D given h

The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths of equ.

$$h_{MAP} = \underset{h \in H}{argmin}\ L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

Minimum Description Length principle:

$$h_{MDL} = \underset{h \in H}{\text{argmin}}\ L_{C_1}(h) + L_{C_2}(D \mid h)$$

Where, codes C_1 and C_2 to represent the hypothesis and the data given the hypothesis

The above analysis shows that if we choose C_1 to be the optimal encoding of hypotheses C_H, and if we choose C_2 to be the optimal encoding $C_{D|h}$, then $h_{MDL} = h_{MAP}$

Application to Decision Tree Learning

Apply the MDL principle to the problem of learning decision trees from some training data. *What should we choose for the representations C1 and C2 of hypotheses and data?*

- For C_1: C_1 might be some obvious encoding, in which the description length grows with the number of nodes and with the number of edges
- For C_2: Suppose that the sequence of instances (x_1 . . .x_m) is already known to both the transmitter and receiver, so that we need only transmit the classifications (f (x_1) . . . f (x_m)).
- Now if the training classifications (f (x_1) . . .f(x_m)) are identical to the predictions of the hypothesis, then there is no need to transmit any information about these examples. The description length of the classifications given the hypothesis ZERO
- If examples are misclassified by h, then for each misclassification we need to transmit a message that identifies which example is misclassified as well as its correct classification
- The hypothesis h_{MDL} under the encoding C_1 and C_2 is just the one that minimizes the sum of these description lengths.

NAIVE BAYES CLASSIFIER

- The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function f (x) can take on any value from some finite set V.
- A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values (a_1, a_2.. .a_m).
- The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value, V_{MAP}, given the attribute values (a_1, a_2.. .a_m) that describe the instance

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

Use Bayes theorem to rewrite this expression as

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j) \qquad \textbf{equ (1)}$$

- The naive Bayes classifier is based on the assumption that the attribute values are conditionally independent given the target value. Means, the assumption is that given the target value of the instance, the probability of observing the conjunction (a_1, a_2.. .a_m), is just the product of the probabilities for the individual attributes:

$$P(a_1, a_2 \ldots a_n | v_j) = \prod_i P(a_i | v_j)$$

Substituting this into Equation (1),

Naive Bayes classifier:

$$\mathbf{V_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j)} \qquad \text{equ (2)}$$

Where, V_{NB} denotes the target value output by the naive Bayes classifier

An Illustrative Example

- Let us apply the naive Bayes classifier to a concept learning problem i.e., classifying days according to whether someone will play tennis.
- The below table provides a set of 14 training examples of the target concept ***PlayTennis***, where each day is described by the attributes Outlook, Temperature, Humidity, and Wind

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

- Use the naive Bayes classifier and the training data from this table to classify the following novel instance:

  < Outlook = sunny, Temperature = cool, Humidity = high, Wind = strong >

- Our task is to predict the target value ***(yes or no)*** of the target concept ***PlayTennis*** for this new instance

$$V_{NB} = \underset{v_j \in \{yes, no\}}{\text{argmax}}\ P(v_j) \prod_i P(a_i | v_j)$$

$$V_{NB} = \underset{v_j \in \{yes, no\}}{\text{argmax}}\ P(v_j)\ P(\text{Outlook=sunny}|v_j)\ P(\text{Temperature=cool}|v_j)\ P(\text{Humidity=high}|v_j)\ P(\text{Wind=strong}|v_j)$$

The probabilities of the different target values can easily be estimated based on their frequencies over the 14 training examples

- P(P1ayTennis = yes) = 9/14 = 0.64
- P(P1ayTennis = no) = 5/14 = 0.36

Similarly, estimate the conditional probabilities. For example, those for Wind = strong

- P(Wind = strong | PlayTennis = yes) = 3/9 = 0.33
- P(Wind = strong | PlayTennis = no) = 3/5 = 0.60

Calculate V_{NB} according to Equation (1)

$$P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$$
$$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$$

Thus, the naive Bayes classifier assigns the target value ***PlayTennis = no*** to this new instance, based on the probability estimates learned from the training data.

By normalizing the above quantities to sum to one, calculate the conditional probability that the target value is ***no***, given the observed attribute values

$$\frac{.0206}{(.0206 + .0053)} = .795$$

Estimating Probabilities

- We have estimated probabilities by the fraction of times the event is observed to occur over the total number of opportunities.
- For example, in the above case we estimated P(Wind = strong | Play Tennis = no) by the fraction n_c /n where, n = 5 is the total number of training examples for which PlayTennis = no, and n_c = 3 is the number of these for which Wind = strong.
- When n_c = 0, then n_c /n will be zero and this probability term will dominate the quantity calculated in Equation (2) requires multiplying all the other probability terms by this zero value
- To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the ***m-estimate*** defined as follows

m **-estimate of probability:**

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

- p is our prior estimate of the probability we wish to determine, and m is a constant called the equivalent sample size, which determines how heavily to weight p relative to the observed data
- Method for choosing p in the absence of other information is to assume uniform priors; that is, if an attribute has k possible values we set p = 1 /k.

BAYESIAN BELIEF NETWORKS

- The naive Bayes classifier makes significant use of the assumption that the values of the attributes $a_1 \ldots a_n$ are conditionally independent given the target value v.
- This assumption dramatically reduces the complexity of learning the target function

A Bayesian belief network describes the probability distribution governing a set of variables by specifying a set of conditional independence assumptions along with a set of conditional probabilities
Bayesian belief networks allow stating conditional independence assumptions that apply to subsets of the variables

Notation

- Consider an arbitrary set of random variables $Y_1 \ldots Y_n$, where each variable Yi can take on the set of possible values $V(Y_i)$.
- The joint space of the set of variables Y to be the cross product $V(Y_1)$ x $V(Y_2)$ x. . . $V(Y_n)$.
- In other words, each item in the joint space corresponds to one of the possible assignments of values to the tuple of variables $(Y_1 \ldots Y_n)$. The probability distribution over this joint' space is called the joint probability distribution.
- The joint probability distribution specifies the probability for each of the possible variable bindings for the tuple $(Y_1 \ldots Y_n)$.
- A Bayesian belief network describes the joint probability distribution for a set of variables.

Conditional Independence

Let X, Y, and Z be three discrete-valued random variables. X is conditionally independent of Y given Z if the probability distribution governing X is independent of the value of Y given a value for Z, that is, if

$$(\forall x_i, y_j, z_k)\ P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

Where,

$$x_i \in V(X),\ y_j \in V(Y),\ \text{and}\ z_k \in V(Z).$$

The above expression is written in abbreviated form as

$$P(X \mid Y, Z) = P(X \mid Z)$$

Conditional independence can be extended to sets of variables. The set of variables $X_1 \ldots X_l$ is conditionally independent of the set of variables $Y_1 \ldots Y_m$ given the set of variables $Z_1 \ldots Z_n$ if

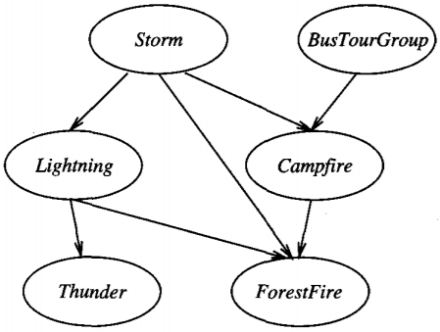
$$P(X_1 \ldots X_l | Y_1 \ldots Y_m, Z_1 \ldots Z_n) = P(X_1 \ldots X_l | Z_1 \ldots Z_n)$$

The naive Bayes classifier assumes that the instance attribute A_1 is conditionally independent of instance attribute A_2 given the target value V. This allows the naive Bayes classifier to calculate $P(A_l, A_2 \mid V)$ as follows,

$$\begin{aligned} P(A_1, A_2|V) &= P(A_1|A_2, V)P(A_2|V) \\ &= P(A_1|V)P(A_2|V) \end{aligned}$$

Representation

A Bayesian belief network represents the joint probability distribution for a set of variables. Bayesian networks (BN) are represented by directed acyclic graphs.



	S,B	S,¬B	¬S,B	¬S,¬B
C	0.4	0.1	0.8	0.2
¬C	0.6	0.9	0.2	0.8

Campfire

The Bayesian network in above figure represents the joint probability distribution over the boolean variables *Storm, Lightning, Thunder, ForestFire, Campfire,* and *BusTourGroup*

A Bayesian network (BN) represents the joint probability distribution by specifying a set of *conditional independence assumptions*

- BN represented by a directed acyclic graph, together with sets of local conditional probabilities
- Each variable in the joint space is represented by a node in the Bayesian network
- The network arcs represent the assertion that the variable is conditionally independent of its non-descendants in the network given its immediate predecessors in the network.
- A ***conditional probability table*** **(CPT)** is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors

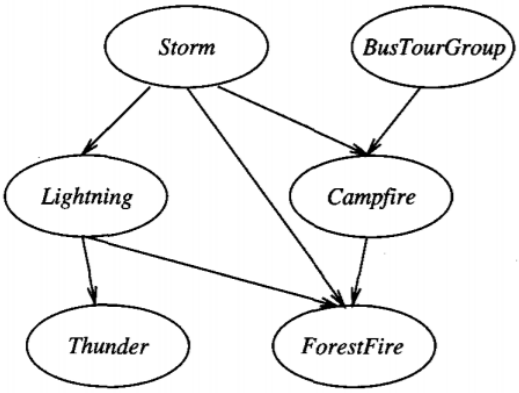
The joint probability for any desired assignment of values ($y_1, \ldots, y_n$) to the tuple of network variables ($Y_1 \ldots Y_m$) can be computed by the formula

$$P(y_1, \ldots, y_n) = \prod_{i=1}^{n} P(y_i | Parents(Y_i))$$

Where, Parents(Y_i) denotes the set of immediate predecessors of Y_i in the network.

Example:

Consider the node ***Campfire***. The network nodes and arcs represent the assertion that ***Campfire*** is conditionally independent of its non-descendants ***Lightning*** and ***Thunder***, given its immediate parents Storm and ***BusTourGroup***.



	S,B	S,¬B	¬S,B	¬S,¬B
C	0.4	0.1	0.8	0.2
¬C	0.6	0.9	0.2	0.8

Campfire

This means that once we know the value of the variables ***Storm*** and ***BusTourGroup***, the variables ***Lightning*** and ***Thunder*** provide no additional information about ***Campfire***

The conditional probability table associated with the variable ***Campfire.*** The assertion is

P(Campfire = True | Storm = True, BusTourGroup = True) = 0.4

Inference

- Use a Bayesian network to infer the value of some target variable (e.g., ForestFire) given the observed values of the other variables.
- Inference can be straightforward if values for all of the other variables in the network are known exactly.
- A Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables.
- An arbitrary Bayesian network is known to be NP-hard

Learning Bayesian Belief Networks

Affective algorithms can be considered for learning Bayesian belief networks from training data by considering several different settings for learning problem

- First, the network structure might be given in advance, or it might have to be inferred from the training data.
- Second, all the network variables might be directly observable in each training example, or some might be unobservable.
  - In the case where the network structure is given in advance and the variables are fully observable in the training examples, learning the conditional probability tables is straightforward and estimate the conditional probability table entries
  - In the case where the network structure is given but only some of the variable values are observable in the training data, the learning problem is more difficult. The learning problem can be compared to learning weights for an ANN.

Gradient Ascent Training of Bayesian Network

The gradient ascent rule which maximizes P(D|h) by following the gradient of ***ln P(D|h)*** with respect to the parameters that define the conditional probability tables of the Bayesian network.

Let w_{ijk} denote a single entry in one of the conditional probability tables. In particular w_{ijk} denote the conditional probability that the network variable Y_i will take on the value y_i, given that its immediate parents U_i take on the values given by u_{ik}.

The gradient of ***ln P(D|h)*** is given by the derivatives $\frac{\partial \ln P(D|h)}{\partial w_{ijk}}$ for each of the w_{ijk}.
As shown below, each of these derivatives can be calculated as

$$\frac{\partial \ln P(D|h)}{\partial w_{ij}} = \sum_{d \in D} \frac{P(Y_i = y_{ij}, U_i = u_{ik}|d)}{w_{ijk}} \qquad \textbf{equ(1)}$$

Derive the gradient defined by the set of derivatives $\frac{\partial P_h(D)}{\partial w_{ijk}}$ for all ***i, j,*** and ***k***. Assuming the training examples ***d*** in the data set D are drawn independently, we write this derivative as

$$\begin{aligned}\frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \frac{\partial}{\partial w_{ijk}} \ln \prod_{d \in D} P_h(d) \\ &= \sum_{d \in D} \frac{\partial \ln P_h(d)}{\partial w_{ijk}} \\ &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial P_h(d)}{\partial w_{ijk}}\end{aligned}$$

We write the abbreviation $P_h(D)$ to represent $P(D|h)$.

This last step makes use of the general equality $\frac{\partial \ln f(x)}{\partial x} = \frac{1}{f(x)} \frac{\partial f(x)}{\partial x}$. We can now introduce the values of the variables Y_i and $U_i = Parents(Y_i)$, by summing over their possible values $y_{ij'}$ and $u_{ik'}$.

$$\frac{\partial \ln P_h(D)}{\partial w_{ijk}} = \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} \sum_{j',k'} P_h(d|y_{ij'}, u_{ik'}) P_h(y_{ij'}, u_{ik'})$$

$$= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} \sum_{j',k'} P_h(d|y_{ij'}, u_{ik'}) P_h(y_{ij'}|u_{ik'}) P_h(u_{ik'})$$

This last step follows from the product rule of probability . Now consider the rightmost sum in the final expression above. Given that $w_{ijk} \equiv P_h(y_{ij}|u_{ik})$, the only term in this sum for which $\frac{\partial}{\partial w_{ijk}}$ is nonzero is the term for which $j' = j$ and $i' = i$. Therefore

$$\frac{\partial \ln P_h(D)}{\partial w_{ijk}} = \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} P_h(d|y_{ij}, u_{ik}) P_h(y_{ij}|u_{ik}) P_h(u_{ik})$$

$$= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} P_h(d|y_{ij}, u_{ik}) w_{ijk} P_h(u_{ik})$$

$$= \sum_{d \in D} \frac{1}{P_h(d)} P_h(d|y_{ij}, u_{ik}) P_h(u_{ik})$$

Applying Bayes theorem to rewrite $P_h(d|y_{ij}, u_{ik})$, we have

$$\frac{\partial \ln P_h(D)}{\partial w_{ijk}} = \sum_{d \in D} \frac{1}{P_h(d)} \frac{P_h(y_{ij}, u_{ik}|d) P_h(d) P_h(u_{ik})}{P_h(y_{ij}, u_{ik})}$$

$$= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d) P_h(u_{ik})}{P_h(y_{ij}, u_{ik})}$$

$$= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{P_h(y_{ij}|u_{ik})}$$

$$= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{w_{ijk}} \qquad \textbf{equ (2)}$$

Thus, we have derived the gradient given in Equation (1). There is one more item that must be considered before we can state the gradient ascent training procedure. In particular, we require that as the weights w_{ijk} are updated they must remain valid probabilities in the interval [0,1]. We also require that the sum $\sum_j w_{ijk}$ remains 1 for all i, k. These constraints can be satisfied by updating weights in a two-step process. First we update each w_{ijk} by gradient ascent

$$w_{ijk} \leftarrow w_{ijk} + \eta \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{w_{ijk}}$$

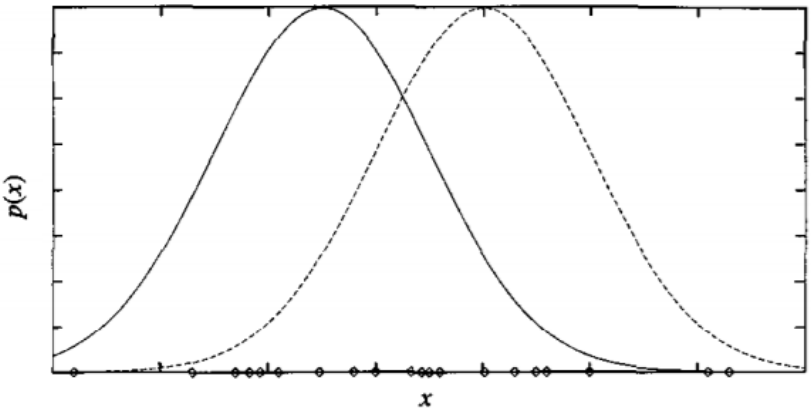
where η is a small constant called the learning rate. Second, we renormalize the weights w_{ijk} to assure that the above constraints are satisfied.
this process will converge to a locally maximum likelihood hypothesis for the conditional probabilities in the Bayesian network.

THE EM ALGORITHM

The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

Estimating Means of k Gaussians

- Consider a problem in which the data D is a set of instances generated by a probability distribution that is a mixture of k distinct Normal distributions.



- This problem setting is illustrated in Figure for the case where k = 2 and where the instances are the points shown along the x axis.
- Each instance is generated using a two-step process.
  - First, one of the k Normal distributions is selected at random.
  - Second, a single random instance x_i is generated according to this selected distribution.
- This process is repeated to generate a set of data points as shown in the figure.

- To simplify, consider the special case
  - The selection of the single Normal distribution at each step is based on choosing each with uniform probability
  - Each of the k Normal distributions has the same variance σ^2, known value.
- The learning task is to output a hypothesis h = (μ_1 , . . . ,μ_k) that describes the means of each of the k distributions.
- We would like to find a maximum likelihood hypothesis for these means; that is, a hypothesis ***h*** that maximizes p(D |h).

$$\mu_{ML} = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{m} (x_i - \mu)^2 \qquad (1)$$

In this case, the sum of squared errors is minimized by the sample mean

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i \qquad (2)$$

- Our problem here, however, involves a mixture of k different Normal distributions, and we cannot observe which instances were generated by which distribution.
- Consider full description of each instance as the triple (x_i, z_{i1}, z_{i2}),
  - where x_i is the observed value of the ith instance and
  - where z_{i1} and z_{i2} indicate which of the two Normal distributions was used to generate the value x_i
- In particular, z_{ij} has the value 1 if x_i was created by the j^{th} Normal distribution and 0 otherwise.
- Here x_i is the observed variable in the description of the instance, and z_{il} and z_{i2} are hidden variables.
- If the values of z_{il} and z_{i2} were observed, we could use following Equation to solve for the means p_1 and p_2
- Because they are not, we will instead use the EM algorithm

EM algorithm

Step 1: Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij}, assuming the current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds.

Step 2: Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, assuming the value taken on by each hidden variable z_{ij} is its expected value $E[z_{ij}]$ calculated in Step 1. Then replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by the new hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

Let us examine how both of these steps can be implemented in practice. Step 1 must calculate the expected value of each z_{ij}. This $E[z_{ij}]$ is just the probability that instance x_i was generated by the jth Normal distribution

$$E[z_{ij}] = \frac{p(x = x_i|\mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i|\mu = \mu_n)}$$

$$= \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i-\mu_n)^2}}$$

Thus the first step is implemented by substituting the current values $\langle \mu_1, \mu_2 \rangle$ and the observed x_i into the above expression.

In the second step we use the $E[z_{ij}]$ calculated during Step 1 to derive a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$.
maximum likelihood hypothesis in this case is given by

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

MODULE 3

ARTIFICIAL NEURAL NETWORKS

INTRODUCTION

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued target functions.

Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired by the observation that biological learning systems are built of very complex webs of interconnected ***Neurons***
- Human information processing system consists of brain ***neuron***: basic building block cell that communicates information to and from various parts of body

Facts of Human Neurobiology

- Number of neurons ~ 10^{11}
- Connection per neuron ~ 10^{4-5}
- Neuron switching time ~ 0.001 second or 10^{-3}
- Scene recognition time ~ 0.1 second
- 100 inference steps doesn't seem like enough
- Highly parallel computation based on distributed representation

Properties of Neural Networks

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)

NEURAL NETWORK REPRESENTATIONS

- A prototypical example of ANN learning is provided by Pomerleau's system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways
- The input to the neural network is a 30x32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network output is the direction in which the vehicle is steered

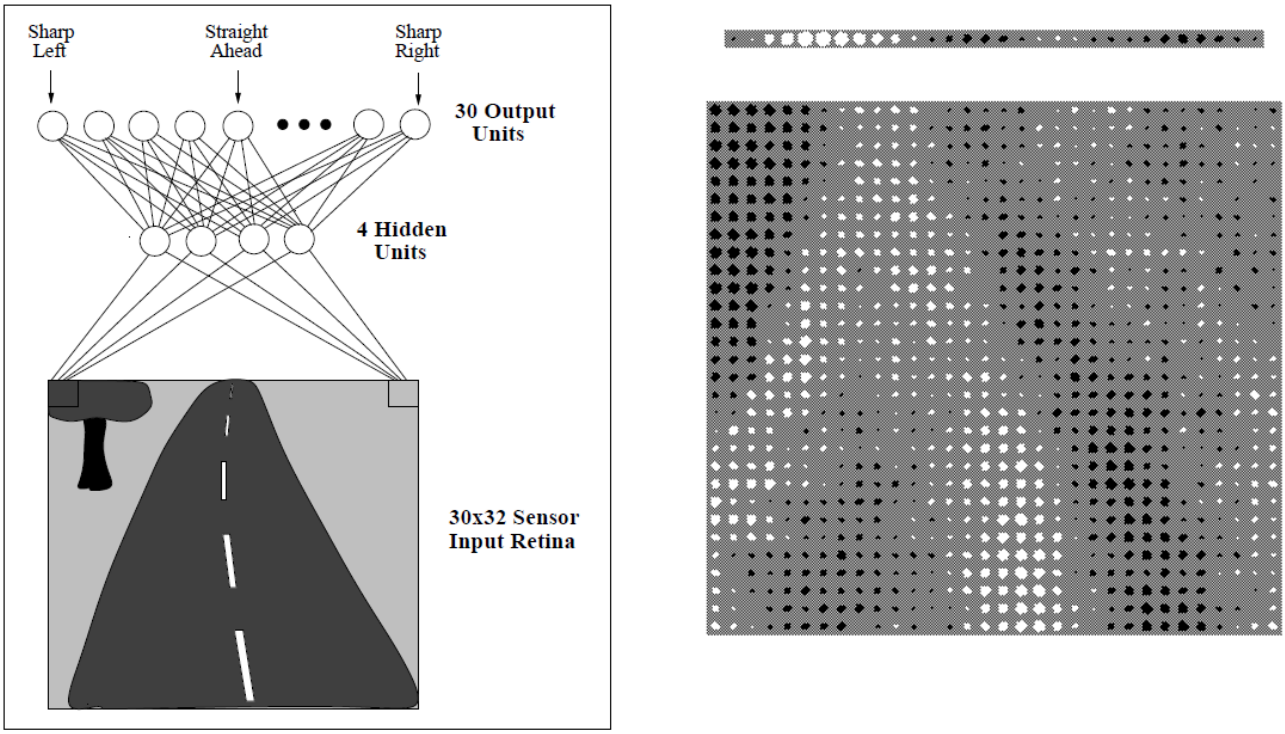


Figure: Neural network learning to steer an autonomous vehicle.

- Figure illustrates the neural network representation.
- The network is shown on the left side of the figure, with the input camera image depicted below it.
- Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs.
- There are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs
- These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
- Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.
- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.

ANN is appropriate for problems with the following characteristics:

1. Instances are represented by many attribute-value pairs.
2. The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
3. The training examples may contain errors.
4. Long training times are acceptable.
5. Fast evaluation of the learned target function may be required
6. The ability of humans to understand the learned target function is not important

PERCEPTRON

- One type of ANN system is based on a unit called a perceptron. Perceptron is a single layer neural network.

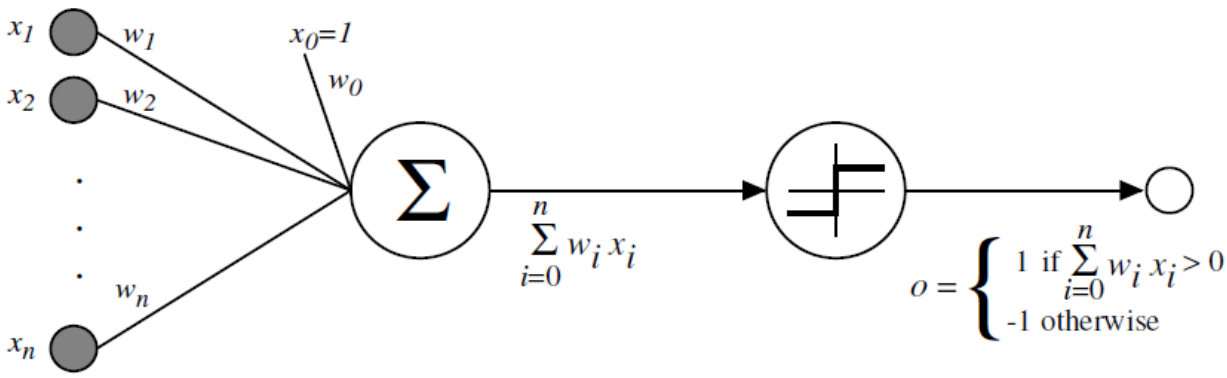


Figure: A perceptron

- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.
- Given inputs ***x*** through ***x,*** the output $O(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

- Where, each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- $-w_0$ is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

$$O(\vec{x}) = \text{sgn}\ (\vec{w} \cdot \vec{x})$$

Where,

$$\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

Representational Power of Perceptrons

- The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points)
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in below figure

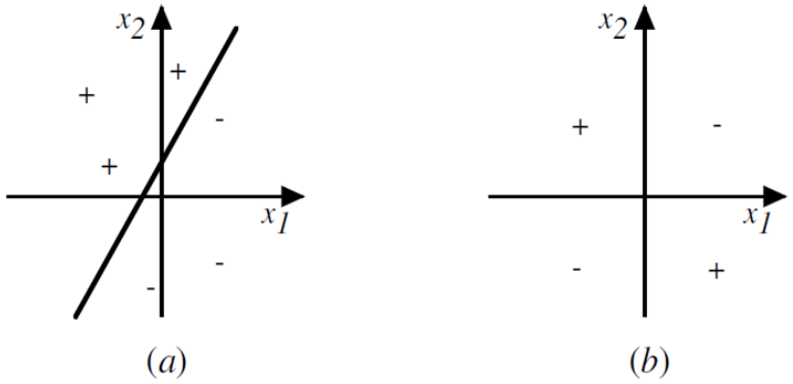
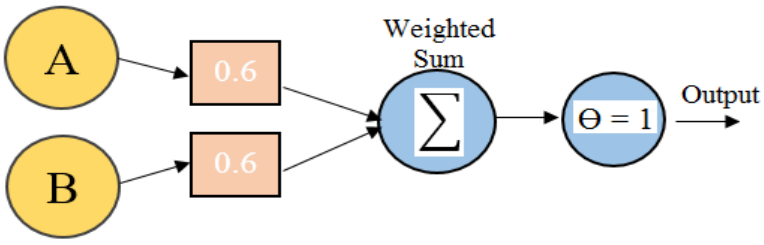


Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

Perceptrons can represent all of the primitive Boolean functions AND, OR, NAND (~ AND), and NOR (~OR)
Some Boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$

Example: Representation of AND functions

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1



If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.
This is not greater than the threshold of 1, so the output = 0.
If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.
This is not greater than the threshold, so the output = 0.
If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.
This is not greater than the threshold, so the output = 0.
If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.
This exceeds the threshold, so the output = 1.

Drawback of perceptron

- The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable

The Perceptron Training Rule

The learning problem is to determine a weight vector that causes the perceptron to produce the correct + 1 or - 1 output for each of the given training examples.

To learn an acceptable weight vector

- Begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input xi according to the rule.

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = \eta(t - o)x_i$$

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

- The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases

Drawback:
The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.

Gradient Descent and the Delta Rule

- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

To understand the delta training rule, consider the task of training an unthresholded perceptron. That is, a linear unit for which the output ***O*** is given by

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

$$O(\vec{x}) = (\vec{w} \cdot \vec{x}) \qquad \text{equ. (1)}$$

To derive a weight learning rule for linear units, specify a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.

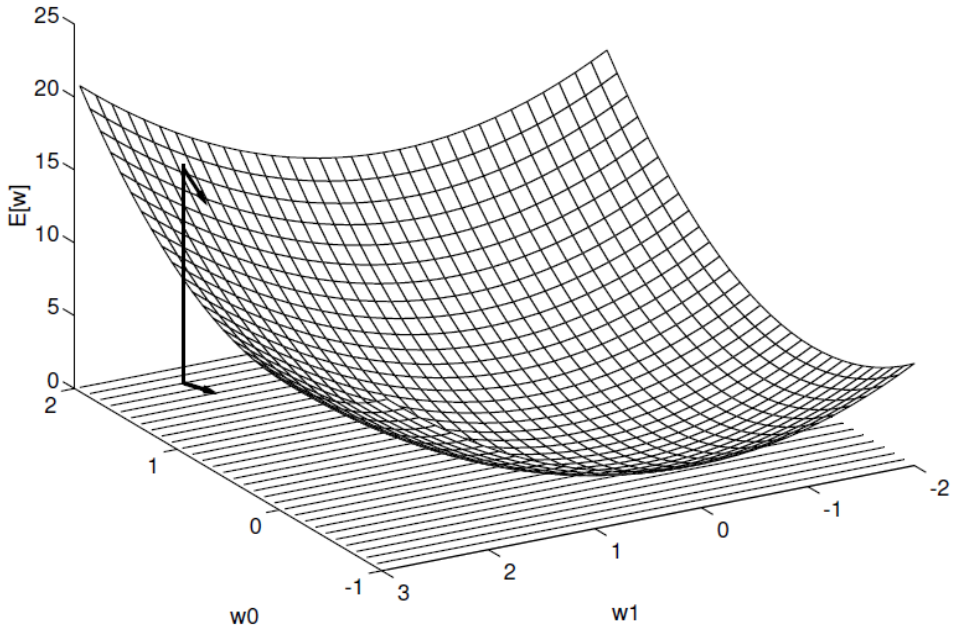
$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad \text{equ. (2)}$$

Where,

- D is the set of training examples,
- t_d is the target output for training example d,
- o_d is the output of the linear unit for training example d
- E ($\vec{w}$) is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

Visualizing the Hypothesis Space

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values as shown in below figure.
- Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.
- The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.
- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space



- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.
At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in above figure. This process continues until the global minimum error is reached.

Derivation of the Gradient Descent Rule

How to calculate the direction of steepest descent along the error surface?

The direction of steepest can be found by computing the derivative of E with respect to each component of the vector $\overrightarrow{w}$. This vector derivative is called the gradient of E with respect to $\overrightarrow{w}$, written as

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right] \qquad \text{equ. (3)}$$

The gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

Where,

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \qquad \text{equ. (4)}$$

- Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search.
- The negative sign is present because we want to move the weight vector in the direction that decreases E.

This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \qquad \text{equ. (5)}$$

Calculate the gradient at each step. The vector of $\frac{\partial E}{\partial wi}$ derivatives that form the gradient can be obtained by differentiating E from Equation (2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \qquad \text{equ. (6)}
\end{aligned}$$

Substituting Equation (6) into Equation (5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)\, x_{id} \qquad \text{equ. (7)}$$

GRADIENT DESCENT algorithm for training a linear unit

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$*, where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

To summarize, the gradient descent algorithm for training linear units is as follows:

- Pick an initial random weight vector.
- Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process

Issues in Gradient Descent Algorithm

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

1. The hypothesis space contains continuously parameterized hypotheses
2. The error can be differentiated with respect to these hypothesis parameters

The key practical difficulties in applying gradient descent are

1. Converging to a local minimum can sometimes be quite slow
2. If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

Stochastic Approximation to Gradient Descent

- The gradient descent training rule presented in Equation (7) computes weight updates after summing over all the training examples in D
- The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example

$$\Delta w_i = \eta\,(t - o)\,x_i$$

- where t, o, and x_i are the target value, unit output, and i^{th} input for the training example in question

---

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \eta(t - o)\,x_i \qquad (1)$$

---

stochastic approximation to gradient descent

One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

- Where, t_d and o_d are the target value and the unit output value for training example d.
- Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$
- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E_d(\vec{w})$
- By making the value of η sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely

The key differences between standard gradient descent and stochastic gradient descent are

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

Consider the example:

- Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d" (i.e., "hid," "had," "head," "hood," etc.).
- The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest.
- The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.

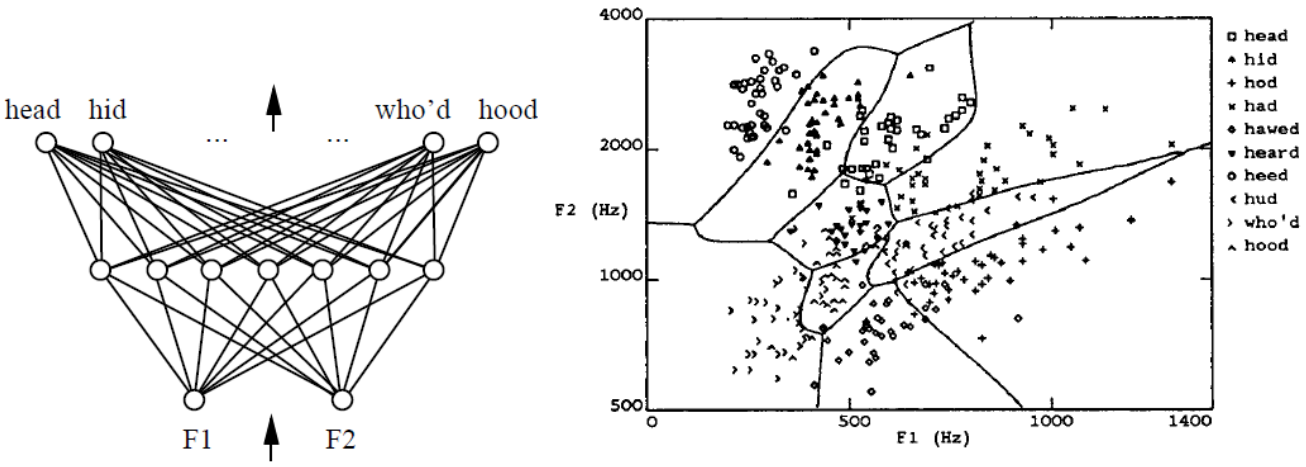


Figure: Decision regions of a multilayer feedforward network.

A Differentiable Threshold Unit (Sigmoid unit)

- Sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

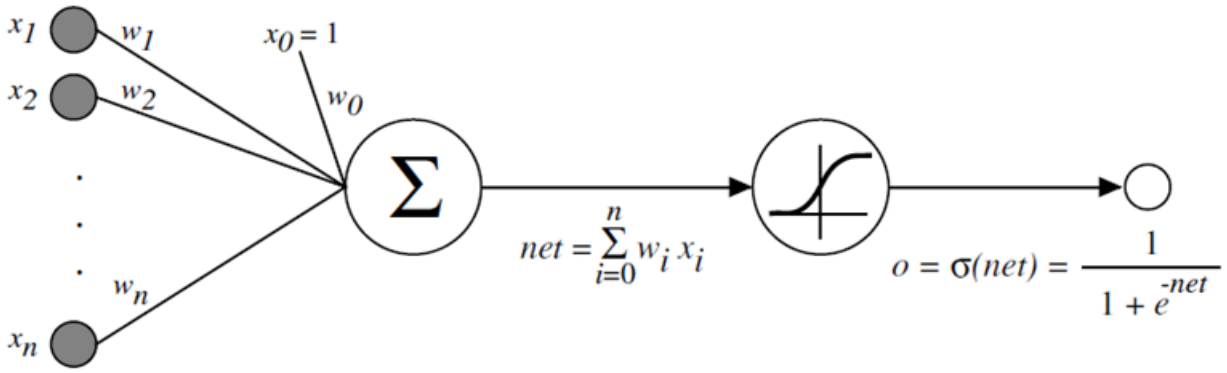


Figure: A Sigmoid Threshold Unit

- The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result and the threshold output is a continuous function of its input.
- More precisely, the sigmoid unit computes its output O as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

Where,

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

σ is the sigmoid function

The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- In BACKPROPAGATION algorithm, we consider networks with multiple output units rather than single units as before, so we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{.....equ. (1)}$$

where,

- ***outputs*** - is the set of output units in the network
- t_{kd} and O_{kd} - the target and output values associated with the k_{th} output unit
- ***d*** - training example

Algorithm:

BACKPROPAGATION (*training_example*, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form $(\vec{x}, \vec{t})$, *where* $(\vec{x})$ *is the vector of network input values,* $(\vec{t})$ *and is the vector of target network output values.*
η *is the learning rate (e.g., .05).* n_i, *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*
The input from unit i into unit j is denoted x_{ji}, *and the weight from unit i to unit j is denoted* w_{ji}

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each $(\vec{x}, \vec{t})$, in training examples, Do
    Propagate the input forward through the network:
  1. Input the instance $\vec{x}$, to the network and compute the output o_u of every unit u in the network.
     Propagate the errors backward through the network:
  2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  3. For each hidden unit *h*, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

  4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Where

$$\Delta w_{ji} = \eta \delta_j x_{i,j}$$

Adding Momentum

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. The most common is to alter the weight-update rule the equation below

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1)th iteration, as follows:

$$\Delta w_{ji}(n) = \eta \, \delta_j \, x_{ji} + \alpha \Delta w_{ji}(n-1)$$

Learning in arbitrary acyclic networks

- BACKPROPAGATION algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule is retained, and the only change is to the procedure for computing δ values.
- In general, the ***δ***, value for a unit ***r*** in layer ***m*** is computed from the ***δ*** values at the next deeper layer m + 1 according to

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\, m+1} w_{sr} \, \delta_s$$

- The rule for calculating ***δ*** for any internal unit

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr} \, \delta_s$$

Where, Downstream(r) is the set of units immediately downstream from unit ***r*** in the network: that is, all units whose inputs include the output of unit ***r***

Derivation of the BACKPROPAGATION Rule

- Deriving the stochastic gradient descent rule: Stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example
- For each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{.....equ. (1)}$$

where, E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in output} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit k for training example d, and o_k is the output of unit k given training example d.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables

- x_{ji} = the i^{th} input to unit j
- w_{ji} = the weight associated with the i^{th} input to unit j
- $net_j = \Sigma_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- outputs = the set of units in the final layer of the network
- Downstream(j) = the set of units whose immediate inputs include the output of unit j

derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule

seen in Equation $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

notice that weight w_{ji} can influence the rest of the network only through net_j.

Use chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \quad \text{.....equ(2)}$$

Derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

Consider two cases: The case where unit j is an output unit for the network, and the case where j is an internal unit (hidden unit).

Case 1: Training Rule for Output Unit Weights.

w_{ji} can influence the rest of the network only through net_j , net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \qquad \text{.....equ(3)}$$

To begin, consider just the first term in Equation (3)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \qquad \text{.....equ(4)}\end{aligned}$$

Next consider the second term in Equation (3). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j) \qquad \text{.....equ(5)}\end{aligned}$$

Substituting expressions (4) and (5) into (3), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j) \qquad \text{.....equ(6)}$$

and combining this with Equations (1) and (2), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\,(t_j - o_j)\, o_j(1 - o_j)x_{ji} \qquad \text{.....equ(7)}$$

Case 2: Training Rule for Hidden Unit Weights.

- In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d.
- For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network and denoted this set of units by Downstream(j).
- net_j can influence the network outputs only through the units in Downstream(j). Therefore, we can write

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j) \qquad \text{.....equ (8)}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

and

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

- The BACKPROPAGATION multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice.
- Local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.

Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima
2. Use stochastic gradient descent rather than true gradient descent
3. Train multiple networks using the same data, but initializing each network with different random weights

2. Representational Power of Feedforward Networks

What set of functions can be represented by feed-forward networks?
The answer depends on the width and depth of the networks. There are three quite general results are known about which function classes can be described by which types of Networks

1. Boolean functions – Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs

2. Continuous functions – Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units

3. Arbitrary functions – Any function can be approximated to arbitrary accuracy by a network with three layers of units.

3. Hypothesis Space Search and Inductive Bias

- Hypothesis space is the n-dimensional Euclidean space of the ***n*** network weights and hypothesis space is continuous.

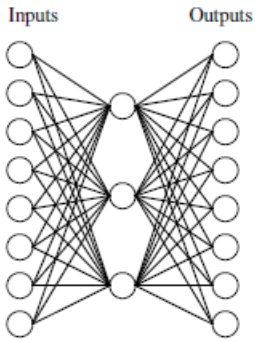
- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.
- It is difficult to characterize precisely the inductive bias of BACKPROPAGATION algorithm, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points.

4. Hidden Layer Representations

BACKPROPAGATION can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider example, the network shown in below Figure

A network:



Learned hidden layer representation:

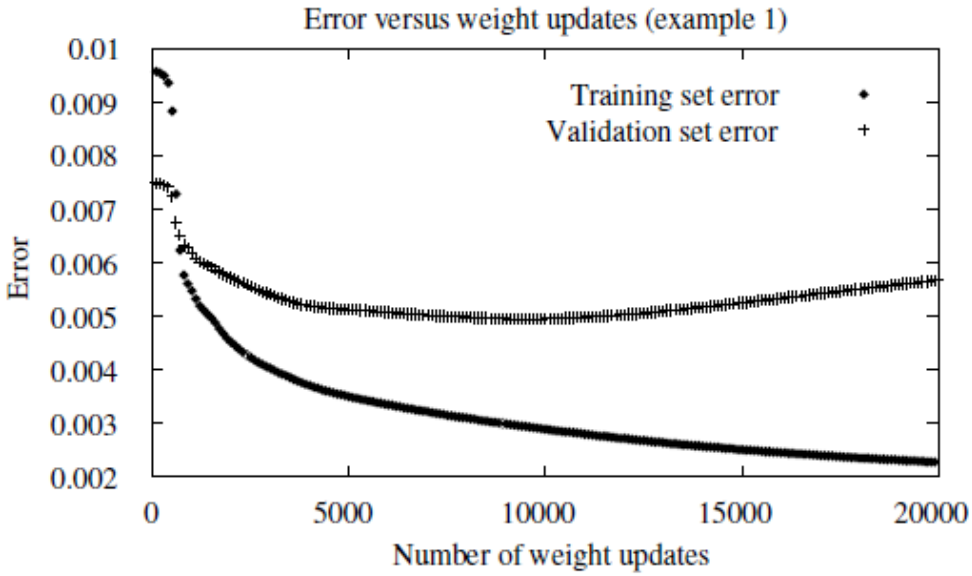
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

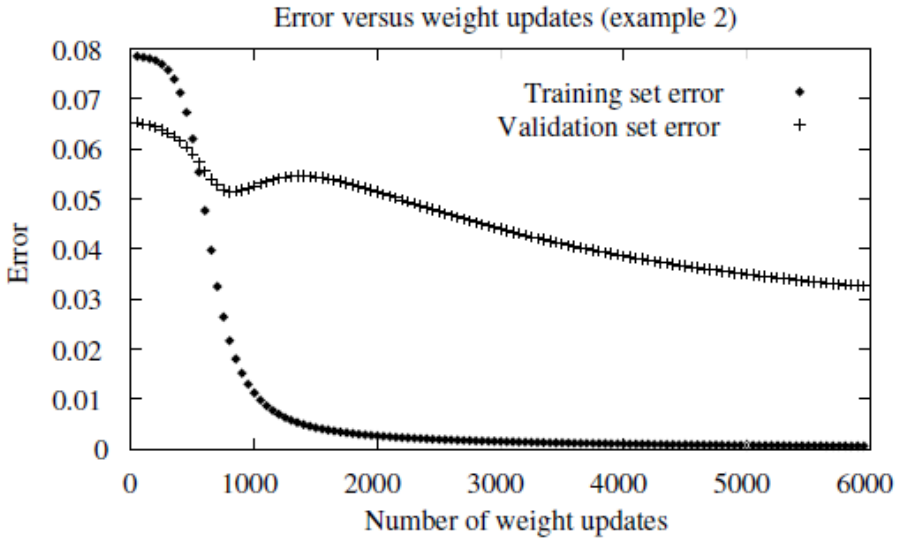
- Consider training the network shown in Figure to learn the simple target function f (x) = x, where x is a vector containing seven 0's and a single 1.
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.
- When BACKPROPAGATION applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., 000,001,010,. . . , 111). The exact values of the hidden units for one typical run of shown in Figure.
- This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning

5. Generalization, Overfitting, and Stopping Criterion

What is an appropriate condition for terminating the weight update loop? One choice is to continue training until the error E on the training examples falls below some predetermined threshold.

To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations





- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyond the training data.

- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies

- Why does overfitting tend to occur during later iterations, but not during earlier iterations?
  By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

Genetic Algorithm (GA)

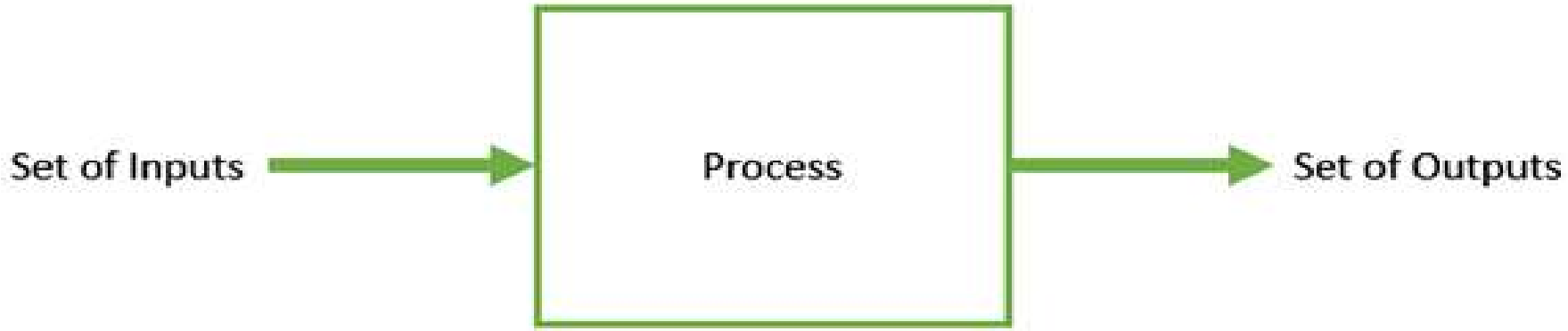
Genetic Algorithm (GA) is a search-based optimization technique based on the principles of **Genetics and Natural Selection**.

It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve.

It is frequently used to solve optimization problems, in research, and in machine learning.

Introduction to Optimization

Optimization is the process of **making something better**. In any process, we have a set of inputs and a set of outputs as shown in the following figure.



Optimization refers to finding the values of inputs in such a way that we get the “best” output values. The definition of “best” varies from problem to problem, but in mathematical terms, it refers to maximizing or minimizing one or more objective functions, by varying the input parameters.

The set of all possible solutions or values which the inputs can take make up the search space.

In this search space, lies a point or a set of points which gives the optimal solution.

The aim of optimization *is to find that point or set of points in the search space.*

Genetic Algorithm

- Optimization Algorithm
- Based on natural phenomenon
- Nature inspired approach based on Darwin's law of Survival of the fittest and bio-inspired operators such as Pairing Crossover and Mutation.

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

What are Genetic Algorithms?

Nature has always been a great source of inspiration to all mankind. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics.

GAs are a subset of a much larger branch of computation known as **Evolutionary Computation**.

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.

- The process of natural selection starts with the selection of fittest individuals from a population.
- They produce offspring which inherit the characteristics of the parents and will be added to the next generation.
- If parents have better fitness, their offspring will be better than parents and have a better chance at surviving.
- This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

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

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.

Limitations of GAs

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

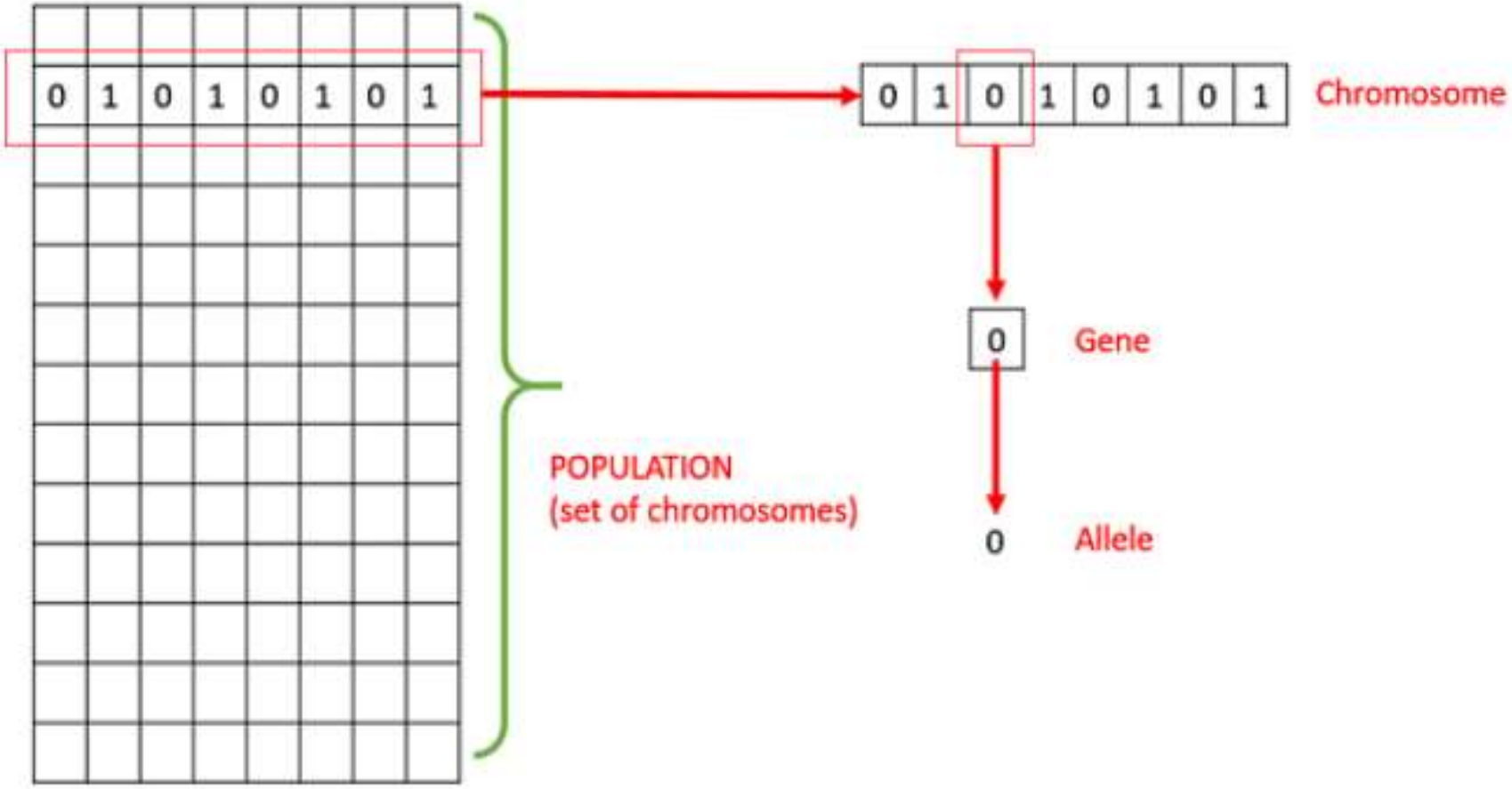
- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality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

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 chapter.

- **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.

- **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.

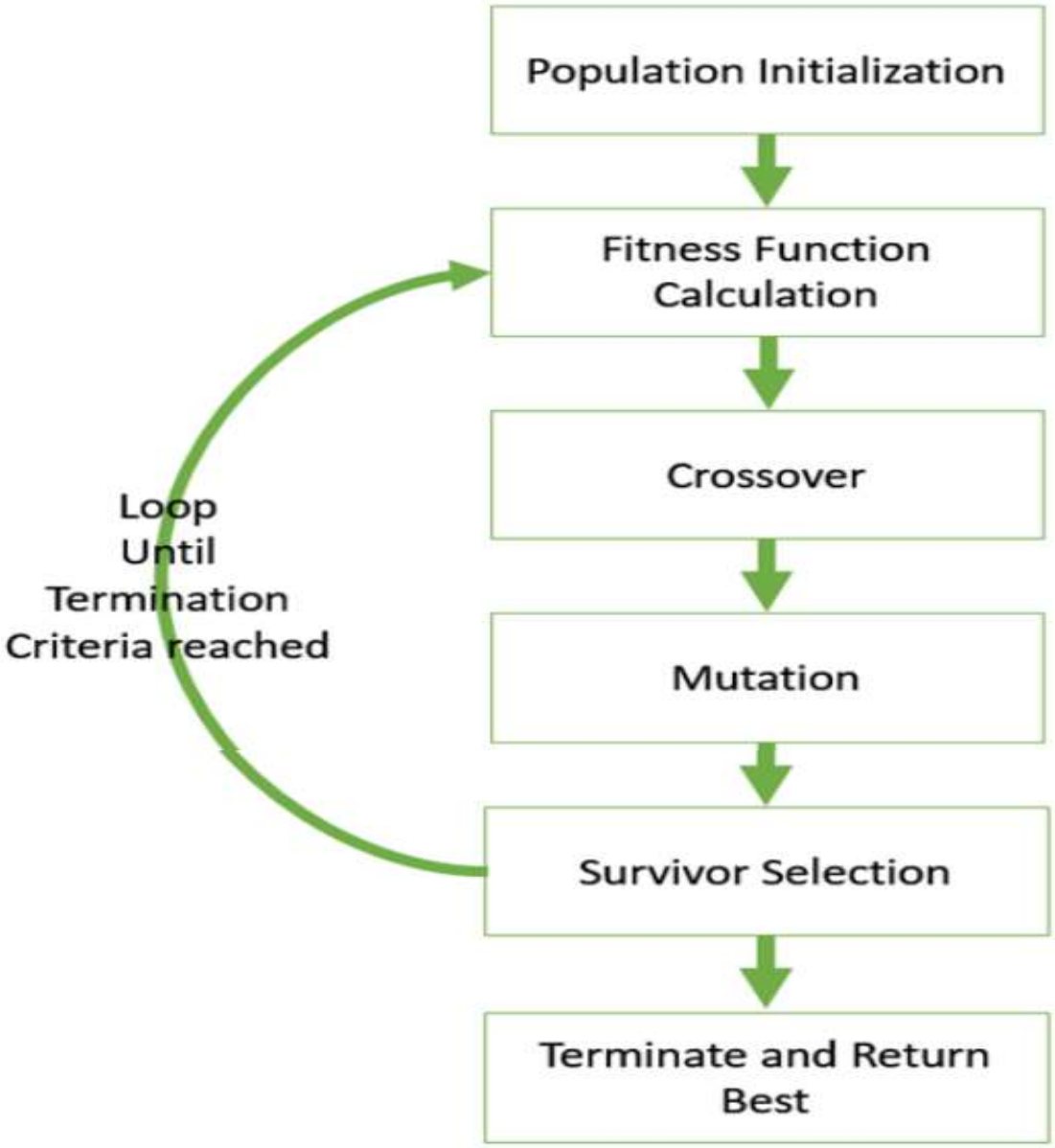

0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1
Chromosome
0
Gene
0
Allele
POPULATION
(set of chromosomes)

Basic Structure

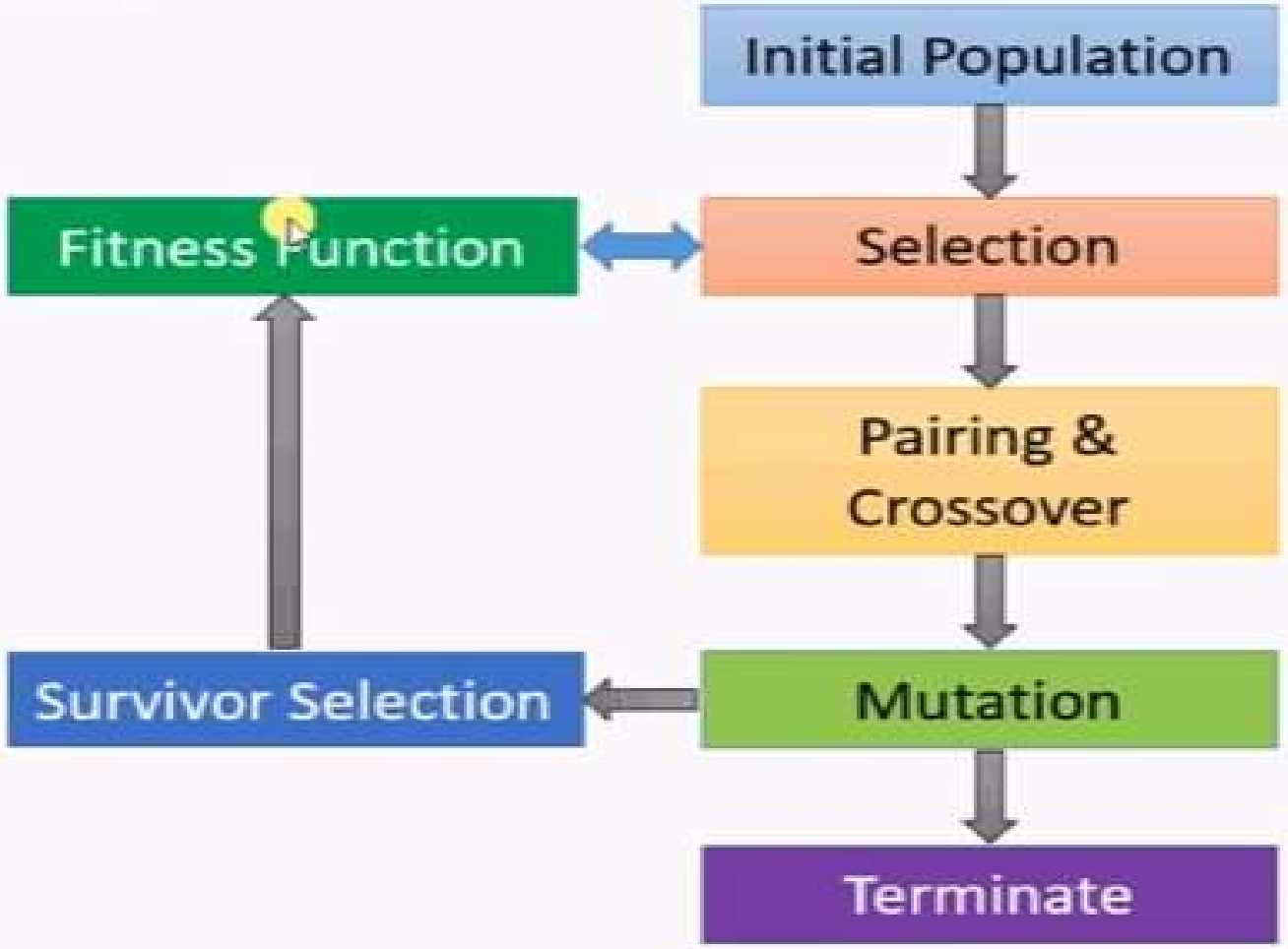
The basic structure of a GA is as follows –

We start with an initial population (which may be generated at random or seeded by other heuristics), select parents from this population for mating. Apply crossover and mutation operators on the parents to generate new off-springs. And finally these off-springs replace the existing individuals in the population and the process repeats. In this way genetic algorithms actually try to mimic the human evolution to some extent.

Each of the following steps are covered as a separate chapter later in this tutorial.


Population Initialization
Fitness Function Calculation
Crossover
Mutation
Survivor Selection
Terminate and Return Best
Loop Until Termination Criteria reached

Concept


Initial Population
Selection
Pairing &
Crossover
Mutation
Terminate
Survivor Selection
Fitness Function

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.

Fitness Function

The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a **fitness score** to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection

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

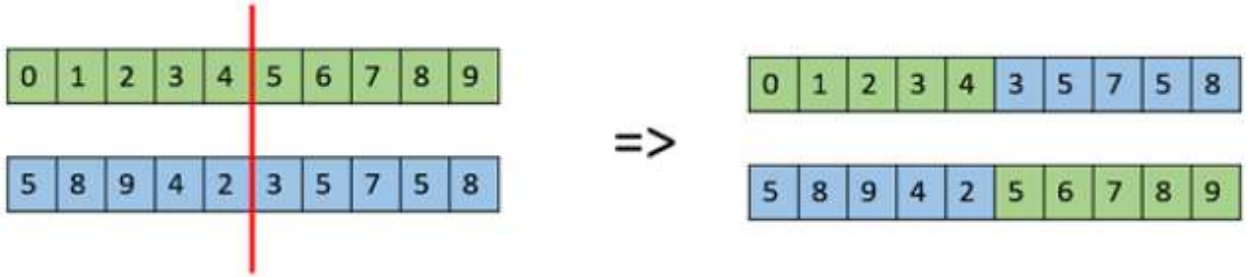
Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover Operators

In this section we will discuss some of the most popularly used crossover operators. It is to be noted that these crossover operators are very generic and the GA Designer might choose to implement a problem-specific crossover operator as well.

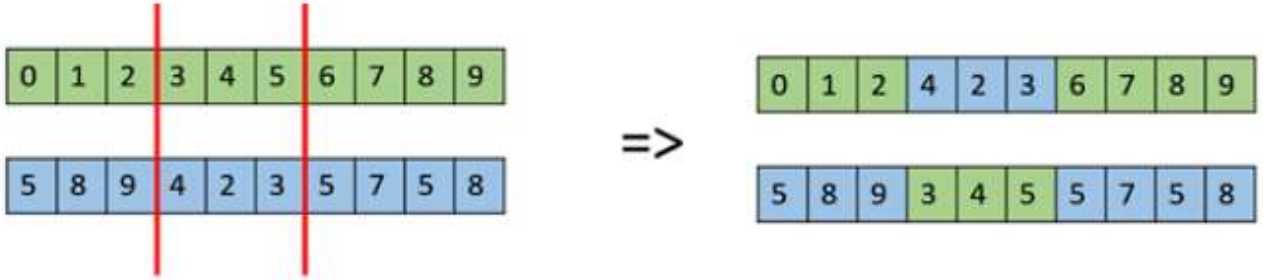
One Point Crossover

In this one-point crossover, a random crossover point is selected and the tails of its two parents are swapped to get new off-springs.



Multi Point Crossover

Multi point crossover is a generalization of the one-point crossover wherein alternating segments are swapped to get new off-springs.

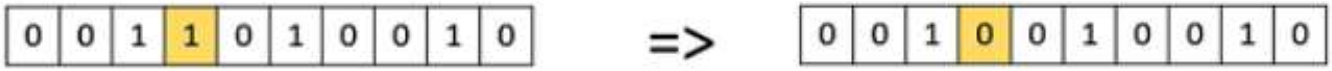


Mutation Operators

In this section, we describe some of the most commonly used mutation operators. Like the crossover operators, this is not an exhaustive list and the GA designer might find a combination of these approaches or a problem-specific mutation operator more useful.

Bit Flip Mutation

In this bit flip mutation, we select one or more random bits and flip them. This is used for binary encoded GAs.

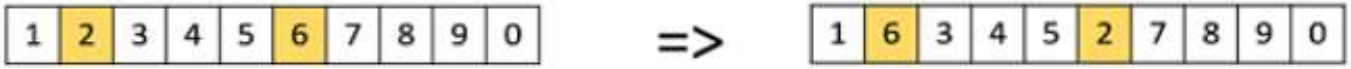


Random Resetting

Random Resetting is an extension of the bit flip for the integer representation. In this, a random value from the set of permissible values is assigned to a randomly chosen gene.

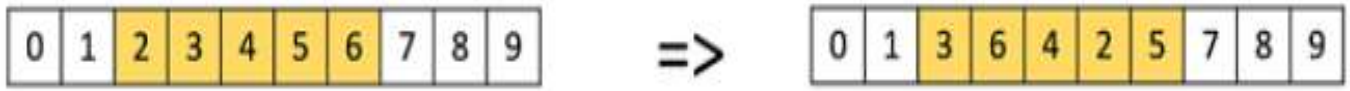
Swap Mutation

In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.



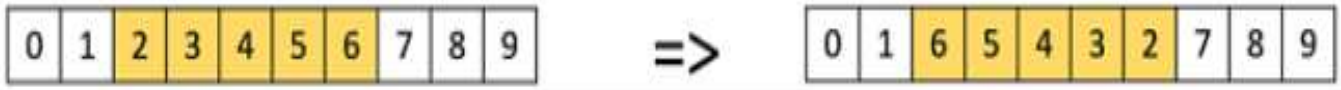
Scramble Mutation

Scramble mutation is also popular with permutation representations. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.



Inversion Mutation

In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.



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.

MOTIVATION

Genetic algorithms (GAS) provide a learning method motivated by an analogy to biological evolution.

Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAS generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses.

At each step, a collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses.

The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next.

The popularity of GAS is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAS can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

- This chapter describes the genetic algorithm approach, illustrates its use, and examines the nature of its hypothesis space search.
- It also describe a variant called genetic programming, in which entire computer programs are evolved to certain fitness criteria.
- Genetic algorithms and genetic programming are two of the more popular approaches in a field that is sometimes called evolutionary computation.

Representing Hypotheses

- Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover.

- The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and postcondition.

- To see how if-then rules can be encoded by bit strings,
- First consider how we might use a bit string to describe a constraint on the value of a single attribute.
- To pick an example, consider the attribute ***Outlook,*** which can take on any of the three values ***Sunny, Overcast,*** or ***Rain.***
- One obvious way to represent a constraint on ***Outlook*** is to use a bit string of length three, in which each bit position corresponds to one of its three possible values.
- Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value.

For example, the string 010 represents the constraint that ***Outlook*** must take on the second of these values or ***Outlook = Overcast.***

Similarly, the string 011 represents the more general constraint that allows two possible values, or ***(Outlook = Overcast v Rain).***

Note 111 represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on.

- Given this method for representing constraints on a single attribute, conjunctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings.
- For example, consider a second attribute, ***Wind,*** that can take on the value ***Strong*** or ***Weak.***
- ***A*** rule precondition such as

  (Outlook = Overcast ∨ Rain) A (Wind = Strong)

can then be represented by the following bit string of length five:

- ***Outlook Wind***

  011 10

Rule postconditions (such as ***Play Tennis = yes)*** can be represented in a similar fashion.

Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition.

Rule postconditions (such as ***Play Tennis = yes)*** can be represented in a similar fashion.

Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition.

• For example, the rule

IF Wind = Strong THEN ***PlayTennis = yes***

would be represented by the string

Outlook Wind PlayTennis

111 10 10

where the first three bits describe the "don't care" constraint on ***Outlook,***

the next two bits describe the constraint on ***Wind,***

and the final two bits describe the rule postcondition

(here we assume ***PlayTennis*** can take on the values ***Yes*** or ***No).***

- In designing a bit string encoding for some hypothesis space, it is useful to arrange for every syntactically legal bit string to represent a well-defined hypothesis.
- To illustrate, note in the rule encoding in the above paragraph the bit string

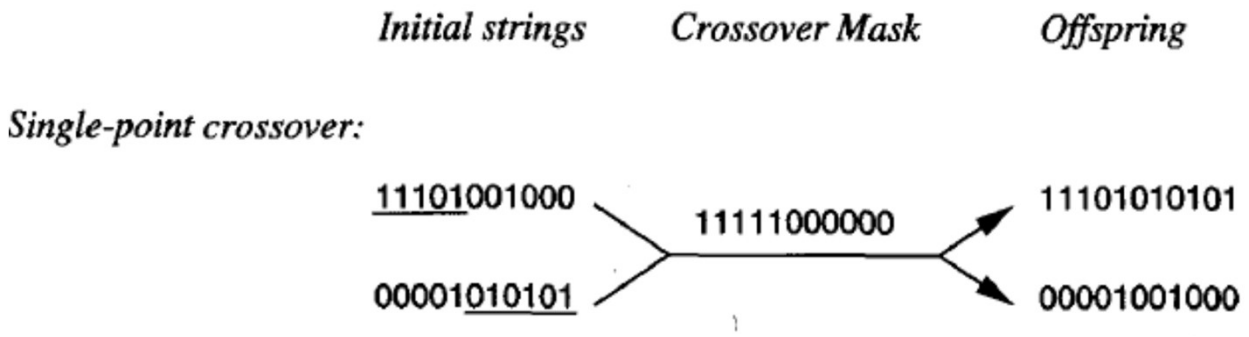
  111 10 11 represents a rule whose postcondition does not constrain the target attribute ***PlayTennis.***

- If we wish to avoid considering this hypothesis, we may employ a different encoding (e.g., allocate just one bit to the ***PlayTennis*** postcondition to indicate whether the value is ***Yes*** or ***No),*** alter the genetic operators so that they explicitly avoid constructing such bit strings, or simply assign a very low fitness to such bit strings

- The generation of successors in a GA is determined by a set of operators that recombine and mutate selected members of the current population.
- These operators correspond to idealized versions of the genetic operations found in biological evolution. The two most common operators are ***crossover*** and ***mutation.***
- 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 Consider the topmost of the two offspring in this case.



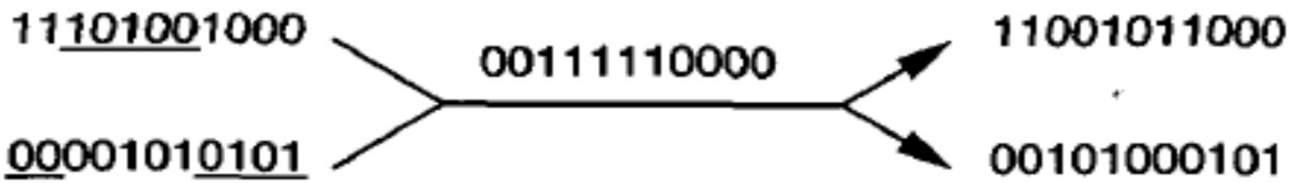
This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the crossover mask 11 11 1000000 specifies these choices for each of the bit positions.
The second offspring uses the same crossover mask, but switches the roles of the two parents. Therefore, it contains the bits that were not used by the first offspring.

In single-point crossover, the crossover mask is always constructed so that it begins with a string containing ***n*** contiguous 1s, followed by the necessary number of 0s to complete the string.

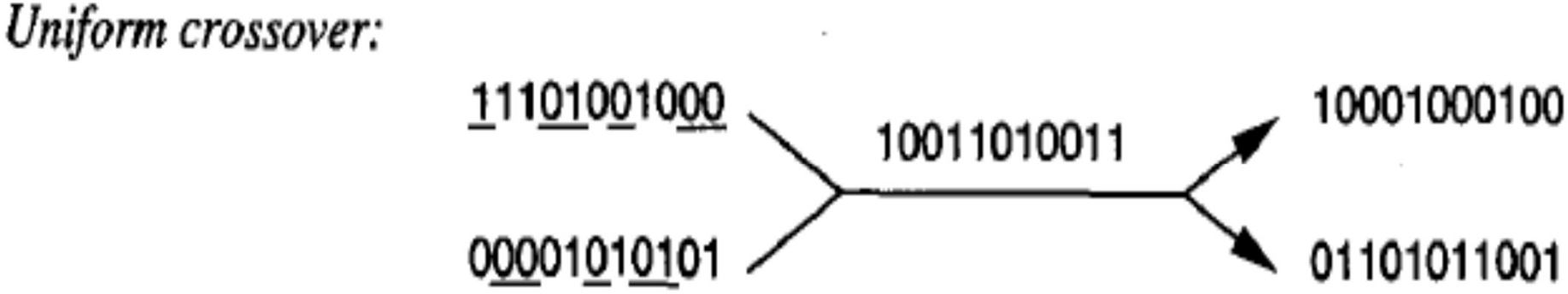
This results in offspring in which the first n bits are contributed by one parent and the remaining bits by the second parent. Each time the single-point crossover operator is applied the crossover point n is chosen at random, and the crossover mask is then created and applied.

- In ***two-point crossover,*** offspring are created by substituting intermediate segments of one parent into the middle of the second parent string.
- Put another way, the crossover mask is a string beginning with ***no*** zeros, followed by a contiguous string of ***nl*** ones, followed by the necessary number of zeros to complete the string.
- Each time the two-point crossover operator is applied, a mask is generated by randomly choosing the integers ***no*** and ***nl.***
- For instance, the offspring are created using a mask for which ***no*** = 2 and ***n*** 1 = 5.
- Again, the two offspring are created by switching the roles played by the two parents.

Two-point crossover:

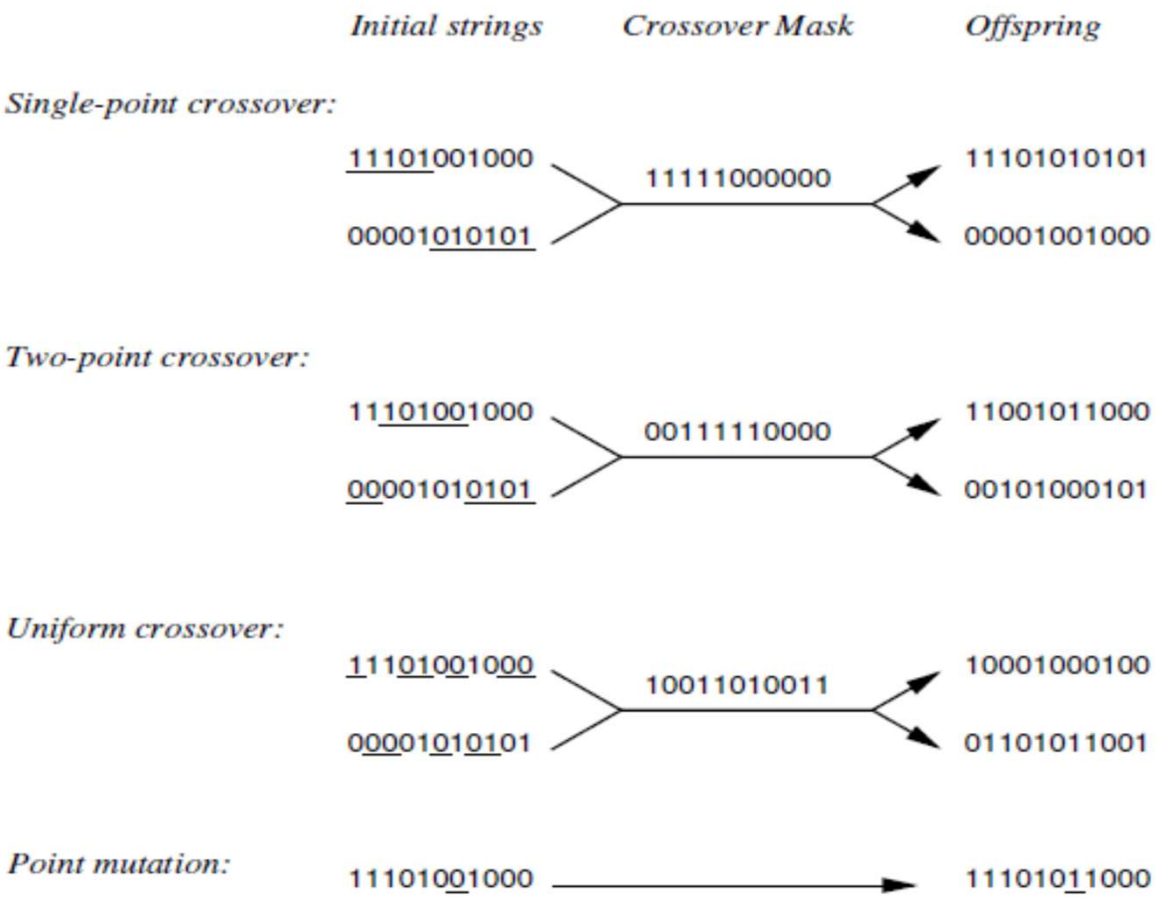


- ***Uniform crossover*** combines bits sampled uniformly from the two parents. In this case the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others.



- *Single-point crossover* splits the strings into two parts and merges them.
- *Two-point crossover* splits the strings into three parts and merges them.
- *Uniform crossover* each bit is chosen with equal and independent probability.
- *Point mutation* makes small random changes to the string.

Operators for Genetic Algorithms



Genetic Algorithms (GA) use principles of natural evolution. There are Five important features of GA:

Fitness Function *represents the main requirements of the desired solution of a problem (i.e. cheapest price, shortest route, most compact arrangement, etc). This function calculates and returns the fitness of an individual solution.*

Encoding *possible solutions of a problem are considered as individuals in a population. If the solutions can be divided into a series of small steps (building blocks), then these steps are represented by genes and a series of genes (a chromosome) will encode the whole solution. This way different solutions of a problem are represented in GA as chromosomes of individuals.*

Selection *operator defines the way individuals in the current population are selected for reproduction. There are many strategies for that (e.g. roulette–wheel, ranked, tournament selection, etc), but usually the individuals which are more fit are selected.*

Crossover *operator defines how chromosomes of parents are mixed in order to obtain genetic codes of their offspring (e.g. one–point, two–point, uniform crossover, etc). This operator implements the inheritance property (offspring inherit genes of their parents).*

Mutation *operator creates random changes in genetic codes of the offspring. This operator is needed to bring some random diversity into the genetic code. In some cases GA cannot find the optimal solution without mutation operator (local maximum problem).*

- Suppose a genetic algorithm uses chromosomes of the form x = abcdefgh with a fixed length of eight genes. Each gene can be any digit between 0 and 9. Let the fitness of individual x be calculated as:

$$f(x) = (a+b) - (c+d) + (e+f) - (g+h) \,,$$

and let the initial population consist of four individuals with the following chromosomes:

$$\begin{aligned} x_1 &= 65413532 \\ x_2 &= 87126601 \\ x_3 &= 23921285 \\ x_4 &= 41852094 \end{aligned}$$

a) Evaluate the fitness of each individual, showing all your workings, and arrange them in order with the fittest first and the least fit last.

Answer:

$$
\begin{aligned}
f(x_1) &= (6+5)-(4+1)+(3+5)-(3+2) = 9 \\
f(x_2) &= (8+7)-(1+2)+(6+6)-(0+1) = 23 \\
f(x_3) &= (2+3)-(9+2)+(1+2)-(8+5) = -16 \\
f(x_4) &= (4+1)-(8+5)+(2+0)-(9+4) = -19
\end{aligned}
$$

The order is x_2, x_1, x_3 and x_4.

b) Perform the following crossover operations:

i) Cross the fittest two individuals using one–point crossover at the middle point.

Answer: *One–point crossover on x_2 and x_1:*

$$\begin{array}{rl} x_2 = & \mathbf{8\,7\,1\,2} \,\big|\, \mathbf{6\,6\,0\,1} \\ x_1 = & 6\,5\,4\,1 \,\big|\, 3\,5\,3\,2 \end{array} \Rightarrow \begin{array}{rl} O_1 = & \mathbf{8\,7\,1\,2}3\,5\,3\,2 \\ O_2 = & 6\,5\,4\,1\mathbf{6\,6\,0\,1} \end{array}$$

ii) Cross the second and third fittest individuals using a two–point crossover (points b and f).

Answer: *Two–point crossover on x_1 and x_3*

$$\begin{array}{rl} x_1 = & \mathbf{6\,5} \,\big|\, \mathbf{4\,1\,3\,5} \,\big|\, \mathbf{3\,2} \\ x_3 = & 2\,3 \,\big|\, 9\,2\,1\,2 \,\big|\, 8\,5 \end{array} \Rightarrow \begin{array}{rl} O_3 = & \mathbf{6\,5}9\,2\,1\,\mathbf{2\,3\,2} \\ O_4 = & 2\,3\mathbf{4\,1\,3\,5}8\,5 \end{array}$$

iii) Cross the first and third fittest individuals (ranked 1st and 3rd) using a uniform crossover.

Answer: *In the simplest case uniform crossover means just a random exchange of genes between two parents. For example, we may swap genes at positions a, d and f of parents x_2 and x_3:*

$$\begin{aligned} x_2 &= \underline{8}71\underline{2}6\underline{6}01 \\ x_3 &= \underline{2}39\underline{2}1\underline{2}85 \end{aligned} \quad \Rightarrow \quad \begin{aligned} O_5 &= 27126201 \\ O_6 &= 83921685 \end{aligned}$$

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 $Fitness(h)$

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

- 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 $\frac{r \cdot p}{2}$ 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.

- 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 ***hi*** is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)} \qquad (9.1).$$

- 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.

The parent hypotheses are chosen probabilistically from the current population, again using the probability function given by Equation (9.1).

Fitness Function and Selection

- The fitness function defines the criterion for
  - Ranking potential hypotheses
  - Probabilistically selecting them for inclusion in the next generation population.
- If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples.
- 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 (e.g., when the bit string represents a collection of if-then rules that will be chained together to control a robotic device), the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.

- In our prototypical GA , 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 as seen in Equation .
- This method is sometimes Called ***fitness proportionate selection,*** or roulette wheel selection.
- Other methods for using fitness to select hypotheses have also been proposed.
- For example, in ***tournament selection,*** two hypotheses are first chosen at random from the current population. With some predefined probability p the more fit of these two is then selected, and with probability (1 - p) the less fit hypothesis is selected. ***Tournament selection*** often yields a more diverse population than ***fitness proportionate selection*** (Goldberg and Deb 1991).
- In another method called ***rank selection,*** the hypotheses in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

- GAS have been applied to a number of optimization problems outside machine learning, including problems such as circuit layout and job-shop scheduling.
- Within machine learning, they have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems.
- To illustrate the use of GAS for concept learning, we briefly summarize the GABIL system described by DeJong et al. (1993).

- **Representation.** Each hypothesis in GABIL corresponds to a disjunctive set of propositional rules, encoded as described in Section 9.2.1.
- In particular, the hypothesis space of rule preconditions consists of a conjunction of constraints on a fixed set of attributes, as described in that earlier section.
- To represent a set of rules, the bit-string representations of individual rules are concatenated.

- To illustrate, consider a hypothesis space in which rule preconditions are conjunctions of constraints over two boolean attributes, ***a1*** and ***a2***. The rule postcondition is described by a single bit that indicates **the** predicted value of the target attribute c.
- Thus, the hypothesis consisting of the two rules

$$\text{IF } a_1 = T \wedge a_2 = F \text{ THEN } c = T; \qquad \text{IF } a_2 = T \text{ THEN } c = F$$

would be represented by the string

a_1	a_2	c	a_1	a_2	c
10	01	1	11	10	0

- Genetic operators. GABIL uses the standard mutation operator of Table 9.2, in which a single bit is chosen at random and replaced by its complement.
- The crossover operator that it uses is a fairly standard extension to the two-point crossover operator described in Table 9.2.

- In particular, to accommodate the variable-length bit strings that encode rule sets, and to constrain the system so that crossover occurs only between like sections of the bit strings that encode rules, the following approach is taken.
- To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string
- Let d1 (d2) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left.
- The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same dl and d2 value.

- For example, if the two parent strings are.

	a_1	a_2	c	a_1	a_2	c
h_1 :	10	01	1	11	10	0

and

	a_1	a_2	c	a_1	a_2	c
h_2 :	01	11	0	10	01	0

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

	a_1	a_2	c	a_1	a_2	c
h_1 :	1[0	01	1	11	1]0	0

where "[" and "]" indicate crossover points, then $d_1 = 1$ and $d_2 = 3$. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions $\langle 1, 3\rangle$, $\langle 1, 8\rangle$, and $\langle 6, 8\rangle$. If the pair $\langle 1, 3\rangle$ happens to be chosen,

	a_1	a_2	c	a_1	a_2	c
h_2 :	0[1	1]1	0	10	01	0

then the two resulting offspring will be

	a_1	a_2	c
h_3 :	11	10	0

and

	a_1	a_2	c		a_1	a_2	c		a_1	a_2	c
h_4 :	00	01	1		11	11	0		10	01	0

As this example illustrates, this crossover operation enables offspring to contain a different number of rules than their parents, while assuring that all bit strings generated in this fashion represent well-defined rule sets.

- **Fitness function.** The fitness of each hypothesized rule set is based on its classification accuracy over the training data. In particular, the function used to measure fitness is

$$Fitness(h) = (correct(h))^2$$

where $correct(h)$ is the percent of all training examples correctly classified by hypothesis h.

HYPOTHESIS SPACE SEARCH

- As illustrated above, GAS employ a randomized beam search method to seek a maximally fit hypothesis.
- This search is quite different from that of other learning methods we have considered in this book.
- To contrast the hypothesis space search of GAS with that of neural network BACKPROPAGATION for example, the gradient descent search in BACKPROPAGATION moves smoothly from one hypothesis to a new hypothesis that is very similar.
- In contrast, the GA search can move much more abruptly, replacing a parent hypothesis by an offspring that may be radically different from the parent. Note the GA search is therefore less likely to fall into the same kind of local minima that can plague gradient descent methods.

- One practical difficulty in some GA applications is the problem of ***crowding.***
- Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and very similar individuals take over a large fraction of the population.
- The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA.
- Several strategies have been explored for reducing crowding.

- One approach is to alter the selection function, using criteria such as tournament selection or rank selection in place of fitness proportionate roulette wheel selection.
- A related strategy is "fitness sharing," in which the measured fitness of an individual is reduced by the presence of other, similar individuals in the population.
- A third approach is to restrict the kinds of individuals allowed to recombine to form offspring.
- For example, by allowing only the most similar individuals to recombine, we can encourage the formation of clusters of similar individuals, or multiple "subspecies" within the population.
- A related approach is to spatially distribute individuals and allow only nearby individuals to recombine.
- Many of these techniques are inspired by the analogy to biological evolution

Population Evolution and the Schema Theorem

- It is interesting to ask whether one can mathematically characterize the evolution over time of the population within a GA.
- The schema theorem of Holland (1975) provides one such characterization.
- It is based on the concept of ***schemas,*** or patterns that describe sets of bit strings.
- To be precise, a schema is any string composed of 0s, 1s, and *'s.
- Each schema represents the set of bit strings containing the indicated 0s and 1s, with each **"*"** interpreted as a "don't care."

1. $S_1 = 000000$ (every bit is defined, only one string)
2. $S_2 = 111111$ (every bit is defined, only one string)
3. $S_3 = 101010$ (every bit is defined, only one string)
4. $S_4 = 0****0$ (only beginning and final bits are defined, the rest middle 4 positions may be filled by 16 z^4 = possible strings)

For example, the schema 0*10 represents the set of bit strings that includes exactly 0010 and 01 10.
An individual bit string can be viewed as a representative of each of the different schemas that it matches.
For example, the bit string 0010 can be thought of as a representative of **24** distinct

schemas including 00**, O* 10, ****, etc.

- Similarly, a population of bit strings can be viewed in terms of the set of schemas that it represents and the number of individuals associated with each of these schema

Length and order of a Schema

The *defining length d(S)* of a schema *S* is the distance between the first defined bit and the last defined bit i.e. it is the distance between first and last non * gene is schema S.

Example 2: d(101010) = 5;
d(**10**) = 1;
d(**1*1*) = 2.

Thus length of a schema is position (last defined) – position (first defined).

The *order o(s)* of a schema *s* is the number of *defined* bits it contains, i.e. order is the number of non * genes in schema S.

Example 3: o(101010) = 6;
o(**10**) = 2;
o(**1*1*) = 2; o(**1*1*) = 2.

Here it can be noted that for cardinality, k, there are $(k+1)\ell$ schema in string of length l.

E1) Write the schema for the gene sequence $\{0111000\}$ and $\{1110011\}$.

E2) Write at least 4 chromosome set, which are identified by schema $S = (01*1*)$.

E3) Find the length and order of the following schema.

i) $S_1 = (0**11*0**)$

ii) $S_2 = (*11*0**)$

iii) $S_3 = (***0***)$

E1) $S = (*11*0**)$

E2) The chromosome sets are (01010), (01011), (01110) and (01111).

E3) i) $d(S_1) = 7 - 1 = 6$

$0(S_1) = 4$

ii) $d(S_2) = 5 - 2 = 3$

$0(S_2) = 3$

iii) $d(S_3) = 4 - 4 = 0$

$0(S_3) = 1$

E4) Mutation is used to promote intensification (where local optimisation is required) and of course, to promote diversity.

Genetic Algorithms at glance

- A “population” of binary strings, called “chromosomes”.
- The “population” evolves using kind of “natural selection” together with the genetics-inspired operators of crossover, mutation, and inversion.
- Bits in a “chromosome” represent genes, and each “gene” is an instance of a particular “allele”, 0 or 1.
- The selection operator chooses those chromosomes in the population that will be allowed to reproduce, and on average the fitter chromosomes produce more offspring than the less fit ones.
- Crossover exchange subparts of two chromosomes
- Mutation randomly changes the allele values of some locations in the chromosome.

Basic Structure

1. Randomly generate initial population of n strings ("chromosomes")
2. Evaluate the fitness of each string in the population
3. Repeat the following steps until next generation of n individual strings produced
   a. Select pair of parent chromosomes from current population according to their fitness, i.e. chromosomes with higher fitness are selected more often
   b. Apply crossover (with probability)
   c. Apply mutation (with probability of occurrence)
4. Apply generational replacement
5. Go to 2 or terminate if termination condition met

Schema theorem

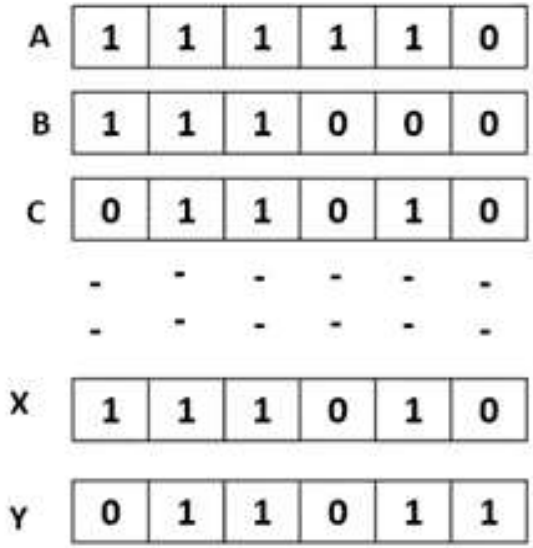
- Derived from original work by Prof. John Holland.
- To investigate Mathematical foundation of Genetic Algorithms(GA)
- Serves as the analysis tool for the GA process
- Shows the expectation of schema survival
- Applicable to a canonical GA
  - binary representation
  - fixed length individuals
  - fitness proportional selection
  - single point crossover
  - gene-wise mutation

What schema Theorem says?

- Schema theorem mathematically characterize the evolution of GA strings (Population) over the time in terms of number of instances representing each schema.
- Expected number of string to be represented by the schema H.
- The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema

Population of binary strings

- Lets look at population of individuals created randomly and represented using binary codes (binary representation).



Schema templates

- Templates specifying groups (sets) of "similar" chromosomes
- Its a set of binary strings that match the template for schema **H**.
- A template is made up of 1s, 0s, and *s where * is the 'don't care' symbol that matches either 0 or 1
- For comparison lets consider following two schema templates

H_1 :

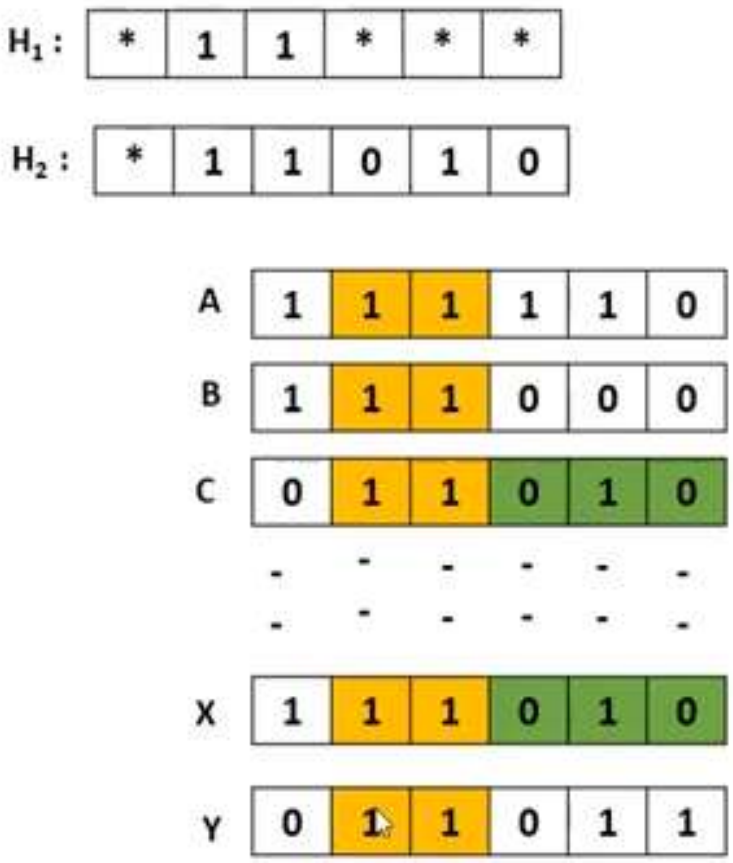
*	1	1	*	*	*

H_2 :

*	1	1	0	1	0

Schema template…

- How many individuals from the population of individuals are going to follow a particular template. Lets say H_1, here H_1 says * 1 1 ***. Now if we see the population , the first string **A** follows H_1 , second sting **B** also follows H_1 so we can see that all the 5 strings except the hidden one in the population follow the template H_1
- Similarly H_2 is * 11010 Which is followed by String **C** and **X**
- So all the strings visible here are going to follow template H_1 and two strings C and X follow H_2 also.



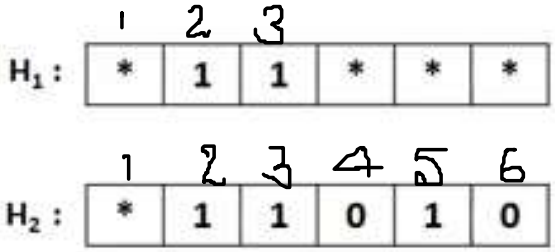
Terms

Order of Schema: O(H)

- The order of a schema is the number of its fixed bits, i.e. the number of bits that are not '*' in the schema H
- Example: if H = 10*1* then O(H) = 3

$O(H_1) = 2$

$O(H_2) = 5$



Defining Length of Schema : δ(H)

- The defining length is the distance between its first and the last fixed bits positions in a string.
- Example: $\delta(H_1) = 3-2= 1$

$\delta(H_2) = 6-2= 4$

if H = 1***** then $\delta(H) = 1 - 1 = 0$

❑ Two Terms:

- **Order of schema O(H):** No. of fixed positions (bits) present in a schema.
  For example: $O(H_1) = 2; O(H_2) = 5$

- **Defining length of schema δ(H):** Distance between the first and last fixed positions in a string.
  For example: $\delta(H_1) = 3\text{-}2 = 1; \delta(H_2) = 6\text{-}2 = 4$

GA operators

Reproduction (proportional selection)

Probability of selection is proportionate to fitness

Let **m(H, t):** Number of strings belonging to schema H at t^{th} generation.

m(H, t+1): No. of strings belonging to schema H at $(t+1)^{th}$ generation.

Let **N** : population size

***f*(H):** Schema fitness or Average fitness of the string represented by schema H at t^{th} generation

$\sum f$: Total fitness

Fitness of each string in the population

- Fitness: number of ones in the string

  $f_A = 5$

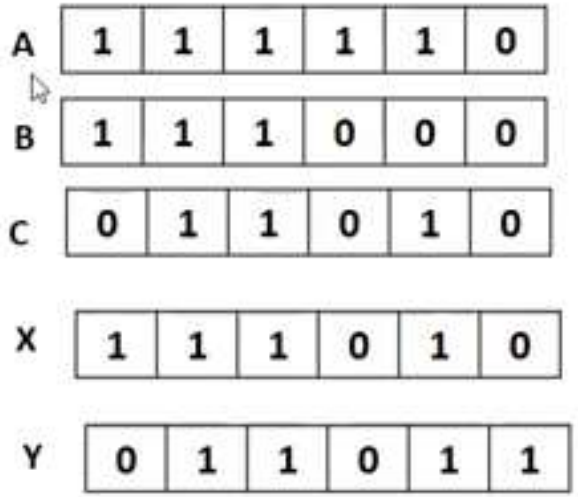
  $f_B = 3$

  $f_C = 3$

  $f_x = 4$

  $f_y = 4$

- Average fitness $\bar{f} = (19)/5 = 3.8$
- **The average fitness in the population of possible solutions increases with every generation.**



GA operators

❑ Reproduction (Proportionate Selection)

- **Probability α fitness**

  Let $m(H, t)$: No. of strings belonging to schema H at t^{th} generation

  $m(H, t+1)$: No. of strings belonging to schema H at $(t+1)^{th}$ Generation

  N: Population size

  f (H): Schema fitness or Average fitness of the strings represented by schema H at t^{th} generation

  Σf: Total fitness

$$m(H, t+1) = m(H, t) N \frac{f(H)}{\sum f}$$

$$m(H, t+1) = m(H, t) \frac{f(H)}{\bar{f}}$$

where $\bar{f} = \frac{\sum f}{N}$

❑ Crossover (Single-point)

Let p_c: Probability of crossover

L: String length

A schema is destroyed if the crossover site falls within the defining length

Probability of destruction = $p_c \frac{\delta(H)}{L-1}$

Probability of survival, $p_s = 1 - p_c \frac{\delta(H)}{L-1}$

Combining the effect of reproduction and crossover, we get

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{L-1} \right]$$

❑ Mutation (Bit–wise mutation)

To protect a schema, mutation should not occur at the fixed bits

Let p_m : probability of mutation

probability of destruction = p_m

probability of survival = $1 - p_m$

Probability of survival considering all the fixed bits in a schema,

$$p_s = (1 - p_m)(1 - p_m) \ldots\ldots\ldots O(H)$$

$$= (1 - p_m)^{O(H)}$$

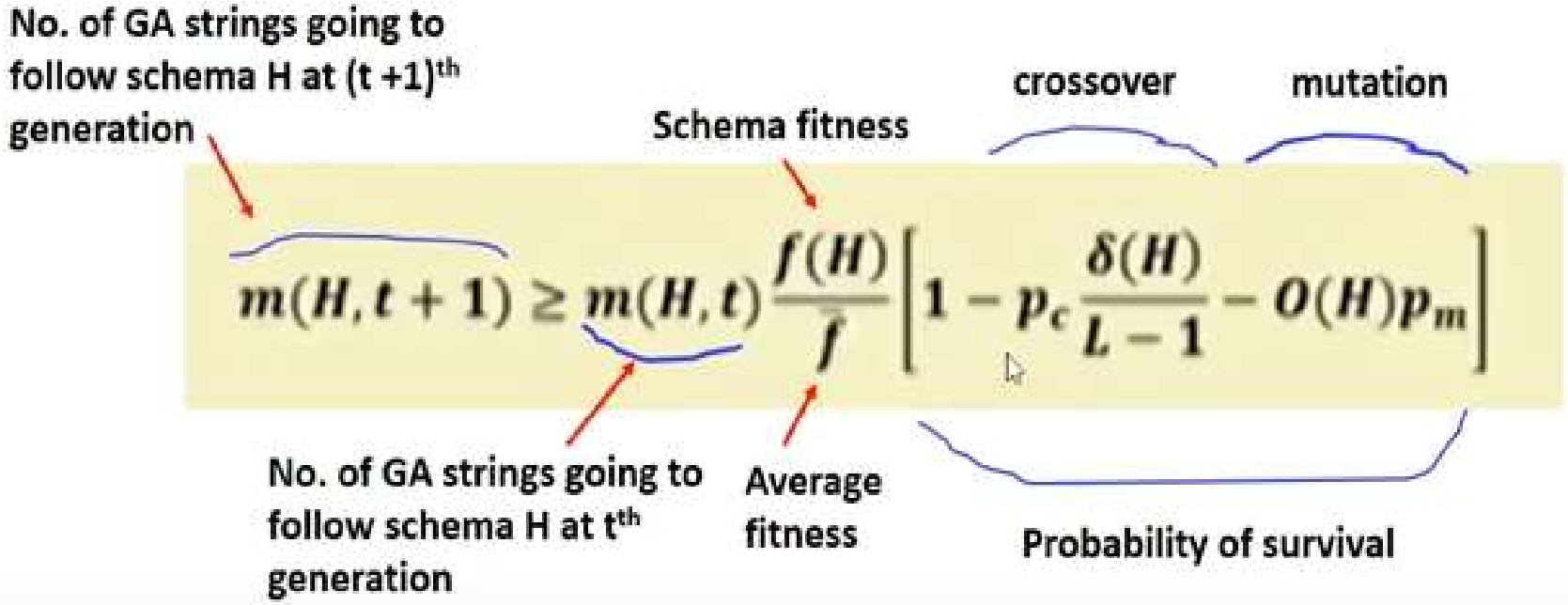
$$= 1 - O(H)\, p_m \qquad \text{as } p_m \ll 1$$

Considering the contributions of all three operators, we get

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{L-1} - O(H)p_m\right]$$

All three operators

- Considering the contributions of all three operators we get,



Good Schema

$f(H) > \overline{f}$	whose average fitness is more than the average fitness of the population
$\delta(H)$ = **short**	having short defining length
O(H) = low	low order

The schema having these properties will be the building block of GA.

A Numerical Example

An initial random population of size N=10 of a binary-coded GA is given below.

100100
010100
010111
111000
010110
100110
011001
100101

110011
011001

The fitness of a GA-string is to assumed to be equal to its decoded value. Calculate the expected number of strings to be represented by the schema H: *1**1*, at the end of first generation, considering a single-point crossover of probability $p_c = 0.9$ and a bit-wise mutation of probability $p_m = 0.01$.

Solution:

2^5	2^4	2^3	2^2	2^1	2^0	Fitness/ Decoded value
1	0	0	1	0	0	36
0	1	0	1	0	0	20
0	1	0	1	1	1	23
1	1	1	0	0	0	56
0	1	0	1	1	0	22
1	0	0	1	1	0	38
0	1	1	0	0	1	25
1	0	0	1	0	1	37
1	1	0	0	1	1	51
0	1	1	0	0	1	25

Total fitness = 333

Average fitness

$$\bar{f} = \frac{333}{10} = 33.3$$

Schema H: *11* is followed by 3rd , 5th and 9th strings**

$$f(H) = \frac{23 + 22 + 51}{3} = 32$$

$$m(H, 1) = 3$$

$$p_c = 0.9$$

$$p_m = 0.01$$

$$L = 6$$

$$\delta(H) = 5 - 2 = 3$$

$$O(H) = 2$$

$$m(H,2) \geq m(H,1)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{L-1} - O(H)p_m\right]$$

$$\geq 3 \times \frac{32}{33.3}\left[1 - 0.9 \times \frac{3}{5} - 2 \times 0.01\right]$$

$$\geq 1.268(say)$$

Probability of distraction is also high, hence the number of string that follow the given schema in the next generation is less. Hence, this schema is not a good schema

Schema Theorem.

- **Highly fit,**
  short defining length,
  low order schemas
  propagate from generation to generation and give exponential increase of samples to the observed best.

- Thus, **every schema represents a subset of chromosomes with a particular average fitness** and
- **selection more often chooses for reproduction chromosomes containing highly fit schemata, so these schemata tend to stay and reproduce in the population from generation to generation.**
- **Although the schema 111*111* has higher fitness, due to the long defining length it has more chances to be destroyed by crossover than the schema *****11* with shorter defining length.**

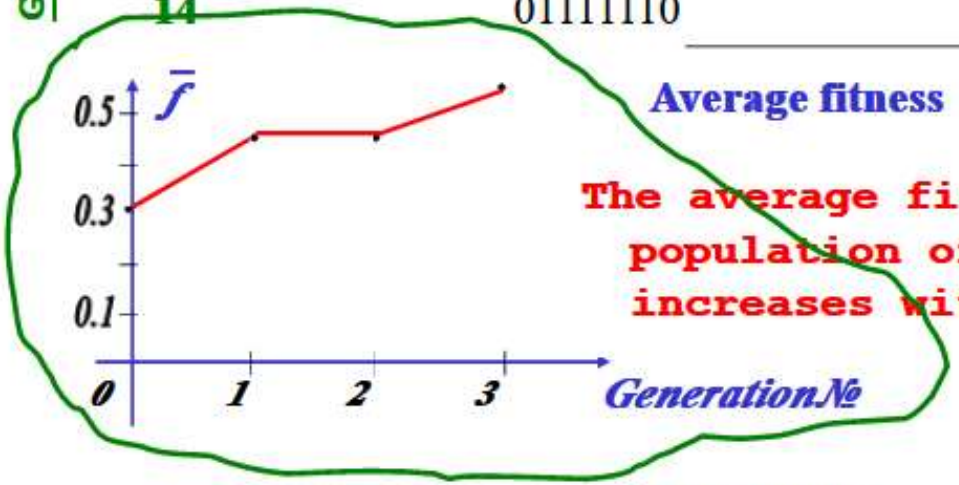
Example of a Genetic Algorithm.

2. **Evaluate** the fitness of each string in the population

Generation 3

Chromosome index	Chromosome string	Fitness = number of ones in the string
8	11100110	$f_8 = 5$
12	00101110	$f_{12} = 4$
13	11011110	$f_{13} = 6$
14	01111110	$f_{14} = 6$

Average fitness $\bar{f} = (5+4+6+6)/4 = \mathbf{5.25}$



The average fitness in the population of possible solutions increases with every generation.

Example

Schema Theorem: Highly fit, short defining length, low order schemas propagate from generation to generation and give exponential increase of samples to the observed best.

Strings	Fitness	Times to be selected for reproduction
A	5	3
B	3	1
C	3	1
X	4	2
Y	4	2

N=5

H_1 :

*	1	1	*	*	*

Building block

A	1	1	1	1	1	0
B	1	1	1	0	0	0
C	0	1	1	0	1	0
X	1	1	1	0	1	0
Y	0	1	1	0	1	1

It can be seen that the *11*** building block is present in all the strings. This schema represents a subset of chromosomes with the fitness not less than 3.

Example…

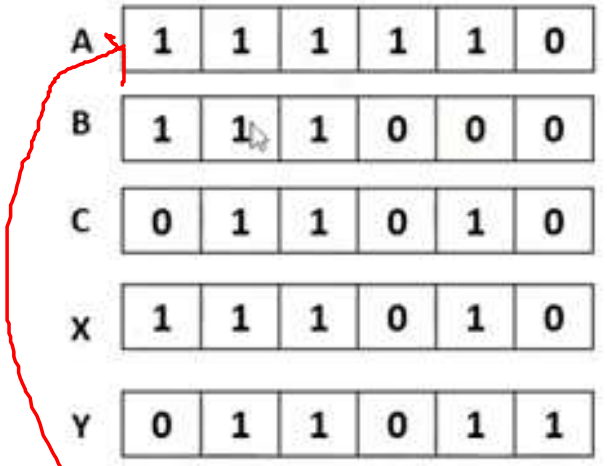
Strings	Fitness	Times to be selected for reproduction
A	5	3
B	3	1
C	3	1
X	4	2
Y	4	2

N=5

H_1 :

*	1	1	1	*	*

Building block



The schema *111** is present in the A string, and it represents a subset of chromosomes with the fitness no less than 5. Thus, every schema represents a subset of chromosomes with a particular average fitness, and selection more often chooses for reproduction chromosomes containing highly fit schemata, so these schemata tend to stay and reproduce in the population from generation to generation.

Example…

Strings	Fitness	Times to be selected for reproduction
A	5	3
B	3	1
C	3	1
X	4	2
Y	4	2

N=5

H_1 :

*	1	1	1	*	*

Building block

Schema Theorem: Highly fit, short defining length, low order schemas propagate from generation to generation and give exponential increase of samples to the observed best.

The schema *11*** block is present in all the strings. It has a defining length d and represents chromosomes with the fitness ≥ 3. The schema *111**is present in the First string .It has a defining length d^2 and represents chromosomes with the fitness ≥ 5. Although the schema *111*111 has higher fitness, due to the long defining length it has more chances to be destroyed by crossover than the schema *11*** with shorter defining length.

Example…

Strings	Fitness	Times to be selected for reproduction
A	5	3
B	3	1
C	3	1
X	4	2
Y	4	2

N=5

H_1 :

*	1	1	1	*	*

Building block

Schema Theorem: Highly fit, short defining length, low order schemas propagate from generation to generation and give exponential increase of samples to the observed best.

Irrelevant to the defining length, chances to be destroyed by mutation will rather depend on the number of non-* symbols in the schema, i.e. the order of the schema, although these chances are much smaller compare to the ones due to crossover, as the mutation probability p_m is usually much smaller than the probability of crossover p_c.

Example…

Strings	Fitness	Times to be selected for reproduction
A	5	3
B	3	1
C	3	1
X	4	2
Y	4	2

N=5

H_1 :

*	1	1	1	*	*

Building block

The schema *11*** is present in all the strings. It has an order = 2 and represents chromosomes with the fitness ≥ 2.The schema *111* is present in the First string. It has an order = 5 and represents chromosomes with the fitness ≥ 5. **Chances to be destroyed by mutation are equal to D_m= order × p_m. Thus, chances to be destroyed by mutation will be higher for the high order schema 111*111* . Though these chances are still much smaller than the ones due to crossover, as mutation probability p_m << p_c the probability of crossover .**