UNIT 1

INTRODUCTION

Ever since computers were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn-to improve automatically with experience-the impact would be dramatic.

- Imagine computers learning from medical records which treatments are most effective for new diseases
- Houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants.
- Personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper

A successful understanding of how to make computers learn would open up many new uses of computers and new levels of competence and customization

Some successful applications of machine learning

- Learning to recognize spoken words
- Learning to drive an autonomous vehicle
- Learning to classify new astronomical structures
- Learning to play world-class backgammon

Why is Machine Learning Important?

- Some tasks cannot be defined well, except by examples (e.g., recognizing people).
- Relationships and correlations can be hidden within large amounts of data. Machine Learning/Data Mining may be able to find these relationships.
- Human designers often produce machines that do not work as well as desired in the environments in which they are used.
- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans (e.g., medical diagnostic).
- Environments change over time.
- New knowledge about tasks is constantly being discovered by humans. It may be difficult to continuously re-design systems “by hand”.

WELL-POSED LEARNING PROBLEMS

Definition: A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

To have a well-defined learning problem, three features needs to be identified:

1. The class of tasks
2. The measure of performance to be improved
3. The source of experience

Examples

1. ***Checkers game:*** A computer program that learns to play ***checkers*** might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself.

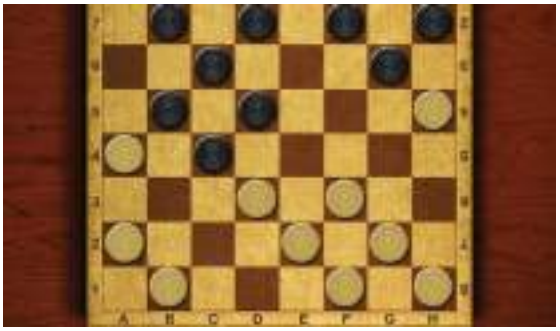


Fig: Checker game board

***A checkers learning problem*:**

- Task T: playing checkers
- Performance measure P: percent of games won against opponents
- Training experience E: playing practice games against itself

2. ***A handwriting recognition learning problem:***
    - Task T: recognizing and classifying handwritten words within images
    - Performance measure P: percent of words correctly classified
    - Training experience E: a database of handwritten words with given classifications
3. ***A robot driving learning problem*:**
    - Task T: driving on public four-lane highways using vision sensors
    - Performance measure P: average distance travelled before an error (as judged by human overseer)
    - Training experience E: a sequence of images and steering commands recorded while observing a human driver

DESIGNING A LEARNING SYSTEM

The basic design issues and approaches to machine learning are illustrated by designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament

1. Choosing the Training Experience
2. Choosing the Target Function
3. Choosing a Representation for the Target Function
4. Choosing a Function Approximation Algorithm
    1. Estimating training values
    2. Adjusting the weights
5. The Final Design

1. Choosing the Training Experience

- The first design choice is to choose the type of training experience from which the system will learn.
- The type of training experience available can have a significant impact on success or failure of the learner.

There are three attributes which impact on success or failure of the learner

1. Whether the training experience provides ***direct or indirect feedback*** regarding the choices made by the performance system.

    For example, in checkers game:
    In learning to play checkers, the system might learn from ***direct training examples*** consisting of ***individual checkers board states*** and ***the correct move for each***.

    Indirect training examples consisting of the ***move sequences*** and ***final outcomes*** of various games played. The information about the correctness of specific moves early in the game must be inferred indirectly from the fact that the game was eventually won or lost.

    Here the learner faces an additional problem of ***credit assignment***, or determining the degree to which each move in the sequence deserves credit or blame for the final outcome. Credit assignment can be a particularly difficult problem because the game can be lost even when early moves are optimal, if these are followed later by poor moves. Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

2. The degree to which the ***learner controls the sequence of training examples***

   For example, in checkers game:
   The learner might depends on the ***teacher*** to select informative board states and to provide the correct move for each.

   Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move.

   The learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with ***no teacher present***.

3. How well it represents the ***distribution of examples*** over which the final system performance P must be measured

   For example, in checkers game:
   In checkers learning scenario, the performance metric P is the percent of games the system wins in the world tournament.

   If its training experience E consists only of games played against itself, there is a danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested.
   It is necessary to learn from a distribution of examples that is different from those on which the final system will be evaluated.

2. Choosing the Target Function

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program.

Let's consider a checkers-playing program that can generate the legal moves from any board state.
The program needs only to learn how to choose the best move from among these legal moves. We must learn to choose among the legal moves, the most obvious choice for the type of information to be learned is a program, or function, that chooses the best move for any given board state.

1. Let ***ChooseMove*** be the target function and the notation is

   $$ChooseMove : B \rightarrow M$$

   which indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M.

ChooseMove is a choice for the target function in checkers example, but this function will turn out to be very difficult to learn given the kind of indirect training experience available to our system

2. An alternative target function is an ***evaluation function*** that assigns a ***numerical score*** to any given board state
   Let the target function V and the notation

$$V : B \rightarrow R$$

   which denote that V maps any legal board state from the set B to some real value. Intend for this target function V to assign higher scores to better board states. If the system can successfully learn such a target function V, then it can easily use it to select the best move from any current board position.

Let us define the target value V(b) for an arbitrary board state b in B, as follows:

- If b is a final board state that is won, then V(b) = 100
- If b is a final board state that is lost, then V(b) = -100
- If b is a final board state that is drawn, then V(b) = 0
- If b is a not a final state in the game, then V(b) = V(b'),

Where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game

3. Choosing a Representation for the Target Function

Let's choose a simple representation - for any given board state, the function ***c*** will be calculated as a linear combination of the following board features:

- x_1: the number of black pieces on the board
- x_2: the number of red pieces on the board
- x_3: the number of black kings on the board
- x_4: the number of red kings on the board
- x_5: the number of black pieces threatened by red (i.e., which can be captured on red's next turn)
- x_6: the number of red pieces threatened by black

Thus, learning program will represent as a linear function of the form

$$\hat{V}(b) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$$

Where,

- w_0 through w_6 are numerical coefficients, or weights, to be chosen by the learning algorithm.
- Learned values for the weights w1 through w6 will determine the relative importance of the various board features in determining the value of the board
- The weight w_0 will provide an additive constant to the board value

4. Choosing a Function Approximation Algorithm

In order to learn the target function **f** we require a set of training examples, each describing a specific board state b and the training value $V_{train}(b)$ for b.

Each training example is an ordered pair of the form $(b, V_{train}(b))$.

For instance, the following training example describes a board state b in which black has won the game (note $x_2 = 0$ indicates that red has no remaining pieces) and for which the target function value $V_{train}(b)$ is therefore +100.

$$((x_1=3, x_2=0, x_3=1, x_4=0, x_5=0, x_6=0), +100)$$

Function Approximation Procedure

1. Derive training examples from the indirect training experience available to the learner
2. Adjusts the weights w_i to best fit these training examples

1. Estimating training values

   A simple approach for estimating training values for intermediate board states is to assign the training value of $V_{train}(b)$ for any intermediate board state b to be $\hat{V}(Successor(b))$

   Where ,
   - $\hat{V}$ is the learner's current approximation to V
   - Successor(b) denotes the next board state following b for which it is again the program's turn to move

   Rule for estimating training values

$$\mathbf{V_{train}(b) \leftarrow \hat{V}(Successor(b))}$$

2. Adjusting the weights

Specify the learning algorithm for choosing the weights w_i to best fit the set of training examples $\{(b, V_{train}(b))\}$

A first step is to define what we mean by the bestfit to the training data.

One common approach is to define the best hypothesis, or set of weights, as that which minimizes the squared error E between the training values and the values predicted by the hypothesis.

$$E \equiv \sum_{\{b, V_{train}(b)\} \in \text{training examples}} (V_{train}(b) - \hat{V}(b))^2$$

Several algorithms are known for finding weights of a linear function that minimize E. One such algorithm is called the ***least mean squares, or LMS training rule***. For each observed training example it adjusts the weights a small amount in the direction that reduces the error on this training example

LMS weight update rule :- For each training example $(b, V_{train}(b))$

Use the current weights to calculate $\hat{V}(b)$

For each weight w_i, update it as

$$w_i \leftarrow w_i + \eta\,(V_{train}(b) - \hat{V}(b))\,x_i$$

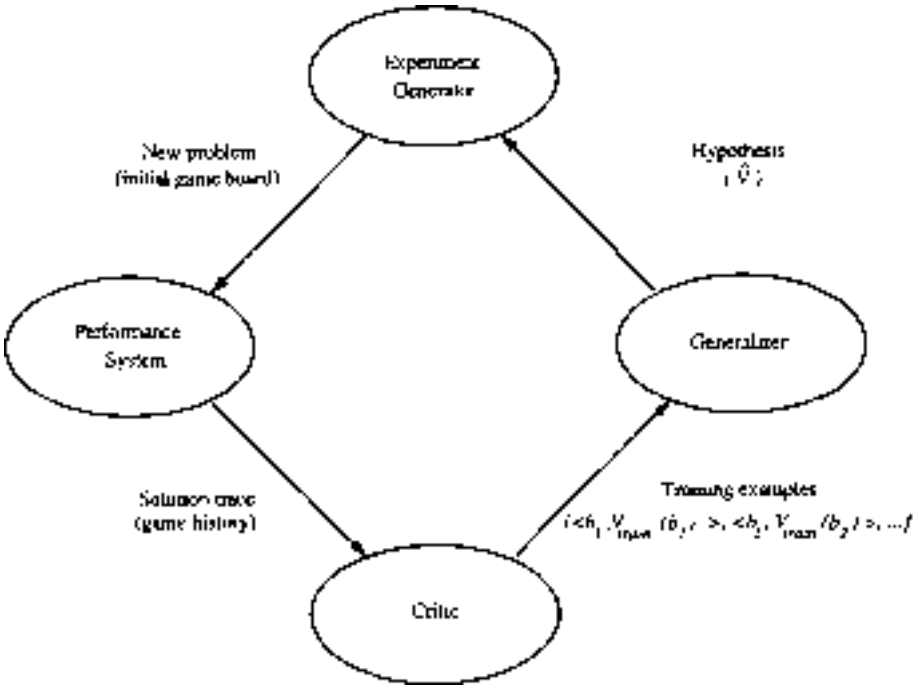
Here η is a small constant (e.g., 0.1) that moderates the size of the weight update.

Working of weight update rule

- When the error $(V_{train}(b) - \hat{V}(b))$ is zero, no weights are changed.
- When $(V_{train}(b) - \hat{V}(b))$ is positive (i.e., when $\hat{V}(b)$ is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of $\hat{V}(b)$, reducing the error.
- If the value of some feature x_i is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board.

5. The Final Design

The final design of checkers learning system can be described by four distinct program modules that represent the central components in many learning systems



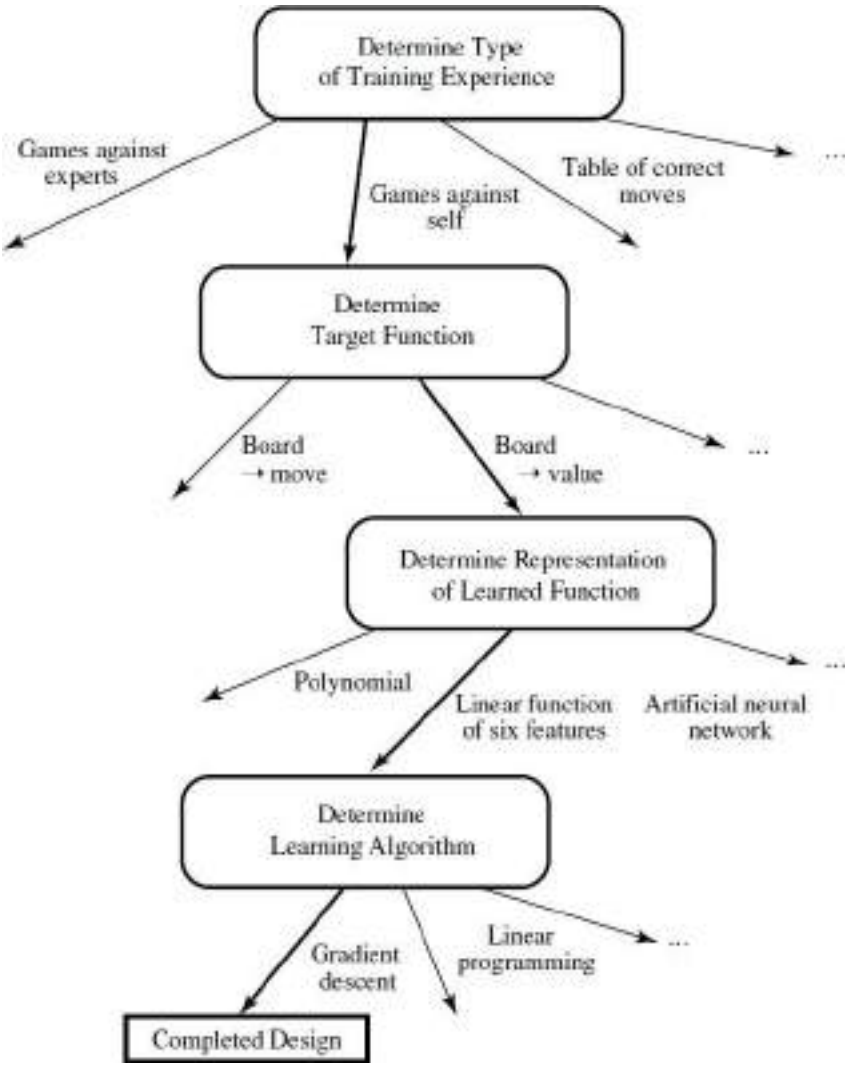
1. **The Performance System** is the module that must solve the given performance task by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.

2. **The Critic** takes as input the history or trace of the game and produces as output a set of training examples of the target function

3. **The Generalizer** takes as input the training examples and produces an output hypothesis that is its estimate of the target function. It generalizes from the specific training examples, hypothesizing a general function that covers these examples and other cases beyond the training examples.

4. **The Experiment Generator** takes as input the current hypothesis and outputs a new problem (i.e., initial board state) for the Performance System to explore. Its role is to pick new practice problems that will maximize the learning rate of the overall system.

The sequence of design choices made for the checkers program is summarized in below figure



PERSPECTIVES AND ISSUES IN MACHINE LEARNING

Issues in Machine Learning

The field of machine learning, and much of this book, is concerned with answering questions such as the following

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?

- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?
- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

CONCEPT LEARNING

- Learning involves acquiring general concepts from specific training examples. Example: People continually learn general concepts or categories such as "bird," "car," "situations in which I should study more in order to pass the exam," etc.
- Each such concept can be viewed as describing some subset of objects or events defined over a larger set
- Alternatively, each concept can be thought of as a Boolean-valued function defined over this larger set. (Example: A function defined over all animals, whose value is true for birds and false for other animals).

Definition: Concept learning **-** Inferring a Boolean-valued function from training examples of its input and output

A CONCEPT LEARNING TASK

Consider the example task of learning the target concept "Days on which ***Aldo*** enjoys his favorite water sport"

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Table: Positive and negative training examples for the target concept ***EnjoySport.***

The task is to learn to predict the value of ***EnjoySport*** for an arbitrary day, based on the values of its other attributes?

What hypothesis representation is provided to the learner?

- Let's consider a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes.
- Let each hypothesis be a vector of six constraints, specifying the values of the six attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*.

For each attribute, the hypothesis will either

- Indicate by a "?' that any value is acceptable for this attribute,
- Specify a single required value (e.g., Warm) for the attribute, or
- Indicate by a "Φ" that no value is acceptable

If some instance ***x*** satisfies all the constraints of hypothesis ***h***, then ***h*** classifies ***x*** as a positive example (***h(x) = 1***).

The hypothesis that ***PERSON*** enjoys his favorite sport only on cold days with high humidity is represented by the expression

(?, Cold, High, ?, ?, ?)

The most general hypothesis-that every day is a positive example-is represented by

(?, ?, ?, ?, ?, ?)

The most specific possible hypothesis-that no day is a positive example-is represented by

(Φ, Φ, Φ, Φ, Φ, Φ)

Notation

- The set of items over which the concept is defined is called the ***set of instances***, which is denoted by X.

Example: X is the set of all possible days, each represented by the attributes: Sky, AirTemp, Humidity, Wind, Water, and Forecast

- The concept or function to be learned is called the ***target concept***, which is denoted by c.

c can be any Boolean valued function defined over the instances X

$$c: X \rightarrow \{O, 1\}$$

Example: The target concept corresponds to the value of the attribute ***EnjoySport*** (i.e., c(x) = 1 if ***EnjoySport*** = Yes, and c(x) = 0 if ***EnjoySport*** = No).

- Instances for which c(x) = 1 are called ***positive examples***, or members of the target concept.
- Instances for which c(x) = 0 are called ***negative examples***, or non-members of the target concept.
- The ordered pair (x, c(x)) to describe the training example consisting of the instance x and its target ***concept value c(x).***
- ***D*** to denote the set of available training examples

- The symbol ***H*** to denote the set of all possible hypotheses that the learner may consider regarding the identity of the target concept. Each hypothesis ***h*** in ***H*** represents a Boolean-valued function defined over ***X***

$$h: X \rightarrow \{O, 1\}$$

The goal of the learner is to find a hypothesis h such that h(x) = c(x) for all x in X.

- Given:
  - Instances X: Possible days, each described by the attributes
    - *Sky* (with possible values Sunny, Cloudy, and Rainy),
    - *AirTemp* (with values Warm and Cold),
    - *Humidity* (with values Normal and High),
    - *Wind* (with values Strong and Weak),
    - *Water* (with values Warm and Cool),
    - *Forecast* (with values Same and Change).

  - Hypotheses *H*: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The constraints may be "?" (any value is acceptable), "Φ" (no value is acceptable), or a specific value.

  - Target concept *c*: ***EnjoySport*** : $X \rightarrow \{0, 1\}$
  - Training examples *D*: Positive and negative examples of the target function

- Determine:
  - A hypothesis h in H such that *h(x) = c(x)* for all *x* in X.

Table: The ***EnjoySport*** concept learning task.

The inductive learning hypothesis

Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

CONCEPT LEARNING AS SEARCH

- Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation.
- The goal of this search is to find the hypothesis that best fits the training examples.

Example:
Consider the instances X and hypotheses H in the *EnjoySport* learning task. The attribute Sky has three possible values, and *AirTemp, Humidity, Wind, Water, Forecast* each have two possible values, the instance space X contains exactly

3.2.2.2.2.2 = 96 distinct instances

5.4.4.4.4.4 = 5120 syntactically distinct hypotheses within H.

Every hypothesis containing one or more "Φ" symbols represents the empty set of instances; that is, it classifies every instance as negative.

1 + (4.3.3.3.3.3) = 973. Semantically distinct hypotheses

General-to-Specific Ordering of Hypotheses

Consider the two hypotheses

h_1 = (Sunny, ?, ?, Strong, ?, ?)

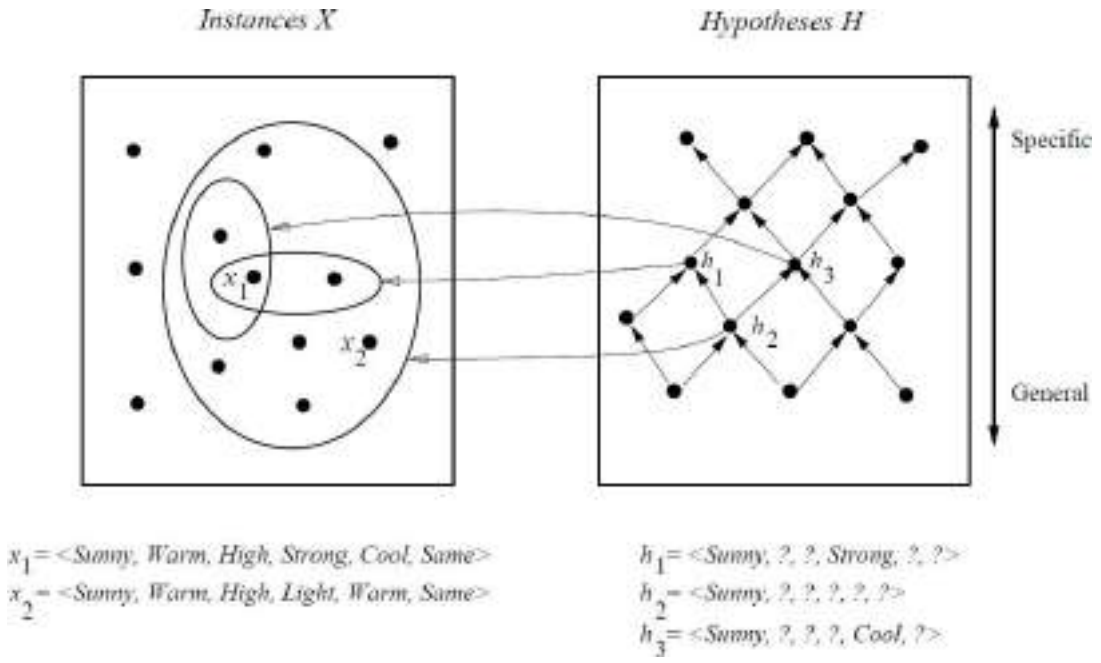
h_2 = (Sunny, ?, ?, ?, ?, ?)

- Consider the sets of instances that are classified positive by h_l and by h_2.
- h_2 imposes fewer constraints on the instance, it classifies more instances as positive. So, any instance classified positive by h_l will also be classified positive by h_2. Therefore, h_2 is more general than h_l.

Given hypotheses h_j and h_k, h_j is more-general-than or- equal do h_k if and only if any instance that satisfies h_k also satisfies h_i

Definition: Let h_j and h_k be Boolean-valued functions defined over X. Then h_j is ***more general-than-or-equal-to*** h_k (written $h_j \geq h_k$) if and only if

$$(\forall\ x \in X\ )\ [(h_k\ (x) = 1) \rightarrow (h_j\ (x) = 1)]$$



- In the figure, the box on the left represents the set X of all instances, the box on the right the set H of all hypotheses.
- Each hypothesis corresponds to some subset of X-the subset of instances that it classifies positive.
- The arrows connecting hypotheses represent the more - general -than relation, with the arrow pointing toward the less general hypothesis.
- Note the subset of instances characterized by h_2 subsumes the subset characterized by h_l , hence h_2 is more - general– than h_1

FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

FIND-S Algorithm

1. Initialize *h* to the most specific hypothesis in *H*

2. For each positive training instance *x*

For each attribute constraint a_i in *h*

If the constraint a_i is satisfied by *x*

Then do nothing

Else replace a_i in *h* by the next more general constraint that is satisfied by *x*

3. Output hypothesis *h*

To illustrate this algorithm, assume the learner is given the sequence of training examples from the ***EnjoySport*** task

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

- The first step of FIND-S is to initialize h to the most specific hypothesis in H

  h - (Ø, Ø, Ø, Ø, Ø, Ø)

- Consider the first training example

  x_1 = <Sunny Warm Normal Strong Warm Same>, +

  Observing the first training example, it is clear that hypothesis *h* is too specific. None of the "Ø" constraints in h are satisfied by this example, so each is replaced by the next *more general constraint* that fits the example

  h_1 = <Sunny Warm Normal Strong Warm Same>

- Consider the second training example

  x_2 = <Sunny, Warm, High, Strong, Warm, Same>, +

  The second training example forces the algorithm to further generalize h, this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example

  h_2 = <Sunny Warm ? Strong Warm Same>

- Consider the third training example

  x3 = <Rainy, Cold, High, Strong, Warm, Change>, -

  Upon encountering the third training the algorithm makes no change to h. The FIND-S algorithm simply ignores every negative example.

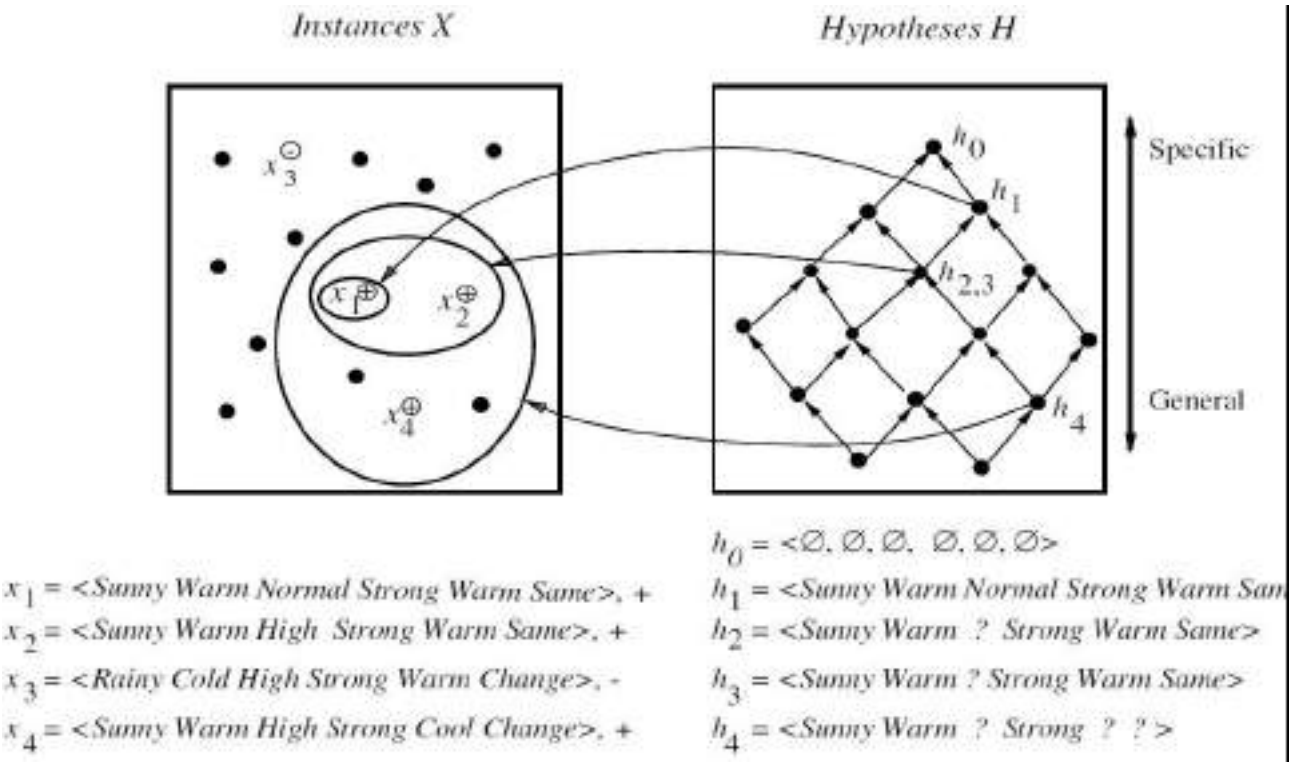
  h_3 = < Sunny Warm ? Strong Warm Same>

- Consider the fourth training example

  x_4 = <Sunny Warm High Strong Cool Change>, +

  The fourth example leads to a further generalization of h

  h_4 = < Sunny Warm ? Strong ? ? >



The key property of the FIND-S algorithm

- FIND-S is guaranteed to output the most specific hypothesis within H that is consistent with the positive training examples
- FIND-S algorithm's final hypothesis will also be consistent with the negative examples provided the correct target concept is contained in H, and provided the training examples are correct.

Unanswered by FIND-S

1. Has the learner converged to the correct target concept?
2. Why prefer the most specific hypothesis?
3. Are the training examples consistent?
4. What if there are several maximally specific consistent hypotheses?

VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

The key idea in the CANDIDATE-ELIMINATION algorithm is to output a description of the set of all ***hypotheses consistent with the training examples***

Representation

Definition: consistent- A hypothesis h is **consistent** with **a** set of training examples ***D*** if **and** only if ***h(x)*** = ***c(x)*** for each example ***(x, c(x))*** in ***D.***

$$Consistent\ (h, D) \equiv (\forall\ \langle x, c(x)\rangle \in D)\ h(x) = c(x))$$

Note difference between definitions of *consistent* and *satisfies*

- An example *x* is said to ***satisfy*** hypothesis ***h*** when ***h(x)*** = 1, regardless of whether *x* is a positive or negative example of the target concept.
- An example *x* is said to ***consistent*** with hypothesis ***h*** iff ***h(x)*** = ***c(x)***

Definition: version space- The **version space,** denoted $VS_{H, D}$ with respect to hypothesis space ***H*** and training examples D, is the subset of hypotheses from ***H*** consistent with the training examples in D

$$VS_{H, D} \equiv \{h \in H \mid Consistent\ (h, D)\}$$

The LIST-THEN-ELIMINATION algorithm

The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H and then eliminates any hypothesis found inconsistent with any training example.

1. ***VersionSpace c*** a list containing every hypothesis in H
2. For each training example, (x, c(x))
   remove from ***VersionSpace*** any hypothesis h for which h(x) ≠ c(x)
3. Output the list of hypotheses in ***VersionSpace***

The LIST-THEN-ELIMINATE Algorithm

- List-Then-Eliminate works in principle, so long as version space is finite.
- However, since it requires exhaustive enumeration of all hypotheses in practice it is not feasible.

A More Compact Representation for Version Spaces

The version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

Definition: The **general boundary** G, with respect to hypothesis space ***H*** and training data *D*, is the set of maximally general members of *H* consistent with *D*

$$G \equiv \{g \in H \mid Consistent\ (g, D) \wedge (\neg \exists g' \in H)[(g' >_g g) \wedge Consistent(g', D)]\}$$

Definition: The **specific boundary** S, with respect to hypothesis space *H* and training data *D*, is the set of minimally general (i.e., maximally specific) members of *H* consistent with *D*.

$$S \equiv \{s \in H \mid Consistent\ (s, D) \wedge (\neg \exists s' \in H)[(s >_g s') \wedge Consistent(s', D)]\}$$

Theorem: Version Space representation theorem

Theorem: Let X be an arbitrary set of instances and Let H be a set of Boolean-valued hypotheses defined over X. Let c: X $\rightarrow$ {O, 1} be an arbitrary target concept defined over X, and let D be an arbitrary set of training examples {(x, c(x))). For all X, H, c, and D such that S and G are well defined,

$$VS_{H,D} = \{\ h \in H \mid (\exists s \in S\ )\ (\exists g \in G\ )\ (\ g \geq_g h \geq_g s\ )\}$$

To Prove:

1. Every h satisfying the right hand side of the above expression is in $\boldsymbol{VS_{H, D}}$

2. Every member of $\boldsymbol{VS_{H, D}}$ satisfies the right-hand side of the expression

Sketch of proof:

1. let g, h, s be arbitrary members of G, H, S respectively with $g \geq_g h \geq_g s$
   - By the definition of ***S,*** **s** must be satisfied by all positive examples in D. Because $h \geq_g s$, h must also be satisfied by all positive examples in D.
   - By the definition of ***G,*** g cannot be satisfied by any negative example in D, and because $g \geq_g h$ h cannot be satisfied by any negative example in D. Because h is satisfied by all positive examples in D and by no negative examples in D, h is consistent with D, and therefore h is a member of $VS_{H,D}$.
2. It can be proven by assuming some h in $VS_{H,D}$,that does not satisfy the right-hand side of the expression, then showing that this leads to an inconsistency

CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINTION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.

Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

- If d is a positive example
  - Remove from G any hypothesis inconsistent with d
  - For each hypothesis s in S that is not consistent with d
    - Remove s from S
    - Add to S all minimal generalizations h of s such that
      - h is consistent with d, and some member of G is more general than h
    - Remove from S any hypothesis that is more general than another hypothesis in S

- If d is a negative example
  - Remove from S any hypothesis inconsistent with d
  - For each hypothesis g in G that is not consistent with d
    - Remove g from G
    - Add to G all minimal specializations h of g such that
      - h is consistent with d, and some member of S is more specific than h
    - Remove from G any hypothesis that is less general than another hypothesis in G

CANDIDATE- ELIMINTION algorithm using version spaces

An Illustrative Example

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

CANDIDATE-ELIMINTION algorithm begins by initializing the version space to the set of all hypotheses in H;

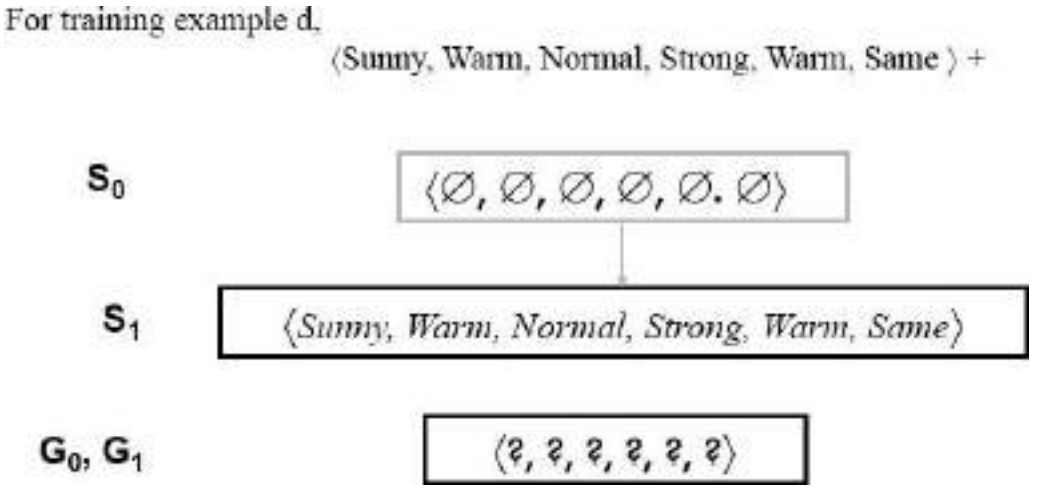
Initializing the G boundary set to contain the most general hypothesis in H

$G_0 \langle ?, ?, ?, ?, ?, ? \rangle$

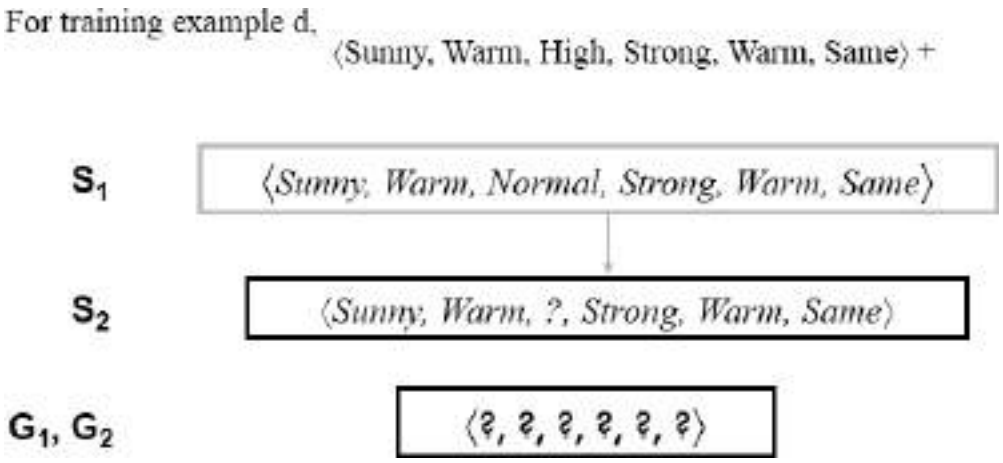
Initializing the S boundary set to contain the most specific (least general) hypothesis

$S_0 \langle \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$

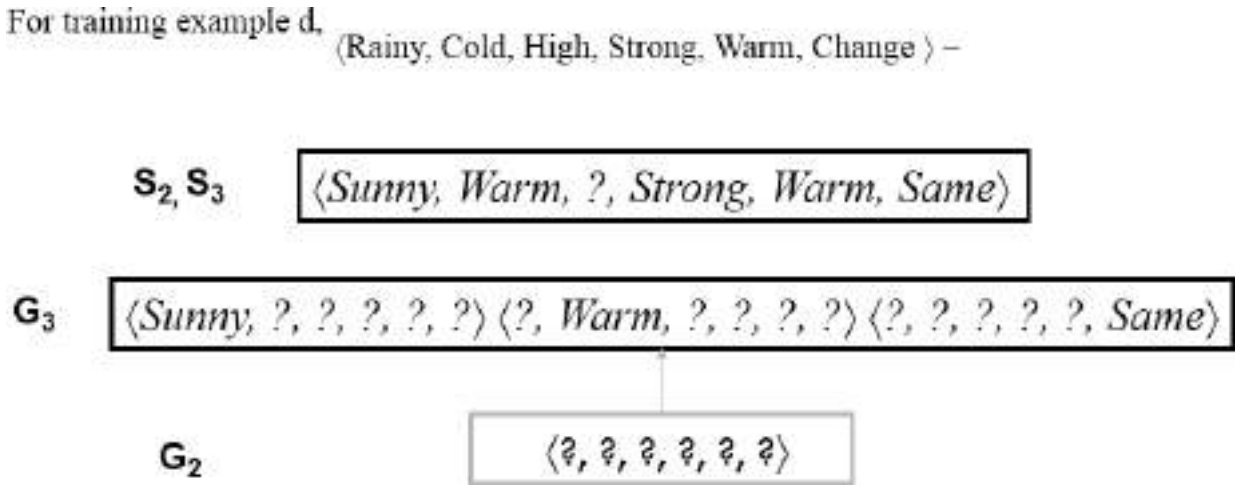
- When the first training example is presented, the CANDIDATE-ELIMINTION algorithm checks the S boundary and finds that it is overly specific and it fails to cover the positive example.
- The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example
- No update of the G boundary is needed in response to this training example because G_o correctly covers this example



- When the second training example is observed, it has a similar effect of generalizing S further to S_2, leaving G again unchanged i.e., $G_2 = G_1 = G_0$



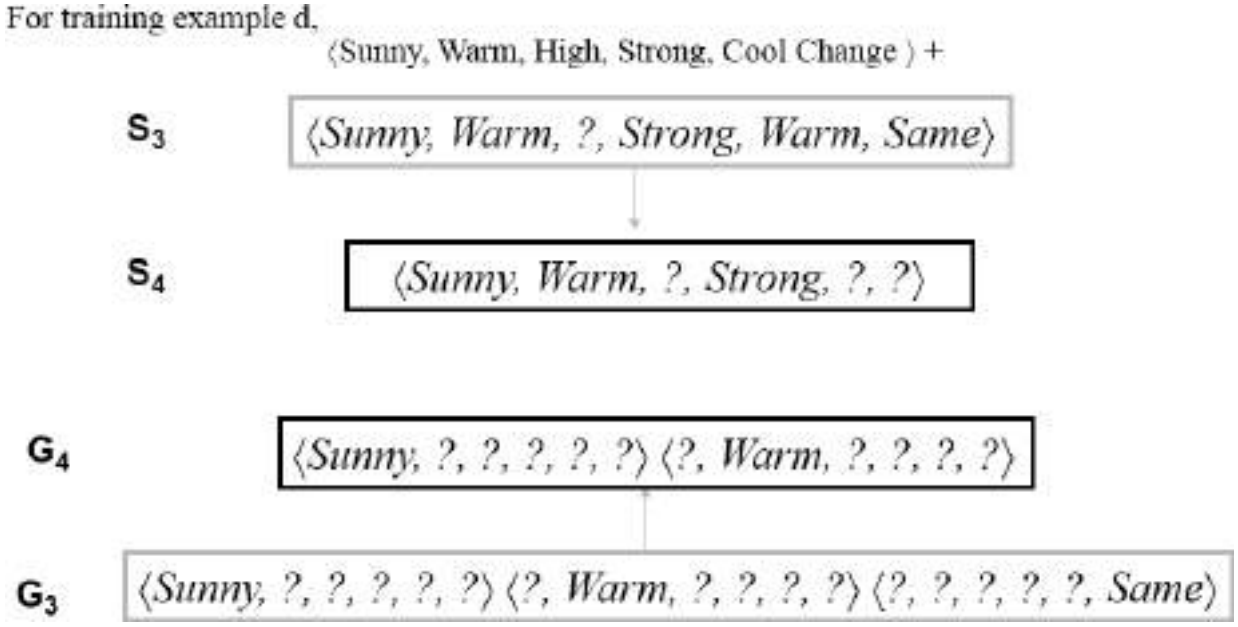
- Consider the third training example. This negative example reveals that the G boundary of the version space is overly general, that is, the hypothesis in G incorrectly predicts that this new example is a positive example.
- The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example



Given that there are six attributes that could be specified to specialize G_2, why are there only three new hypotheses in G_3?

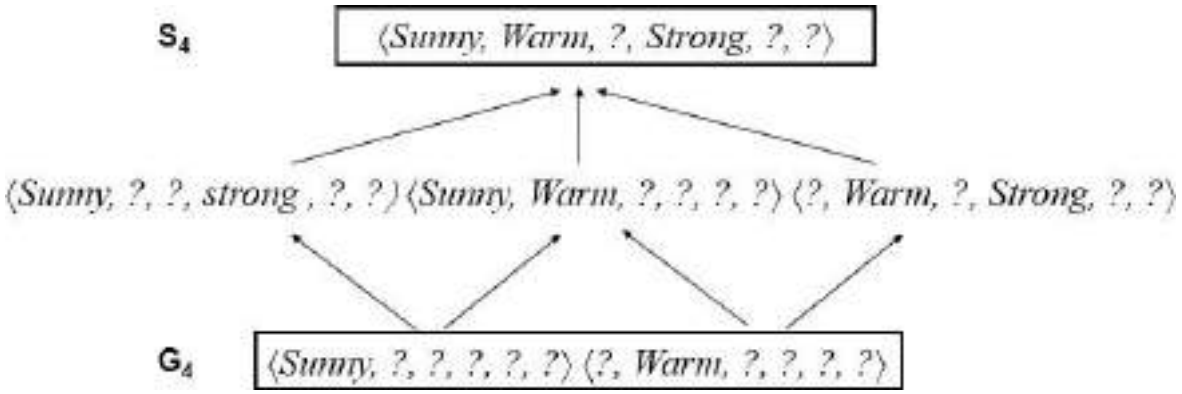
For example, the hypothesis h = (?, ?, Normal, ?, ?, ?) is a minimal specialization of G_2 that correctly labels the new example as a negative example, but it is not included in G_3. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples

- Consider the fourth training example.



- This positive example further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example

After processing these four examples, the boundary sets S_4 and G_4 delimit the version space of all hypotheses consistent with the set of incrementally observed training examples.



INDUCTIVE BIAS

The fundamental questions for inductive inference

1. What if the target concept is not contained in the hypothesis space?
2. Can we avoid this difficulty by using a hypothesis space that includes every possible hypothesis?
3. How does the size of this hypothesis space influence the ability of the algorithm to generalize to unobserved instances?
4. How does the size of the hypothesis space influence the number of training examples that must be observed?

These fundamental questions are examined in the context of the CANDIDATE-ELIMINTION algorithm

A Biased Hypothesis Space

- Suppose the target concept is not contained in the hypothesis space *H*, then obvious solution is to enrich the hypothesis space to include every possible hypothesis.
  - Consider the ***EnjoySport*** example in which the hypothesis space is restricted to include only conjunctions of attribute values. Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as

    "Sky = Sunny or Sky = Cloudy."
  - The following three training examples of disjunctive hypothesis, the algorithm would find that there are zero hypotheses in the version space

⟨Sunny Warm Normal Strong Cool Change⟩	Y
⟨Cloudy Warm Normal Strong Cool Change⟩	Y
⟨Rainy Warm Normal Strong Cool Change⟩	N

  - If Candidate Elimination algorithm is applied, then it end up with empty Version Space. After first two training example

    S= ⟨? Warm Normal Strong Cool Change⟩

  - This new hypothesis is overly general and it incorrectly covers the third negative training example! So H does not include the appropriate c.
  - In this case, a more expressive hypothesis space is required.

An Unbiased Learner

- The solution to the problem of assuring that the target concept is in the hypothesis space H is to provide a hypothesis space capable of representing every teachable concept that is representing every possible subset of the instances X.
- The set of all subsets of a set X is called the power set of X

  - In the ***EnjoySport*** learning task the size of the instance space X of days described by the six attributes is 96 instances.
  - Thus, there are 2^{96} distinct target concepts that could be defined over this instance space and learner might be called upon to learn.
  - The conjunctive hypothesis space is able to represent only 973 of these - a biased hypothesis space indeed

  - Let us reformulate the ***EnjoySport*** learning task in an unbiased way by defining a new hypothesis space H' that can represent every subset of instances
  - The target concept "Sky = Sunny or Sky = Cloudy" could then be described as

(Sunny, ?, ?, ?, ?, ?) v (Cloudy, ?, ?, ?, ?, ?)

The Futility of Bias-Free Learning

Inductive learning requires some form of prior assumptions, or inductive bias

Definition:

Consider a concept learning algorithm L for the set of instances X.

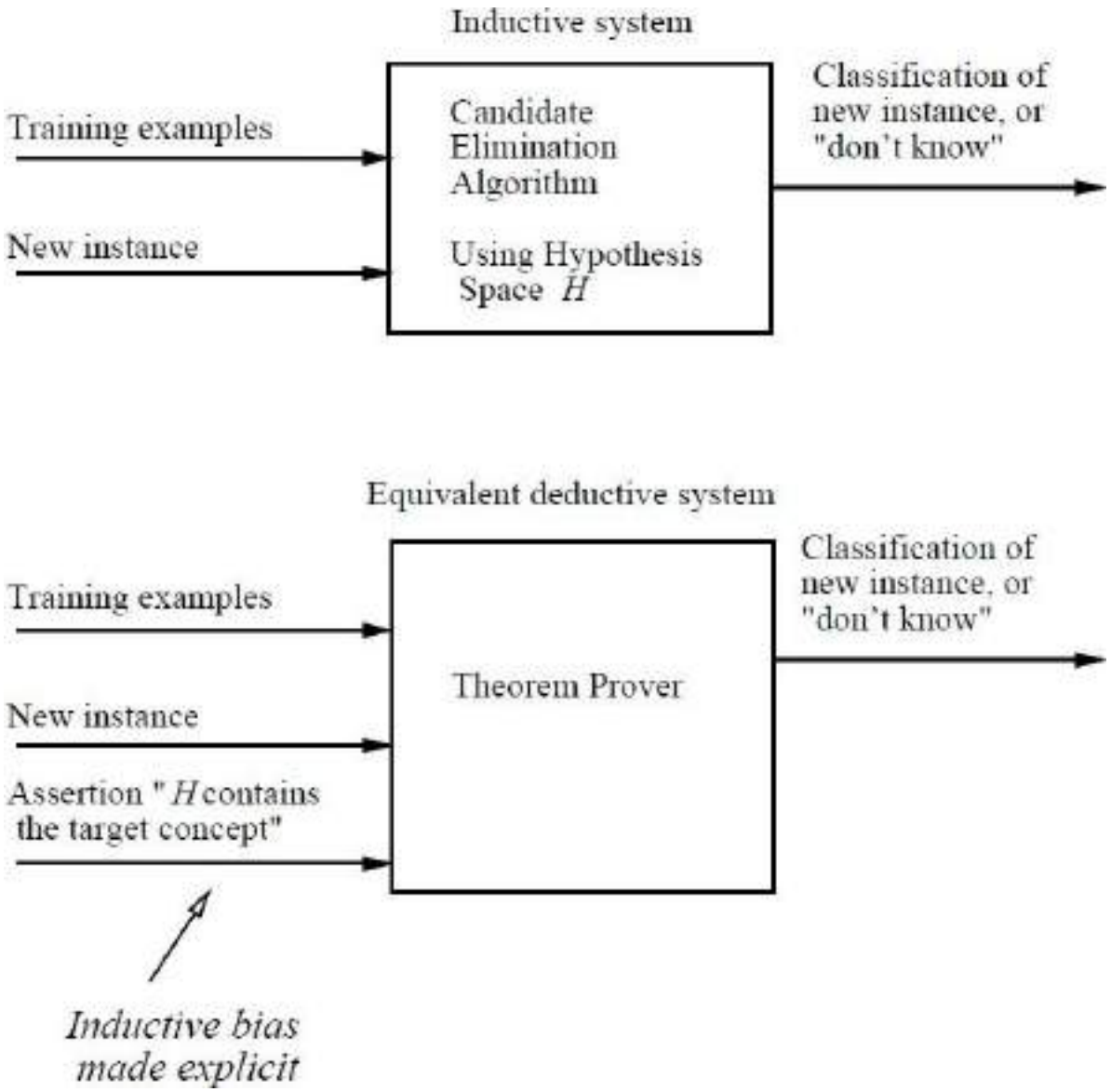
- Let c be an arbitrary concept defined over X
- Let $D_c = \{(x , c(x))\}$ be an arbitrary set of training examples of c.

- Let $L (x_i , D_c)$ denote the classification assigned to the instance x_i by L after training on the data D_c.
- The inductive bias of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c

- $(\forall \langle x_i \in X) [(B \wedge D_c \wedge x_i) \vdash L (x_i, D_c)]$

The below figure explains

- Modelling inductive systems by equivalent deductive systems.
- The input-output behavior of the CANDIDATE-ELIMINATION algorithm using a hypothesis space H is identical to that of a deductive theorem prover utilizing the assertion "H contains the target concept." This assertion is therefore called the inductive bias of the CANDIDATE-ELIMINATION algorithm.
- Characterizing inductive systems by their inductive bias allows modelling them by their equivalent deductive systems. This provides a way to compare inductive systems according to their policies for generalizing beyond the observed training data.



UNIT 2

DECISION TREE LEARNING

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree.

DECISION TREE REPRESENTATION

- Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance.
- Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute.
- An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

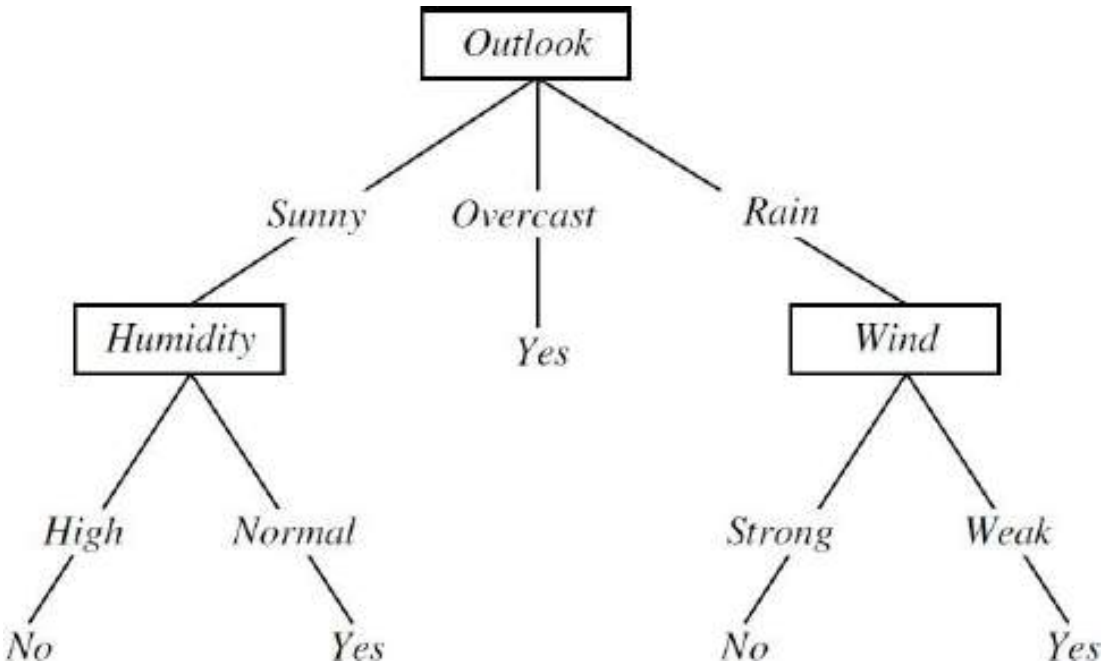


FIGURE: A decision tree for the concept ***PlayTennis***. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf

- Decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances.
- Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions

For example, the decision tree shown in above figure corresponds to the expression

(Outlook = Sunny ∧ Humidity = Normal)

∨ (Outlook = Overcast)

∨ (Outlook = Rain ∧ Wind = Weak)

APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Decision tree learning is generally best suited to problems with the following characteristics:

1. ***Instances are represented by attribute-value pairs*** – Instances are described by a fixed set of attributes and their values

2. ***The target function has discrete output values*** – The decision tree assigns a Boolean classification (e.g., yes or no) to each example. Decision tree methods easily extend to learning functions with more than two possible output values.

3. ***Disjunctive descriptions may be required***

4. ***The training data may contain errors*** – Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.

5. ***The training data may contain missing attribute values –*** Decision tree methods can be used even when some training examples have unknown values

THE BASIC DECISION TREE LEARNING ALGORITHM

The basic algorithm is ID3 which learns decision trees by constructing them top-down

---

ID3(Examples, Target_attribute, Attributes)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a Root node for the tree
- If all Examples are positive, Return the single-node tree Root, with label = +
- If all Examples are negative, Return the single-node tree Root, with label = -
- If Attributes is empty, Return the single-node tree Root, with label = most common value of Target_attribute in Examples

- Otherwise Begin
  - A ← the attribute from Attributes that best* classifies Examples
  - The decision attribute for Root ← A
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test A = v_i
    - Let *Examples* $_{vi}$, be the subset of Examples that have value v_i for *A*
    - If *Examples* $_{vi}$, is empty
      - Then below this new branch add a leaf node with label = most common value of Target_attribute in Examples
      - Else below this new branch add the subtree
        ID3(*Examples* $_{vi}$, Targe_tattribute, Attributes – {A}))
- End
- Return Root

---

* The best attribute is the one with highest information gain

TABLE: Summary of the ID3 algorithm specialized to learning Boolean-valued functions. ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used

Which Attribute Is the Best Classifier?

- The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree.
- A statistical property called ***information gain*** that measures how well a given attribute separates the training examples according to their target classification.
- ID3 uses ***information gain*** measure to select among the candidate attributes at each step while growing the tree.

ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

To define information gain, we begin by defining a measure called entropy. *Entropy measures the impurity of a collection of examples.*

Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is

$$Entropy\ (S) \equiv -p_{\oplus} log_2 p_{\oplus} - p_{\ominus} log_2 p_{\ominus}$$

Where,

p_+ is the proportion of positive examples in S
p_- is the proportion of negative examples in S.

Example:
Suppose S is a collection of 14 examples of some boolean concept, including 9 positive and 5 negative examples. Then the entropy of S relative to this boolean classification is

$$Entropy([9+, 5-]) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14)$$
$$= 0.940$$

- The entropy is 0 if all members of S belong to the same class
- The entropy is 1 when the collection contains an equal number of positive and negative examples
- If the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1

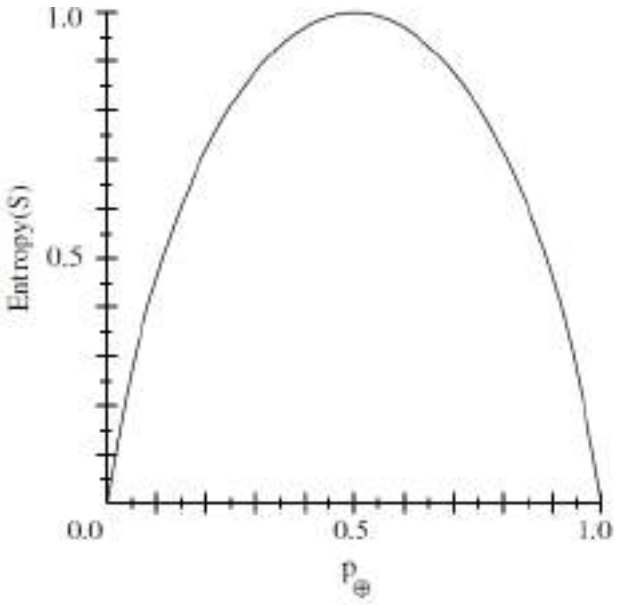


FIGURE The entropy function relative to a boolean classification, as the proportion, $p_{\oplus}$, of positive examples varies between 0 and 1.

INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

- ***Information gain,*** is the expected reduction in entropy caused by partitioning the examples according to this attribute.
- The information gain, Gain(S, A) of an attribute A, relative to a collection of examples S, is defined as

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

Example: Information gain

Let, *Values*(*Wind*) = {*Weak, Strong*}

S = [9+, 5−]

S_{Weak} = [6+, 2−]

S_{Strong} = [3+, 3−]

Information gain of attribute *Wind*:

Gain(*S, Wind*) = *Entropy*(*S*) − 8/14 *Entropy* (S_{Weak}) − 6/14 *Entropy* (S_{Strong})

= 0.94 – (8/14)* 0.811 – (6/14) *1.00

= 0.048

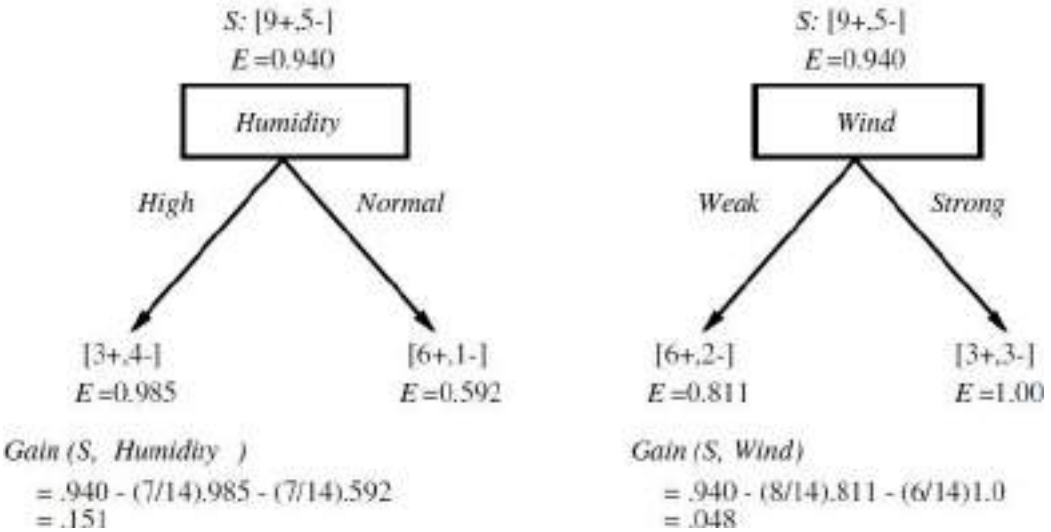
An Illustrative Example

- To illustrate the operation of ID3, consider the learning task represented by the training examples of below table.
- Here the target attribute ***PlayTennis***, which can have values ***yes*** or ***no*** for different days.
- Consider the first step through the algorithm, in which the topmost node of the decision tree is created.

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

- ID3 determines the information gain for each candidate attribute (i.e., Outlook, Temperature, Humidity, and Wind), then selects the one with highest information gain.

Which attribute is the best classifier?



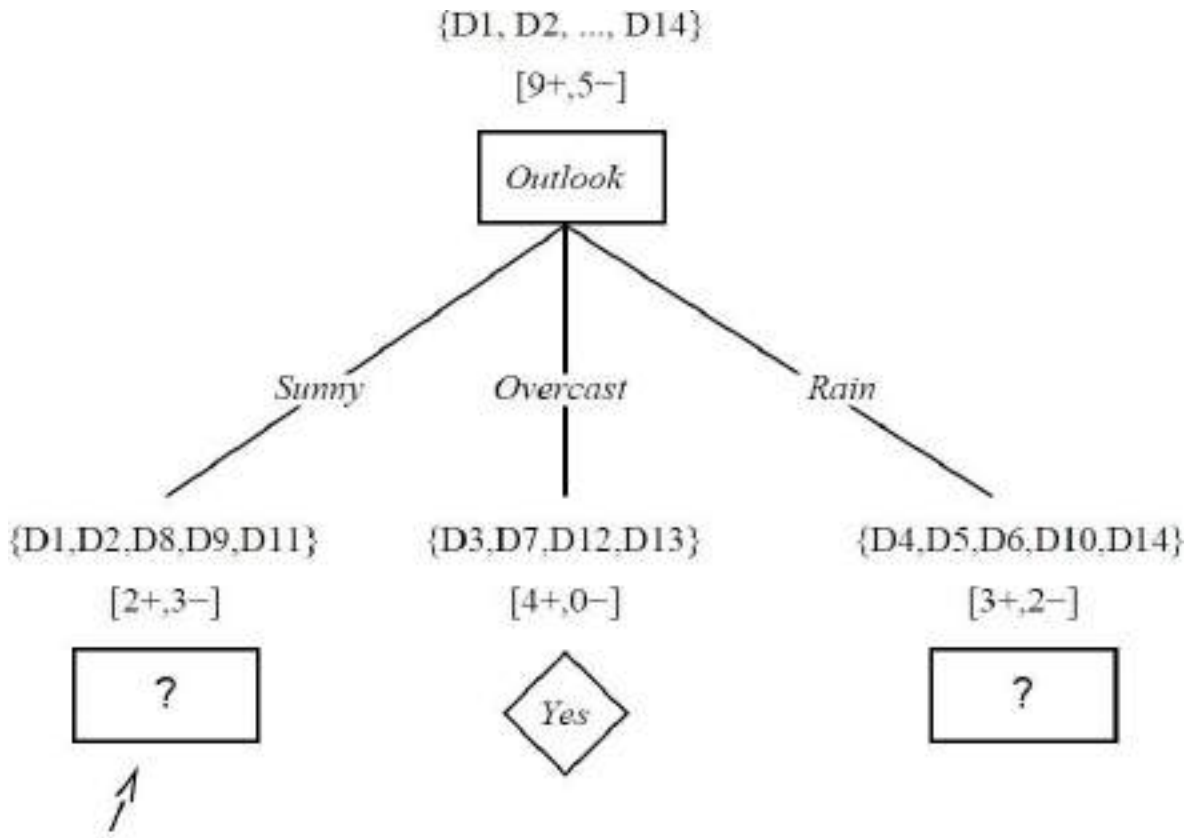
Gain (S, Humidity)
= .940 - (7/14).985 - (7/14).592
= .151

Gain (S, Wind)
= .940 - (8/14).811 - (6/14)1.0
= .048

- The information gain values for all four attributes are

Gain(S, Outlook)	= 0.246
Gain(S, Humidity)	= 0.151
Gain(S, Wind)	= 0.048
Gain(S, Temperature)	= 0.029

- According to the information gain measure, the ***Outlook*** attribute provides the best prediction of the target attribute, ***PlayTennis***, over the training examples. Therefore, ***Outlook*** is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values i.e., Sunny, Overcast, and Rain.



S_{sunny} = {D1,D2,D8,D9,D11}

$Gain\ (S_{sunny}, Humidity) = .970 - (3/5)\ 0.0 - (2/5)\ 0.0 = .970$

$Gain\ (S_{sunny}, Temperature) = .970 - (2/5)\ 0.0 - (2/5)\ 1.0 - (1/5)\ 0.0 = .570$

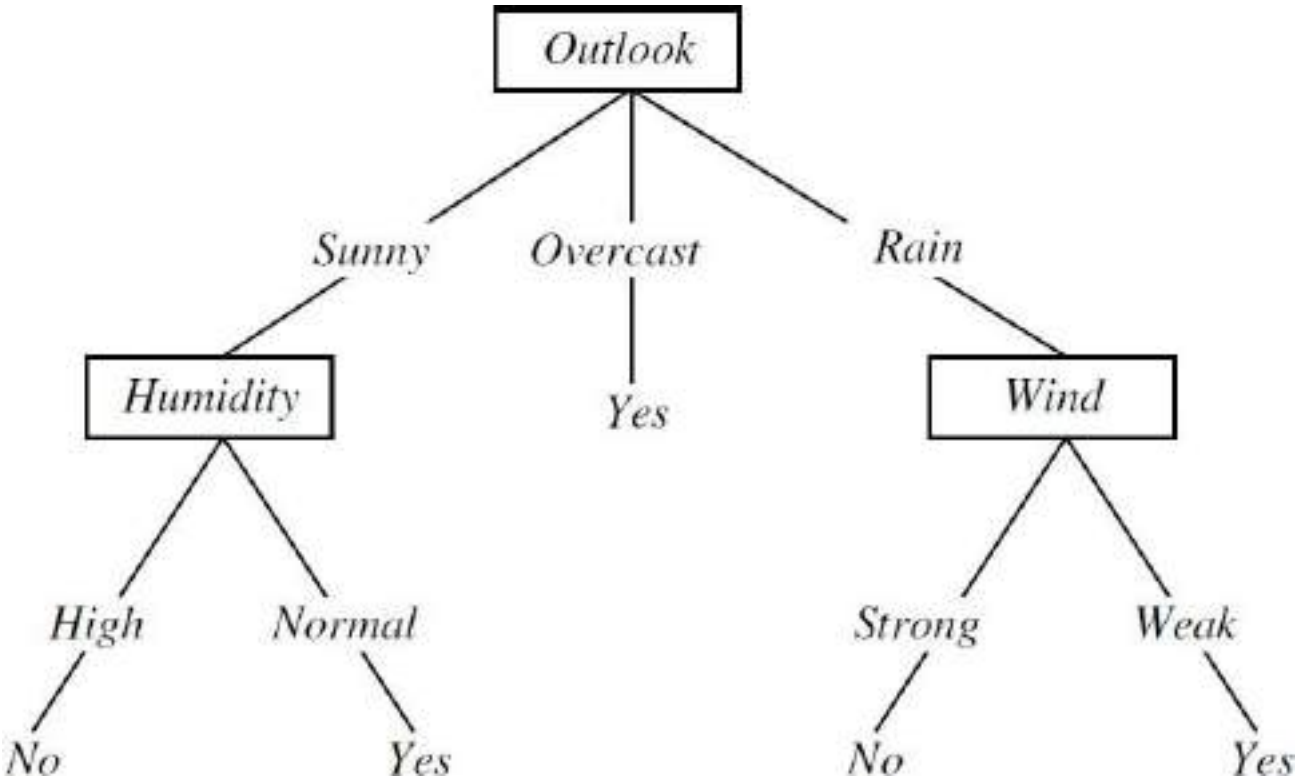
$Gain\ (S_{sunny}, Wind) = .970 - (2/5)\ 1.0 - (3/5)\ .918 = .019$

S_{Rain} = { D4, D5, D6, D10, D14}

Gain (S_{Rain} , *Humidity)* = 0.970 – (2/5)1.0 – (3/5)0.917 = 0.019

Gain (S_{Rain} , *Temperature)* =0.970 – (0/5)0.0 – (3/5)0.918 – (2/5)1.0 = 0.019

Gain (S_{Rain} , *Wind)* =0.970 – (3/5)0.0 – (2/5)0.0 = 0.970



HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

- ID3 can be characterized as searching a space of hypotheses for one that fits the training examples.
- The hypothesis space searched by ID3 is the set of possible decision trees.
- ID3 performs a simple-to complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data

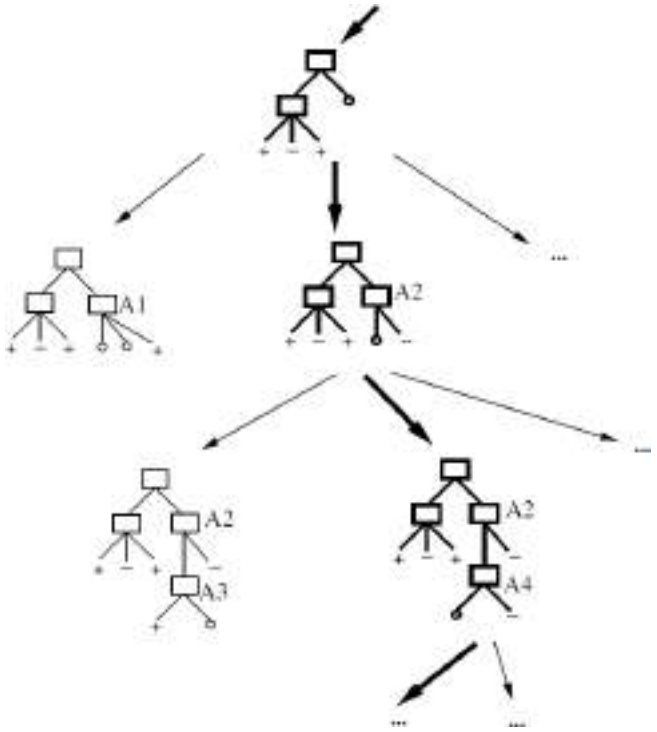


Figure: Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

By viewing ID3 in terms of its search space and search strategy, there are some insight into its capabilities and limitations

1. *ID3's hypothesis space of all decision trees is a complete space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree*
   ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces: that the hypothesis space might not contain the target function.

2. *ID3 maintains only a single current hypothesis as it searches through the space of decision trees.*
   For example, with the earlier version space candidate elimination method, which maintains the set of all hypotheses consistent with the available training examples.

   By determining only a single hypothesis, ID3 loses the capabilities that follow from explicitly representing all consistent hypotheses.
   For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses

3. *ID3 in its pure form performs no backtracking in its search. Once it selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice.*
   In the case of ID3, a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search.

4. *ID3 uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis.*
   One advantage of using statistical properties of all the examples is that the resulting search is much less sensitive to errors in individual training examples.
   ID3 can be easily extended to handle noisy training data by modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

INDUCTIVE BIAS IN DECISION TREE LEARNING

Inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances

Given a collection of training examples, there are typically many decision trees consistent with these examples. Which of these decision trees does ID3 choose?

ID3 search strategy

- Selects in favour of shorter trees over longer ones
- Selects trees that place the attributes with highest information gain closest to the root.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees

- Consider an algorithm that begins with the empty tree and searches breadth first through progressively more complex trees.
- First considering all trees of depth 1, then all trees of depth 2, etc.
- Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes).
- Let us call this breadth-first search algorithm BFS-ID3.
- BFS-ID3 finds a shortest decision tree and thus exhibits the bias "shorter trees are preferred over longer trees.

A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

- ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.
- Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3.
- In particular, it does not always find the shortest consistent tree, and it is biased to favour trees that place attributes with high information gain closest to the root.

Restriction Biases and Preference Biases

Difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION Algorithm.

ID3:

- ID3 searches a complete hypothesis space
- It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met
- Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias

CANDIDATE-ELIMINATION Algorithm:

- The version space CANDIDATE-ELIMINATION Algorithm searches an incomplete hypothesis space
- It searches this space completely, finding every hypothesis consistent with the training data.
- Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias

Preference bias **–** The inductive bias of ID3 is a preference for certain hypotheses over others (e.g., preference for shorter hypotheses over larger hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is called a preference bias or a search bias.

Restriction bias – The bias of the CANDIDATE ELIMINATION algorithm is in the form of a categorical restriction on the set of hypotheses considered. This form of bias is typically called a restriction bias or a language bias.

Which type of inductive bias is preferred in order to generalize beyond the training data, a preference bias or restriction bias?

- A preference bias is more desirable than a restriction bias, because it allows the learner to work within a complete hypothesis space that is assured to contain the unknown target function.
- In contrast, a restriction bias that strictly limits the set of potential hypotheses is generally less desirable, because it introduces the possibility of excluding the unknown target function altogether.

Why Prefer Short Hypotheses?

Occam's razor

- Occam's razor: is the problem-solving principle that the simplest solution tends to be the right one. When presented with competing hypotheses to solve a problem, one should select the solution with the fewest assumptions.

- Occam's razor: “Prefer the simplest hypothesis that fits the data”.

Argument in favour of Occam’s razor:

- Fewer short hypotheses than long ones:
  - Short hypotheses fits the training data which are less likely to be coincident
  - Longer hypotheses fits the training data might be coincident.

- Many complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data.

Argument opposed:

- There are few small trees, and our priori chance of finding one consistent with an arbitrary set of data is therefore small. The difficulty here is that there are very many small sets of hypotheses that one can define but understood by fewer learner.
- The size of a hypothesis is determined by the representation used internally by the learner. Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners, both justifying their contradictory conclusions by Occam's razor. On this basis we might be tempted to reject Occam's razor altogether.

ISSUES IN DECISION TREE LEARNING

Issues in learning decision trees include

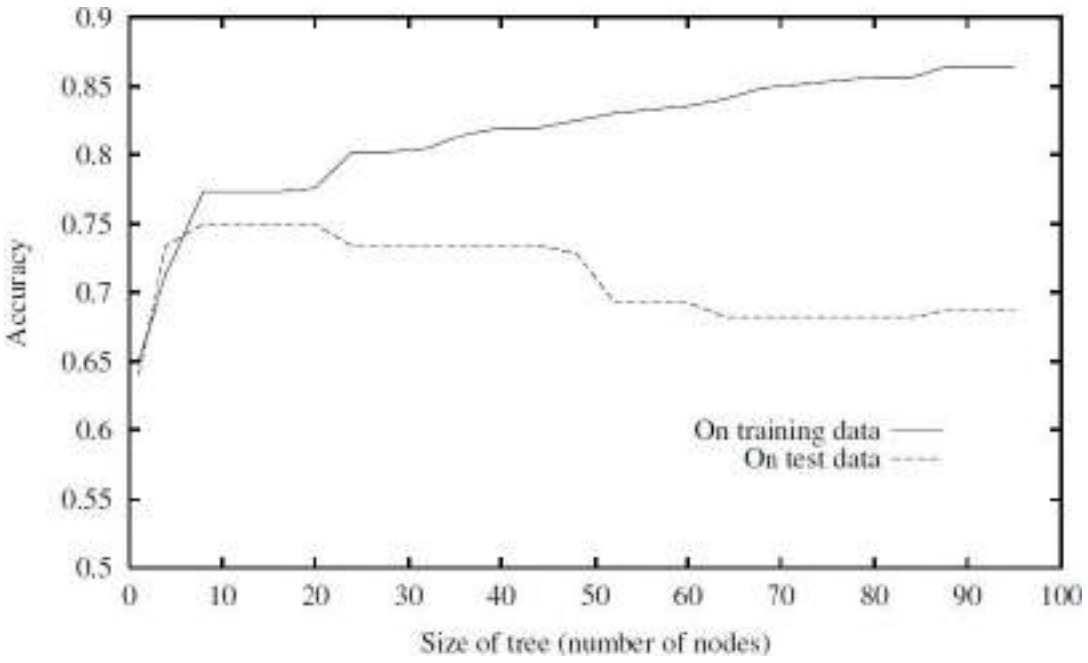
1. Avoiding Overfitting the Data
   Reduced error pruning
   Rule post-pruning
2. Incorporating Continuous-Valued Attributes
3. Alternative Measures for Selecting Attributes
4. Handling Training Examples with Missing Attribute Values
5. Handling Attributes with Differing Costs

1. Avoiding Overfitting the Data

- The ID3 algorithm grows each branch of the tree just deeply enough to perfectly classify the training examples but it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function. This algorithm can produce trees that overfit the training examples.

- ***Definition - Overfit:*** Given a hypothesis space H, a hypothesis h ∈ H is said to overfit the training data if there exists some alternative hypothesis h' ∈ H, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

The below figure illustrates the impact of overfitting in a typical application of decision tree learning.



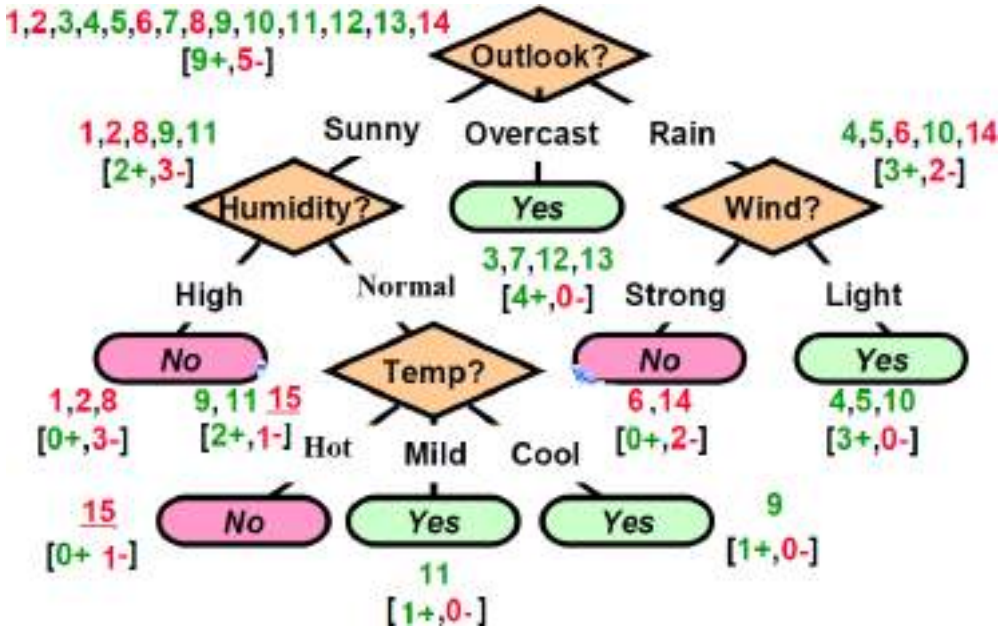
- *The horizontal axis* of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree.
- *The solid line* shows the accuracy of the decision tree over the training examples. The broken line shows accuracy measured over an independent set of test example
- The accuracy of the tree over the training examples increases monotonically as the tree is grown. The accuracy measured over the independent test examples first increases, then decreases.

How can it be possible for tree h to fit the training examples better than h', but for it to perform more poorly over subsequent examples?

1. Overfitting can occur when the training examples contain random errors or noise
2. When small numbers of examples are associated with leaf nodes.

Noisy Training Example

- Example 15: <Sunny, Hot, Normal, Strong, ->
- Example is noisy because the correct label is +
- Previously constructed tree misclassifies it



Approaches to avoiding overfitting in decision tree learning

- Pre-pruning (avoidance): Stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data
- Post-pruning (recovery): Allow the tree to overfit the data, and then post-prune the tree

Criterion used to determine the correct final tree size

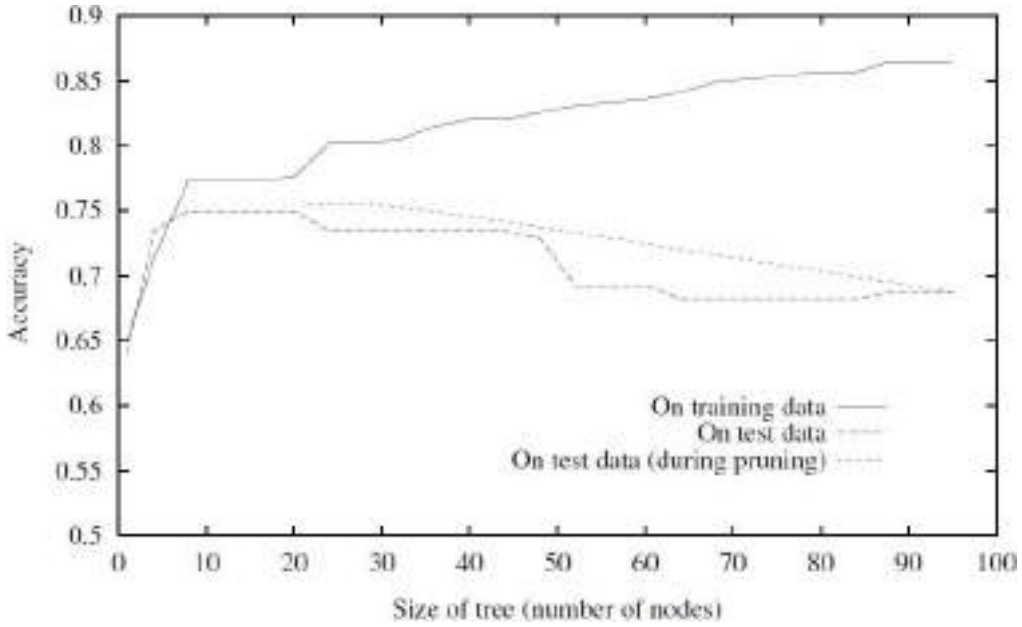
- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond the training set
- Use measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. This approach is called the Minimum Description Length

MDL – Minimize : size(tree) + size (misclassifications(tree))

Reduced-Error Pruning

- Reduced-error pruning, is to consider each of the decision nodes in the tree to be candidates for pruning
- ***Pruning*** a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node
- Nodes are removed only if the resulting pruned tree performs no worse than-the original over the validation set.
- Reduced error pruning has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in below figure



- The additional line in figure shows accuracy over the test examples as the tree is pruned. When pruning begins, the tree is at its maximum size and lowest accuracy over the test set. As pruning proceeds, the number of nodes is reduced and accuracy over the test set increases.
- The available data has been split into three subsets: the training examples, the validation examples used for pruning the tree, and a set of test examples used to provide an unbiased estimate of accuracy over future unseen examples. The plot shows accuracy over the training and test sets.

Pros and Cons

Pro: Produces smallest version of most accurate T (subtree of T)

Con: Uses less data to construct T

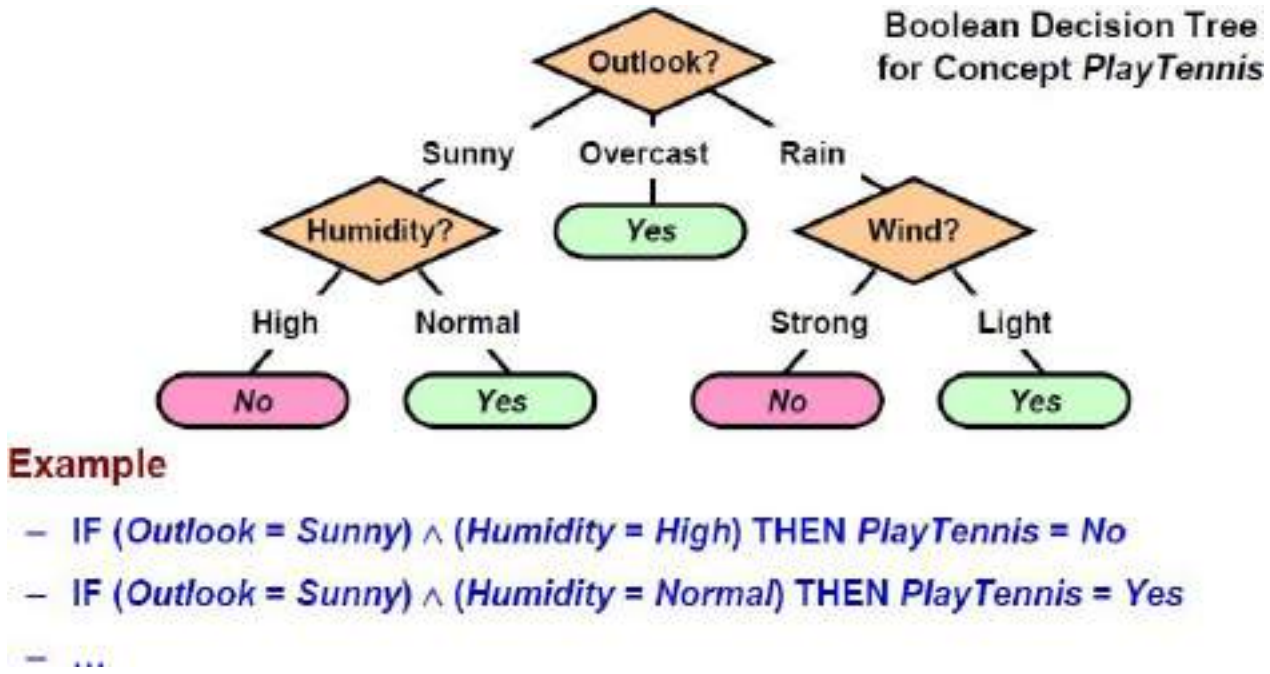
Can afford to hold out $D_{validation}$?. If not (data is too limited), may make error worse (insufficient D_{train})

Rule Post-Pruning

Rule post-pruning is successful method for finding high accuracy hypotheses

- Rule post-pruning involves the following steps:
- Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.
- Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.
- Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
- Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

Converting a Decision Tree into Rules



For example, consider the decision tree. The leftmost path of the tree in below figure is translated into the rule.

IF (Outlook = Sunny) ^ (Humidity = High)
THEN PlayTennis = No

Given the above rule, rule post-pruning would consider removing the preconditions

(Outlook = Sunny) and (Humidity = High)

- It would select whichever of these pruning steps produced the greatest improvement in estimated rule accuracy, then consider pruning the second precondition as a further pruning step.
- No pruning step is performed if it reduces the estimated rule accuracy.

There are three main advantages by converting the decision tree to rules before pruning

1. Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path.
2. Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, it avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the subtree below this test.
3. Converting to rules improves readability. Rules are often easier for to understand.

2. Incorporating Continuous-Valued Attributes

Continuous-valued decision attributes can be incorporated into the learned tree.

There are two methods for Handling Continuous Attributes

1. Define new discrete valued attributes that partition the continuous attribute value into a discrete set of intervals.

E.g., {high ≡ Temp > 35º C, med ≡ 10º C < Temp ≤ 35º C, low ≡ Temp ≤ 10º C}

2. Using thresholds for splitting nodes

e.g., A ≤ a produces subsets A ≤ a and A > a

What threshold-based Boolean attribute should be defined based on Temperature?

Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

- Pick a threshold, c, that produces the greatest information gain
- In the current example, there are two candidate thresholds, corresponding to the values of Temperature at which the value of PlayTennis changes: (48 + 60)/2, and (80 + 90)/2.
- The information gain can then be computed for each of the candidate attributes, Temperature $_{>54}$, and Temperature $_{>85}$ and the best can be selected (Temperature $_{>54}$)

3. Alternative Measures for Selecting Attributes

- The problem is if attributes with many values, Gain will select it ?
- Example: consider the attribute Date, which has a very large number of possible values. (e.g., March 4, 1979).
- If this attribute is added to the PlayTennis data, it would have the highest information gain of any of the attributes. This is because Date alone perfectly predicts the target attribute over the training data. Thus, it would be selected as the decision attribute for the root node of the tree and lead to a tree of depth one, which perfectly classifies the training data.
- This decision tree with root node Date is not a useful predictor because it perfectly separates the training data, but poorly predict on subsequent examples.

One Approach: Use GainRatio instead of Gain

The gain ratio measure penalizes attributes by incorporating a split information, that is sensitive to how broadly and uniformly the attribute splits the data

$$GainRatio(S,A) \equiv \frac{Gain(S,A)}{SplitInformation(S,A)}$$

$$SplitInformation(S,A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Where, Si is subset of S, for which attribute A has value vi

4. Handling Training Examples with Missing Attribute Values

The data which is available may contain missing values for some attributes
Example: Medical diagnosis

- <Fever = true, Blood-Pressure = normal, …, Blood-Test = ?, …>
- Sometimes values truly unknown, sometimes low priority (or cost too high)

Strategies for dealing with the missing attribute value

- If node n test A, assign most common value of A among other training examples sorted to node n
- Assign most common value of A among other training examples with same target value
- Assign a probability pi to each of the possible values vi of A rather than simply assigning the most common value to A(x)

5. Handling Attributes with Differing Costs

- In some learning tasks the instance attributes may have associated costs.
- For example: In learning to classify medical diseases, the patients described in terms of attributes such as Temperature, BiopsyResult, Pulse, BloodTestResults, etc.
- These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort
- Decision trees use low-cost attributes where possible, depends only on high-cost attributes only when needed to produce reliable classifications

How to Learn A Consistent Tree with Low Expected Cost?

One approach is replace Gain by Cost-Normalized-Gain

Examples of normalization functions

- Tan and Schlimmer

$$\frac{Gain^2(S,A)}{Cost(A)}.$$

- Nunez

$$\frac{2^{Gain(S,A)}-1}{(Cost(A)+1)^w}$$

where $w \in [0,1]$ determines importance of cost

UUNIT-2

ARTIFICIAL NEURAL
NETWORKS

CONTENT

Introduction

Neural Network Representation

Appropriate Problems for Neural Network Learning

Perceptrons

Multilayer Networks and BACKPROPAGATION Algorithms

Remarks on the BACKPROPAGATION Algorithms

INTRODUCTION

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued,

discrete-valued, and vector-valued target functions from examples.

Biological Motivation

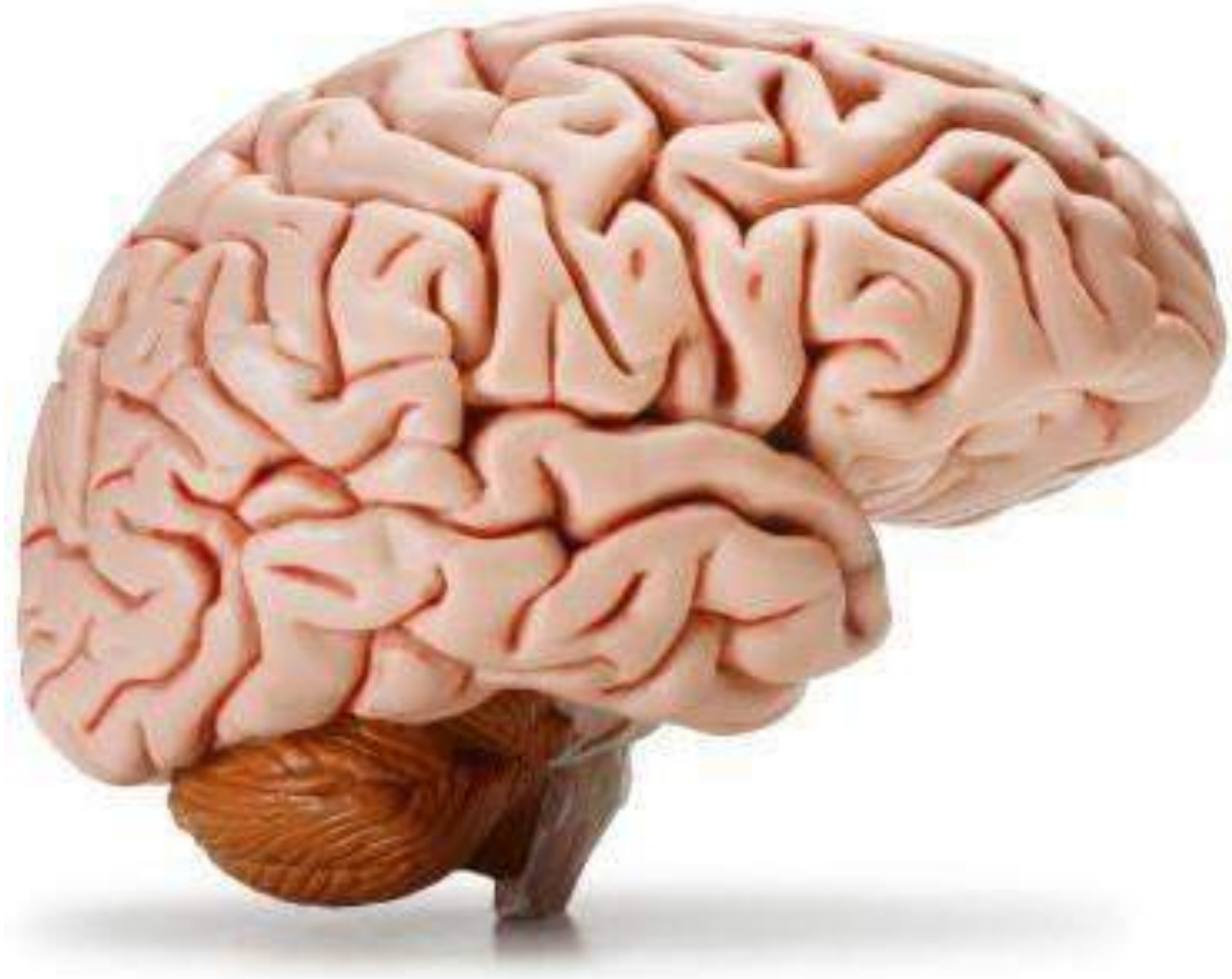
The study of artificial neural networks (ANNs) has been inspired by the observation that biological learning systems are built of very complex webs of interconnected Neurons

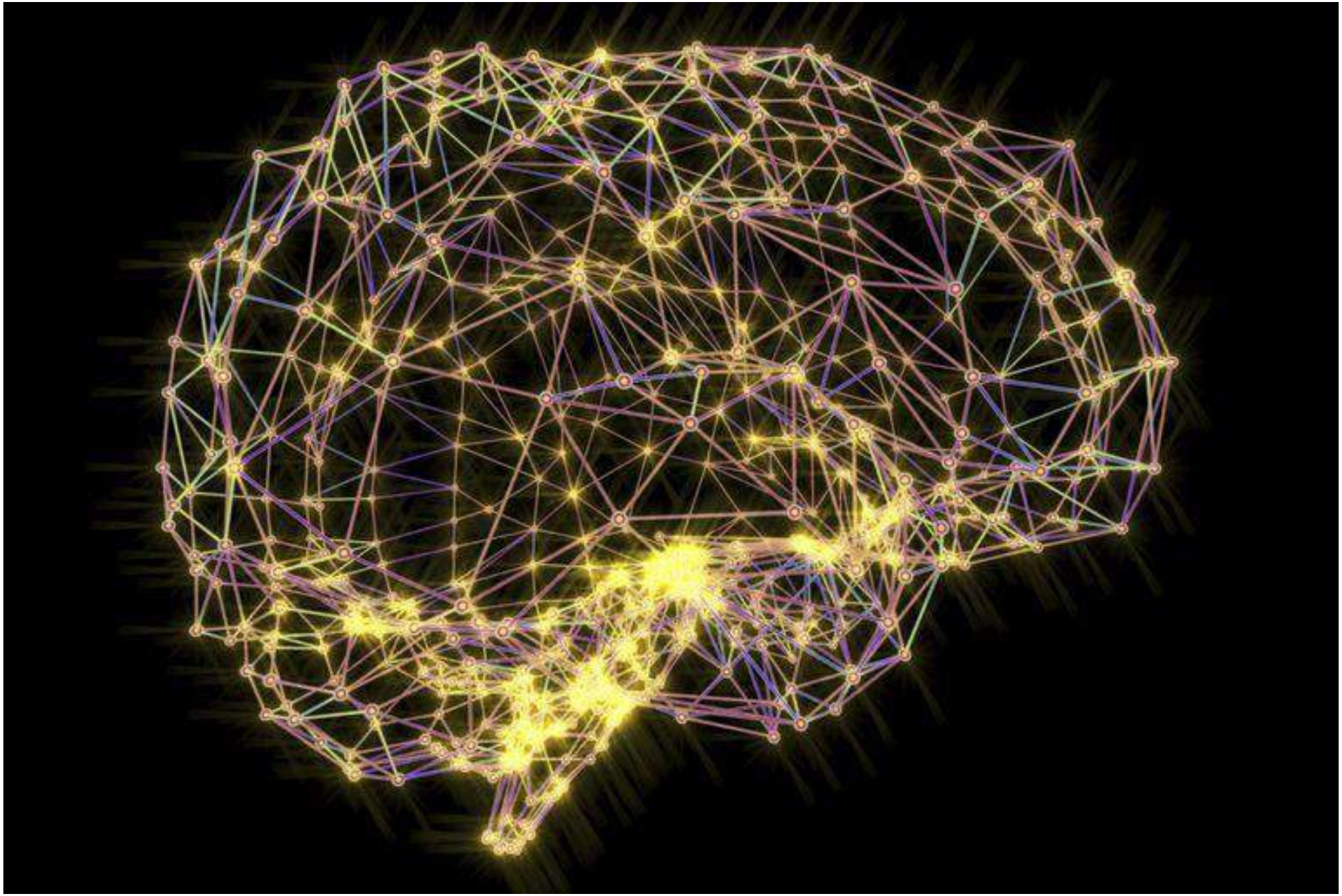
Human information processing system consists of brain neuron: basic building block cell that communicates information to and from various parts of body

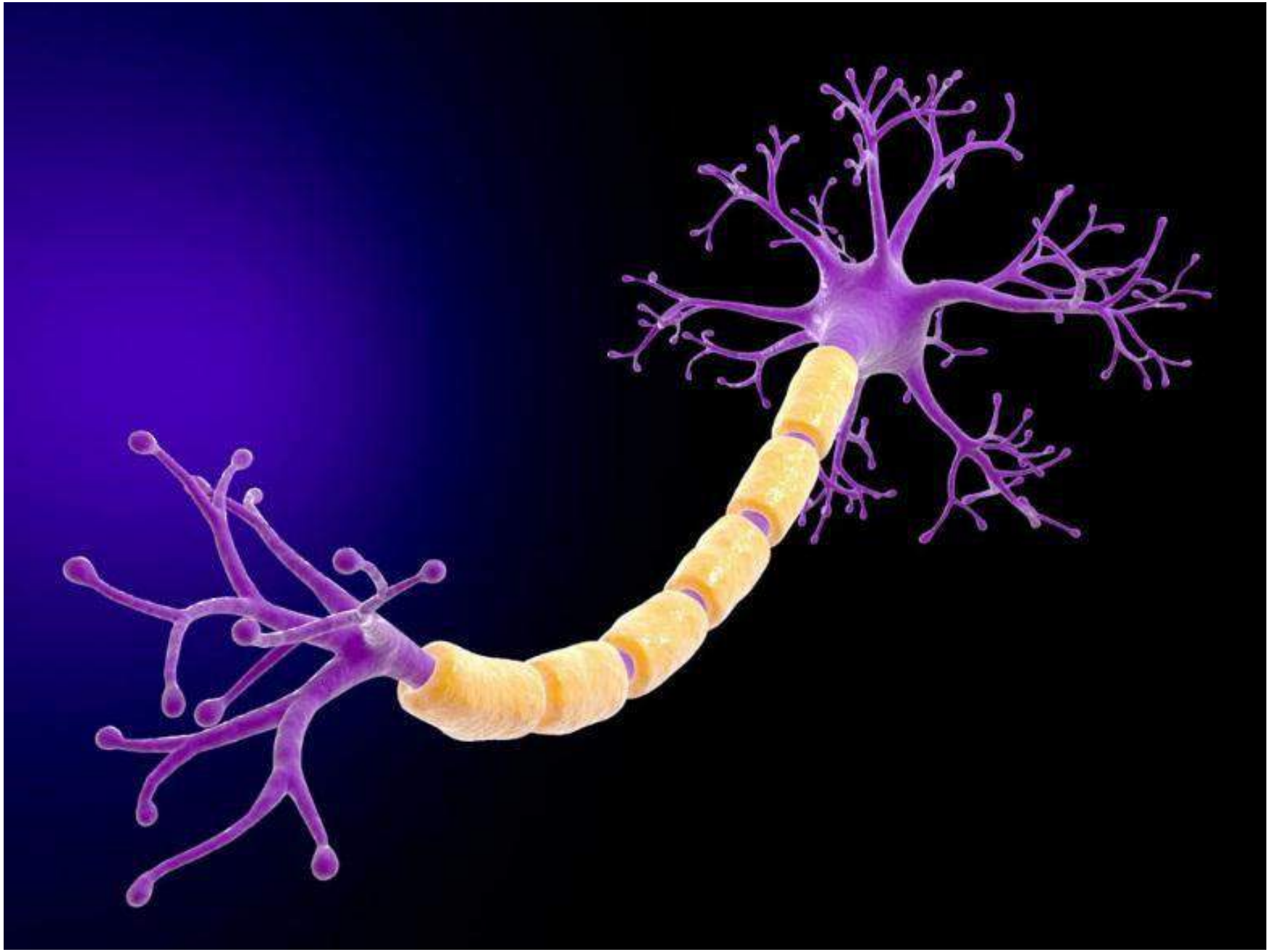
Simplest model of a neuron: considered as a threshold unit –a processing element (PE)

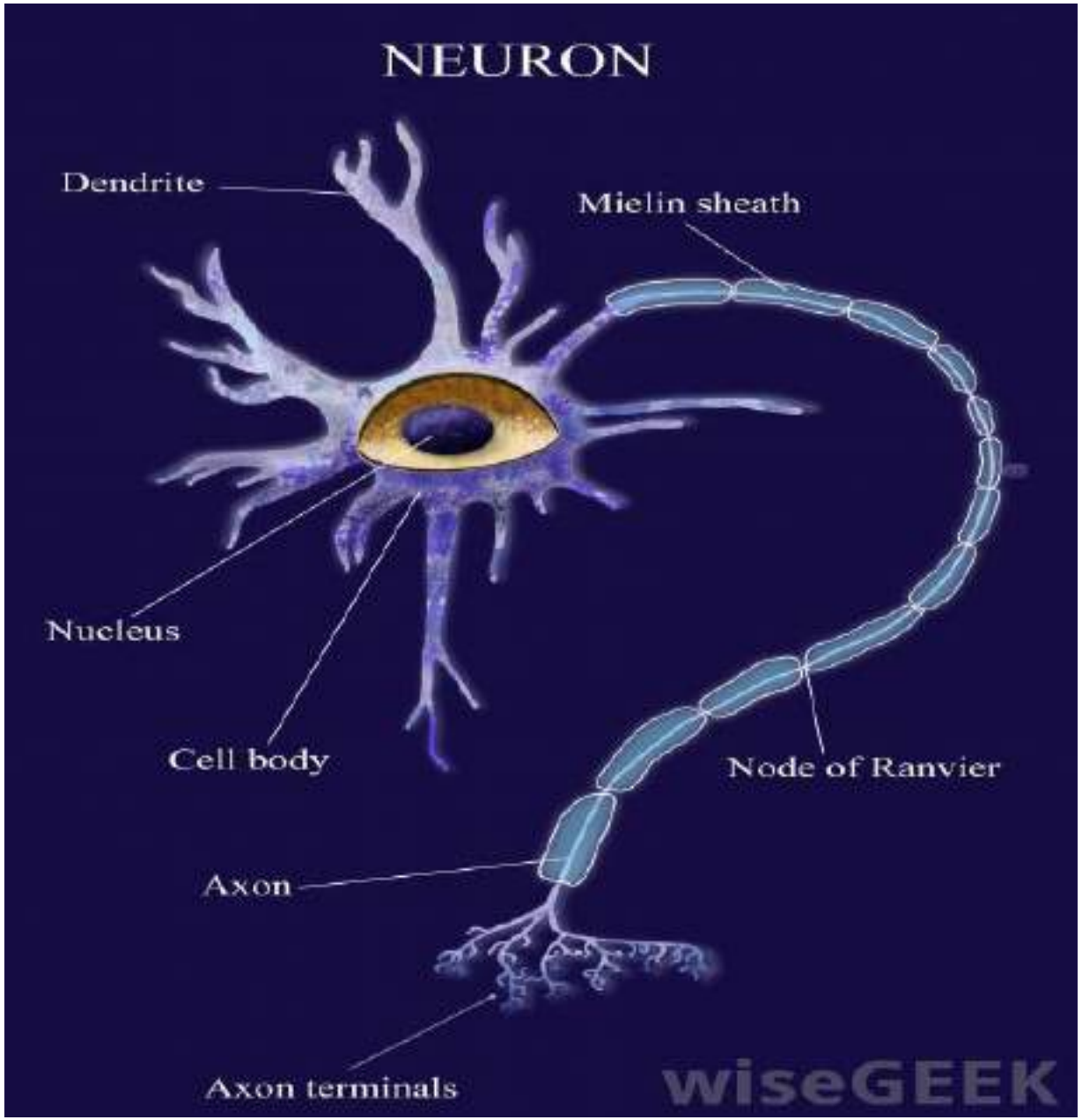
Collects inputs & produces output if the sum of the input exceeds an internal

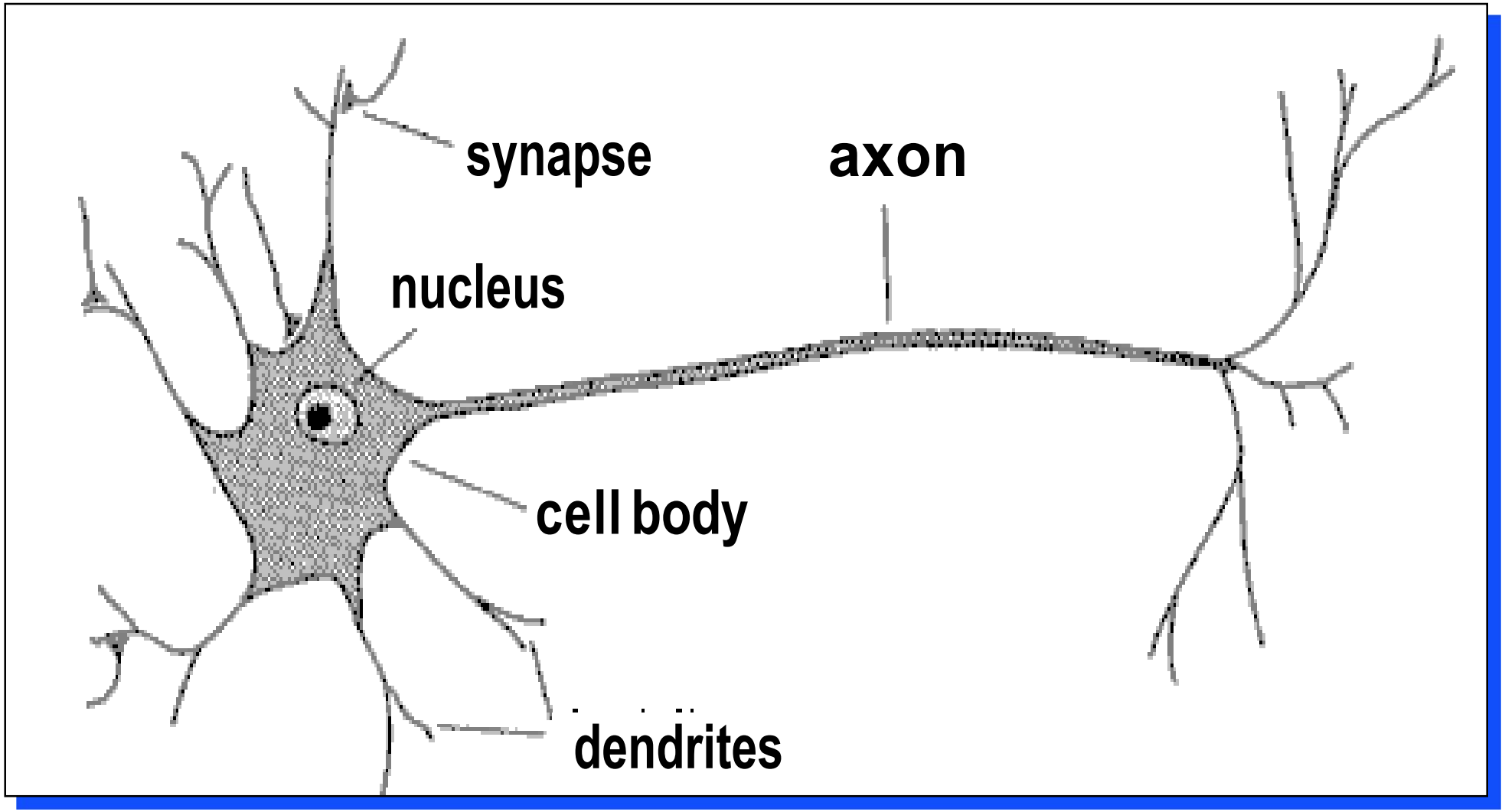
threshold value








NEURON
Dendrite
Mielin sheath
Nucleus
Cell body
Node of Ranvier
Axon
Axon terminals
wiseGEEK


synapse
axon
nucleus
cell body
dendrites

Facts of Human Neurobiology

Number of neurons ~ 1011

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)

When to consider Neural Networks ?

Input is a high-dimensional discrete or real-valued (e.g., sensor input)

Output is discrete or real-valued

Output is a vector of values

Possibly noisy data

Form of target function is unknown

Human readability of result is unimportant

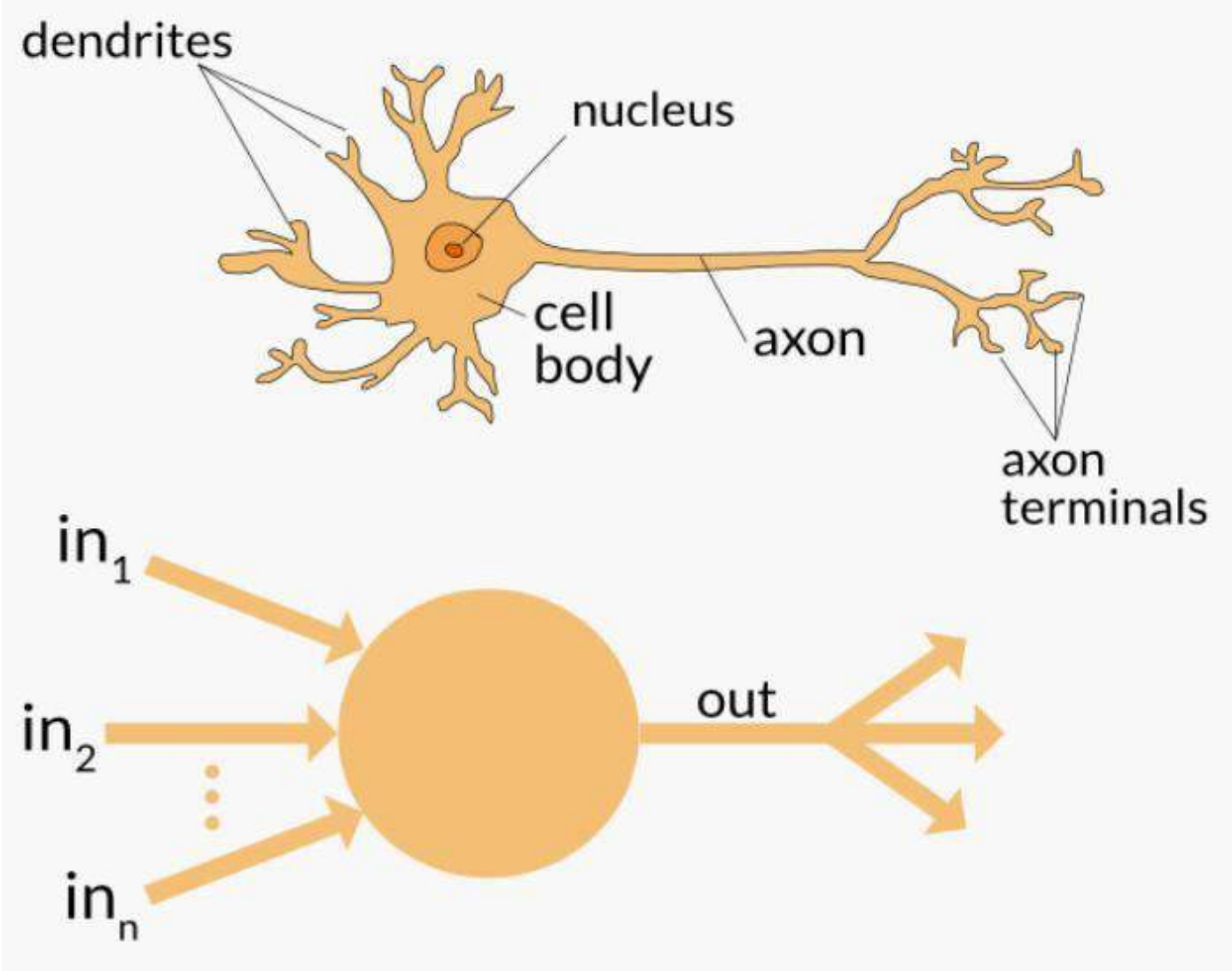
Examples:

Speech phoneme recognition

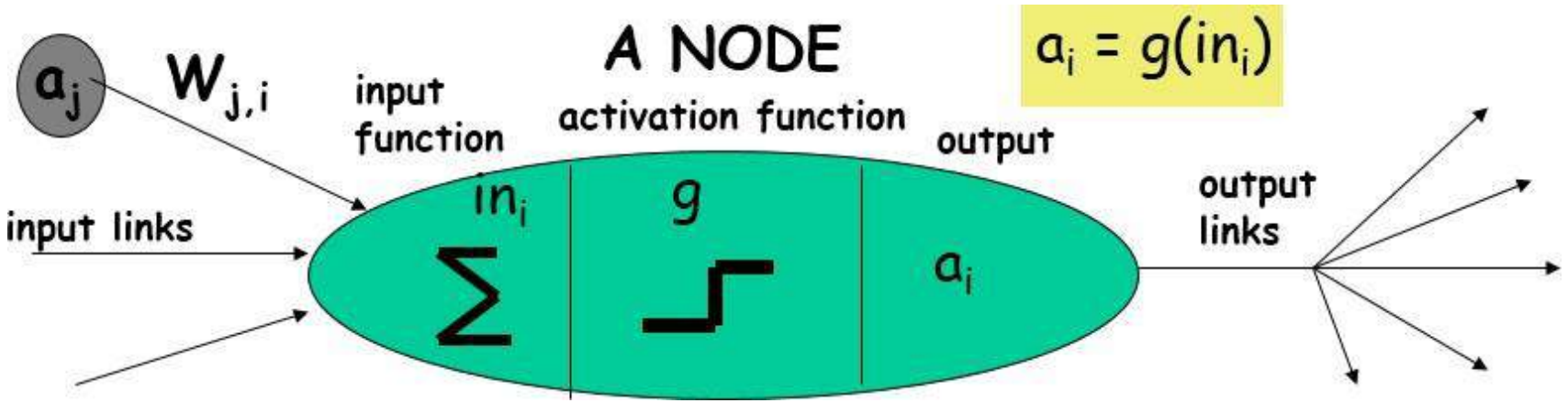
Image classification

Financial perdition

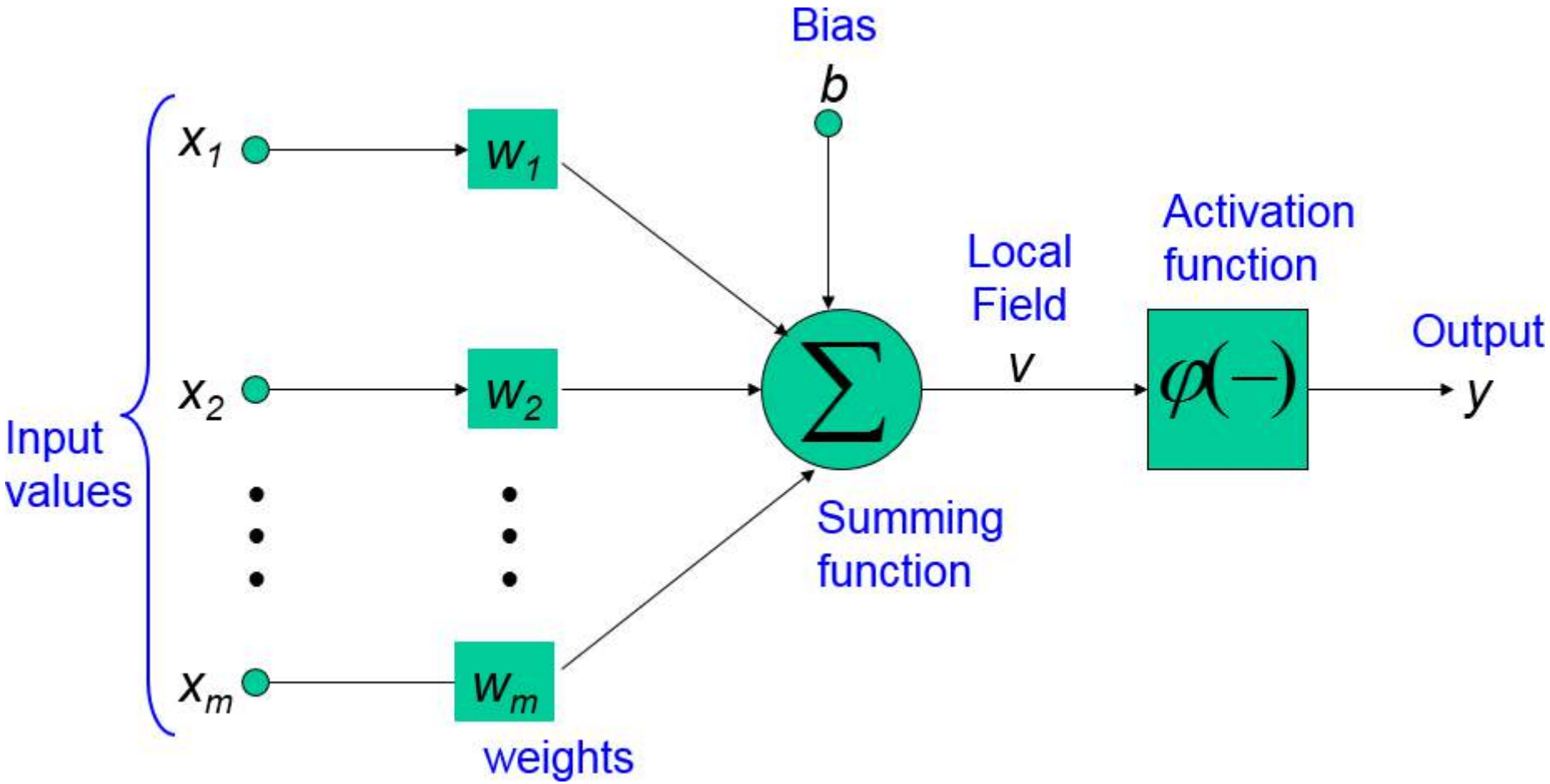
Neuron

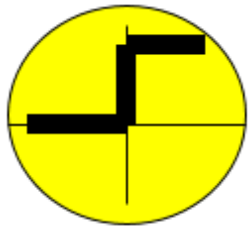

dendrites
nucleus
cell body
axon
axon terminals
in1
in2
inn
out

Neuron


A NODE
$a_i = g(in_i)$
a_j
$W_{j,i}$
input function
activation function
output
in_i
g
a_i
input links
output links

Neuron

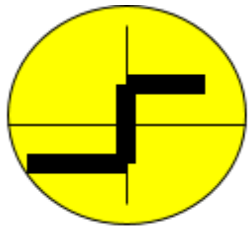

Bias
b
x_1
w_1
x_2
w_2
x_m
w_m
Input values
weights
Local Field
v
Activation function
$\varphi(-)$
Output
y
Summing function



Step function

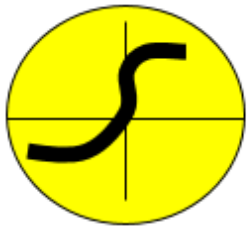
(Linear Threshold Unit)

step(x) = 1, if x >= threshold
0, if x < threshold



Sign function

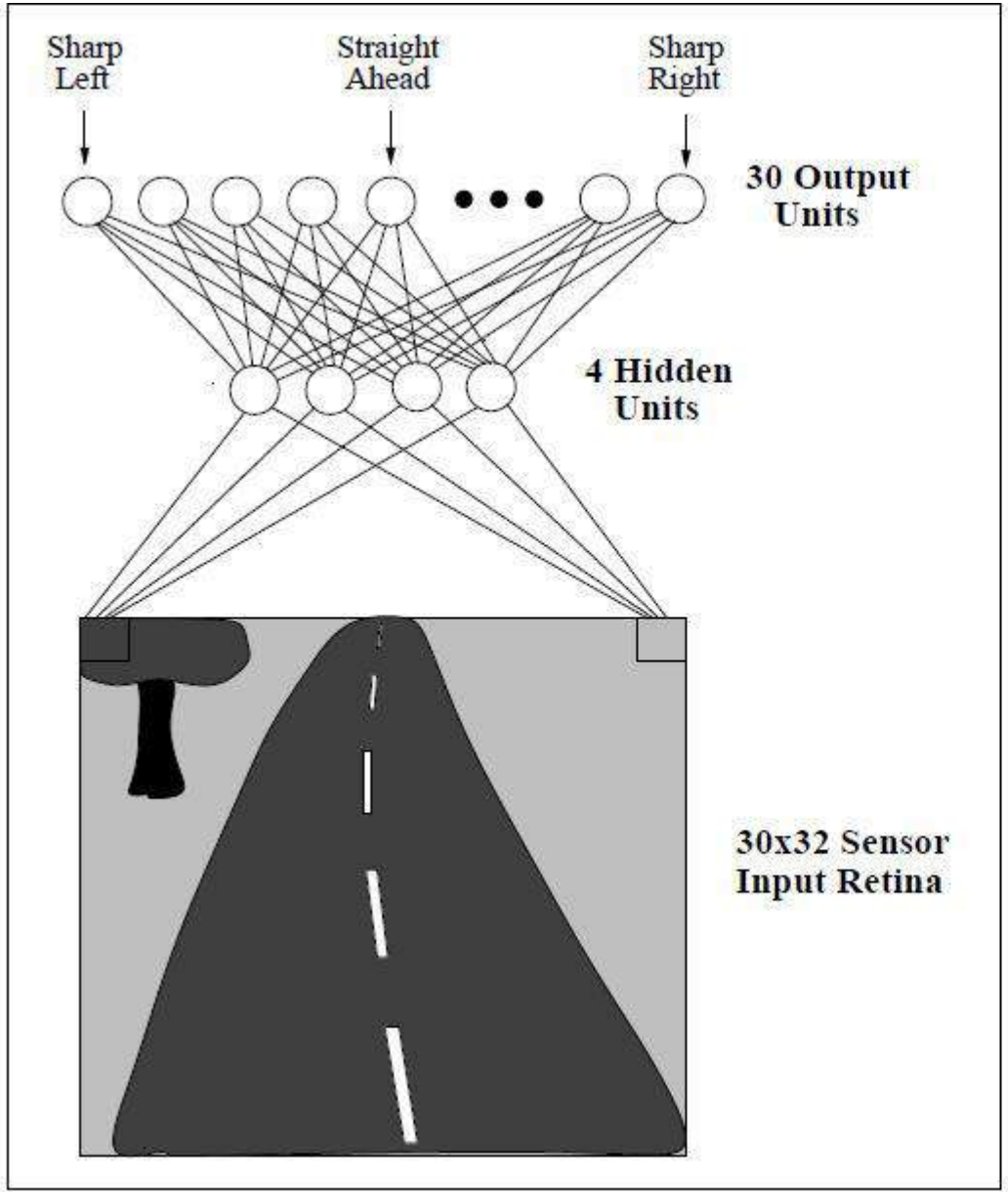
sign(x) = +1, if x >= 0
-1, if x < 0

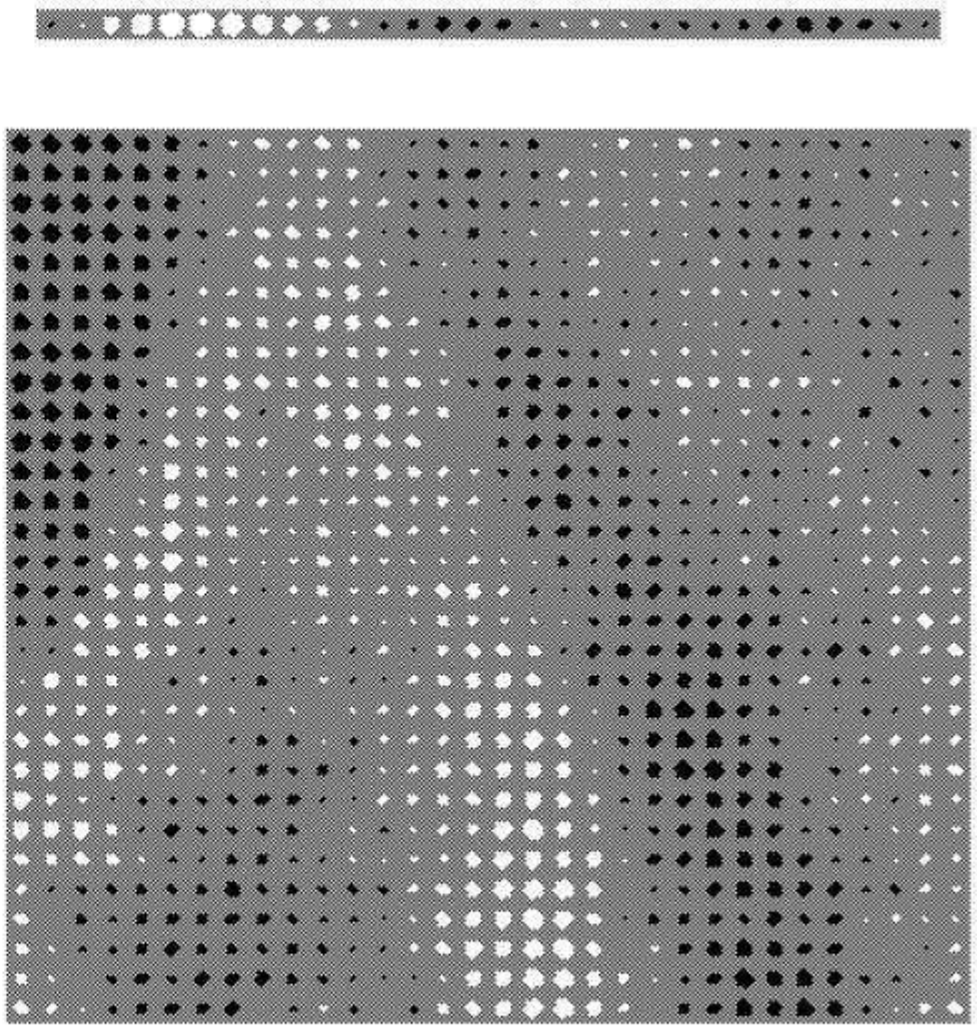


Sigmoid function

sigmoid(x) = $1/(1+e^{-x})$

NEURAL NETWORK REPRESENTATIONS


Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



A prototypical example of ANN learning is provided by Pomerleau's (1993) 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 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 is appropriate for problems with the following characteristics :

Instances are represented by many attribute-value pairs.

The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.

The training examples may contain errors.

Long training times are acceptable.

Fast evaluation of the learned target function may be required

The ability of humans to understand the learned target function is not important

Architectures of Artificial Neural Networks

An artificial neural network can be divided into three parts (layers), which are known as:

Input layer: This layer is responsible for receiving information (data), signals, features, or measurements from the external environment. These inputs are usually normalized within the limit values produced by activation functions

Hidden, intermediate, or invisible layers: These layers are composed of neurons which are responsible for extracting patterns associated with the process or system being analysed. These layers perform most of the internal processing from a network.

Output layer : This layer is also composed of neurons, and thus is responsible for producing and presenting the final network outputs, which result from the processing performed by the neurons in the previous layers.

Architectures of Artificial Neural Networks

The main architectures of artificial neural networks, considering the neuron disposition, how they are interconnected and how its layers are composed, can be divided as follows:

Single-layer feedforward network

Multi-layer feedforward networks

Recurrent or Feedback networks

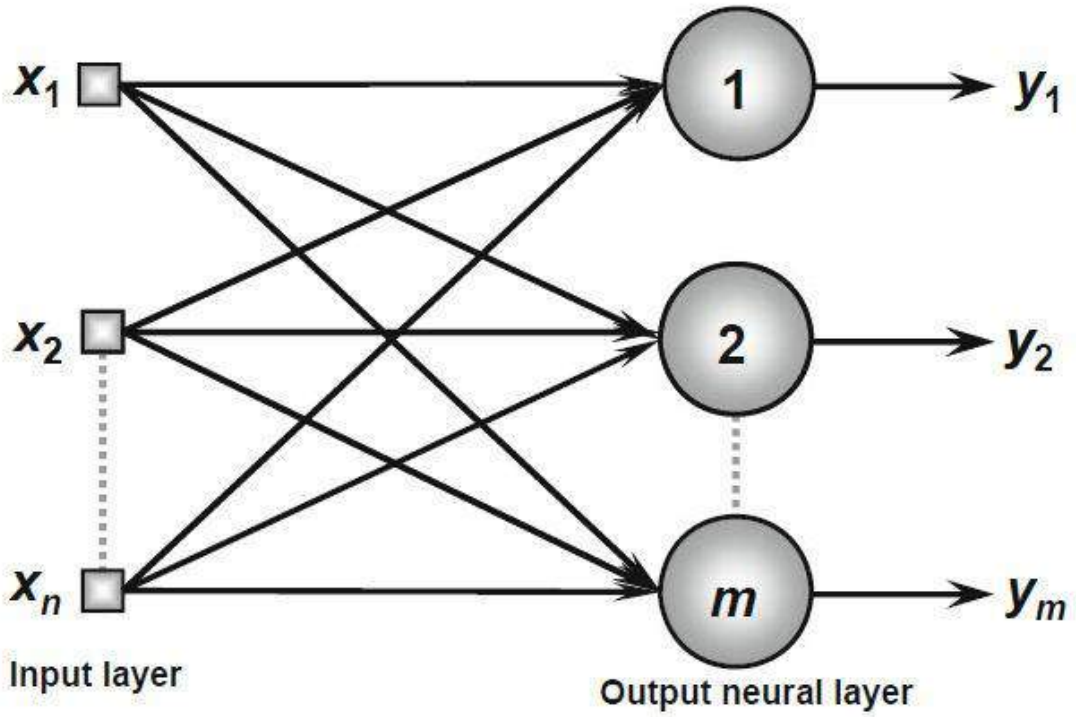
Mesh networks

Single-Layer Feedforward Architecture

This artificial neural network has just one input layer and a single neural layer, which is also the output layer.

Figure illustrates a simple-layer feedforward network composed of n inputs and m outputs.

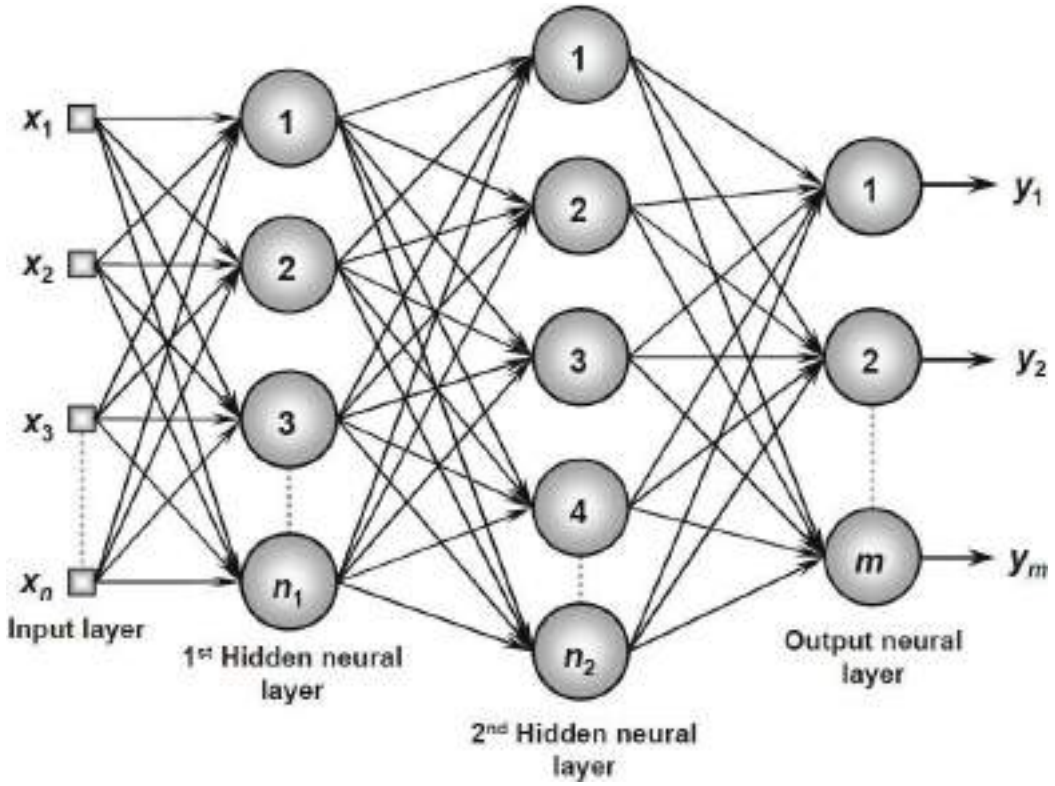
The information always flows in a single direction (thus, unidirectional), which is from the input layer to the output layer



Multi-Layer Feedforward Architecture

This artificial neural feedforward networks with multiple layers are composed of one or more hidden neural layers.

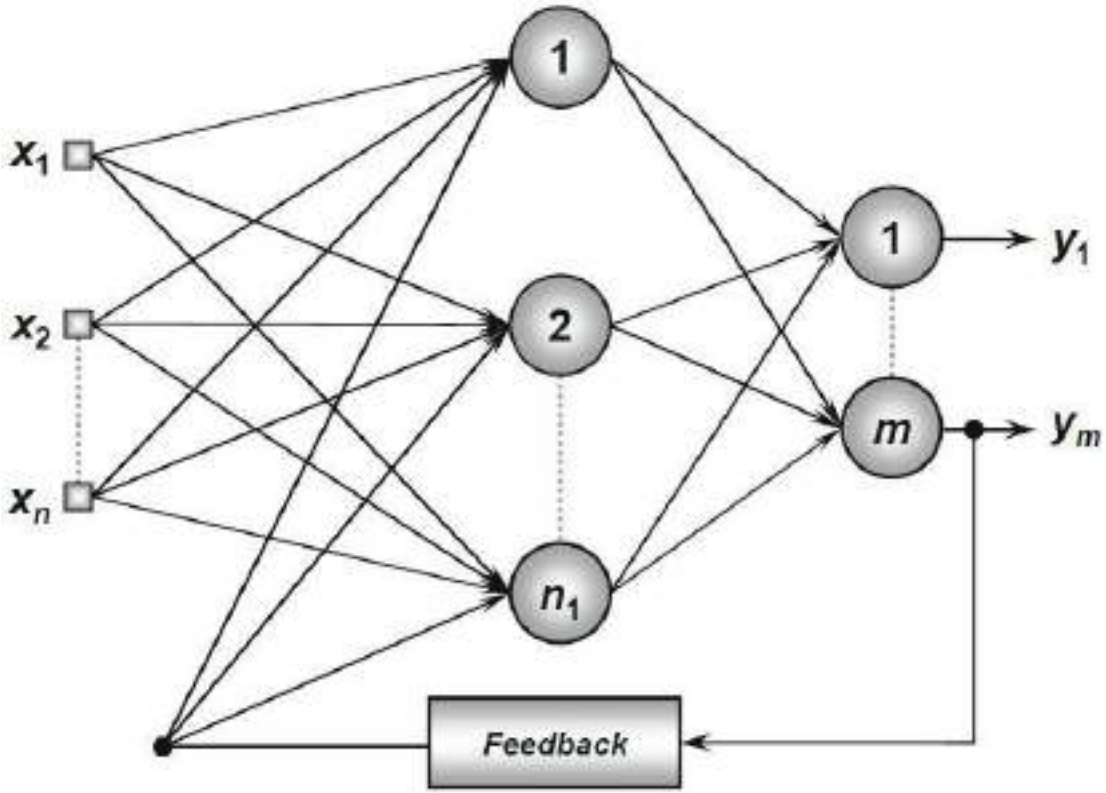
Figure shows a feedforward network with multiple layers composed of one input layer with n sample signals, two hidden neural layers consisting of n1 and n2 neurons respectively, and, finally, one output neural layer composed of m neurons representing the respective output values of the problem being analyzed



Recurrent or Feedback Architecture

In these networks, the outputs of the neurons are used as feedback inputs for other neurons.

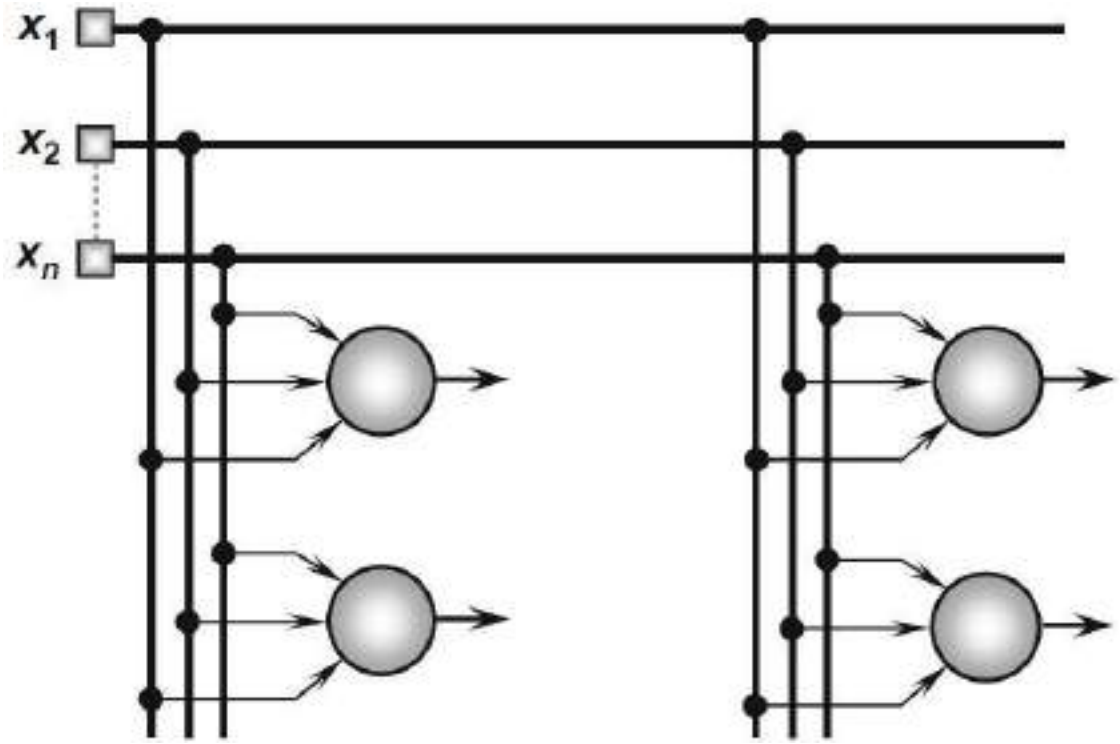
Figure illustrates an example of a Perceptron network with feedback, where one of its output signals is fed back to the middle layer.


1
2
n_1
x_1
x_2
x_n
1
m
y_1
y_m
Feedback

Mesh Architectures

The main features of networks with mesh structures reside in considering the spatial arrangement of neurons for pattern extraction purposes, that is, the spatial localization of the neurons is directly related to the process of adjusting their synaptic weights and thresholds.

Figure illustrates an example of the Kohonen network where its neurons are arranged within a two- dimensional space



PERCEPTRONS

Perceptron is a single layer neural network.

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 x1 through xn, the output O(x1, . . . , xn) computed by the perceptron is

where each wi is a real-valued constant, or weight, that determines the contribution

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

of input xi to the perceptron output.

-w0 is a threshold that the weighted combination of inputs w1x1 + . . . + wnxn 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 w0 , . . . , wn . Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

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

Why do we need Weights and Bias?

Weights shows the strength of the particular node.

A bias value allows you to shift the activation function curve up or down

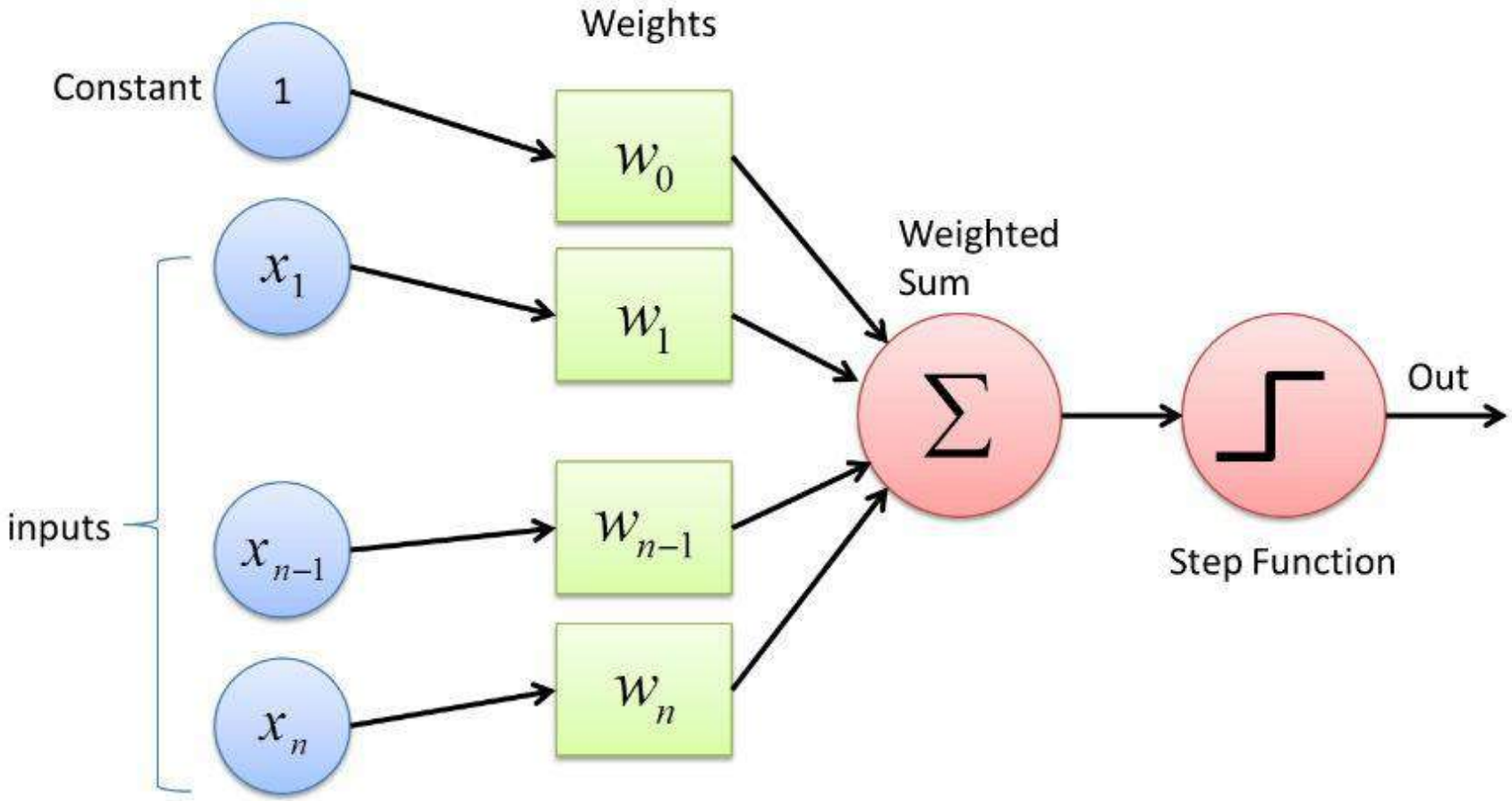
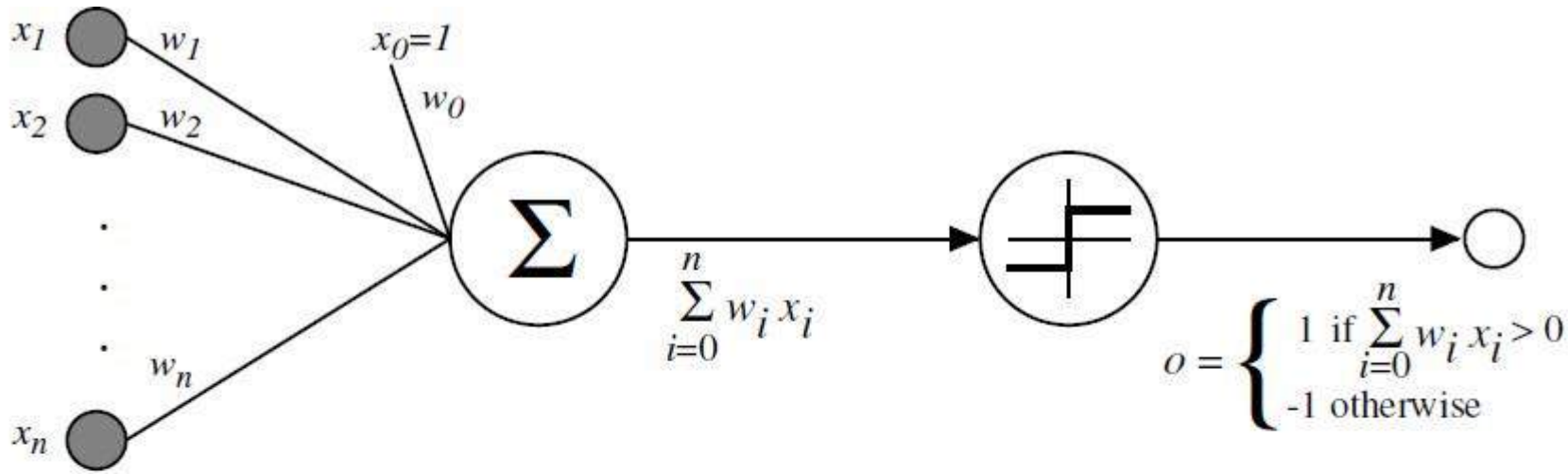


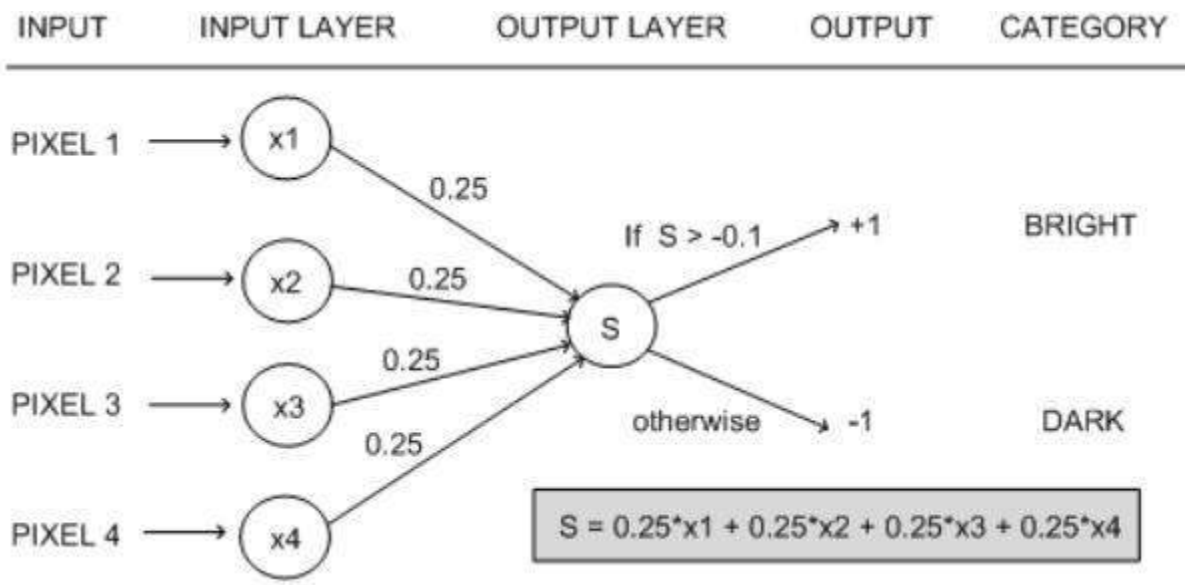
Fig : Perceptron



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

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$


INPUT
INPUT LAYER
OUTPUT LAYER
OUTPUT
CATEGORY
PIXEL 1
PIXEL 2
PIXEL 3
PIXEL 4
x1
x2
x3
x4
0.25
0.25
0.25
0.25
S
If S > -0.1
+1
BRIGHT
otherwise
-1
DARK
S = 0.25*x1 + 0.25*x2 + 0.25*x3 + 0.25*x4

Representational Power of Perceptrons

The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances.

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

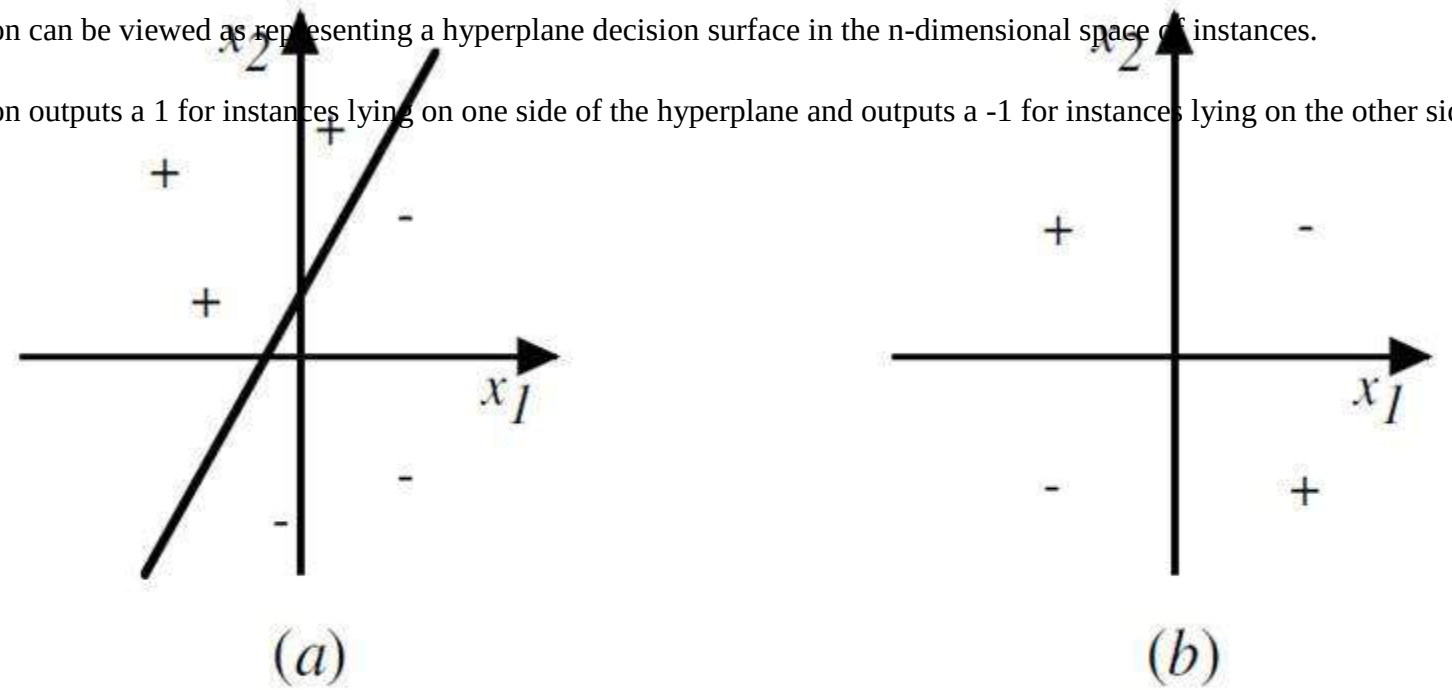
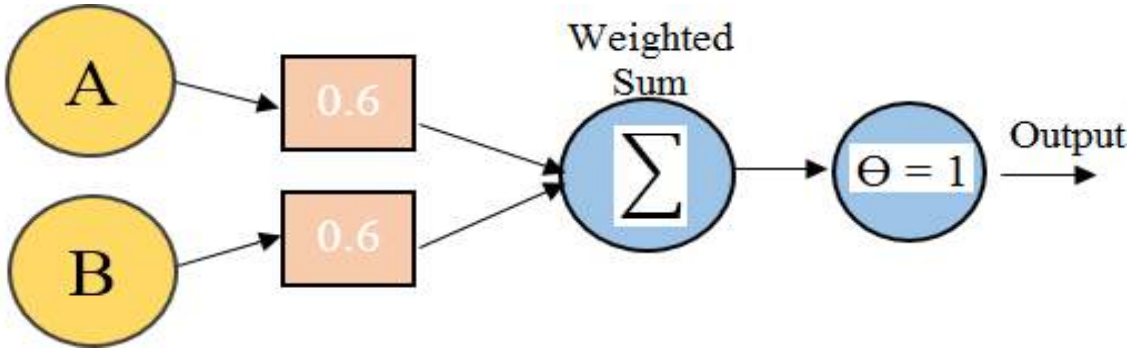


Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated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

A single perceptron can be used to represent many Boolean functions

AND function

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.

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 wi 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,

td is the target output for training example d,

od is the output of the linear unit for training example d

E [w] is simply half the squared difference between the target output td and the linear unit output od, 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.

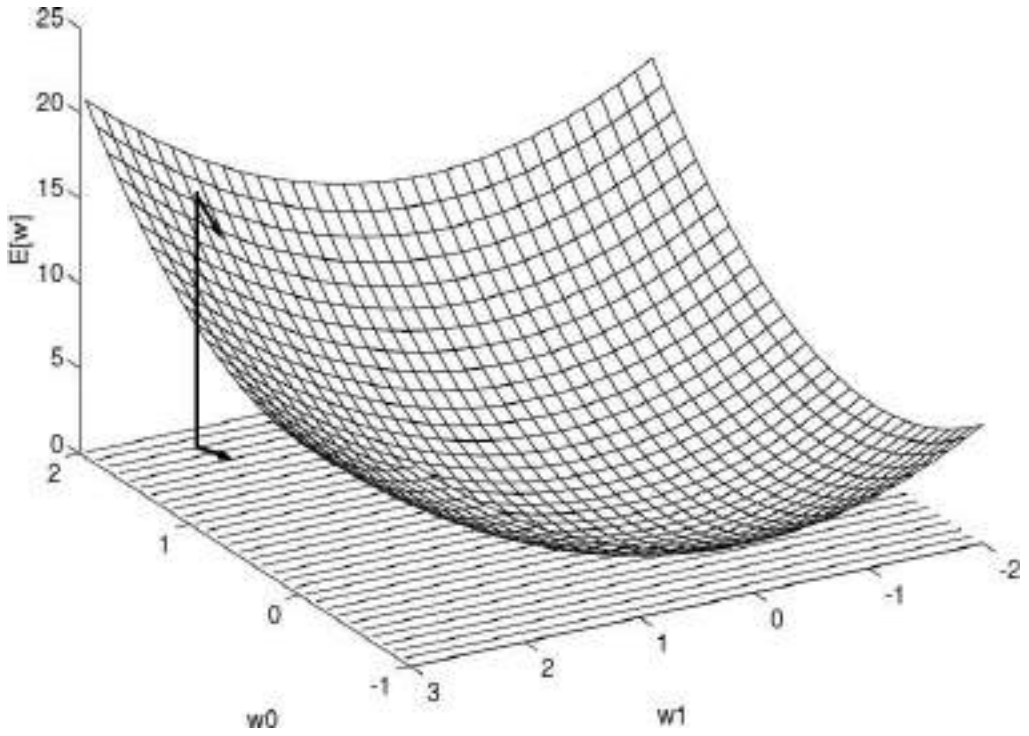
Here the axes w0 and wl represent possible values for the two weights of a simple linear unit. The w0, wl plane therefore represents the entire hypothesis space.

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 w0, wl 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

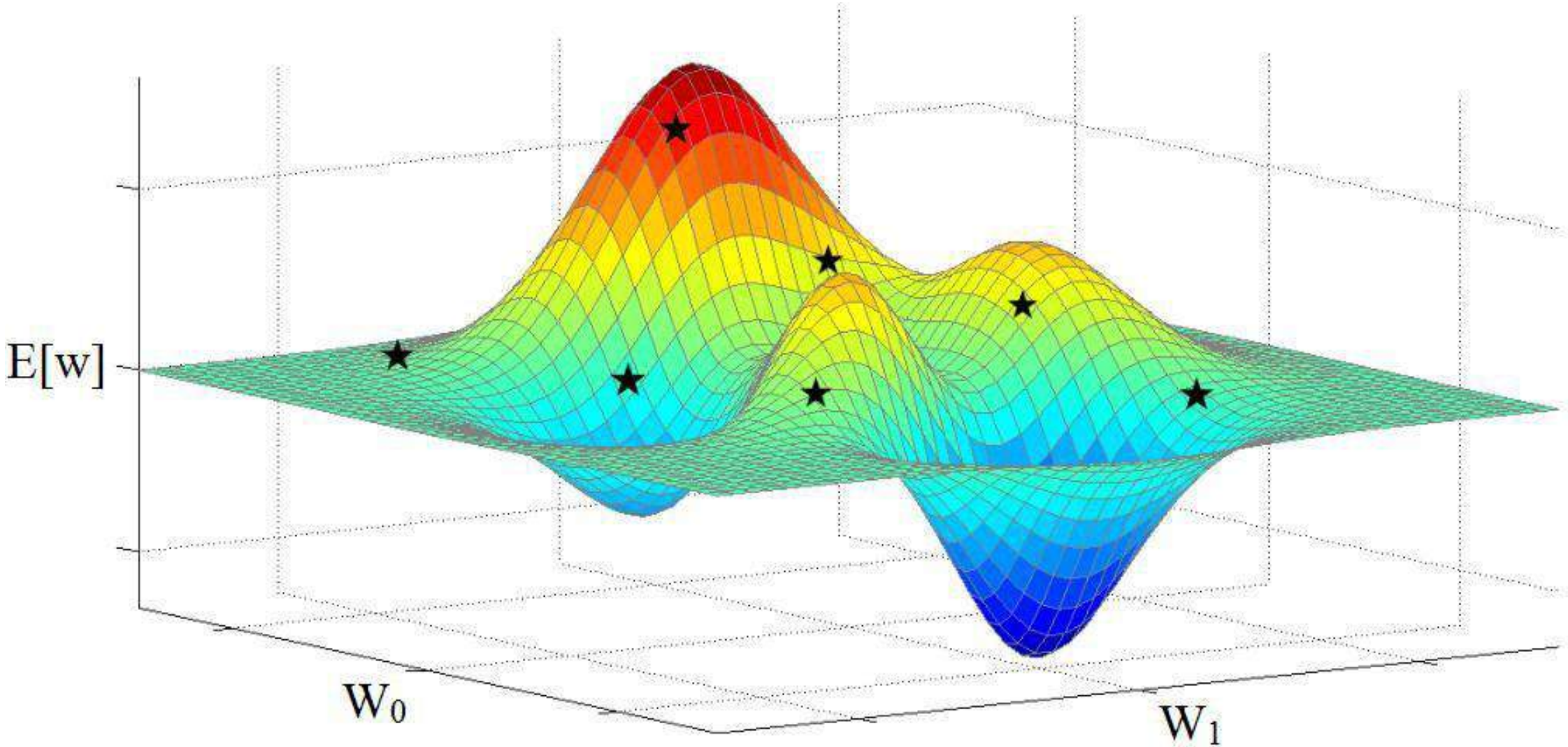


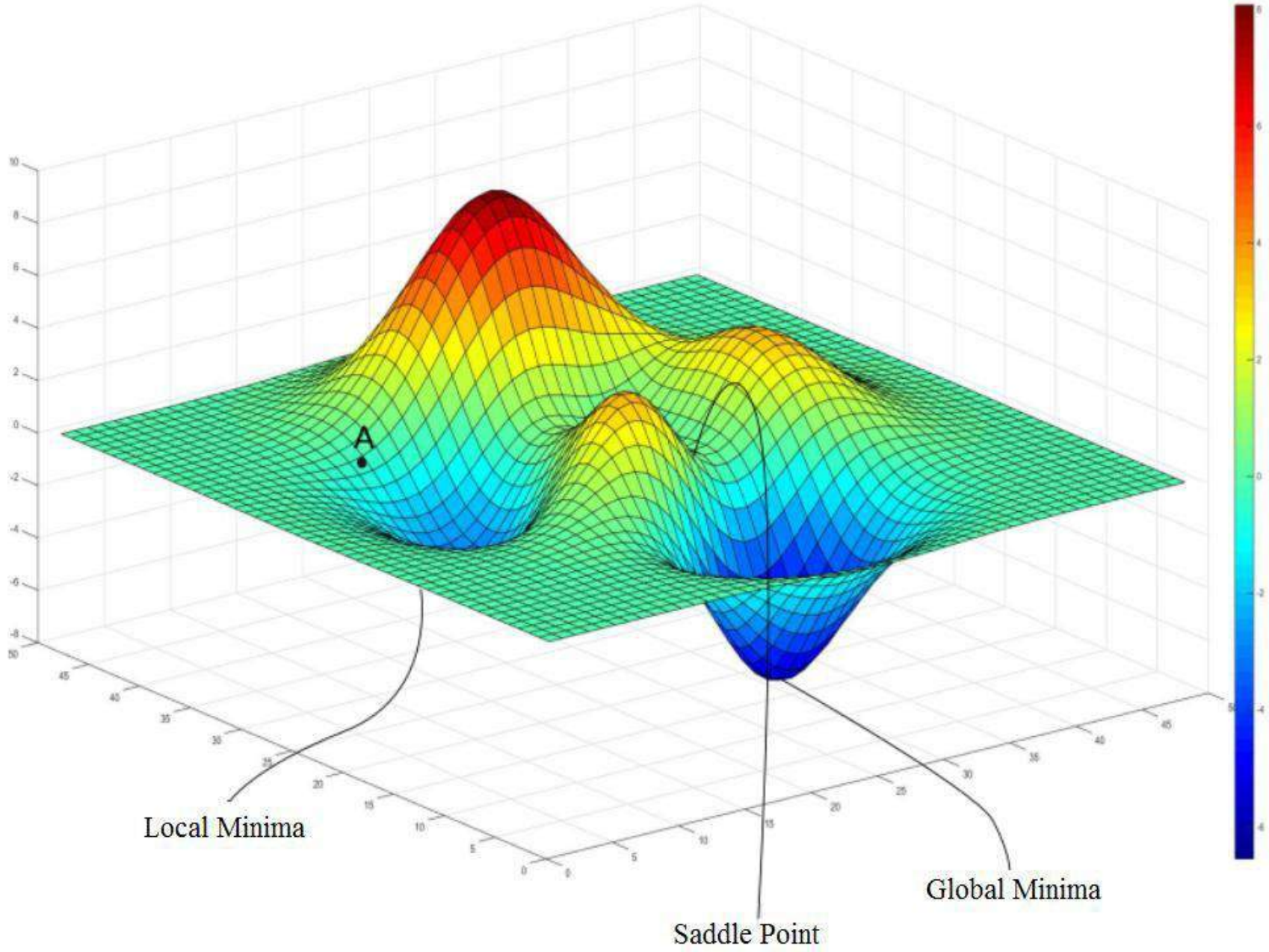
Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

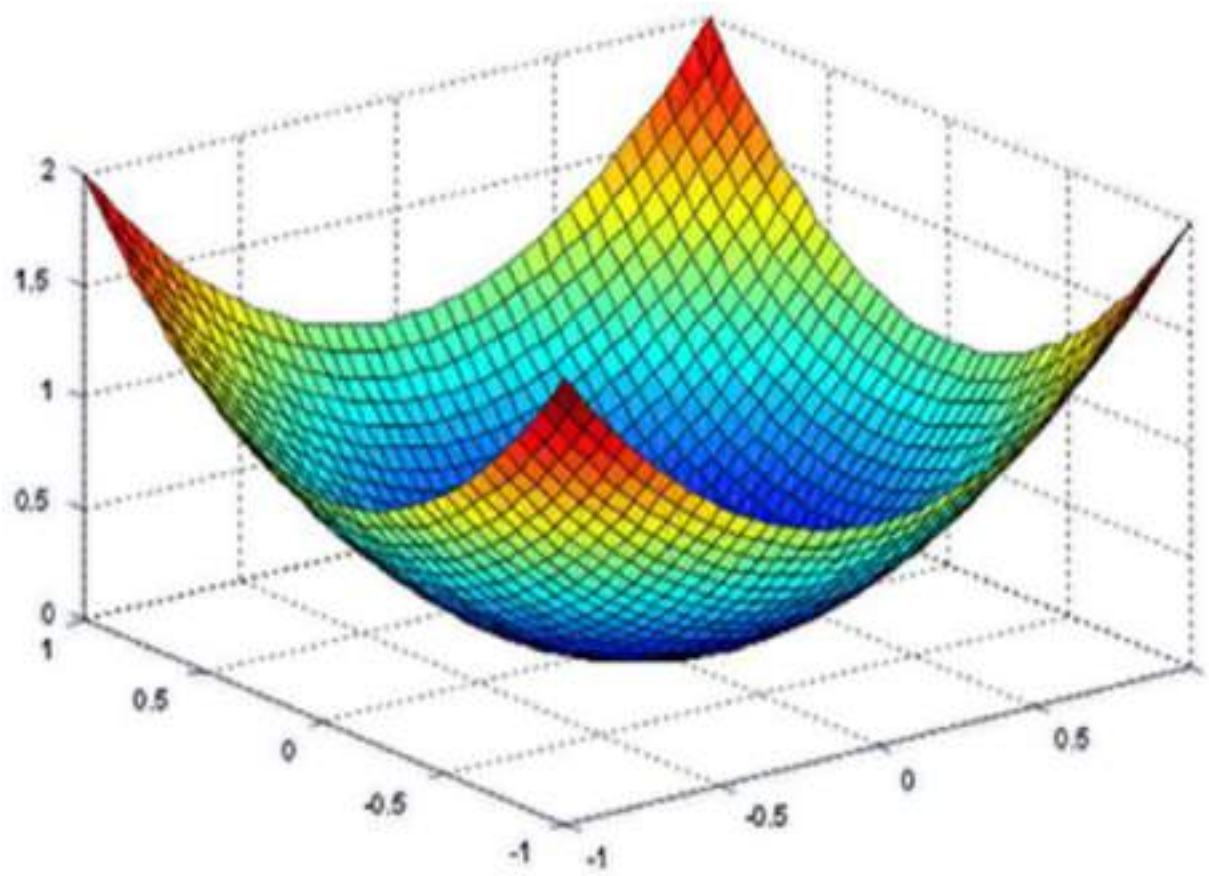
Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary

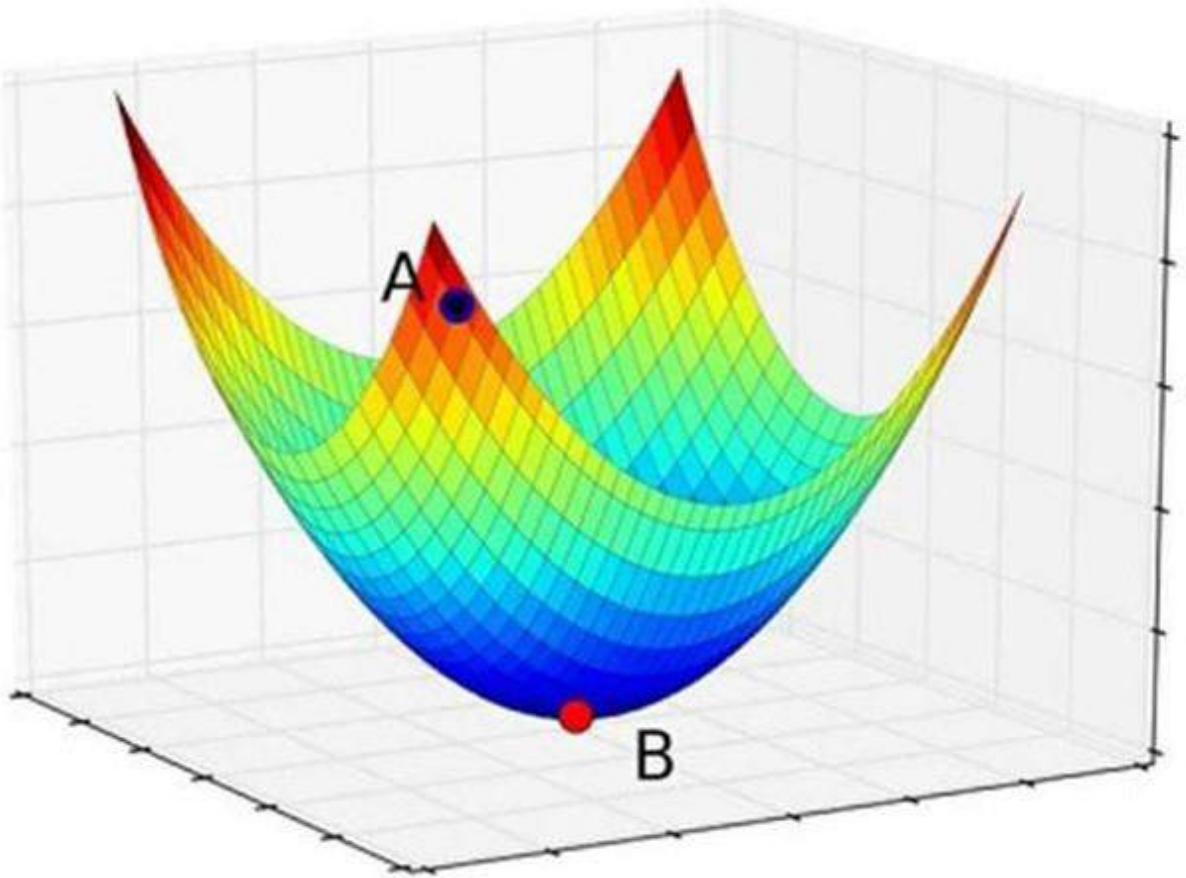
initial weight vector, then repeatedly modifying it in small steps.

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.


E[w]
W0
W1


A
Local Minima
Saddle Point
Global Minima


2
1.5
1
0.5
0
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5


A
B

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 w . This vector derivative is called the gradient of E with respect to w , written as

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

Notice $\nabla E[\vec{w}]$ is itself a vector, whose components are the partial derivatives of ***E*** with respect to each of the w_i

When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E.

The negative of this vector therefore gives the direction of steepest decrease.

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 □□ ___ derivatives that form the gradient can be obtained by

□□□

differentiating E from Equation (2), as

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

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*, 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

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

  - 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

Features of 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

The hypothesis space contains continuously parameterized hypotheses

The error can be differentiated with respect to these hypothesis parameters

The key practical difficulties in applying gradient descent are

Converging to a local minimum can sometimes be quite slow

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 xi are the target value, unit output, and ith 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

$$w_i \leftarrow w_i + \eta(t - o)\, x_i \tag{1}$$

stochastic approximation to gradient descent

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

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

One way to view this stochastic gradient descent is to consider a distinct error function Ed (w) for each individual training example d as follows

Where, td and od are the target value and the unit output value for training example d.

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

Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to Ed(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 Ed (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 $\nabla E_d(w)$ rather than $\nabla E(w)$ to guide its search

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

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

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Multilayer networks learned by the BACKPROPACATION algorithm are capable

of expressing a rich variety of nonlinear decision surfaces

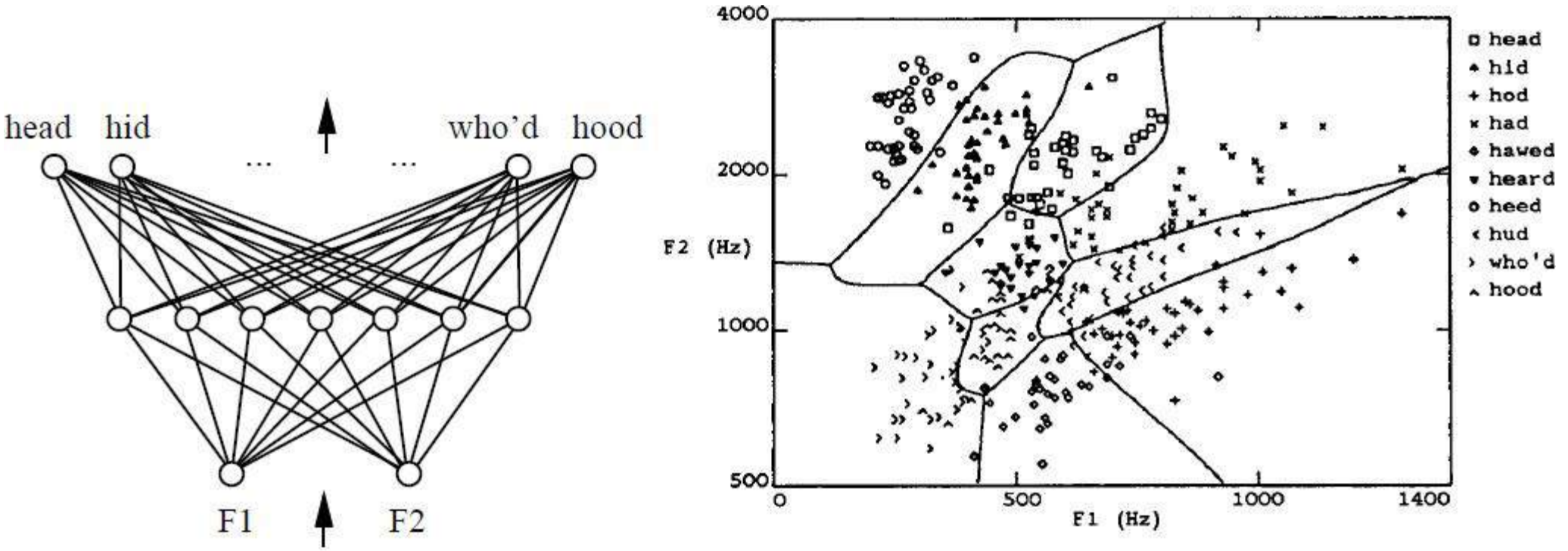


Figure: Decision regions of a multilayer feedforward network.

Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). 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.

A Differentiable Threshold Unit

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

The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, 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

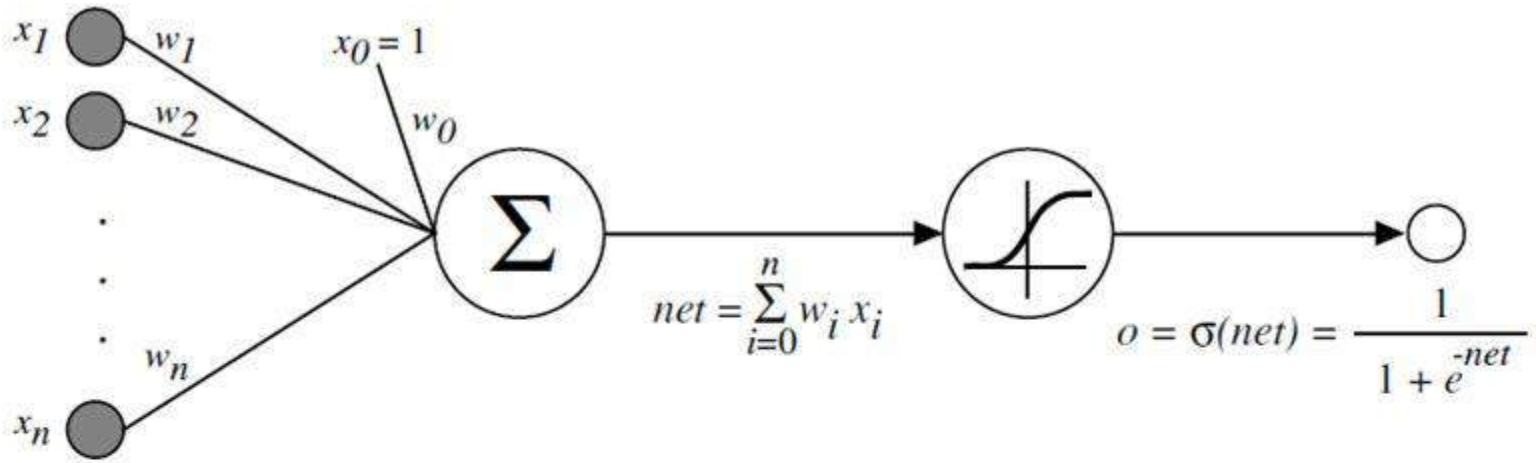


Figure: A Sigmoid Threshold Unit

$\sigma(y)$ is the sigmoid function

$$\frac{1}{1 + e^{-y}}$$

Nice property: $\frac{d\sigma(y)}{dy} = \sigma(y)(1 - \sigma(y))$

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

tkd and Okd - the target and output values associated with the kth output unit

d - training example

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

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

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 Ed with respect to this single example

For each training example d every weight wji is updated by adding to it Δ wji

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

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, tk is the target value of unit k for

training example d, and ok 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

xji = the ith input to unit j

wji = the weight associated with the ith input to unit j netj = Σi wjixji (the weighted sum of inputs for unit j) oj = the output computed by unit j

tj = 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

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

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

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

Consider two cases in turn: 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.

wji can influence the rest of the network only through netj , netj can influence the network only through oj.

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,

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

$$= o_j(1 - o_j) \qquad \text{.....equ(5)}$$

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 wji must take into account the indirect ways in which wji can influence the network outputs and hence Ed.

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).

netj 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:

Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima

Use stochastic gradient descent rather than true gradient descent

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

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

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

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

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.

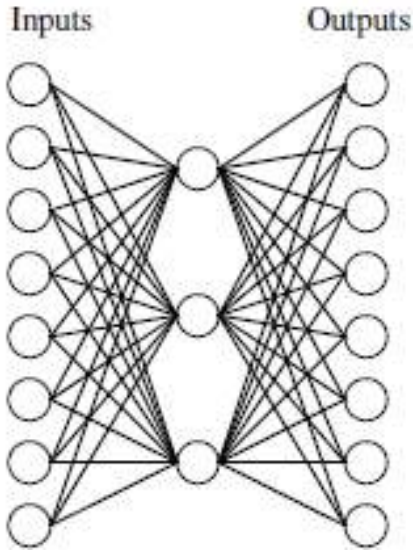
It is difficult to characterize precisely the inductive bias of BACKPROPAGATION algorithm, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points.

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.

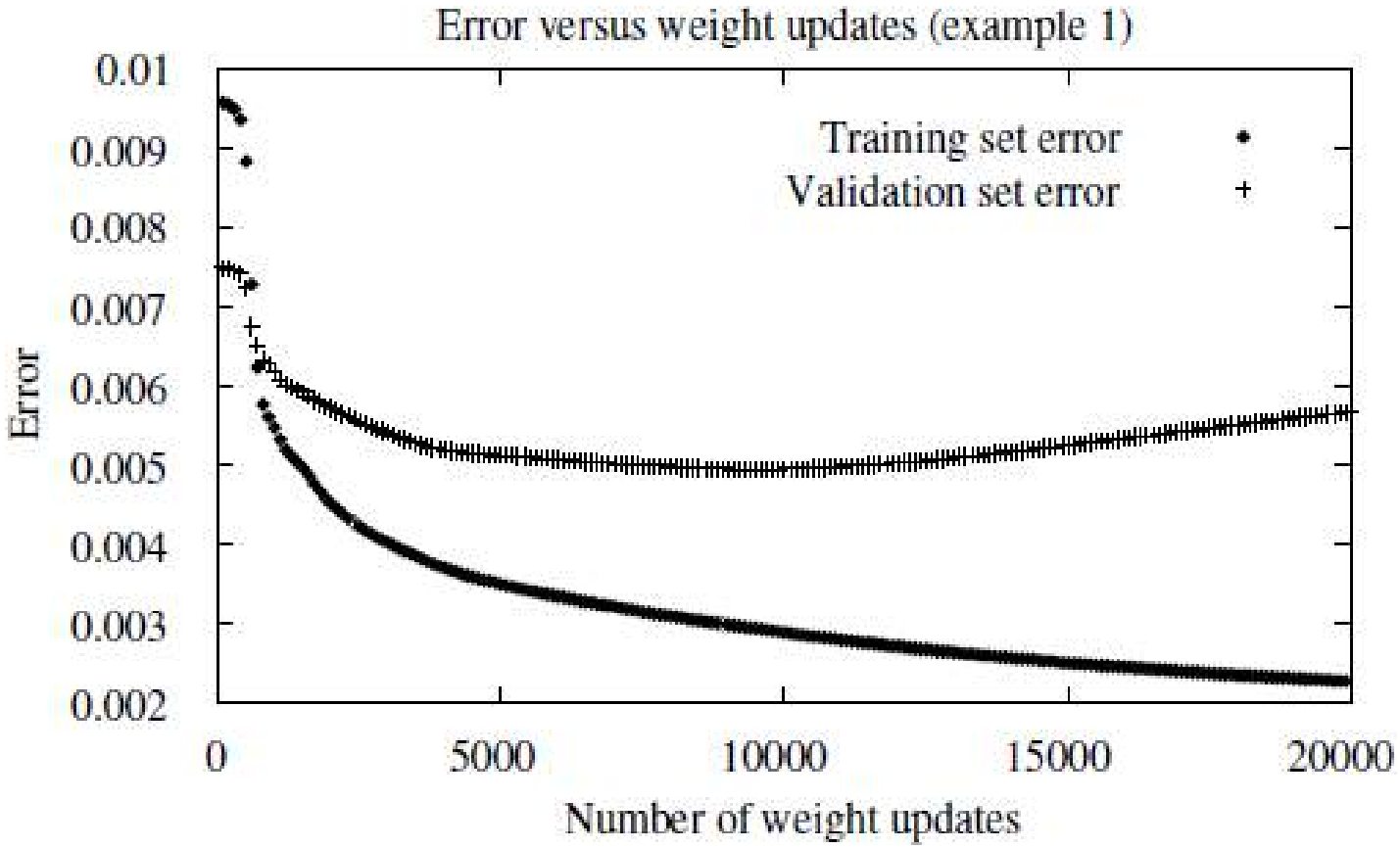
This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning

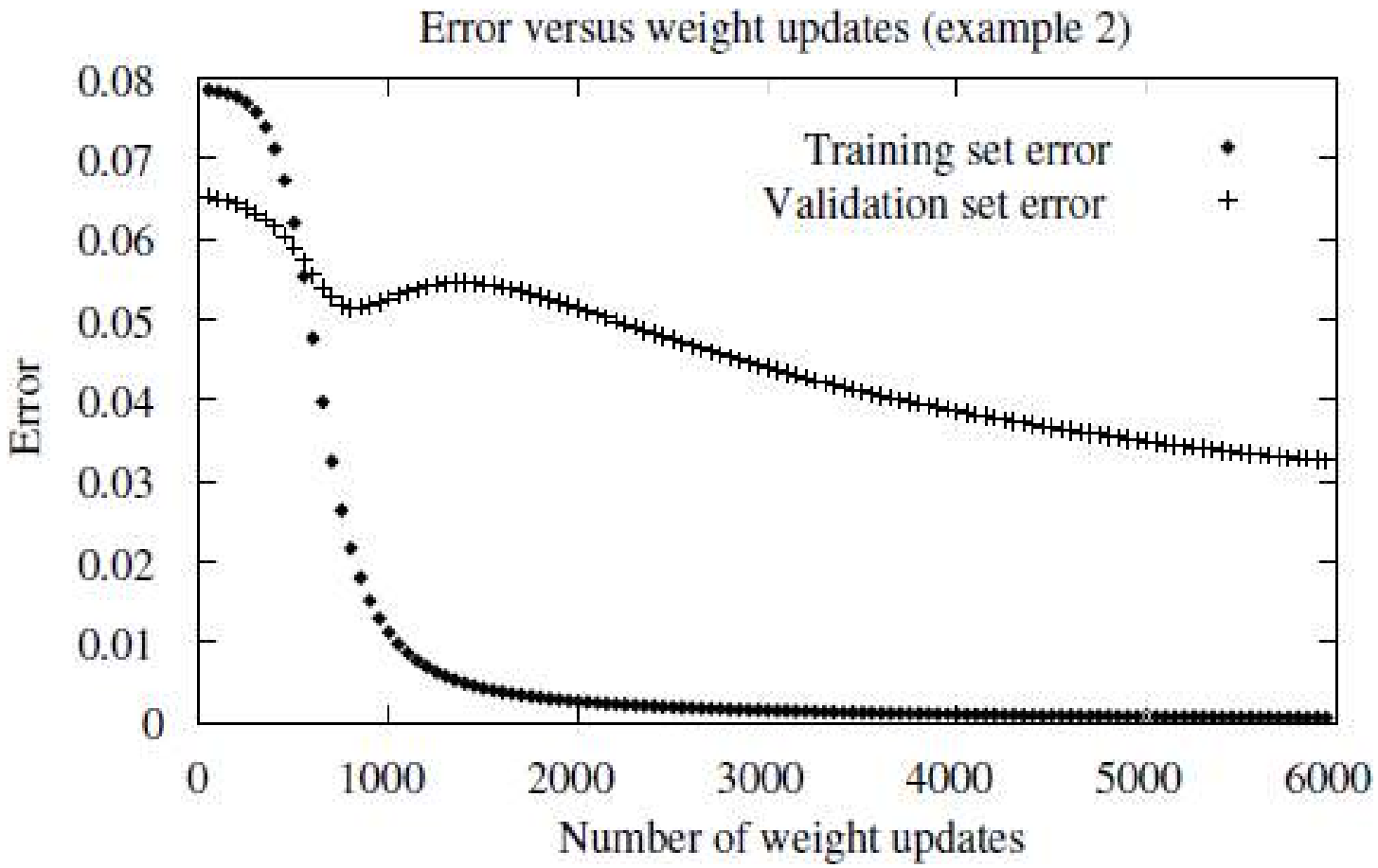
Generalization, Overfitting, and Stopping Criterion

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


Error versus weight updates (example 1)
Training set error
Validation set error
Error
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0
5000
10000
15000
20000
Number of weight updates


Error versus weight updates (example 2)
Training set error
Validation set error
Error
Number of weight updates
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
1000
2000
3000
4000
5000
6000

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.

THANK YOU

Unit-3

Bayesian learning –

Bayesian learning and the frequents method can also be considered as two ways of looking at the tasks of estimating values of unknown parameters given some observations caused by those parameters. For certain tasks, either the concept of uncertainty is meaningless or interpreting prior beliefs is too complex

Bayes theorem:

Bayes Theorem is a method to determine conditional probabilities – that is, the **probability** of one event occurring given that another event has already occurred. ... Thus, conditional probabilities are a must in determining accurate predictions and probabilities in **Machine Learning**

Bayes's formula

Below is Bayes's formula.

$P(A|B) = P(B|A)P(A)/P(B)\}$

The formula provides the relationship between P(A|B) and P(B|A). It is mainly derived from conditional probability formula discussed in the previous post.

Consider the below formulas for conditional probabilities P(A|B) and P(B|A)

$P(A|B) = P(A \cap B)/P(B)\}$ —-(1)

$P(B|A) = P(B \cap A)/P(A)\}$ —-(2)

Since $P(B \cap A) = P(A \cap B)$, we can replace $P(A \cap B)$ in the first formula with $P(B|A)P(A)$

After replacing, we get the given formula.

Product Rule
Product rule states that

$P(x \cap y)=p(x|y)*p(y)$(1)
So the joint probability that both X and Y will occur is equal to the product of two terms:

From the product rule :

implies **P(X|Y) = P(X)/P(Y)**

implies **P(X|Y) = 1**

Chain rule

When the above product rule is generalized we lead to chain rule. Let there are E_1, E_2,....E_n. n events . Then, the joint probability is given by

$$P(\bigcap_{i=1,..,n} E_i) = P(E_n|\bigcap_{i=1,..,n-1} E_i) * P(\bigcap_{i=1,..,n-1} E_i) \quad (2)$$

Bayes' Theorem

From the product rule, $P(X \cap Y) = P(X|Y)P(Y)$ and $P(Y \cap X) = P(Y|X)P(X)$. As $P(X \cap Y)$ and $P(Y \cap X)$ are same .

$$P(Y|X) = \frac{P(X|Y) * P(Y)}{P(X)} \quad (3)$$

where $P(X) = P(X \cap Y) + P(X \cap Y^c)$.

Example : Box P has 2 red balls and 3 blue balls and box Q has 3 red balls and 1 blue ball. A ball is selected as follows:

(i) Select a box

(ii) Choose a ball from the selected box such that each ball in

the box is equally likely to be chosen. The probabilities of

selecting boxes P and Q are (1/3) and (2/3), respectively.

Given that a ball selected in the above process is a red ball, the probability that it came from the box P is (GATE CS 2005)

(A) 4/19

(B) 5/19

(C) 2/9

(D) 19/30

Solution:

```
R --> Event that red ball is selected
B --> Event that blue ball is selected
P --> Event that box P is selected
Q --> Event that box Q is selected

We need to calculate P(P|R)?
```

$$P(P|R) = \frac{P(R|P)P(P)}{P(R)}$$

```
P(R|P) = A red ball selected from box P
       = 2/5
P(P) = 1/3
P(R) = P(P)*P(R|P) + P(Q)*P(R|Q)
     = (1/3)*(2/5) + (2/3)*(3/4)
     = 2/15 + 1/2
     = 19/30
```

Putting above values in the Bayes's Formula

P(P|R) = (2/5)*(1/3) / (19/30)

= 4/19

Practice question

Exercise A company buys 70% of its computers from company X and 30% from company Y. Company X produces 1 faulty computer per 5 computers and company Y produces 1 faulty computer per 20 computers. A computer is found faulty what is the probability that it was bought from company X?

Bayes' Theorem

Bayes' Theorem or Bayes' Rule is named after Reverend Thomas Bayes. It describes the probability of an event, based on prior knowledge of conditions that might be related to that event. It can also be considered for conditional probability examples.

For example: There are 3 bags, each containing some white marbles and some black marbles in each bag. If a white marble is drawn at random. With probability to find that this white marble is from the first bag. In cases like such, we use the Bayes' Theorem. It is used where the probability of occurrence of a particular event is calculated based on other conditions which are also called conditional probability. So before jumping into detail let's have a brief discussion on **The theorem of Total Probability**.

Theorem of Total Probability

Let $E_1, E_2, \ldots\ldots\ldots\ldots E_n$ is mutually exclusive and exhaustive events associated with a random experiment and lets E be an event that occurs with some E_i. Then, prove that

$$P(E) = {}^{n}\sum_{i=1} P(E/E_i) \,.\, P(E_j)$$

Proof:

$S = E_1 \cup E_2 \cup E_3 \cup \ldots\ldots\ldots\ldots\ldots\ldots \cup En$ and $E_i \cap E_j = \emptyset$ for $i \neq j$.

Therefore, $E = E \cap S = E \cap (E_1 \cup E_2 \cup E_3 \cup \ldots\ldots\ldots\ldots\ldots\ldots \cup E_n)$

$= (E \cap E_1) \cup (E \cap E_2) \cup \ldots\ldots \cup (E \cap E_n)$

$\Rightarrow P(E) = P\{(E \cap E_1) \cup (E \cap E_2) \cup \ldots\ldots \cup (E \cap E_n)\}$

$= P(E \cap E_1) + P(E \cap E_2) + \ldots\ldots + P(E \cap E_n)$

$= \{$Therefore, $(E \cap E_1), (E \cap E_2), \ldots\ldots\ldots\ldots, (E \cap E_n)\}$ are pairwise disjoint$\}$

$= P(E/E_1) \,.\, P(E_1) + P(E/E_2) \,.\, P(E_2) + \ldots\ldots\ldots\ldots\ldots\ldots + P(E/E_n) \,.\, P(E_n)$ [by multiplication theorem]

$= {}^{n}\sum_{i=1} P(E/E_i) \,.\, P(E_i)$

Examples

Example 1: A person has undertaken a job. The probabilities of completion of the job on time with and without rain are 0.44 and 0.95 respectively. If the probability that it will rain is 0.45, then determine the probability that the job will be completed on time?

Solution:

Let E1 be the event that the mining job will be completed on time and E2 be the event that it rai ns. We have,

P(A) = 0.45,

P(no rain) = P(B) = 1 − P(A) = 1 − 0.45 = 0.55

By multiplication law of probability,

P(E1) = 0.44

P(E2) = 0.95

Since, events A and B form partitions of the sample space S, by total probability theorem, we h ave

$$P(E) = P(A)\,P(E1) + P(B)\,P(E2)$$

= 0.45 × 0.44 + 0.55 × 0.95

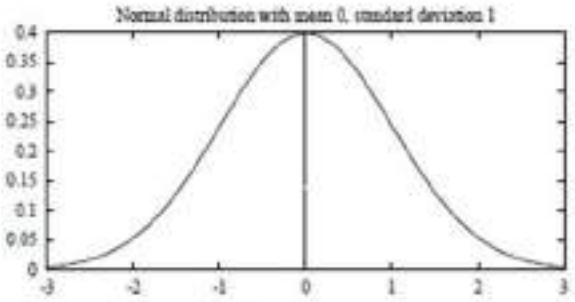
= 0.198 + 0.5225 = 0.7205

So, the probability that the job will be completed on time is 0.684.

Bayes Theorem & Concept Learning

- Bayes theorem is a principled way to calculate posterior probability of each hypothesis
- Assumptions: The training data is noise free $(\text{i.e., } d_i = c(x_i))$.The target concept is contained in . We have no a priori reason to believe that any hypothesis is more probable than any ot
- $P(h) = \frac{1}{|H|}$ for all h in H
- $$P(D|h) = \begin{cases} 1 & \text{if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 & \text{otherwise} \end{cases}$$
- $P(D) = \sum_{h_i \in H} P(D|h_i)P(h_i) = \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|} = \frac{|VS_{H,D}|}{|H|}$
- So if h is inconsistent with D, $P(h|D) = \frac{P(D|h)P(h)}{P(D)} = \frac{0 \cdot P(h)}{P(D)} = 0$
- So if h is consistent with D, $P(h|D) = \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} = \frac{1}{|VS_{H,D}|}$
- every consistent hypothesis is a MAP hypothesis

Basic Concepts from Probability Theory



A Normal Distribution (Gaussian Distribution) is a bell-shaped distribution defined by the ***probability density function***

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

- A Normal distribution is fully determined by two parameters in the formula: μ and σ .
- If the random variable X follows a normal distribution:
  - The probability that X will fall into the interval (a, b) is $\int_a^b p(x)dx$
  - The expected, or ***mean value of X***, $E[X] = \mu$
  - The ***variance of X***, $\mathrm{Var}(X) = \sigma^2$
  - The ***standard deviation of X***, $\sigma_x = \sigma$
- The ***Central Limit Theorem*** states that the sum of a large number of independent, identically distributed random variables follows a distribution that is approximately ***Normal***.

Maximum likelihood and least squared error hypotheses

Two commonly used approaches to estimate population parameters from a random sample are th e **maximum likelihood** estimation method (default) and the **least squares** estimation method. T he **likelihood** function indicates how likely the observed sample is as a function of possible para meter values.

Maximum Likelihood

The equation above looks simple but it is notoriously tricky to compute in practice — because of extremely large hypothesis space and complexity in evaluating integrals over complicated probability distribution functions.

However, in the quest of our search for the '*most probable hypothesis given data,*' we can simplify it further.

- We can drop the term in the denominator it does not have any term containing ***h*** i.e. hypothesis. We can imagine it as a normalizer to make total probability sum up to 1.
- **Uniform prior assumption** — this essentially relaxes any assumption on the nature of ***P*(*h*)** by making it uniform i.e. all hypotheses are probable. Then it is a constant number 1/|Vsd| where |Vsd| is the size of the **version space** i.e. a set of all hypothesis consistent with the training data. Then it does not actually figure in the determination of the maximally probable hypothesis.

After these two simplifying assumptions, the **maximum likelihood (ML)** hypothesis can be given by,

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

This simply means the most likely hypothesis is the one for which the conditional probability o f the observed data (given the hypothesis) reaches maximum.

Noise in the data

We generally start using least-square error while learning about simple linear regression back i n Stats 101 but this simple-looking loss function resides firmly inside pretty much every super vised machine learning algorithm viz. linear models, splines, decision trees, or deep learning n etworks.

So, what's special about it? Is it related to the Bayesian inference in any way?

It turns out that, the key connection between the least-square error and Bayesian inference is th rough the assumed nature of the error or residuals.

Measured/observed data is never error-free and there is always random noise associated with d ata, which can be thought of the signal of interest. Task of a machine learning algorithm is to e stimate/approximate the function which could have generated the data by separating the signal from the noise.

But what can we say about the nature of this noise? It turns out that noise can be modeled as a r andom variable. Therefore, we can associate a probability distribution of our choice to this ran dom variable.

One of the key assumptions of least-square optimization is that probability distribution over res iduals is our trusted old friend — Gaussian Normal.

This means that every data point (d) in a supervised learning training data set can be written as the sum of the unknown function f(x) (which the learning algorithm is trying to approximate) a nd an error term which is drawn from a Normal distribution of zero mean (μ) and unknown var iance σ². This is what I mean,

$$d_i = f(x_i) + e(i) = f(x_i) + \boldsymbol{N}(\mu = 0, \sigma^2)$$

And from this assertion, we can easily derive that the maximum likely hypothesis is the one which minimizes the least-square error.

Derivation of least-square from Maximum Likelihood hypothesis

We start with the **maximum likelihood hypothesis**:

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

We assume a fixed set of training instances $\langle x_1, x_2, \ldots, x_m \rangle$.
Therefore, we consider the data D to be the corresponding sequence of target values $D = \langle d_1, d_2, \ldots, d_n \rangle$
Here, $d_i = f(x_i) + e_i$ where e_i is the ***Normally distributed*** error.
Assuming the training examples are mutually independent given h, we can write $P(D|h)$ as the product of the various $p(d_i|h)$

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

Given that the noise e_i obeys a Normal distribution with zero mean(μ) and unknown variance σ^2, each d_i must also obey a Normal distribution with variance σ^2 centered around the true target value $f(x_i)$ rather than zero. Therefore $p(d_i|h)$ can be written as a Normal distribution with variance σ^2 and mean $\mu = f(x_i)$. Because we are writing the expression for the probability of d_i given that h is the correct description of the target function f, we will also substitute $\mu = f(x_i) = h(x_i)$, yielding,

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

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

We now apply **log-likelihood** transformation. This is justified because $ln(p)$ is a monotonic function of p. Therefore maximizing $ln(p)$ also maximizes p. So, the product becomes summation.

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

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

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

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

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

Finally, we can again discard constants that are independent of h.

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

MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

In the problem setting of the previous section we determined that the maximum likelihood hypothesis is the one that minimizes the sum of squared errors over the training examples.

• In this section we derive an analogous criterion for a second setting that is common in neural network learning: learning to predict probabilities.

• Consider the setting in which we wish to learn a nondeterministic (probabilistic) function f : X {0, 1}, which has two discrete output values.

• For example, the instance space X might represent medical patients in terms of their symptoms, and the target function f (x) might be 1 if the patient survives the disease and 0 if not.

• Alternatively, X might represent loan applicants in terms of their past credit history, and f (x) might be 1 if the applicant successfully repays their next loan and 0 if not. In both of these cases we might well expect f to be probabilistic.

• For example, among a collection of patients exhibiting the same set of observable symptoms, we might find that 92% survive, and 8% do not. This unpredictability could arise from our inability to observe all the important distinguishing features of the patients, or from some genuinely probabilistic mechanism in the evolution of the disease.

• Whatever the source of the problem, the effect is that we have a target function f (x) whose output is a probabilistic function of the input.

Example•

Given this problem setting, we might wish to learn a neural network (or other real-valued function approximator) whose output is the probability that f (x) = 1.

• In other words, we seek to learn the target function, f' : X [O, 1], such that f '(x) = P(f (x) = 1).

• In the above medical patient example, if x is one of those indistinguishable patients of which 92% survive, then f'(x) = 0. 92 whereas the probabilistic function f (x) will be equal to 1 in 92% of cases and equal to 0 in the remaining 8%.

• How can we learn f' using, say, a neural network? One obvious, bruteforce 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.

• As we shall see below, we can instead train a neural network directly from the observed training examples of f, yet still derive a maximum likelihood hypothesis for f '.

• What criterion should we optimize in order to find a maximum likelihood hypothesis for f' in this setting? To answer this question we must first obtain an expression for P(D|h).

• Let us assume the training data D is of the form D = {(x 1, d 1). . . (xm, dm)}, where di is the observed 0 or 1 value for f (xi).

• Recall that in the maximum likelihood, least-squared error analysis of the previous section, we made the simplifying assumption that the instances (x 1. . . xm) were fixed.

• This enabled us to characterize the data by considering only the target values di.

- Consider the setting in which we wish to learn a nondeterministic function f: $X \rightarrow \{0,1\}$
- We might expect f to be probabilistics
- For example, for neural network learning, we might output the f(x)=1 with probability 92%
  - $f': X \rightarrow [0,1]$, $f'=P(f(x)=1)$

v

- Brute-Force
  - Collecting d_i is the observed 0 or 1 for $f(x_i)$
- Assume training data D is of the form $D=\{<x_1,d_1>...<x_m,d_m>\}$

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

$$P(d_i \mid h, x_i) = \begin{cases} h(x_i) & d_i = 1 \\ 1 - h(x_i) & d_i = 0 \end{cases} = h(x_i)^{d_i} (1 - h(x_i))^{1-d_i}$$

$$P(D \mid h) = \prod_{i=1}^{m} h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} P(x_i)$$

$$\begin{aligned} h_{ML} &= \arg\max_{h \in H} \prod_{i=1}^{m} h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} p(x_i) \\ &= \arg\max_{h \in H} \prod_{i=1}^{m} h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} \\ &= \arg\max_{h \in H} \sum_{i=1}^{m} d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)) \end{aligned}$$

Minimum description length principle

Occam's razor: prefer the shortest hypothesis

MDL: prefer the hypothesis h that minimizes

$$h_{MDL} = \arg\min_{h \in H} L_{C1}(h) + L_{C2}(D \mid h)$$

where $L_C(x)$ is the description length of x under encoding C

Example:

- H = decision trees, D = training data labels
- $L_{C1}(h)$ is # bits to describe tree h
- $L_{C2}(D|h)$ is #bits to describe D given h
  - Note $L_{C2}(D|h) = 0$ if examples classified perfectly by h. Need only describe exceptions
- Hence h_{MDL} trades off tree size for training errors

$$\begin{aligned} h_{MAP} &= \arg\max_{h \in H} P(D \mid h) P(h) \\ &= \arg\max_{h \in H} \log_2 P(D \mid h) + \log_2 P(h) \\ &= \arg\min_{h \in H} -\log_2 P(D \mid h) - \log_2 P(h) \quad (1) \end{aligned}$$

Interesting fact from information theory:

The optimal (shortest expected length) code for an event with probability p is $\log_2 p$ bits.

So interpret (1):

$-\log_2 P(h)$ is the length of h under optimal code

$-\log_2 P(D|h)$ is length of D given h in optimal code

→ prefer the hypothesis that minimizes

length(h)+length(misclassifications)

Bayes optimal classifier

- Normally we consider:
  - What is the most probable ***hypothesis*** given the training data?
- We can also consider:
  - what is the most probable ***classification*** of the new instance given the training data?

- Consider a hypothesis space containing three hypotheses, hl, h2, and h3.
  - Suppose that the posterior probabilities of these hypotheses given the training data are .4, .3, and .3 respectively.
  - Thus, hl is the MAP hypothesis.
  - Suppose a new instance x is encountered, which is classified positive by *hl*, but negative by *h2* and h3.
  - Taking all hypotheses into account, the probability that x is positive is .4 (the probability associated with *h1)*, and the probability that it is negative is therefore .6.
  - The most probable classification (negative) in this case is different from the classification generated by the MAP hypothesis.

- The most probable classification of the new instance is obtained by combining the predictions of all hypotheses, weighted by their posterior probabilities.
- If the possible classification of the new example can take on any value v_j from some set V, then the probability $P(v_j \mid D)$ that the correct classification for the new instance is v_j :

$$P(v_j|D) = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

- **Bayes optimal classification:**

$$\underset{v_j \in V}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

Example:

$$P(h_1|D) = .4,\ P(\ominus|h_1) = 0,\ P(\oplus|h_1) = 1$$
$$P(h_2|D) = .3,\ P(\ominus|h_2) = 1,\ P(\oplus|h_2) = 0$$
$$P(h_3|D) = .3,\ P(\ominus|h_3) = 1,\ P(\oplus|h_3) = 0$$

Probabilities:

$$\sum_{h_i \in H} P(\oplus|h_i)P(h_i|D) = .4$$

$$\sum_{h_i \in H} P(\ominus|h_i)P(h_i|D) = .6$$

Result:

$$\underset{v_j \in \{\oplus, \ominus\}}{\text{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) = \ominus$$

- Although the Bayes optimal classifier obtains the best performance that can be achieved from the given training data, it can be quite costly to apply.
  - The expense is due to the fact that it computes the posterior probability for every hypothesis in H and then combines the predictions of each hypothesis to classify each new instance.

- An alternative, less optimal method is the Gibbs algorithm:

1. Choose a hypothesis ***h*** from H at random, according to the posterior probability distribution over H.
2. Use ***h*** to predict the classification of the next instance ***x***.

Gibbs Algorithm

•• Bayes Optimal is quite costly to apply. It computes the posterior probabilities for every hypothesis in H and combines the predictions of each hypothesis to classify each new instance

- An alternative (less optimal) method:

1. Choose a hypothesis h from H at random, according to the posterior probability distribution over H.
2. Use h to predict the classification of the next instance x.

- Under certain conditions the expected misclassification error for Gibbs algorithm is at most twice the expected error of the Bayes optimal classifier.

- Bayes Optimal is quite costly to apply. It computes the posterior probabilities for every hypothesis in H and combines the predictions of each hypothesis to classify each new instance

- An alternative (less optimal) method:

1. Choose a hypothesis h from H at random, according to the posterior probability distribution over H.
2. Use h to predict the classification of the next instance x.

- Under certain conditions the expected misclassification error for Gibbs algorithm is at most twice the expected error of the Bayes optimal classifier.

Naive Bayes Classifiers

A classifier is a machine learning model that is used to discriminate different objects based on certain features.

A Naive Bayes classifier is a probabilistic machine learning model that's used for classification task. The crux of the classifier is based on the Bayes theorem.

Naive Bayes classifiers are a collection of classification algorithms based on **Bayes' Theorem**. It is not a single algorithm but a family of algorithms where all of them share a common principle, i.e. every pair of features being classified is independent of each other.

Bayes Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Using Bayes theorem, we can find the probability of **A** happening, given that **B** has occurred. Here, **B** is the evidence and **A** is the hypothesis. The assumption made here is that the predictors/features are independent. That is presence of one particular feature does not affect the other. Hence it is called naive.

Example:

Let us take an example to get some better intuition. Consider the problem of playing golf. The dataset is represented as below.

Consider a fictional dataset that describes the weather conditions for playing a game of golf. Given the weather conditions, each tuple classifies the conditions as fit("Yes") or unfit("No") for plaing golf.

Here is a tabular representation of our dataset.

	OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY GOLF
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No

We classify whether the day is suitable for playing golf, given the features of the day. The columns represent these features and the rows represent individual entries. If we take the first row of the dataset, we can observe that is not suitable for playing golf if the outlook is rainy, temperature is hot, humidity is high and it is not windy. We make two assumptions here, one as stated above we consider that these predictors are independent. That is, if the temperature is hot, it does not necessarily mean that the humidity is high. Another assumption made here is that all the predictors have an equal effect on the outcome. That is, the day being windy does not have more importance in deciding to play golf or not.

According to this example, Bayes theorem can be rewritten as:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

The variable **y** is the class variable(play golf), which represents if it is suitable to play golf or not given the conditions. Variable **X** represent the parameters/features.

X is given as,

$$X = (x_1, x_2, x_3, ..., x_n)$$

Here x_1,x_2….x_n represent the features, i.e they can be mapped to outlook, temperature, humidity and windy. By substituting for **X** and expanding using the chain rule we get

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)}$$

Now, you can obtain the values for each by looking at the dataset and substitute them into the equation. For all entries in the dataset, the denominator does not change, it remain static. Therefore, the denominator can be removed and a proportionality can be introduced.

$$P(y|x_1, ..., x_n) \propto P(y) \prod_{i=1}^{n} P(x_i|y)$$

In our case, the class variable(**y**) has only two outcomes, yes or no. There could be cases where the classification could be multivariate. Therefore, we need to find the class **y** with maximum probability.

$$y = argmax_y P(y) \prod_{i=1}^{n} P(x_i|y)$$

Types of Naive Bayes Classifier:

Multinomial Naive Bayes:

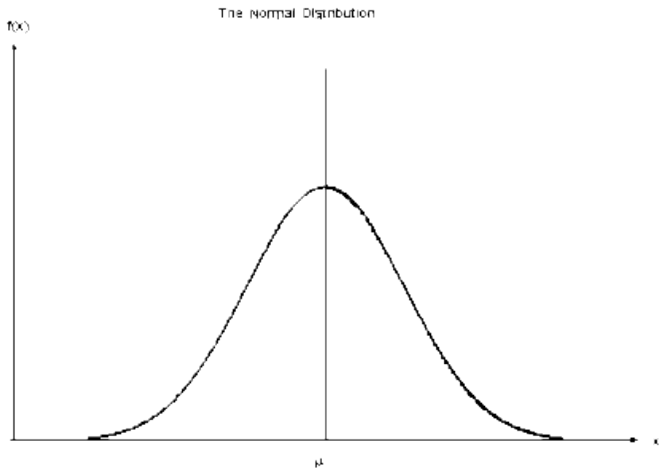
This is mostly used for document classification problem, i.e whether a document belongs to the category of sports, politics, technology etc. The features/predictors used by the classifier are the frequency of the words present in the document.

Bernoulli Naive Bayes:

This is similar to the multinomial naive bayes but the predictors are boolean variables. The parameters that we use to predict the class variable take up only values yes or no, for example if a word occurs in the text or not.

Gaussian Naive Bayes:

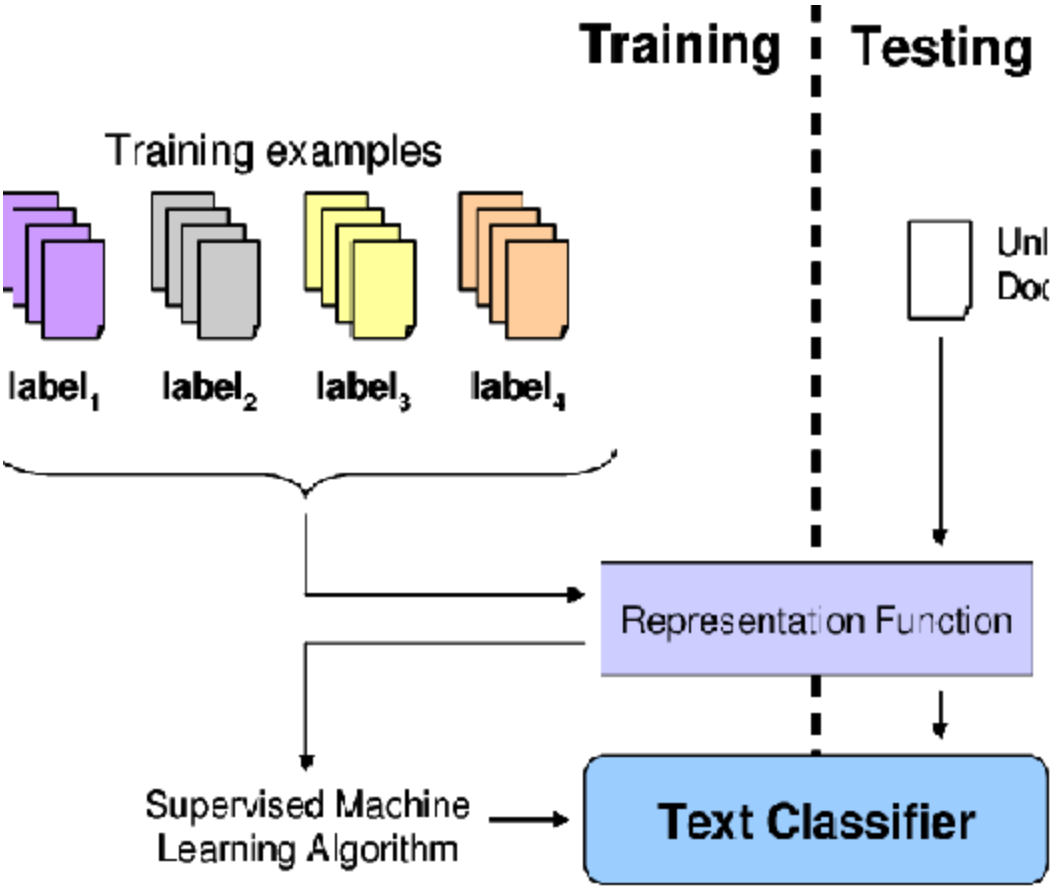
When the predictors take up a continuous value and are not discrete, we assume that these values are sampled from a gaussian distribution.



Since the way the values are present in the dataset changes, the formula for conditional probability changes to,

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$$

An example learning to classify text



Bayesian belief networks

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network, belief network, decision network**, or **Bayesian model**.

Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

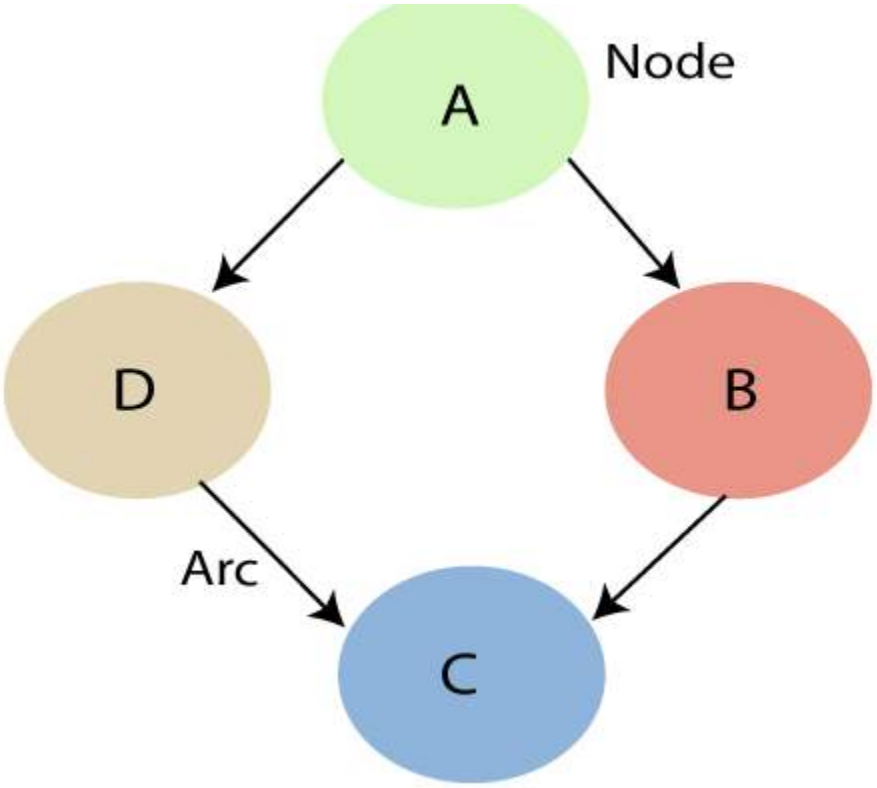
Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction**, and **decision making under uncertainty**.

Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

- **Directed Acyclic Graph**
- **Table of conditional probabilities.**

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an **Influence diagram**.

A Bayesian network graph is made up of nodes and Arcs (directed links), where:



- Each **node** corresponds to the random variables, and a variable can be **continuous** or **discrete**.
- **Arc or directed arrows** represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph.

  These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other

- **In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.**
- **If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.**
- **Node C is independent of node A.**

The Bayesian network has mainly two components:

- **Causal Component**
- **Actual numbers**

Each node in the Bayesian network has condition probability distribution $\mathbf{P(X_i | Parent(X_i))}$, which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables x1, x2, x3,....., xn, then the probabilities of a different combination of x1, x2, x3.. xn, are known as Joint probability distribution.

$\mathbf{P[x_1, x_2, x_3, ..., x_n]}$, it can be written as the following way in terms of the joint probability distribution.

$\mathbf{= P[x_1 | x_2, x_3, ..., x_n] P[x_2, x_3, ..., x_n]}$

$\mathbf{= P[x_1 | x_2, x_3, ..., x_n] P[x_2 | x_3, ..., x_n] .... P[x_{n-1} | x_n] P[x_n].}$

In general for each variable Xi, we can write the equation as:

$$P(X_i | X_{i-1}, ..., X_1) = P(X_i | Parents(X_i))$$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.
- The conditional distributions for each node are given as conditional probabilities table or CPT.
- Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
- In CPT, a boolean variable with k boolean parents contains 2^K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

- **Burglary (B)**
- **Earthquake(E)**
- **Alarm(A)**
- **David Calls(D)**
- **Sophia calls(S)**

We can write the events of problem statement in the form of probability: **P[D, S, A, B, E]**, can rewrite the above probability statement using joint probability distribution:

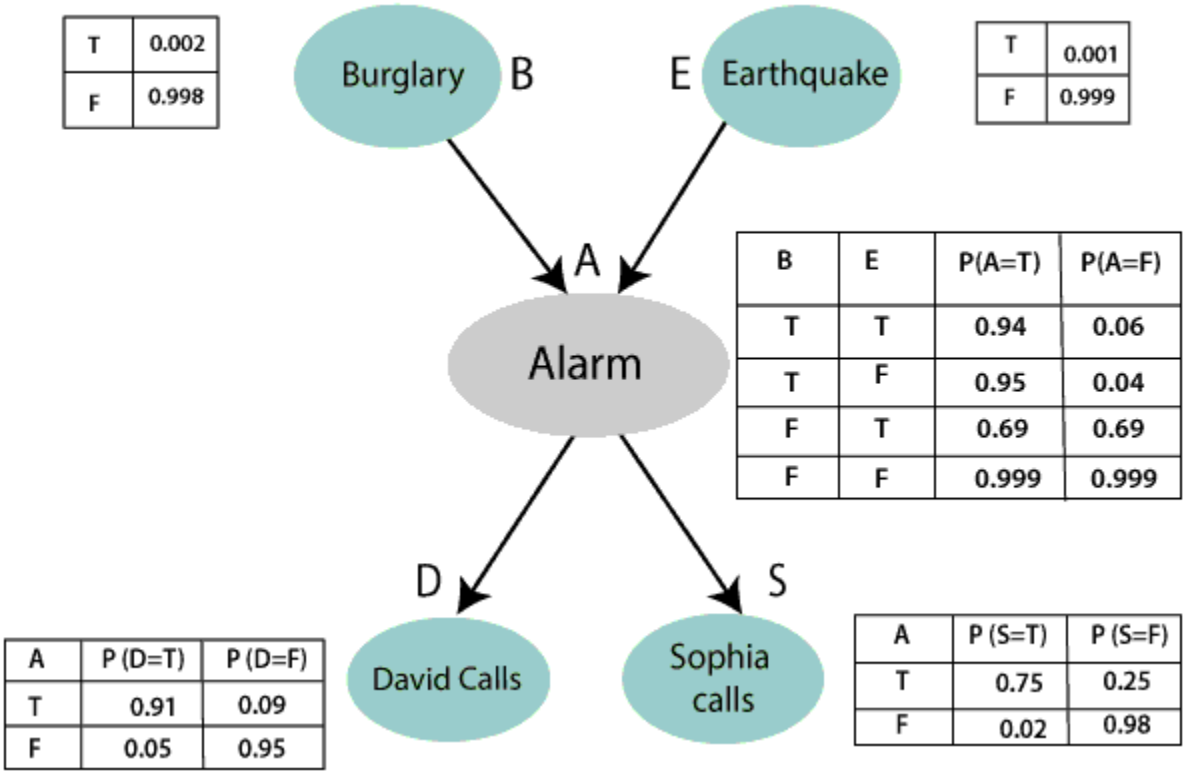
P[D, S, A, B, E]= P[D | S, A, B, E]. P[S, A, B, E]

=P[D | S, A, B, E]. P[S | A, B, E]. P[A, B, E]

= P [D| A]. P [S| A, B, E]. P[A, B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B |E]. P[E]



T	0.002
F	0.998

T	0.001
F	0.999

B	E	P(A=T)	P(A=F)
T	T	0.94	0.06
T	F	0.95	0.04
F	T	0.69	0.69
F	F	0.999	0.999

A	P (D=T)	P (D=F)
T	0.91	0.09
F	0.05	0.95

A	P (S=T)	P (S=F)
T	0.75	0.25
F	0.02	0.98

MODULE 5

INSTANCE BASED LEARNING

INTRODUCTION

- Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions.
- Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the new query instance
- Instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified

Advantages of Instance-based learning

1. Training is very fast
2. Learn complex target function
3. Don’t lose information

Disadvantages of Instance-based learning

- The cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered.
- In many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider all attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

k- NEAREST NEIGHBOR LEARNING

- The most basic instance-based method is the K- Nearest Neighbor Learning. This algorithm assumes all instances correspond to points in the n-dimensional space R^n.
- The nearest neighbors of an instance are defined in terms of the standard Euclidean distance.
- Let an arbitrary instance x be described by the feature vector

  $$((a_1(x), a_2(x), \ldots\ldots\ldots, a_n(x))$$

  Where, $a_r(x)$ denotes the value of the r^{th} attribute of instance x.

- Then the distance between two instances x_i and x_j is defined to be $d(x_i , x_j)$
  Where,

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

- In nearest-neighbor learning the target function may be either discrete-valued or real-valued.

Let us first consider learning ***discrete-valued target functions*** of the form

$$f : \Re^n \rightarrow V.$$

Where, V is the finite set $\{v_1, \ldots v_s\}$

The k- Nearest Neighbor algorithm for approximation a **discrete-valued target function** is given below:

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

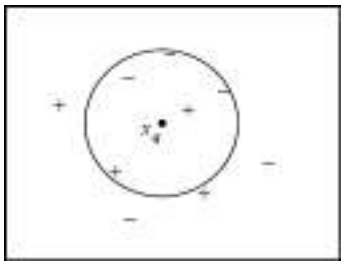
- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

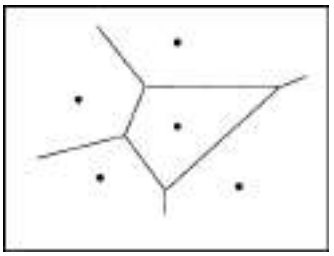
- The value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of f among the k training examples nearest to x_q.
- If k = 1, then the 1- Nearest Neighbor algorithm assigns to $\hat{f}(x_q)$ the value f(xi). Where x_i is the training instance nearest to x_q.
- For larger values of k, the algorithm assigns the most common value among the k nearest training examples.

- Below figure illustrates the operation of the k-Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is Boolean valued.



  - The positive and negative training examples are shown by "+" and "-" respectively. A query point x_q is shown as well.
  - The 1-Nearest Neighbor algorithm classifies x_q as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.

- Below figure shows the shape of this **decision surface** induced by 1- Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples.



  - For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the ***Voronoi diagram*** of the set of training example

The K- Nearest Neighbor algorithm for approximation a **real-valued target function** is given below $f : \Re^n \rightarrow \Re$

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Distance-Weighted Nearest Neighbor Algorithm

- The refinement to the k-NEAREST NEIGHBOR Algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors.
- For example, in the k-Nearest Neighbor algorithm, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q

Distance-Weighted Nearest Neighbor Algorithm for approximation a discrete-valued target functions

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

Distance-Weighted Nearest Neighbor Algorithm for approximation a Real-valued target functions

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

Terminology

- **Regression** means approximating a real-valued target function.
- **Residual** is the error $\hat{f}(x) - f(x)$ in approximating the target function.
- **Kernel function** is the function of distance that is used to determine the weight of each training example. In other words, the kernel function is the function K such that $w_i = K(d(x_i, x_q))$

LOCALLY WEIGHTED REGRESSION

- The phrase "**locally weighted regression**" is called **local** because the function is approximated based only on data near the query point, **weighted** because the contribution of each training example is weighted by its distance from the query point, and **regression** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

- Given a new query instance x_q, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding x_q. This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance.

Locally Weighted Linear Regression

- Consider locally weighted regression in which the target function ***f*** is approximated near x_q using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

Where, $a_i(x)$ denotes the value of the i^{th} attribute of the instance x

- Derived methods are used to choose weights that minimize the squared error summed over the set D of training examples using gradient descent

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2$$

Which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D} (f(x) - \hat{f}(x)) a_j(x)$$

Where, η is a constant learning rate

- Need to modify this procedure to derive a local approximation rather than a global one. The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below.

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \qquad \text{equ(1)}$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from x_q :

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \text{equ(2)}$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \text{equ(3)}$$

If we choose criterion three and re-derive the gradient descent rule, we obtain the following training rule

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x))\ (f(x) - \hat{f}(x))\ a_j(x)$$

The differences between this new rule and the rule given by Equation (3) are that the contribution of instance x to the weight update is now multiplied by the distance penalty **$K(d(x_q, x))$,** and that the error is summed over only the k nearest training examples.

RADIAL BASIS FUNCTIONS

- One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions
- In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad \text{equ (1)}$$

- Where, each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance d(xu, x) increases.
- Here k is a user provided constant that specifies the number of kernel functions to be included.
- $\hat{f}$ is a global approximation to f (x), the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point x_u.

Choose each function $K_u(d(x_u, x))$ to be a Gaussian function centred at the point x_u with some variance σ_u^2

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

- The functional form of equ(1) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width σ^2 of each kernel can be separately specified
- The function given by equ(1) can be viewed as describing a two layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values

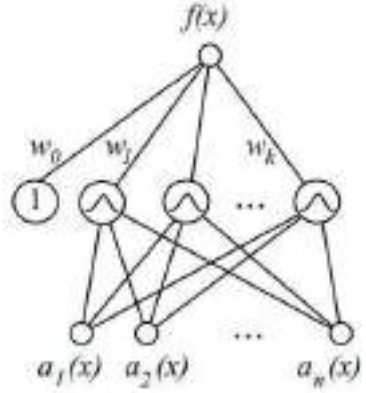
Example: Radial basis function (RBF) network

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process.

1. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 that define its kernel function $K_u(d(x_u, x))$
2. Second, the weights w, are trained to maximize the fit of the network to the training data, using the global error criterion given by

$$E \equiv \frac{1}{2}\sum_{x \in D}(f(x) - \hat{f}(x))^2$$

Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently



Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions.

- One approach is to allocate a Gaussian kernel function for each training example $(x_i, f(x_i))$, centring this Gaussian at the point x_i.
  Each of these kernels may be assigned the same width σ^2. Given this approach, the RBF network learns a global approximation to the target function in which each training example $(x_i, f(x_i))$ can influence the value of ***f*** only in the neighbourhood of x_i.
- A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large.

Summary

- Radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions.
- The value for any given kernel function is non-negligible only when the input x falls into the region defined by its particular centre and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function.
- One key advantage to RBF networks is that they can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION.

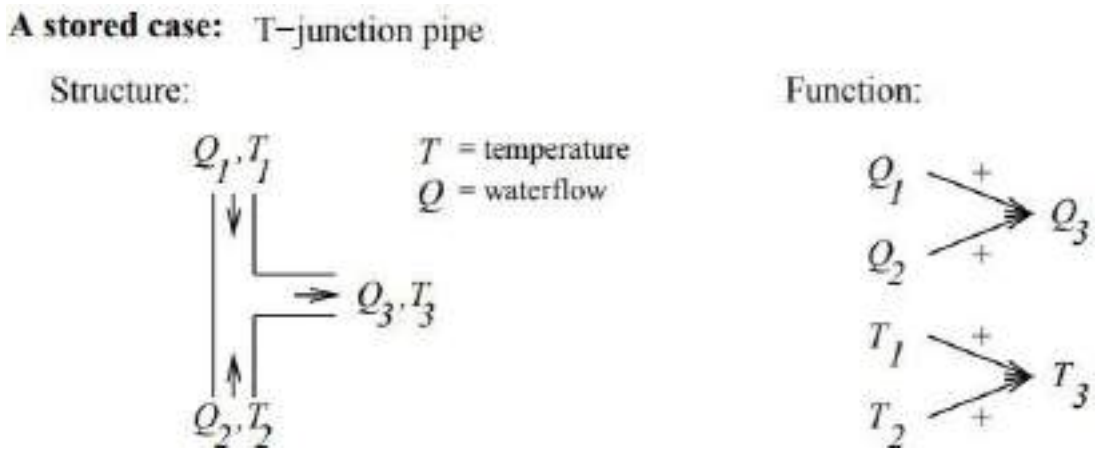
CASE-BASED REASONING

- Case-based reasoning (CBR) is a learning paradigm based on lazy learning methods and they classify new query instances by analysing similar instances while ignoring instances that are very different from the query.
- In CBR represent instances are not represented as real-valued points, but instead, they use a ***rich symbolic*** representation.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs, reasoning about new legal cases based on previous rulings, and solving planning and scheduling problems by reusing and combining portions of previous solutions to similar problems

A prototypical example of a case-based reasoning

- The CADET system employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets.
- It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems.
- Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

The problem setting is illustrated in below figure



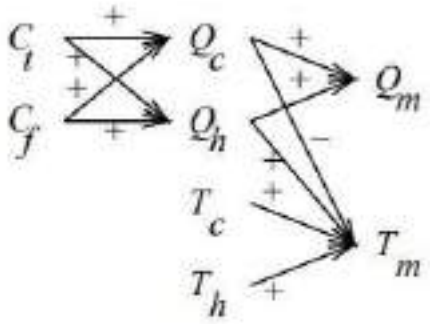
- The function is represented in terms of the qualitative relationships among the water-flow levels and temperatures at its inputs and outputs.
- In the functional description, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. A "-" label indicates that the variable at the head decreases with the variable at the tail.
- Here Q_c refers to the flow of cold water into the faucet, Q_h to the input flow of hot water, and Q_m to the single mixed flow out of the faucet.
- T_c, T_h, and T_m refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable C_t denotes the control signal for temperature that is input to the faucet, and C_f denotes the control signal for waterflow.
- The controls C_t and C_f are to influence the water flows Q_c and Q_h, thereby indirectly influencing the faucet output flow Q_m and temperature T_m.

A problem specification: Water faucet

Structure:

?

Function:



- CADET searches its library for stored cases whose functional descriptions match the design problem. If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem. If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification.

REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.

INTRODUCTION

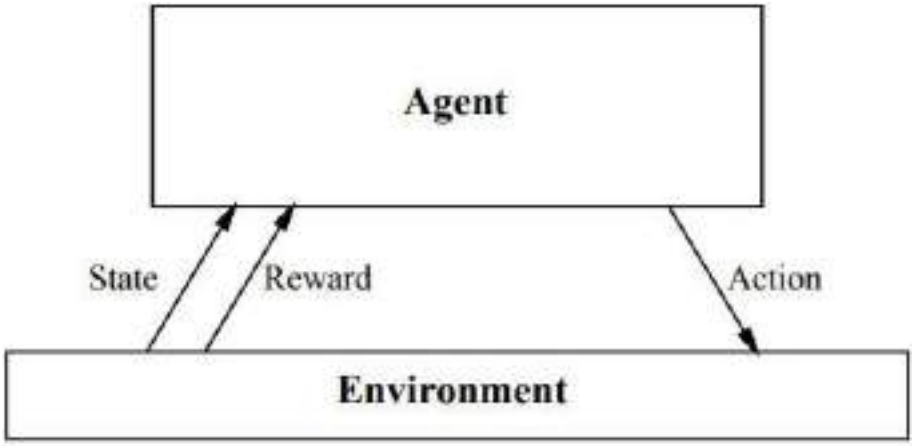
- Consider building a **learning robot**. The robot, or ***agent***, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.
- Its task is to learn a control strategy, or ***policy***, for choosing actions that achieve its goals.
- The goals of the agent can be defined by a ***reward function*** that assigns a numerical value to each distinct action the agent may take from each distinct state.
- This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot.
- The **task** of the robot is to perform sequences of actions, observe their consequences, and learn a control policy.
- The control policy is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

Example:

- A mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn."
- The robot may have a goal of docking onto its battery charger whenever its battery level is low.
- The goal of docking to the battery charger can be captured by assigning a positive reward (Eg., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

Reinforcement Learning Problem

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.
- Agent perform any of a set of possible actions A. Each time it performs an action a, in some state s_t the agent receives a real-valued reward r, that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure.
- The agent's task is to learn a control policy, □**: S** → **A**, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \text{ , where } 0 \leq \gamma < 1$$

Reinforcement learning problem characteristics

1. **Delayed reward**: The task of the agent is to learn a target function □ that maps from the current state s to the optimal action a = □ (s). In reinforcement learning, training information is not available in (s, □ (s)). Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of ***temporal credit assignment***: determining which of the actions in its sequence are to be credited with producing the eventual rewards.

2. **Exploration:** In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a trade-off in choosing whether to favor exploration of unknown states and actions, or exploitation of states and actions that it has already learned will yield high reward.

3. **Partially observable states:** The agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. In such cases, the agent needs to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

4. **Life-long learning:** Robot requires to learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

THE LEARNING TASK

- Consider Markov decision process (MDP) where the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it.
- The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.

The task of the agent is to learn a policy, $\pi: S \rightarrow A$, for selecting its next action a, based on the current observed state s_t; that is, $\pi(s_t) = a_t$.

How shall we specify precisely which policy π we would like the agent to learn?

1. One approach is to require the policy that produces the greatest possible ***cumulative reward*** for the robot over time.

- To state this requirement more precisely, define the cumulative value $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad \text{equ (1)}$$

- Where, the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions.
- Here $0 \leq \gamma \leq 1$ is a constant that determines the relative value of delayed versus immediate rewards. if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.
- The quantity $V^{\pi}(s_t)$ is called the ***discounted cumulative reward*** achieved by policy π from initial state ***s***. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later.

2. Other definitions of total reward is ***finite horizon reward,***

$$\sum_{i=0}^{h} r_{t+i}$$

Considers the undiscounted sum of rewards over a finite number ***h*** of steps

3. Another approach is ***average reward***

$$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

Considers the average reward per time step over the entire lifetime of the agent.

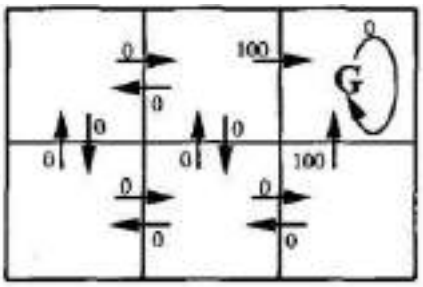
We require that the agent learn a policy π that maximizes $V^{\pi}(s_t)$ for all states s. such a policy is called an ***optimal policy*** and denote it by π*

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \qquad \textbf{equ (2)}$$

Refer the value function $V^{\pi*}(s)$ an optimal policy as V*(s). V*(s) gives the maximum discounted cumulative reward that the agent can obtain starting from state ***s***.

Example:

A simple grid-world environment is depicted in the diagram

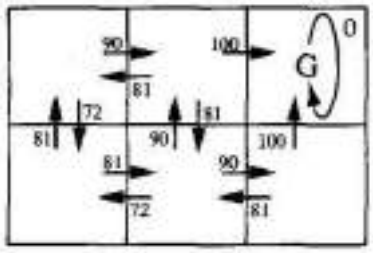


$r(s, a)$ (immediate reward) values

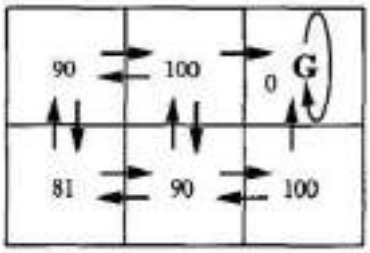
- The six grid squares in this diagram represent six possible states, or locations, for the agent.
- Each arrow in the diagram represents a possible action the agent can take to move from one state to another.
- The number associated with each arrow represents the immediate reward ***r(s, a)*** the agent receives if it executes the corresponding state-action transition
- The immediate reward in this environment is defined to be zero for all state-action transitions except for those leading into the state labelled G. The state G as the goal state, and the agent can receive reward by entering this state.

Once the states, actions, and immediate rewards are defined, choose a value for the discount factor γ, determine the optimal policy π * and its value function V*(s).

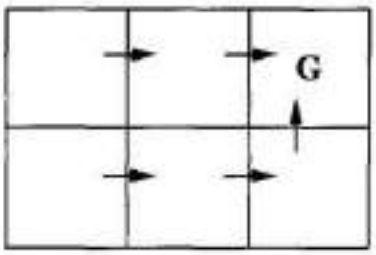
Let's choose γ = 0.9. The diagram at the bottom of the figure shows one optimal policy for this setting.



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

Values of V*(s) and Q(s, a) follow from r(s, a), and the discount factor γ = 0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.

The discounted future reward from the bottom centre state is

$$0+\gamma\ 100+\gamma^2\ 0+\gamma^3\ 0+\ldots = 90$$

Q LEARNING

How can an agent learn an optimal policy π * for an arbitrary environment?

The training information available to the learner is the sequence of immediate rewards $r(s_i,a_i)$ for i = 0, 1,2, ... Given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn?

One obvious choice is V*. The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1

The optimal action in state ***s*** is the action ***a*** that maximizes the sum of the immediate reward r(s, a) plus the value V* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s, a) + \gamma V^*(\delta(s, a))] \qquad \text{equ (3)}$$

The *Q* Function

The value of Evaluation function ***Q(s, a)*** is the reward received immediately upon executing action a from state ***s***, plus the value (discounted by γ) of following the optimal policy thereafter

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \qquad \text{equ (4)}$$

Rewrite Equation (3) in terms of ***Q(s, a)*** as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \qquad \text{equ (5)}$$

Equation (5) makes clear, it need only consider each available action ***a*** in its current state ***s*** and choose the action that maximizes ***Q(s, a)***.

An Algorithm for Learning *Q*

- Learning the Q function corresponds to learning the **optimal policy**.
- The key problem is finding a reliable way to estimate training values for ***Q***, given only a sequence of immediate rewards ***r*** spread out over time. This can be accomplished through ***iterative approximation***

$$V^*(s) = \max_{a'} Q(s,a')$$

Rewriting Equation

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a')$$

- **Q learning algorithm:**

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.

Observe the current state s

Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

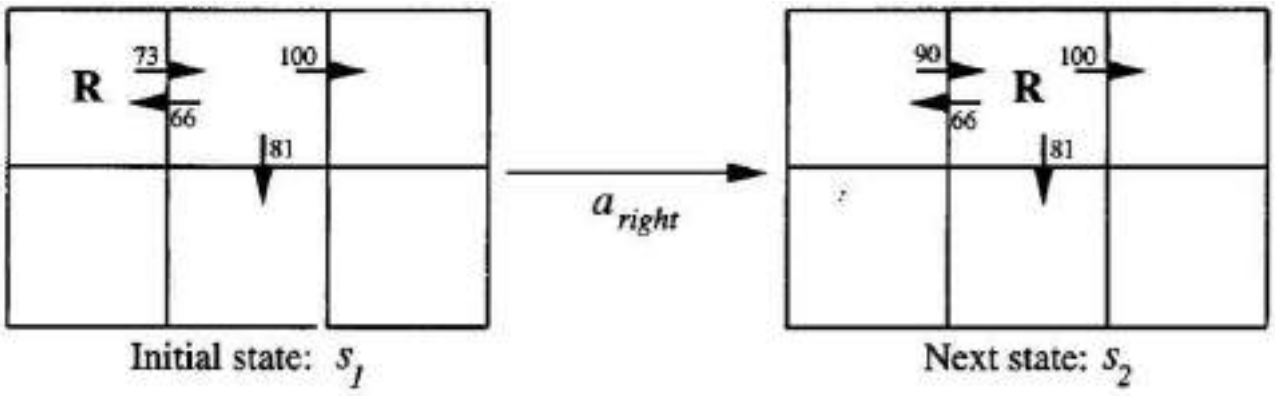
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

- Q learning algorithm assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$
- $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function

An Illustrative Example

- To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to $\hat{Q}$ shown in below figure



- The agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition.
- Apply the training rule of Equation

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

to refine its estimate Q for the state-action transition it just executed.

- According to the training rule, the new $\hat{Q}$ estimate for this transition is the sum of the received reward (zero) and the highest $\hat{Q}$ value associated with the resulting state (100), discounted by γ (.9).

$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{66, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

Convergence

Will the Q Learning Algorithm converge toward a Q equal to the true Q function?

Yes, under certain conditions.

1. Assume the system is a deterministic MDP.
2. Assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, **| r(s, a)| < c**
3. Assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

Theorem Convergence of Q learning for deterministic Markov decision processes.

Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$.

The Q learning agent uses the training rule of Equation $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$ initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the nth update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all s, a.

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use s' to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned} |\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\ &\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \end{aligned}$$

$$|\hat{Q}_{n+1}(s, a) - Q(s, a)| \leq \gamma \Delta_n$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n.

Thus, the updated $Q_{n+1}(s, a)$ for any s, a is at most γ times the maximum error in the $\hat{Q}_n$ table, Δ_n. The largest error in the initial table, Δ_0, is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all s, a. Now after the first interval during which each s, a is visited, the largest error in the table will be at most $\gamma \Delta_0$. After k such intervals, the error will be at most $\gamma^k \Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \to 0$ as $n \to \infty$. This proves the theorem.

Experimentation Strategies

The Q learning algorithm does not specify how actions are chosen by the agent.

- One obvious strategy would be for the agent in state ***s*** to select the action ***a*** that maximizes $\hat{Q}(s, a)$, thereby exploiting its current approximation $\hat{Q}$.
- However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high Q values, while failing to explore other actions that have even higher values.
- For this reason, Q learning uses a probabilistic approach to selecting actions. Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability.
- One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Where, $\mathbf{P(a_i|s)}$ is the probability of selecting action $\mathbf{a_i}$, given that the agent is in state **s**, and $\mathbf{k > 0}$ is a constant that determines how strongly the selection favors actions with high $\hat{Q}$ values

UNIT-4

We can learn sets of rules by using **ID3 and then converting the tree to rules**. We can also use a genetic algorithm that encodes the rules as bit strings. But, these only work with predicate rules (no variables). They also consider the set of rules as a whole, not one rule at a time.

Learning Sets of Rules are

1. Sequential Covering Algorithms
2. Learning First Order Rules
3. FOIL
4. Induction as Inverted Deduction
5. Inductive Logic Programming

Sequential Covering Algorithms

Sequential Covering is a popular algorithm based on Rule-Based Classification used for learning a disjunctive set of rules. The basic idea here is to learn one rule, remove the data that it covers, then repeat the same process. In this process, In this way, it covers all the rules involved with it in a sequential manner during the training phase.

Algorithm Involved:

```
Sequential_covering (Target_attribute, Attributes, Examples, Threshold):
      Learned_rules = {}
      Rule = Learn-One-Rule(Target_attribute, Attributes, Examples)

      while Performance(Rule, Examples) > Threshold :
            Learned_rules = Learned_rules + Rule
            Examples = Examples - {examples correctly classified by Rule}
            Rule = Learn-One-Rule(Target_attribute, Attributes, Examples)

      Learned_rules = sort Learned_rules according to performance over Examples
```

```
return Learned_rules
```

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

Working on the Algorithm:

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

Step 2 – 'Learn-One-Rule' Algorithm

It extracts the best rule for a particular class 'y', where a rule is defined as: (Fig.2)

General Form of Rule

$$r_i : (\text{condition}_1, \ldots, \text{condition}_i) \rightarrow y_i$$

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

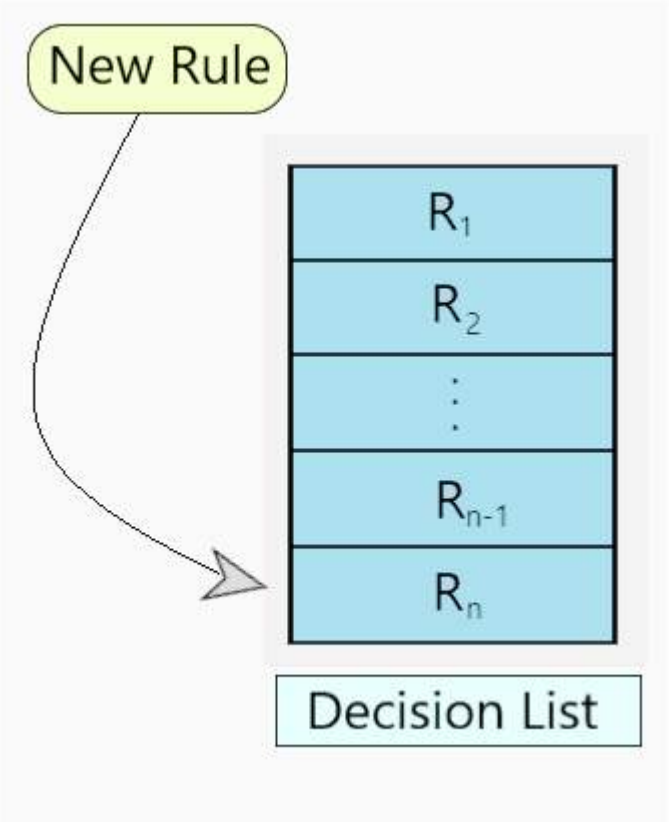
Step 2.b – else if all training examples ∉ class 'y', then it's classified as negative example.

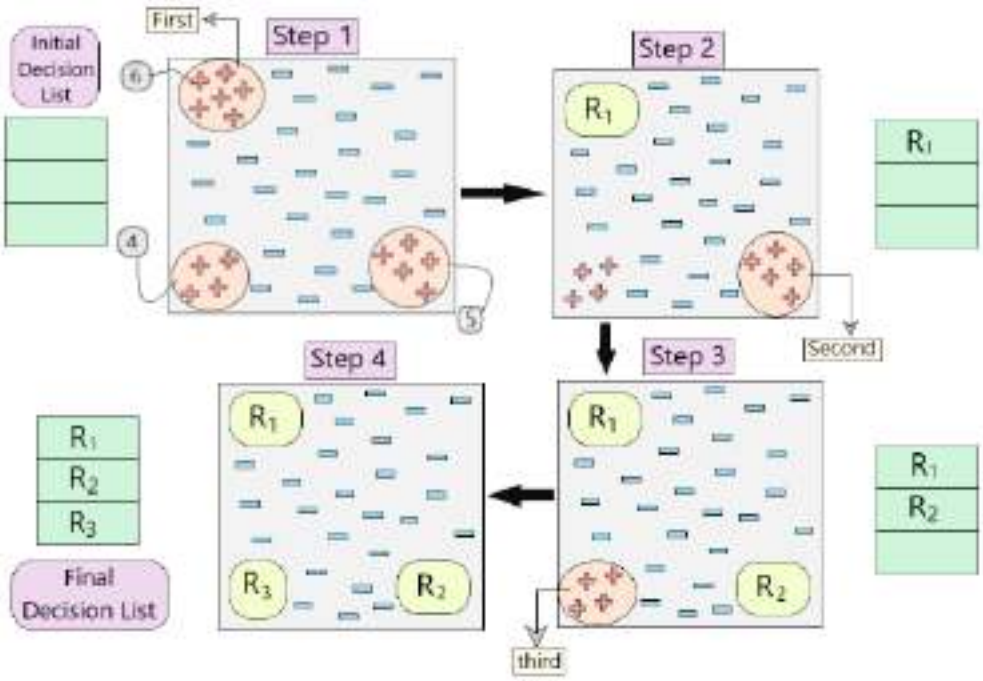
Step 3 – The rule becomes 'desirable' when it covers a majority of the positive examples.

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

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

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





- Let us understand step by step how the algorithm is working in the example shown in Fig.4.
- First, we created an empty decision list. During Step 1, we see that there are three sets of positive examples present in the dataset. So, as per the algorithm, we consider the one with maximum no of positive example. *(6, as shown in Step 1 of Fig 4)*
- Once we cover these 6 positive examples, we get our first rule R_1, which is then pushed into the decision list and those positive examples are removed from the dataset. *(as shown in Step 2 of Fig 4)*
- Now, we take the next majority of positive examples *(5, as shown in Step 2 of Fig 4)* and follow the same process until we get rule R_2. (Same for R_3)
- In the end, we obtain our final decision list with all the desirable rules.

Learning Rule Sets Summary

- A **sequential covering** algorithm learns one rule at a time, removing the covered examples and repeating with the rest.
- Meanwhile, **simultaneous covering** algorithms like ID3 learn the entire set of disjuncts simultaneously.
- Which is better?
- ID3 chooses attributes by comparing the partitions of the data they generate.
- CN2 chooses among alternative attribute-value pairs by comparing the subsets of data they cover.
- Thus, CN2 makes a larger number of independent choices. So it is better if there is plenty of data.

- Learn-One-Rule searches from general to specific.
- Find-S searches from specific to general.
- There are many maximally specific, but only one maximally general.
- Learn-One-Rule is generate-then-test search.
- It could be example-driven, where individual training examples constrain the generation of hypotheses.
- In generate-then-test each choice in the search is based on the hypothesis performance over many examples, so impact of noisy data is minimize. Noise can have large impact in example-driven.

Learning First Order Rules

1. Try to learn sets of rules such as

$$\forall\ x, y : \text{Ancestor}(x,y) \leftarrow \text{Parent}(x,y)$$
$$\forall\ x, y : \text{Ancestor}(x,y) \leftarrow \text{Parent}(x,z) \wedge \text{Ancestor}(z,y)$$

2. This looks a lot like a Prolog program.

```
Ancestor(x,y) :- Parent(x,y)
Ancestor(x,y) :- Parent(x,z), Ancestor(z,y)
```

3. Which is why inductive learning of first-order rules is often referred to as inductive logic programming.

4. Classifying web page A:

course(A) ← has-word(A, instructor) ∧ ¬has-word(A,good) ∧ link-from(A, B) ∧ has-word(B, problem) ∧ ¬link-from(B, C)

First-Order Logic Definitions

1. Every expression is composed of constants, variables, predicates, and functions.
2. A term is any constant, or variable, or any function applied to any term
3. A literal is any predicate (or its negation) applied to any set of terms.

   Female(Mary), ¬Female(x)
4. A ground literal is a literal that does not contain any variables.
5. A clause is any disjunction of literals whose variables are universally quantified.$\forall\ x :$ $\text{Female}(x) \vee \text{Male}(x)$
6. A Horn clause is an expression of the form

   $H \leftarrow L1 \wedge L2 \wedge \ldots \wedge Ln$
7. For any A and B

   $A \leftarrow B \Rightarrow A \vee \neg B$
8. so the horn clause above can be re-written as

   $H \wedge \neg L1 \vee \neg L2 \vee \ldots \vee \neg Ln$
9. A substitution is any function that replaces variables by terms e.g.,

{x/3, y/z} replaces x for 3 and y for z.

10. A unifying substitution θ for two literals L1 and L2 is one where

$$L1\theta = L2\theta$$

Learning First-Order Horn Clauses

- Say we are trying to learn the concept Daughter(x,y) from examples.
- Each person is described by the attributes: Name, Mother, Father, Male, Female.
- Each example is a pair of instances, say a and b:
- Name(a) = Sharon, Mother(a) = Louise, Father(a) = Bob, Male(a) = False, Female(a) = True
  Name(b) = Bob, Mother(b) = Nora, Father(b) = Victor, Male(b) = True, Female(b) = False, Daughter(a,b) = True
- If we give a bunch of these examples to CN2 or C4.5 they will output a set of rules like:
- **IF** Father(a) = Bob ∧ Name(b) = Bob ∧ Female(a) **THEN** Daughter(a,b)
- A first-order learner would output more general rules like
- **IF** Father(x) = y ∧ Female(x) **THEN** Daughter(x,y)

FOIL

```
FOIL(target-predicate, predicates, examples)
        pos = those examples for which target-predicate is true
        teg = those examples for which target-predicate is false
        learnedRules = {}
        while pos do
          ;learn a new rule
          newRule = the rule that predicts target-predicate with no preconditions
          newRuleNeg = neg
          while newRuleNeg do
            ;add a new literal to specialize newRule
            candidateLiterals = candidate new literals for newRule based on
    predicates
            bestLiteral = argmax l ∈ candidateLiterals Foil-gain(l, newRule)
            add bestLiteral to preconditions of newRule
            newRuleNeg = subset of newRuleNeg that satisfies newRuleNeg
    preconditions
            learnedRules = learnedRules + newRule
```

```
        pos = pos - {member of pos covered by newRule}
        return learnedRules
```

- Its a natural extension of Sequential-Covering and Learn-One-Rule.
- Each iteration of the outer loop adds a new rule to its disjunctive hypothesis Learned-rules (specific-to-general).
- In the inner loop we add conjunctions that form the preconditions of the rule (general-to-specific hill-climbing).

First-Order Logic:

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

1. constants — e.g. tyler, 23, a
2. variables — e.g. A, B, C
3. predicate symbols — e.g. male, father (True or False values only)
4. function symbols — e.g. age (can take on any constant as a value)
5. connectives — e.g. ∧ , ∨ , ¬, →, ←
6. quantifiers — e.g. ∀ , ∃

Generating Candidate Specializations

- How do we generate Candidate-literals?
- Suppose the current rule (NewRule) is P(x1,x2,...xk) ←L1∧ ...∧ Ln.
- FOIL considers new literals Ln+1 of the following form
    - Q(v1,...,vr) where Q is any predicate in Predicates and vi are variables, at least one of them must already exist in the rule.
    - Equal(x,y) where x and y are variables already in the rule.
    - The negation of any of these two.

FOIL Example

- Say we are tying to predict the Target-predicate GrandDaughter(x,y).
- FOIL begins with
  NewRule = GrandDaughter(x,y) ←
- To specialize it, generate these candidate additions to the preconditions:
  Equal(x,y), Female(x), Female(y), Father(x,y), Father(y.x), Father(x,z), Father(z,x), Father(y,z), Father(z,y)
  and their negations.
- FOIL might greedily select Father(x,y) as most promising, then
  NewRule = GrandDaughter(x,y) ← Father(y,z).

- Foil now considers all the literals from the previous step as well as: Female(z), Equal(z,x), Equal(z,y), Father(z,w), Father(w,z) and their negations.
- Foil might select Father(z,x), and on the next step Female(y) leading to
  NewRule = GrandDaughter (x,y) ← Father(y,z) ∧ Father(z,x) ∧ Female(y)
- If this covers only positive examples it terminates the search for further specialization.
- FOIL now removes all positive examples covered by this new rule. If more are left then the outer loop continues.

Search in FOIL

- Again, we are trying to learn set of rules for Target-predicate = GrandDaughter(x,y).
- Let the Examples contain
  GrandDaughter(Victor,Sharon), Father(Sharon, Bob), Father(Tom, Bob), Female(Sharon), father(Bob, Victor)
- To select best specialization, FOIL considers ways to bind the variables. With 4 constants (Victor, Sharon, Tom, and Bob) and 2 variables (x,y) we have 4*4 = 16 possible bindings.
- {x/Victor, y/Sharon} is the only **positive example binding** for the rule "GrandDaughter(x,y) ←". The other 15 are **negative bindings**.
- At each step, each rule is evaluated based on the sets of positive and negative variable bindings. Which one do we pick?

Foil-Gain

- We select the literal with biggest
  gain Foil-Gain (L,R)≡t(log 2 p 1 p 1 +n 1 -log 2 p 0 p 0 +n 0) Where
  - L is the candidate literal to add to rule R
  - p 0 = number of positive bindings of R
  - n 0 = number of negative bindings of R
  - p 1 = number of positive bindings of R +L
  - n 1 = number of negative bindings of R +L
  - t is the number of positive bindings of R also covered by R +L
- It's interesting to note that - log 2 p 0 p 0 +n 0 is the optimal number of bits needed to indicate the class of a positive binding covered by R

- FOIL extends CN2 to handle first-order formulas.

- It does a general-to-specific search search, adding a single new literal to the preconditions at each step.
- The Foil-Gain function is used to select the best literal.
- It can learn recursive rules.
- If data is noisy, the search will continue until some trade-off occurs between rule accuracy, coverage, and complexity.
- FOIL stops when the length of the length of the rule is larger than the data.
- FOIL also post-prunes each rule it learns, using the same strategy as ID3 (both by Quinlan).

Learning Recursive Rule Sets

- If we include the target predicate in Predicates then FOIL will consider it.
- This allows for the formation of recursive rules
  Ancestor(x,y) ← Parent(x,y)
  Ancestor(x,y) ← Parent(x,z) ∧ Ancestor(z,y)
- So, it is possible.

Induction As Inverted Deduction

- Induction is finding h such
  that $\forall \langle x_i, f(x_i) \rangle \in D\ B \wedge h \wedge x_i \rightarrow f(x_i)$ where x_i is i th training instance
  $f(x_i)$ is the target function value for x_i
  B is other background knowledge
- Design an inductive algorithm by inverting the operators for automated deduction.

7.1 Inverted Example

- Target concept is Child(u,v) s.t. the child of u is v.
- We are given the single positive example (x_i)
  Child(Bob,Sharon)
  and the data is described by
  Male(Bob), Female(Sharon), Father(Sharon,Bob).
- We have the general background knowledge (B)
  Parent(u,v) ← Father(u,v).
- So we have:
  x_i : Male(Bob), Female(Sharon), Father(Sharon,Bob)

f (x i) : Child(Bob,Sharon)
B : Parent(u,v) ← Father(u,v)

- What satisfies $\forall \langle x_i, f(x_i) \rangle \in D \; B \wedge h \wedge x_i \rightarrow f(x_i)$?
  h 1 : Child(u,v) ← Father(v,u)
  h 2 : Child(u,v) ← Parent(v,u)
- Notice that h 1 does not require B .
- This process of augmenting the set of predicates, based on background knowledge, is often referred to as **constructive induction**.

Induction and Deduction

- The relationship between these two has been known for a while.

> *Induction is, in fact, the inverse operation of deduction, and cannot be conceived to exist without the corresponding operation, so that the question of relative importance cannot arise. Who thinks of asking whether addition or subtraction is the more important process in arithmetic? But at the same time much difference in difficulty may exist between a direct and inverse operation; ... it must be allowed that inductive investigations are of a far higher degree of difficulty and complexity than any questions of deduction...*
> *(Jevons 1874)*

- There are many well-known algorithms for deduction in first-order logic. Can we reverse them?

Inverse Entailment

- An **inverse entailment operator** O(B,D) takes the training data D and background knowledge B as input and outputs a hypothesis h such
  that O (B,D)=h where $\forall \langle x_i, f(x_i) \rangle \in D (B \wedge h \wedge x_i) \rightarrow f(x_i)$
- Of course, there will usually be many h that satisfy this.
- A common heuristic is to use the Minimum Description Length principle.

Inverse Entailment Pros and Cons

Pros:

- Subsumes earlier idea of finding h that “fits” training data
- Domain theory B helps define meaning of “fit” the data $B \wedge h \wedge x_i \rightarrow f(x_i)$
- Suggests algorithms that search H guided by B

Cons:

- Doesn't allow for noisy data.
  Consider $\forall \langle x_i, f(x_i) \rangle \in D \ (B \wedge h \wedge x_i) \rightarrow f(x_i)$
- First order logic gives a huge hypothesis space H . This leads to over-fitting and the intractability of calculating all acceptable h 's.
- The complexity of space search increases as the background knowledge B is increased.

Resolution Rule

- The resolution rule is a **sound** [6] and **complete** [7] rule for deductive inference in first-order logic.
- The rule is:
  $P \vee L$
  $\neg L \vee R$

  $P \vee R$
- More generally

1. Given initial clauses C1 and C2, find a literal L from C1 such that ¬L is in C2.
2. Form the resolvent C by including all literals from C1 and C2 except for L and ¬L:
   $C = (C1 - \{L\}) \cup (C2 - \{\neg L\})$

Inverting Resolution

- The inverse entailment operator must derive C2 given the resolvent C and C1.
- Say $C = A \vee B$, and $C1 = B \vee D$. How do we derive C2 s.t. $C1 \wedge C2 \rightarrow C$?
- Find the L that appears in C1 and not in C, then form C2 by including the following literals
  $C2 = (C - (C1 - \{L\})) \cup \{\neg L\}$

so
$C2 = A \vee \neg D$

- C2 can also be $A \vee \neg D \vee B$.
- In general, inverse resolution can produce multiple clauses C2.

Learning With Inverted Resolution

- Use inverse entailment to construct hypotheses that, together with the background information, entail the training data.
- Use sequential covering algorithm to iteratively learn a set of Horn clauses in this way.

1. Select a training example that is not yet covered by learned clauses.
2. Use inverse resolution rule to generate candidate hypothesis h that satisfies $B \wedge h \wedge x \rightarrow f(x)$, where B = background knowledge plus any learned clauses.

- This is example-driven search.
- If multiple candidate hypotheses then choose one with highest accuracy over the other examples.

First-Order Resolution

- In general

    1. Find a literal L1 from clause C1, L2 from clause C2, and substitution θ such that $L1\ \theta = \neg L2\ \theta$.
    2. Form the resolvent C by including all literals from $C1\theta$ and $C2\theta$, except for $L1\theta$ and $\neg L2\theta$. More precisely, the set of literals occurring in the conclusion C is
    $C = (C1 - \{L1\})\theta \cup (C2 - \{L2\})\theta$

- That is, θ is a unifying substitution for L1 and $\neg$ L2.
- For example, let $C1 = White(x) \leftarrow Swan(x)$ and $C2 = Swan(Fred)$.
  C1 can be re-written as $White(x) \vee \neg Swan(x)$
  Then $L1 = \neg Swan(x)$, $L2 = Swan(Fred)$ if $\theta = \{x/Fred\}$
  So C = White(Fred).

Inverting First-Order Resolution

- θ can be uniquely factored into θ1 and θ2 where θ1 contains all the substitutions involving variables from C1, and θ2 for C2.
- So now,
  $C = (C1 - \{L1\})\theta1 \cup (C2 - \{L2\})\theta2$
  which can be re-written as
  $C - (C1 - \{L1\})\theta1 = (C2 - \{L2\})\theta2$
  then by definition we have that $L2 = \neg L1\ \theta1\ \theta2^{-1}$ so
  $C2 = (C - (C1 - \{L1\}\theta1)\theta2^{-1} \cup \{\neg L1\ \theta1\ \theta2^{-1}\}$
- In applying this we will find multiple choices for L1, θ1, and θ2.

Inverted First-Order Example

- Target concept = GrandChild(x,y)
  D = GrandChild(Bob, Shanon)
  B = {Father(Shanon,Tom), Father(Tom,Bob)}.
- C = GrandChild(Bob,Shanon)
  C1 = Father(Shanon, Tom)
  L1 = Father(Shanon, Tom)
  $\theta1^{-1} = \{\}$
  $\theta2^{-1} = \{Shanon/x\}$
- Then, the resulting C2 is
  GrandChild(Bob,x) ∨ ¬Father(x,Tom)
- This inferred clause may now be used as the conclusion C for a second inverse resolution.

Inverse Resolution Summary

- Generate hypothesis h that satisfy the constraint $B \wedge h \wedge x\ i \rightarrow f(x\ i)$
- Many might be generated, but they all satisfy the equation, unlike FOIL.
- Still, search is often unfocused and inefficient because it only considers a small fraction of the data when generating the hypothesis.

Generalization, θ-Subsumption, and Entailment

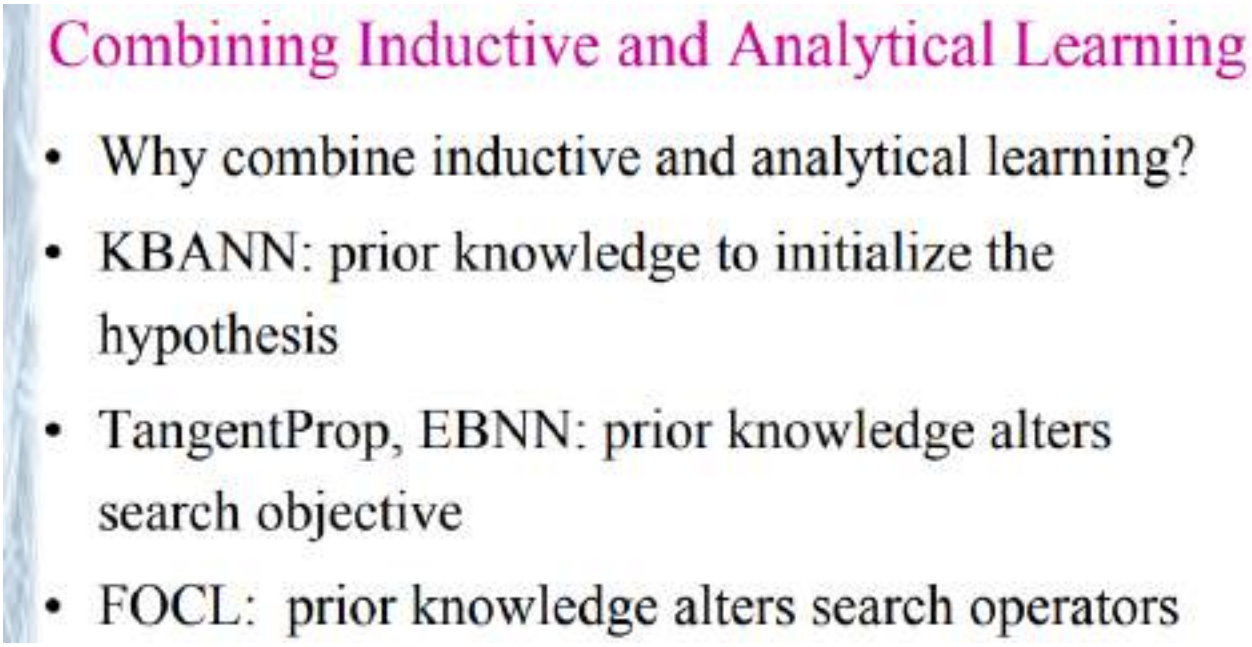
- **more-general-than**: Given two boolean functions h i (x) and h j (x) we say that h i is more general than h j if $\forall\ x: h\ j(x) \rightarrow h\ i(x)$ (used by Candidate-Elimination).
- **θ-subsumption**: Clause C1 θ-subsumes C2 iff there exists a θ such that $C1\theta \subseteq C2$ (i.e., the set of literals is a subset).

- **Entailment** C1 entails C2 iff C2 follows deductively from C1.
- more-general-than is a special case of θ-subsumption which is a special case of entailment.

PROGOL

- The idea in the PROGOL system is to reduce the combinatorial explosion by generating the most specific acceptable h .
- User specifies H by stating predicates, functions, and forms of arguments allowed for each.
- PROGOL uses sequential covering algorithm. For each $\langle x_i, f(x_i) \rangle$ it finds the most specific hypothesis h_i s.t. $B \wedge h_i \wedge x_i \rightarrow f(x_i)$. (actually, it considers only k -step entailment).
- It then conducts a general-to-specific search bounded by specific hypothesis h_i , choosing hypotheses with minimum description length.

Combining Inductive and Analytical Learning –
Motivation



Inductive Learning

- Goal: hypothesis fits data
- Justification: statistical inference
- Advantages: requires little prior knowledge
- Pitfalls: scarce data, incorrect bias

Analytical Learning

- Goal: hypothesis fits domain theory
- Justification: deductive inference
- Advantages: learns from scarce data
- Pitfalls: imperfect domain theory

Desirable Properties of a Combined System

- Given no domain theory, it should learn at least as effectively as purely inductive methods
- Given a perfect domain theory, it should learn at least as effectively as analytical methods

- Given an imperfect domain theory and imperfect training data, it should combine the two to outperform either purely inductive or purely analytical methods
- It should accommodate an unknown level of data in the training data
- It should accommodate an unknown level of error in the domain theory

Motivation

Inductive and Analytical Learning

Inductive learning	Analytical learning
Hypothesis fits data	Hypothesis fits domain theory
Statistical inference	Deductive inference
Requires little prior knowledge	Learns from scarce data
Syntactic inductive bias	Bias is domain theory

- *Difference:* Justification
  - Analytical: Logical justification; the output hypothesis follows deductively from the domain theory and the training examples
  - Inductive: Statistical justification; the output hypothesis follows from the assumption that the set of training examples is sufficiently large and that it is a representation of the underlying distribution of the examples

Plentiful data
No prior knowledge

Scarce data
Perfect prior knowledge

- General purpose learning method:
- No domain theory → learn as well as inductive methods
- Perfect domain theory → learn as well as PROLOG-EBG
- Accommodate arbitrary and unknown errors in domain theory
- Accommodate arbitrary and unknown errors in training data

Pure inductive methods formulate general hypotheses by recognising empirical regularities in the training examples

- Advantage: Don't require explicit prior knowledge
  Learn regularities based solely on the training data
- Disadvantage: Fail when insufficient training data is given
  Can be mislead by the implicit inductive bias they must cope with in order to generalise beyond the observed data

Pure analytical methods use prior knowledge to derive general hyp. deductively

- Advantage: Accurately generalise from a few training examples by using prior knowledge
- Disadvantage: Can be mislead when given incorrect or insufficient prior knowledge

Combination: For the better accuracy on the generalisation when prior knowledge is available and the reliance on the observed training data overcomes the shortcomings of the prior knowledge

Given no domain theory	It should learn at least as effectively as purely inductive methods
Given perfect domain theory	It should learn at least as effectively as pure analytical methods
Given an imperfect domain theory and imperfect training data	It should combine the two to outperform either purely inductive or purely analytical methods
	It should accommodate an unknown level of errors in the training data
	It should accommodate an unknown level of error in the domain theory

- Active current research => we do not yet have algorithms that satisfy all these constraints in a fully general fashion

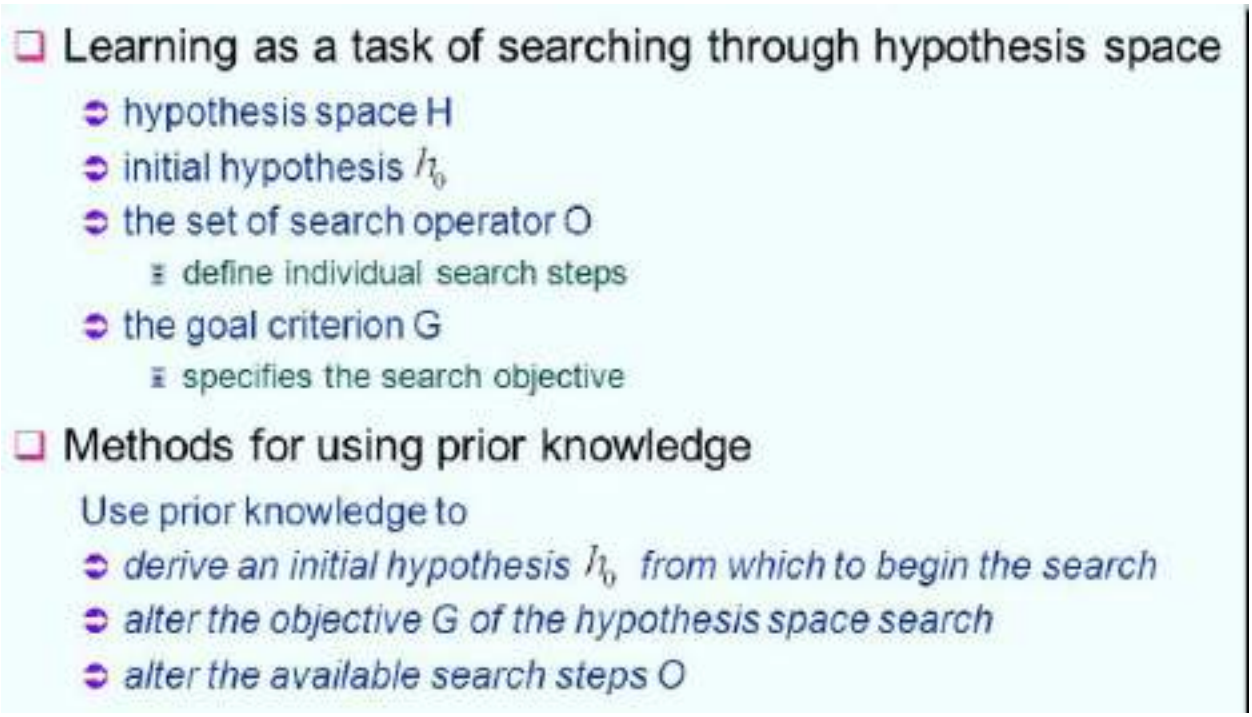
Learning Framework

- Given D: the training data, possibly containing errors
- Given B: the domain theory, possibly containing errors
- Given H: the hypothesis space
- Determine h: a hypothesis that best fits the training data and domain theory
- How can we determine the best fit? One idea is to find $\text{argmin}_{h \in H} k_D \text{error}_D(h) + k_B \text{error}_B(h)$

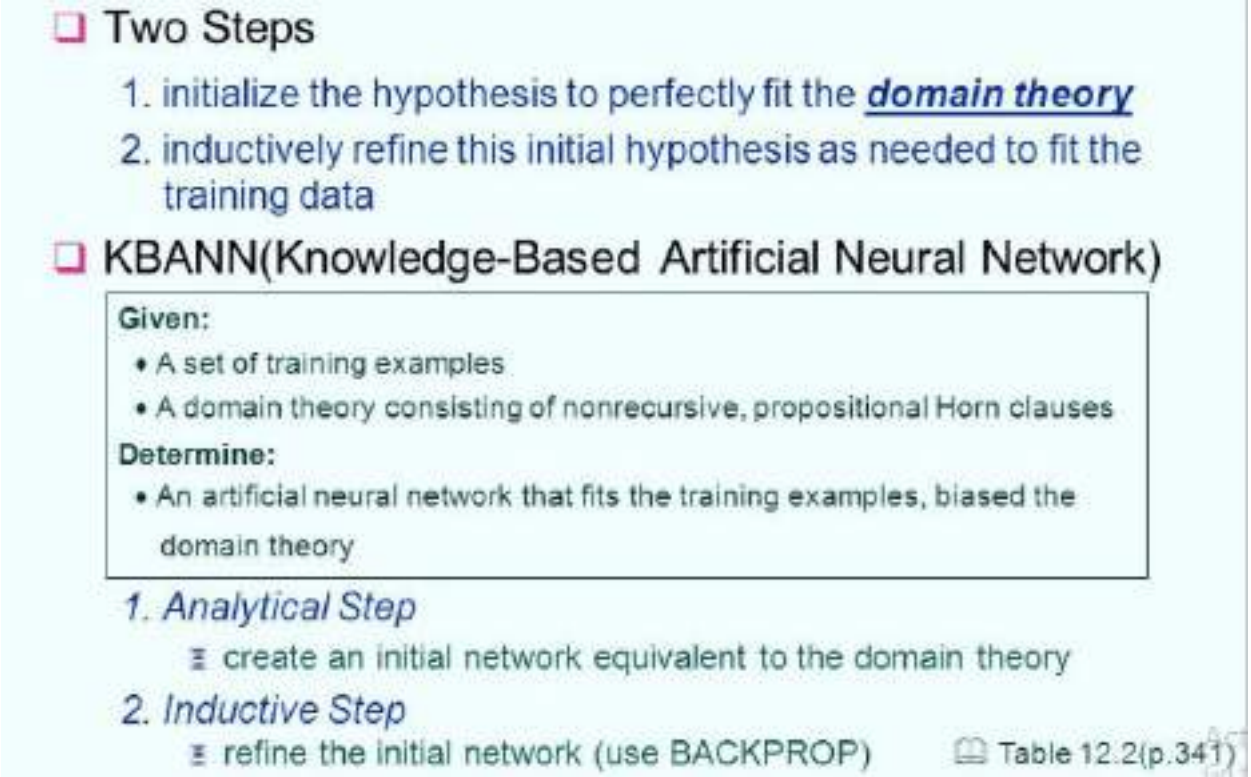
- Using Prior Knowledge to Initialize the Hypothesis
  - The KBANN Algorithm
- Using Prior Knowledge to Alter the Search Objective
  - The TANGENTPROP Algorithm
  - The EBNN Algorithm
- Using Prior Knowledge to Augment Search Operators
  - The FOCL Algorithm

Hypothesis Space Search Techniques

- Use prior knowledge to derive an initial hypothesis from which to begin the search - KBANN
- Use prior knowledge to alter the objective of the hypothesis space search - EBNN
- Use prior knowledge to alter the available search steps - FOCL



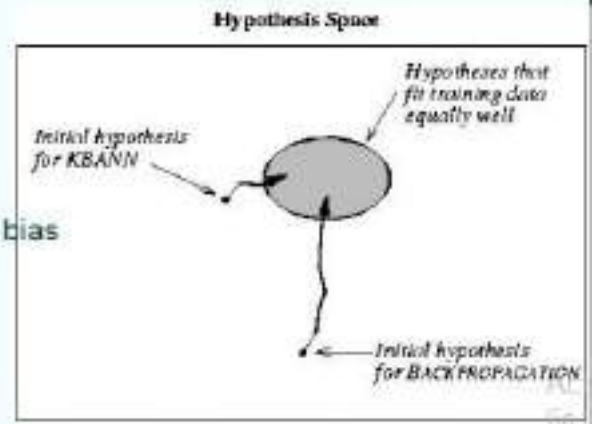
Using Prior Knowledge to Initialize the Hypothesis



KBANN

- Knowledge-Based Artificial Neural Network
- KBANN's bias comes from the domain-specific theory used to initialize the weights
- Input: D
- Input: B, a domain theory consisting of nonrecursive, propositional Horn clauses
- First, construct a neural network that classifies D according to the domain theory (see below)
- Second, employ backpropagation to fit D
- KBANN typically generalizes more accurately than standard backpropagation. However, B must be fairly accurate and B is limited to a particular syntactic form.

- KBANN vs. Backpropagation
  - when given an approximately correct domain theory & scarce training data
    - KBANN generalizes more accurately than Backpropagation
    - Classifying promoter regions in DNA
      - Backpropagation: error rate 8/106
      - KBANN: error rate 4/106
  - bias
    - KBANN
      - domain-specific theory
    - Backpropagation
      - domain-independent syntactic bias toward small weight values



Constructing the Initial KBANN Network

1. For each instance attribute create a network input
2. For each Horn clause in B, create a network unit as follows
   - Connect the antecedents of the Horn clause to the consequent, creating new network nodes as needed
   - For each non-negated antecedent of the clause, assign a weight of W to the corresponding sigmoid unit input
   - For each negated antecedent of the clause, assign a weight of -W to the corresponding sigmoid unit input
   - Set the threshold weight w_0 for this unit to -(n - .5)W where n is the number of non-negated antecedents of the clause
3. Add additional connections among the network units, connecting each network unit at depth i from the input layer to all network units at depth i+1. Assign random near-zero weights to these additional connections

Domain theory example

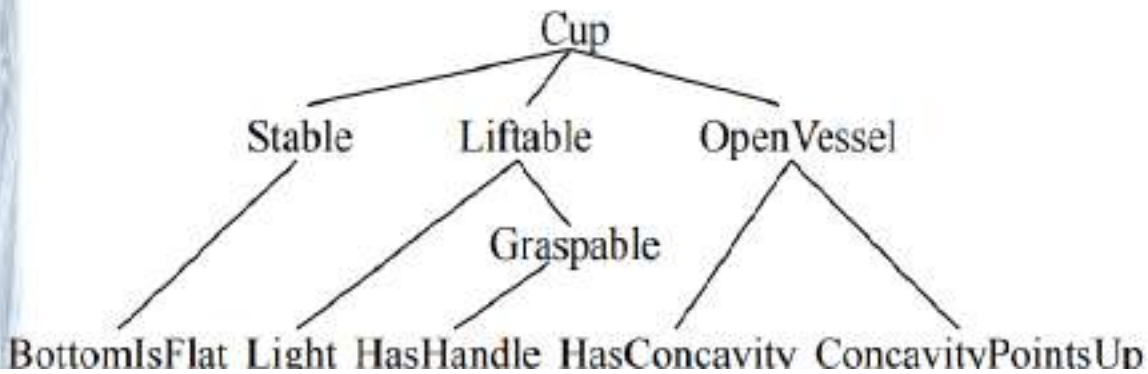
Cup ← Stable, Liftable, OpenVessel

Stable ← BottomIsFlat

Liftable ← Graspable, Light

Graspable ← HasHandle

OpenVessel ← HasConcavity, ConcavityPointsUp



	Cups				Non-Cups					
BottomIsFlat	√	√	√	√	√	√	√			√
ConcavityPointsUp	√	√	√	√	√		√	√		
Expensive	√		√				√		√	
Fragile	√	√			√	√		√		√
HandleOnTop					√		√			
HandleOnSide	√			√					√	
HasConcavity	√	√	√	√	√		√	√	√	√
HasHandle	√			√	√		√		√	
Light	√	√	√	√	√	√	√		√	
MadeOfCeramic	√				√		√	√		
MadeOfPaper				√					√	
MadeOfStyroForm		√	√			√				√

Using Prior Knowledge to Alter the Search Objective

- Use of prior knowledge
  - incorporate it into the error criterion minimized by gradient descent
  - network must fit a combined function of the training data & domain theory
- Form of prior knowledge
  - derivatives of the target function
  - certain type of prior knowledge can be expressed quite naturally
  - example: recognizing handwritten characters
    - "the identity of the character is independent of small translations and rotations of the image."

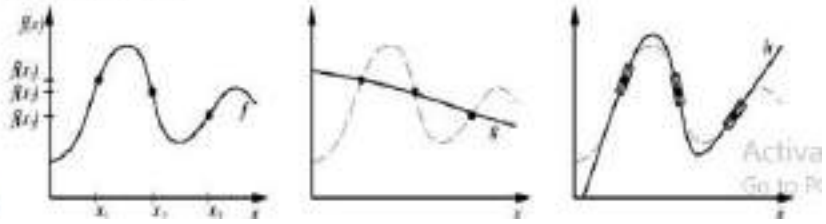
- Using prior knowledge to incorporate it into the error criterion minimised by the gradient descent, so that the network must fit a combined function of the training data and the domain theory
- Prior knowledge is given in the form of known derivatives of the target function
- Certain types of prior knowledge can be expressed quite naturally in this form
- *Example:*
  - Training neural networks to recognise handwritten characters;
  - *Derivatives:* the identity of the character is independent of small translations and rotations of the image

The TANGENTPROP Algorithm

- Domain knowledge expressed as derivatives of the target function with respect to transformations of its inputs
- Learning task involving an instance space X and target function f
- Training pair so far: $\langle x_i, f(x_i) \rangle$
- Various training derivatives of the target function are also provided is described by a single real value

$$\left\langle x_i, f(x_i), \frac{df(x)}{dx} | x_i \right\rangle$$

- *Example:* Learn target function f



- TangentProp can accept training derivatives with respect to various transformations
- Example: Learning to recognise handwritten characters
  - *Input:* x corresponding to an image of a single handwritten character
  - *Task:* Correctly classify the character
  - Interested in informing the learner that the target function is invariant to small rotations
  - Define a transformation $s(\alpha, x)$ which rotates the image x by α
  - Rotational invariance

    If $\dfrac{\partial F(s(\alpha, x))}{\partial \alpha} = 0$ then it is invariant to rotation

- *Question:* How are such training derivattives used by TangentProp to constrain the weights of neural networks?
- Answer: the training derivatives are incorporated into the error function that is minimised by gradient descent:

$$E=\sum_i (f(x_i)-\tilde{f}(x_i))^2$$

  - x_i Denotes the i-th training instance
  - f denotes the true target function
  - $\tilde{f}$ Denotes the function represented by the learned neural network
- *Here:* Additional term is added to the error function to penalise the discrepancies between the training derivatives and the actual derivatives of the learned neural network

Each transformation must be of the form $s_j(\alpha, x)$ where

- α is a continuous parameter
- s_j is differentiable

$$s_j(0, x) = x$$

- The modified error function

$$E=\sum_i \left[(f(x_i)-\tilde{f}(x_i))^2 + \mu \sum_j \left(\frac{\partial f(s_j(\alpha, x_i))}{\partial \alpha} - \frac{\partial \tilde{f}(s_j(\alpha, x_i))}{\partial \alpha} \right)^2_{\alpha=0} \right]$$

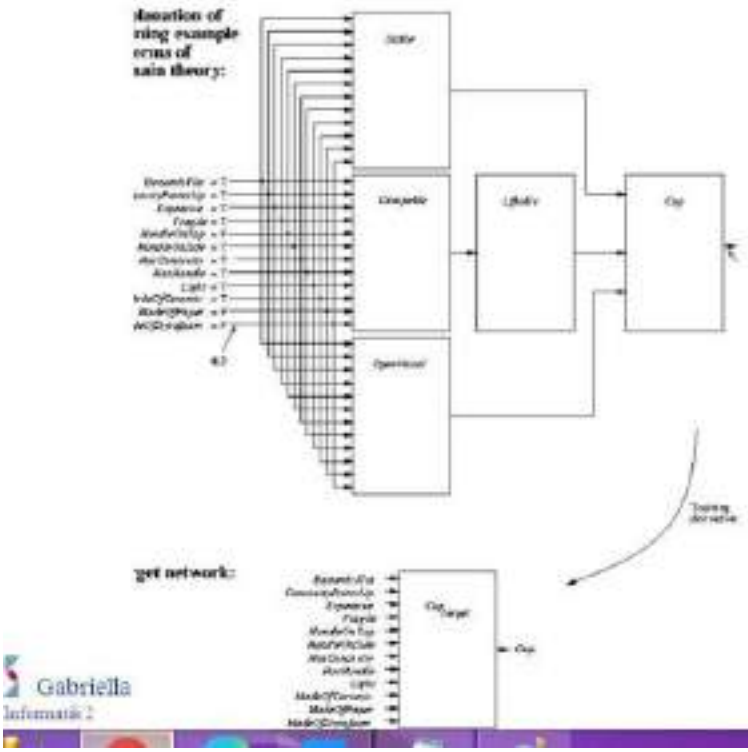
where

- μ is a constant provided by the user to determine the relative importance of fitting training values versus fitting training derivatives

- TangentProp uses prior knowledge in the form of derivatives of the desired target function with respect to the transformations of its input.
- It combines this prior knowledge with the observed training data, by minimising an objective function that measures both the network's error with respect to the training example values and its error with respect to the desired derivation
- The value μ determines the degree to which the network will fit one or the other of these two components in the total error
- *Disadvantage:* not robust to the errors of the prior knowledge
  - Algorithm will attempt to fit incorrect derivatives => ignoring the prior knowledge would have led to better generalisation => select μ
- EBNN automatically selects μ on an example-by-example basis

EBNN Algorithm

- EBNN computes the training derivatives by itself
- These are calculated by explaining each training example in terms of the given domain theory, then extracting training derivatives from this explanation
- EBNN addresses the issue of how to weigh the relative importance of inductive and analytical components of learning
- The value μ is chosen independently for each training example, based on a heuristic that considers how accurately the domain theory predicts the training value for this particular example
- Input:
  - Set of training examples $\langle x_i, f(x_i) \rangle$ with no training derivatives provided
  - Domain theory analogous to that used in explanation-based learning
- Output:
  - New neural network that approximates the target function f
- Network is trained to fit both training examples and training derivatives of f



- Each rectangular block represents a distinct neural network in the domain theory
- The network *Graspable* takes as input the description of an instance and produces as output a value indicating whether the object is graspable
- Some networks take the outputs of other networks as their input

- ***Goal***: learn a new neural network to describe the target function (target network)
- EBNN learns a target network by involving TangentProp
- EBNN passes the training values $\langle x_i, f(x_i) \rangle$ to TangentProp and provides it with derivatives calculated from the domain theory
- EBNN calculates the derivatives with respect to each feature of the input instance
- ***Example:*** x_i is described by *MadeOfStyrofoam = 0.2 (False)* +
  the domain theory prediction is that Cup = 0.8
  EBNN calculates the partial derivative of this prediction with respect to each instance feature yielding the set of derivatives

$$\left[\frac{\partial cup}{\partial BottomIsFlat}, \frac{\partial cup}{\partial ConcativityPointsUp}, \cdots, \frac{\partial cup}{\partial MadeOfStyrofoam} \right]_{x=x_i}$$

This set of derivations is the gradient of the domain theory prediction function with respect to the input instance

- General case where the target function has multiple output units, the gradient is computed for each of these outputs => the matrix of the gradients called the Jacobian of the target function
- Importance of the training derivatives:
  - If the feature *Expensive* is irrelevant => $\frac{\partial Cup}{\partial Expensive}$ has the value *0*
  - Large positive or negative derivatives correspond to the assertion that the feature is highly relevant to determine the target value

- Given *D* and *B* create a new fully connected feedforward network to represent the target function
  - Network is initialised with small random weights
  - For each $\langle x_i, f(x_i) \rangle$ determine the corresponding training derivatives
    - Use the *B* to predict the value of the target function for instance x_i : $A(x_i)$
    - The weights and actions of the domain theory network are analysed to extract the derivatives of $A(x_i)$ with respect to each of the components
  - Use a minor variant of TangentProp to train the target network to fit the error

$$E = \left[(f(x_i) - \tilde{f}(x_i))^2 + \mu_i \sum_j \left(\frac{\partial A(x)}{\partial x^j} - \frac{\partial \tilde{f}(x)}{\partial x^j} \right)^2_{(x=x_i)} \right] \quad \text{where}$$

$$\mu_i = 1 - \frac{|A(x_i) - f(x_i)|}{c}$$

x_j denotes the j-th component of the vector x, $0 \le \mu_i \le 1$

The coefficient *c* is a normalising constant whose value is chosen to assure that for all *i*

- EBNN uses a domain theory expressed as set of previously learned neural networks together with a set of training examples to train its output hypothesis (target network)
- For each training example EBNN uses its domain theory to explain the example then extracts training derivatives from this explanation
- For each attribute of the instance a training derivative is computed that describes how the target function value is influenced by a small change to this value according the domain theory
- Fitting the derivatives constrains the learned network to fit the dependencies given by the domain theory, while fitting the training values constrains it to fit the observed data itself
- The weight placed on fitting the derivatives is determined independently for each training example, based on how accurately the domain theory predicts the training value for this example

Using Prior Knowledge to Augment Search Operators

- Using prior knowledge to alter the hypothesis space search: using it to alter the set of operators that define legal steps in the search through the hypothesis space
- FOCL (Pazzani, Kibler 1992)

FOCL Algorithm

- *FOIL* and *FOCL* learn sets of first-order Horn clauses to cover the observed training examples
- They employ a sequential covering algorithm that learns a single Horn clause, removes the positive example covered by this new Horn clause and than iterates this procedure over the remaining training examples
- A new clause is created by performing general-to-specific search, beginning with the most general one
- Several candidate specializations of the current clause are then generated and the specialization with the greatest information gain relative to the training examples is chosen
- *Difference:* The way of how the candidates are specialised
- *Def:* A literal is *operational* if it is allowed to be used in describing an output hypothesis <=> *nonoperational:* occurs only in *B*
- *Example:* In *Cup* only 12 attributes are allowed as operational

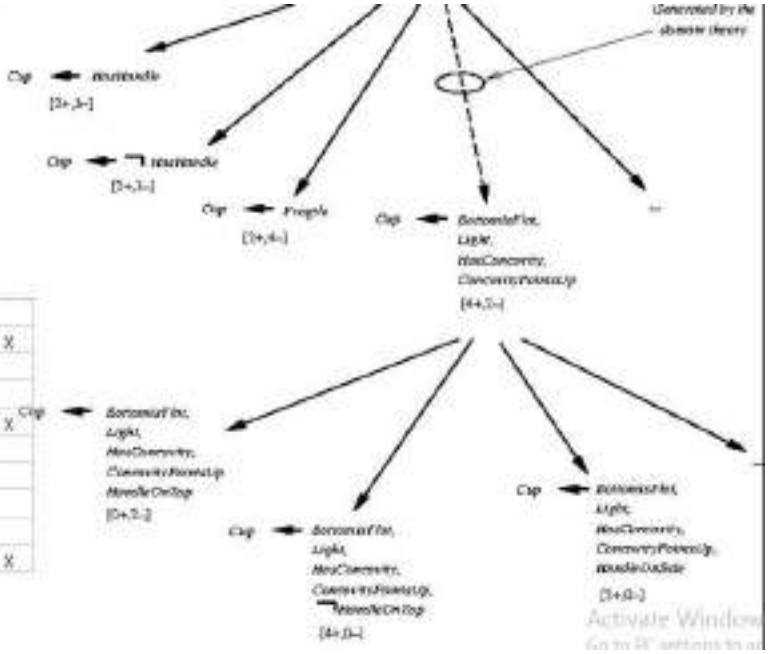
At each point in its general-to-specific search, FOCL expands its current hypothesis h using the following two operators:

1. For each operational literal that is not part of h, create a specialization of h by adding this single literal to the precondition
2. Create an operational, logically sufficient condition for the target concept according to B

   Add this set of literals to the current preconditions of h.

   Prune the preconditions of h by removing any literal that is unnecessary according to the training data.

Cup ← Stable, Liftable, OpenVessel
Stable ← BottomIsFlat
Liftable ← Graspable, Light
Graspable ← HasHandle
OpenVessel ← HasConcavity, ConcavityPointsUp



	Cups					
BottomIsFlat	x	x	x	x	x	x
ConcavityPointsUp	x	x	x	x	x	
Expensive	x		x			
Fragile	x	x			x	x
HandleOnTop					x	
HandleOnSide	x			x		
HasConcavity	x	x	x	x	x	
HasHandle	x			x	x	
Light	x	x	x	x	x	x

- FOCL uses the domain theory to increase the number of candidate spezializations considered at each step of the search for a single Horn clause
- FOCL uses both a syntactic generation of candidate specializations and a domain theory driven generation of candidate specialization at each step of the search
- *Example 1:* legal chessboard position:
  - 60 training example (30 legal and 30 illegal endgame board positions)
  - FOIL 86% over an independent set of examples
  - FOCL has domain theory with 76% accuracy
    It produced a hypothesis with generalisation accuracy 94%
- *Example 2:* Telephone network problem

Reinforcement learning

Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation.
Reinforcement learning differs from the supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

Example: The problem is as follows: We have an agent and a reward, with many hurdles in between. The agent is supposed to find the best possible path to reach the reward. The following problem explains the problem more easily.

- Input: The input should be an initial state from which the model will start
- Output: There are many possible output as there are variety of solution to a particular problem
- Training: The training is based upon the input, The model will return a state and the user will decide to reward or punish the model based on its output.
- The model keeps continues to learn.
- The best solution is decided based on the maximum reward.

Difference between Reinforcement learning and Supervised learning:

Reinforcement learning	Supervised learning
Reinforcement learning is all about making decisions sequentially. In simple words we can say that the output depends on the state of the current input and the next input depends on the output of the previous input	In Supervised learning the decision is made on the initial input or the input given at the start
In Reinforcement learning decision is dependent, So we give labels to sequences of dependent decisions Ex:Chess game	Supervised learning the decisions are independent of each other so labels are given to each decision. Ex:Object recognition

Types of Reinforcement: There are two types of Reinforcement:

1. **Positive –**
   Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.
   Advantages of reinforcement learning are:
   - Maximizes Performance
   - Sustain Change for a long period of time

   Disadvantages of reinforcement learning:
   - Too much Reinforcement can lead to overload of states which can diminish the results

2 Negative –
Negative Reinforcement is defined as strengthening of a behavior because a negative condition is stopped or avoided.
Advantages of reinforcement learning:
- Increases Behavior
- Provide defiance to minimum standard of performance

Disadvantages of reinforcement learning:
- It Only provides enough to meet up the minimum behavior

Various Practical applications of Reinforcement Learning –

- RL can be used in robotics for industrial automation.
- RL can be used in machine learning and data processing
- RL can be used to create training systems that provide custom instruction and materials according to the requirement of students.

RL can be used in large environments in the following situations:

1. A model of the environment is known, but an analytic solution is not available;
2. Only a simulation model of the environment is given (the subject of simulation-based optimization)
3. The only way to collect information about the environment is to interact with it.

The Learning Task

Target Concept

"Days on which Aldo enjoys his favorite water sport"

(you may find it more intuitive to think of "Days on which the beach will be crowded" concept)

Task

Learn to **predict** the value of *EnjoySport/Crowded* for an **arbitrary** day

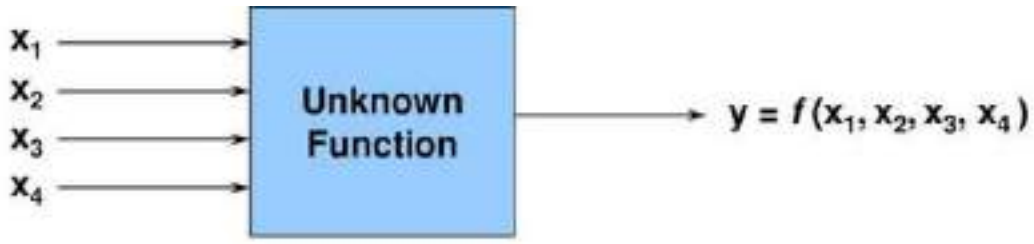
Training Examples for the Target Concept

Example	Sky	Air Temp	Humidity	Wind	Water	Forecast	Enjoy Sport
0	Sunny	Warm	Normal	Strong	Warm	Same	**Yes**
1	Sunny	Warm	High	Strong	Warm	Same	**Yes**
2	Rainy	Cold	High	Strong	Warm	Change	**No**
3	Sunny	Warm	High	Strong	Cool	Change	**Yes**

- 6 attributes (Nominal-valued (symbolic) attributes):

  Sky (SUNNY, RAİNY, CLOUDY), Temp (WARM,COLD), Humidity (NORMAL, HIGH),

  Wind (STRONG, WEAK), Water (WARM, COOL), Forecast (SAME, CHANGE)



Example	x_1	x_2	x_3	x_4	y
0	0	1	1	0	0
1	0	0	0	0	0
2	0	0	1	1	1
3	1	0	0	1	1
4	0	1	1	0	0
5	1	1	0	0	0
6	0	1	0	1	0

Hypothesis Space (H): Set of all possible hypotheses that the learner may consider during learning the target concept.

Q-Learning

Q-Learning is a basic form of Reinforcement Learning which uses Q-values (also called action values) to iteratively improve the behavior of the learning agent.

- **Q-Values or Action-Values:** Q-values are defined for states and actions. $Q(S, A)$ is an estimation of how good is it to take the action A at the state S. This estimation of $Q(S, A)$ will be iteratively computed using the **TD-Update rule** which we will see in the upcoming sections.
- **Rewards and Episodes:** An agent over the course of its lifetime starts from a start state, makes a number of transitions from its current state to a next state based on its choice of action and also the environment the agent is interacting in. At every step of transition, the agent from a state takes an action, observes a reward from the environment, and then transits to another state. If at any point of time the agent ends up in one of the terminating states that means there are no further transition possible. This is said to be the completion of an episode.
- **Temporal Difference or TD-Update:**

The Temporal Difference or TD-Update rule can be represented as follows :

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma Q(S', A') - Q(S, A)\right)$$

This update rule to estimate the value of Q is applied at every time step of the agents interaction with the environment. The terms used are explained below. :

- S : Current State of the agent.
- A : Current Action Picked according to some policy.
- S' : Next State where the agent ends up.
- A' : Next best action to be picked using current Q-value estimation, i.e. pick the action with the maximum Q-value in the next state.
- R : Current Reward observed from the environment in Response of current action.
- γ (>0 and <=1) : Discounting Factor for Future Rewards. Future rewars are less valuable than current rewards so they must be discounted. Since Q-value is an estimation of expected rewards from a state, discounting rule applies here as well.
- α : Step length taken to update the estimation of Q(S, A).
- **Choosing the Action to take using ϵ-greedy policy:**
- ϵ-greedy policy of is a very simple policy of choosing actions using the current Q-value estimations. It goes as follows :
- With probability $(1-\epsilon)$ choose the action which has the highest Q-value.
- With probability (ϵ) choose any action at random.
-

Now with all the theory required in hand let us take an example. We will use OpenAI's gym environment to train our Q-Learning model.

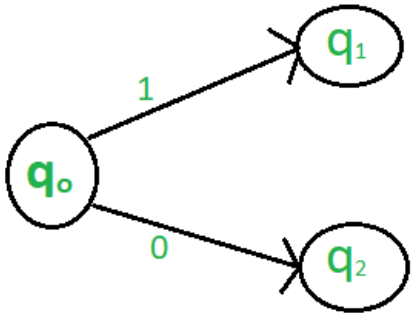
Command to Install **gym** –

```
pip install gym
```

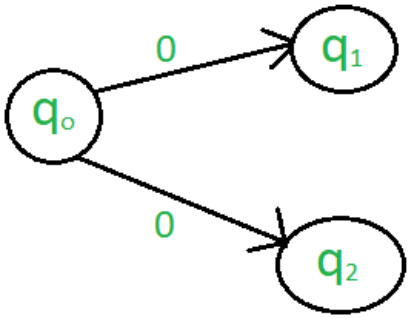
Before starting with example, you will need some helper code in order to visualize the working of the algorithms. There will be two helper files which need to be downloaded in the working directory. One can find the files .

Difference between Deterministic and Non-deterministic Algorithms

In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in case of **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it



Deterministic Algorithm



Non-Deterministic Algorithm

Some of the terms related to the non-deterministic algorithm are defined below:

- **choice(X) :** chooses any value randomly from the set X.
- **failure() :** denotes the unsuccessful solution.
- **success() :** Solution is successful and current thread terminates.

Example:

Problem Statement : *Search an element x on A[1:n] where n>=1, on successful search return j if a[j] is equals to x otherwise return 0.*

Non-deterministic Algorithm for this problem :

```
1.j= choice(a, n)
2.if(A[j]==x) then
   {
     write(j);
     success();
   }
4.       write(0); failure();
```

Deterministic Algorithm	Non-deterministic Algorithm
For a particular input the computer will give always same output.	For a particular input the computer will give different output on different execution.
Can solve the problem in polynomial time.	Can’t solve the problem in polynomial time.
Can determine the next step of execution.	Cannot determine the next step of execution due to more than one path the algorithm can take.