J.B. INSTITUTE OF ENGINEERING AND TECHNOLOGY

(UGC AUTONOMOUS)

Bhaskar Nagar, Moinabad Mandal, R.R. District, Hyderabad -500075

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT: MACHINE LEARNING

Branch: CSE

IV B. TECH I SEM (R-18)

Prepared by S.Gayathri Devi

Assistant Professor

LECTURE NOTES

Unit 1

What is Machine learning?

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. **Machine learning focuses on the development of computer programs** that can access data and use it learn for themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. **The primary aim is to allow the computers learn automatically** without human intervention or assistance and adjust actions accordingly.

Some machine learning methods

Machine learning algorithms are often categorized as supervised or unsupervised.

- **Supervised machine learning algorithms** can apply what has been learned in the past to new data using labeled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors in order to modify the model accordingly.
- In contrast, **unsupervised machine learning algorithms** are used when the information used to train is neither classified nor labeled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data. The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data.

- **Semi-supervised machine learning algorithms** fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training – typically a small amount of labeled data and a large amount of unlabeled data. The systems that use this method are able to considerably improve learning accuracy. Usually, semi-supervised learning is chosen when the acquired labeled data requires skilled and relevant resources in order to train it / learn from it. Otherwise, acquiring unlabeled data generally doesn't require additional resources.
- **Reinforcement machine learning algorithms** is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Trial and error search and delayed reward are the most relevant characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal.

Machine learning enables analysis of massive quantities of data. While it generally delivers faster, more accurate results in order to identify profitable opportunities or dangerous risks, it may also require additional time and resources to train it properly. Combining machine learning with AI and cognitive technologies can make it even more effective in processing large volumes of information.

Well-Posed Learning Problems:

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. Some of the examples for well-posed learning problem are given below:

1. Learning to classify chemical compounds

2. Learning to drive an autonomous vehicle
3. Learning to play bridge
4. Learning to parse natural language sentences

Driverless car

Can we phrase this as a well-posed learning problem?

T = driving a car on a busy interstate highway given image data from a forward-looking camera

P = number of errors made, as judged by a human observer

E = image and control information recorded from driving sessions with a human driving the same vehicle

Designing a Learning System:

In designing a learning system, we have to deal with (at least) the following issues:

1. Training experience
2. Target function
3. Learned function
4. Learning algorithm

Example: Consider the task T of parsing Swedish sentences, using the performance measure P of labeled precision and recall in a given test corpus (gold standard).

Training Experience

Issues concerning the training experience:

1. Direct or indirect evidence (supervised or unsupervised).
2. Controlled or uncontrolled sequence of training examples.
3. Representativity of training data in relation to test data.

Training data for a syntactic parser:

1. Treebank versus raw text corpus.
2. Constructed test suite versus random sample.
3. Training and test data from the same/similar/different sources with the same/similar/different annotations.

Target Function and Learned Function:

The problem of improving performance can often be reduced to the problem of learning some particular target function.

A shift-reduce parser can be trained by learning a transition function f : C -> C, where C is the set of possible parser configurations.

In many cases, we can only hope to acquire some approximation to the ideal target function.

The transition function f can be approximated by a function ˆf : Σ->Action from stack (top) symbols to parse actions.

Learning Algorithm:

In order to learn the (approximated) target function we require:

1. A set of training examples (input arguments)
2. A rule for estimating the value corresponding to each training example (if this is not directly available)
3. An algorithm for choosing the function that best fits the training data

* Given a treebank on which we can simulate the shift-reduce parser, we may decide to choose the function that maps each stack symbol σ to the action that occurs most frequently when σ is on top of the stack.

Designing of a Learning System:

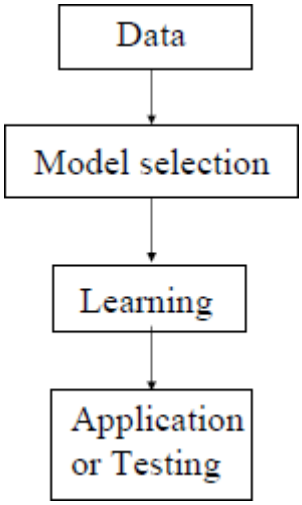


Fig: Designing of a Learning system (First View)

1. Data: $D = \{d_1, d_2, ..., d_n\}$

2. Model selection:

- **Select a model** or a set of models (with parameters)

  E.g. $y = ax + b + \varepsilon \qquad \varepsilon = N(0, \sigma)$

- **Select the error function** to be optimized

  E.g. $\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2$

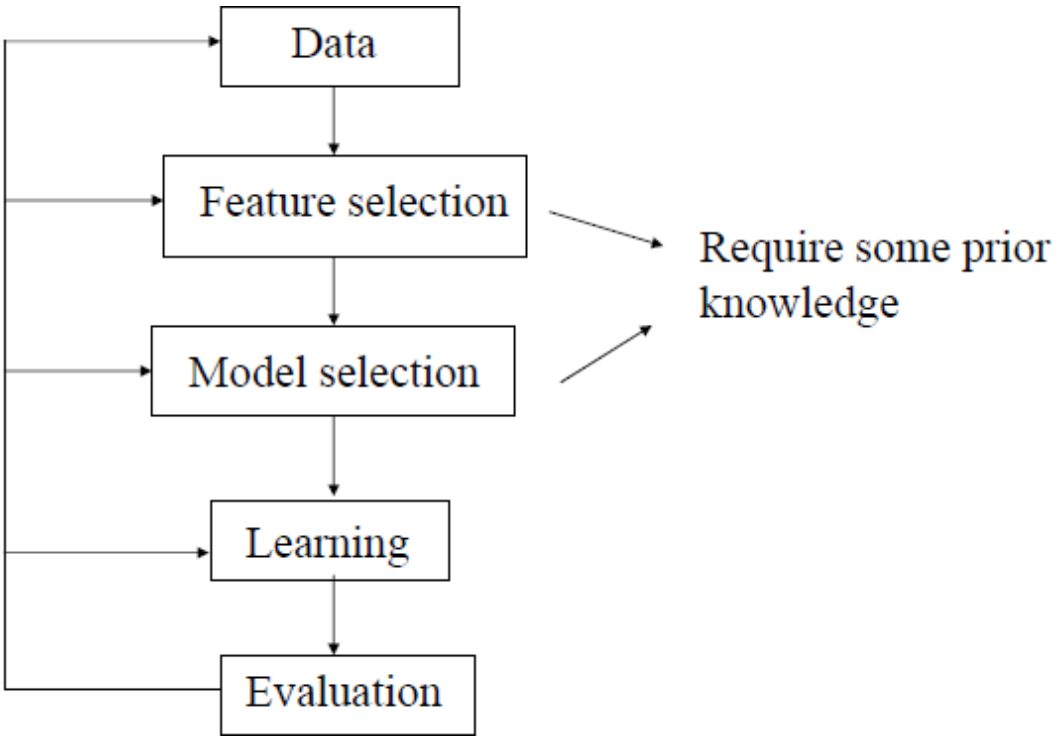
3. Learning:

- **Find the set of parameters optimizing the error function**
  - The model and parameters with the smallest error

4. Application (Evaluation):

- **Apply the learned model**
  - E.g. predict ys for new inputs **x** using learned $f(\mathbf{x})$

Design Cycle of a Learning System:



Data

Data may need a lot of:

- **Cleaning**
- **Preprocessing (conversions)**

Cleaning:

- **Get rid of errors, noise,**
- **Removal of redundancies**

Preprocessing:

- **Renaming**
- **Rescaling (normalization)**
- **Discretizations**
- **Abstraction**
- **Agreggation**
- **New attributes**

Data preprocessing

- **Renaming** (relabeling) categorical values to numbers
  - dangerous in conjunction with some learning methods
  - numbers will impose an order that is not warrantied
- **Rescaling (normalization):** continuous values transformed to some range, typically [-1, 1] or [0,1].
- **Discretizations (binning):** continuous values to a finite set of discrete values
- **Abstraction:** merge together categorical values
- **Aggregation:** summary or aggregation operations, such minimum value, maximum value etc.
- **New attributes:**
  - example: obesity-factor = weight/height

Data biases

- **Watch out for data biases:**
  - Try to understand the data source
  - It is very easy to derive "unexpected" results when data used for analysis and learning are biased (pre-selected)
  - Results (conclusions) derived for pre-selected data do not hold in general !!!

Data biases

Example 1: Risks in pregnancy study

- Sponsored by DARPA at military hospitals
- Study of a large sample of pregnant woman
- **Conclusion:** the factor with the largest impact on reducing risks during pregnancy (statistically significant) is a pregnant woman being single
- Single woman -> the smallest risk
- What is wrong?

Feature selection

- **The size (dimensionality) of a sample** can be enormous

  $x_i = (x_i^1, x_i^2, ..., x_i^d)$ d - very large
- **Example: document classification**
  - 10,000 different words
  - Inputs: counts of occurrences of different words
  - Too many parameters to learn (not enough samples to justify the estimates the parameters of the model)
- **Dimensionality reduction: replace inputs with features**
  - **Extract relevant inputs** (e.g. mutual information measure)
  - **PCA** – principal component analysis
  - **Group (cluster) similar words** (uses a similarity measure)
    - Replace with the group label

Model selection

- **What is the right model to learn?**
  - A prior knowledge helps a lot, but still a lot of guessing
  - Initial data analysis and visualization
    - We can make a good guess about the form of the distribution, shape of the function
  - Independences and correlations
  - **Overfitting problem**
    - Take into account the **bias and variance** of error estimates

Learning

Learning = optimization problem

- Optimization problems can be hard to solve. Right choice of a model and an error function makes a difference.
- **Parameter optimizations**
  - Gradient descent, Conjugate gradient
  - Newton-Rhapson
  - Levenberg-Marquard

  Some can be carried **on-line** on a sample by sample basis

  Combinatorial optimizations (over discrete spaces):
  - Hill-climbing
  - Simulated-annealing
  - Genetic algorithms

Parametric optimizations

- Sometimes can be solved directly but this depends on the error function and the model
  - Example: squared error criterion for linear regression
- Very often the error function to be optimized is not that nice.

  $Error(\mathbf{w}) = f(\mathbf{w})$ $\quad \mathbf{w} = (w_0, w_1, w_2 \ldots w_k)$

  - a complex function of weights (parameters)

  Goal: $\mathbf{w}^* = \arg\min_{\mathbf{w}} f(\mathbf{w})$
- Typical solution: **iterative methods**.
- **Example: Gradient-descent method**

  Idea: move the weights (free parameters) gradually in the error decreasing direction

Evaluation.

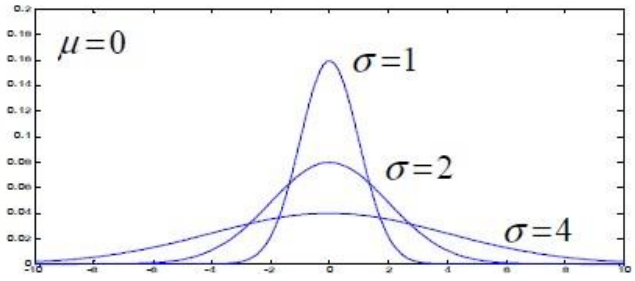
- **Simple holdout method.**
  - Divide the data to the training and test data.
- **Other more complex methods**
  - Based on cross-validation, random sub-sampling.
- What if we want to compare the predictive performance on a classification or a regression problem for two different learning methods?
- **Solution:** compare the error results on the test data set
- **Possible answer**: the method with better (smaller) testing error gives a better generalization error.
- Is this a good answer? How sure are we about the method with a better test score being truly better?

Evaluation.

- **Problem:** we cannot be 100 % sure about generalization errors
- **Solution:** test the statistical significance of the result
- **Central limit theorem:**

  Let random variables $X_1, X_2, \cdots X_n$ form a random sample from a distribution with mean μ and variance σ, then if the sample n is large, the distribution

$$\sum_{i=1}^{n} X_i \approx N(n\mu, n\sigma^2) \quad \text{or} \quad \frac{1}{n}\sum_{i=1}^{n} X_i \approx N(\mu, \sigma^2/n)$$



Issues in Machine Learning:

- 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?

Perspectives of Machine Learning:

- One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner.
- The learner's task is to search through a vast space to locate the hypothesis that is most consistent with the available training examples.The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value. This algorithm works well when the hypothesis

representation considered by the learner defines a continuously parameterized space of potential hypotheses.

- To be precise, it's better to follow this perspective of learning as a search problem in order to characterize learning methods by their search strategies and by the underlying structure of the search spaces they explore. This viewpoint is useful in formally analyzing the relationship between the size of the hypothesis space to be searched, the number of training examples available, and the confidence we can have that a hypothesis consistent with the training data will correctly generalize to unseen examples.

Concept Learning:

Concept learning: Inferring a boolean-valued function from training examples of its input and output.

Terminology and notation:

1. The set of items over which the concept is defined is called the set of instances and denoted by X.

2. The concept or function to be learned is called the target concept and denoted by

c : X -> {0, 1}.

3. Training examples consist of an instance x ∈X along with its target concept value c(x). (An instance x is positive if c(x) = 1 and negative if c(x) = 0.)

Hypothesis Spaces and Inductive Learning

Given a set of training examples of the target concept c, the problem faced by the learner is to hypothesize, or estimate, c.

- The set of all possible hypotheses that the learner may consider is denoted H.
- The goal of the learner is to find a hypothesis h ∈ H such that h(x) = c(x) for all x ∈ X.

- 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.

Hypothesis Representation

- The hypothesis space is usually determined by the human designer's choice of hypothesis representation.
- We assume:

1. An instance is represented as a tuple of attributes

<a1 = v1, . . . , a_n = v_n>

2. A hypothesis is represented as a conjunction of constraints on instance attributes.

3. Possible constraints are a_i = v (specifying a single value),? (any value is acceptable), and ; ∅ (no value is acceptable).

A Simple Concept Learning Task

Target concept: Proper name.

Instances: Words (in text).

Instance attributes:

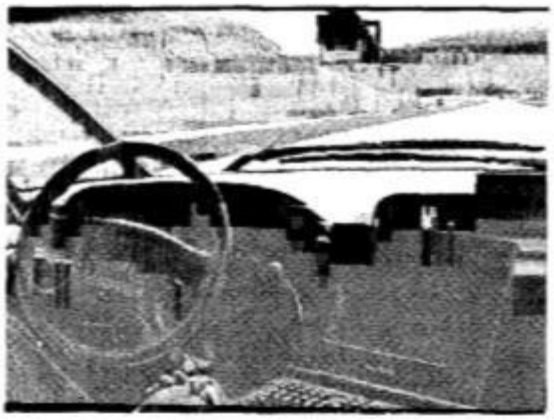
1. Capitalized: Yes, No.
2. Sentence-initial: Yes, No.
3. Contains hyphen: Yes, No.

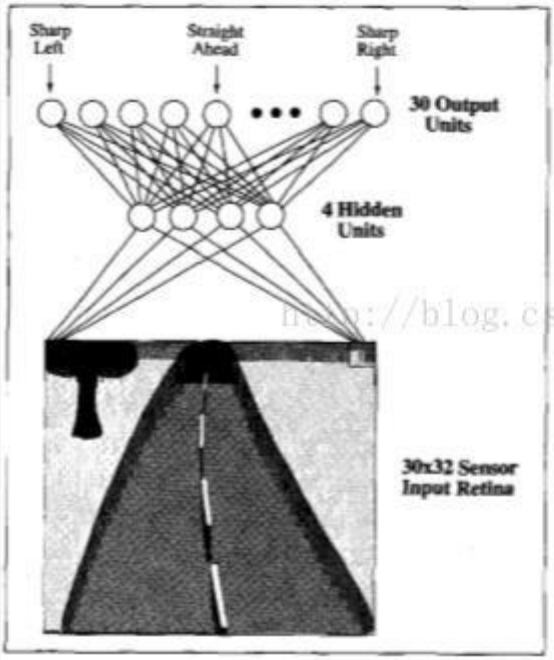
Training examples: $\langle\langle Yes, No, No\rangle, 1\rangle, \langle\langle No, No, No\rangle, 0\rangle, \ldots\rangle$

UNIT-2

Artificial Neural Networks

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPAGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs. ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.





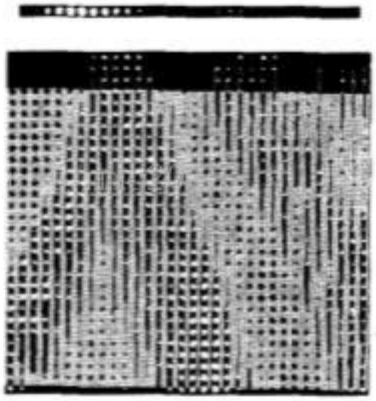


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 x 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

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.

The network structure of ALYINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph(DAG). In general, ANNs can be graphs with many types of structures-acyclic or cyclic, directed or undirected. This chapter will focus on the most common and practical ANN approaches, which are based on the BACKPROPAGATION algorithm. The BACK- PROPAGATION algorithm assumes the network is a fixed structure that corresponds to a directed graph, possibly containing cycles. Learning corresponds to choosing a weight value for each edge in the graph. Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNINGIt 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. Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.

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

PERCEPTRON - the most simple ANN system

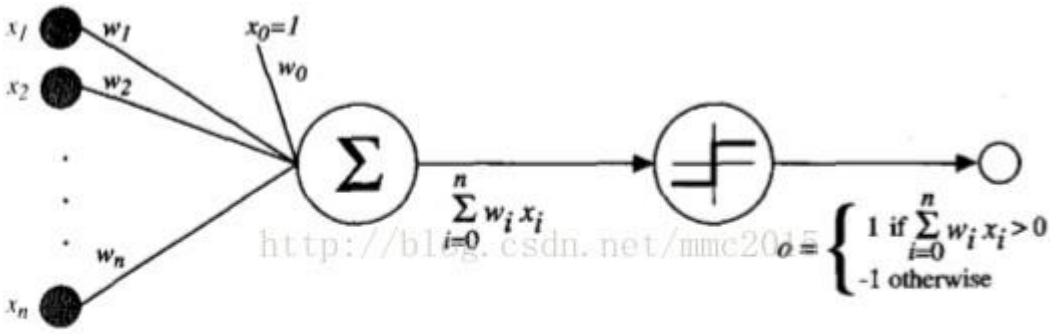


FIGURE 4.2
A perceptron.

Representational Power of Perceptron

In fact, AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least m of the n inputs to the perceptron must be true. The OR function corresponds to rn = 1 and the AND function to m = n. Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold wo accordingly.

In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage.

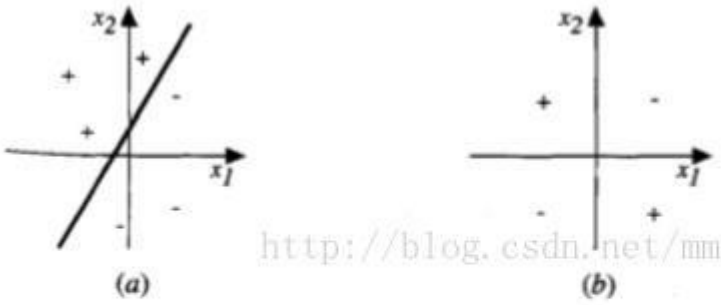


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the perceptron inputs. Positive examples are indicated by "+", negative by "−".

The Perceptron Training Rule

Let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct f 1 output for each of the given training examples. Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule (a variant of the LMS rule used in Chapter 1 for learning evaluation functions).

One way to learn an acceptable weight vector is to 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 wi <- wi + ∆wi, where ∆wi = q(t - o)xi. Here t is the target output for the current training example, o is the output generated by the perceptron, and q 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.

Why should this update rule converge toward successful weight values? In this case, (t - o) is zero, making ∆wi zero, so that no weights are updated. Suppose the perceptron outputs a -1, when the target output is + 1....

In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, provided the training examples are linearly separable and provided a sufficiently small 7 is used (see Minsky and Papert 1969). If the data are not linearly separable, convergence is not assured.

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. This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \qquad (4.2)$$

where D is the set of training examples, td is the target output for training example d, and od is the output of the linear unit for training example d.

VISUALIZING THE HYPOTHESIS SPACE

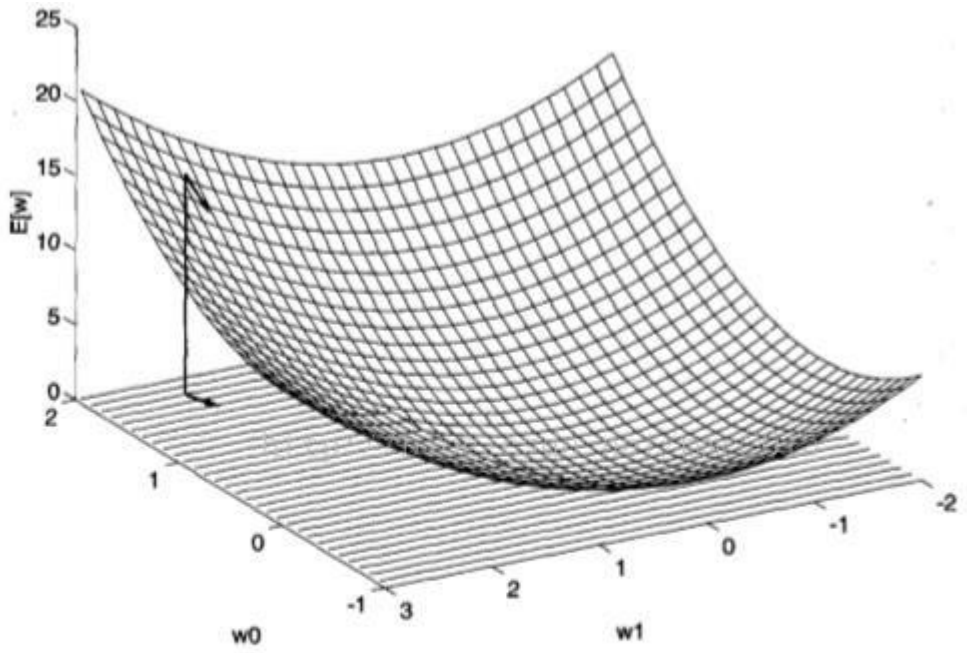


FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

DERIVATION OF THE GRADIENT DESCENT RULE

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id} \qquad (4.7)$$

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 Awi for each weight according to Equation (4.7). Update each weight wi by adding Awi, then repeat this process. This algorithm is given in Table 4.1.

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $(\vec{x}, t)$, *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 $(\vec{x}, t)$ 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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \quad \text{(T4.1)}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \quad \text{(T4.2)}$$

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate q is used. If r) is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of r) as the number of gradient descent steps grows.

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

One common variation on gradient descent intended to alleviate these difficulties is called incremental gradient descent, or alternatively stochastic gradient descent. Whereas the gradient descent training rule presented in Equation (4.7) 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. The modified training rule is like the training rule given by Equation (4.7) except that as we iterate through each training example we update the weight according to $\Delta w_i = q(t - o)x_i$, where t, o, and xi are the target value, unit output, and ith input for the training example in question.

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

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \quad (4.11)$$

where t, and od are the target value and the unit output value for training example d.

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 E(->w), stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\Delta E_d(\rightarrow w)$ rather than $\Delta E(\rightarrow w)$ to guide its search.

Both stochastic and standard gradient descent methods are commonly used in practice.

Notice the delta rule in Equation (4.10) is similar to the perceptron training rule in Equation (4.4.2). In fact, the two expressions appear to be identical. However, the rules are different because in the delta rule o refers to the linear unit output o(->x) = ->w * ->x, whereas for the perceptron rule o refers to the thresholded output o(->x) = sgn(->w * ->x).

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. *What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs*. One solution is the sigmoid unit- a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

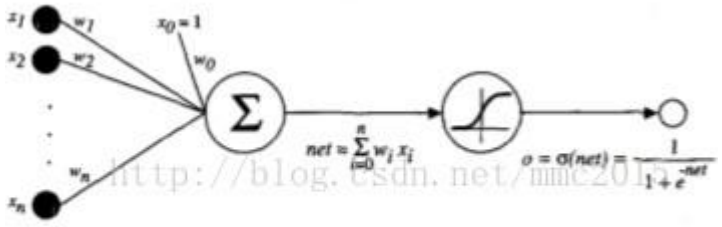


FIGURE 4.6
The sigmoid threshold unit.

The sigmoid function has the useful property that its derivative is easily expressed in terms of its output, as we shall see, the gradient descent learning rule makes use of this derivative.

$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))]$$

The BACKPROPAGATION Algorithm

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad (4.13)$$

where $outputs$ is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications.

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $(\vec{x}, \vec{t})$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} 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_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- 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

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

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

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

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

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

The BACKPROPAGATION algorithm is presented in Table 4.2. The notation used here is the same as that used in earlier sections, with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} denotes the corresponding weight.
- δ_n denotes the error term associated with unit n. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

The gradient descent weight-update rule (Equation [T4.5] in Table 4.2) is similar to the delta training rule (Equation [4.10]). Like the delta rule, it updates each weight in proportion to the learning rate η, the input value x_{ji} to which the weight is applied, and the error in the output of the unit. The only difference is that the error $(t - o)$ in the delta rule is replaced by a more complex error term, δ_j. The exact form of δ_j follows from the derivation of the weight-tuning rule given in Section 4.5.3. To understand it intuitively, first consider how δ_k is computed for each network *output* unit k (Equation [T4.3] in the algorithm). δ_k is simply the familiar $(t_k - o_k)$ from the delta rule, multiplied by the factor $o_k(1 - o_k)$, which is the derivative of the sigmoid squashing function. The δ_h value for each *hidden* unit h has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values t_k *only* for network outputs, no target values are directly available to indicate the error of hidden units' values. Instead, the error term for hidden unit h is calculated by summing the error terms δ_k for each output unit influenced by h, weighting each of the δ_k's by w_{kh}, the weight from hidden unit h to output unit k. This weight characterizes the degree to which hidden unit h is "responsible for" the error in output unit k.

ADDING MOMENTUM

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \qquad (4.18)$$

To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of a! is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next. This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.

LEARNING IN ARBITRARY ACYCLIC NETWORKS

The definition of BACKPROPAGATION presented in Table 4.2 applies only to two-layer networks. However, the algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule seen in Equation (T4.5) is retained, and the only change is to the procedure for computing δ values. In general, the δ_r value for a unit r in layer m is computed from the δ values at the next deeper layer $m + 1$ according to

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\, m+1} w_{sr} \delta_s \tag{4.19}$$

Notice this is identical to Step 3 in the algorithm of Table 4.2, so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers as we have assumed up to now. In the case that they are not, the rule for calculating δ for any internal unit (i.e., any unit that is not an output) is

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr} \delta_s \tag{4.20}$$

where $Downstream(r)$ is the set of units immediately downstream from unit r in the network: that is, all units whose inputs include the output of unit r. It is this general form of the weight-update rule that we derive in Section 4.5.3.

Derivation of the BACKPROPAGATIONRefer

P101-P103

REMARKS ON THE BACKPROPAGATION ALGORITHM

Convergence and Local Minima

Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.

Common heuris- tics to attempt to alleviate the problem of local minima include:

Add a momentum term to the weight-update rule as described in Equa- tion (4.18).

Use stochastic gradient descent rather than true gradient descent.

Train multiple networks using the same data, but initializing each network with different random weights. If the different training efforts lead to different local minima, then the network with the best performance over a separate validation data set can be selected. Alternatively, all networks can be retained and treated as a "committee" of networks whose output is the (possibly weighted) average of the individual network outputs.

Representational Power of Feedforward Networks

What set of functions can be represented by feedfonvard networks? Of course the answer depends on the width and depth of the networks. Although much is still unknown about which function classes can be described by which types of networks, three quite general results are known:

Boolean functions. To see how this can be done, consider the following general scheme for representing an arbitrary boolean function: For each possible input vector, create a distinct hidden unit and set its weights so that it activates if and only if this specific vector is input to the network. This produces a hidden layer that will always have exactly one unit active. Now implement the output unit as an OR gate that activates just for the desired input patterns.

Continuous functions.

Arbitraryfunctions. The proof of this involves showing that any function can be approximated by a linear combination of many localized functions that have value 0 everywhere except for some small region, and then showing that two layers of sigmoid units are sufficient to produce good local approximations.

Hypothesis Space Search and Inductive Bias

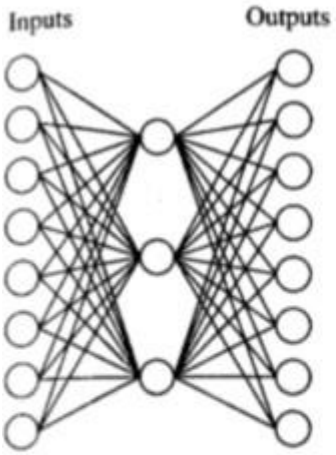
Notice this hypothesis space is continuous, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations. The fact that it is continuous, together with the fact that E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis. This structure is quite different from the general-to-specific ordering used to organize the search for symbolic concept

learning algorithms, or the simple-to-complex ordering over decision trees used by the ID3 and C4.5 algorithms.

One can roughly characterize it as smooth in- terpolation between data points. Given two positive training examples with no negative examples between them, BACKPROPAGATION will tend to label points in between as positive examples as well.

Hidden Layer Representation

Consider, for example, the network shown in Figure 4.7. Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7
Learned Hidden Layer Representation. This 8 × 3 × 8 network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

Generalization, Overfitting, and Stopping Criterion

One obvious choice is to continue training until the errcr E on the training examples falls below some predetermined threshold. In fact, this is a poor strategy

because BACKPROPAGATION is susceptible to overfitting the training examples at the cost of decreasing generalization accuracy over other unseen examples.

Given enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresen- tative characteristics of the particular training sample. This overfitting problem is analogous to the overfitting problem in decision tree learning (see Chapter 3).

Several techniques are available to address the overfitting problem for BACK- PROPAGATION learning:

One approach, known as weight decay, is to decrease each weight by some small factor during each iteration. This is equivalent to modifying the definition of E to include a penalty term corresponding to the total magnitude of the network weights. The motivation for this approach is to keep weight values small, to bias learning against complex decision surfaces.

One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data. The algorithm monitors the error with respect to this validation set, while using the training set to drive the gradient descent search. Clearly, it should use the number of iterations that produces the lowest error over the validation set, since this is the best indicator of network performance over unseen examples. In typical implementations of this approach, two copies of the network weights are kept: one copy for training and a separate copy of the best-performing weights thus far, measured by their error over the validation set. Once the trained weights reach a significantly higher error over the validation set than the stored weights, training is terminated and the stored weights are returned as the final hypothesis.

In general, the issue of overfitting and how to overcome it is a subtle one. The above cross-validation approach works best when extra data are available to provide a validation set. Unfortunately, however, the problem of overfitting is most severe for small training sets. In these cases, a k-fold cross-validation approach is sometimes used, in which cross validation is performed k different times, each

time using a different partitioning of the data into training and validation sets, and the results are then averaged. In one version of this approach, the m available examples are partitioned into k disjoint subsets, each of size m/k. The cross-validation procedure is then run k times, each time using a different one of these subsets as the validation set and combining the other subsets for the training set. Thus, each example is used in the validation set for one of the experiments and in the training set for the other k - 1 experiments. On each experiment the above cross-validation approach is used to determine the number of iterations i that yield the best performance on the validation set. The mean /i of these estimates for i is then calculated, and a final run of BACKPROPAGATION is performed training on all n examples for /i iterations, with no validation set. This procedure is closely related to the procedure for comparing two learning methods based on limited data, described in Chapter 5.

Decision Tree Learning:

Decision tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions. This chapter describes a family of decision tree learning algorithms that includes widely used algorithms such as ID3, ASSISTANT, and C4.5. These decision tree learning methods search a completely expressive hypothesis space and thus avoid the difficulties of restricted hypothesis spaces. Their inductive bias is a preference for small trees over large trees.

FIGURE 3.1
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 (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

In general, 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 Figure 3.1 corresponds to the expression: (Outlook = Sunny ^ Humidity = Normal) v (Outlook = Overcast) v (Outlook = Rain A Wind = Weak).

Decision tree learning is generally best suited to problems with the following characteristics: instances are represented by attribute-value pairs; the target function has discrete output values; disjunctive descriptions may be required; the training data may contain errors; the training data may contain missing attribute values.

THE BASIC DECISION TREE LEARNING ALGORITHM

Our basic algorithm, ID3, learns decision trees by constructing them topdown, beginning with the question "which attribute should be tested at the root of the tree?'To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as the test at the root node of the tree. A descendant of the root node is then created for each possible value of this

attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute). The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices. A simplified version of the algorithm, specialized to learning boolean-valued functions (i.e., concept learning), is described in Table 3.1.

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
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for A
    - If $Examples_{v_i}$ 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($Examples_{v_i}$, *Target_attribute*, *Attributes* − {A}))
- End
- Return *Root*

* The best attribute is the one with highest *information gain*, as defined in Equation (3.4).

TABLE 3.1
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? We will define a statistical property, called informution gain, that measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at eachstep while growing the tree.

In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called entropy, that characterizes the (im)purity of an arbitrary 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

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \quad (3.1)$$

where p+, is the proportion of positive examples in S and p-, is the proportion of negative examples in S. In all calculations involving entropy we define 0 log 0 to be 0.

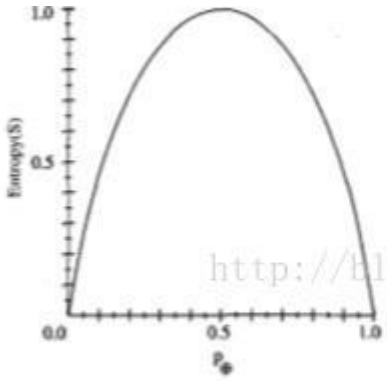


FIGURE 3.2
The entropy function relative to a boolean classification, as the proportion, $p_{\oplus}$, of positive examples varies between 0 and 1.

entropy is between 0 and 1. Figure 3.2 shows the form of the entropy function relative to a boolean classification, as $p_{\oplus}$ varies between 0 and 1.

More generally, if the target attribute can take on c different values, then the entropy of S relative to this c-wise classification is defined as

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \quad (3.3)$$

where pi is the proportion of S belonging to class i. Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in bits. Note also that if the target attribute can take on c possible values, the entropy can be as large as log2c.

The measure we will use, called information gain, is simply the expected reduction in entropy caused by partitioning the examples according to this

attribute. More precisely, the information gain, Gain(S, A) of an attribute A, relative to a collection of examples S, is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \qquad (3.4)$$

where Values(A) is the set of all possible values for attribute A, and Sv is the subset of S for which attribute A has value v (i.e., Sv = {s ∈ SIA(s) = v}).

Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized in Figure 3.3.

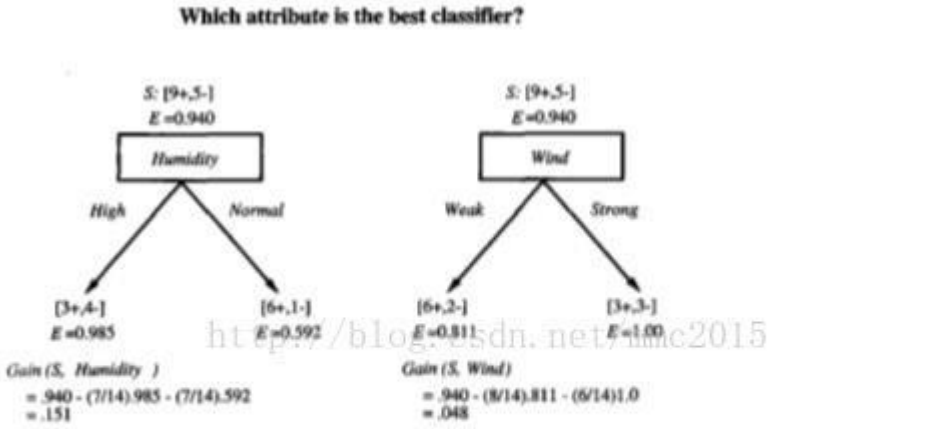


FIGURE 3.3
Humidity provides greater information gain than *Wind*, relative to the target classification. Here, *E* stands for entropy and *S* for the original collection of examples. Given an initial collection *S* of 9 positive and 5 negative examples, [9+, 5−], sorting these by their *Humidity* produces collections of [3+, 4−] (*Humidity* = *High*) and [6+, 1−] (*Humidity* = *Normal*). The information gained by this partitioning is .151, compared to a gain of only .048 for the attribute *Wind*.

An Illustrative Example

...

HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

By viewing ID3 in terms of its search space and search strategy, we can get some insight into its capabilities and limitations:

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 (such as methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.

ID3 maintains only a single current hypothesis as it searches through the space of decision trees. This contrasts, 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.

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. Therefore, it is susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal. Below we discuss an extension that adds a form of backtracking (post-pruning the decision tree.

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 (e.g., information gain) 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

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

Restriction Biases and Preference Biases

The inductive bias of ID3 is thus a preference for certain hypotheses over others (e.g., for shorter hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is typically called a preference bias (or, alternatively, a search bias). In contrast, the bias of the CANDIDATE-ELIMINATION alorithm 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, alternatively, a language bias).

Typically, 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.

Whereas ID3 exhibits a purely preference bias and CANDIDATE-ELIMINATION a purely restriction bias, some learning systems combine both. Consider, for example, the program described in Chapter 1 for learning a numerical evaluation function for game playing. In this case, the learned evaluation function is represented by a linear combination of a fixed set of board features, and the learning algorithm adjusts the parameters of this linear combination to best fit the available training data. In this case, the decision to use a linear function to represent the evaluation function introduces a restriction bias (nonlinear evaluation functions cannot be represented in this form). At the same time, the choice of a particular parameter tuning method (the LMS algorithm in this case) introduces a preference bias stemming from the ordered search through the space of all possible parameter values.

Why Prefer Short Hypotheses?

Is ID3's inductive bias favoring shorter decision trees a sound basis for generalizing beyond the training data? Philosophers and others have debated this question for centuries, and the debate remains unresolved to this day. William of

Occam was one of the first to discusst the question, around the year 1320, so this bias often goes by the name of Occam's razor.

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

Of course giving an inductive bias a name does not justify it. Why should one prefer simpler hypotheses? Notice that scientists sometimes appear to follow this inductive bias. One argument is that because there are fewer short hypotheses than long ones (based on straightforward combinatorial arguments), it is less likely that one will find a short hypothesis that coincidentally fits the training data. In contrast there are often many very complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data.

Upon closer examination, it turns out there is a major difficulty with the above argument. By the same reasoning we could have argued that one should prefer decision trees containing exactly 17 leaf nodes with 11 nonleaf nodes, that use the decision attribute A1 at the root, and test attributes A2 through All, in numerical order. There are relatively few such trees, and we might argue (by the same reasoning as above) that our a priori chance of finding one consistent with an arbitrary set of data is therefore small.....

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used internally by the learner. Two learners using different internal representations could therefore anive at different hypotheses, both justifying their contradictory conclusions by Occam's razor!

ISSUES IN DECISION TREE LEARNING

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure, andling training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency. Below we discuss each of these issues and extensions to the basic ID3 algorithm that address them. ID3 has itself been extended to address most of these issues, with the resulting system renamed C4.5 (Quinlan 1993).

Avoiding Overfitting the Data

Definition: Given a hypothesis space H, a hypothesis h E H is said to overlit the training data if there exists some alternative hypothesis h' E 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.

Random noise in the training examples can lead to overfitting. In fact, overfitting is possible even when the training data are noise-free, especially when small numbers of examples are associated with leaf nodes. In this case, it is quite possible for coincidental regularities to occur, in which some attribute happens to partition the examples very well, despite being unrelated to the actual target function. Whenever such coincidental regularities exist, there is a risk of overfitting.

There are several approaches to avoiding overfitting in decision tree learning. These can be grouped into two classes: approaches that stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data; approaches that allow the tree to overfit the data, and then post-prune the tree. Although the first of these approaches might seem.more direct, the second approach of post-pruning overfit trees has been found to be more successful in practice. This is due to the difficulty in the first approach of estimating precisely when to stop growing the tree.

A key question is what criterion is to be used to determine the correct final tree size. Approaches include:

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 an explicit 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, based on a heuristic called the Minimum Description

Length principle, is discussed further in Chapter 6, as well as in Quinlan and Rivest (1989) and Mehta et al. (199.5).

The first of the above approaches is the most common and is often referred to as a training and validation set approach. Of course, it is important that the validation set be large enough to itself provide a statistically significant sample of the instances. One common heuristic is to withhold one-third of the available examples for the validation set, using the other two-thirds for training.

REDUCED ERROR PRUNING

Nodes are removed only if the resulting pruned tree performs no worse than the original over the validation set. This 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.

Using a separate set of data to guide pruning is an effective approach provided a large amount of data is available. The major drawback of this approach is that when data is limited, withholding part of it for the validation set reduces even further the number of examples available for training.

RULE POST-PRUNING

Rule post-pruning involves the following steps:

1. 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.

2. 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.

3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.

4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

To illustrate, consider again the decision tree in Figure 3.1.....

Why convert the decision tree to rules before pruning? There are three main advantages:

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. In contrast, if the tree itself were pruned, the only two choices would be to remove the decision node completely, or to retain it in its original form.

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, we 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.

Converting to rules improves readability. Rules are often easier for to understand.

Incorporating Continuous-Valued Attributes

This can be accomplished by dynamically defining new discrete- valued attributes that partition the continuous attribute value into a discrete set of intervals. In particular, for an attribute A that is continuous-valued, the algorithm can dynamically create a new boolean attribute A, that is true if A < c and false otherwise.

The only question is how to select the best value for the threshold c. Clearly, we would like to pick a threshold, c, that produces the greatest information gain. As an example, suppose we wish to include the continuous-valued attribute *Temperature* in describing the training example days in the learning task of Table 3.2.....

Alternative Measures for Selecting Attribute

information gain

gain ratio

An alternative to the GainRatio, designed to directly address the above difficulty, is a distance-based measure introduced by Lopez de Mantaras (1991). This measure is based on defining a distance metric between partitions of'the data. Each attribute is evaluated based on the distance between the data partition it

creates and the perfect partition (i.e., the partition that perfectly classifies the training data). The attribute whose partition is closest to the perfect partition is chosen.

A variety of other selection measures have been proposed as well (e.g., see Breiman et al. 1984; Mingers 1989a; Kearns and Mansour 1996; Dietterich et al. 1996).

Handling Training Examples with Missing Attribute Values

One strategy for dealing with the missing attribute value is to assign it the value that is most common among training examples at node n.

A second, more complex procedure is to assign a probability to each of the possible values of A rather than simply assigning the most common value to A(x). These probabilities can be estimated again based on the observed frequencies of the various values for A among the examples at node n.

Handling Attributes with Differing Costs

ID3 can be modified to take into account attribute costs by introducing a cost term into the attribute selection measure. For example, we might divide the Gain by the cost of the attribute, so that lower-cost attributes would be preferred. While such cost-sensitive measures do not guarantee finding an optimal cost-sensitive decision tree, they do bias the search in favor of low-cost attributes.

A large variety of extensions to the basic ID3 algorithm has been developed by different researchers. These include methods for post-pruning trees, handling real-valued attributes, accommodating training examples with missing attribute values, incrementally refining decision trees as new training examples become available, using attribute selection measures other than information gain, and considering costs associated with instance attributes.

Unit-3

Bayesian Learning:

INTRODUCTION

Bayesian learning methods are relevant to our study of machine learning for two different reasons. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems. The second reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

One practical difficulty in applying Bayesian methods is that they typically require initial knowledge of many probabilities. When these probabilities are not known in advance they are often estimated based on background knowledge, previously available data, and assumptions about the form of the underlying distributions. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case (linear in the number of candidate hypotheses).

BAYES THEOREM

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

Bayes theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \qquad (6.1)$$

The most probable hypothesis h ∈ H given the observed data D (or at least one of the maximally probable if there are several) is called a maximum a posteriori (MAP) hypothesis.

More precisely, we will say that h_{MAP} is a MAP hypothesis provided

$$h_{MAP} \equiv \underset{h \in H}{\operatorname{argmax}} P(h|D)$$
$$= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)}$$
$$= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \quad (6.2)$$

Notice in the final step above we dropped the term $P(D)$ because it is a constant independent of h.

We will assume that every hypothesis in H is equally probable a priori (P(hi) = P(hj) for all hi and hj in H). In this case we can further simplify Equation (6.2) and need only consider the term P(D|h) to find the most probable hypothesis.

and any hypothesis that maximizes $P(D|h)$ is called a *maximum likelihood* (ML) hypothesis, h_{ML}.

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h) \quad (6.3)$$

BAYES THEOREM AND CONCEPT LEARNING

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

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

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

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

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

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

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

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

TABLE 6.1
Summary of basic probability formulas.

Brute-Force Bayes Concept Learning

BRUTE-FORCE MAP LEARNING algorithm

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

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

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

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

This algorithm may require significant computation, because it applies Bayes theorem to each hypothesis in H to calculate P(hJ D). While this may prove impractical for large hypothesis spaces, the algorithm is still of interest because it provides a standard against which we may judge the performance of other concept learning algorithms.

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

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if } h \text{ is consistent with } D \\ 0 & \text{otherwise} \end{cases} \quad (6.5)$$

where $|VS_{H,D}|$ is the number of hypotheses from H consistent with D. where

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

, because every hypothesis h in H has the same prior probability;

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases} \quad (6.4)$$

, because we assume noise-free training data, the probability of observing classification di given h is just 1 if di = h(xi) and 0 if di != h(xi).

MAP Hypotheses and Consistent Learners

We will say that a learning algorithm is a **consistent learner** provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability

distribution over H (i.e., P(hi) = P(hj) for all i, j), and ifwe assume deterministic, noise free training data (i.e., P(D Ih) = 1 if D and h are consistent, and 0 otherwise).

To summarize, the Bayesian framework allows one way to characterize the behavior of learning algorithms (e.g., FIND-S),e ven when the learning algorithm does not explicitly manipulate probabilities. By identifying probability distributions P(h) and P(Dlh) under which the algorithm outputs optimal (i.e., MAP) hypotheses, we can characterize the implicit assumptions , under which this algorithm behaves optimally.

MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

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

P165-P166probability densities and Normal distributions：

$$h_{ML} = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{m} (d_i - h(x_i))^2 \qquad (6.6)$$

Thus, Equation (6.6) shows that the maximum likelihood hypothesis hML is the one that minimizes the sum of the squared errors between the observed training values di and the hypothesis predictions h(xi). This holds under the assumption that the observed training values di are generated by adding random noise to the true target value, where this random noise is drawn independently for each example from a Normal distribution with zero mean.

Of course, the maximum likelihood hypothesis might not be the MAP hypothesis, but if one assumes uniform prior probabilities over the hypotheses then it is.

MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

In this section we derive a weight-training rule for neural network learning that seeks to maximize G(h, D) using gradient ascent.

To summarize section 6.4 and section 6.5, these two weight update rules converge toward maximum likelihood hypotheses in two different settings. The rule that minimizes sum of squared error seeks the maximum likelihood hypothesis under the assumption that the training data can be modeled by Normally distributed noise added to the target function value. The rule that minimizes cross entropy seeks the maximum likelihood hypothesis under the assumption that the observed boolean value is a probabilistic function of the input instance.

MINIMUM DESCRIPTION LENGTH PRINCIPLE

Clearly, to minimize the expected code length we should assign shorter codes to messages that are more probable. Shannon and Weaver (1949) showed that the optimal code (i.e., the code that minimizes the expected message length) assigns -log2pi bits to encode message i. We will refer to the number of bits required to encode message i using code C as the description length of message i with respect to C, which we denote by Lc(i).

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

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

Minimum Description Length principle: Choose h_{MDL} where

$$h_{MDL} = \underset{h \in H}{\operatorname{argmin}} L_{C_1}(h) + L_{C_2}(D|h) \qquad (6.17)$$

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

BAYES OPTIMAL CLASSIFIER

Bayes optimal classification:

$$\underset{v_j \in V}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) \qquad (6.18)$$

No other classification method using the same hypothesis space and same prior knowledge can outperform this method on average. This method maximizes the probability that the new instance is classified correctly, given the available data, hypothesis space, and prior probabilities over the hypotheses.

Note one curious property of the Bayes optimal classifier is that the predictions it makes can correspond to a hypothesis not contained in H! One way to view this situation is to think of the Bayes optimal classifier as effectively considering a hypothesis space H' different from the space of hypotheses H to which Bayes theorem is being applied. In particular, H' effectively includes hypotheses that perform comparisons between linear combinations of predictions from multiple hypotheses in H.

GIBBS ALGORITHM

An alternative, less optimal method is the Gibbs algorithm (see Opper and Haussler 1991), defined as follows:

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.

In particular, it implies that if the learner assumes a uniform prior over H, and if target concepts are in fact drawn from such a distribution when presented to the learner, *then classifying the next instance according to a hypothesis drawn at random from the current version space (according to a uniform distribution), will have expected error at most twice that of the Bayes optimal classijier*. Again, we have an example where a Bayesian analysis of a non-Bayesian algorithm yields insight into the performance of that algorithm.

NAIVE BAYES CLASSIFIER

The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target

function f (x) can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values (al, a2.. .a,). The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach:

The Bayesian approach to classifying the new instance is to assign the most probable target value, v_{MAP}, given the attribute values $\langle a_1, a_2 \ldots a_n \rangle$ that describe the instance.

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

We can use Bayes theorem to rewrite this expression as

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

$$= \underset{v_j \in V}{\text{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j) \quad (6.19)$$

The problem is that the number of these terms(different P(al, a2.. an | vj) terms) is equal to the number of *possible instances* times the number of *possible target values*. Therefore, we need to see every instance in the instance space many times in order to obtain reliable estimates.

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent(条件独立) given the target value.

Naive Bayes classifier:

$$v_{NB} = \underset{v_j \in V}{\text{argmax}} P(v_j) \prod_i P(a_i | v_j) \quad (6.20)$$

Whenever the naive Bayes assumption of conditional independence is satisfied, this naive Bayes classification VNB is identical to the MAP classification.

Notice that in a naive Bayes classifier the number of distinct P(ai l vj) terms that must be estimated from the training data is just the number of

distinct attribute values times the number of *distinct target values*-a much smaller number than if we were to estimate the P(a1, a2 . . . an l vj) terms as first contemplated.

BAYESIAN BELIEF NETWORKS

In this section we introduce the key concepts and the representation of Bayesian belief networks.

Consider an arbitrary set of random variables Y1 . . . Yn, where each variable Yi can take on the set of possible values V(Yi). We define the joint space of the set of variables Y to be the cross product V(Y1) x V(Y2) x . . . V(Yn). In other words, each item in the joint space corresponds to one of the possible assignments of values to the tuple of variables (Y1 . . . Yn). The probability distribution over this joint space is called the joint probability distribution. A Bayesian belief network describes the joint probability distribution for a set of variables.

Conditional Independenc

Representation

In general, a Bayesian network represents the joint probability distribution by specifying a set of conditional independence assumptions, represented by a directed acyclic graph), together with sets of local conditional probabilities. Each variable in the joint space is represented by a node in the Bayesian network. For each variable two types of information are specified, the network arcs and a conditional probability table.

The joint probability for any desired assignment of values (y1, . . . , yn) to the tuple of network variables (Y1 . . . Yn) can be computed by the formula

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

where $Parents(Y_i)$ denotes the set of immediate predecessors of Y_i in the network. Note the values of $P(y_i|Parents(Y_i))$ are precisely the values stored in the conditional probability table associated with node Y_i.

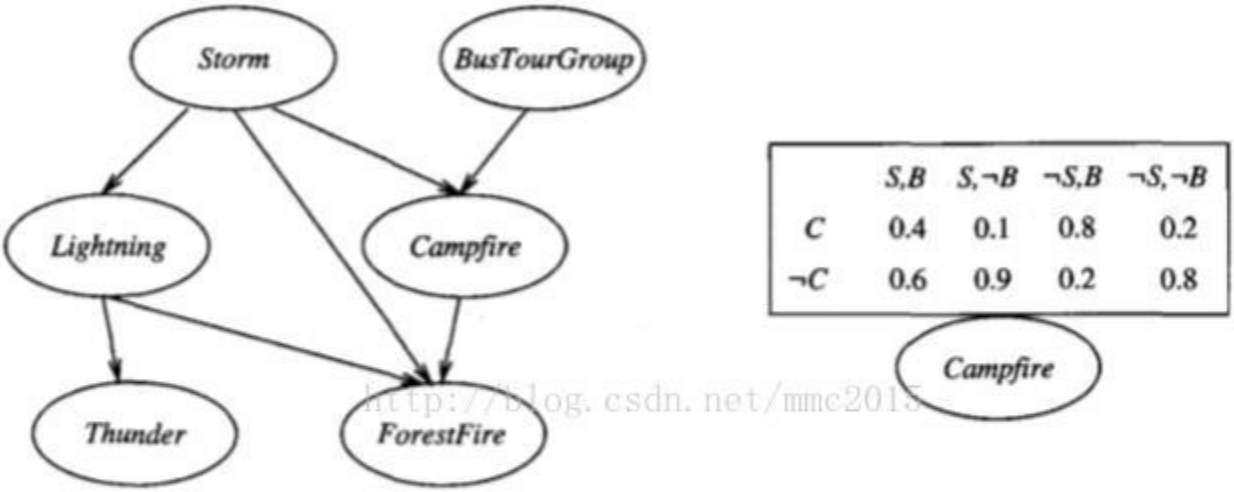


FIGURE 6.3
A Bayesian belief network. The network on the left represents a set of conditional independence assumptions. In particular, each node is asserted to be conditionally independent of its nondescendants, given its immediate parents. Associated with each node is a conditional probability table, which specifies the conditional distribution for the variable given its immediate parents in the graph. The conditional probability table for the *Campfire* node is shown at the right, where *Campfire* is abbreviated to *C*, *Storm* abbreviated to *S*, and *BusTourGroup* abbreviated to *B*.

Inference

In general, a Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables.

Exact inference of probabilities in general for an arbitrary Bayesian network is known to be NP-hard (Cooper 1990). Numerous methods have been proposed for probabilistic inference in Bayesian networks, including exact inference methods and approximate inference methods that sacrifice precision to gain efficiency. For example, Monte Carlo methods provide approximate solutions by *randomly sampling the distributions of the unobserved variables* (Pradham and Dagum 1996).

Learning Bayesian Belief Networks

In the case where the network structure is given in advance and the variables are fully observable in the training examples, learning the conditional

Applied to the problem of estimating the two means for Figure 6.4, the EM algorithm first initializes the hypothesis to $h = \langle \mu_1, \mu_2 \rangle$, where μ_1 and μ_2 are arbitrary initial values. It then iteratively re-estimates h by repeating the following two steps until the procedure converges to a stationary value for h.

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

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

probability tables is straightforward. We simply estimate the conditional probability table entries just as we would for a naive Bayes classifier.

In the case where the network structure is given but only some of the variable values are observable in the training data, the learning problem is more difficult.....6.11.5....

Gradient Ascent Training of Bayesian Networks P188-

P190

Learning the Structure of Bayesian Networks

Learning Bayesian networks when the network structure is not known in advance is also difficult.

THE EM ALGORITHM

In this section we describe the EM algorithm (Dempster et al. 1977), a widely used approach to learning in the presence of unobserved variables. The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

Estimating Means of k Gaussians

General Statement of EM Algorithm

More generally, the EM algorithm can be applied in many settings where we wish to estimate some set of parameters 8 that describe an underlying probability distribution, given only the observed portion of the full data produced by this distribution.

Derivation of the k Means AlgorithmP195-
P196

SUMMARY AND FURTHER READING

Bayesian methods provide the basis for probabilistic learning methods that accommodate (and require) knowledge about the prior probabilities of alternative hypotheses and about the probability of observing various data given the hypothesis. Bayesian methods allow assigning a posterior probability to each candidate hypothesis, based on these assumed priors and the observed data.

Bayesian methods can be used to determine the most probable hypothesis given the data-the maximum a posteriori (MAP) hypothesis. This is the optimal hypothesis in the sense that no other hypothesis is more likely.

The Bayes optimal classifier combines the predictions of all alternative hypotheses, weighted by their posterior probabilities, to calculate the most probable classification of each new instance.

The naive Bayes classifier is a Bayesian learning method that has been found to be useful in many practical applications. It is called "naive" because it incorporates the simplifying assumption that attribute values are conditionally independent, given the classification of the instance. When this assumption is met, the naive Bayes classifier outputs the MAP classification. Even when this assumption is not met, as in the case of learning to classify text, the naive Bayes classifier is often quite effective. Bayesian belief networks provide a more expressive representation for sets of conditional independence assumptions among subsets of the attributes.

The framework of Bayesian reasoning can provide a useful basis for analyzing certain learning methods that do not directly apply Bayes theorem. For example, under certain conditions it can be shown that minimizing the squared error when learning a real-valued target function corresponds to computing the maximum likelihood hypothesis.

The Minimum Description Length principle recommends choosing the hypothesis that minimizes the description length of the hypothesis plus the

description length of the data given the hypothesis. Bayes theorem and basic results from information theory can be used to provide a rationale for this principle.

In many practical learning tasks, some of the relevant instance variables may be unobservable. The EM algorithm provides a quite general approach to learning in the presence of unobservable variables. This algorithm begins with an arbitrary initial hypothesis. It then repeatedly calculates the expected values of the hidden variables (assuming the current hypothesis is correct), and then recalculates the maximum likelihood hypothesis (assuming the hidden variables have the expected values calculated by the first step). This procedure converges to a local maximum likelihood hypothesis, along with estimated values for the hidden variables.

Computational Learning Theory:

This theory seeks to answer questions such as "Under what conditions is successful learning possible and impossible?" and "Under what conditions is a particular learning algorithm assured of learning successfully?' Two specific frameworks for analyzing learning algorithms are considered. Within **the probably approximately correct (PAC) framework**, we identify classes of hypotheses that can and cannot be learned from a polynomial number of training examples and we define a natural measure of complexity for hypothesis spaces that allows bounding the number of training examples required for inductive learning. Within the **mistake bound framework**, we examine the number of training errors that will be made by a learner before it determines the correct hypothesis.

INTRODUCTION

Our goal is to answer questions such as:

Sample complexity. How many training examples are needed for a learner to converge (with high probability) to a successful hypothesis?

Computational complexity. How much computational effort is needed for a learner to converge (with high probability) to a successful hypothesis?

Mistake bound. How many training examples will the learner misclassify before converging to a successful hypothesis?

As we might expect, the answers to the above questions depend on the particular setting, or learning model, we have in mind.

PROBABLY LEARNING AN APPROXIMATELY CORRECT HYPOTHESIS

In this section we consider a particular setting for the learning problem, called the probably approximately correct (PAC) learning model. We begin by specifying the problem setting that defines the PAC learning model, then consider the questions of how many training examples and how much computation are required in order to learn various classes of target functions within this PAC model.

For the sake of simplicity, we restrict the discussion to the case of learning boolean-valued concepts from noise-free training data. However, many of the results can be extended to the more general scenario of learning real-valued target functions (see, for example, Natarajan 1991), and some can be extended to learning from certain types of noisy data (see, for example, Laird 1988; Kearns and Vazirani 1994).

The Problem Setting

X refer to the set of all possible instances over which target functions may be defined.

C refer to some set of target concepts that our learner might be called upon to learn. Each target concept c in C corresponds to some subset of X, or equivalently to some boolean-valued function c : X -> {0, 1}.

We assume instances are generated at random from X according to some probability distribution D. In general, D may be any stationary distribution(not change over time), and it will not generally be known to the learner.

Training examples are generated by drawing an instance x at random according to D, then presenting x along with its target value, c(x), to the learner.

The learner L considers some set H of possible hypotheses when attempting to learn the target concept. For example, H might be the set of all hypotheses describable by conjunctions of the attributes age and height.

After observing a sequence of training examples of the target concept c, L must output some hypothesis h from H, which is its estimate of c. To be fair, we evaluate the success of L by the performance of h over new instances drawn randomly from X according to D, the same probability distribution used to generate the training data.

Error of a Hypothesis

Definition: The **true error** (denoted $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target concept c and distribution $\mathcal{D}$ is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[c(x) \neq h(x)]$$

Here the notation $\Pr_{x \in \mathcal{D}}$ indicates that the probability is taken over the instance distribution $\mathcal{D}$.

Note that error depends strongly on the unknown probability distribution D.

We will use the term *training error* to refer to the fraction of *training examples* misclassified by h, in contrast to the true error defined above. Much of our analysis of the complexity of learning centers around the question "how probable is it that the observed training error for h gives a misleading estimate of the true errorv(h)?"

PAC Learnability

In short, we require only that the learner *probably learn* a hypothesis that is *approximately correct*-hence the term probably approximately correct learning, or PAC learning for short.

Consider some class C of possible target concepts and a learner L using hypothesis space H. Loosely speaking, we will say that the concept class C is PAC-learnable by L using H if, for any target concept c in C, L will with probability $(1 - \delta)$ output a hypothesis h with $error_{\mathcal{D}}(h) < \epsilon$, after observing a reasonable number of training examples and performing a reasonable amount of computation. More precisely,

Definition: Consider a concept class C defined over a set of instances X of length n and a learner L using hypothesis space H. C is **PAC-learnable** by L using H if for all $c \in C$, distributions $\mathcal{D}$ over X, ϵ such that $0 < \epsilon < 1/2$, and δ such that $0 < \delta < 1/2$, learner L will with probability at least $(1 - \delta)$ output a hypothesis $h \in H$ such that $error_{\mathcal{D}}(h) \leq \epsilon$, in time that is polynomial in $1/\epsilon$, $1/\delta$, n, and $size(c)$.

Here, n is the size of instances in X. For example, if instances in X are conjunctions of k boolean features, then n = k. The second space parameter, size(c), is the encoding length of c in C, assuming some representation for C. For example, if concepts in C are conjunctions of up to k boolean features, each described by listing the indices of the features in the conjunction, then size(c) is the number of boolean features actually used to describe c.

In fact, a typical approach to showing that some class C of target concepts is PAC-learnable, is to first show that each target concept in C can be learned from a polynomial number of training examples and then show that the processing time per example is also polynomially bounded.

SAMPLE COMPLEXITY FOR FINITE HYPOTHESIS SPACES

The growth in the number of required training examples with problem size, called the **sample complexity** of the learning problem, is the characteristic that is usually of greatest interest. The reason is that in most practical settings the factor that most limits success of the learner is the limited availability of training data.

Here we present a general bound on the sample complexity for a very broad class of learners, called **consistent learners**. A learner is consistent if it outputs hypotheses that perfectly fit the training data, whenever possible.

To accomplish this, it is useful to recall the definition of version space from Chapter 2. There we defined the version space, VSH,D, to be the set of all hypotheses h ∈ H that correctly classify the training examples D.

$$VS_{H,D} = \{h \in H | (\forall \langle x, c(x) \rangle \in D)\ (h(x) = c(x))\}$$

to bound the number of examples needed by any consistent learner, we need only bound the number of examples needed to assure that the version space contains no unacceptable hypotheses. The following definition, after Haussler (1988), states this condition precisely.

Definition: Consider a hypothesis space H, target concept c, instance distribution $\mathcal{D}$, and set of training examples D of c. The version space $VS_{H,D}$ is said to be **ϵ-exhausted** with respect to c and $\mathcal{D}$, if every hypothesis h in $VS_{H,D}$ has error less than ϵ with respect to c and $\mathcal{D}$.

$$(\forall h \in VS_{H,D})\ error_{\mathcal{D}}(h) < \epsilon$$

The version space is ∈-exhausted just in the case that all the hypotheses consistent with the observed training examples (i.e., those with zero training error) happen to have true error less than ∈.

Theorem 7.1. ϵ-exhausting the version space. If the hypothesis space H is finite, and D is a sequence of $m \geq 1$ independent randomly drawn examples of some target concept c, then for any $0 \leq \epsilon \leq 1$, the probability that the version space $VS_{H,D}$ is not ϵ-exhausted (with respect to c) is less than or equal to

$$|H|e^{-\epsilon m}$$

We have just proved an upper bound on the probability that the version space is not €-exhausted, based on the number of training examples m, the allowed error E, and the size of H.

Let us use this result to determine the number of training examples required to reduce this probability of failure below some desired level δ.

$$|H|e^{-\epsilon m} \leq \delta \tag{7.1}$$

Rearranging terms to solve for m, we find

$$m \geq \frac{1}{\epsilon}(\ln |H| + \ln(1/\delta)) \tag{7.2}$$

To summarize, the inequality shown in Equation (7.2) provides a general bound on the number of training examples sufficient for any consistent learner to successfully learn any target concept in H, for any desired values of 6 and E.

Agnostic Learning and Inconsistent Hypotheses

A learner that makes no assumption that the target concept is representable by H and that simply finds the hypothesis with minimum training error, is often called an **agnostic learner.**

In particular, $error_D(h)$ is defined as the fraction of the training examples in D that are misclassified by h. Note the $error_D(h)$ over the particular sample of training data D may differ from the true error $error_{\mathcal{D}}(h)$ over the entire probability distribution $\mathcal{D}$. Now let h_{best} denote the hypothesis from H having lowest training error over the training examples. How many training examples suffice to ensure (with high probability) that its true error $error_{\mathcal{D}}(h_{best})$ will be no more than $\epsilon + error_D(h_{best})$? Notice the question considered in the previous section is just a special case of this question, when $error_D(h_{best})$ happens to be zero.

This question can be answered (see Exercise 7.3) using an argument analo- gous to the proof of Theorem 7.1. It is useful here to invoke the general Hoeffding bounds (sometimes called the additive Chernoff bounds).

The Hoeffding bounds state that if the training error $error_D(h)$ is measured over the set D containing m randomly drawn examples, then

$$\Pr[error_{\mathcal{D}}(h) > error_D(h) + \epsilon] \leq e^{-2m\epsilon^2}$$

This gives us a bound on the probability that an arbitrarily chosen single hypothesis has a very misleading training error. To assure that the *best* hypothesis found by L has an error bounded in this way, we must consider the probability that any one of the $|H|$ hypotheses could have a large error

$$\Pr[(\exists h \in H)(error_{\mathcal{D}}(h) > error_D(h) + \epsilon)] \leq |H|e^{-2m\epsilon^2}$$

If we call this probability δ, and ask how many examples m suffice to hold δ to some desired value, we now obtain

$$m \geq \frac{1}{2\epsilon^2}(\ln|H| + \ln(1/\delta)) \qquad (7.3)$$

This is the generalization of Equation (7.2) to the case in which the learner still picks the best hypothesis h ∈ H, but where the best hypothesis may have nonzero training error.

Conjunctions of Boolean Literals Are PAC-Learnable

Consider the class C of target concepts described by conjunctions of boolean literals. Is C PAC-learnable? We can show that the answer is yes by first showing that any consistent learner will require only a polynomial number

of training examples to learn any c in C, and then suggesting a specific algorithm that uses polynomial time per training example.

The size |H| of this hypothesis space is 3". To see this, consider the fact that there are only three possibilities for each variable in any given hypothesis: Include the variable as a literal in the hypothesis, include its negation as a literal, or ignore it. Given n such variables, there are 3" distinct hypotheses.

Substituting $|H| = 3^n$ into Equation (7.2) gives the following bound for the sample complexity of learning conjunctions of up to n boolean literals.

$$m \geq \frac{1}{\epsilon}(n \ln 3 + \ln(1/\delta)) \qquad (7.4)$$

Notice that m grows linearly in the number of literals n, linearly in 1/∈, and logarithmically in 1/&.

It is the FIND-S algorithm, which incrementally computes the most specific hypothesis consistent with the training examples. For each new positive training example, this algorithm computes the intersection of the literals shared by the current hypothesis and the new training example, using time linear in n. Therefore, the FIND-S algorithm PAC-learns the concept class of conjunctions of n boolean literals with negations.

Theorem 7.2. PAC-learnability of boolean conjunctions. The class C of conjunctions of boolean literals is PAC-learnable by the FIND-S algorithm using $H = C$.

Proof. Equation (7.4) shows that the sample complexity for this concept class is polynomial in n, $1/\delta$, and $1/\epsilon$, and independent of $size(c)$. To incrementally process each training example, the FIND-S algorithm requires effort linear in n and independent of $1/\delta$, $1/\epsilon$, and $size(c)$. Therefore, this concept class is PAC-learnable by the FIND-S algorithm. □

PAC-Learnability of Other Concept Classes

UNBIASED LEARNERS:

Not all concept classes have polynomially bounded sample complexity, that the sample complexity for the unbiased concept class is exponential in n.

K-TERM DNF AND K-CNF CONCEPTS:

It is also possible to find concept classes that have polynomial sample complexity, but nevertheless cannot be learned in polynomial time.

Although k-term DNF has polynomial sample complexity, it does not have polynomial computational complexity for a learner using H = C.

Although k-CNF is more expressive than k-term DNF, it has both polynomial sample complexity and polynomial time complexity. Hence, the concept class k-term DNF is PAC learnable by an efficient algorithm using H = k-CNF.

SAMPLE COMPLEXITY FOR INFINITE HYPOTHESIS SPACES

Here we consider a second measure of the complexity of H, not |H|, but the Vapnik-Chervonenkis dimension of H (VC dimension, or VC(H), for short).

Shattering a Set of Instances

Given some instance set S, we say that H shatters S if **every** possible dichotomy of S can be represented by **some** hypothesis from H.

Definition: A set of instances S is **shattered** by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy.

The ability of H to shatter a set .of instances is thus a measure of its capacity to represent target concepts defined over these instances.

The Vapnik-Chervonenkis Dimension

Recall from Chapter 2 that an unbiased hypothesis space is one capable of representing every possible concept (dichotomy) definable over the instance space X. Put briefly, an unbiased hypothesis space H is one that shatters the instance space X.

Definition: The **Vapnik-Chervonenkis dimension**, $VC(H)$, of hypothesis space H defined over instance space X is the size of the largest finite subset of X shattered by H. If arbitrarily large finite sets of X can be shattered by H, then $VC(H) \equiv \infty$.

Note that for any finite H, $VC(H) \leq \log_2 |H|$. To see this, suppose that $VC(H) = d$. Then H will require 2^d distinct hypotheses to shatter d instances. Hence, $2^d \leq |H|$, and $d = VC(H) \leq \log_2 |H|$.

ILLUSTRATIW EXAMPLES

To get started, suppose the instance space X is the set of real numbers X = & (e.g., describing the height of people), and H the set of intervals on the real number line, VC(H) = 2.

Next consider the set X of instances corresponding to points on the x, y plane (see Figure 7.4). Let H be the set of all linear decision surfaces in the plane. In other words, H is the hypothesis space corresponding to a single perceptron unit with two inputs (see Chapter 4 for a general discussion of perceptrons), VC(H) = 3. More generally, it can be shown that the VC dimension of linear decision surfaces in an r dimensional space (i.e., the VC dimension of a perceptron with r inputs) is r + 1.

As one final example, suppose each instance in X is described by the conjunction of exactly three boolean literals, and suppose that each hypothesis in H is described by the conjunction of up to three boolean literals. The VC dimension for conjunctions of n boolean literals is at least n. In fact, it is exactly n, though showing this is more difficult, because it requires demonstrating that no set of n + 1 instances can be shattered.

Sample Complexity and the VC Dimension

$VC(H)$ as a measure for the complexity of H, it is possible to derive an alternative answer to this question, analogous to the earlier bound of Equation (7.2). This new bound (see Blumer et al. 1989) is

$$m \geq \frac{1}{\epsilon}(4 \log_2(2/\delta) + 8VC(H) \log_2(13/\epsilon)) \qquad (7.7)$$

Theorem 7.3. Lower bound on sample complexity. Consider any concept class C such that $VC(C) \geq 2$, any learner L, and any $0 < \epsilon < \frac{1}{8}$, and $0 < \delta < \frac{1}{100}$. Then there exists a distribution $\mathcal{D}$ and target concept in C such that if L observes fewer examples than

$$\max\left[\frac{1}{\epsilon}\log(1/\delta), \frac{VC(C)-1}{32\epsilon}\right]$$

then with probability at least δ, L outputs a hypothesis h having $error_{\mathcal{D}}(h) > \epsilon$.

This theorem states that if the number of training examples is too few, then no learner can PAC-learn every target concept in any nontrivial C. Thus, this theorem provides a lower bound on the number of training examples necessary for successful learning, complementing the earlier upper bound that gives a suficient number. Notice this lower bound is determined by the complexity of the concept class C, whereas our earlier upper bounds were determined by H. (why?)

VC Dimension for Neural Networks

Consider a network, G, of units, which forms a layered directed acyclic graph.

It turns out that we can bound the VC dimension of such networks based on their graph structure and the VC dimension of the primitive units from which they are constructed. To formalize this, we must first define a few more terms. Let n be the number of inputs to the network G, and let us assume that there is just one output node. Let each internal unit N_i of G (i.e., each node that is not an input) have at most r inputs and implement a boolean-valued function $c_i : \Re^r \rightarrow \{0, 1\}$ from some function class C. For example, if the internal nodes are perceptrons, then C will be the class of linear threshold functions defined over $\Re^r$.

We can now define the *G-composition of C* to be the class of all functions that can be implemented by the network G assuming individual units in G take on functions from the class C. In brief, the G-composition of C is the hypothesis space representable by the network G.

The following theorem bounds the VC dimension of the G-composition of C, based on the VC dimension of C and the structure of G.

Theorem 7.4. VC-dimension of directed acyclic layered networks. (See Kearns and Vazirani 1994.) Let G be a layered directed acyclic graph with n input nodes and $s \geq 2$ internal nodes, each having at most r inputs. Let C be a concept class over $\Re^r$ of VC dimension d, corresponding to the set of functions that can be described by each of the s internal nodes. Let C_G be the G-composition of C, corresponding to the set of functions that can be represented by G. Then $VC(C_G) \leq 2ds \log(es)$, where e is the base of the natural logarithm.

THE MISTAKE BOUND MODEL OF LEARNING

In this section we consider the mistake bound model of learning, in which the learner is evaluated by the total number of mistakes it makes before it converges to the correct hypothesis.

As in the PAC setting, we assume the learner receives a sequence of training examples. However, here we demand that upon receiving each example x, the learner must **predict** the target value c(x), **before** it is shown the correct target value by the trainer. The question considered is "How many mistakes will the learner make in its predictions before it learns the target concept?'

In the examples below, we consider instead the number of mistakes made before learning the target concept **exactly**. Learning the target concept exactly means converging to a hypothesis such that (Vx)h(x) = c(x).

Mistake Bound for the FIND-S Algorithm

Recall the FIND-S algorithm from Chapter 2, which incrementally computes the maximally specific hypothesis consistent with the training examples. Can we prove a bound on the total number of mistakes that FIND-S will make before exactly learning the target concept c? The answer is yes.

To see this, note first that if $c \in H$, then FIND-S can never mistakenly classify a negative example as positive. The reason is that its current hypothesis h is always at least as specific as the target concept c. Therefore, to calculate the number of mistakes it will make, we need only count the number of mistakes it will make misclassifying truly positive examples as negative.

Consider the first positive example encountered by FIND-S.T he learner will certainly make a mistake classifying this example, because its initial hypothesis labels every instance negative. However, the result will be that half of the 2n terms in its initial hypothesis will be eliminated, leaving only n terms. For each subsequent positive example that is mistakenly classified by the current hypothesis, at least one more of the remaining n terms must be eliminated from the hypothesis. **Therefore, the total number of mistakes can be at most n + 1.** This number of mistakes will be required in the worst case, corresponding to learning the most general possible target concept $(Vx)c(x) = 1$ and corresponding to a worst case sequence of instances that removes only one literal per mistake.

Mistake Bound for the HALVING Algorithm

If the majority of version space hypotheses classify the new instance as positive, then this prediction is output by the learner. Otherwise a negative prediction is output. This combination of learning the version space, together with using a majority vote to make subsequent predictions, is often called the HALVING algorithm. The CANDIDATE-ELIMINATlON algorithm and the LIST-THEN-ELIMINATlON algorithm from Chapter 2 are examples of such algorithms.

In this section we derive a worst-case bound on the number of mistakes that will be made by such a learner, for any finite hypothesis space H, assuming again that the target concept must be learned exactly.

To derive the mistake bound, note that the only time the HALVING algorithm can make a mistake is when the majority of hypotheses in its current version space incorrectly classify the new example. In this case, once the correct classification is revealed to the learner, the version space will be reduced to at most half its current size (i.e., only those hypotheses that voted with the minority will be retained). Given that each mistake reduces the size of the version space by at least half, and given that the initial version space contains only $|H|$ members, the maximum number of mistakes possible before the version space contains just one member is $\log_2 |H|$. In fact one can show the bound is $\lfloor \log_2 |H| \rfloor$. Consider, for example, the case in which $|H| = 7$. The first mistake must reduce $|H|$ to at most 3, and the second mistake will then reduce it to 1.

Note that $\lfloor \log_2 |H| \rfloor$ is a worst-case bound, and that it is possible for the HALVING algorithm to learn the target concept exactly without making any mistakes at all! This can occur because even when the majority vote is correct, the algorithm will remove the incorrect, minority hypotheses. If this occurs over the entire training sequence, then the version space may be reduced to a single member while making no mistakes along the way.

Optimal Mistake Bounds

It is interesting to ask what is the optimal mistake bound for an arbitrary concept class C, assuming H = C. By optimal mistake bound we mean the lowest worst-case mistake bound over all possible learning algorithms.

We define the optimal mistake bound for a concept class C below.

Definition: Let C be an arbitrary nonempty concept class. The **optimal mistake bound** for C, denoted $Opt(C)$, is the minimum over all possible learning algorithms A of $M_A(C)$.

$$Opt(C) \equiv \min_{A \in learning\ algorithms} M_A(C)$$

Speaking informally, this definition states that $Opt(C)$ is the number of mistakes made for the hardest target concept in C, using the hardest training sequence, by the best algorithm. Littlestone (1987) shows that for any concept class C, there is an interesting relationship among the optimal mistake bound for C, the bound of the HALVING algorithm, and the VC dimension of C, namely

$$VC(C) \leq Opt(C) \leq M_{Halving}(C) \leq log_2(|C|)$$

WEIGHTED-MAJORITY Algorithm

The WEIGHTED-MAJORITY algorithm makes predictions by taking a weighted vote among a pool of prediction algorithms and learns by altering the weight associated with each prediction algorithm.

One interesting property of the WEIGHTED-MAJORITY algorithm is that it is able to accommodate inconsistent training data. This is because it does not eliminate a hypothesis that is found to be inconsistent with some training example, but rather reduces its weight.

Whenever a pre- diction algorithm misclassifies a new training example its weight is decreased by multiplying it by some number B, where 0 <= B <= 1. The exact definition of the WEIGHTED-MAJORITY algorithm is given in Table 7.1.

a_i denotes the i^{th} prediction algorithm in the pool A of algorithms. w_i denotes the weight associated with a_i.

- For all i initialize $w_i \leftarrow 1$
- For each training example $\langle x, c(x) \rangle$
  - Initialize q_0 and q_1 to 0
  - For each prediction algorithm a_i
    - If $a_i(x) = 0$ then $q_0 \leftarrow q_0 + w_i$
      If $a_i(x) = 1$ then $q_1 \leftarrow q_1 + w_i$
- If $q_1 > q_0$ then predict $c(x) = 1$
  If $q_0 > q_1$ then predict $c(x) = 0$
  If $q_1 = q_0$ then predict 0 or 1 at random for $c(x)$
- For each prediction algorithm a_i in A do
  If $a_i(x) \neq c(x)$ then $w_i \leftarrow \beta w_i$

TABLE 7.1
WEIGHTED-MAJORITY algorithm.

We now show that the number of mistakes committed by the WEIGHTED- MAJORITY algorithm can be bounded in terms of the number of mistakes made by the best prediction algorithm in the voting pool.

Theorem 7.5. Relative mistake bound for WEIGHTED-MAJORITY. Let D be any sequence of training examples, let A be any set of n prediction algorithms, and let k be the minimum number of mistakes made by any algorithm in A for the training sequence D. Then the number of mistakes over D made by the WEIGHTED-MAJORITY algorithm using $\beta = \frac{1}{2}$ is at most

$$2.4(k + \log_2 n)$$

Proof. We prove the theorem by comparing the final weight of the best prediction algorithm to the sum of weights over all algorithms. Let a_j denote an algorithm from A that commits the optimal number k of mistakes. The final weight w_j associated with a_j will be $(\frac{1}{2})^k$, because its initial weight is 1 and it is multiplied by $\frac{1}{2}$ for each mistake. Now consider the sum $W = \sum_{i=1}^{n} w_i$ of the weights associated with all n algorithms in A. W is initially n. For each mistake made by WEIGHTED-MAJORITY, W is reduced to at most $\frac{3}{4}W$. This is the case because the algorithms voting in the weighted majority must hold at least half of the total weight W, and this portion of W will be reduced by a factor of $\frac{1}{2}$. Let M denote the total number of mistakes committed by WEIGHTED-MAJORITY for the training sequence D. Then the final total weight W is at most $n(\frac{3}{4})^M$. Because the final weight w_j cannot be greater than the final total weight, we have

$$\left(\frac{1}{2}\right)^k \leq n\left(\frac{3}{4}\right)^M$$

SUMMARY AND FURTHER READING

- The probably approximately correct (PAC) model considers algorithms that learn target concepts from some concept class C, using training examples drawn at random according to an unknown, but fixed, probability distribution. It requires that the learner probably (with probability at least $[1-\delta]$) learn a hypothesis that is approximately (within error ϵ) correct, given computational effort and training examples that grow only polynomially with $1/\epsilon$, $1/\delta$, the size of the instances, and the size of the target concept.
- Within the setting of the PAC learning model, any consistent learner using a finite hypothesis space H where $C \subseteq H$ will, with probability $(1-\delta)$, output a hypothesis within error ϵ of the target concept, after observing m randomly drawn training examples, as long as

$$m \geq \frac{1}{\epsilon}(\ln(1/\delta) + \ln|H|)$$

This gives a bound on the number of training examples sufficient for successful learning under the PAC model.

- One constraining assumption of the PAC learning model is that the learner knows in advance some restricted concept class C that contains the target concept to be learned. In contrast, the *agnostic learning* model considers the more general setting in which the learner makes no assumption about the class from which the target concept is drawn. Instead, the learner outputs the hypothesis from H that has the least error (possibly nonzero) over the training data. Under this less restrictive agnostic learning model, the learner is assured with probability $(1-\delta)$ to output a hypothesis within error ϵ of the

best possible hypothesis in H, after observing m randomly drawn training examples, provided

$$m \geq \frac{1}{2\epsilon^2}(\ln(1/\delta) + \ln|H|)$$

- The number of training examples required for successful learning is strongly influenced by the complexity of the hypothesis space considered by the learner. One useful measure of the complexity of a hypothesis space H is its Vapnik-Chervonenkis dimension, $VC(H)$. $VC(H)$ is the size of the largest subset of instances that can be shattered (split in all possible ways) by H.
- An alternative upper bound on the number of training examples sufficient for successful learning under the PAC model, stated in terms of $VC(H)$ is

$$m \geq \frac{1}{\epsilon}(4\log_2(2/\delta) + 8VC(H)\log_2(13/\epsilon))$$

A lower bound is

$$m \geq \max\left[\frac{1}{\epsilon}\log(1/\delta), \frac{VC(C)-1}{32\epsilon}\right]$$

- An alternative learning model, called the *mistake bound model*, is used to analyze the number of training examples a learner will misclassify before it exactly learns the target concept. For example, the HALVING algorithm will make at most $\lfloor \log_2 |H| \rfloor$ mistakes before exactly learning any target concept drawn from H. For an arbitrary concept class C, the best worst-case algorithm will make $Opt(C)$ mistakes, where

$$VC(C) \leq Opt(C) \leq \log_2(|C|)$$

- The WEIGHTED-MAJORITY algorithm combines the weighted votes of multiple prediction algorithms to classify new instances. It learns weights for each of these prediction algorithms based on errors made over a sequence of examples. Interestingly, the number of mistakes made by WEIGHTED-MAJORITY can be bounded in terms of the number of mistakes made by the best prediction algorithm in the pool.

Genetic Algorithms:

This chapter covers both genetic algorithms, in which hypotheses are typically described by *bit strings*, and genetic programming, in which hypotheses are described by *computer programs*.

MOTIVATION

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

Evolution is known to be a successful, robust method for adaptation within biological systems.

GAS can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.

Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

GENETIC ALGORITHMS

The problem addressed by GAS is to search a space of candidate hypotheses to identify the best hypothesis. In GAS the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem at hand, called the hypothesis fitness.

Although different implementations of genetic algorithms vary in their details, they typically share the following structure: The algorithm operates by itera- tively updating a pool of hypotheses, called the population. On each iteration, all members of the population are evaluated according to the fitness function. A new population is then generated by probabilistically selecting the most fit individuals from the current population. Some of these selected individuals are carried forward into the next generation population intact. Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

A prototypical genetic algorithm is described in Table 9.1.

GA($Fitness$, $Fitness\_threshold$, p, r, m)

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

Fitness_threshold: A threshold specifying the termination criterion.

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

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

m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do

  Create a new generation, P_S:

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

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

  2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.
  3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
  4. *Update:* $P \leftarrow P_s$.
  5. *Evaluate:* for each h in P, compute $Fitness(h)$

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

TABLE 9.1

A prototypical genetic algorithm. A population containing p hypotheses is maintained. On each iteration, the successor population P_S is formed by probabilistically selecting current hypotheses according to their fitness and by adding new hypotheses. New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered. Typical crossover and mutation operators are defined in a subsequent table.

Representing Hypotheses

Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover. The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule

precondition and postcondition. Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value.

To pick an example, consider the attribute Outlook, which can take on any of the three values Sunny, Overcast, or Rain; consider a second attribute, Wind, that can take on the value Strong or Weak, then:

Rule postconditions (such as $PlayTennis = yes$) can be represented in a similar fashion. Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition. For example, the rule

IF $Wind = Strong$ THEN $PlayTennis = yes$

would be represented by the string

Outlook	*Wind*	*PlayTennis*
111	10	10

where the first three bits describe the "don't care" constraint on *Outlook*, the next two bits describe the constraint on *Wind*, and the final two bits describe the rule postcondition (here we assume *PlayTennis* can take on the values *Yes* or *No*). Note the bit string representing the rule contains a substring for each attribute in the hypothesis space, even if that attribute is not constrained by the rule preconditions. This yields a fixed length bit-string representation for rules, in which substrings at specific locations describe constraints on specific attributes. Given this representation for single rules, we can represent sets of rules by similarly concatenating the bit string representations of the individual rules.

Genetic Operators

The two most common operators are crossover and mutation.

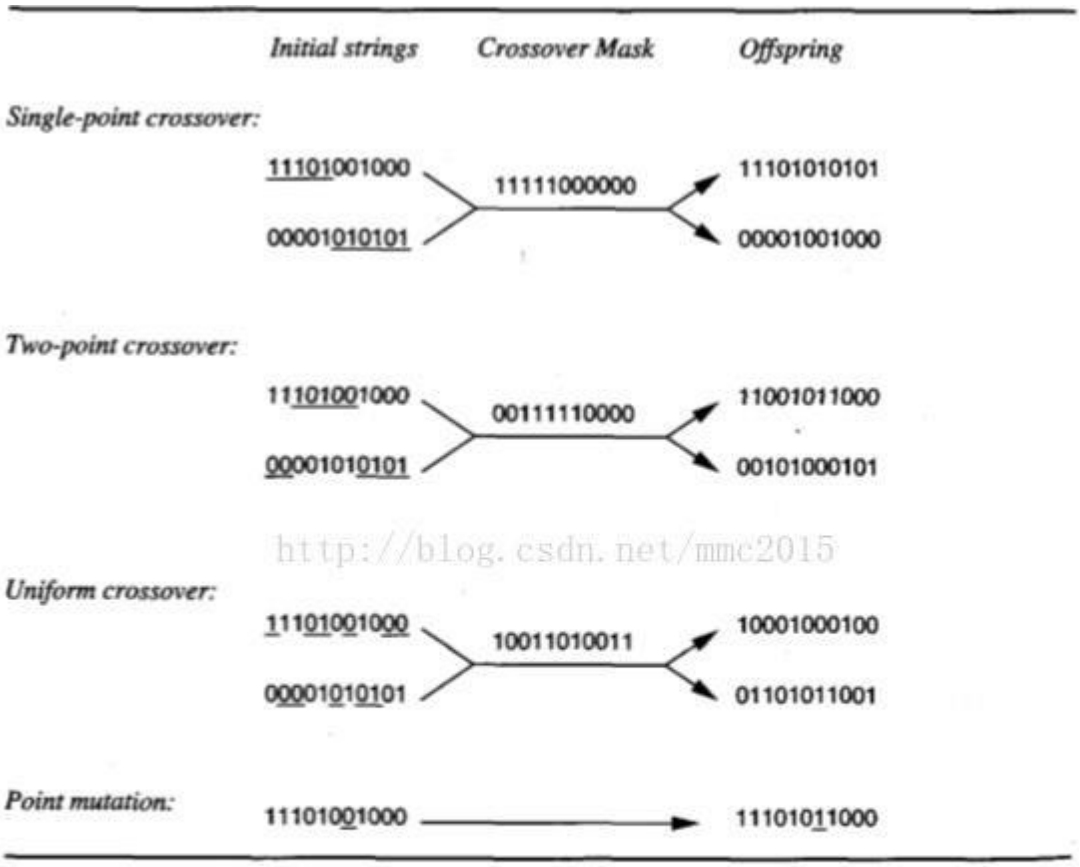


TABLE 9.2
Common operators for genetic algorithms. These operators form offspring of hypotheses represented by bit strings. The crossover operators create two descendants from two parents, using the crossover mask to determine which parent contributes which bits. Mutation creates a single descendant from a single parent by changing the value of a randomly chosen bit.

Fitness Function and Selection

fitness proportionate selection, or roulette wheel selection.

tournament selection

rank selection

AN ILLUSTRATIVE EXAMPLE

Refer。P256-P258。

HYPOTHESIS SPACE SEARCH

The GA search can move much more **abruptly**, replacing a parent hypothesis by an offspring that may be radically different from the parent. And

the GA search is therefore less likely to fall into the same kind of local minima that can plague gradient descent methods.

One practical difficulty in some GA applications is the problem of crowding. Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and 1 very similar individuals take over a large fraction of the population. The negative impact of crowding is that it reduces the diversity of the population, thereby slow- ing further progress by the GA.

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

Population Evolution and the Schema Theorem

It is interesting to ask whether one can **mathematically characterize** the evolution over time of the population within a GA.

The schema theorem of Holland (1975) provides one such characterization. It is based on the concept of schemas, or patterns that describe sets of bit strings. To be precise, a schema is any string composed of Os, Is, and *'s. Each schema represents the set of bit strings containing the indicated 0s and Is, with each "*" interpreted as a "don't care." For example, the schema 0*10 represents the set of bit strings that includes exactly 0010 and 0110.

The schema theorem characterizes the evolution of the population within a GA in terms of **the number of instances representing each schema**. Let m(s, t) denote the number of instances of schema s in the population at time t (i.e., during the tth generation). The schema theorem describes the expected value of m(s, t + 1) in terms of m(s, t) and other properties of the schema, population, and GA algorithm parameters.

Let /f(t) denote the average fitness of **all individuals** in the population at time t and let ^u(s, t) denote the average fitness of instances of **schema s** in the population at time t. then:

$$E[m(s,t+1)] = \frac{\hat{u}(s,t)}{\hat{f}(t)} m(s,t) \qquad (9.3)$$

If we view the GA as performing a virtual parallel search through the space of possible schemas at the same time it performs its explicit parallel search through the space of individuals, then Equation (9.3) indicates that more fit schemas will grow in influence over time.

The schema theorem is perhaps the most widely cited characterization of population evolution within a GA. One way in which it is incomplete is that it fails to consider the (presumably) positive effects of crossover and mutation. Numerous more recent theoretical analyses have been proposed, including analyses based on Markov chain models and on statistical mechanics models. See, for example, Whitley and Vose (1995) and Mitchell (1996).

GENETIC PROGRAMMING

Genetic programming (GP) is a form of evolutionary computation in which the in- dividuals in the evolving population are computer programs rather than bit strings.

Representing Programs

Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given

by its descendant nodes. For example, Figure 9.1 illustrates this tree representation for the function sin(x) + sqrt(x*x + y).

To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, +, -, exponentials),a s well as the terminals (e.g., x, y, constants such as 2). The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives.

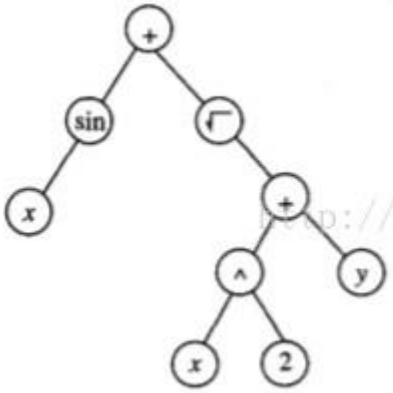


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and mutation. The fitness of a given individual program in the population is typ- ically determined by executing the program on a set of training data.

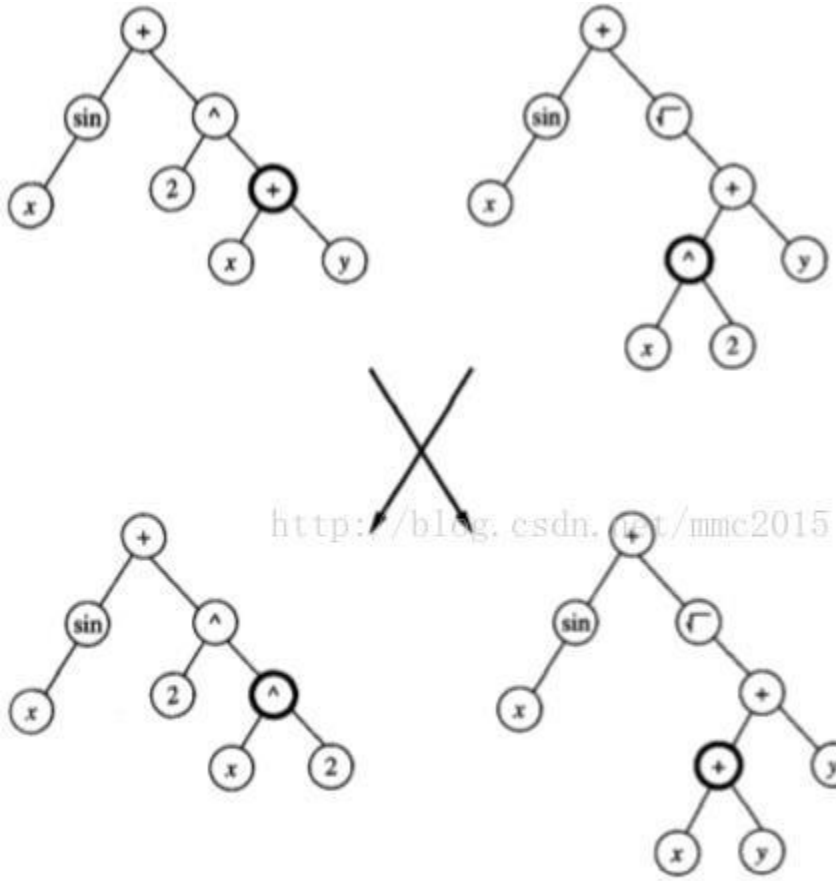


FIGURE 9.2
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

Illustrative Example

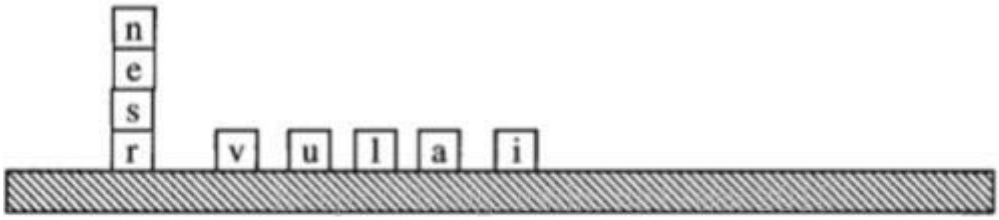


FIGURE 9.3
A block-stacking problem. The task for GP is to discover a program that can transform an arbitrary initial configuration of blocks into a stack that spells the word "universal." A set of 166 such initial configurations was provided to evaluate fitness of candidate programs (after Koza 1992).

As in most GP applications, the choice of problem representation has a significant impact on the ease of solving the problem. In Koza's formulation,

the primitive functions used to compose programs for this task include the following three terminal arguments:

CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.

TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.

NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal" or F if no more blocks are needed.

Remarks on Genetic Programming

Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce intriguing results in a number of applications.

In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function. For this reason, an active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions, thereby allowing the system to dynamically alter the primitives from which it constructs individuals. See, for example, Koza (1994).

MODELS OF EVOLUTION AND LEARNING

One interesting question regarding evolutionary systems is "What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?'

Lamarckian Evolution

Baldwin Effect

PARALLELIZING GENETIC ALGORITHMS

GAS are naturally suited to parallel implementation, and a number of approaches to parallelization have been explored. Coarse grain approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called demes. Each deme is assigned to a different computational node, and a standard GA search is performed at each node. Communication

and cross-fertilization*between demes* occurs on a less frequent basis than *within demes*. In contrast to coarse-grained parallel implementations of GAS, fine-grained implementations typically assign one processor per individual in the population. Recombination then takes place among neighboring individuals.

Instance Based Learning:

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.

One key difference between these approaches and the methods discussed in other chapters is that instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified. In fact, many techniques construct only a local approximation to the target function that applies in the neighborhood of the new query instance, and never construct an approximation designed to perform well over the entire instance space. This has significant advantages when the target function is very complex, but can still be described by a collection of less complex local approximations.

k-NEAREST NEIGHBOR LEARNING

Euclidean distance. More precisely, let an arbitrary instance x be described by the feature vector

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

where $a_r(x)$ denotes the value of the rth 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 : X -> V, where V is the finite set {v1, . . . vs}. The k-NEARESNTE IGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1.

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.

TABLE 8.1
The k-NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \Re^n \rightarrow V$.

The k-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have the algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function $f : \Re^n \rightarrow \Re$ we replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{8.1}$$

Distance-Weighted NEAREST NEIGHBOR Algorithm

One obvious 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 xq, giving greater weight to closer neighbors.

This can be accomplished by replacing the final line of the algorithm by

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \tag{8.2}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2} \tag{8.3}$$

To accommodate the case where the query point x_q exactly matches one of the training instances x_i and the denominator $d(x_q, x_i)^2$ is therefore zero, we assign $\hat{f}(x_q)$ to be $f(x_i)$ in this case. If there are several such training examples, we assign the majority classification among them.

We can distance-weight the instances for real-valued target functions in a similar fashion, replacing the final line of the algorithm in this case by

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{8.4}$$

where w_i is as defined in Equation (8.3). Note the denominator in Equation (8.4) is a constant that normalizes the contributions of the various weights (e.g., it assures that if $f(x_i) = c$ for all training examples, then $\hat{f}(x_q) \leftarrow c$ as well).

Remarks on k-NEARESTN EIGHBOR Algorithm

It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data.

The inductive bias corresponds to an assumption that the classification of an instance x, will be most similar to the classification of other instances that are nearby in Euclidean distance.

One disadvantage of instance-based approaches is that 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.

A second disadvantage to 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. This difficulty, which arises when many irrelevant attributes are

present, is sometimes referred to as the *curse of dimensionality*. Nearest-neighbor approaches are especially sensitive to this problem.

One interesting approach to overcoming this problem is to weight each attribute differently when calculating the distance between two instances. This corresponds to stretching the axes in the Euclidean space, shortening the axes that correspond to less relevant attributes, and lengthening the axes that correspond to more relevant attributes. The amount by which each axis should be stretched can be determined automatically using a cross-validation approach.

An even more drastic alternative is to completely eliminate the least relevant attributes from the instance space. This is equivalent to setting some of the zi scaling factors to zero.

A Note on Terminolog

Much of the literature on nearest-neighbor methods and weighted local regression uses a terminology that has arisen from the field of statistical pattern recognition. In reading that literature, it is useful to know the following terms:

Regression means *approximating* a real-valued target function.

Residual is the *error* ^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 wi = K(d(xi, x,)).

LOCALLY WEIGHTED REGRESSION

The nearest-neighbor approaches described in the previous section can be thought of as approximating the target function f (x) *at the single query point* x = xq. Locally weighted regression is a generalization of this approach. It constructs an explicit approximation to f *over a local region* surrounding xq.

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. The description of $\hat{f}$ may then be deleted, because a different local approximation will be calculated for each distinct query instance.

Locally Weighted Linear Regression

Let us consider the case of locally weighted regression in which the target function f is approximated near x_q using a linear function of the form

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

As before, $a_i(x)$ denotes the value of the ith attribute of the instance x.

Recall that in Chapter 4 we discussed methods such as gradient descent to find the coefficients w0 . . . wn to minimize the error in fitting such linear functions to a given set of training examples. How shall we 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, as following:

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

If we choose criterion three above and rederive the gradient descent rule using the same style of argument as in Chapter 4, we obtain the following training rule (see Exercise 8.1):

$$\Delta w_j = \eta \sum_{x \in \text{k nearest nbrs of } x_q} K(d(x_q, x)) \; (f(x) - \hat{f}(x)) \; a_j(x) \qquad (8.7)$$

In fact, if we are fitting a linear function to a fixed set of training examples, then methods much more efficient than gradient descent are available to directly solve for the desired coefficients w0 . . . wn. Atkeson et al. (1997a) and Bishop (1995) survey several such methods.

RADIAL BASIS FUNCTION

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 (Powell 1987; Broomhead and Lowe 1988; Moody and Darken 1989). In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \tag{8.8}$$

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 $d(x_u, x)$ increases. Here k is a user-provided constant that specifies the number of kernel functions to be included. Even though $\hat{f}(x)$ 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. It is common to choose each function $K_u(d(x_u, x))$ to be a Gaussian function (see Table 5.4) centered at the point x_u with some variance σ_u^2.

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

We will restrict our discussion here to this common Gaussian kernel function.

The function given by Equation (8.8) 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. An example radial basis function (RBF) network is illustrated in Figure 8.2.

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process. 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 $K_u(d(x_u, x))$. Second, the weights w_u are trained to maximize the fit of the network to the training data, using the global error criterion given by Equation (8.5). Because the kernel functions are held fixed during this second stage, the linear weight values w_u 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 (xi, f (xi)), centering this Gaussian at the point xi. Each of these kernels may be assigned the same width &2. A second approach is to choose a set of kernel functions: The set of kernel functions may be distributed with centers spaced uniformly throughout the instance space X. Alternatively, we may wish to distribute the centers nonuniformly, especially if the instances themselves are

found to be distributed nonuniformly over X. Alternatively, we may identify prototypical clusters of instances, then add a kernel function centered at each cluster.

To summarize, 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 center 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. This follows from the fact that the input layer and the output layer of an RBF are trained separately.

CASE-BASED REASONING

Instance-based methods such as k-NEAREST NEIGHBOaRn d locally weighted regression share three key properties. First, they are lazy learning methods in that they defer the decision of how to generalize beyond the training data until a new query instance is observed. Second, they classify new query instances by analyzing similar instances while ignoring instances that are very different from the query. Third, they represent instances as **real-valued points** in an n-dimensional Euclidean space. Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third. In CBR, instances are typically represented using **more rich symbolic descriptions**, and the methods used to retrieve similar instances are correspondingly more elaborate.

Let us consider a prototypical example of a case-based reasoning system to ground our discussion. The CADET system (Sycara et al. 1992) employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets.

Given this functional specification for the new design problem, 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.

It is instructive to examine the correspondence between the problem setting of CADET and the general setting for instance-based methods such as k-NEAREST NEIGHBORIn. CADET each stored training example describes a function graph along with the structure that implements it. New queries correspond to new function graphs. Thus, we can map the CADET problem into our standard notation by defining the space of instances X to be the space ofall function graphs. The target function f maps function graphs to the structures that implement them. Each stored training example (x, f (x)) is a pair that describes some function graph x and the structure f (x) that implements x. The system must learn from the training example cases to output the structuref (xq) that successfully implements the input function graph query xq.

REMARKS ON LAZY AND EAGER LEARNING

In this chapter we considered three lazy learning methods: the k-NEAREST NEIGHBOR algorithm, locally weighted regression, and case-based reasoning. We call these methods lazy because they defer the decision of how to generalize beyond the training data until each new query instance is encountered. We also discussed one eager learning method: the method for learning radial basis function networks. We call this method eager because it generalizes beyond the training data before observing the new query, committing at training time to the network structure and weights that define its approximation to the target function. In this same sense, every other algorithm discussed elsewhere in this book (e.g., BACKPROPAGATION, C4.5) is an eager learning algorithm.

Are there important differences in what can be achieved by lazy versus eager learning? Differences in computation time and differences in the classifications(or inductive bias) produced for new queries and differences in

generalization accuracy(related to the distinction between global and local approximations to the target function). Inductive bias: Lazy methods may consider the query instance x, when deciding how to generalize beyond the training data D; Eager methods cannot. By the time they observe the query instance x, they have already chosen their (global) approximation to the target function.

Lazy methods have the option of selecting a different hypothesis or local approximation to the target function for each query instance. Eager methods using the same hypothesis space are more restricted because they must commit to a single hypothesis that covers the entire instance space. Eager methods can, of course, employ hypothesis spaces that combine multiple local approximations, as in RBF networks. However, even these combined local approximations do not give eager methods the full ability of lazy methods to customize to unknown future query instances.

SUMMARY AND FURTHER READING

Instance-based learning methods differ from other approaches to function approximation because they delay processing of training examples until they must label a new query instance. As a result, they need not form an explicit hypothesis of the entire target function over the entire instance space, independent of the query instance. Instead, they may form a different local approximation to the target function for each query instance.

Advantages of instance-based methods include the ability to model complex target functions by a collection of less complex local approximations and the fact that information present in the training examples is never lost (because the examples themselves are stored explicitly). The main practical difficulties include efficiency of labeling new instances (all processing is done at query time rather than in advance), difficulties in determining an appropriate distance metric for retrieving "related" instances (especially when examples are represented by complex symbolic descriptions), and the negative impact of irrelevant features on the distance metric.

k-NEARESNTE IGHBOR is an instance-based algorithm for approximating real-valued or discrete-valued target functions, assuming instances correspond to points in an n-dimensional Euclidean space. The target function value for a new query is estimated from the known values of the k nearest training examples.

Locally weighted regression methods are a generalization of k-NEAREST NEIGHBOiRn which an explicit local approximation to the target function is constructed for each query instance. The local approximation to the target function may be based on a variety of functional forms such as constant, linear, or quadratic functions or on spatially localized kernel functions.

Radial basis function (RBF) networks are a type of artificial neural network constructed from spatially localized kernel functions. These can be seen as a blend of instance-based approaches (spatially localized influence of each kernel function) and neural network approaches (a global approximation to the target function is formed at training time rather than a local approximation at query time). Radial basis function networks have been used successfully in applications such as interpreting visual scenes, in which the assumption of spatially local influences is well-justified.

Case-based reasoning is an instance-based approach in which instances are represented by complex logical descriptions rather than points in a Euclidean space. Given these complex symbolic descriptions of instances, a rich variety of methods have been proposed for mapping from the training examples to target function values for new instances. Case-based reasoning methods have been used in applications such as modeling legal reasoning and for guiding searches in complex manufacturing and transportation planning problems.

Unit-4

Analytical Learning:

Inductive learning methods such as neural network and decision tree learning require a certain number of training examples to achieve a given level of generalization accuracy, as reflected in the theoretical bounds and experimental results discussed in earlier chapters. Analytical learning uses prior knowledge and deductive reasoning to augment the information provided by the training examples, so that it is not subject to these same bounds. This chapter considers an analytical learning method called explanation-based learning (EBL). In explanation-based learning, prior knowledge is used to analyze, or explain, how each observed training example satisfies the target concept. This explanation is then used to distinguish the relevant features of the training example from the irrelevant, so that examples can be generalized based on logical rather than statistical reasoning.

INTRODUCTION

Previous chapters have considered a variety of inductive learning methods: that is, methods that generalize from observed training examples by identifying features that empirically distinguish positive from negative training examples. Decision tree learning, neural network learning, inductive logic programming, and genetic algorithms are all examples of inductive methods that operate in this fashion. The key practical limit on these inductive learners is that they perform poorly when insufficient data is available. In fact, as discussed in Chapter 7, theoretical analysis shows that there are fundamental bounds on the accuracy that can be achieved when learning inductively from a given number of training examples.

Can we develop learning methods that are not subject to these fundamental bounds on learning accuracy imposed by the amount of training data available? Yes, if we are willing to reconsider the formulation of the learning problem itself. One way is to develop learning algorithms that accept

explicit prior knowledge as an input, in addition to the input training data. Explanation-based learning is one such approach. It uses prior knowledge to analyze, or explain, each training example in order to infer which example features are relevant to the target function and which are irrelevant. These explanations enable it to generalize more accurately than inductive systems that rely on the data alone.

As we saw in the previous chapter, inductive logic programming systems such as CIGOL also use prior background knowledge to guide learning. However, they use their background knowledge to infer features that augment the input descriptions of instances, thereby **increasing** the complexity of the hypothesis space to be searched. In contrast, explanation-based learning uses prior knowledge to **reduce** the complexity of the hypothesis space to be searched, thereby reducing sample complexity and improving generalization accuracy of the learner.

Inductive and Analytical Learning Problems

The essential difference between analytical and inductive learning methods is that they assume two different formulations of the learning problem:

- In inductive learning, the learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D = \{\langle x_1, f(x_1)\rangle, \ldots \langle x_n, f(x_n)\rangle\}$ where $f(x_i)$ is the target value for the instance x_i. The desired output of the learner is a hypothesis h from H that is consistent with these training examples.
- In analytical learning, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input: A *domain theory* B consisting of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B.

The full definition of this analytical learning task is given in Table 11 .1

Given:

- Instance space X: Each instance describes a pair of objects represented by the predicates $Type$, $Color$, $Volume$, $Owner$, $Material$, $Density$, and On.
- Hypothesis space H: Each hypothesis is a set of Horn clause rules. The head of each Horn clause is a literal containing the target predicate *SafeToStack*. The body of each Horn clause is a conjunction of literals based on the same predicates used to describe the instances, as well as the predicates $LessThan$, $Equal$, $GreaterThan$, and the functions $plus$, $minus$, and $times$. For example, the following Horn clause is in the hypothesis space:

$$SafeToStack(x, y) \leftarrow Volume(x, vx) \wedge Volume(y, vy) \wedge LessThan(vx, vy)$$

- Target concept: *SafeToStack(x,y)*
- Training Examples: A typical positive example, $SafeToStack(Obj1, Obj2)$, is shown below:

$On(Obj1, Obj2)$
$Type(Obj1, Box)$
$Type(Obj2, Endtable)$
$Color(Obj1, Red)$
$Color(Obj2, Blue)$
$Volume(Obj1, 2)$
$Owner(Obj1, Fred)$
$Owner(Obj2, Louise)$
$Density(Obj1, 0.3)$
$Material(Obj1, Cardboard)$
$Material(Obj2, Wood)$

- Domain Theory B:

$SafeToStack(x, y) \leftarrow \neg Fragile(y)$
$SafeToStack(x, y) \leftarrow Lighter(x, y)$
$Lighter(x, y) \leftarrow Weight(x, wx) \wedge Weight(y, wy) \wedge LessThan(wx, wy)$
$Weight(x, w) \leftarrow Volume(x, v) \wedge Density(x, d) \wedge Equal(w, times(v, d))$
$Weight(x, 5) \leftarrow Type(x, Endtable)$
$Fragile(x) \leftarrow Material(x, Glass)$
...

Determine:

- A hypothesis from H consistent with the training examples and domain theory.

TABLE 11.1
An analytical learning problem: *SafeToStack(x,y)*.

LEARNING WITH PERFECT DOMAIN THEORIES: PROLOG-EBG

As stated earlier, in this chapter we consider explanation-based learning from domain theories that are perfect, that is, domain theories that are **correct and complete.** A domain theory is said to be correct if each of its assertions is a truthful statement about the world. A domain theory is said to be complete with respect to a given target concept and instance space, if the domain theory covers every positive example in the instance space.

PROLOG-EBG(*TargetConcept*, *TrainingExamples*, *DomainTheory*)

- *LearnedRules* ← {}
- *Pos* ← the positive examples from *TrainingExamples*
- for each *PositiveExample* in *Pos* that is not covered by *LearnedRules*, do
  1. *Explain:*
     - *Explanation* ← an explanation (proof) in terms of the *DomainTheory* that *PositiveExample* satisfies the *TargetConcept*
  2. *Analyze:*
     - *SufficientConditions* ← the most general set of features of *PositiveExample* sufficient to satisfy the *TargetConcept* according to the *Explanation*.
  3. *Refine:*
     - *LearnedRules* ← *LearnedRules* + *NewHornClause*, where *NewHornClause* is of the form

$$TargetConcept \leftarrow SufficientConditions$$

- Return *LearnedRules*

TABLE 11.2
The explanation-based learning algorithm PROLOG-EBG. For each positive example that is not yet covered by the set of learned Horn clauses (*LearnedRules*), a new Horn clause is created. This new Horn clause is created by (1) explaining the training example in terms of the domain theory, (2) analyzing this explanation to determine the relevant features of the example, then (3) constructing a new Horn clause that concludes the target concept when this set of features is satisfied.

EXPLAIN THE TRAINING EXAMPLE:

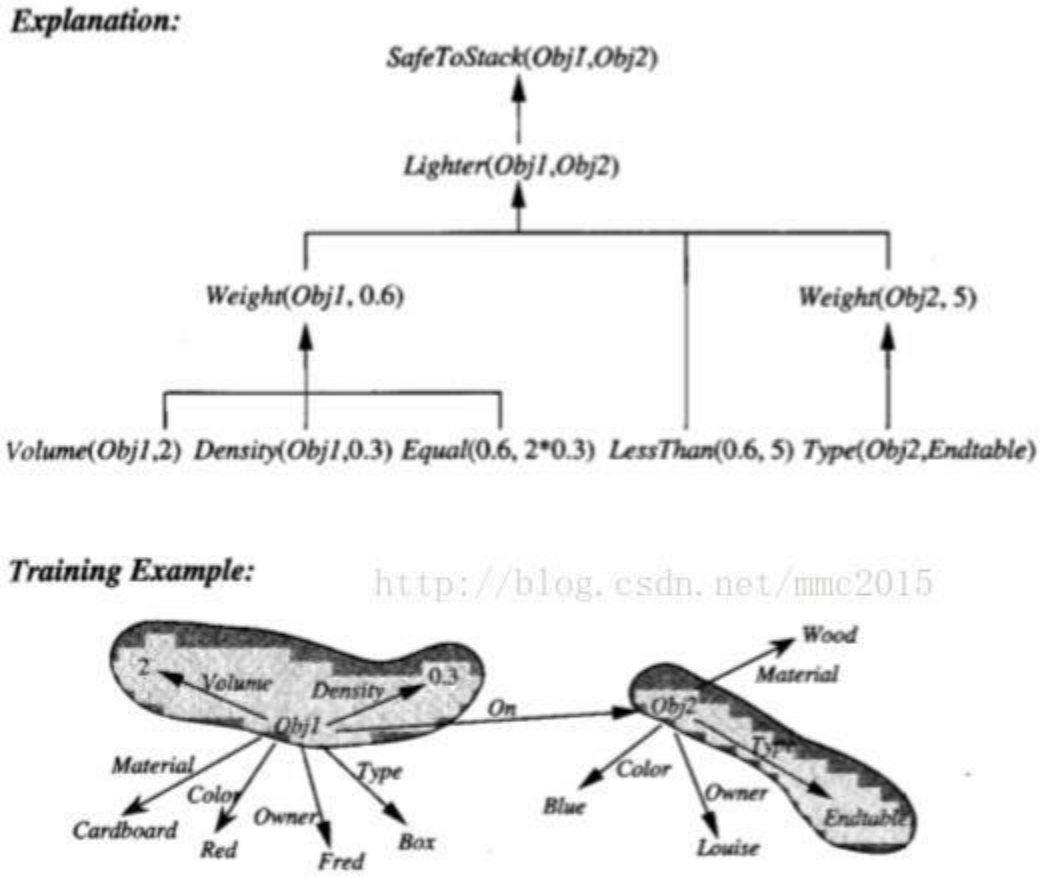


FIGURE 11.2
Explanation of a training example. The network at the bottom depicts graphically the training example *SafeToStack*(*Obj1*, *Obj2*) described in Table 11.1. The top portion of the figure depicts the explanation of how this example satisfies the target concept, *SafeToStack*. The shaded region of the training example indicates the example attributes used in the explanation. The other, irrelevant, example attributes will be dropped from the generalized hypothesis formed from this analysis.

ANALYZE THE EXPLANATION:

By collecting just the features mentioned in the leaf nodes of the explanation in Figure 11.2 and substituting variables x and y for Objl and Obj2, we can form a general rule that is justified by the domain theory: SafeToStack(x, y) <-- Volume(x, 2) ^ Density(x, 0.3) ^ Type(y, Endtable), The body of the above rule includes each leaf node in the proof tree, except for the leaf nodes "Equal(0.6, times(2,0.3)" and "LessThan(0.6,5)." We omit these two because they are by definition always satisfied, independent of x and y.

Although this explanation was formed to cover the observed training example, the same explanation will apply to any instance that matches this general rule. The above rule constitutes a significant generalization of the training example, because it omits many properties of the example (e.g., the Color of the two objects) that are irrelevant to the target concept. **However, an even more general rule can be obtained by more careful analysis of the explanation**. PROLOG-EBG computes the most general rule that can be justified by the explanation, by computing the weakest preimage of the explanation, defined as follows: Definition: The **weakest preimage** of a conclusion C with respect to a proof P is the most general set of initial assertions A, such that A entails C according to P.

For example, the weakest preimage of the target concept *SafeToStack(x,y)*, with respect to the explanation from Table 11.1, is given by the body of the following rule. This is the most general rule that can be justified by the explanation of Figure 11.2:

$$SafeToStack(x, y) \leftarrow Volume(x, vx) \wedge Density(x, dx) \wedge Equal(wx, times(vx, dx)) \wedge LessThan(wx, 5) \wedge Type(y, Endtable)$$

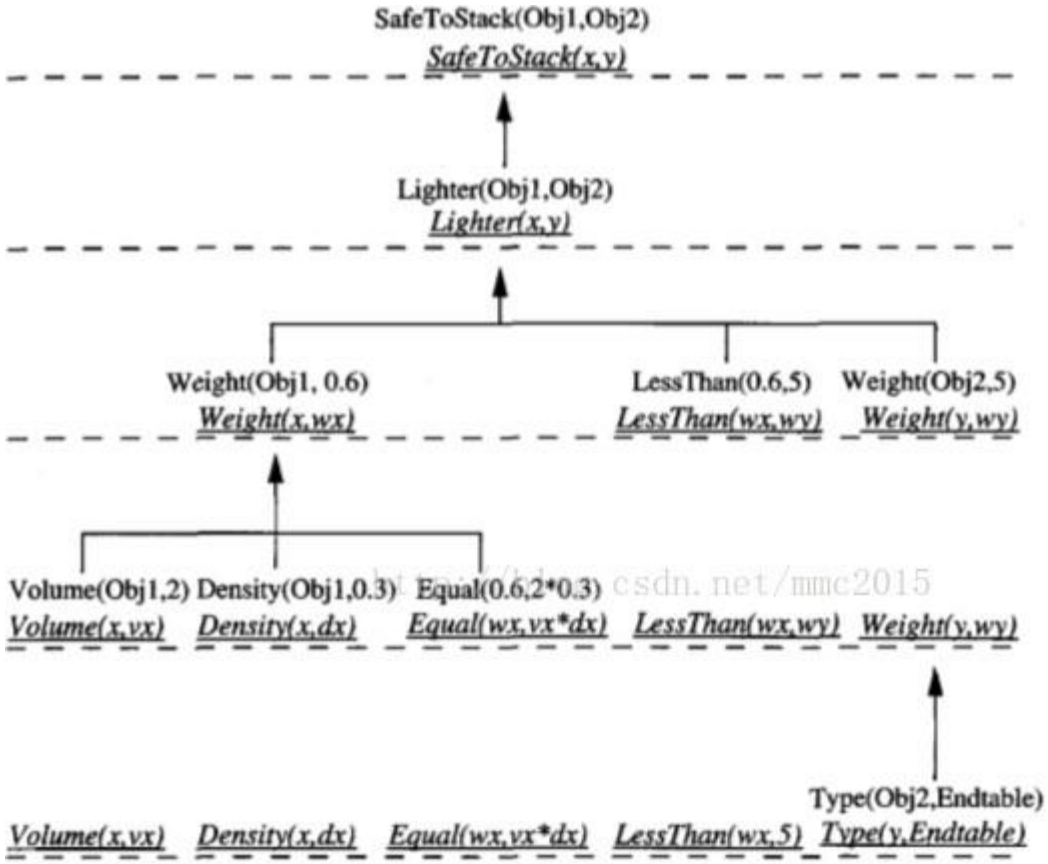


FIGURE 11.3
Computing the weakest preimage of *SafeToStack*(*Obj*1, *Obj*2) with respect to the explanation. The target concept is regressed from the root (conclusion) of the explanation, down to the leaves. At each step (indicated by the dashed lines) the current frontier set of literals (underlined in italics) is regressed backward over one rule in the explanation. When this process is completed, the conjunction of resulting literals constitutes the weakest preimage of the target concept with respect to the explanation. This weakest preimage is shown by the italicized literals at the bottom of the figure.

The heart of the regression procedure is the algorithm that at each step regresses the current frontier of expressions through a single Horn clause from the domain theory. This algorithm is described and illustrated in Table 11.3. The final Horn clause rule output by PROLOG-EBGis formulated as follows: The clause body is defined to be the weakest preconditions calculated by the above procedure. The clause head is the target concept itself, with each substitution from each regression step (i.e., the substitution Oh[in Table 11.3) applied to it.

REGRESS(*Frontier, Rule, Literal*, θ_{hi})

Frontier: Set of literals to be regressed through Rule

Rule: A Horn clause

Literal: A literal in Frontier *that is inferred by Rule in the explanation*

θ_{hi}: *The substitution that unifies the head of Rule to the corresponding literal in the explanation*

Returns the set of literals forming the weakest preimage of Frontier with respect to Rule

- *head* ← *head* of *Rule*
- *body* ← *body* of *Rule*
- θ_{hl} ← the most general unifier of *head* with *Literal* such that there exists a substitution θ_{li} for which

$$\theta_{li}(\theta_{hl}(head)) = \theta_{hi}(head)$$

- Return $\theta_{hl}(Frontier - head + body)$

Example (the bottommost regression step in Figure 11.3):

REGRESS(*Frontier, Rule, Literal*, θ_{hi}) where

Frontier = {*Volume*(*x*, *vs*), *Density*(*x*, *dx*), *Equal*(*wx*, *times*(*vx*, *dx*)), *LessThan*(*wx*, *wy*), *Weight*(*y*, *wy*)}

Rule = *Weight*(*z*, 5) ← *Type*(*z*, *Endtable*)

Literal = *Weight*(*y*, *wy*)

θ_{hi} = {*z*/*Obj*2}

- *head* ← *Weight*(*z*, 5)
- *body* ← *Type*(*z*, *Endtable*)
- θ_{hl} ← {*z*/*y*, *wy*/5}, where θ_{li} = {*y*/*Obj*2}
- Return {*Volume*(*x*, *vs*), *Density*(*x*, *dx*), *Equal*(*wx*, *times*(*vx*, *dx*)), *LessThan*(*wx*, 5), *Type*(*y*, *Endtable*)}

TABLE 11.3

Algorithm for regressing a set of literals through a single Horn clause. The set of literals given by *Frontier* is regressed through *Rule*. *Literal* is the member of *Frontier* inferred by *Rule* in the explanation. The substitution θ_{hi} gives the binding of variables from the head of *Rule* to the corresponding literal in the explanation. The algorithm first computes a substitution θ_{hl} that unifies the *Rule* head to *Literal*, in a way that is consistent with the substitution θ_{hi}. It then applies this substitution θ_{hl} to construct the preimage of *Frontier* with respect to *Rule*. The symbols "+" and "–" in the algorithm denote set union and set difference. The notation {*z*/*y*} denotes the substitution of *y* in place of *z*. An example trace is given.

REMARKS ON EXPLANATION-BASED LEARNING

Discovering New Feature

One interesting capability of PROLOG-EBGis its ability to formulate new features that are not explicit in the description of the training examples, but that are needed to describe the general rule underlying the training example.

Notice this learned "feature" is similar in kind to the types of features represented by the hidden units of neural networks; that is, this feature is one of a very large set of potential features that can be computed from the available instance attributes.

Deductive Learning

In its pure form, PROLOG-EBG is a deductive, rather than inductive, learning process. That is, by calculating the weakest preimage of the explanation it produces a hypothesis h that follows deductively from the domain theory B, while covering the training data D. To be more precise, PROLOG-EBG outputs a hypothesis h that satisfies the following two constraints:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (h \wedge x_i) \vdash f(x_i) \tag{11.1}$$

$$D \wedge B \vdash h \tag{11.2}$$

Using similar notation, we can state the type of knowledge that is required by PROLOG-EBG for its domain theory. In particular, PROLOG-EBG assumes the domain theory B entails the classifications of the instances in the training data:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (B \wedge x_i) \vdash f(x_i) \tag{11.3}$$

This constraint on the domain theory B assures that an explanation can be constructed for each positive example.

It is interesting to compare the PROLOG-EBG learning setting to the setting for inductive logic programming (ILP) discussed in Chapter 10. ILP is an inductive learning system, whereas PROLOG-EBG is deductive.

ILP systems output a hypothesis h that satisfies the following constraint:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (B' \wedge h \wedge x_i) \vdash f(x_i)$$

Note the relationship between this expression and the constraints on h imposed by PROLOG-EBG (given by Equations (11.1) and (11.2)). This ILP constraint on h is a weakened form of the constraint given by Equation (11.1)—the ILP constraint requires only that $(B' \wedge h \wedge x_i) \vdash f(x_i)$, whereas the PROLOG-EBG constraint requires the more strict $(h \wedge x_i) \vdash f(x_i)$. Note also that ILP imposes no constraint corresponding to the PROLOG-EBG constraint of Equation (11.2).

Inductive Bias in Explanation-Based Learning

Approximate inductive bias of PROLOG-EBG: The domain theory B, plus a preference for small sets of maximally general Horn clauses.

Knowledge Level Learning

To summarize, this example illustrates a situation where $B \not\vdash h$, but where $B \wedge D \vdash h$. The learned hypothesis in this case entails predictions that are not entailed by the domain theory alone. The phrase *knowledge-level learning* is sometimes used to refer to this type of learning, in which the learned hypothesis entails predictions that go beyond those entailed by the domain theory. The set of all predictions entailed by a set of assertions Y is often called the *deductive closure* of Y. The key distinction here is that in knowledge-level learning the deductive closure of B is a proper subset of the deductive closure of $B + h$.

deductive closure:

EXPLANATION-BASED LEARNING OF SEARCH CONTROL KNOWLEDGE:

As noted above, the practical applicability of the PROLOG-EBG algorithm is restricted by its requirement that the domain theory be correct and complete. One important class of learning problems where this requirement is easily satisfied is learning to speed up complex search programs. In fact, the largest scale attempts to apply explanation-based learning have addressed the problem of learning to control search, or what is sometimes called "speedup" learning.

In such problems the definitions of the legal search operators, together with the definition of the search objective, provide a complete and correct domain theory for learning search control knowledge.

An example of a rule learned by PRODIGYfo r this target concept in a simple block-stacking problem domain is

IF	One subgoal to be solved is $On(x, y)$, and One subgoal to be solved is $On(y, z)$
THEN	Solve the subgoal $On(y, z)$ before $On(x, y)$

In fact, there are significant practical problems with applying EBL to learning search control. First, in many cases the number of control rules that must be learned is very large (e.g., many thousands of rules). As the system learns more and more control rules to improve its search, it must pay a larger

and larger cost at each step to match this set of rules against the current search state. Note this problem is not specific to explanation-based learning; it will occur for any system that represents its learned knowledge by a growing set of rules. A second practical problem with applying explanation-based learning to learning search control is that in many cases it is intractable even to construct the explanations for the desired target concept.

SUMMARY AND FURTHER READING

In contrast to purely inductive learning methods that seek a hypothesis to fit the training data, purely analytical learning methods seek a hypothesis that fits the learner's prior knowledge and covers the training examples. Humans often make use of prior knowledge to guide the formation of new hypotheses. This chapter examines purely analytical learning methods. The next chapter examines combined inductive-analytical learning.

Explanation-based learning is a form of analytical learning in which the learner processes each novel training example by (1) explaining the observed target value for this example in terms of the domain theory, (2) analyzing this explanation to determine the general conditions under which the explanation holds, and (3) refining its hypothesis to incorporate these general conditions.

PROLOG-EBG is an explanation-based learning algorithm that uses first-order Horn clauses to represent both its domain theory and its learned hypotheses. In PROLOG-EBG an explanation is a PROLOG proof, and the hypothesis extracted from the explanation is the weakest preimage of this proof. As a result, the hypotheses output by PROLOG-EBGfo llow deductively from its domain theory.

Analytical learning methods such as PROLOG-EBG construct useful intermediate features as a side effect of analyzing individual training examples. This analytical approach to feature generation complements the statistically based generation of intermediate features (eg., hidden unit features) in inductive methods such as BACKPROPAGATION.

Although PROLOG-EBG does not produce hypotheses that extend the deductive closure of its domain theory, other deductive learning procedures can. For example, a domain theory containing determination assertions (e.g., "nationality determines language") can be used together with observed data to deductively infer hypotheses that go beyond the deductive closure of the domain theory.

One important class of problems for which a correct and complete domain theory can be found is the class of large state-space search problems. Systems such as PRODIGY and SOAR have demonstrated the utility of explanation-based learning methods for automatically acquiring effective search control knowledge that speeds up problem solving in subsequent cases.

Despite the apparent usefulness of explanation-based learning methods in humans, purely deductive implementations such as PROLOG-EBG suffer the disadvantage that the output hypothesis is only as correct as the domain theory. In the next chapter we examine approaches that combine inductive and analytical learning methods in order to learn effectively from imperfect domain theories and limited training data.

Learning Sets of Rules:

One of the most expressive and human readable representations for learned hypothe- ses is sets of if-then rules. This chapter explores several algorithms for learning such sets of rules.

INTRODUCTION

As shown in Chapter 3, one way to learn sets of rules is to first learn a decision tree, then translate the tree into an equivalent set of rules-one rule for each leaf node in the tree. A second method, illustrated in Chapter 9, is to use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space. In this chapter we explore a variety of algorithms that directly learn rule sets and that differ from these algorithms in two key respects. First, they are designed to learn sets of first-order rulesthat contain variables. This is significant because first-order rules are much more expressive than propositional rules. Second, the algorithms

discussed here use sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules.

In this chapter we begin by considering algorithms that learn sets of propositional rules; that is, rules without variables. Algorithms for searching the hypothesis space to learn disjunctive sets of rules are most easily understood in this setting. We then consider extensions of these algorithms to learn first-order rules. Two general approaches to inductive logic programming are then considered, and the fundamental relationship between inductive and deductive inference is explored.

SEQUENTIAL COVERING ALGORITHMS

Here we consider a family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process. Such algorithms are called sequential covering algorithms. A prototypical sequential covering algorithm is described in Table 10.1.

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

- *Learned_rules* ← {}
- *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- while PERFORMANCE(*Rule*, *Examples*) > *Threshold*, do
  - *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* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

TABLE 10.1
The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

This sequential covering algorithm is one of the most widespread approaches to learning disjunctive sets of rules. It reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned. Because it performs a greedy search, formulating a sequence of rules without backtracking, it is not guaranteed to find the smallest or best set of rules that cover the training examples.

General to Specific Beam Search

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

Returns a single rule that covers some of the Examples. Conducts a general_to_specific greedy beam search for the best rule, guided by the PERFORMANCE *metric.*

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

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

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

TABLE 10.2
One implementation for LEARN-ONE-RULE is a general-to-specific beam search. The frontier of current hypotheses is represented by the variable *Candidate_hypotheses*. This algorithm is similar to that used by the CN2 program described by Clark and Niblett (1989).

LEARNING RULE SETS: SUMMARY

This section considers several key dimensionsin the design space of such rule learning algorithms.

First, sequential covering algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples. In contrast, decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. We might, therefore, call algorithms such as ID3 simultaneous covering algorithms, in contrast to sequential covering algorithms such as CN2. Which

should we prefer? The key difference occurs in the choice made at the most primitive step in the search. At each search step ID3 chooses among alternative **attributes** by comparing the **partitions** of the data they generate. In contrast, CN2 chooses among alternative **attribute-value pairs**, by comparing the **subsets** of data they cover.

A second dimension along which approaches vary is the direction of the search in LEARN-ONE-RUILn E. the algorithm described above, the search is from general to specijic hypotheses. Other algorithms we have discussed (e.g., FIND-S from Chapter 2) search from specijic to general.

A third dimension is whether the LEARN-ONE-RULE search is a generate then test search through the syntactically legal hypotheses, as it is in our suggested implementation, or whether it is example-driven so that individual training examples constrain the generation of hypotheses. One important advantage of the generate and test approach is that each choice in the search is based on the hypothesis performance over many examples, so that the impact of noisy data is minimized. In contrast, example-driven algorithms that refine the hypothesis based on individual examples are more easily misled by a single noisy training example and are therefore less robust to errors in the training data.

A fourth dimension is whether and how rules are post-pruned.

A final dimension is the particular definition of rule PERFORMANCE used to guide the search in LEARN-ONE-RULE.

Relative frequency: Nc / N

m-estimate of accuracy: (Nc + M*p) / (N + M)

(negative of) Entropy.

LEARNING FIRST-ORDER RULES

In the previous sections we discussed algorithms for learning sets of propositional (i.e., variable-free) rules. In this section, we consider learning **rules that contain variables-in particular, learning first-order Horn** theories. Our motivation for considering such rules is that they are much more expressive than propositional rules.

First-Order Horn Clauses

The problem is that propositional representations offer no general way to describe the essential relations among the values of the attributes.

First-order Horn clauses may also refer to variables in the preconditions that do not occur in the postconditions.

It is also possible to use the same predicates in the rule postconditions and preconditions, enabling the description of recursive rules.

Terminology

- Every well-formed expression is composed of *constants* (e.g., $Mary$, 23, or Joe), *variables* (e.g., x), *predicates* (e.g., $Female$, as in $Female(Mary)$), and *functions* (e.g., age, as in $age(Mary)$).
- A *term* is any constant, any variable, or any function applied to any term. Examples include $Mary$, x, $age(Mary)$, $age(x)$.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include $Female(Mary)$, $\neg Female(x)$, $Greater\_than(age(Mary), 20)$.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg Female(Joe)$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg Female(Joe)$).
- A *positive literal* is a literal with no negation sign (e.g., $Female(Mary)$).
- A *clause* is any disjunction of literals $M_1 \vee \ldots M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

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

  where $H, L_1 \ldots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.
- For any literals A and B, the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z. Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L.
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

TABLE 10.3
Basic definitions from first-order logic.

The FOIL algorithm is summarized in Table 10.4.

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

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

TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

Notice the outer loop corresponds to a variant of the SEQUENTIAL-COVERING algorithm discussed earlier; that is, it learns new rules one at a time, removing the positive examples covered by the latest rule before attempting to learn the next rule. The inner loop corresponds to a variant of our earlier LEARN-ONE-RULE algorithm, extended to accommodate first-order rules.

The hypothesis space search performed by FOIL is best understood by viewing it hierarchically. Each iteration through FOIL'S outer loop adds a new rule to its disjunctive hypothesis, Learned_rules. The effect of each new rule is to generalize the current disjunctive hypothesis (i.e., to increase the number of instances it classifies as positive), by adding a,new disjunct. Viewed at this level, the search is a specific-to-general search through the space of hypotheses, beginning with the most specific empty disjunction and terminating when the hypothesis is sufficiently general to cover all positive training examples. The inner loop of FOIL performs a finer-grained search to determine the exact definition of each new rule. This inner loop searches a second hypothesis space, consisting of conjunctions of literals, to find a

conjunction that will form the preconditions for the new rule. Within this hypothesis space, it conducts a general-to-specific, hill-climbing search, beginning with the most general preconditions possible (the empty precondition), then adding literals one at a time to specialize the rule until it avoids all negative examples.

Generating Candidate Specializations in FOIL: Candidate_literals

To generate candidate specializations of the current rule, FOIL generates a variety of new literals, each of which may be individually added to the rule preconditions. More precisely, suppose the current rule being considered is

$$P(x_1, x_2, \ldots, x_k) \leftarrow L_1 \ldots L_n$$

where $L_1 \ldots L_n$ are literals forming the current rule preconditions and where $P(x_1, x_2, \ldots, x_k)$ is the literal that forms the rule head, or postconditions. FOIL generates candidate specializations of this rule by considering new literals L_{n+1} that fit one of the following forms:

- $Q(v_1, \ldots, v_r)$, where Q is any predicate name occurring in *Predicates* and where the v_i are either new variables or variables already present in the rule. At least one of the v_i in the created literal must already exist as a variable in the rule.
- $Equal(x_j, x_k)$, where x_j and x_k are variables already present in the rule.

The negation of either of the above forms of literals.

Guiding the Search in FOIL: Foil_Gain(L, R)

The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the new literal. More precisely, consider some rule R, and a candidate literal L that might be added to the body of R. Let R' be the rule created by adding literal L to rule R. The value $Foil\_Gain(L, R)$ of adding L to R is defined as

$$Foil\_Gain(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right) \qquad (10.1)$$

where p_0 is the number of positive bindings of rule R, n_0 is the number of negative bindings of R, p_1 is the number of positive bindings of rule R', and n_1 is the number of negative bindings of R'. Finally, t is the number of positive bindings of rule R that are still covered after adding literal L to R. When a new variable is introduced into R by adding L, then any original binding is considered to be covered so long as some binding extending it is present in the bindings of R'.

Summary of FOIL

In the case of noise-free training data, FOIL may continue adding new literals to the rule until it covers no negative examples. To handle noisy data, the search is continued until some tradeoff occurs between rule accuracy, coverage, and complexity. FOIL uses a minimum description length approach to halt the growth of rules, in which new literals are added only when their description length is shorter than the description length of the training data they explain. The details of this strategy are given in Quinlan (1990). In addition, FOIL post-prunes each rule it learns, using the same rule post-pruning strategy used for decision trees (Chapter 3).

INDUCTION AS INVERTED DEDUCTION

assume as usual that the training data D is a set of training examples, each of the form $\langle x_i, f(x_i)\rangle$. Here x_i denotes the ith training instance and $f(x_i)$ denotes its target value. Then learning is the problem of discovering a hypothesis h, such that the classification $f(x_i)$ of each training instance x_i follows deductively from the hypothesis h, the description of x_i, and any other background knowledge B known to the system.

$$(\forall\langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i) \qquad (10.2)$$

The expression $X \vdash Y$ is read "Y follows deductively from X," or alternatively "X entails Y." Expression (10.2) describes the constraint that must be satisfied by the learned hypothesis h; namely, for every training instance x_i, the target classification $f(x_i)$ must follow deductively from B, h, and x_i.

each training instance xi follows deductively from the hypothesis h; X entails Y

In the remainder of this chapter we will explore this view of induction as the inverse of deduction. The general issue we will be interested in here is designing *inverse entailment operators*. An inverse entailment operator, $O(B, D)$ takes the training data $D = \{\langle x_i, f(x_i)\rangle\}$ and background knowledge B as input and produces as output a hypothesis h satisfying Equation (10.2).

$$O(B, D) = h \text{ such that } (\forall\langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

Of course there will, in general, be many different hypotheses h that satisfy $(\forall\langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$. One common heuristic in ILP for choosing among such hypotheses is to rely on the heuristic known as the Minimum Description Length principle (see Section 6.6).

Research on inductive logic programing following this formulation has encountered several practical difficulties: noisy training data, the number of

hy- potheses is so large, the complexity of the hypothesis space search increases as background knowledge B is increased.

In the following section, we examine one quite general inverse entailment operator that constructs hypotheses by inverting a deductive inference rule.

INVERTING RESOLUTION Resolution operator of propositional form:

1. Given initial clauses C_1 and C_2, find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2.
2. Form the resolvent C by including all literals from C_1 and C_2, except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

where $\cup$ denotes set union, and "$-$" denotes set difference.

TABLE 10.5
Resolution operator (propositional form). Given clauses C_1 and C_2, the resolution operator constructs a clause C such that $C_1 \wedge C_2 \vdash C$.

Inverse resolution operator (propositional form).:

1. Given initial clauses C_1 and C, find a literal L that occurs in clause C_1, but not in clause C.
2. Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

TABLE 10.6
Inverse resolution operator (propositional form). Given two clauses C and C_1, this computes a clause C_2 such that $C_1 \wedge C_2 \vdash C$.

First-Order Resolution

The resolution rule extends easily to first-order expressions. As in the propositional case, it takes two clauses as input and produces a third clause as output. The key difference from the propositional case is that the process is now based on the notion of unifying substitutions.

1. Find a literal L_1 from clause C_1, literal L_2 from clause C_2, and substitution θ such that $L_1\theta = \neg L_2\theta$.
2. Form the resolvent C by including all literals from $C_1\theta$ and $C_2\theta$, except for $L_1\theta$ and $\neg L_2\theta$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

TABLE 10.7
Resolution operator (first-order form).

Inverting Resolution: First-Order Case

Inverse resolution:

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1\theta_1\theta_2^{-1}\} \tag{10.4}$$

Equation (10.4) gives the inverse resolution rule for first-order logic. As in the propositional case, this inverse entailment operator is nondeterministic. In particular, in applying it we may in general find multiple choices for the clause C_1 to be resolved and for the unifying substitutions θ_1 and θ_2. Each set of choices may yield a different solution for C_2.

10.7.4 Generalization, θ-Subsumption, and Entailment

- *more_general_than*. In Chapter 2, we defined the *more_general_than_or_equal_to* relation ($\geq_g$) as follows: Given two boolean-valued functions $h_j(x)$ and $h_k(x)$, we say that $h_j \geq_g h_k$ if and only if $(\forall x)h_k(x) \rightarrow h_j(x)$. This $\geq_g$ relation is used by many learning algorithms to guide search through the hypothesis space.
- *θ-subsumption*. Consider two clauses C_j and C_k, both of the form $H \vee L_1 \vee \ldots L_n$, where H is a positive literal, and the L_i are arbitrary literals. Clause C_j is said to *θ-subsume* clause C_k if and only if there exists a substitution θ such that $C_j\theta \subseteq C_k$ (where we here describe any clause C by the set of literals in its disjunctive form). This definition is due to Plotkin (1970).
- *Entailment*. Consider two clauses C_j and C_k. Clause C_j is said to *entail* clause C_k (written $C_j \vdash C_k$) if and only if C_k follows deductively from C_j.

SUMMARY AND FURTHER READING

The sequential covering algorithm learns a disjunctive set of rules by first learning a single accurate rule, then removing the positive examples covered by this rule and iterating the process over the remaining training examples. It provides an efficient, greedy algorithm for learning rule sets, and an alternative to top-down decision tree learning algorithms such as ID3, which can be viewed as simultaneous, rather than sequential covering algorithms.

In the context of sequential covering algorithms, a variety of methods have been explored for learning a single rule. These methods vary in the search strategy they use for examining the space of possible rule preconditions. One popular approach, exemplified by the CN2 program, is to conduct a general-to-specific beam search, generating and testing progressively more specific rules until a sufficiently accurate rule is found. Alternative approaches search from specific to general hypotheses, use an example-driven search rather than generate and test, and employ different statistical measures of rule accuracy to guide the search.

Sets of first-order rules (i.e., rules containing variables) provide a highly expressive representation. For example, the programming language PROLOG represents general programs using collections of first-order Horn clauses. The problem of learning first-order Horn clauses is therefore often referred to as the problem of inductive logic programming.

One approach to learning sets of first-order rules is to extend the sequential covering algorithm of CN2 from propositional to first-order representations. This approach is exemplified by the FOIL program, which can learn sets of first-order rules, including simple recursive rule sets.

A second approach to learning first-order rules is based on the observation that induction is the inverse of deduction. In other words, the problem of induction is to find a hypothesis h that satisfies the constraint

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

where B is general background information, $x_1 \ldots x_n$ are descriptions of the instances in the training data D, and $f(x_1) \ldots f(x_n)$ are the target values of the training instances.

Following the view of induction as the inverse of deduction, some programs search for hypotheses by using operators that invert the well-known operators for deductive reasoning. For example, CIGOL uses inverse resolution, an operation that is the inverse of the deductive resolution operator commonly used for mechanical theorem proving. PROGOL combines an inverse entailment strategy with a general-to-specific strategy for searching the hypothesis space.

Unit 5

Combining Inductive and Analytical Learning:

Purely inductive learning methods formulate general hypotheses by finding empirical regularities over the training examples. Purely analytical methods use prior knowledge to derive general hypotheses deductively., This chapter considers methods that combine inductive and analytical mechanisms to obtain the benefits of both approaches: better generalization accuracy when prior knowledge is available and reliance(rely) on observed training data to overcome shortcomings in prior knowledge. The resulting combined methods outperform both purely inductive and purely analyti- cal learning methods. This chapter considers inductive-analytical learning methods based on both symbolic and artificial neural network representations.

MOTIVATION

	Inductive learning	Analytical learning
Goal:	Hypothesis fits data	Hypothesis fits domain theory
Justification:	Statistical inference	Deductive inference
Advantages:	Requires little prior knowledge	Learns from scarce data
Pitfalls:	Scarce data, incorrect bias	Imperfect domain theory

TABLE 12.1
Comparison of purely analytical and purely inductive learning.

What criteria should we use to compare alternative approaches to combining inductive and analytical learning? Some specific properties we would like from such a learning method include(Notice this list of desirable properties is quite ambitious):

- 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 purely 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 error in the training data.
- It should accommodate an unknown level of error in the domain theory.

INDUCTIVE-ANALYTICAL APPROACHES TO LEARNING

The Learning Problem

To summarize, the learning problem considered in this chapter is

Given:

- A set of training examples D, possibly containing errors
- A domain theory B, possibly containing errors
- A space of candidate hypotheses H

Determine:

- A hypothesis that best fits the training examples and domain theory

What precisely shall we mean by "the hypothesis that **best fits** the training examples and domain theory?'.

For example, we could require the hypothesis that minimizes some combined measure of these errors, such as:

$$\underset{h \in H}{\text{argmin}}\ k_D error_D(h) + k_B error_B(h)$$

An alternative perspective on the question of how to weight prior knowl- edge and data is the Bayesian perspective. Recall from Chapter 6 that Bayes theorem describes how to compute the posterior probability P(h1D) of hypothesis h given observed training data D. Unfortunately, Bayes theorem implicitly assumes pe$ect knowledge about the probability distributions P(h), P(D), and P(Dlh). When these quantities are only imperfectly known, Bayes theorem alone does not prescribe how to combine them with the observed data.

We will revisit the question of what we mean by "best" fit to the hypothesis and data as we examine specific algorithms. For now, we will simply say that the learning problem is to**minimize some combined measure of the error of the hypothesis over the data and the domain theory**.

Hypothesis Space Search

One way to understand the range of possible approaches is to return to our view of learning as a task of searching through the space of alternative hypotheses. We can characterize most learning methods as search algorithms by describing the hypothesis space H they search, the initial hypothesis ho at which they begin their search, the set of search operators 0 that define individual search steps, and the goal criterion G that specifies the search objective.

In this chapter we explore three different methods for using prior knowledge to alter the search performed by purely inductive methods:

Use prior knowledge to derive an initial hypothesis from which to begin the search. In this approach the domain theory B is used to construct an initial hypothesis ho that is consistent with B. A standard inductive method is then applied, starting with the initial hypothesis ho.

Use prior knowledge to alter the objective of the hypothesis space search. In this approach, the goal criterion G is modified to require that the output hypothesis fits the domain theory as well as the training examples.

Use prior knowledge to alter the available search steps. In this approach, the set of search operators 0 is altered by the domain theory.

USING PRIOR KNOWLEDGE TO INITIALIZE THE HYPOTHESIS

It is easy to see the motivation for this technique: if the domain theory is correct, the initial hypothesis will correctly classify all the training examples and there will be no need to revise it. However, if the initial hypothesis is found to imperfectly classify the training examples, then it will be refined inductively to improve its fit to the training examples.

The KBANN Algorithm

The two stages of the KBANN algorithm are first to create an artificial neural network that perfectly fits the domain theory and second to use the BACKPROPA-CATION algorithm to refine this initial network to fit the training

examples. The details of this algorithm, including the algorithm for creating the initial network, are given in Table 12.2 and illustrated in Section 12.3.2.

KBANN(*Domain_Theory*, *Training_Examples*)
Domain_Theory: Set of propositional, nonrecursive Horn clauses.
Training_Examples: Set of ⟨*input output*⟩ *pairs of the target function.*

Analytical step: Create an initial network equivalent to the domain theory.

1. For each instance attribute create a network input.
2. For each Horn clause in the *Domain_Theory*, create a network unit as follows:
   - Connect the inputs of this unit to the attributes tested by the clause antecedents.
   - 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.

Inductive step: Refine the initial network.

4. Apply the BACKPROPAGATION algorithm to adjust the initial network weights to fit the *Training_Examples*.

TABLE 12.2
The KBANN algorithm. The domain theory is translated into an equivalent neural network (steps 1–3), which is inductively refined using the BACKPROPAGATION algorithm (step 4). A typical value for the constant W is 4.0.

An Illustrative Example

Domain theory:

$$Cup \leftarrow Stable, Liftable, OpenVessel$$
$$Stable \leftarrow BottomIsFlat$$
$$Liftable \leftarrow Graspable, Light$$
$$Graspable \leftarrow HasHandle$$
$$OpenVessel \leftarrow HasConcavity, ConcavityPointsUp$$

Training examples:

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

TABLE 12.3
The *Cup* learning task. An approximate domain theory and a set of training examples for the target concept *Cup*.

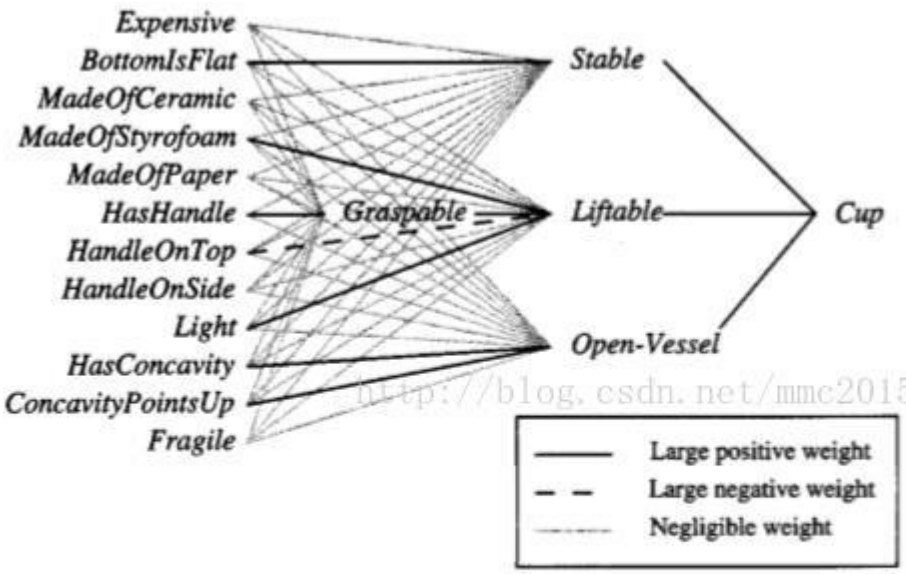


FIGURE 12.3
Result of inductively refining the initial network. KBANN uses the training examples to modify the network weights derived from the domain theory. Notice the new dependency of *Liftable* on *MadeOfStyrofoam* and *HandleOnTop*.

Remarks

The chief benefit of KBANN over purely inductive BACKPROPAGATION (beginning with random initial weights) is that it typically generalizes more accurately than BACKPROPAGATION when given an approximately correct domain theory, es- pecially when training data is scarce.

Limitations of KBANN include the fact that it can accommodate only propositional domain theories; that is, collections of variable-free Horn clauses. It is also possible for KBANN to be misled when given highly inaccurate domain theories, so that its generalization accuracy can deteriorate below the level of BACKPROPA-GATION. Nevertheless, it and related algorithms have been shown to be useful for several practical problems.

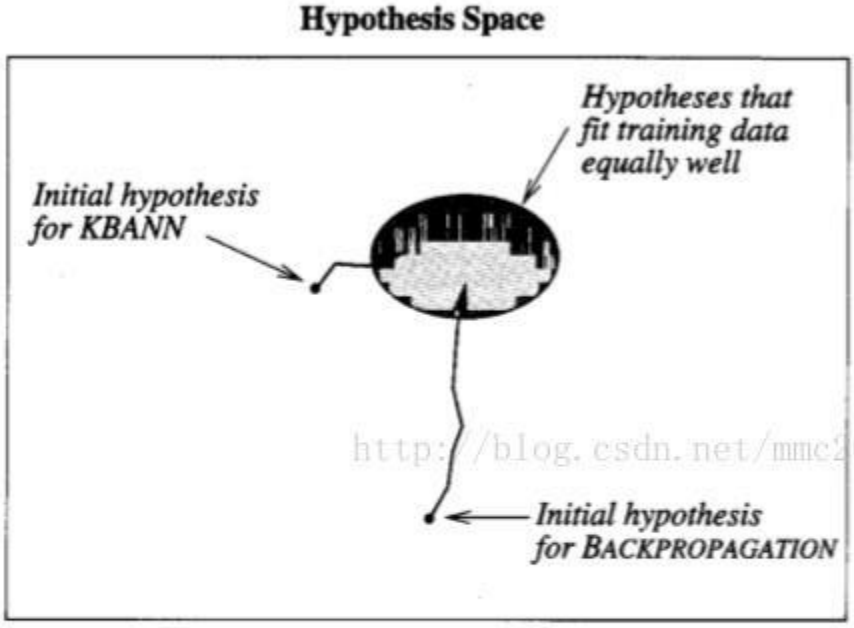


FIGURE 12.4
Hypothesis space search in KBANN. KBANN initializes the network to fit the domain theory, whereas BACKPROPAGATION initializes the network to small random weights. Both then refine the weights iteratively using the same gradient descent rule. When multiple hypotheses can be found that fit the training data (shaded region), KBANN and BACKPROPAGATION are likely to find different hypotheses due to their different starting points.

USING PRIOR KNOWLEDGE TO ALTER THE SEARCH OBJECTIVE(使用先验知识改变搜索目标)

An alternative way of using prior knowledge is to incorporate it into the error criterion minimized by gradient descent, so that the network must fit a combined function of the training data and domain theory. In this section, we consider using prior knowledge in this fashion. In particular, we consider prior knowledge in the form of known derivatives of the target function.

The TANGENTPROP Algorithm

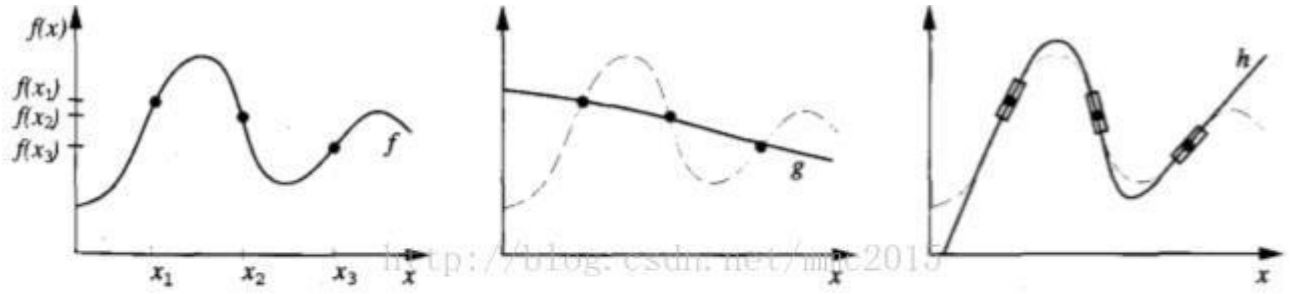


FIGURE 12.5
Fitting values and derivatives with TANGENTPROP. Let f be the target function for which three examples $\langle x_1, f(x_1)\rangle$, $\langle x_2, f(x_2)\rangle$, and $\langle x_3, f(x_3)\rangle$ are known. Based on these points the learner might generate the hypothesis g. If the derivatives are also known, the learner can generalize more accurately h.

The modified error function is:

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu \sum_j \left(\frac{\partial f(s_j(\alpha, x_i))}{\partial \alpha} - \frac{\partial \hat{f}(s_j(\alpha, x_i))}{\partial \alpha} \right)^2_{\alpha=0} \right] \quad (12.1)$$

where μ is a constant provided by the user to determine the relative importance of fitting training values versus fitting training derivatives. Notice the first term in this definition of E is the original squared error of the network versus training *values*, and the second term is the squared error in the network versus training *derivatives*.

Remarks

Although TANGENTPROP succeeds in combining prior knowledge with training data to guide learning of neural networks, it is not robust to errors in the prior knowledge. Consider what will happen when prior knowledge is incorrect, that is, when the training derivatives input to the learner do not correctly reflect the derivatives of the true target function. In this case the algorithm will attempt to fit incorrect derivatives. It may therefore generalize less accurately than if it ignored this prior knowledge altogether and used the purely inductive BACKPROPAGATION algorithm. If we knew in advance the degree of error in the training derivatives, we might use this information to select the constant p that determines the relative importance of fitting training values and fitting training derivatives. However, this information is unlikely to be known in advance.

It is interesting to compare the search through hypothesis space (weight space) performed by TANGENTPROP, KBANN, and BACKPROPAGATION.

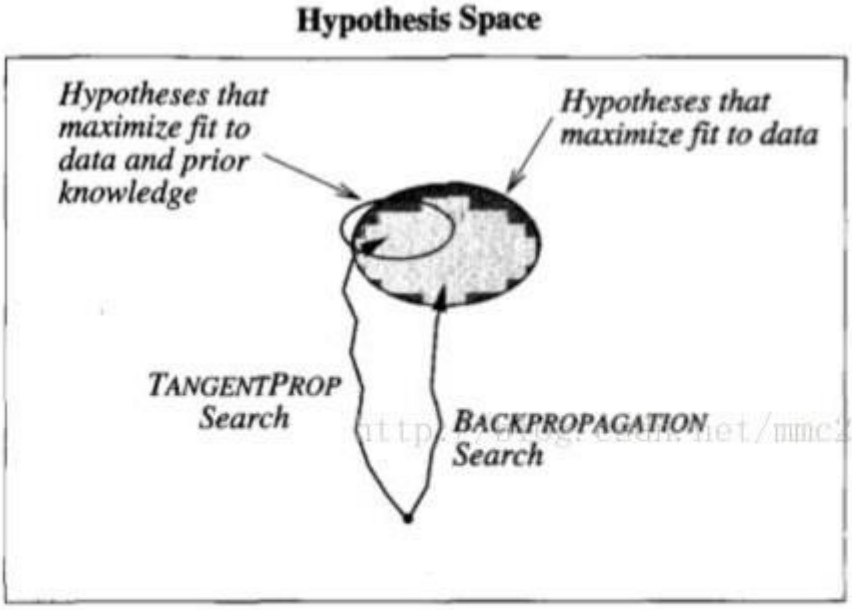


FIGURE 12.6
Hypothesis space search in TANGENTPROP. TANGENTPROP initializes the network to small random weights, just as in BACKPROPAGATION. However, it uses a different error function to drive the gradient descent search. The error used by TANGENTPROP includes both the error in predicting training *values* and in predicting the training *derivatives* provided as prior knowledge.

The EBNN Algorithm

The EBNN (Explanation-Based Neural Network learning) algorithm (Mitchell and Thrun 1993a; Thrun 1996) builds on the TANGENTPROP algorithm in two significant ways. First, instead of relying on the user to provide training derivatives, EBNN computes training derivatives itself for each observed training example. These training derivatives are calculated by explaining each training example in terms of a given domain theory, then extracting training derivatives from this explanation. Second, EBNN addresses the issue of how to weight the relative importance of the inductive and analytical components of learning (i.e., how to select the parameter μ in Equation [12.1]). The value of 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. Thus, the analytical component of learning is emphasized for those training examples that are correctly

explained by the domain theory and de-emphasized for training examples that are poorly explained.

The inputs to EBNN include (1) a set of training examples of the form $\langle x_i, f(x_i)\rangle$ with no training derivatives provided, and (2) a domain theory analogous to that used in explanation-based learning (Chapter 11) and in KBANN, but represented by a set of previously trained neural networks rather than a set of Horn clauses. The output of EBNN is a new neural network that approximates the target function f. This learned network is trained to fit both the training examples $\langle x_i, f(x_i)\rangle$ and training derivatives of f extracted from the domain theory. Fitting the training examples $\langle x_i, f(x_i)\rangle$ constitutes the inductive component of learning, whereas fitting the training derivatives extracted from the domain theory provides the analytical component.

Above we described how the domain theory prediction can be used to generate a set of training derivatives. To be more precise, the full EBNN algorithm is as follows. Given the training examples and domain theory, EBNN first creates a new, fully connected feedforward network to represent the target function. This target network is initialized with small random weights, just as in BACKPROPAGATION. Next, for each training example $\langle x_i, f(x_i)\rangle$ EBNN determines the corresponding training derivatives in a two-step process. First, it uses the domain theory to predict the value of the target function for instance x_i. Let $A(x_i)$ denote this domain theory prediction for instance x_i. In other words, $A(x_i)$ is the function defined by the composition of the domain theory networks forming the explanation for x_i. Second, the weights and activations of the domain theory networks are analyzed to extract the derivatives of $A(x_i)$ with respect to each of the components of x_i (i.e., the Jacobian of $A(x)$ evaluated at $x = x_i$). Extracting these derivatives follows a process very similar to calculating the δ terms in the BACKPROPAGATION algorithm (see Exercise 12.5). Finally, EBNN uses a minor variant of the TANGENTPROP algorithm to train the target network to fit the following error function

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu_i \sum_j \left(\frac{\partial A(x)}{\partial x^j} - \frac{\partial \hat{f}(x)}{\partial x^j} \right)^2_{(x=x_i)} \right] \tag{12.2}$$

where

$$\mu_i \equiv 1 - \frac{|A(x_i) - f(x_i)|}{c} \tag{12.3}$$

Here x_i denotes the ith training instance and $A(x)$ denotes the domain theory prediction for input x. The superscript notation x^j denotes the jth component of the vector x (i.e., the jth input node of the neural network). The coefficient c is a normalizing constant whose value is chosen to assure that for all i, $0 \leq \mu_i \leq 1$.

The relative importance of the inductive and analytical learning components is determined in EBNN by the constant μ_i, defined in Equation (12.3). The value of μ_i is determined by the discrepancy between the domain theory prediction $A(x_i)$ and the training value $f(x_i)$. The analytical component of learning is thus weighted more heavily for training examples that are correctly predicted by the domain theory and is suppressed for examples that are not correctly predicted.

Remarks

To summarize, the EBNN algorithm uses a domain theory expressed as a set of previously learned neural networks, together with a set of training examples, to train its output hypothesis (the 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 attribute value, according to the domain theory. These training derivatives are provided to a variant of TANGENTPROP, which fits the target network to these derivatives and to the training example values. Fitting the derivatives constrains the learned network to fit dependencies given by the domain theory, while fitting the training values constrains it to fit the observed data itself. The weight pi 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.

EBNN bears an interesting relation to other explanation-based learning methods, such as PROLOG-EBGde scribed in Chapter 11.

There are several differences in capabilities between EBNN and the symbolic explanation-based methods of Chapter 11. The main difference is that EBNN accommodates imperfect domain theories, whereas PROLOG-EBGdo es not. This difference follows from the fact that EBNN is built on the inductive mechanism of fitting the observed training values and uses the domain theory only as an additional constraint on the learned hypothesis. A second important difference follows from the fact that PROLOG-EBGle arns a growing set of Horn clauses, whereas EBNN learns a fixed-size neural network. As discussed in Chapter 11, one difficulty in learning sets of Horn clauses is that the cost of classifying a new instance grows as learning proceeds and new Horn clauses are added. This problem is avoided in EBNN because the fixed-size target network requires constant time to classify new instances. However, the fixed-size neural network suffers the corresponding disadvantage that it may be unable to represent sufficiently complex functions, whereas a growing set of Horn clauses can represent increasingly complex functions.

USING PRIOR KNOWLEDGE TO AUGMENT SEARCH OPERATORS

In this section we consider a third way of 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.

The FOCL Algorithm

We will say a literal is **operational** if it is allowed to be used in describing an output hypothesis. For example, in the Cup example of Figure 12.3 we allow output hypotheses to refer only to the 12 attributes that describe the training examples (e.g., HasHandle, HandleOnTop). Literals based on these 12 attributes are thus considered operational. In contrast, literals that occur only as intermediate features in the domain theory, but not as primitive attributes of the instances, are considered nonoperational. An example of a nonoperational attribute in this case is the attribute Stable.

At each point in its general-to-specific search, FOCL expands its current hypothesis h using the following two operators:

For each operational literal that is not part of h, create a specialization of h by adding this single literal to the preconditio s. This is also the method used by FOIL to generate candidate successors. he solid arrows in Figure 12.8 denote this type of specialization.

Create an operational, logically sufficient condition for the target concept according to the domain theory. Add this set of literals to the current precon-ditions of h. Finally, prune the preconditions of h by removing any literals that are unnecessary according to the training data. The dashed arrow in Figure 12.8 denotes this type of specialization.

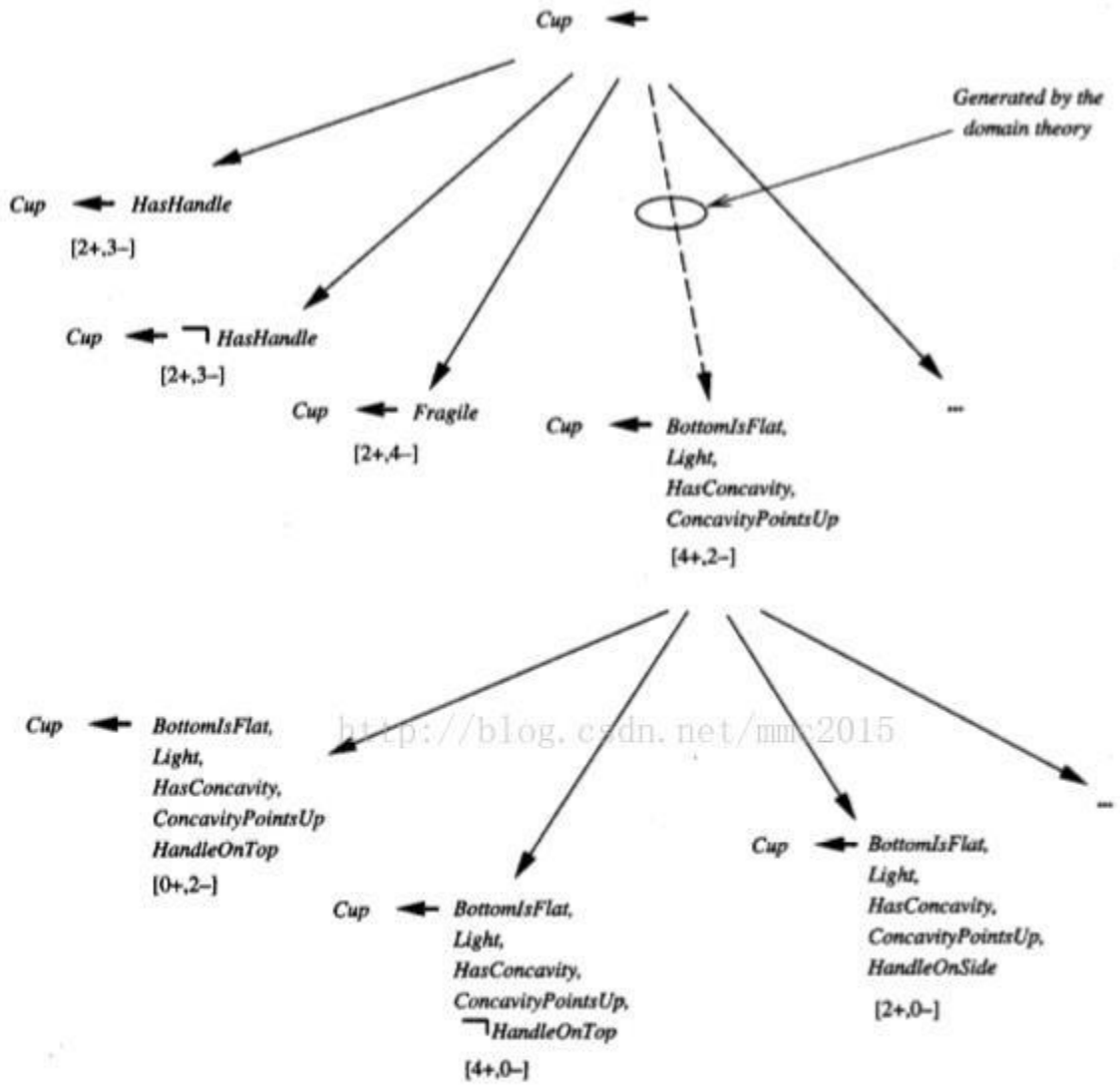


FIGURE 12.8
Hypothesis space search in FOCL. To learn a single rule, FOCL searches from general to increasingly specific hypotheses. Two kinds of operators generate specializations of the current hypothesis. One kind adds a single new literal (solid lines in the figure). A second kind of operator specializes the rule by adding a set of literals that constitute logically sufficient conditions for the target concept, according to the domain theory (dashed lines in the figure). FOCL selects among all these candidate specializations, based on their performance over the data. Therefore, imperfect domain theories will impact the hypothesis only if the evidence supports the theory. This example is based on the same training data and domain theory as the earlier KBANN example.

Once candidate specializations of the current hypothesis have been generated, using both of the two operations above, the candidate with highest information gain is selected.

Remarks

To summarize, FOCL uses both a syntactic generation of candidate specializations and a domain theory driven generation of candidate specializations at each step in the search. The algorithm chooses among these candidates based solely on their empirical support over the training data. Thus, the domain theory is used in a fashion that biases the learner, but leaves final search choices to be made based on performance over the training data. The bias introduced by the domain theory is a preference in favor of Horn clauses most similar to operational, logically sufficient conditions entailed by the domain theory. This bias is combined with the bias of the purely inductive FOIL program, which is a preference for shorter hypotheses.

FOCL has been shown to generalize more accurately than the purely inductive FOIL algorithm in a number of application domains in which an imperfect domain theory is available.

12.7 SUMMARY AND FURTHER READING

Approximate prior knowledge, or domain theories, are available in many practical learning problems. Purely inductive methods such as decision tree induction and neural network BACKPROPAGATION fail to utilize such domain theories, and therefore perform poorly when data is scarce. Purely analytical learning methods such as PROLOG-EBG utilize such domain theories, but produce incorrect hypotheses when given imperfect prior knowledge. Methods that blend inductive and analytical learning can gain the benefits of both approaches: reduced sample complexity and the ability to overrule incorrect prior knowledge.

One way to view algorithms for combining inductive and analytical learning is to consider how the domain theory affects the hypothesis space search. In this chapter we examined methods that use imperfect domain theories to (1) create the initial hypothesis in the search, (2) expand the set of search operators that generate revisions to the current hypothesis, and (3) alter the objective of the search.

A system that uses the domain theory to initialize the hypothesis is KBANN. This algorithm uses a domain theory encoded as propositional rules to analytically construct an artificial neural network that is equivalent to the domain theory. This network is then inductively refined using the BACKPROPAGATION algorithm, to improve its performance over the training

data. The result is a network biased by the original domain theory, whose weights are refined inductively based on the training data.

TANGENTPROP uses prior knowledge represented by desired derivatives of the target function. In some domains, such as image processing, this is a natural way to express prior knowledge. TANGENTPROP incorporates this knowledge by altering the objective function minimized by gradient descent search through the space of possible hypotheses.

EBNN uses the domain theory to alter the objective in searching the hypothesis space of possible weights for an artificial neural network. It uses a domain theory consisting of previously learned neural networks to perform a neural network analog to symbolic explanation-based learning. As in symbolic explanation-based learning, the domain theory is used to explain individual examples, yielding information about the relevance of different example features. With this neural network representation, however, information about relevance is expressed in the form of derivatives of the target function value with respect to instance features. The network hypothesis is trained using a variant of the TANGENTPROP algorithm, in which the error to be minimized includes both the error in network output values and the error in network derivatives obtained from explanations.

FOCL uses the domain theory to expand the set of candidates considered at each step in the search. It uses an approximate domain theory represented by first order Horn clauses to learn a set of Horn clauses that approximate the target function. FOCL employs a sequential covering algorithm, learning each Horn clause by a general-to-specific search. The domain theory is used to augment the set of next more specific candidate hypotheses considered at each step of this search. Candidate hypotheses are then evaluated based on their performance over the training data. In this way, FOCL combines the greedy, general-to-specific inductive search strategy of FOIL with the rule-chaining, analytical reasoning of analytical methods.

The question of how to best blend prior knowledge with new observations remains one of the key open questions in machine learning.

Reinforcement Learning:

Reinforcement learning addresses the question of how an autonomous agent(agent) that senses and acts in its environment can learn to choose optimal actions to achieve its goals. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward. This chapter focuses on an algorithm called Q learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

INTRODUCTION

This general setting for robot learning is summarized in Figure 13.1.

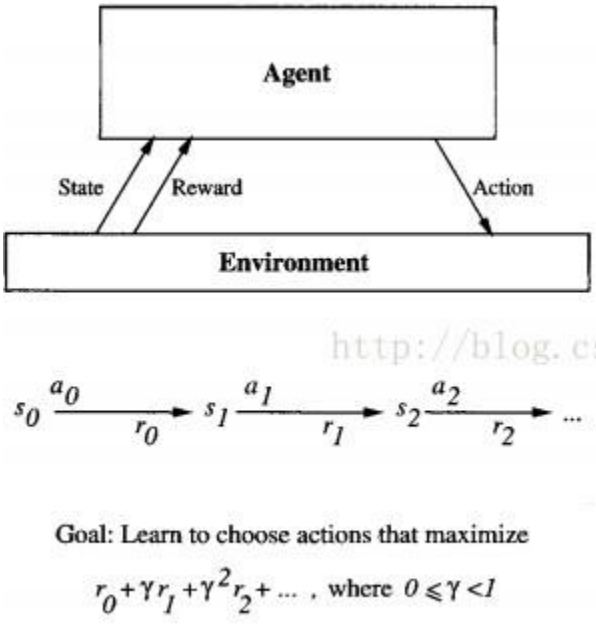


FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the 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, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

The problem of learning a control policy to choose actions is similar in some respects to the function approximation problems discussed in other chapters. The target function to be learned in this case is a control policy, π : S -> A, that outputs an appropriate action a from the set A, given the current state s from the set S. However, this reinforcement learning problem differs from other function approximation tasks in several important respects:

Delayed reward: 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.

Exploration: The learner faces a tradeoff in choosing whether to favor exploration of unknown states and actions (to gather new information), or

exploitation of states and actions that it has already learned will yield high reward (to maximize its cumulative reward).

Partially observable states: In many practical situations sensors provide only partial information. For example, a robot with a forward-pointing camera cannot see what is behind it. In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions.

Life-long learning: Robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

THE LEARNING TASK

Here we define one quite general formulation of the problem, based on Markov decision processes. This formulation of the problem follows the problem illustrated in Figure 13.1.

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (13.1)$$

In a Markov decision process (MDP) 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 δ and r are part of the environment and are not necessarily known to the agent. In an MDP, 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. In this chapter we consider only the case in which S and A are finite. In general, δ and r may be nondeterministic functions, but we begin by considering only the deterministic case.

We are now in a position to state precisely the agent's learning task. We require that the agent learn a policy π that maximizes $V^{\pi}(s)$ for all states s. We will call such a policy an *optimal policy* and denote it by π^*.

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \tag{13.2}$$

To simplify notation, we will refer to the value function $V^{\pi^*}(s)$ of such an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state s; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state s.

an example:

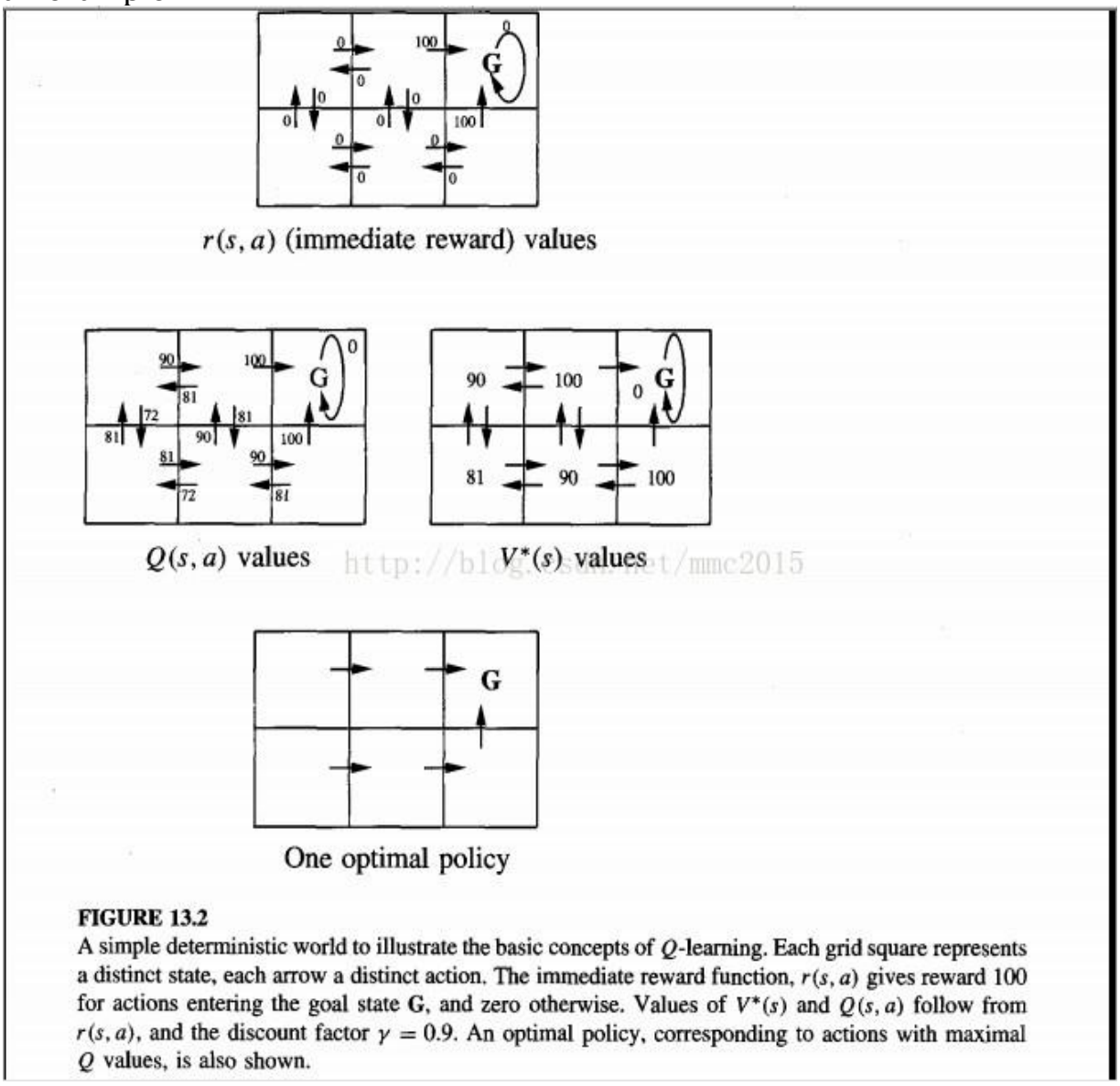


FIGURE 13.2
A simple deterministic world to illustrate the basic concepts of Q-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state G, and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown.

Q LEARNING

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))] \tag{13.3}$$

The Q Function

令 $$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \tag{13.4}$$

则 $$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \tag{13.5}$$

It may at first seem surprising that one can choose globally optimal action sequences by reacting repeatedly to the local values of Q for the current state. This means the agent can choose the optimal action without ever conducting a lookahead search to explicitly consider what state results from the action. Part of the beauty of Q learning is that the evaluation function is defined to have precisely this property—the value of Q for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s.

To illustrate, Figure 13.2 shows the Q values for every state and action in the simple grid world. Notice that the Q value for each state-action transition equals the r value for this transition plus the V^* value for the resulting state discounted by γ. Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal Q values.

An Algorithm for Learning Q

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

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a') \tag{13.6}$$

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q (Watkins 1989). To describe the algorithm, we will use the symbol $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function. In this algorithm the learner represents its hypothesis $\hat{Q}$ by a large table with a separate entry for each state-action pair. The table entry for the pair $\langle s, a \rangle$ stores the value for $\hat{Q}(s, a)$—the learner's current hypothesis about the actual but unknown value $Q(s, a)$. The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero). The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. It then updates the table entry for $\hat{Q}(s, a)$ following each such transition, according to the rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \tag{13.7}$$

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

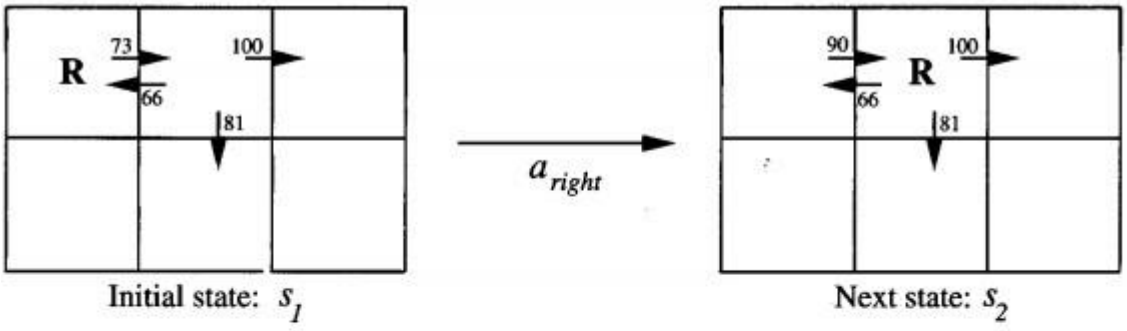
- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

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

- $s \leftarrow s'$

TABLE 13.1
Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.

An Illustrative Example



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

FIGURE 13.3
The update to $\hat{Q}$ after executing a single action. The diagram on the left shows the initial state s_1 of the robot (**R**) and several relevant $\hat{Q}$ values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72.9$, where a_{right} refers to the action that moves **R** to its right. When the robot executes the action a_{right}, it receives immediate reward $r = 0$ and transitions to state s_2. It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its $\hat{Q}$ estimates for the new state s_2. Here $\gamma = 0.9$.

Convergence

Theorem 13.1. 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 (13.7), 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 $n \to \infty$, for all s, a.

NONDETERMINISTIC REWARDS AND ACTIONS

$$V^{\pi}(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

$$
\begin{aligned}
Q(s,a) &\equiv E[r(s,a) + \gamma V^*(\delta(s,a))] \\
&= E[r(s,a)] + \gamma E[V^*(\delta(s,a))] \\
&= E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \qquad (13.8)
\end{aligned}
$$

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

$$
Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \qquad (13.9)
$$

To summarize, we have simply redefined V and Q in the nondeterministic case to be the expected value of its previously defined quantity for the deterministic case.

TEMPORAL DIFFERENCE LEARNING

To explore this issue further, recall that our Q learning training rule calculates a training value for $\hat{Q}(s_t, a_t)$ in terms of the values for $\hat{Q}(s_{t+1}, a_{t+1})$ where s_{t+1} is the result of applying action a_t to the state s_t. Let $Q^{(1)}(s_t, a_t)$ denote the training value calculated by this one-step lookahead

$$
Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)
$$

One alternative way to compute a training value for $Q(s_t, a_t)$ is to base it on the observed rewards for two steps

$$
Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)
$$

or, in general, for n steps

$$
Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)
$$

Sutton (1988) introduces a general method for blending these alternative training estimates, called TD(λ). The idea is to use a constant $0 \leq \lambda \leq 1$ to combine the estimates obtained from various lookahead distances in the following fashion

$$
Q^\lambda(s_t, a_t) \equiv (1 - \lambda) \left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots \right]
$$

An equivalent recursive definition for Q^λ is

$$
\begin{aligned}
Q^\lambda(s_t, a_t) = r_t + \gamma [& (1 - \lambda) \max_a \hat{Q}(s_t, a_t) \\
& + \lambda \, Q^\lambda(s_{t+1}, a_{t+1})]
\end{aligned}
$$

13.8 SUMMARY AND FURTHER READING

- Reinforcement learning addresses the problem of learning control strategies for autonomous agents. It assumes that training information is available in the form of a real-valued reward signal given for each state-action transition. The goal of the agent is to learn an action policy that maximizes the total reward it will receive from any starting state.
- The reinforcement learning algorithms addressed in this chapter fit a problem setting known as a Markov decision process. In Markov decision processes, the outcome of applying any action to any state depends only on this action and state (and not on preceding actions or states). Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems.
- Q learning is one form of reinforcement learning in which the agent learns an evaluation function over states and actions. In particular, the evaluation function $Q(s, a)$ is defined as the maximum expected, discounted, cumulative reward the agent can achieve by applying action a to state s. The Q learning algorithm has the advantage that it can be employed even when the learner has no prior knowledge of how its actions affect its environment.
- Q learning can be proven to converge to the correct Q function under certain assumptions, when the learner's hypothesis $\hat{Q}(s, a)$ is represented by a lookup table with a distinct entry for each $\langle s, a \rangle$ pair. It can be shown to converge in both deterministic and nondeterministic MDPs. In practice, Q learning can require many thousands of training iterations to converge in even modest-sized problems.
- Q learning is a member of a more general class of algorithms, called temporal difference algorithms. In general, temporal difference algorithms learn by iteratively reducing the discrepancies between the estimates produced by the agent at different times.
- Reinforcement learning is closely related to dynamic programming approaches to Markov decision processes. The key difference is that historically these dynamic programming approaches have assumed that the agent possesses knowledge of the state transition function $\delta(s, a)$ and reward function $r(s, a)$. In contrast, reinforcement learning algorithms such as Q learning typically assume the learner lacks such knowledge.