In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning methods simply store the training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance. Instance-based learning includes nearest neighbor and locally weighted regression methods that assume instances can be represented as points in a Euclidean space. It also includes case-based reasoning methods that use more complex, symbolic representations for instances. Instance-based methods are sometimes referred to as “lazy” learning methods because they delay processing until a new instance must be classified. A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

8.1 INTRODUCTION

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

Instance-based methods can also use more complex, symbolic representations for instances. In case-based learning, instances are represented in this fashion and the process for identifying “neighboring” instances is elaborated accordingly. Case-based reasoning has been applied to tasks such as storing and reusing past experience at a help desk, reasoning about legal cases by referring to previous cases, and solving complex scheduling problems by reusing relevant portions of previously solved problems.

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. Therefore, techniques for efficiently indexing training examples are a significant practical issue in reducing the computation required at query time. 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.

In the next section we introduce the \( k \)-NEAREST NEIGHBOR learning algorithm, including several variants of this widely-used approach. The subsequent section discusses locally weighted regression, a learning method that constructs local approximations to the target function and that can be viewed as a generalization of \( k \)-NEAREST NEIGHBOR algorithms. We then describe radial basis function networks, which provide an interesting bridge between instance-based and neural network learning algorithms. The next section discusses case-based reasoning, an instance-based approach that employs symbolic representations and knowledge-based inference. This section includes an example application of case-based reasoning to a problem in engineering design. Finally, we discuss the fundamental differences in capabilities that distinguish lazy learning methods discussed in this chapter from eager learning methods discussed in the other chapters of this book.

### 8.2 \( k \)-NEAREST NEIGHBOR LEARNING

The most basic instance-based method is the \( k \)-NEAREST NEIGHBOR algorithm. This algorithm assumes all instances correspond to points in the \( n \)-dimensional space \( \mathbb{R}^n \). The nearest neighbors of an instance are defined in terms of the standard
Euclidean distance. More precisely, let an arbitrary instance \( x \) be described by the feature vector

\[
(a_1(x), a_2(x), \ldots, a_n(x))
\]

where \( a_r(x) \) denotes the value of the \( r \)-th attribute of instance \( x \). Then the distance between two instances \( x_i \) and \( x_j \) is defined to be \( d(x_i, x_j) \), where

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

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 \( f : \mathbb{R}^n \rightarrow V \), where \( V \) is the finite set \( \{v_1, \ldots, v_k\} \). The \( k \)-NEAREST NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1. As shown there, the value \( \hat{f}(x_q) \) returned by this algorithm as its estimate of \( f(x_q) \) is just the most common value of \( f \) among the \( k \) training examples nearest to \( x_q \). If we choose \( k = 1 \), then the 1-NEAREST NEIGHBOR algorithm assigns to \( \hat{f}(x_q) \) the value \( f(x_i) \) where \( x_i \) is the training instance nearest to \( x_q \). For larger values of \( k \), the algorithm assigns the most common value among the \( k \) nearest training examples.

Figure 8.1 illustrates the operation of the \( k \)-NEAREST NEIGHBOR algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by “+” and “−” respectively. A query point \( x_q \) is shown as well. Note the 1-NEAREST NEIGHBOR algorithm classifies \( x_q \) as a positive example in this figure, whereas the 5-NEAREST NEIGHBOR algorithm classifies it as a negative example.

What is the nature of the hypothesis space \( H \) implicitly considered by the \( k \)-NEAREST NEIGHBOR algorithm? Note the \( k \)-NEAREST NEIGHBOR algorithm never forms an explicit general hypothesis \( \hat{f} \) regarding the target function \( f \). It simply computes the classification of each new query instance as needed. Nevertheless,

---

**Training algorithm:**
- For each training example \((x, f(x))\), add the example to the list \textit{training examples}

**Classification algorithm:**
- Given a query instance \( x_q \) to be classified,
  - Let \( x_1 \ldots x_k \) denote the \( k \) instances from \textit{training examples} that are nearest to \( x_q \)
  - Return

  \[
  \hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i))
  \]

  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 \( f : \mathbb{R}^n \rightarrow V \).
of nearest neighbor.

-Nearest Neighbor.

-Nearest Neighbor.

-Nearest Neighbor.

binary value.

Figure 8.1 illustrates the decision surface induced by the 1-Nearest Neighbor algorithm for a typical set of training examples. The convex polygon surrounding each training example indicates the region of instance closest to that point (i.e., the instances for which the 1-Nearest Neighbor algorithm will assign the classification belonging to that training example).

We can still ask what the implicit general function is, or what classifications would be assigned if we were to hold the training examples constant and query the algorithm with every possible instance in \( X \). The diagram on the right side of Figure 8.1 shows the shape of this decision surface induced by 1-Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the Voronoi diagram of the set of training examples.

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 \( f : \mathbb{R}^d \rightarrow \mathbb{R} \), we replace the final line of the above algorithm by the line

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

**8.2.1 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 \( x_q \), giving greater weight to closer neighbors. For example, in the algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from \( x_q \).
This can be accomplished by replacing the final line of the algorithm by

\[
\hat{f}(x_q) \leftarrow \arg \max_{v \in V} \sum_{i=1}^{k} w_i \delta(v, f(x_i))
\]  

(8.2)

where

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

(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

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

(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).

Note all of the above variants of the \(k\)-NEAREST NEIGHBOR algorithm consider only the \(k\) nearest neighbors to classify the query point. Once we add distance weighting, there is really no harm in allowing all training examples to have an influence on the classification of the \(x_q\), because very distant examples will have very little effect on \(\hat{f}(x_q)\). The only disadvantage of considering all examples is that our classifier will run more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a global method. If only the nearest training examples are considered, we call it a local method. When the rule in Equation (8.4) is applied as a global method, using all training examples, it is known as Shepard’s method (Shepard 1968).

### 8.2.2 Remarks on \(k\)-NEAREST NEIGHBOR Algorithm

The distance-weighted \(k\)-NEAREST NEIGHBOR algorithm is a highly effective inductive inference method for many practical problems. It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data. Note that by taking the weighted average of the \(k\) neighbors nearest to the query point, it can smooth out the impact of isolated noisy training examples.

What is the inductive bias of \(k\)-NEAREST NEIGHBOR? The basis for classifying new query points is easily understood based on the diagrams in Figure 8.1. The inductive bias corresponds to an assumption that the classification of an instance \(x_q\) will be most similar to the classification of other instances that are nearby in Euclidean distance.

One practical issue in applying \(k\)-NEAREST NEIGHBOR algorithms is that the distance between instances is calculated based on all attributes of the instance (i.e., both discrete and continuous). This means that each attribute contributes only as much as its relative effect on errors. In many cases, each attribute is measured in units that are relatively small, making the effect of this attribute on decision error sometimes negligible.

As a result, for a typical small \(k\), all 20 attributes might be used, even though the dominant ones have less impact. The curse of dimensionality is also present in this problem.

attributes of the instances. The correlation between the attributes can be quite high, which can bias the algorithm. The first rule of thumb to follow is the very first rule: do not use all attributes validly. For example, if certain data were not available until the end to the algorithm, the attributes were computed very early in the process. The bias (equal weights of \(k\) attributes) was irrelevant.

attribute weighting is important to scaling up the algorithm to large datasets. One common method is to use the cosine similarity of the algorithm. However, this can be valid when the attributes are independent, if the training data is not available, and if no attributes are used. Note that the constant weighting method is the inductive bias for all attributes, it is also infeasible when the attributes are used.
(8.2)

(8.3)

of the
reasonable, we
functions in a

(8.4)

(8.4) is
assures

consider
the distance
have an
will have
samples is
considered

method.
method.
training

induc-
training
training
contribution
to the
classifying
1. The
instance

arbitrarily
that the
instance

(i.e., on all axes in the Euclidean space containing the instances). This lies in
count to methods such as rule and decision tree learning systems that select
only a subset of the instance attributes when forming the hypothesis. To see the
effect of this policy, consider applying $k$-NEAREST NEIGHBOR to a problem in which
each instance is described by 20 attributes, but where only 2 of these attributes
are relevant to determining the classification for the particular target function. In
this case, instances that have identical values for the 2 relevant attributes may
nevertheless be distant from one another in the 20-dimensional instance space.
As a result, the similarity metric used by $k$-NEAREST NEIGHBOR—depending on
all 20 attributes—will be misleading. The distance between neighbors will be
dominated by the large number of irrelevant attributes. 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. To see how,
first note that we wish to stretch (multiply) the $j$th axis by some factor $z_j$, where
the values $z_1, \ldots, z_n$ are chosen to minimize the true classification error of the
learning algorithm. Second, note that this true error can be estimated using
cross-validation. Hence, one algorithm is to select a random subset of the available
data to use as training examples, then determine the values of $z_1, \ldots, z_n$ that lead
to the minimum error in classifying the remaining examples. By repeating this
process multiple times the estimate for these weighting factors can be made more
accurate. This process of stretching the axes in order to optimize the performance
of $k$-NEAREST NEIGHBOR provides a mechanism for suppressing the impact of
irrelevant attributes.

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 $z_j$
scaling factors to zero. Moore and Lee (1994) discuss efficient cross-validation
methods for selecting relevant subsets of the attributes for $k$-NEAREST NEIGHBOR
algorithms. In particular, they explore methods based on leave-one-out cross-
validation, in which the set of $m$ training instances is repeatedly divided into a
training set of size $m - 1$ and test set of size 1, in all possible ways. This leave-one-
out approach is easily implemented in $k$-NEAREST NEIGHBOR algorithms because
no additional training effort is required each time the training set is redefined.
Note both of the above approaches can be seen as stretching each axis by some
constant factor. Alternatively, we could stretch each axis by a value that varies over
the instance space. However, as we increase the number of degrees of freedom
available to the algorithm for redefining its distance metric in such a fashion, we
also increase the risk of overfitting. Therefore, the approach of locally stretching
the axes is much less common.
One additional practical issue in applying \textit{k-Nearest Neighbor} is efficient memory indexing. Because this algorithm delays all processing until a new query is received, significant computation can be required to process each new query. Various methods have been developed for indexing the stored training examples so that the nearest neighbors can be identified more efficiently at some additional cost in memory. One such indexing method is the \textit{kd-tree} (Bentley 1975; Friedman et al. 1977), in which instances are stored at the leaves of a tree, with nearby instances stored at the same or nearby nodes. The internal nodes of the tree sort the new query $x_q$ to the relevant leaf by testing selected attributes of $x_q$.

### 8.2.3 A Note on Terminology

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

### 8.3 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 $x = x_q$. Locally weighted regression is a generalization of this approach. It constructs an explicit approximation to $f$ over a local region surrounding $x_q$. Locally weighted regression uses nearby or distance-weighted training examples to form this local approximation to $f$. For example, we might approximate the target function in the neighborhood surrounding $x_q$ using a linear function, a quadratic function, a multilayer neural network, or some other functional form. The phrase “locally weighted regression” is called \textit{local} because the function is approximated based only on data near the query point, \textit{weighted} because the contribution of each training example is weighted by its distance from the query point, and \textit{regression} because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

Given a new query instance $x_q$, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding $x_q$. This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance. The description of $\hat{f}$ may then be deleted, because a different local approximation will be calculated for each distinct query instance.

#### 8.3.1 Locally Weighted Regression

Let us consider a regression function $\hat{f}$ that is expressed as

$$\hat{f}(x) = \sum_{i=1}^{n} w_i f_i(x),$$

where $n$ is the number of training examples, $\mathbf{w}$ is a vector of weights $w_i$, $\mathbf{f}$ is a vector of target values $f_i$, and $f(x)$ is a function of $x$. This equation can be expressed in matrix form as

$$\mathbf{y} = \mathbf{K} \mathbf{w},$$

where $\mathbf{y}$ is a vector of target values, $\mathbf{K}$ is a matrix of kernel functions $K(x_i, x)$ evaluated at every training point, and $\mathbf{w}$ is a vector of weights.

As before, we find the coefficients $w_i$ by minimizing an objective function $J$ that relates $\mathbf{y}$ and $\hat{f}(x)$. This is often more complex than a global objective function, because each weight $w_i$ is involved in fitting the value of $\hat{f}(x)$ at the query point $x$. One way of understanding this is to view $\mathbf{w}$ as a function $\mathbf{w}(\mathbf{x})$, which depends on the query point $x$. However, we write the objective function as a function of the weights $\mathbf{w}$, without explicitly making this dependence.

1. Minimize the function $J()$ to find $\mathbf{w}$ with

2. Minimize the function $J()$ to find $\mathbf{w}$ with

3. Combine the two functions $J()$ to find $\mathbf{w}$ with

Criterions for every training example.
8.3.1 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 \( i \)th attribute of the instance \( x \).

Recall that in Chapter 4 we discussed methods such as gradient descent to find the coefficients \( w_0, \ldots, w_n \) to minimize the error in fitting such linear functions to a given set of training examples. In that chapter we were interested in a global approximation to the target function. Therefore, we derived methods to choose weights that minimize the squared error summed over the set \( D \) of training examples

\[
E = \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2
\]

which led us to the gradient descent training rule

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

where \( \eta \) is a constant learning rate, and where the training rule has been re-expressed from the notation of Chapter 4 to fit our current notation (i.e., \( t \to f(x) \), \( o \to \hat{f}(x) \), and \( x_j \to a_j(x) \)).

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. Three possible criteria are given below. Note we write the error \( E(x_q) \) to emphasize the fact that now the error is being defined as a function of the query point \( x_q \).

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

\[
E_1(x_q) = \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2
\]

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

\[
E_2(x_q) = \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 K(d(x_q, x))
\]

3. Combine 1 and 2:

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

Criterion two is perhaps the most esthetically pleasing because it allows every training example to have an impact on the classification of \( x_q \). However,
this approach requires computation that grows linearly with the number of training examples. Criterion three is a good approximation to criterion two and has the advantage that computational cost is independent of the total number of training examples; its cost depends only on the number \( k \) of neighbors considered.

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):

\[
\Delta w_j = \eta \sum_{x \in k \text{ nearest nbs of } x_q} K(d(x_q, x)) (f(x) - \hat{f}(x)) a_j(x)
\] (8.7)

Notice the only differences between this new rule and the rule given by Equation (8.6) are that the contribution of instance \( x \) to the weight update is now multiplied by the distance penalty \( K(d(x_q, x)) \), and that the error is summed over only the \( k \) nearest training examples. 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 \( w_0 \ldots w_n \).

Atkeson et al. (1997a) and Bishop (1995) survey several such methods.

### 8.3.2 Remarks on Locally Weighted Regression

Above we considered using a linear function to approximate \( f \) in the neighborhood of the query instance \( x_q \). The literature on locally weighted regression contains a broad range of alternative methods for distance weighting the training examples, and a range of methods for locally approximating the target function. In most cases, the target function is approximated by a constant, linear, or quadratic function. More complex functional forms are not often found because (1) the cost of fitting more complex functions for each query instance is prohibitively high, and (2) these simple approximations model the target function quite well over a sufficiently small subregion of the instance space.

### 8.4 RADIAL BASIS FUNCTIONS

One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions (Powell 1987; Broomhead and Lowe 1988; Moody and Darken 1989). In this approach, the learned hypothesis is a function of the form

\[
\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))
\] (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 \( K_u \) to be centered at \( x_u \).

We will not pursue the design of \( K_u \) and the associated training algorithm in this book. As shown in Chapter 3, a simple approximation to a Gaussian is a polynomial of degree \( n \) in \( d(x_u, x) \) for a fixed-degree polynomial.

The above approach is typically used in the first-layer of a neural network with each neuron having a locally weighted function \( K_u(d(x_u, x)) \) for some \( m \). First-layer neurons on higher levels are typically trained with a backpropagation technique. This design is motivated by the large number of GPs that could be used to approximate any function with high accuracy if given a sufficiently large number of training examples.

GPs are often represented as the ratio of a polynomial degree \( n \) to the variance \( \sigma^2 \). Given the target function \( f(x) \), the variance of \( \hat{f}(x) \) is

\[
\text{var}(\hat{f}(x)) = \sigma^2 + \sum_{u=1}^{k} w_u^2 \text{var}(K_u(d(x_u, x)))
\]

Here \( \text{var}(K_u(d(x_u, x))) \) is the variance of the kernel function centered at \( x_u \). It is common to choose \( \sigma^2 \) to be small so that the variance of \( \hat{f}(x) \) is small as well as to prevent \( \hat{f}(x) \) from approximating \( f(x) \) too closely.
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 $\sigma_u^2$.

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

We will restrict our discussion here to this common Gaussian kernel function.
As shown by Hartman et al. (1990), the functional form of Equation (8.8) can
approximate any function with arbitrarily small error, provided a sufficiently large
number $k$ of such Gaussian kernels and provided the width $\sigma^2$ of each kernel can
be separately specified.

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 illus-
trated 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 $\sigma_u^2$ 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 $(x_i, f(x_i))$, centering
this Gaussian at the point $x_i$. Each of these kernels may be assigned the same width
$\sigma^2$. Given this approach, the RBF network learns a global approximation to the
target function in which each training example $(x_i, f(x_i))$ can influence the value of $\hat{f}$
only in the neighborhood of $x_i$. One advantage of this choice of kernel
functions is that it allows the RBF network to fit the training data exactly. That
is, for any set of $m$ training examples the weights $w_0 \ldots w_m$ for combining the
$m$ Gaussian kernel functions can be set so that $\hat{f}(x_i) = f(x_i)$ for each training
elementary example $(x_i, f(x_i))$.

$$f(x)$$

**FIGURE 8.2**
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at
some instance $x_u$. Therefore, its activation will be close to zero unless the input $x$ is near $x_u$. The output unit produces a linear
combination of the hidden unit activations. Although the network
shown here has just one output, multiple output units can also
be included.
A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large. 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 $X$. In this later case, we can pick kernel function centers by randomly selecting a subset of the training instances, thereby sampling the underlying distribution of instances. Alternatively, we may identify prototypical clusters of instances, then add a kernel function centered at each cluster. The placement of the kernel functions in this fashion can be accomplished using unsupervised clustering algorithms that fit the training instances (but not their target values) to a mixture of Gaussians. The EM algorithm discussed in Section 6.12.1 provides one algorithm for choosing the means of a mixture of $k$ Gaussians to best fit the observed instances. In the case of the EM algorithm, the means are chosen to maximize the probability of observing the instances $x_i$, given the $k$ estimated means. Note the target function value $f(x_i)$ of the instance does not enter into the calculation of kernel centers by unsupervised clustering methods. The only role of the target values $f(x_i)$ in this case is to determine the output layer weights $w_u$.

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 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.

### 8.5 CASE-BASED REASONING

Instance-based methods such as $k$-NEAREST NEIGHBOR and 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. CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs (Sycara et al. 1992), reasoning about new legal cases based on previous rulings (Ashley 1990), and solving planning and scheduling problems.