12 *Local Models*

We continue our discussion of multilayer neural networks with models where the first layer contains locally receptive units that respond to instances in a localized region of the input space. The second layer on top learns the regression or classification function for these local regions. We discuss learning methods for finding the local regions of importance as well as the models responsible in there.

12.1 Introduction

ONE WAY to do function approximation is to divide the input space into local patches and learn a separate fit in each local patch. In chapter 7, we discussed statistical methods for clustering that allowed us to group input instances and model the input distribution. Competitive methods are neural network methods for online clustering. In this chapter, we discuss the online version of k-means, as well as two neural network extensions, adaptive resonance theory (ART), and the self-organizing map (SOM).

We then discuss how supervised learning is implemented once the inputs are localized. If the fit in a local patch is constant, then the technique is named the radial basis function (RBF) network; if it is a linear function of the input, it is called the mixture of experts (MoE). We discuss both regression and classification, and also compare this approach with MLP, which we discussed in chapter 11.

12.2 Competitive Learning

In chapter 7, we used the semiparametric Gaussian mixture density, which assumes that the input comes from one of k Gaussian sources. In this section, we make the same assumption that there are k groups (or clusters) in the data, but our approach is not probabilistic in that we do not enforce a parametric model for the sources. Another difference is that the learning methods we propose are online. We do not have the whole sample at hand during training; we receive instances one by one and update model parameters as we get them. The term *competitive learning* is used because it is as if these groups, or rather the units representing these groups, compete among themselves to be the one responsible for representing an instance. The model is also called *winner-take-all*; it is as if one group wins and gets updated, and the others are not updated at all.

COMPETITIVE LEARNING

WINNER-TAKE-ALL

These methods can be used by themselves for online clustering, as opposed to the batch methods discussed in chapter 7. An online method has the usual advantages that (1) we do not need extra memory to store the whole training set; (2) updates at each step are simple to implement, for example, in hardware; and (3) the input distribution may change in time and the model adapts itself to these changes automatically. If we were to use a batch algorithm, we would need to collect a new sample and run the batch method from scratch over the whole sample.

Starting in section 12.3, we will also discuss how such an approach can be followed by a supervised method to learn regression or classification problems. This will be a two-stage system that can be implemented by a two-layer network, where the first stage (-layer) models the input density and finds the responsible local model, and the second stage is that of the local model generating the final output.

12.2.1 Online k-Means

In equation 7.3, we defined the reconstruction error as

$$E(\{\mathbf{m}_i\}_{i=1}^k|\mathcal{X}) = \frac{1}{2}\sum_t \sum_i b_i^t \|\mathbf{x}^t - \mathbf{m}_i\|^2 \tag{12.1}$$

where

$$b_i^t = \begin{cases} 1 & \text{if } \|\mathbf{x}^t - \mathbf{m}_i\| = \min_l \|\mathbf{x}^t - \mathbf{m}_l\| \\ 0 & \text{otherwise} \end{cases} \tag{12.2}$$

$\mathcal{X} = \{\boldsymbol{x}^t\}_t$ is the sample and $\boldsymbol{m}_i, i = 1, \ldots, k$ are the cluster centers. b_i^t is 1 if $\boldsymbol{m}_i$ is the closest center to $\boldsymbol{x}^t$ in Euclidean distance. It is as if all $\boldsymbol{m}_l, l = 1, \ldots, k$ compete and $\boldsymbol{m}_i$ wins the competition because it is the closest.

The batch algorithm, k-means, updates the centers as

$$\boldsymbol{m}_i = \frac{\sum_t b_i^t \boldsymbol{x}^t}{\sum_t b_i^t} \tag{12.3}$$

which minimizes equation 12.1, once the winners are chosen using equation 12.2. As we saw before, these two steps of calculating b_i^t and updating $\boldsymbol{m}_i$ are iterated until convergence.

ONLINE k-MEANS

We can obtain *online k-means* by doing stochastic gradient descent, considering the instances one by one, and doing a small update at each step, not forgetting the effect of the previous updates. The reconstruction error for a single instance is

$$E^t(\{\boldsymbol{m}_i\}_{i=1}^k | \boldsymbol{x}^t) = \frac{1}{2}\sum_i b_i^t \|\boldsymbol{x}^t - \boldsymbol{m}_i\|^2 = \frac{1}{2}\sum_i \sum_{j=1}^d b_i^t (x_j^t - m_{ij})^2 \tag{12.4}$$

where b_i^t is defined as in equation 12.2. Using gradient descent on this, we get the following update rule for each instance $\boldsymbol{x}^t$:

$$\Delta m_{ij} = -\eta \frac{\partial E^t}{\partial m_{ij}} = \eta b_i^t (x_j^t - m_{ij}) \tag{12.5}$$

This moves the closest center (for which $b_i^t = 1$) toward the input by a factor given by η. The other centers have their $b_l^t, l \neq i$ equal to 0 and are not updated (see figure 12.1). A batch procedure can also be defined by summing up equation 12.5 over all t. Like in any gradient descent procedure, a momentum term can also be added. For convergence, η is gradually decreased to 0. But this implies the *stability-plasticity dilemma*: If η is decreased toward 0, the network becomes stable but we lose adaptivity to novel patterns that may occur in time because updates become too small. If we keep η large, $\boldsymbol{m}_i$ may oscillate.

STABILITY-PLASTICITY DILEMMA

The pseudocode of online k-means is given in figure 12.2. This is the online version of the batch algorithm given in figure 7.3.

The competitive network can be implemented as a one-layer recurrent network as shown in figure 12.3. The input layer contains the input vector $\boldsymbol{x}$; note that there is no bias unit. The values of the output units are the b_i and they are perceptrons:

$$b_i = \boldsymbol{m}_i^T \boldsymbol{x} \tag{12.6}$$

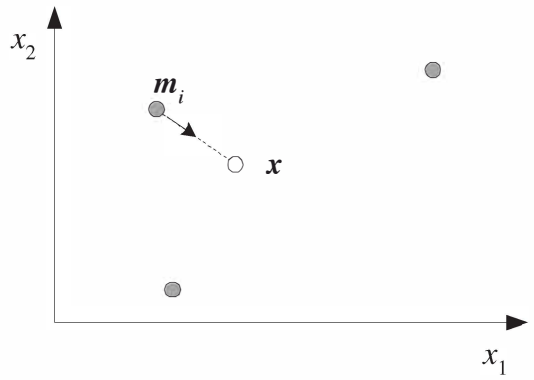


Figure 12.1 Shaded circles are the centers and the empty circle is the input instance. The online version of k-means moves the closest center along the direction of $(\boldsymbol{x} - \boldsymbol{m}_i)$ by a factor specified by η.

LATERAL INHIBITION

Then we need to choose the maximum of the b_i and set it equal to 1, and set the others, $b_l, l \neq i$ to 0. If we would like to do everything purely neural, that is, using a network of concurrently operating processing units, the choosing of the maximum can be implemented through *lateral inhibition*. As shown in figure 12.3, each unit has an excitatory recurrent connection (i.e., with a positive weight) to itself, and inhibitory recurrent connections (i.e., with negative weights) to the other output units. With an appropriate nonlinear activation function and positive and negative recurrent weight values, such a network, after some iterations, converges to a state where the maximum becomes 1 and all others become 0 (Grossberg 1980; Feldman and Ballard 1982).

The dot product used in equation 12.6 is a similarity measure, and we saw in section 5.5 (equation 5.26) that if $\boldsymbol{m}_i$ have the same norm, then the unit with the minimum Euclidean distance, $\|\boldsymbol{m}_i - \boldsymbol{x}\|$, is the same as the one with the maximum dot product, $\boldsymbol{m}_i^T \boldsymbol{x}$.

Here, and later when we discuss other competitive methods, we use the Euclidean distance, but we should keep in mind that using the Euclidean distance implies that all input attributes have the same variance and that they are not correlated. If this is not the case, this should be reflected in the distance measure, that is, by using the Mahalanobis distance, or suitable normalization should be done, for example, by PCA, at

Initialize $\mathbf{m}_i, i = 1, \ldots, k$, for example, to k random $\mathbf{x}^t$
Repeat
  For all $\mathbf{x}^t \in \mathcal{X}$ in random order
    $i \leftarrow \arg\min_j \|\mathbf{x}^t - \mathbf{m}_j\|$
    $\mathbf{m}_i \leftarrow \mathbf{m}_i + \eta(\mathbf{x}^t - \mathbf{m}_i)$
Until $\mathbf{m}_i$ converge

Figure 12.2 Online k-means algorithm. The batch version is given in figure 7.3.

a preprocessing stage before the Euclidean distance is used.

We can rewrite equation 12.5 as

$$\Delta m_{ij}^t = \eta b_i^t x_j^t - \eta b_i^t m_{ij} \tag{12.7}$$

Let us remember that m_{ij} is the weight of the connection from x_j to b_i. An update of the form, as we see in the first term

$$\Delta m_{ij}^t = \eta b_i^t x_j^t \tag{12.8}$$

HEBBIAN LEARNING

is *Hebbian learning*, which defines the update as the product of the values of the presynaptic and postsynaptic units. It was proposed as a model for neural plasticity: A synapse becomes more important if the units before and after the connection fire simultaneously, indicating that they are correlated. However, with only Hebbian learning, the weights grow without bound ($x_j^t \geq 0$), and we need a second force to decrease the weights that are not updated. One possibility is to explicitly normalize the weights to have $\|\mathbf{m}_i\| = 1$; if $\Delta m_{ij} > 0$ and $\Delta m_{il} = 0, l \neq j$, once we normalize $\mathbf{m}_i$ to unit length, m_{il} decrease. Another possibility is to introduce a weight decay term (Oja 1982), and the second term of equation 12.7 can be seen as such. Hertz, Krogh, and Palmer (1991) discuss competitive networks and Hebbian learning in more detail and show, for example, how such networks can learn to do PCA. Mao and Jain (1995) discuss online algorithms for PCA and LDA.

As we saw in chapter 7, one problem is to avoid dead centers, namely, the ones that are there but are not effectively utilized. In the case of competitive networks, this corresponds to centers that never win the competition because they are initialized far away from any input. There are various ways we can avoid this:

1. We can initialize $\mathbf{m}_i$ by randomly chosen input instances, and make sure that they start from where there is data.

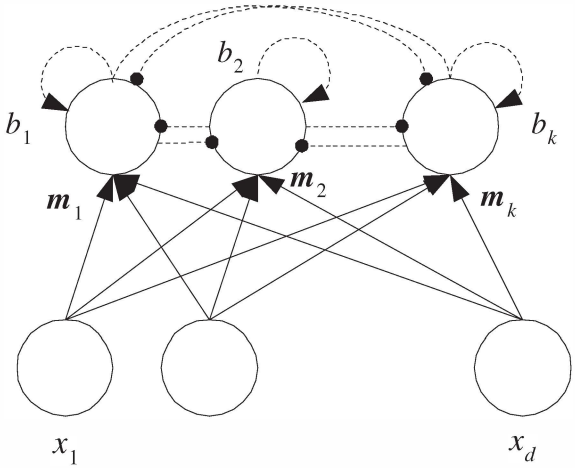


Figure 12.3 The winner-take-all competitive neural network, which is a network of k perceptrons with recurrent connections at the output. Dashed lines are recurrent connections, of which the ones that end with an arrow are excitatory and the ones that end with a circle are inhibitory. Each unit at the output reinforces its value and tries to suppress the other outputs. Under a suitable assignment of these recurrrent weights, the maximum suppresses all the others. This has the net effect that the one unit whose $\boldsymbol{m}_i$ is closest to $\boldsymbol{x}$ ends up with its b_i equal to 1 and all others, namely, $b_l, l \neq i$ are 0.

2. We can use a leader-cluster algorithm and add units one by one, always adding them at a place where they are needed. One example is the ART model, which we discuss in section 12.2.2.

3. When we update, we do not update only the center of the closest unit but some others as well. As they are updated, they also move toward the input, move gradually toward parts of the input space where there are inputs, and eventually win the competition. One example that we discuss in section 12.2.3 is SOM.

4. Another possibility is to introduce a *conscience* mechanism (DeSieno 1988): A unit that has won the competition recently feels guilty and allows others to win.

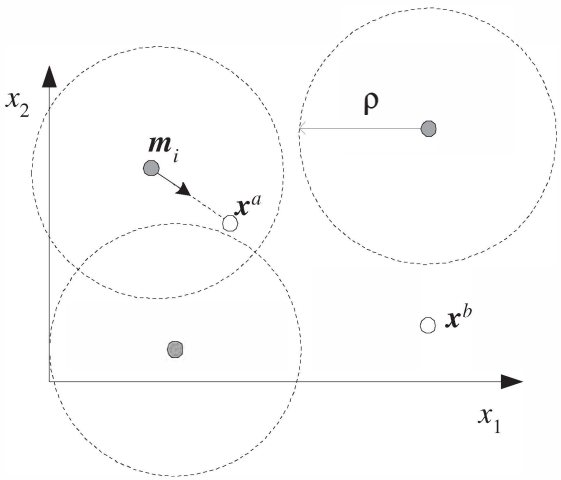


Figure 12.4 The distance from $\boldsymbol{x}^a$ to the closest center is less than the vigilance value ρ and the center is updated as in online k-means. However, $\boldsymbol{x}^b$ is not close enough to any of the centers and a new group should be created at that position.

12.2.2 Adaptive Resonance Theory

The number of groups, k, should be known and specified before the parameters can be calculated. Another approach is *incremental*, where one starts with a single group and adds new groups as they are needed. We discuss the *adaptive resonance theory* (ART) algorithm (Carpenter and Grossberg 1988) as an example of an incremental algorithm. In ART, given an input, all of the output units calculate their values and the one most similar to the input is chosen. This is the unit with the maximum value if the unit uses the dot product as in equation 12.6, or it is the unit with the minimum value if the unit uses the Euclidean distance.

ADAPTIVE RESONANCE THEORY

Let us assume that we use the Euclidean distance. If the minimum value is smaller than a certain threshold value, named the *vigilance*, the update is done as in online k-means. If this distance is larger than vigilance, a new output unit is added and its center is initialized with the instance. This defines a hypersphere whose radius is given by the vigilance defining the volume of scope of each unit; we add a new unit whenever we have an input that is not covered by any unit (see figure 12.4).

VIGILANCE

Denoting vigilance by ρ, we use the following equations at each update:

$$b_i = \|\boldsymbol{m}_i - \boldsymbol{x}^t\| = \min_{l=1}^{k} \|\boldsymbol{m}_l - \boldsymbol{x}^t\| \tag{12.9}$$

$$\begin{cases} \boldsymbol{m}_{k+1} \leftarrow \boldsymbol{x}^t & \text{if } b_i > \rho \\ \Delta \boldsymbol{m}_i = \eta(\boldsymbol{x}^t - \boldsymbol{m}_i) & \text{otherwise} \end{cases}$$

Putting a threshold on distance is equivalent to putting a threshold on the reconstruction error per instance, and if the distance is Euclidean and the error is defined as in equation 12.4, this indicates that the maximum reconstruction error allowed per instance is the square of vigilance.

12.2.3 Self-Organizing Maps

SELF-ORGANIZING MAP

One way to avoid having dead units is by updating not only the winner but also some of the other units as well. In the *self-organizing map* (SOM) proposed by Kohonen (1990, 1995), unit indices, namely, i as in $\boldsymbol{m}_i$, define a *neighborhood* for the units. When $\boldsymbol{m}_i$ is the closest center, in addition to $\boldsymbol{m}_i$, its neighbors are also updated. For example, if the neighborhood is of size 2, then $\boldsymbol{m}_{i-2}, \boldsymbol{m}_{i-1}, \boldsymbol{m}_{i+1}, \boldsymbol{m}_{i+2}$ are also updated but with less weight as the neighborhood increases. If i is the index of the closest center, the centers are updated as

$$\Delta \boldsymbol{m}_l = \eta\, e(l,i)(\boldsymbol{x}^t - \boldsymbol{m}_l) \tag{12.10}$$

where $e(l,i)$ is the *neighborhood function*. $e(l,i) = 1$ when $l = i$ and decreases as $|l - i|$ increases, for example, as a Gaussian, $\mathcal{N}(i, \sigma)$:

$$e(l,i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(l-i)^2}{2\sigma^2}\right] \tag{12.11}$$

For convergence, the support of the neighborhood function decreases in time, for example, σ decreases, and at the end, only the winner is updated.

Because neighboring units are also moved toward the input, we avoid dead units since they get to win competition sometime later, after a little bit of initial help from their neighboring friends (see figure 12.5).

Updating the neighbors has the effect that, even if the centers are randomly initialized, because they are moved toward the same input together, once the system converges, units with neighboring indices will also be neighbors in the input space.

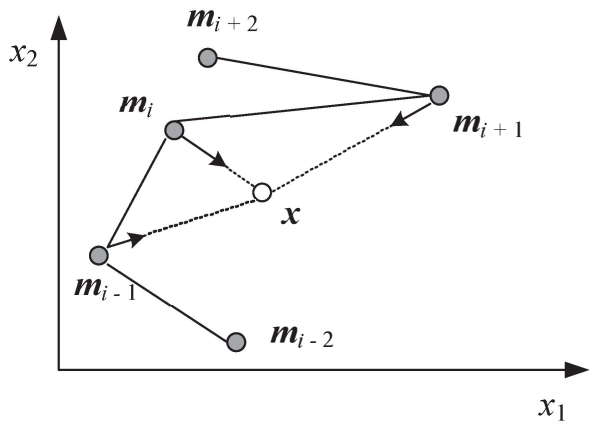


Figure 12.5 In the SOM, not only the closest unit but also its neighbors, in terms of indices, are moved toward the input. Here, neighborhood is 1; $\boldsymbol{m}_i$ and its 1-nearest neighbors are updated. Note here that $\boldsymbol{m}_{i+1}$ is far from $\boldsymbol{m}_i$, but as it is updated with $\boldsymbol{m}_i$, and as $\boldsymbol{m}_i$ will be updated when $\boldsymbol{m}_{i+1}$ is the winner, they will become neighbors in the input space as well.

In most applications, the units are organized as a two-dimensional *map*. That is, each unit will have two indices, $\boldsymbol{m}_{i,j}$, and the neighborhood will be defined in two dimensions. If $\boldsymbol{m}_{i,j}$ is the closest center, the centers are updated as

$$\Delta \boldsymbol{m}_{k,l} = \eta e(k,l,i,j)(\boldsymbol{x}^t - \boldsymbol{m}_{k,l}) \tag{12.12}$$

where the neighborhood function is now in two dimensions. After convergence, this forms a two-dimensional *topographical map* of the original d-dimensional input space. The map contains many units in parts of the space where density is high, and no unit will be dedicated to parts where there is no input. Once the map converges, inputs that are close in the original space are mapped to units that are close in the map. In this regard, the map can be interpreted as doing a nonlinear form of multidimensional scaling, mapping from the original $\boldsymbol{x}$ space to the two dimensions, (i, j). Similarly, if the map is one-dimensional, the units are placed on the curve of maximum density in the input space, as a *principal curve*.

TOPOGRAPHICAL MAP

12.3 Radial Basis Functions

In a multilayer perceptron (chapter 11) where hidden units use the dot product, each hidden unit defines a hyperplane and with the sigmoid nonlinearity, a hidden unit has a value between 0 and 1, coding the position of the instance with respect to the hyperplane. Each hyperplane divides the input space in two, and typically for a given input, many of the hidden units have nonzero output. This is called a *distributed representation* because the input is encoded by the simultaneous activation of many hidden units.

DISTRIBUTED REPRESENTATION

LOCAL REPRESENTATION

Another possibility is to have a *local representation* where for a given input, only one or a few units are active. It is as if these *locally tuned units* partition the input space among themselves and are selective to only certain inputs. The part of the input space where a unit has nonzero response is called its *receptive field*. The input space is then paved with such units.

RECEPTIVE FIELD

Neurons with such response characteristics are found in many parts of the cortex. For example, cells in the visual cortex respond selectively to stimulation that is both local in retinal position and local in angle of visual orientation. Such locally tuned cells are typically arranged in topographical cortical maps in which the values of the variables to which the cells respond vary by their position in the map, as in a SOM.

The concept of locality implies a distance function to measure the similarity between the given input $\boldsymbol{x}$ and the position of unit h, $\boldsymbol{m}_h$. Frequently this measure is taken as the Euclidean distance, $\|\boldsymbol{x} - \boldsymbol{m}_h\|$. The response function is chosen to have a maximum where $\boldsymbol{x} = \boldsymbol{m}_h$ and decreasing as they get less similar. Commonly we use the Gaussian function (see figure 12.6):

$$p_h^t = \exp\left[-\frac{\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2}{2s_h^2}\right] \tag{12.13}$$

Strictly speaking, this is not Gaussian density, but we use the same name anyway. $\boldsymbol{m}_j$ and s_j respectively denote the center and the spread of the local unit j, and as such define a radially symmetric basis function. One can use an elliptic one with different spreads on different dimensions, or even use the full Mahalanobis distance to allow correlated inputs, at the expense of using a more complicated model (exercise 2).

The idea in using such local basis functions is that in the input data, there are groups or clusters of instances and for each such cluster, we

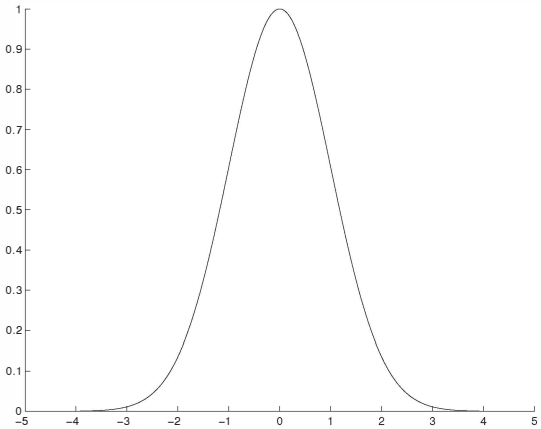


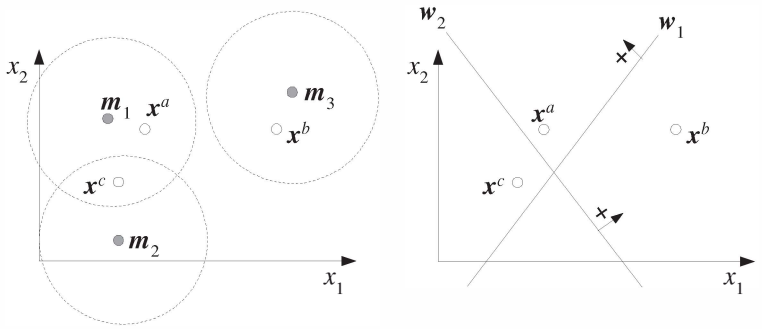
Figure 12.6 The one-dimensional form of the bell-shaped function used in the radial basis function network. This one has $m = 0$ and $s = 1$. It is like a Gaussian but it is not a density; it does not integrate to 1. It is nonzero between $(m - 3s, m + 3s)$, but a more conservative interval is $(m - 2s, m + 2s)$.

define a basis function, p_h^t, which becomes nonzero if instance $\boldsymbol{x}^t$ belongs to cluster h. One can use any of the online competitive methods discussed in section 12.2 to find the centers, $\boldsymbol{m}_h$. There is a simple and effective heuristic to find the spreads: Once we have the centers, for each cluster, we find the most distant instance covered by that cluster and set s_h to half its distance from the center. We could have used one-third, but we prefer to be conservative. We can also use the statistical clustering method, for example, EM on Gaussian mixtures, that we discussed in chapter 7 to find the cluster parameters, namely, means, variances (and covariances).

$p_h^t, h = 1, \ldots, H$ define a new H-dimensional space and form a new representation of $\boldsymbol{x}^t$. We can also use b_h^t (equation 12.2) to code the input but b_h^t are 0/1; p_h^t have the additional advantage that they code the distance to their center by a value in $(0, 1)$. How fast the value decays to 0 depends on s_h. Figure 12.7 gives an example and compares such a *local representation* with a *distributed* representation as used by the multilayer perceptron. Because Gaussians are local, typically we need many more local units than what we would need if we were to use a distributed representation, especially if the input is high-dimensional.

DISTRIBUTED VS LOCAL REPRESENTATION

In the case of supervised learning, we can then use this new local rep-



Local representation in the space of (p_1, p_2, p_3)

$\boldsymbol{x}^a$: (1.0, 0.0, 0.0)
$\boldsymbol{x}^b$: (0.0, 0.0, 1.0)
$\boldsymbol{x}^c$: (1.0, 1.0, 0.0)

Distributed representation in the space of (h_1, h_2)

$\boldsymbol{x}^a$: (1.0, 1.0)
$\boldsymbol{x}^b$: (0.0, 1.0)
$\boldsymbol{x}^c$: (1.0, 0.0)

Figure 12.7 The difference between local and distributed representations. The values are hard, 0/1, values. One can use soft values in $(0, 1)$ and get a more informative encoding. In the local representation, this is done by the Gaussian RBF that uses the distance to the center, $\boldsymbol{m}_i$, and in the distributed representation, this is done by the sigmoid that uses the distance to the hyperplane, $\boldsymbol{w}_i$.

resentation as the input. If we use a perceptron, we have

$$y^t = \sum_{h=1}^{H} w_h p_h^t + w_0 \tag{12.14}$$

RADIAL BASIS FUNCTION

where H is the number of basis functions. This structure is called a *radial basis function* (RBF) network (Broomhead and Lowe 1988; Moody and Darken 1989). Normally, people do not use RBF networks with more than one layer of Gaussian units. H is the complexity parameter, like the number of hidden units in a multilayer perceptron. Previously we denoted it by k, when it corresponded to the number of centers in the case of unsupervised learning.

Here, we see the advantage of using p_h instead of b_h. Because b_h are 0/1, if equation 12.14 contained b_h instead of the p_h, it would give a piecewise constant approximation with discontuinities at the unit region boundaries. p_h values are soft and lead to a smooth approximation, taking a weighted average while passing from one region to another. We can

easily see that such a network is a universal approximator in that it can approximate any function with desired accuracy, given enough units. We can form a grid in the input space to our desired accuracy, define a unit that will be active for each cell, and set its outgoing weight, w_h, to the desired output value.

This architecture bears much similarity to the nonparametric estimators, for example, Parzen windows, we saw in chapter 8, and p_h may be seen as kernel functions. The difference is that now we do not have a kernel function over all training instances but group them using a clustering method to make do with fewer kernels. H, the number of units, is the complexity parameter, trading off simplicity and accuracy. With more units, we approximate the training data better, but we get a complex model and risk overfitting; too few may underfit. Again, the optimal value is determined by cross-validation.

Once $\mathbf{m}_h$ and s_h are given and fixed, p_h are also fixed. Then w_h can be trained easily batch or online. In the case of regression, this is a linear regression model (with p_h as the inputs) and the w_h can be solved analytically without any iteration (section 4.6). In the case of classification, we need to resort to an iterative procedure. We discussed learning methods for this in chapter 10 and do not repeat them here.

What we do here is a two-stage process: We use an unsupervised method for determining the centers, then build a supervised layer on top of that. This is called *hybrid learning*. We can also learn all parameters, including $\mathbf{m}_h$ and s_h, in a supervised manner. The radial basis function of equation 12.13 is differentiable and we can backpropagate, just as we backpropagated in a multilayer perceptron to update the first-layer weights. The structure is similar to a multilayer perceptron with p_h as the hidden units, $\mathbf{m}_h$ and s_h as the first-layer parameters, the Gaussian as the activation function in the hidden layer, and w_h as the second-layer weights (see figure 12.8).

HYBRID LEARNING

But before we discuss this, we should remember that training a two-layer network is slow. Hybrid learning trains one layer at a time and is faster. Another technique, called the *anchor* method, sets the centers to the randomly chosen patterns from the training set without any further update. It is adequate if there are many units.

ANCHOR

On the other hand, the accuracy normally is not as high as when a completely supervised method is used. Consider the case when the input is uniformly distributed. Then k-means clustering places the units uniformly. If the function is changing significantly in a small part of the

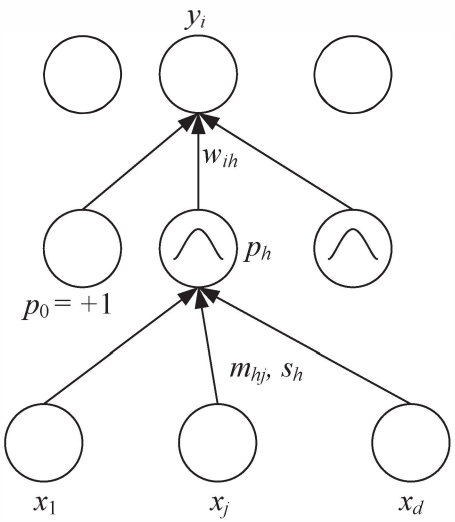


Figure 12.8 The RBF network where p_h are the hidden units using the bell-shaped activation function. $\boldsymbol{m}_h, s_h$ are the first-layer parameters, and $\boldsymbol{w}_i$ are the second-layer weights.

space, it is a better idea to have as many centers in places where the function changes fast, to make the error as small as possible; this is what the completely supervised method would do.

Let us discuss how all of the parameters can be trained in a fully supervised manner. The approach is the same as backpropagation applied to multilayer perceptrons. Let us see the case of regression with multiple outputs. The batch error is

$$E(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \frac{1}{2}\sum_t \sum_i (r_i^t - y_i^t)^2 \tag{12.15}$$

where

$$y_i^t = \sum_{h=1}^{H} w_{ih} p_h^t + w_{i0} \tag{12.16}$$

Using gradient descent, we get the following update rule for the second-layer weights:

$$\Delta w_{ih} = \eta \sum_t (r_i^t - y_i^t) p_h^t \tag{12.17}$$

This is the usual perceptron update rule, with p_h as the inputs. Typically, p_h do not overlap much and at each iteration, only a few p_h are nonzero and only their w_h are updated. That is why RBF networks learn very fast, and faster than multilayer perceptrons that use a distributed representation.

Similarly, we can get the update equations for the centers and spreads by backpropagation (chain rule):

$$\Delta m_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) w_{ih} \right] p_h^t \frac{(x_j^t - m_{hj})}{s_h^2} \tag{12.18}$$

$$\Delta s_h = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) w_{ih} \right] p_h^t \frac{\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2}{s_h^3} \tag{12.19}$$

Let us compare equation 12.18 with equation 12.5: First, here we use p_h instead of b_h, which means that not only the closest one but all units are updated, depending on their centers and spreads. Second, here the update is supervised and contains the backpropagated error term. The update depends not only on the input but also on the final error $(r_i^t - y_i^t)$, the effect of the unit on the output, w_{ih}, the activation of the unit, p_h, and the input, $(\boldsymbol{x} - \boldsymbol{m}_h)$.

In practice, equations 12.18 and 12.19 need some extra control. We need to explicitly check that s_h do not become very small or very large to be useless; we also need to check that $\boldsymbol{m}_h$ stay in the valid input range.

In the case of classification, we have

$$y_i^t = \frac{\exp\left[\sum_h w_{ih} p_h^t + w_{i0}\right]}{\sum_k \exp\left[\sum_h w_{kh} p_h^t + w_{k0}\right]} \tag{12.20}$$

and the cross-entropy error is

$$E(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h} | \mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t \tag{12.21}$$

Update rules can similarly be derived using gradient descent (exercise 3).

Let us look again at equation 12.14. For any input, if p_h is nonzero, then it contributes w_h to the output. Its contribution is a constant fit, as given by w_h. Normally Gaussians do not overlap much, and one or two of them have a nonzero p_h value. In any case, only few units contribute to the output. w_0 is the constant offset and is added to the weighted sum

of the active (nonzero) units. We also see that $y = w_0$ if all p_h are 0. We can therefore view w_0 as the "default" value of y: If no Gaussian is active, then the output is given by this value. So a possibility is to make this "default model" a more powerful "rule." For example, we can write

$$y^t = \underbrace{\sum_{h=1}^{H} w_h p_h^t}_{exceptions} + \underbrace{\mathbf{v}^T \mathbf{x}^t + v_0}_{rule} \tag{12.22}$$

In this case, the rule is linear: $\mathbf{v}^T \mathbf{x}^t + v_0$. When they are nonzero, Gaussians work as localized "exceptions" and modify the output to make up for the difference between the desired output and the rule output. Such a model can be trained in a supervised manner, and the rule can be trained together with the exceptions (exercise 4). We discuss a similar model, cascading, in section 17.11 where we see it as a combination of two learners, one general rule and the other formed by a set of exceptions.

12.4 Incorporating Rule-Based Knowledge

PRIOR KNOWLEDGE

The training of any learning system can be much simpler if we manage to incorporate *prior knowledge* to initialize the system. For example, prior knowledge may be available in the form of a set of rules that specify the input/output mapping that the model, for example, the RBF network, has to learn. This occurs frequently in industrial and medical applications where rules can be given by experts. Similarly, once a network has been trained, rules can be extracted from the solution in such a way as to better understand the solution to the problem.

The inclusion of prior knowledge has the additional advantage that if the network is required to extrapolate into regions of the input space where it has not seen any training data, it can rely on this prior knowledge. Furthermore, in many control applications, the network is required to make reasonable predictions right from the beginning. Before it has seen sufficient training data, it has to rely primarily on this prior knowledge.

In many applications we are typically told some basic rules that we try to follow in the beginning but that are then refined and altered through experience. The better our initial knowledge of a problem, the faster we can achieve good performance and the less training that is required.

Such inclusion of prior knowledge or extraction of learned knowledge is easy to do with RBF networks because the units are local. This makes *rule extraction* easier (Tresp, Hollatz, and Ahmad 1997). An example is

RULE EXTRACTION

$$\text{IF } ((x_1 \approx a) \text{ AND } (x_2 \approx b)) \text{ OR } (x_3 \approx c) \text{ THEN } y = 0.1 \tag{12.23}$$

where $x_1 \approx a$ means "x_1 is approximately a." In the RBF framework, this rule is encoded by two Gaussian units as

$$p_1 = \exp\left[-\frac{(x_1-a)^2}{2s_1^2}\right] \cdot \exp\left[-\frac{(x_2-b)^2}{2s_2^2}\right] \text{ with } w_1 = 0.1$$

$$p_2 = \exp\left[-\frac{(x_3-c)^2}{2s_3^2}\right] \text{ with } w_2 = 0.1$$

"Approximately equal to" is modeled by a Gaussian where the center is the ideal value and the spread denotes the allowed difference around this ideal value. Conjunction is the product of two univariate Gaussians that is a bivariate Gaussian. Then, the first product term can be handled by a two-dimensional, namely, $\boldsymbol{x} = [x_1, x_2]$, Gaussian centered at (a, b), and the spreads on the two dimensions are given by s_1 and s_2. Disjunction is modeled by two separate Gaussians, each one handling one of the disjuncts.

Given labeled training data, the parameters of the RBF network so constructed can be fine-tuned after the initial construction, using a small value of η.

This formulation is related to the fuzzy logic approach where equation 12.23 is named a *fuzzy rule*. The Gaussian basis function that checks for approximate equality corresponds to a *fuzzy membership function* (Berthold 1999; Cherkassky and Mulier 1998).

FUZZY RULE

FUZZY MEMBERSHIP FUNCTION

12.5 Normalized Basis Functions

In equation 12.14, for an input, it is possible that all of the p_h are 0. In some applications, we may want to have a *normalization* step to make sure that the values of the local units sum up to 1, thus making sure that for any input there is at least one nonzero unit:

$$g_h^t = \frac{p_h^t}{\sum_{l=1}^{H} p_l^t} = \frac{\exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2/2s_h^2]}{\sum_l \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_l\|^2/2s_l^2]} \tag{12.24}$$

An example is given in figure 12.9. Taking p_h as $p(\boldsymbol{x}|h)$, g_h correspond to $p(h|\boldsymbol{x})$, the posterior probability that $\boldsymbol{x}$ belongs to unit h. It is as if

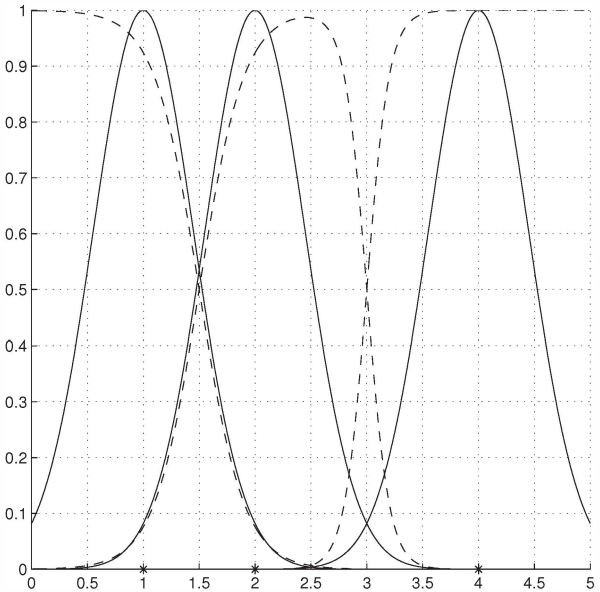


Figure 12.9 (-) Before and (- -) after normalization for three Gaussians whose centers are denoted by '*'. Note how the nonzero region of a unit depends also on the positions of other units. If the spreads are small, normalization implements a harder split; with large spreads, units overlap more.

the units divide the input space among themselves. We can think of g_h as a classifier in itself, choosing the responsible unit for a given input. This classification is done based on distance, as in a parametric Gaussian classifier (chapter 5).

The output is a weighted sum

$$y_i^t = \sum_{h=1}^{H} w_{ih} g_h^t \tag{12.25}$$

where there is no need for a bias term because there is at least one nonzero g_h for each $\boldsymbol{x}$. Using g_h instead of p_h does not introduce any extra parameters; it only couples the units together: p_h depends only on $\boldsymbol{m}_h$ and s_h, but g_h, because of normalization, depends on the centers and spreads of all of the units.

In the case of regression, we have the following update rules using gradient descent:

$$\Delta w_{ih} = \eta \sum_t (r_i^t - y_i^t) g_h^t \tag{12.26}$$

$$\Delta m_{hj} = \eta \sum_t \sum_i (r_i^t - y_i^t)(w_{ih} - y_i^t) g_h^t \frac{(x_j^t - m_{hj})}{s_h^2} \tag{12.27}$$

The update rule for s_h as well as the rules for classification can similarly be derived. Let us compare these with the update rules for the RBF with unnormalized Gaussians (equation 12.17). Here, we use g_h instead of p_h, which makes a unit's update dependent not only on its own parameters, but also on the centers and spreads of other units as well. Comparing equation 12.27 with equation 12.18, we see that instead of w_{ih}, we have $(w_{ih} - y_i^t)$, which shows the role of normalization on the output. The "responsible" unit wants to decrease the difference between its output, w_{ih}, and the final output, y_i^t, proportional to its responsibility, g_h.

12.6 Competitive Basis Functions

As we have seen up until now, in an RBF network the final output is determined as a weighted sum of the contributions of the local units. Though the units are local, it is the final weighted sum that is important and that we want to make as close as possible to the required output. For example, in regression we minimize equation 12.15, which is based on the probabilistic model

$$p(\boldsymbol{r}^t|\boldsymbol{x}^t) = \prod_i \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(r_i^t - y_i^t)^2}{2\sigma^2}\right] \tag{12.28}$$

where y_i^t is given by equation 12.16 (unnormalized) or equation 12.25 (normalized). In either case, we can view the model as a *cooperative* one since the units cooperate to generate the final output, y_i^t. We now discuss the approach using *competitive basis functions* where we assume that the output is drawn from a mixture model

COMPETITIVE BASIS FUNCTIONS

$$p(\boldsymbol{r}^t|\boldsymbol{x}^t) = \sum_{h=1}^{H} p(h|\boldsymbol{x}^t) p(\boldsymbol{r}^t|h, \boldsymbol{x}^t) \tag{12.29}$$

$p(h|\boldsymbol{x}^t)$ are the mixture proportions and $p(\boldsymbol{r}^t|h, \boldsymbol{x}^t)$ are the mixture components generating the output if that component is chosen. Note that both of these terms depend on the input $\boldsymbol{x}$.

The mixture proportions are

$$p(h|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|h)p(h)}{\sum_l p(\boldsymbol{x}|l)p(l)} \tag{12.30}$$

$$g_h^t = \frac{a_h \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2/2s_h^2]}{\sum_l a_l \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_l\|^2/2s_l^2]} \tag{12.31}$$

We generally assume a_h to be equal and ignore them. Let us first take the case of regression where the components are Gaussian. In equation 12.28, noise is added to the weighted sum; here, one component is chosen and noise is added to its output, y_{ih}^t.

Using the mixture model of equation 12.29, the log likelihood is

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \exp\left[-\frac{1}{2}\sum_i (r_i^t - y_{ih}^t)^2\right] \tag{12.32}$$

where $y_{ih}^t = w_{ih}$ is the constant fit done by component h for output i, which, strictly speaking, does not depend on $\boldsymbol{x}$. (In section 12.8.2, we discuss the case of competitive mixture of experts where the local fit is a linear function of $\boldsymbol{x}$.) We see that if g_h^t is 1, then it is responsible for generating the right output and needs to minimize the squared error of its prediction, $\sum_i (r_i^t - y_{ih}^t)^2$.

Using gradient ascent to maximize the log likelihood, we get

$$\Delta w_{ih} = \eta \sum_t (r_i^t - y_{ih}^t) f_h^t \tag{12.33}$$

where

$$f_h^t = \frac{g_h^t \exp[-\frac{1}{2}\sum_i (r_i^t - y_{ih}^t)^2]}{\sum_l g_l^t \exp[-\frac{1}{2}\sum_i (r_i^t - y_{il}^t)^2]} \tag{12.34}$$

$$p(h|\boldsymbol{r}, \boldsymbol{x}) = \frac{p(h|\boldsymbol{x})p(\boldsymbol{r}|h, \boldsymbol{x})}{\sum_l p(l|\boldsymbol{x})p(\boldsymbol{r}|l, \boldsymbol{x})} \tag{12.35}$$

$g_h^t \equiv p(h|\boldsymbol{x}^t)$ is the posterior probability of unit h given the input, and it depends on the centers and spreads of all of the units. $f_h^t \equiv p(h|\boldsymbol{r}, \boldsymbol{x}^t)$ is the posterior probability of unit h given the input and the desired output, also taking the error into account in choosing the responsible unit.

Similarly, we can derive a rule to update the centers:

$$\Delta m_{hj} = \eta \sum_t (f_h^t - g_h^t) \frac{(x_j^t - m_{hj})}{s_h^2} \tag{12.36}$$

f_h is the posterior probability of unit h also taking the required output into account, whereas g_h is the posterior probability using only the input space information. Their difference is the error term for the centers. Δs_h can be similarly derived. In the cooperative case, there is no force on the units to be localized. To decrease the error, means and spreads can take any value; it is even possible sometimes for the spreads to increase and flatten out. In the competitive case, however, to increase the likelihood, units are forced to be localized with more separation between them and smaller spreads.

In classification, each component by itself is a multinomial. Then the log likelihood is

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \prod_i (y_{ih}^t)^{r_i^t} \tag{12.37}$$

$$= \sum_t \log \sum_h g_h^t \exp\left[\sum_i r_i^t \log y_{ih}^t\right] \tag{12.38}$$

where

$$y_{ih}^t = \frac{\exp w_{ih}}{\sum_k \exp w_{kh}} \tag{12.39}$$

Update rules for w_{ih}, $\boldsymbol{m}_h$, and s_h can be derived using gradient ascent, which will include

$$f_h^t = \frac{g_h^t \exp[\sum_i r_i^t \log y_{ih}^t]}{\sum_l g_l^t \exp[\sum_i r_i^t \log y_{il}^t]} \tag{12.40}$$

In chapter 7, we discussed the EM algorithm for fitting Gaussian mixtures to data. It is possible to generalize EM for supervised learning as well. Actually, calculating f_h^t corresponds to the E-step. $f_h^t \equiv p(\boldsymbol{r}|h, \boldsymbol{x}^t)$ replaces $p(h|\boldsymbol{x}^t)$, which we used in the E-step in chapter 7 when the application was unsupervised. In the M-step for regression, we update the parameters as

$$\boldsymbol{m}_h = \frac{\sum_t f_h^t \boldsymbol{x}^t}{\sum_t f_h^t} \tag{12.41}$$

$$\mathbf{S}_h = \frac{\sum_t f_h^t (\boldsymbol{x}^t - \boldsymbol{m}_h)(\boldsymbol{x}^t - \boldsymbol{m}_h)^T}{\sum_t f_h^t} \tag{12.42}$$

$$w_{ih} = \frac{\sum_t f_h^t r_i^t}{\sum_t f_h^t} \tag{12.43}$$

We see that w_{ih} is a weighted average where weights are the posterior probabilities of units, given the input and the desired output. In the case

of classification, the M-step has no analytical solution and we need to resort to an iterative procedure, for example, gradient ascent (Jordan and Jacobs 1994).

12.7 Learning Vector Quantization

Let us say we have H units for each class, already labeled by those classes. These units are initialized with random instances from their classes. At each iteration, we find the unit, $\mathbf{m}_i$, that is closest to the input instance in Euclidean distance and use the following update rule:

$$\begin{cases} \Delta \mathbf{m}_i = \eta(\mathbf{x}^t - \mathbf{m}_i) & \text{if } \mathbf{x}^t \text{ and } \mathbf{m}_i \text{ have the same class label} \\ \Delta \mathbf{m}_i = -\eta(\mathbf{x}^t - \mathbf{m}_i) & \text{otherwise} \end{cases} \tag{12.44}$$

If the closest center has the correct label, it is moved toward the input to better represent it. If it belongs to the wrong class, it is moved away from the input in the expectation that if it is moved sufficiently away, a center of the correct class will be the closest in a future iteration. This is the *learning vector quantization* (LVQ) model proposed by Kohonen (1990, 1995).

LEARNING VECTOR QUANTIZATION

The LVQ update equation is analogous to equation 12.36 where the direction in which the center is moved depends on the difference between two values: Our prediction of the winner unit based on the input distances and what the winner should be based on the required output.

12.8 The Mixture of Experts

In RBFs, corresponding to each local patch we give a constant fit. In the case where for any input, we have one g_h 1 and all others 0, we get a piecewise constant approximation where for output i, the local fit by patch h is given by w_{ih}. From the Taylor expansion, we know that at each point, the function can be written as

$$f(x) = f(a) + (x - a)f'(a) + \cdots \tag{12.45}$$

Thus a constant approximation is good if x is close enough to a and $f'(a)$ is close to 0—that is, if $f(x)$ is flat around a. If this is not the case, we need to divide the space into a large number of patches, which is particularly serious when the input dimensionality is high, due to the curse of dimensionality.

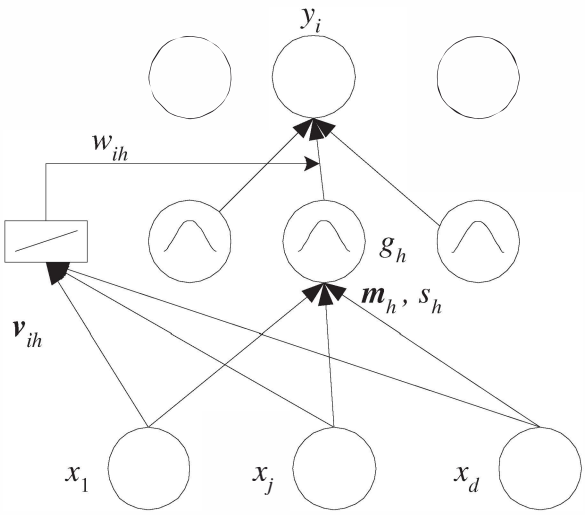


Figure 12.10 The mixture of experts can be seen as an RBF network where the second-layer weights are outputs of linear models. Only one linear model is shown for clarity.

PIECEWISE LINEAR APPROXIMATION

MIXTURE OF EXPERTS

An alternative is to have a *piecewise linear approximation* by taking into account the next term in the Taylor expansion, namely, the linear term. This is what is done by the *mixture of experts* (Jacobs et al. 1991). We write

$$y_i^t = \sum_{h=1}^{H} w_{ih} g_h^t \tag{12.46}$$

which is the same as equation 12.25 but here, w_{ih}, the contribution of patch h to output i is not a constant but a linear function of the input:

$$w_{ih}^t = \boldsymbol{v}_{ih}^T \boldsymbol{x}^t \tag{12.47}$$

$\boldsymbol{v}_{ih}$ is the parameter vector that defines the linear function and includes a bias term, making the mixture of experts a generalization of the RBF network. The unit activations can be taken as normalized RBFs:

$$g_h^t = \frac{\exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_h\|^2/2s_h^2]}{\sum_l \exp[-\|\boldsymbol{x}^t - \boldsymbol{m}_l\|^2/2s_l^2]} \tag{12.48}$$

This can be seen as an RBF network except that the second-layer weights are not constants but are outputs of linear models (see figure 12.10). Ja-

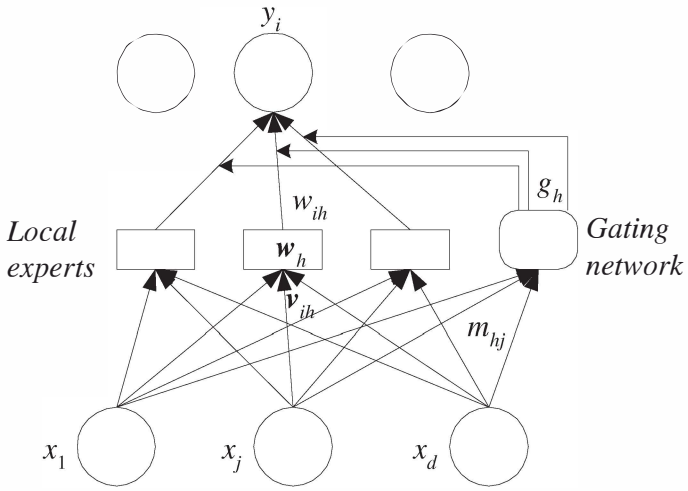


Figure 12.11 The mixture of experts can be seen as a model for combining multiple models. $\mathbf{w}_h$ are the models and the gating network is another model determining the weight of each model, as given by g_h. Viewed in this way, neither the experts nor the gating are restricted to be linear.

cobs et al. (1991) view this in another way: They consider $\mathbf{w}_h$ as linear models, each taking the input, and call them *experts*. g_h are considered to be the outputs of a *gating network*. The gating network works as a classifier does with its outputs summing to 1, assigning the input to one of the experts (see figure 12.11).

Considering the gating network in this manner, any classifier can be used in gating. When $\mathbf{x}$ is high-dimensional, using local Gaussian units may require a large number of experts and Jacobs et al. (1991) propose to take

$$g_h^t = \frac{\exp[\mathbf{m}_h^T \mathbf{x}^t]}{\sum_l \exp[\mathbf{m}_l^T \mathbf{x}^t]} \tag{12.49}$$

which is a linear classifier. Note that $\mathbf{m}_h$ are no longer centers but hyperplanes, and as such include bias values. This gating network is implementing a classification where it is dividing linearly the input region for which expert h is responsible from the expertise regions of other experts. As we will see again in chapter 17, the mixture of experts is a general architecture for combining multiple models; the experts and the gating

may be nonlinear, for example, contain multilayer perceptrons, instead of linear perceptrons (exercise 6).

An architecture similar to the mixture of experts and running line smoother (section 8.8.3) has been proposed by Bottou and Vapnik (1992). In their approach, no training is done initially. When a test instance is given, a subset of the data close to the test instance is chosen from the training set (as in the k-nearest neighbor, but with a large k), a simple model, for example, a linear classifier, is trained with this local data, the prediction is made for the instance, and then the model is discarded. For the next instance, a new model is created, and so on. On a handwritten digit recognition application, this model has less error than the multilayer perceptron, k-nearest neighbor, and Parzen windows; the disadvantage is the need to train a new model on the fly for each test instance.

12.8.1 Cooperative Experts

In the cooperative case, y_i^t is given by equation 12.46, and we would like to make it as close as possible to the required output, r_i^t. In regression, the error function is

$$E(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \frac{1}{2}\sum_t \sum_i (r_i^t - y_i^t)^2 \tag{12.50}$$

Using gradient descent, second-layer (expert) weight parameters are updated as

$$\Delta \boldsymbol{v}_{ih} = \eta \sum_t (r_i^t - y_i^t) g_h^t \boldsymbol{x}^t \tag{12.51}$$

Compared with equation 12.26, we see that the only difference is that this new update is a function of the input.

If we use softmax gating (equation 12.49), using gradient descent we have the following update rule for the hyperplanes:

$$\Delta m_{hj} = \eta \sum_t \sum_i (r_i^t - y_i^t)(w_{ih}^t - y_i^t) g_h^t x_j^t \tag{12.52}$$

If we use radial gating (equation 12.48), only the last term, $\partial p_h / \partial m_{hj}$, differs.

In classification, we have

$$y_i = \frac{\exp\left[\sum_h w_{ih} g_h^t\right]}{\sum_k \exp\left[\sum_h w_{kh} g_h^t\right]} \tag{12.53}$$

with $w_{ih} = \boldsymbol{v}_{ih}^T \boldsymbol{x}$, and update rules can be derived to minimize the cross-entropy using gradient descent (exercise 7).

12.8.2 Competitive Experts

Just like the competitive RBFs, we have

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \exp\left[-\frac{1}{2}\sum_i (r_i^t - y_{ih}^t)^2\right] \tag{12.54}$$

where $y_{ih}^t = w_{ih}^t = \boldsymbol{v}_{ih}\boldsymbol{x}^t$. Using gradient ascent, we get

$$\Delta \boldsymbol{v}_{ih} = \eta \sum_t (r_i^t - y_{ih}^t) f_h^t \boldsymbol{x}^t \tag{12.55}$$

$$\Delta \boldsymbol{m}_h = \eta \sum_t (f_h^t - g_h^t) \boldsymbol{x}^t \tag{12.56}$$

assuming softmax gating as given in equation 12.49.

In classification, we have

$$\mathcal{L}(\{\boldsymbol{m}_h, s_h, w_{ih}\}_{i,h}|\mathcal{X}) = \sum_t \log \sum_h g_h^t \prod_i (y_{ih}^t)^{r_i^t} \tag{12.57}$$

$$= \sum_t \log \sum_h g_h^t \exp\left[\sum_i r_i^t \log y_{ih}^t\right] \tag{12.58}$$

where

$$y_{ih}^t = \frac{\exp w_{ih}^t}{\sum_k \exp w_{kh}^t} = \frac{\exp[\boldsymbol{v}_{ih}\boldsymbol{x}^t]}{\sum_k \exp[\boldsymbol{v}_{kh}\boldsymbol{x}^t]} \tag{12.59}$$

Jordan and Jacobs (1994) generalize EM for the competitive case with local linear models. Alpaydın and Jordan (1996) compare cooperative and competitive models for classification tasks and see that the cooperative model is generally more accurate but the competitive version learns faster. This is because in the cooperative case, models overlap more and implement a smoother approximation, and thus it is preferable in regression problems. The competitive model makes a harder split; generally only one expert is active for an input and therefore learning is faster.

12.9 Hierarchical Mixture of Experts

HIERARCHICAL MIXTURE OF EXPERTS

In figure 12.11, we see a set of experts and a gating network that chooses one of the experts as a function of the input. In a *hierarchical mixture*

of experts, we replace each expert with a complete system of mixture of experts in a recursive manner (Jordan and Jacobs 1994). Once an architecture is chosen—namely, the depth, the experts, and the gating models—the whole tree can be learned from a labeled sample. Jordan and Jacobs (1994) derive both gradient descent and EM learning rules for such an architecture (see exercise 9).

We may also interpret this architecture as a decision tree (chapter 9) and its gating networks as decision nodes. In the decision trees we discussed earlier, a decision node makes a hard decision and takes one of the branches, so we take only one path from the root to one of the leaves. What we have here is a *soft decision tree* where, because the gating model returns us a probability, we take all the branches but with different probabilities; so we traverse all the paths to all the leaves and we take a weighted sum over all the leaf values where weights are equal to the product of the gating values on the path to each leaf. The advantage of this averaging is that the boundaries between leaf regions are no longer hard but there is a transition from one to the other and this smooths the response (İrsoy, Yıldız, and Alpaydın 2012).

12.10 Notes

An RBF network can be seen as a neural network, implemented by a network of simple processing units. It differs from a multilayer perceptron in that the first and second layers implement different functions. Omohundro (1987) discusses how local models can be implemented as neural networks and also addresses hierarchical data structures for fast localization of relevant local units. Specht (1991) shows how Parzen windows can be implemented as a neural network.

Platt (1991) proposed an incremental version of RBF where new units are added as necessary. Fritzke (1995) similarly proposed a growing version of SOM.

Lee (1991) compares k-nearest neighbor, multilayer perceptron, and RBF network on a handwritten digit recognition application and concludes that these three methods all have small error rates. RBF networks learn faster than backpropagation on a multilayer perceptron but use more parameters. Both of these methods are superior to the k-NN in terms of classification speed and memory need. Such practical con-

straints like time, memory, and computational complexity may be more important than small differences in error rate in real-world applications.

Kohonen's SOM (1990, 1995) was one of the most popular neural network methods, having been used in a variety of applications including exploratory data analysis and as a preprocessing stage before a supervised learner. One interesting and successful application is the traveling salesman problem (Angeniol, Vaubois, and Le Texier 1988). Just like the difference between k-means clustering and EM on Gaussian mixtures (chapter 7), *generative topographic mapping* (GTM) (Bishop, Svensén, and Williams 1998) is a probabilistic version of SOM that optimizes the log likelihood of the data using a mixture of Gaussians whose means are constrained to lie on a two-dimensional manifold (for topological ordering in low dimensions).

GENERATIVE TOPOGRAPHIC MAPPING

In an RBF network, once the centers and spreads are fixed (e.g., by choosing a random subset of training instances as centers, as in the anchor method), training the second layer is a linear model. This model is equivalent to support vector machines with Gaussian kernels where during learning the best subset of instances, named the *support vectors*, are chosen; we discuss them in chapter 13. Gaussian processes (chapter 16) where we interpolate from stored training instances are also similar.

12.11 Exercises

1. Show an RBF network that implements XOR.

   SOLUTION: There are two possibilities (see figure 12.12): (a) We can have two circular Gaussians centered on the two positive instances and the second layer ORs them, or (b) we can have one elliptic Gaussian centered on (0.5, 0.5) with negative correlation to cover the two positive instances.

2. Write down the RBF network that uses elliptic units instead of radial units as in equation 12.13.

   SOLUTION:

   $$p_h^t = \exp\left[-\frac{1}{2}(\mathbf{x}^t - \mathbf{m}_h)^T \mathbf{S}_h^{-1}(\mathbf{x}^t - \mathbf{m}_h)\right]$$

   where $\mathbf{S}_h$ is the local covariance matrix.

3. Derive the update equations for the RBF network for classification (equations 12.20 and 12.21).

4. Show how the system given in equation 12.22 can be trained.

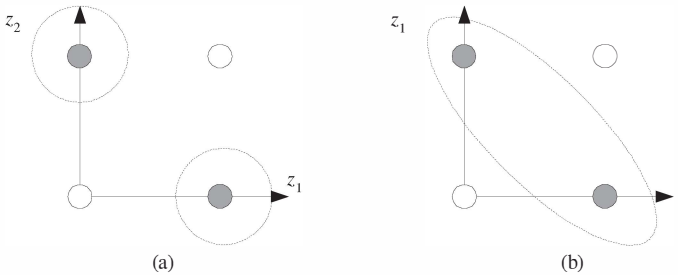


Figure 12.12 Two ways of implementing XOR with RBF.

5. Compare the number of parameters of a mixture of experts architecture with an RBF network.

   SOLUTION: With d inputs, K classes and H Gaussians, an RBF network needs $H \cdot d$ parameters for the centers, H parameters for the spreads and $(H+1)K$ parameters for the second-layer weights. For the case of the MoE, for each second-layer weight, we need a $d+1$ dimensional vector of the linear model, but there is no bias; hence we have $HK(d+1)$ parameters.

   Note that the number of parameters in the first layer is the same with RBF and it is the same whether we have Gaussian or softmax gating: For each hidden unit, in the case of Gaussian gating, we need d parameters for the center and 1 for the spread; in the case of softmax gating, the linear model has $d+1$ parameters (d inputs and a bias).

6. Formalize a mixture of experts architecture where the experts and the gating network are multilayer perceptrons. Derive the update equations for regression and classification.

7. Derive the update equations for the cooperative mixture of experts for classification.

8. Derive the update equations for the competitive mixture of experts for classification.

9. Formalize the hierarchical mixture of experts architecture with two levels. Derive the update equations using gradient descent for regression and classification.

   SOLUTION: The following is from Jordan and Jacobs 1994; notation is slightly changed to match the notation of the book.

   Let us see the case of regression with a single output: y is the overall output, y_i are the outputs on the first level, and y_{ij} are the outputs on the second level, which are the leaves in a model with two levels. Similarly, g_i are the

gating outputs on the first level and $g_{j|i}$ are the outputs on the second level, that is, the gating value of expert j on the second level given that we have chosen the branch i on the first level:

$$\begin{aligned} y &= \sum_i g_i y_i \\ y_i &= \sum_j g_{j|i} y_{ij} \quad \text{and} \quad g_i = \frac{\exp \boldsymbol{m}_i^T \boldsymbol{x}}{\sum_k \exp \boldsymbol{m}_k^T \boldsymbol{x}} \\ y_{ij} &= \boldsymbol{v}_{ij}^T \boldsymbol{x} \quad \text{and} \quad g_{j|i} = \frac{\exp \boldsymbol{m}_{ij}^T \boldsymbol{x}}{\sum_l \exp \boldsymbol{m}_{il}^T \boldsymbol{x}} \end{aligned}$$

In regression, the error to be minimized is as follows (note that we are using a competitive version here):

$$E = \sum_t \log \sum_i g_i^t \sum_j g_{j|i}^t \exp\left[-\frac{1}{2}(r^t - y_{ij}^t)^2\right]$$

and using gradient descent, we get the following update equations:

$$\begin{aligned} \Delta \boldsymbol{v}_{ij} &= \eta \sum_t f_i^t f_{j|i}^t (r^t - y^t) \boldsymbol{x}^t \\ \Delta \boldsymbol{m}_i &= \eta \sum_t (f_i^t - g_i^t) \boldsymbol{x}^t \\ \Delta \boldsymbol{m}_{ij} &= \eta \sum_t f_i^t (f_{j|i}^t - g_{j|i}^t) \boldsymbol{x}^t \end{aligned}$$

where we make use of the following posteriors:

$$\begin{aligned} f_i^t &= \frac{g_i^t \sum_j g_{j|i}^t \exp[-(1/2)(r^t - y_{ij}^t)^2]}{\sum_k g_k^t \sum_j g_{j|k}^t \exp[-(1/2)(r^t - y_{kj}^t)^2]} \\ f_{j|i}^t &= \frac{g_{j|i}^t \exp[-(1/2)(r^t - y_{ij}^t)^2]}{\sum_l g_{l|i}^t \exp[-(1/2)(r^t - y_{il}^t)^2]} \\ f_{ij}^t &= \frac{g_i^t g_{j|i}^t \exp[-(1/2)(r^t - y_{ij}^t)^2]}{\sum_k g_k^t \sum_l g_{l|k}^t \exp[-(1/2)(r^t - y_{kl}^t)^2]} \end{aligned}$$

Note how we multiply the gating values on the path starting from the root to a leaf expert.

For the case of classification with $K > 2$ classes, one possibility is to have K separate HMEs as above (having single output experts), whose outputs we softmax to maximize the log likelihood:

$$\begin{aligned} \mathcal{L} &= \sum_t \log \sum_i g_i^t \sum_j g_{j|i}^t \exp\left[\sum_c r_c^t \log p_c^t\right] \\ p_c^t &= \frac{\exp y_c^t}{\sum_k \exp y_k^t} \end{aligned}$$

where each y_c^t denotes the output of one single-output HME. The more interesting case of a single multiclass HME where experts have K softmax outputs is discussed in Waterhouse and Robinson 1994.

10. In the mixture of experts, because different experts specialize in different parts of the input space, they may need to focus on different inputs. Discuss how dimensionality can be locally reduced in the experts.

12.12 References

Alpaydın, E., and M. I. Jordan. 1996. "Local Linear Perceptrons for Classification." *IEEE Transactions on Neural Networks* 7:788–792.

Angeniol, B., G. Vaubois, and Y. Le Texier. 1988. "Self Organizing Feature Maps and the Travelling Salesman Problem." *Neural Networks* 1:289–293.

Berthold, M. 1999. "Fuzzy Logic." In *Intelligent Data Analysis: An Introduction*, ed. M. Berthold and D. J. Hand, 269–298. Berlin: Springer.

Bishop, C. M., M. Svensén, and C. K. I. Williams. 1998. "GTM: The Generative Topographic Mapping." *Neural Computation* 10:215–234.

Bottou, L., and V. Vapnik. 1992. "Local Learning Algorithms." *Neural Computation* 4:888–900.

Broomhead, D. S., and D. Lowe. 1988. "Multivariable Functional Interpolation and Adaptive Networks." *Complex Systems* 2:321–355.

Carpenter, G. A., and S. Grossberg. 1988. "The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network." *IEEE Computer* 21 (3): 77–88.

Cherkassky, V., and F. Mulier. 1998. *Learning from Data: Concepts, Theory, and Methods*. New York: Wiley.

DeSieno, D. 1988. "Adding a Conscience Mechanism to Competitive Learning." In *IEEE International Conference on Neural Networks*, 117–124. Piscataway, NJ: IEEE Press.

Feldman, J. A., and D. H. Ballard. 1982. "Connectionist Models and their Properties." *Cognitive Science* 6:205–254.

Fritzke, B. 1995. "Growing Cell Structures: A Self Organizing Network for Unsupervised and Supervised Training." *Neural Networks* 7:1441–1460.

Grossberg, S. 1980. "How Does the Brain Build a Cognitive Code?" *Psychological Review* 87:1–51.

Hertz, J., A. Krogh, and R. G. Palmer. 1991. *Introduction to the Theory of Neural Computation*. Reading, MA: Addison-Wesley.

İrsoy, O., O. T. Yıldız, and E. Alpaydın. 2012. "Soft Decision Trees." In *International Conference on Pattern Recognition*, 1819–1822. Piscataway, NJ: IEEE Press.

Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton. 1991. "Adaptive Mixtures of Local Experts." *Neural Computation* 3:79–87.

Jordan, M. I., and R. A. Jacobs. 1994. "Hierarchical Mixtures of Experts and the EM Algorithm." *Neural Computation* 6:181–214.

Kohonen, T. 1990. "The Self-Organizing Map." *Proceedings of the IEEE* 78:1464–1480.

Kohonen, T. 1995. *Self-Organizing Maps*. Berlin: Springer.

Lee, Y. 1991. "Handwritten Digit Recognition Using k-Nearest Neighbor, Radial Basis Function, and Backpropagation Neural Networks." *Neural Computation* 3:440–449.

Mao, J., and A. K. Jain. 1995. "Artificial Neural Networks for Feature Extraction and Multivariate Data Projection." *IEEE Transactions on Neural Networks* 6:296–317.

Moody, J., and C. Darken. 1989. "Fast Learning in Networks of Locally-Tuned Processing Units." *Neural Computation* 1:281–294.

Oja, E. 1982. "A Simplified Neuron Model as a Principal Component Analyzer." *Journal of Mathematical Biology* 15:267–273.

Omohundro, S. M. 1987. "Efficient Algorithms with Neural Network Behavior." *Complex Systems* 1:273–347.

Platt, J. 1991. "A Resource Allocating Network for Function Interpolation." *Neural Computation* 3:213–225.

Specht, D. F. 1991. "A General Regression Neural Network." *IEEE Transactions on Neural Networks* 2:568–576.

Tresp, V., J. Hollatz, and S. Ahmad. 1997. "Representing Probabilistic Rules with Networks of Gaussian Basis Functions." *Machine Learning* 27:173–200.

Waterhouse, S. R., and A. J. Robinson. 1994. "Classification Using Hierarchical Mixtures of Experts." In *IEEE Workshop on Neural Networks for Signal Processing*, 177–186. Piscataway, NJ: IEEE Press.