Chapter 5

Radial Basis Function Networks

"Two birds disputed about a kernel, when a third swooped down and carried it off."—African Proverb

5.1 Introduction

Radial basis function (RBF) networks represent a fundamentally different architecture from what we have seen in the previous chapters. All the previous chapters use a feed-forward network in which the inputs are transmitted forward from layer to layer in a similar fashion in order to create the final outputs. A feed-forward network might have many layers, and the nonlinearity is typically created by the repeated composition of activation functions. On the other hand, an RBF network typically uses only an input layer, a single hidden layer (with a special type of behavior defined by RBF functions), and an output layer. Although it is possible to replace the output layer with multiple feed-forward layers (like a conventional network), the resulting network is still quite shallow, and its behavior is strongly influenced by the nature of the special hidden layer. For simplicity in discussion, we will work with only a single output layer. As in feed-forward networks, the input layer is not really a computational layer, and it only carries the inputs forward. The nature of the computations in the hidden layer are very different from what we have seen so far in feed-forward networks. In particular, the hidden layer performs a computation based on a comparison with a *prototype vector*, which has no exact counterpart in feed-forward networks. The structure and the computations performed by the special hidden layer is the key to the power of the RBF network.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_5

One can characterize the difference in the functionality of the hidden and output layers as follows:

1. The hidden layer takes the input points, in which the class structure might not be linearly separable, and transforms them into a new space that is (often) linearly separable. The hidden layer often has higher dimensionality than the input layer, because transformation to a higher-dimensional space is often required in order to ensure linear separability. This principle is based on *Cover's theorem on separability of patterns* [84], which states that pattern classification problems are more likely to be linearly separable when cast into a high-dimensional space with a nonlinear transformation. Furthermore, certain types of transformations in which features represent small localities in the space are more likely to lead to linear separability. Although the dimensionality of the hidden layer is typically greater than the input dimensionality, it is always less than or equal to the number of training points. An extreme case in which the dimensionality of the hidden layer is equal to the number of training points can be shown to be roughly equivalent to *kernel learners*. Examples of such models include kernel regression and kernel support vector machines.

2. The output layer uses linear classification or regression modeling with respect to the inputs from the hidden layer. The connections from the hidden to the output layer have weights attached to them. The computations in the output layer are performed in the same way as in a standard feed-forward network. Although it is also possible to replace the output layer with multiple feed-forward layers, we will consider only the case of a single feed-forward layer for simplicity.

Just as the perceptron is a variant of the linear support vector machine, the RBF network is a generalization of kernel classification and regression. Special cases of the RBF network can be used to implement kernel regression, least-squares kernel classification, and the kernel support-vector machine. The differences among these special cases is in terms of how the output layer and the loss function is structured. In feed-forward networks, increasing nonlinearity is obtained by increasing depth. However, in an RBF network, a single hidden layer is usually sufficient to achieve the required level of nonlinearity because of its special structure. Like feed-forward networks, RBF networks are universal function approximators.

The layers of the RBF network are designed as follows:

1. The input layer simply transmits from the input features to the hidden layers. Therefore, the number of input units is exactly equal to the dimensionality d of the data. As in the case of feed-forward networks, no computation is performed in the input layers. As in all feed-forward networks, the input units are fully connected to the hidden units and carry their input forward.

2. The computations in the hidden layers are based on comparisons with *prototype vectors*. Each hidden unit contains a d-dimensional prototype vector. Let the prototype vector of the ith hidden unit be denoted by $\overline{\mu}_i$. In addition, the ith hidden unit contains a bandwidth denoted by σ_i. Although the prototype vectors are always specific to particular units, the bandwidths of different units σ_i are often set to the same value σ. The prototype vectors and bandwidth(s) are usually learned either in an unsupervised way, or with the use of mild supervision.

Then, for any input training point $\overline{X}$, the activation $\Phi_i(\overline{X})$ of the ith hidden unit is defined as follows:

$$h_i = \Phi_i(\overline{X}) = \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right) \quad \forall i \in \{1, \ldots, m\} \tag{5.1}$$

The total number of hidden units is denoted by m. Each of these m units is designed to have a high level of influence on the particular cluster of points that is closest to its prototype vector. Therefore, one can view m as the number of clusters used for modeling, and it represents an important hyper-parameter available to the algorithm. For low-dimensional inputs, it is typical for the value of m to be larger than the input dimensionality d, but smaller than the number of training points n.

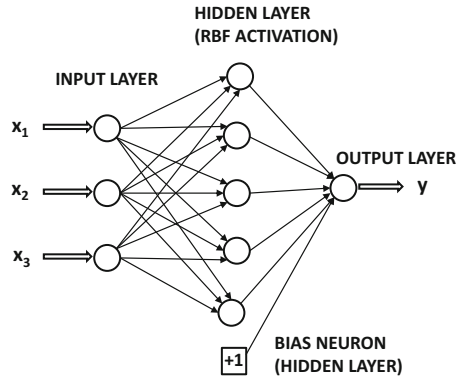

Figure 5.1: An RBF network: Note that the hidden layer is broader than the input layer, which is typical (but not mandatory).

3. For any particular training point $\overline{X}$, let h_i be the output of the ith hidden unit, as defined by Equation 5.1. The weights of the connections from the hidden to the output nodes are set to w_i. Then, the prediction $\hat{y}$ of the RBF network in the output layer is defined as follows:

$$\hat{y} = \sum_{i=1}^{m} w_i h_i = \sum_{i=1}^{m} w_i \Phi_i(\overline{X}) = \sum_{i=1}^{m} w_i \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right)$$

The variable $\hat{y}$ has a circumflex on top to indicate the fact that it is a predicted value rather than observed value. If the observed target is real-valued, then one can set up a least-squares loss function, which is much like that in a feed-forward network. The values of the weights $w_1 \ldots w_m$ need to be learned in a supervised way.

An additional detail is that the hidden layer of the neural network contains bias neurons. Note that the bias neuron can be implemented by a single hidden unit in the output layer, which is always on. One can also implement this type of neuron by creating a hidden unit in which the value of σ_i is ∞. In either case, it will be assumed throughout the discussions in this chapter that this special hidden unit is absorbed among the m hidden units. Therefore, it is not treated in any special way. An example of an RBF network is illustrated in Figure 5.1.

In the RBF network, there are two sets of computations corresponding to the hidden layer and the output layer. The parameters $\overline{\mu}_i$ and σ_i of the hidden layer are learned in

an unsupervised way, whereas those of the output layer are learned in a supervised way with gradient descent. The latter is similar to the case of the feed-forward network. The prototypes $\overline{\mu}_i$ may either be sampled from the data, or be set to be the m centroids of an m-way clustering algorithm. The latter solution is used frequently. The different ways of training the neural network are discussed in Section 5.2.

RBF networks can be shown to be direct generalizations of the class of kernel methods. This is primarily because the prediction in the output node can be shown to be equivalent to a weighted nearest neighbor estimator, where the weights are products of the coefficients w_i and Gaussian RBF similarities to *prototypes*. The prediction function in almost all kernel methods can also be shown to be equivalent to a weighted nearest neighbor estimator, where the weights are learned in a supervised way. Therefore, kernel methods represent a special case of RBF methods in which the number of hidden nodes is equal to the number of training points, the prototypes are set to the training points, and each σ_i has the same value. This suggests that RBF networks have greater power and flexibility than do kernel methods; this relationship will be discussed in detail in Section 5.4.

When to Use RBF Networks

A key point is that the hidden layer of the RBF network is created in an unsupervised way, tending to make it robust to all types of noise (including adversarial noise). Indeed, this property of RBF networks is shared by support vector machines. At the same time, there are limitations with respect to how much structure in the data an RBF network can learn. Deep feed-forward networks are effective at learning from data with a rich structure because the multiple layers of nonlinear activations force the data to follow specific types of patterns. Furthermore, by adjusting the structure of connections, one can incorporate domain-specific insights in feed-forward networks. Examples of such settings include recurrent and convolutional neural networks. The single layer of an RBF network limits the amount of structure that one can learn. Although both RBF networks and deep feed-forward networks are known to be universal function approximators, there are differences in terms of their generalization performance on different types of data sets.

Chapter Organization

This chapter is organized as follows. The next section discusses the various training methods for RBF networks. The use of RBF networks in classification and interpolation is discussed in Section 5.3. The relationship of the RBF method to kernel regression and classification is discussed in Section 5.4. A summary is provided in Section 5.5.

5.2 Training an RBF Network

The training of an RBF network is very different from that of a feed-forward network, which is fully integrated across different layers. In an RBF network, the training of the hidden layer is typically done in an unsupervised manner. While it is possible, in principle, to train the prototype vectors and the bandwidths using backpropagation, the problem is that there are more local minima on the loss surface of RBF networks compared to feed-forward networks. Therefore, the supervision in the hidden layer (when used) is often relatively gentle, or it is restricted only to fine-tuning weights that have already been learned. Nevertheless, since overfitting seems to be a pervasive problem with the supervised training of the hidden

layer, our discussion will be restricted to unsupervised methods. In the following, we will first discuss the training of the hidden layer of an RBF network, and then discuss the training of the output layer.

5.2.1 Training the Hidden Layer

The hidden layer of the RBF network contains several parameters, including the prototype vectors $\overline{\mu}_1 \ldots \overline{\mu}_m$, and the bandwidths $\sigma_1 \ldots \sigma_m$. The hyperparameter m controls the number of hidden units. In practice, a separate value of σ_i is not set for each unit, and all units have the same bandwidth σ. However, the mean values $\overline{\mu}_i$ for the various hidden units are different because they define the all-important prototype vectors. The complexity of the model is regulated by the number of hidden units and the bandwidth. The combination of a small bandwidth and a large number of hidden units increases the model complexity, and is a useful setting when the amount of data is large. Smaller data sets require fewer units and larger bandwidths to avoid overfitting. The value of m is typically larger than the input data dimensionality, but it is never larger than the number of training points. Setting the value of m equal to the number of training points, and using each training point as a prototype in a hidden node, makes the approach equivalent to traditional kernel methods.

The bandwidth also depends on the chosen prototype vectors $\overline{\mu}_1 \ldots \overline{\mu}_m$. Ideally, the bandwidths should be set in a way that each point should be (significantly) influenced by only a small number of prototype vectors, which correspond to its closest clusters. Setting the bandwidth too large or too small compared to the inter-prototype distance will lead to under-fitting and over-fitting, respectively. Let d_{max} be maximum distance between pairs of prototype centers, and d_{ave} be the average distance between them. Then, two heuristic ways of setting the bandwidth are as follows:

$$\sigma = \frac{d_{max}}{\sqrt{m}}$$

$$\sigma = 2 \cdot d_{ave}$$

One problem with this choice of σ is that the optimal value of the bandwidth might vary in different parts of the input space. For example, the bandwidth in a dense region in the data space should be smaller than the bandwidth in a sparse region of the space. The bandwidth should also depend on how the prototype vectors are distributed in the space. Therefore, one possible solution is to choose the bandwidth σ_i of the ith prototype vector to be equal to its distance to its rth nearest neighbor among the prototypes. Here, r is a small value like 5 or 10.

However, these are only heuristic rules. It is possible to fine-tune these values by using a held-out portion of data set. In other words, candidate values of σ are generated in the neighborhood of the above recommended values of σ (as an initial reference point). Then, multiple models are constructed using these candidate values of σ (including the training of the output layer). The choice of σ that provides the least error on the held-out portion of the training data set is used. This type of approach does use a certain level of supervision in the selection of the bandwidth, without getting stuck in local minima. However, the nature of the supervision is quite gentle, which is particularly important when dealing with the parameters of the first layer in an RBF network. It is noteworthy that this type of tuning of the bandwidth is also performed when using the Gaussian kernel with a kernel support-vector machine. This similarity is not a coincidence because the kernel support-vector machine is a special case of RBF networks (cf. Section 5.4).

The selection of the prototype vectors is somewhat more complex. In particular, the following choices are often made:

1. The prototype vectors can be randomly sampled from the n training points. A total of $m < n$ training points are sampled in order to create the prototype vectors. The main problem with this approach is that it will over-represent prototypes from dense regions of the data, whereas sparse regions might get few or no prototypes. As a result, the prediction accuracy in such regions will suffer.

2. A k-means clustering algorithm can be used in order to create m clusters. The centroid of each of these m clusters can be used as a prototype. The use of the k-means algorithm is the most common choice for learning the prototype vectors.

3. Variants of clustering algorithms that partition the data *space* (rather than the points) are also used. A specific example is the use of decision trees to create the prototypes.

4. An alternative method for training the hidden layer is by using the *orthogonal least-squares algorithm*. This approach uses a certain level of supervision. In this approach, the prototype vectors are selected one by one from the training data in order to minimize the residual error of prediction on an out-of-sample test set. Since this approach requires understanding of the training of the output layer, its discussion will be deferred to a later section.

In the following, we briefly describe the k-means algorithm for creating the prototypes because it is the most common choice in real implementations. The k-means algorithm is a classical technique in the clustering literature. It uses the cluster prototypes as the prototypes for the hidden layer in the RBF method. Broadly speaking, the k-means algorithm proceeds as follows. At initialization, the m cluster prototypes are set to m random training points. Subsequently, each of the n data points is assigned to the prototype to which it has the smallest Euclidean distance. The assigned points of each prototype are averaged in order to create a new cluster center. In other words, the centroid of the created clusters is used to replace its old prototype with a new prototype. This process is repeated iteratively to convergence. Convergence is reached when the cluster assignments do not change significantly from one iteration to the next.

5.2.2 Training the Output Layer

The output layer is trained after the hidden layer has been trained. The training of the output layer is quite straightforward, because it uses only a single layer with linear activation. For ease in discussion, we will first consider the case in which the target of the output layer is real-valued. Later, we will discuss other settings. The output layer contains an m-dimensional vector of weights $\overline{W} = [w_1 \ldots w_m]$ that needs to be learned. Assume that the vector $\overline{W}$ is a row vector.

Consider a situation in which the training data set contains n points $\overline{X_1} \ldots \overline{X_n}$, which create the representations $\overline{H_1} \ldots \overline{H_n}$ in the hidden layer. Therefore, each $\overline{H_i}$ is an m-dimensional row vector. One could stack these n row vectors on top of one another to create an $n \times m$ matrix H. Furthermore, the observed targets of the n training points are denoted by $y_1, y_2, \ldots y_n$, which can be written as the n-dimensional column vector $\overline{y} = [y_1 \ldots y_n]^T$.

The predictions of the n training points are given by the elements of the n-dimensional column vector $H\overline{W}^T$. Ideally, we would like these predictions to be as close to the observed vector $\overline{y}$ as possible. Therefore, the loss function L for learning the output-layer weights is as follows:

$$L = \frac{1}{2}||H\overline{W}^T - \overline{y}||^2$$

In order to reduce overfitting, one can add Tikhonov regularization to the objective function:

$$L = \frac{1}{2}||H\overline{W}^T - \overline{y}||^2 + \frac{\lambda}{2}||\overline{W}||^2 \tag{5.2}$$

Here, $\lambda > 0$ is the regularization parameter. By computing the partial derivative of L with respect to the elements of the weight vector, we obtain the following:

$$\frac{\partial L}{\partial \overline{W}} = H^T(H\overline{W}^T - \overline{y}) + \lambda \overline{W}^T = 0$$

The above derivative is written in matrix calculus notation where $\frac{\partial L}{\partial \overline{W}}$ refers to the following:

$$\frac{\partial L}{\partial \overline{W}} = \left(\frac{\partial L}{\partial w_1} \cdots \frac{\partial L}{\partial w_d}\right)^T \tag{5.3}$$

By re-adjusting the above condition, we obtain the following:

$$(H^T H + \lambda I)\overline{W}^T = H^T \overline{y}$$

When $\lambda > 0$, the matrix $H^T H + \lambda I$ is positive-definite and is therefore invertible. In other words, one obtains a simple solution for the weight vector in closed form:

$$\overline{W}^T = (H^T H + \lambda I)^{-1} H^T \overline{y} \tag{5.4}$$

Therefore, a simple matrix inversion is sufficient to find the weight vector, and backpropagation is completely unnecessary.

However, the reality is that the use of a closed-form solution is not viable in practice because the size of the matrix $H^T H$ is $m \times m$, which can be large. For example, in kernel methods, we set $m = n$, in which the matrix is too large to even materialize, let alone invert. Therefore, one uses stochastic gradient descent to update the weight vector in practice. In such a case, the gradient-descent updates (with all training points) are as follows:

$$\begin{aligned}\overline{W}^T &\Leftarrow \overline{W}^T - \alpha \frac{\partial L}{\partial \overline{W}} \\ &= \overline{W}^T(1 - \alpha\lambda) - \alpha H^T \underbrace{(H\overline{W}^T - \overline{y})}_{\text{Current Errors}}\end{aligned}$$

One can also choose to use mini-batch gradient descent in which the matrix H in the above update can be replaced with a random subset of rows H_r from H, corresponding to the mini-batch. This approach is equivalent to what would normally be used in a traditional neural network with mini-batch stochastic gradient descent. However, it is applied only to the weights of the connections incident on the output layer in this case.

5.2.2.1 Expression with Pseudo-Inverse

In the case in which the regularization parameter λ is set to 0, the weight vector $\overline{W}$ is defined as follows:

$$\overline{W}^T = (H^T H)^{-1} H^T \overline{y} \tag{5.5}$$

The matrix $(H^T H)^{-1} H^T$ is said to be the *pseudo-inverse* of the matrix H. The pseudo-inverse of the matrix H is denoted by H^+. Therefore, one can write the weight vector $\overline{W}^T$ as follows:

$$\overline{W}^T = H^+ \overline{y} \tag{5.6}$$

The pseudo-inverse is a generalization of the notion of an inverse for non-singular or rectangular matrices. In this particular case, $H^T H$ is assumed to be invertible, although the pseudo-inverse of H can be computed even in cases where $H^T H$ is not invertible. In the case where H is square and invertible, the pseudo-inverse is the same as its inverse.

5.2.3 Orthogonal Least-Squares Algorithm

We revisit the training phase of the hidden layer. The training approach discussed in this section will use the predictions of the output layer in choosing the prototypes. Therefore, the training process of the hidden layer is supervised, although the supervision is restricted to iterative selections from the original training points. The orthogonal least-squares algorithm chooses the prototype vector one by one from the training points in order to minimize the error of prediction.

The algorithm starts by building an RBF network with a single hidden node and trying each possible training point as a prototype in order to compute the prediction error. One then selects the prototype from the training points that minimizes the error of prediction. In the next iteration, one more prototype is added to the selected prototype in order to build an RBF network with two prototypes. As in the previous iteration, all $(n-1)$ remaining training points are tried as possible prototypes in order to add to the current bag of prototypes, and the criterion for adding to the bag is the minimization of prediction error. In the $(r+1)$th iteration, one tries all the $(n-r)$ remaining training points, and adds one of them to the bag of prototypes so that the prediction error is minimized. Some of the training points in the data are held out, and are not used in the computations of the predictions or as candidates for prototypes. These out-of-sample points are used in order to test the effect of adding a prototype to the error. At some point, the error on this held-out set begins to rise as more prototypes as added. An increase in error on the held-out test set is a sign of the fact that further increase in prototypes will increase overfitting. This is the point at which one terminates the algorithm.

The main problem with this approach is that it is extremely inefficient. In each iteration, one must run n training procedures, which is computationally prohibitive for large training data sets. An interesting procedure in this respect is the *orthogonal least-squares algorithm* [65], which is known to be efficient. This algorithm is similar to the one described above in the sense that the prototype vectors are added iteratively from the original training

data set. However, the procedure with which the prototype is added is far more efficient. A set of orthogonal vectors are constructed in the space spanned by the hidden unit activations from the training data set. These orthogonal vectors can be used to directly compute which prototype should be selected from the training data set.

5.2.4 Fully Supervised Learning

The orthogonal least-squares algorithm represents a type of mild supervision in which the prototype vector is selected from one of the training points based on the effect to the overall prediction error. It is also possible to perform stronger types of supervision in which one can backpropagate in order to update the prototype vectors and the bandwidth. Consider the loss function L over the various training points:

$$L = \frac{1}{2}\sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)^2 \tag{5.7}$$

Here, $\overline{H_i}$ represents the m-dimensional vector of activations in the hidden layer for the ith training point $\overline{X_i}$.

The partial derivative with respect to each bandwidth σ_j can be computed as follows:

$$\begin{aligned}\frac{\partial L}{\partial \sigma_j} &= \sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i) w_j \frac{\partial \Phi_j(\overline{X_i})}{\partial \sigma_j} \\ &= \sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)\, w_j \Phi_j(\overline{X_i})\, \frac{||\overline{X_i} - \overline{\mu}_j||^2}{\sigma_j^3}\end{aligned}$$

If all bandwidths σ_j are fixed to the same value σ, as is common in RBF networks, then the derivative can be computed using the same trick commonly used for handling shared weights:

$$\begin{aligned}\frac{\partial L}{\partial \sigma} &= \sum_{j=1}^{m} \frac{\partial L}{\partial \sigma_j} \cdot \underbrace{\frac{\partial \sigma_j}{\partial \sigma}}_{=1} \\ &= \sum_{j=1}^{m} \frac{\partial L}{\partial \sigma_j} \\ &= \sum_{j=1}^{m}\sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)\, w_j \Phi_j(\overline{X_i})\, \frac{||\overline{X_i} - \overline{\mu}_j||^2}{\sigma^3}\end{aligned}$$

One can also compute a partial derivative with respect to each element of the prototype vector. Let μ_{jk} represent the kth element of $\overline{\mu}_j$. Similarly, let x_{ik} represent the kth element of the ith training point $\overline{X_i}$. The partial derivative with respect to μ_{jk} is computed as follows:

$$\frac{\partial L}{\partial \mu_{jk}} = \sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)\, w_j \Phi_j(\overline{X_i})\, \frac{(x_{ik} - \mu_{jk})}{\sigma_j^2} \tag{5.8}$$

Using these partial derivatives, one can update the bandwidth and the prototype vectors together with the weights. Unfortunately, this type of strong approach to supervision does not seem to work very well. There are two main drawbacks with this approach:

1. An attractive characteristic of RBFs is that they are efficient to train, if unsupervised methods are used. Even the orthogonal least-squares method can be run in a reasonably amount of time. However, this advantage is lost, if one resorts to full backpropagation. In general, the two-stage training of RBF is an efficiency feature of RBF networks.

2. The loss surface of RBFs has many local minima. This type of approach tends to get stuck in local minima from the point of view of generalization error.

Because of these characteristics of RBF networks, supervised training is rarely used. In fact, it has been shown in [342] that supervised training tends to increase the bandwidths and encourage generalized responses. When supervision is used, it should be used in a very controlled way by repeatedly testing performance on out-of-sample data in order to reduce the risk of overfitting.

5.3 Variations and Special Cases of RBF Networks

The above discussion only considers the case in which the supervised training is designed for numeric target variables. In practice, it is possible for the target variables to be binary. One possibility is to treat binary class labels in $\{-1, +1\}$ as numeric responses, and use the same approach of setting the weight vector according to Equation 5.4:

$$\overline{W}^T = (H^T H + \lambda I)^{-1} H^T \overline{y}$$

As discussed in Section 2.2.2.1 of Chapter 2, this solution is also equivalent to the Fisher discriminant and the Widrow-Hoff method. The main difference is that these methods are being applied on a hidden layer of increased dimensionality, which promotes better results in more complex distributions. It is also helpful to examine other loss functions that are commonly used in feed-forward neural networks for classification.

5.3.1 Classification with Perceptron Criterion

Using the notations introduced in the previous section, the prediction of the ith training instance is given by $\overline{W} \cdot \overline{H_i}$. Here, $\overline{H_i}$ represents the m-dimensional vector of activations in the hidden layer for the ith training instance $\overline{X_i}$. Then, as discussed in Section 1.2.1.1 of Chapter 1, the perceptron criterion corresponds to the following loss function:

$$L = \max\{-y_i(\overline{W} \cdot \overline{H_i}), 0\} \tag{5.9}$$

In addition, a Tikhonov regularization term with parameter $\lambda > 0$ is often added to the loss function.

Then, for each mini-batch S of training instances, let S^+ represent the misclassified instances. The misclassified instances are defined as those for which the loss L is non-zero. For such instances, applying the sign function to $\overline{H_i} \cdot \overline{W}$ will yield a prediction with opposite sign to the observed label y_i.

Then, for each mini-batch S of training instances, the following updates are used for the misclassified instances in S^+:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{H_i}, y_i) \in S^+} y_i \overline{H_i} \tag{5.10}$$

Here, $\alpha > 0$ is the learning rate.

5.3.2 Classification with Hinge Loss

The hinge loss is used frequently in the support vector machine. Indeed, the use of hinge loss in the Gaussian RBF network can be viewed as a generalization of the support-vector machine. The hinge loss is a shifted version of the perceptron criterion:

$$L = \max\{1 - y_i(\overline{W} \cdot \overline{H_i}), 0\} \tag{5.11}$$

Because of the similarity in loss functions between the hinge loss and the perceptron criterion, the updates are also very similar. The main difference is that S^+ includes only misclassified points in the case of the perceptron criterion, whereas S^+ includes both misclassified points and marginally classified points in the case of hinge loss. This is because S^+ is defined by the set of points for which the loss function is non-zero, but (unlike the perceptron criterion) the hinge loss function is non-zero even for marginally classified points. Therefore, with this modified definition of S^+, the following updates are used:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{H_i}, y_i) \in S^+} y_i \overline{H_i} \tag{5.12}$$

Here, $\alpha > 0$ is the learning rate, and $\lambda > 0$ is the regularization parameter. Note that one can easily define similar updates for the logistic loss function (cf. Exercise 2).

5.3.3 Example of Linear Separability Promoted by RBF

The main goal of the hidden layer is to perform a transformation that promotes linear separability, so that even linear classifiers work well on the transformed data. Both the perceptron and the linear support vector machine with hinge loss are known to perform poorly when the classes are not linearly separable. The Gaussian RBF classifier is able to separate out classes that are not linearly separable in the input space when loss functions such as the perceptron criterion and hinge loss are used. The key to this separability is the local transformation created by the hidden layer. An important point is that a Gaussian kernel with a small bandwidth often results in a situation where only a small number of hidden units in particular local regions get activated to significant non-zero values, whereas the other values are almost zeros. This is because of the exponentially decaying nature of the Gaussian function, which takes on near-zero values outside a particular locality. The identification of prototypes with cluster centers often divides the space into local regions, in which significant non-zero activation is achieved only in small portions of the space. As a practical matter, each local region of the space is assigned its own feature, corresponding to the hidden unit that is activated most strongly by it.

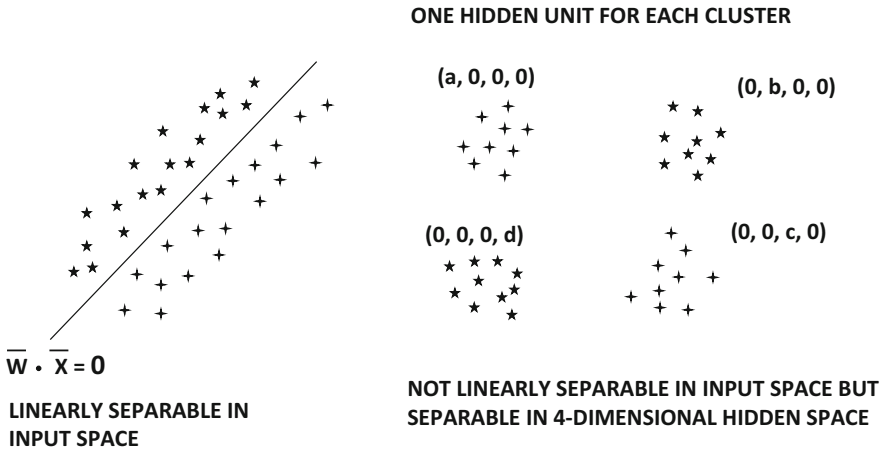

Figure 5.2: Revisiting Figure 1.4: The Gaussian RBF promotes separability because of the transformation to the hidden layer.

Examples of two data sets are illustrated in Figure 5.2. These data sets were introduced in Chapter 1 to illustrate cases that the (traditional) perceptron can or cannot solve. The traditional perceptron of Chapter 1 is able to find a solution for the data set on the left, but does not work well for the data set on the right. However, the transformation used by the Gaussian RBF method is able to address this issue of separability for the clustered data set on the right. Consider a case in which each of the centroids of the four clusters in Figure 5.2 is used as a prototype. This will result in a 4-dimensional hidden representation of the data. Note that the hidden dimensionality is higher than the input dimensionality, which is common in these settings. With appropriate choice of bandwidth, only one hidden unit will be activated strongly corresponding to the cluster identifier to which the point belongs. The other hidden units will be activated quite weakly, and will be close to 0. This will result in a rather sparse representation, as shown in Figure 5.2. We have shown the approximate 4-dimensional representations for the points in each cluster. The values of a, b, c, and d in Figure 5.2 will vary over the different points in the corresponding cluster, although they will always be strongly non-zero compared to the other coordinates. Note that one of the classes is defined by strongly non-zero values in the first and third dimensions, whereas the second class is defined by strongly non-zero values in the second and fourth dimensions. As a result, the weight vector $\overline{W} = [1, -1, 1, -1]$ will provide excellent non-linear separation between the two classes. The key point to understand is that the Gaussian RBF creates *local* features that result in separable distributions of the classes. This is exactly how a kernel support-vector machine achieves linear separability.

5.3.4 Application to Interpolation

One of the earliest applications of the Gaussian RBF was its use in interpolation of the value of a function over a set of points. The goal here is to perform *exact* interpolation of the provided points, so that the resulting function passes through all the input points. One can view interpolation as a special case of regression in which each training point is a prototype, and therefore the number of weights m in $\overline{W}$ is exactly equal to the number of training examples n. In such cases, it is possible to find a n-dimensional weight vector

$\overline{W}$ with zero error. In such a case, the activations $\overline{H_1} \ldots \overline{H_n}$ represent n-dimensional row vectors. Therefore, the matrix H obtained by stacking these row vectors on top of each other has a size of $n \times n$. Let $\overline{y} = [y_1, y_2, \ldots y_n]^T$ be the n-dimensional column vector of observed variables.

In linear regression, one attempts to minimize the loss function $||H\overline{W}^T - \overline{y}||^2$ in order to determine $\overline{W}$. This is because the matrix H is not square, and the system of equations $H\overline{W}^T = \overline{y}$ is over-complete. However, in the case of linear interpolation, the matrix H is square, and the system of equations is no longer over-complete. Therefore, it is possible to find an exact solution (with zero loss) satisfying the following system of equations:

$$H\overline{W}^T = \overline{y} \tag{5.13}$$

It can be shown that this system of equations has a unique solution when the training points are distinct from one another [323]. The value of the weight vector $\overline{W}^T$ can then be computed as follows:

$$\overline{W}^T = H^{-1}\overline{y} \tag{5.14}$$

It is noteworthy that this equation is a special case of Equation 5.6 because the pseudo-inverse of a square and non-singular matrix is the same as its inverse. In the case where the matrix H is non-singular, one can simplify the pseudo-inverse as follows:

$$\begin{aligned} H^+ &= (H^T H)^{-1} H^T \\ &= H^{-1} \underbrace{(H^T)^{-1} H^T}_{I} \\ &= H^{-1} \end{aligned}$$

Therefore, the case of linear interpolation is a special case of least-squares regression. Stated in another way, least-squares regression is a form of noisy interpolation, where it is impossible to fit the function through all the training points because of the limited degrees of freedom in the hidden layer. Relaxing the size of the hidden layer to the training data size allows exact interpolation. Exact interpolation is not necessarily better for computing the function value of out-of-sample points, because it might be the result of overfitting.

5.4 Relationship with Kernel Methods

The RBF network gains its power by mapping the input points into a high-dimensional hidden space in which linear models are sufficient to model nonlinearities. This is the same principle used by kernel methods like kernel regression and kernel SVMs. In fact, it can be shown that certain special cases of the RBF network reduce to kernel regression and kernel SVMs.

5.4.1 Kernel Regression as a Special Case of RBF Networks

The weight vector $\overline{W}$ in RBF networks is trained to minimize the squared loss of the following prediction function:

$$\hat{y}_i = \overline{H_i}\,\overline{W}^T = \sum_{j=1}^{m} w_j \Phi_j(\overline{X_i}) \tag{5.15}$$

Now consider the case in which the prototypes are the same as the training points, and therefore we set $\overline{\mu}_j = \overline{X_j}$ for each $j \in \{1 \ldots n\}$. Note that this approach is the same as that used in function interpolation, in which the prototypes are set to all the training points. Furthermore, each bandwidth σ is set to the same value. In such a case, one can write the above prediction function as follows:

$$\hat{y}_i = \sum_{j=1}^{n} w_j \exp\left(-\frac{||\overline{X_i} - \overline{X_j}||^2}{2\sigma^2}\right) \tag{5.16}$$

The exponentiated term on the right-hand side of Equation 5.16 can be written as the Gaussian kernel similarity between points $\overline{X_i}$ and $\overline{X_j}$. This similarity is denoted by $K(\overline{X_i}, \overline{X_j})$. Therefore, the prediction function becomes the following:

$$\hat{y}_i = \sum_{j=1}^{n} w_j K(\overline{X_i}, \overline{X_j}) \tag{5.17}$$

This prediction function is exactly the same as that used in kernel regression with bandwidth σ, where the prediction function $\hat{y}_i^{kernel}$ is defined[1] in terms of the *Lagrange multipliers* λ_j instead of weight w_j (see, for example, [6]):

$$\hat{y}_i^{kernel} = \sum_{j=1}^{n} \lambda_j y_j K(\overline{X_i}, \overline{X_j}) \tag{5.18}$$

Furthermore, the (squared) loss function is the same in the two cases. Therefore, a one-to-one correspondence will exist between the Gaussian RBF solutions and the kernel regression solutions, so that setting $w_j = \lambda_j y_j$ leads to the same value of the loss function. Therefore, their optimal values will be the same as well. In other words, the Gaussian RBF network provides the same results as kernel regression in the special case where the prototype vectors are set to the training points. However, the RBF network is more powerful and general because it can choose different prototype vectors; therefore, the RBF network can model cases that are not possible with kernel regression. In this sense, it is helpful to view the RBF network as a flexible neural variant of kernel methods.

5.4.2 Kernel SVM as a Special Case of RBF Networks

Like kernel regression, the kernel support vector machine (SVM) is also a special case of RBF networks. As in the case of kernel regression, the prototype vectors are set to the training points, and the bandwidths of all hidden units are set to the same value of σ. Furthermore, the weights w_j are learned in order to minimize the hinge loss of the prediction.

In such a case, it can be shown that the prediction function of the RBF network is as follows:

$$\hat{y}_i = \text{sign}\left\{\sum_{j=1}^{n} w_j \exp\left(-\frac{||\overline{X_i} - \overline{X_j}||^2}{2\sigma^2}\right)\right\} \tag{5.19}$$

$$\hat{y}_i = \text{sign}\left\{\sum_{j=1}^{n} w_j K(\overline{X_i}, \overline{X_j})\right\} \tag{5.20}$$

[1] A full explanation of the kernel regression prediction of Equation 5.18 is beyond the scope of this book. Readers are referred to [6].

It is instructive to compare this prediction function with that used in kernel SVMs (see, for example, [6]) with the Lagrange multipliers λ_j:

$$\hat{y}_i^{kernel} = \text{sign}\left\{\sum_{j=1}^{n} \lambda_j y_j K(\overline{X_i}, \overline{X_j})\right\} \tag{5.21}$$

This prediction function is of a similar form as that used in kernel SVMs, with the exception of a slight difference in the variables used. The hinge-loss is used as the objective function in both cases. By setting $w_j = \lambda_j y_j$ one obtains the same result in both cases in terms of the value of the loss function. Therefore, the optimal solutions in the kernel SVM and the RBF network will also be related according to the condition $w_j = \lambda_j y_j$. In other words, the kernel SVM is also a special case of RBF networks. Note that the weight w_j can also be considered the coefficient of each data point, when the *representer theorem* is used in kernel methods [6].

5.4.3 Observations

One can extend the arguments above to other linear models, such as the kernel Fisher discriminant and kernel logistic regression, by changing the loss function. In fact, the kernel Fisher discriminant can be obtained by simply using the binary variables as the targets and then applying kernel regression technique. However, since the Fisher discriminant works under the assumption of centered data, a bias needs to be added to the output layer to absorb any offsets from uncentered data. Therefore, the RBF network can simulate virtually any kernel method by choosing an appropriate loss function. A key point is that the RBF network provides more flexibility than kernel regression or classification. For example, one has much more flexibility in choosing the number of nodes in the hidden layer, as well as the number of prototypes. Choosing the prototypes wisely in a more economical way helps in both accuracy and efficiency. There are a number of key trade-offs associated with these choices:

1. Increasing the number of hidden units increases the complexity of the modeled function. It can be useful for modeling difficult functions, but it can cause overfitting, if the modeled function is not truly complex.

2. Increasing the number of hidden units increases the complexity of training.

One way of choosing the number of hidden units is to hold out a portion of the data, and estimate the accuracy of the model on the held-out set with different numbers of hidden units. The number of hidden units is then set to a value that optimizes this accuracy.

5.5 Summary

This chapter introduces radial basis function (RBF) networks, which represent a fundamentally different way of using the neural network architecture. Unlike feed-forward networks, the hidden layer and output layer are trained in a somewhat different way. The training of the hidden layer is unsupervised, whereas that of the output layer is supervised. The hidden layer usually has a larger number of nodes than the input layer. The key idea is to transform the data points into high-dimensional space with the use of locality-sensitive transformations, so that the transformed points become linearly separable. The approach

can be used for classification, regression, and linear interpolation by changing the nature of the loss function. In classification, one can use different types of loss functions such as the Widrow-Hoff loss, the hinge loss, and the logistic loss. Special cases of different loss functions specialize to well-known kernel methods such as kernel SVMs and kernel regression. The RBF network has rarely been used in recent years, and it has become a forgotten category of neural architectures. However, it has significant potential to be used in any scenario where kernel methods are used. Furthermore, it is possible to combine this approach with feed-forward architectures by using multi-layered representations following the first hidden layer.

5.6 Bibliographic Notes

RBF networks were proposed by Broomhead and Lowe [51] in the context of function interpolation. The separability of high-dimensional transformations is shown in Cover's work [84]. A review of RBF networks may be found in [363]. The books by Bishop [41] and Haykin [182] also provide good treatments of the topic. An overview of radial basis functions is provided in [57]. The proof of universal function approximation with RBF networks is provided in [173, 365]. An analysis of the approximation properties of RBF networks is provided in [366].

Efficient training algorithms for RBF networks are described in [347, 423]. An algorithm for learning the center locations in RBF networks is proposed in [530]. The use of decision trees to initialize RBF networks is discussed in [256]. The orthogonal least-squares algorithm was proposed in [65]. Early comparisons of supervised and unsupervised training of RBF networks are provided in [342]. According to this analysis, full supervision seems to increase the likelihood of the network getting trapped in local minima. Some ideas on improving the generalization power of RBF networks are provided in [43]. Incremental RBF networks are discussed in [125]. A detailed discussion of the relationship between RBF networks and kernel methods is provided in [430].

5.7 Exercises

Some exercises require additional knowledge about machine learning that is not discussed in this book. Exercises 5, 7, and 8 require additional knowledge of kernel methods, spectral clustering, and outlier detection.

1. Consider the following variant of radial basis function networks in which the hidden units take on either 0 or 1 values. The hidden unit takes on the value of 1, if the distance to a prototype vector is less than σ. Otherwise it takes on the value of 0. Discuss the relationship of this method to RBF networks, and its relative advantages/disadvantages.

2. Suppose that you use the sigmoid activation in the final layer to predict a binary class label as a probability in the output node of an RBF network. Set up a negative log-likelihood loss for this setting. Derive the gradient-descent updates for the weights in the final layer. How does this approach relate to the logistic regression methods discussed in Chapter 2? In which case will this approach perform better than logistic regression?

3. Discuss why an RBF network is a supervised variant of a nearest-neighbor classifier.

4. Discuss how you can extend the three multi-class models discussed in Chapter 2 to RBF networks. In particular discuss the extension of the (a) multi-class perceptron, (b) Weston-Watkins SVM, and (c) softmax classifier with RBF networks. Discuss how these models are more powerful than the ones discussed in Chapter 2.

5. Propose a method to extend RBF networks to unsupervised learning with autoencoders. What will you reconstruct in the output layer? A special case of your approach should be able to roughly simulate kernel singular value decomposition.

6. Suppose that you change your RBF network so that you keep only the top-k activations in the hidden layer, and set the remaining activations to 0. Discuss why such an approach will provide improved classification accuracy with limited data.

7. Combine the top-k method of constructing the RBF layer in Exercise 6 with the RBF autoencoder in Exercise 5 for unsupervised learning. Discuss why this approach will create representations that are better suited to clustering. Discuss the relationship of this method with spectral clustering.

8. The manifold view of outliers is to define them as points that do not naturally fit into the nonlinear manifolds of the training data. Discuss how you can use RBF networks for unsupervised outlier detection.

9. Suppose that instead of using the RBF function in the hidden layer, you use dot products between prototypes and data points for activation. Show that a special case of this setting reduces to a linear perceptron.

10. Discuss how you can modify the RBF autoencoder in Exercise 5 to perform semi-supervised classification, when you have a lot of unlabeled data, and a limited amount of labeled data.