14 Architectures

This chapter looks at some of the ways that functions can be connected to *interpolate* among and *extrapolate* beyond observations. The essence of successful generalization lies in finding a form that is just capable enough, but no more. There is a recurring tension between describing too little (using a function that does not have enough flexibility to follow the data) and too much (using a function that is so flexible that it fits noise, or produces artifacts absent in the original data). Along with matching the data, supporting concerns will be how easy a family of functions is to use and to understand. Most anyone who has done any kind of mathematical modeling has made such choices, often without being aware of the problems of old standbys and the power of less well-known alternatives.

Polynomials are often the first, and sometimes the last, functional form encountered for fitting. Familiarity can breed contempt; in this chapter we'll see some of the problems with using polynomials for nontrivial problems, and consider some of the appealing alternatives.

14.1 POLYNOMIALS

14.1.1 Padé Approximants

As we've already seen many times, a *polynomial* expansion has the form (in 1D)

$$y(x) = \sum_{n=0}^{N} a_n x^n \quad . \tag{14.1}$$

One failing of polynomials is that they have a hard time matching a function that has a pole. Since $(1-x)^{-1} = 1+x+x^2+x^3+\ldots$, a simple pole is equivalent to an infinite-order power series. This is more serious than you might think because even if a function is evaluated only along the real axis, a nearby complex pole can slow down the convergence of a polynomial expansion. An obvious improvement is to use a ratio of polynomials,

$$y(x) = \frac{\sum_{n=1}^{N} a_n x^n}{1 + \sum_{m=1}^{M} b_m x^m} \quad . \tag{14.2}$$

This is called a *Padé approximant* [Baker & Graves-Morris, 1996]. By convention the constant term in the denominator is taken to be 1 since it's always possible to rescale the coefficients by multiplying the numerator and denominator by a constant. The order of the approximation is written $[N/M]$.

One application of Padé approximants is for functional approximation. If a function has a known Taylor expansion

$$f(x) = \sum_{l=0}^{\infty} c_l x^l \quad , \tag{14.3}$$

we can ask that it be matched up to some order by a Padé approximant

$$\frac{\sum_{n=1}^{N} a_n x^n}{1 + \sum_{m=1}^{M} b_m x^m} = \sum_{l=0}^{L} c_l x^l \quad . \tag{14.4}$$

The coefficients can be found by multiplying both sides by the denominator of the left hand side,

$$\sum_{n=0}^{N} a_n x^n = \sum_{l=0}^{L} c_l x^l + \sum_{m=1}^{M} \sum_{l=0}^{L} b_m c_l x^{l+m} \quad , \tag{14.5}$$

and equating powers of x. For powers $n \leq N$ this gives

$$a_n = c_n + \sum_{m=1}^{N} b_m c_{n-m} \quad (n = 0, \ldots, N) \tag{14.6}$$

and for higher powers

$$0 = c_l + \sum_{m=1}^{M} b_m c_{l-m} \quad (l = N+1, \ldots, N+M) \quad . \tag{14.7}$$

The latter relationship gives us M equations that can be solved to find the M b_m's in terms of the c's, which can then be plugged into the former equation to find the a's.

Alternatively, given a data set $\{y_i, x_i\}$ we can look for the best agreement by a Padé approximant. Relabeling the coefficients, this can be written as

$$\frac{\sum_{n=1}^{N} a_n x_i^n}{1 + \sum_{n=N+1}^{M+N} a_n x_i^{n-N}} = y_i \quad . \tag{14.8}$$

Once again multiplying both sides by the denominator of the left hand side and rearranging terms gives

$$\sum_{n=0}^{N} x_i^n a_n - \sum_{n=N+1}^{N+M} x_i^{n-N} y_i a_n = y_i \quad , \tag{14.9}$$

or $\mathbf{M} \cdot \vec{a} = \vec{y}$, where the ith row of $\mathbf{M}$ is $(x_i^0, \ldots, x_i^N, -x_i^1 y_i, \ldots, -x_i^N y_i)$, $\vec{a} = (a_0, \ldots, a_{N+M})$, and $\vec{y}$ is the vector of observations. This can now be solved by SVD to find $\vec{a} = \mathbf{M}^{-1} \cdot \vec{y}$, the set of coefficients that minimize the sum of the squares of the errors.

When Padé approximants are appropriate they work very well. They can converge remarkably quickly (Problem 13.1), and can have an almost mystical ability to find analytical structure in a function that lets the approximation be useful far beyond where there are any small parameters in an expansion. However, their use requires care. After all they're guaranteed to have poles whether or not the function being approximated does; the domain and rate of convergence can be difficult to anticipate.

14.1.2 Splines

Another problem with global polynomials is that they have a hard time following local behavior, resulting in undesirable artifacts such as overshooting and ringing (Problem 13.2). In Chapter 9 we saw one solution to this problem: define polynomials in finite elements, and then patch these local functions together with appropriate matching conditions. *Splines* use cubic polynomials to match their value and first and second derivatives at the boundaries (although the term is sometimes used even if just slopes are being matched). There are almost as many types of splines as there are applications, differing in how the constraints are imposed. Splines are not commonly used for data fitting or functional approximation, because they require finding and keeping track of the boundaries as well as the coefficients, but they are the workhorse of computer graphics, where they are used to represent curves and surfaces.

Natural splines pass through a given set of points, matching the derivatives as they cross the points. This is the most obvious way to define a spline, but it has the liability that moving one point affects all of the others. To be useful for an interactive application such as computer graphics it must be possible to make a local change and not have to recompute the entire curve. *Nonuniform B-splines* provide a clever solution to this problem [Foley *et al*., 1990]. The "B" part refers to the bases the polynomials provide, and the "nonuniform" part refers to the flexibility in defining how they join together.

A nonuniform B-spline curve is defined by

$$\begin{aligned}\vec{x}_i(t) = \vec{c}_{i-3}\varphi_{i-3}(t) + \vec{c}_{i-2}\varphi_{i-2}(t) \\ +\vec{c}_{i-1}\varphi_{i-1}(t) + \vec{c}_i\varphi_i(t) \quad (t_i \le t < t_{i+1}) \ .\end{aligned} \tag{14.10}$$

t parameterizes the position along the curve. The $\vec{c}_i$'s are *control points*, defined at given values of t called *knots*, t_i. The φ_i's are the basis (or *blending*) functions that weight the control points. The basis functions are found recursively:

$$\begin{aligned}\varphi_i^{(1)}(t) &= \begin{cases} 1 & (t_i \le t < t_{i+1}) \\ 0 & \text{otherwise} \end{cases} \\ \varphi_i^{(2)}(t) &= \frac{t - t_i}{t_{i+1} - t_i}\varphi_i^{(1)}(t) + \frac{t_{i+2} - t}{t_{i+2} - t_{i+1}}\varphi_{i+1}^{(1)}(t) \\ \varphi_i^{(3)}(t) &= \frac{t - t_i}{t_{i+2} - t_i}\varphi_i^{(2)}(t) + \frac{t_{i+3} - t}{t_{i+3} - t_{i+1}}\varphi_{i+1}^{(2)}(t) \\ \varphi_i(t) \equiv \varphi_i^{(4)}(t) &= \frac{t - t_i}{t_{i+3} - t_i}\varphi_i^{(3)}(t) + \frac{t_{i+4} - t}{t_{i+4} - t_{i+1}}\varphi_{i+1}^{(3)}(t) \ .\end{aligned} \tag{14.11}$$

The first line sets the basis function to zero except in intervals where the knots are increasing. The second linearly interpolates between these constants, the third does a linear interpolation between these linear functions to give quadratic polynomials, and the last line does one more round of linear interpolation to provide the final cubics. This hierarchy of linear interpolation preserves the original normalization, so that in each interval the basis functions sum to 1. This means that the curve is bounded by the polygon that has the control points as vertices, with the basis functions controlling how close it comes to them.

Figure 14.1 shows the construction of the basis functions for the knot sequence [1 2 3 4 5 6 7 8]. The curve will approach each control point in turn, but not quite

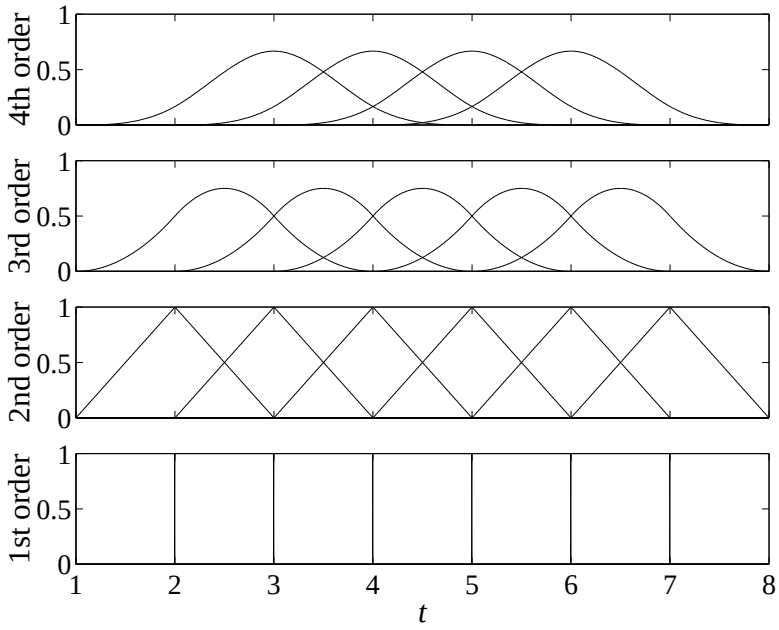

Figure 14.1. Construction of non-uniform B-spline basis functions for the knot sequence [1 2 3 4 5 6 7 8].

reach it because the maximum of the basis functions is less than 1. Figure 14.2 repeats the construction for the knot sequence [0 0 0 0 1 1 1 1]. The repetitions force the curve to go through the first and last control points, and the two intermediate control points set the slopes at the boundaries. In the context of computer graphics the B-spline is called a *Bézier curve*, and the basis functions are called *Bernstein polynomials*.

B-splines can be generalized to higher dimensions, parameterized by more than one degree of freedom. Another important extension is to divide the polynomials for each coordinate by a scale-setting polynomial. These are called *nonuniform rational B-splines*, or *NURBS*. Along with the other benefits of ratios of polynomials described in the last section, this is convenient for implementing transformations such as a change of perspective. Nonuniform nonrational B-splines are a special case with the divisor polynomial equal to 1.

14.2 ORTHOGONAL FUNCTIONS

In a polynomial fit the coefficients at each order are intimately connected. Dropping a high-order term gives a completely different function; it's necessary to re-fit to find a lower-order approximation. For many applications it's convenient to have a functional representation that lets successive terms be added to improve the agreement without changing the coefficients that have already been found. For example, if an image is

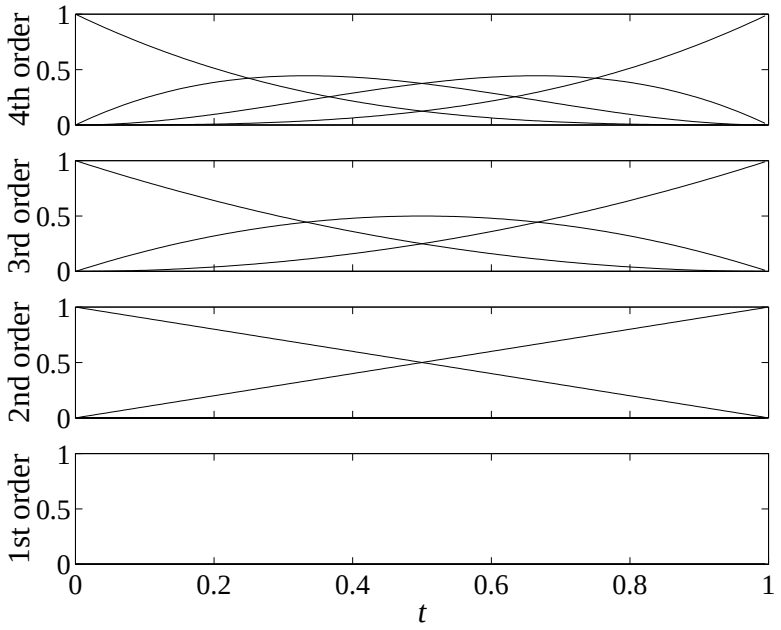

Figure 14.2. Construction of nonuniform B-spline basis functions for the knot sequence [0 0 0 0 1 1 1 1].

stored this way then the fidelity with which it is retrieved can be varied by changing the number of terms used based on the available display, bandwidth, and processor.

Expansions that let successive corrections be added without changing lower-order approximations are done with *orthogonal functions*, the functional analog to the orthogonal transformations that we saw in the last chapter. We have been writing an expansion of a function as a sum of scaled basis terms

$$y(\vec{x}) = \sum_{i=1}^{M} a_i f_i(\vec{x}) \tag{14.12}$$

without worrying about how the f_i's relate to each other. However, if we choose them to be *orthonormal*

$$\int_{-\infty}^{\infty} f_i(\vec{x}) f_j(\vec{x})\, d\vec{x} = \delta_{ij} \quad , \tag{14.13}$$

then the expansion coefficients can be found by projection:

$$\begin{aligned} \int_{-\infty}^{\infty} f_i(\vec{x}) y(\vec{x})\, d\vec{x} &= \int_{-\infty}^{\infty} f_i(\vec{x}) \sum_{j=1}^{M} a_j f_j(\vec{x})\, d\vec{x} \\ &= \sum_{j=1}^{M} a_j \delta_{ij} \\ &= a_i \quad . \end{aligned} \tag{14.14}$$

Such an expansion is useful if we can analytically or numerically do the integral on the left hand side of equation (14.14).

Assume that we're given an arbitrary *complete* set of basis functions $\varphi_i(\vec{x})$ that can represent any function of interest, for example polynomials up to a specified order. An orthonormal set f_i can be constructed by *Gram–Schmidt orthogonalization*. This is exactly the same procedure used to construct orthogonal vectors, with the dot product between vectors replaced by an integral over the product of functions. The first orthonormal function is found by scaling the first basis function

$$f_1(\vec{x}) = \frac{\varphi_1(\vec{x})}{[\int_{-\infty}^{\infty} \varphi_1(\vec{x})\varphi_1(\vec{x})\, d\vec{x}]^{1/2}} \tag{14.15}$$

so that it is normalized:

$$\int_{-\infty}^{\infty} f_1(\vec{x})f_1(\vec{x})\, d\vec{x} \;=\; \frac{\int_{-\infty}^{\infty} \varphi_1(\vec{x})\varphi_1(\vec{x})\, d\vec{x}}{\int_{-\infty}^{\infty} \varphi_1(\vec{x})\varphi_1(\vec{x})\, d\vec{x}} \;=\; 1 \quad . \tag{14.16}$$

The next member of the set is found by subtracting off the f_1 component of φ_2

$$f_2'(\vec{x}) = \varphi_2(\vec{x}) - f_1(\vec{x}) \int_{-\infty}^{\infty} f_1(\vec{x})\varphi_2(\vec{x})\, d\vec{x} \tag{14.17}$$

to produce a function that is orthogonal to f_1

$$\begin{aligned} \int_{-\infty}^{\infty} f_1(\vec{x})f_2'(\vec{x})\, d\vec{x} &= \int_{-\infty}^{\infty} f_1(\vec{x})\varphi_2(\vec{x})\, d\vec{x} \\ &\quad - \underbrace{\int_{-\infty}^{\infty} f_1(\vec{x})f_1(\vec{x})\, d\vec{x}}_{1} \int_{-\infty}^{\infty} f_1(\vec{x})\varphi_2(\vec{x})\, d\vec{x} \\ &= 0 \quad , \end{aligned} \tag{14.18}$$

and then normalizing it

$$f_2(\vec{x}) = \frac{f_2'(\vec{x})}{[\int_{-\infty}^{\infty} f_2'(\vec{x})f_2'(\vec{x})\, d\vec{x}]^{1/2}} \quad . \tag{14.19}$$

The third one is found by subtracting off these first two components and normalizing:

$$\begin{aligned} f_3'(\vec{x}) &= \varphi_3 - f_1 \int_{-\infty}^{\infty} f_1\varphi_3\, d\vec{x} - f_2 \int_{-\infty}^{\infty} f_2\varphi_3\, d\vec{x} \\ f_3(\vec{x}) &= \frac{f_3'(\vec{x})}{[\int_{-\infty}^{\infty} f_3'(\vec{x})f_3'(\vec{x})\, d\vec{x}]^{1/2}} \quad , \end{aligned} \tag{14.20}$$

and so forth until the basis is constructed.

If a set of experimental observations $\{y_n, \vec{x}_n\}_{n=1}^{N}$ is available instead of a function form $y(\vec{x})$ then it is not possible to directly evaluate the projection integrals in equation (14.14). We can still use orthonormal functions in this case, but they must now be constructed to be orthonormal with respect to the unknown probability density $p(\vec{x})$ of the measurements (assuming that the system is stationary so that the density exists). If we use the same expansion (equation 14.12) and now choose the f_i's so that

$$\langle f_i(\vec{x})f_j(\vec{x})\rangle = \int_{-\infty}^{\infty} f_i(\vec{x})f_j(\vec{x})p(\vec{x})\, d\vec{x} = \delta_{ij} \quad , \tag{14.21}$$

then the coefficients are found by the expectation

$$\begin{aligned}\langle y(\vec{x})f_i(\vec{x})\rangle &= \int_{-\infty}^{\infty} y(\vec{x})f_i(\vec{x})p(\vec{x})\ d\vec{x} \\ &= a_i \quad .\end{aligned} \tag{14.22}$$

By definition an integral over a distribution can be approximated by an average over variables drawn from the distribution, providing an experimental means to find the coefficients

$$\begin{aligned}a_i &= \int_{-\infty}^{\infty} y(\vec{x})f_i(\vec{x})p(\vec{x})\ d\vec{x} \\ &\approx \frac{1}{N}\sum_{n=1}^{N} y_n f_i(\vec{x}_n) \quad .\end{aligned} \tag{14.23}$$

But where do these magical f_i's come from? The previous functions that we constructed were orthogonal with respect to a flat distribution. To orthogonalize with respect to the experimental distribution all we need to do is replace the integrals with sums over the data,

$$\begin{aligned}f_1(\vec{x}) &= \frac{\varphi_1(\vec{x})}{[\int_{-\infty}^{\infty} \varphi_1(\vec{x})\varphi_1(\vec{x})p(\vec{x})]^{1/2}\ d\vec{x}} \\ &\approx \frac{\varphi_1(\vec{x})}{[\frac{1}{N}\sum_{n=1}^{N} \varphi_1(\vec{x}_n)\varphi_1(\vec{x}_n)]^{1/2}} \\ f_2'(\vec{x}) &= \varphi_2(\vec{x}) - f_1(\vec{x})\int_{-\infty}^{\infty} f_1(\vec{x})\varphi_2(\vec{x})p(\vec{x})\ d\vec{x} \\ &\approx \varphi_2(\vec{x}) - f_1(\vec{x})\frac{1}{N}\sum_{n=1}^{N} f_1(\vec{x}_n)\varphi_2(\vec{x}_n)\end{aligned} \tag{14.24}$$

and so forth. This construction lets functions be fit to data by evaluating experimental expectations rather than doing an explicit search for the fit coefficients. Further work can be saved if the starting functions are chosen to satisfy known constraints in the problem, such as using sin functions to force the expansion to vanish at the boundaries of a rectangular region.

Basis functions can of course be orthogonalized with respect to known distributions. For polynomials, many of these families have been named because of their value in mathematical methods, including the *Hermite polynomials* (orthogonal to with respect to e^{-x^2}), *Laguerre polynomials* (orthogonal with respect to e^{-x}), and the *Chebyshev polynomials* (orthogonal with respect to $(1-x^2)^{-1/2}$) [Arfken & Weber, 1995].

14.3 RADIAL BASIS FUNCTIONS

Even dressed up as a family of orthogonal functions, there is a serious problem inherent in the use of any kind of polynomial expansion: the only way to improve a fit is to add higher-order terms that diverge ever more quickly. Matching data that isn't similarly diverging requires a delicate balancing of the coefficients, resulting in a function that is increasingly "wiggly." Even if it passes near the training data it will be useless for

interpolation or extrapolation. This is particularly true for functions that have isolated features, such as discontinuities or sharp peaks (Problem 13.2).

Radial basis functions (*RBFs*) offer a sensible alternative to the hopeless practice of fighting divergences with still faster divergences. The idea is to use as the basis a set of identical functions that depend only on the radius from the data point to a set of "representative" locations $\vec{c}_i$:

$$y(\vec{x}) = \sum_{i=1}^{M} f(|\vec{x} - \vec{c}_i|; \vec{a}_i) \quad . \tag{14.25}$$

The $\vec{a}_i$ are coefficients associated with the ith center $\vec{c}_i$. Now extra terms can be added without changing how quickly the model diverges, and the centers can be placed where they're needed to improve the fit in particular areas. And unlike splines the basis functions are defined everywhere, eliminating the need to keep track of element boundaries.

If the centers are fixed and the coefficients enter linearly then fitting an RBF becomes a linear problem

$$y = \sum_{i=1}^{M} a_i f(|\vec{x} - \vec{c}_i|) \quad . \tag{14.26}$$

Common choices for these f include linear ($f(r) = r$), cubic ($f(r) = r^3$), and fixed-width Gaussian ($f(r) = \exp(-r^2)$). $f(r) = r^2$ is not included in this list because it is equivalent to a single global second-order polynomial ($|\vec{x} - \vec{c}|^2 = \sum_j [x_j - c_j]^2$). For example, fitting $f(r) = r^3$ requires a singular value decomposition to invert

$$\begin{pmatrix} |\vec{x}_1 - \vec{c}_1|^3 & |\vec{x}_1 - \vec{c}_2|^3 & \cdots & |\vec{x}_1 - \vec{c}_M|^3 \\ |\vec{x}_2 - \vec{c}_1|^3 & |\vec{x}_2 - \vec{c}_2|^3 & \cdots & |\vec{x}_2 - \vec{c}_M|^3 \\ |\vec{x}_3 - \vec{c}_1|^3 & |\vec{x}_3 - \vec{c}_2|^3 & \cdots & |\vec{x}_3 - \vec{c}_M|^3 \\ \vdots & \vdots & \cdots & \vdots \\ |\vec{x}_{N-1} - \vec{c}_1|^3 & |\vec{x}_{N-1} - \vec{c}_2|^3 & \cdots & |\vec{x}_{N-1} - \vec{c}_M|^3 \\ |\vec{x}_N - \vec{c}_1|^3 & |\vec{x}_N - \vec{c}_2|^3 & \cdots & |\vec{x}_N - \vec{c}_M|^3 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix}$$

to find the solution with the least squares error.

Linear and cubic basis functions appear to violate the spirit of locality of RBFs because they try to expand the unknown function in a set of globally diverging functions, which might seem to require the same kind of delicate cancellation that dooms a high-order polynomial expansion. This intuition misses the fact that all of the terms have the same order and so this can be a sensible thing to do. In fact, a possibly counter-intuitive result is that these diverging basis functions can have better error convergence properties than local ones [Powell, 1992]. They share the crucial attribute that if the data are rescaled so that $r \to \alpha r$, then $r^3 \to (\alpha r)^3 = \alpha^3 r^3$, simply changing the amplitude of the term, which we are going to fit anyway. But for a Gaussian $\exp(-r^2) \to \exp(-\alpha^2 r^2)$, changing the variance of the Gaussian which cannot be modified by the amplitude of the term.

There are many strategies for choosing the $\vec{c}_i$: randomly, uniformly, drawn from the data, uniformly on the support of the data (regions where the probability to see a point is nonzero). If the basis coefficients enter nonlinearly (for example, changing the variance of a Gaussian) then an iterative nonlinear search is needed and RBFs are closely related to the network architectures to be discussed in Section 14.6. If the centers are allowed to

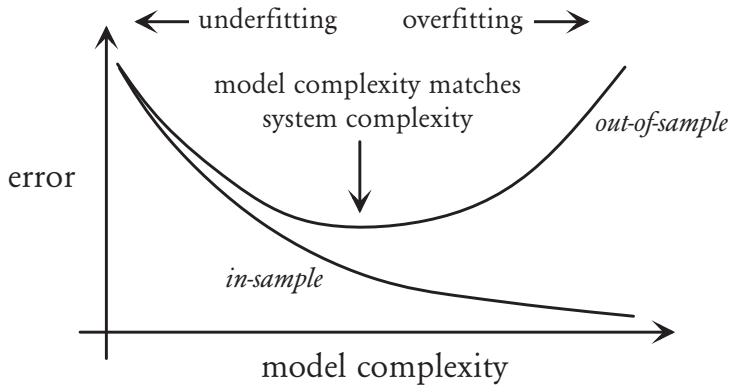

Figure 14.3. Cross-validation.

move (for example, by varying the mean of a Gaussian), RBFs become a particular case of the general theory of approximation by clustering (Chapter 16). The last decision in using an RBF is the number of centers M, to be discussed in the next section.

14.4 OVERFITTING

Overfitting is a simple idea with deep implications. Any reasonable model with N free parameters should be able to exactly fit N data points, but that is almost always a bad idea. The goal of fitting usually is to be able to interpolate or extrapolate. If the data are at all noisy, a model that passes through every point is carefully fitting idiosyncracies of the noise that do not generalize. On the other hand, if too few parameters are used, the model may be forced to not only ignore the noise but also miss the meaningful behavior of the data. Successful function fitting requires balancing *underfitting*, where there are *model mistmatch errors* because the model can not describe the data, with *overfitting*, where there are *model estimation errors* from poor parameter choices in a flexible model. This is another example of a bias–variance tradeoff: you must choose between a reliable estimate of a biased model, or a poor estimate of a model that is capable of a better fit.

Unlike the parameters of a model that are determined by fitting, the *hyper-parameters* that control the architecture of a model must be determined by some kind of procedure outside the fitting. *Bootstrap* sampling (Section 12.5) provides a way to estimate the reliability of a model's parameters, but by mixing up the data used for training and evaluating the model it can easily lead to overconfidence in a model's ability to generalize on entirely new measurements. *Cross-validation* separates these functions. The idea is to fit your model on part of the data set, but evaluate the fit on another part of the data set that you have withheld (for example, by randomly choosing a subset of the data). Now repeat this procedure as you vary the complexity of the model, such as the number of RBF centers. The *in-sample* error will continue to decrease if the model is any good, but the *out-of-sample* error will initially decrease but then stop improving and possibly start increasing once the model begins to overfit (Figure 14.3). The best model is the one at the minimum of this curve (although the curve usually won't be this nice). Once

this set of hyper-parameters has been found the model can be retrained on the full data set to make predictions for completely new values. The essential observation is that the only way to distinguish between a signal and noise, without any extra information, is by testing the model's generalization ability. Cross-validation is a sensible, useful way to do that, albeit with a great deal of associated debate about when and how to use it properly, e.g., [Goutte, 1997].

A strict Bayesian views (correctly) cross-validation as an *ad hoc* solution to finding hyperparameters. Better to use equation (12.1) and integrate over all possible models [Buntine & Weigend, 1991; Besag *et al*., 1995]. This is an enormous amount of work, and requires specifying priors on all the variables in the problem. There are some problems where this effort is justified, but for the common case of there being many equally acceptable models cross-validation is a quick and easy way to find one of them.

Finally, just as the Fisher information (Section 12.6) can be used to bound the performance of an estimator, it can be possible to relate the descriptive power of a model architecture to the amount of data needed to use it, in advance of making any measurements. The *Vapnik–Chervonenkis* (VC) dimension is equal to the size of the largest set of points that can be classified in all possible ways by a model, and for various architectures it can be used to find the worst-case scaling of the number of points required to learn a model to a given uncertainty [Vapnik & Chervonenkis, 1971, Kearns & Vazirani, 1994]. Unfortunately, as with so many other grand unifying principles in learning theory, the real-world problems where such guidance is most needed is where it is most difficult to apply.

14.5 CURSE OF DIMENSIONALITY

There are two broad classes of basis functions: those with linear coefficients

$$y(\vec{x}) = \sum_{i=1}^{M} a_i f_i(\vec{x}) \quad , \tag{14.27}$$

and those with coefficients that enter nonlinearly

$$y(\vec{x}) = \sum_{i=1}^{M} f_i(\vec{x}; \vec{a}_i) \quad . \tag{14.28}$$

The former has a single global least-squares minimum that can be found by singular value decomposition; in general the latter requires an iterative search with the attendant problems of avoiding local minima and deciding when to stop. It's impossible to find an algorithm that can solve an arbitrary nonlinear search problem in a single step because the function f could be chosen to implement the mapping performed by a general-purpose computer from an input program $\vec{x}$ to an output y; if such an omniscient search algorithm existed it would be able to solve computationally intractable problems [Garey & Johnson, 1979].

For a nonparametric fit where we have no idea what the functional form should be, why would we ever want to use nonlinear coefficients? The answer is the *curse of dimensionality*. This ominous sounding problem is invoked by a few different incantations, all

relating to serious difficulties that occur as the dimensionality of a problem is increased. Let's say that we wanted to do a second-order polynomial fit. In 1D this is simply

$$y = a_0 + a_1 x + a_2 x^2 \quad . \tag{14.29}$$

In 2D, we have

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 + a_4 x_1^2 + a_5 x_2^2 \quad . \tag{14.30}$$

We've gone from three to six terms. In 3D we need ten terms:

$$\begin{aligned} y = a_0 &+ a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_1 x_2 + a_5 x_1 x_3 + a_6 x_2 x_3 \\ &+ a_7 x_1^2 + a_8 x_2^2 + a_9 x_3^2 \quad , \end{aligned} \tag{14.31}$$

and in d dimensions we need $1 + 2d + d(d-1)/2$ terms. As d is increased the quadratic d^2 part quickly dominates. If we try to get around this by leaving off some of the polynomial terms then we do not let those particular variables interact at that order. As we increase the order of the polynomial, the exponent of the number of terms required increases accordingly. A 10th-order fit in 10D will require on the order of 10^{10} terms, clearly prohibitive. This rapid increase in the number of terms required is an example of the curse of dimensionality. Because of this many algorithms that give impressive results for a low-dimensional test problem fail miserably for a realistic high-dimensional problem.

The reason to use nonlinear coefficients is to avoid the curse. Given some mild assumptions, a nonlinear function of the form of equation (14.27) will have a typical approximation error of $\mathcal{O}(1/M^{2/d})$ when fitting an arbitrary but reasonable function (one with a bound on the first moment of the Fourier transform), where M is the number of terms and d is the dimension of the space, while a linear function of the form of equation (14.28) will have an error of order $\mathcal{O}(1/M)$ [Barron, 1993]. In low dimensions the difference is small, but in high dimensions it is essential: exponentially more terms are needed if linear coefficients are used. The nonlinear coefficients are much more powerful, able to steer the basis functions where they are needed in a high-dimensional space. Therefore, in low dimensions it's crazy not to use linear coefficients, while in high dimensions it is crazy to use them.

There is another sense of the curse of dimensionality to be mindful of. In a high-dimensional space, everything is surface. To understand this consider the volume of a hyper-sphere of radius r in a d-dimensional space:

$$V(r) = \frac{\pi^{d/2} r^d}{(d/2)!} \tag{14.32}$$

(noninteger factorials are defined by the gamma function $\Gamma(n+1) = n!$). Now let's look at the fraction of volume in a shell of thickness ϵ, compared to the total volume of the sphere, in the limit that $d \to \infty$:

$$\begin{aligned} \frac{V(r) - V(r-\epsilon)}{V(r)} &= \frac{r^d - (r-\epsilon)^d}{r^d} \\ &= 1 - \underbrace{\left(1 - \frac{\epsilon}{r}\right)^d}_{\lim_{d\to\infty} = 0} \\ &= 1 \quad . \end{aligned} \tag{14.33}$$

As $d \to \infty$ all of the volume is in a thin shell near the surface. This is because as d is increased there is more surface available. Now think about what this implies for data collection. "Typical" points come from the interior of a distribution, not near the edges, but in a high-dimensional space distributions are essentially all edges. This means that in a huge data set there might not be a single point in the interior of the distribution, and so any analysis will be dominated by edge effects. Most obviously, the data will appear to be drawn from a lower-dimensional distribution [Gershenfeld, 1992]. Even if your basis functions work in high dimensions you may not be able to collect enough data to use them.

14.6 NEURAL NETWORKS

The study of *neural networks* started as an attempt to build mathematical models that worked in the same way that brains do. While biology is so complex that such explicit connections between computation *in vivo* and *in silico* have been hard to make outside of specialized areas [Richards, 1988], the effort to do so has lead to a powerful language for using large flexible nonlinear models. The spirit of neural network or *connectionist* modeling is to use fully nonlinear functions (to handle the curse of dimensionality), and use a large number of terms (so that model mismatch errors are not a concern). Instead of matching the architecture of the model to a problem, a model is used that can describe almost anything, and careful training of the model is used to constrain it to describe the data.

The input to a typical neural network is a vector of elements $\{x_k\}$ (Figure 14.4). These are combined by a series of linear filters with weights w_{jk} to give the inputs to the *hidden units*

$$h_j = \sum_k w_{jk} x_k \quad . \tag{14.34}$$

Next comes the nonlinearity: these inputs are passed through a layer of *activation functions* $g(h_j)$ to give the output

$$X_j = g(h_j) = g(\sum_k w_{jk} x_k) \quad . \tag{14.35}$$

The activation or "squashing" functions are usually chosen to clip for large magnitudes to keep the response bounded; common choices are

$$g(h) = \frac{1}{1 + e^{-2\beta h}} \tag{14.36}$$

or

$$g(h) = \tanh(\beta h) = \frac{e^{\beta h} - e^{-\beta h}}{e^{\beta h} + e^{-\beta h}} \tag{14.37}$$

(shown in Figure 14.5). β controls the steepness; typical values are 1/2 or 1. If it is too steep the neural network becomes a binary function and loses its ability to do continuous optimizations; if it is too flat then the network is linear. It's also possible to use other nonlinear functions such as Gaussians.

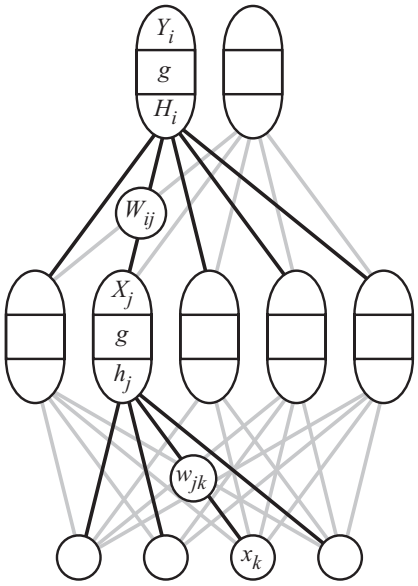

Figure 14.4. A neural network with one hidden layer.

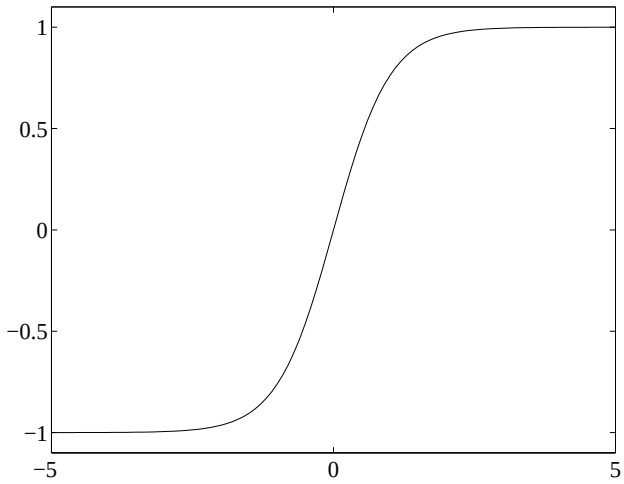

Figure 14.5. tanh function.

The outputs from the hidden units can then go through another layer of filters,

$$H_i = \sum_j W_{ij} X_j = \sum_j W_{ij}\ g\left(\sum_k w_{jk} x_k\right) \quad , \tag{14.38}$$

and be fed through another layer of squashing functions to finally produce the outputs

$$Y_i = g(H_i) = g\left[\sum_j W_{ij}\ g\left(\sum_k w_{jk} x_k\right)\right] \quad . \tag{14.39}$$

If the network is being used for regressing variables instead of classifying features, linear output functions should be used instead [Rumelhart *et al*., 1996].

This model is very general. With no hidden units such a network can classify only linearly separable problems (ones for which the possible output values can be separated by global hyperplanes). These are called *perceptrons*, and this negative result led people to assume that nonlinear networks are not useful for general tasks [Minsky & Papert, 1988]. However, it has since been shown that with one hidden layer a network can describe any continuous function (if there are enough hidden units), and that with two hidden layers it can describe any function at all [Kolmogorov, 1957; Cybenko, 1989]. For local nonlinear basis functions such as Gaussian RBFs a single layer suffices [Hartman *et al*., 1990].

14.6.1 Back Propagation

The purpose of training a neural network is to find coefficients that reduce the error between the set of outputs and given test data $y_i(\vec{x}_n)$. This is usually done by minimizing the least squares error

$$\begin{aligned}\chi^2 &= \frac{1}{2}\sum_n\sum_i [y_i(\vec{x}_n) - Y_i(\vec{x}_n)]^2 \\ &= \frac{1}{2}\sum_n\sum_i \left[y_i(\vec{x}_n) - g\left(\sum_j W_{ij}\ g\left(\sum_k w_{jk}x_{n,k}\right)\right)\right]^2 \quad . \end{aligned} \tag{14.40}$$

One way to reduce χ^2 is to use gradient descent. The update step in the output weights can be found by differentiating (η is a scale factor that controls how big the step is):

$$\begin{aligned}\Delta W_{ij} &= -\eta\frac{\partial\chi^2}{\partial W_{ij}} \\ &= \eta\sum_n [y_i(\vec{x}_n) - Y_i(\vec{x}_n)]g'(H_i)X_j \\ &\equiv \eta\sum_n \Delta_i X_j \quad , \end{aligned} \tag{14.41}$$

with the definition

$$\Delta_i = [y_i(\vec{x}_n) - Y_i(\vec{x}_n)]g'(H_i) \quad . \tag{14.42}$$

The update in the input weights can be found from the chain rule:

$$\begin{aligned}\Delta w_{jk} &= -\eta\frac{\partial\chi^2}{\partial w_{jk}} \\ &= \eta\sum_n\sum_i [y_i(\vec{x}_n) - Y_i(\vec{x}_n)]\ g'(H_i)W_{ij}\ g'(h_j)x_{n,k} \\ &= \eta\sum_n\sum_i \Delta_i W_{ij}\ g'(h_j)x_k \\ &\equiv \eta\sum_x \delta_j x_k \quad , \end{aligned} \tag{14.43}$$

defining

$$\delta_j = g'(h_j) \sum_i W_{ij} \Delta_i \quad . \tag{14.44}$$

The deltas for the input layer are found in terms of the deltas for the output layer by running them backwards through the network W_{ij}'s. This is straightforward to generalize to networks with more than one hidden layer. Training a network by gradient descent, feeding the errors backwards through the network like this, is called *back propagation*. In the last chapter we saw that gradient descent slows down near minima, so the Levenberg–Marquardt method can be used to improve the convergence. The search can easily get stuck in local minima; some easy improvements are to add momentum so that it rolls out of small wells, or add some random fluctuations to help kick it out of minima. These ideas will be discussed in the next chapter, as well as why such a simple-minded search in the high-dimensional space of weights works so surprisingly well.

We have been discussing *feed-forward* networks, in which the inputs control the outputs. Just as a finite impulse response filter can be generalized to an infinite impulse response filter by letting the outputs influence the inputs, in a *recurrent network* the outputs are fed back to the inputs. This is a good thing to do if the net should learn to model such feedback behavior. Much of the art in using neural networks lies in choosing an architecture that reflects the structure in a problem. For example, for *multistationary* data that switch among different regimes it's advantageous to train a collection of networks that are forced to specialize in particular regimes [Weigend *et al*., 1995].

14.7 REGULARIZATION

In neural networks, model complexity was originally controlled by *early stopping*. This uses a large network, and continues training it through back propagation until the out-of-sample predictions degrade. In effect, degrees of freedom are introduced by the training. Rather than match the form of the model to the data *a priori*, an enormous model is used that can describe anything, and it is constrained by the training. In fact, it is common for neural networks to have more parameters than the size of the training data set! Unlike a polynomial fit, the network can choose from a much larger space of possible models. The order of a polynomial restricts how many variables can interact; in a neural network all of the variables can always interact (although some of the connections will decay during training).

Early stopping had some early successes, but does not provide a way to guide the training with advance knowledge. A more general framework for constraining flexible models is provided by the powerful concept of *regularization* [Weigend *et al*., 1990; Girosi *et al*., 1995]. This is the name for the knob that any good algorithm should have that controls a trade-off between matching the data and enforcing your prior beliefs. As we saw in Section 12.1, maximum *a posteriori* estimation requires finding

$$\max_{\text{model}} p(\text{model}|\text{data}) = \max_{\text{model}} \frac{p(\text{data}|\text{model})\, p(\text{model})}{p(\text{data})} \quad , \tag{14.45}$$

or taking logarithms

$$\max_{\text{model}} \log p(\text{model}|\text{data}) = \log p(\text{data}|\text{model}) + \log p(\text{model}) - \log p(\text{data}) \quad . \tag{14.46}$$

The first term on the right hand side measures how well the model matches the data. This could be a Gaussian error model, or it might be defined to be the profit in a stock trading model. The second term, called a *prior*, expresses advance beliefs about what kind of model is reasonable. The final term comes in only if training is done over multiple data sets. Taken together, these components define a *cost function* or *penalty function* to be searched to find the best model (using methods to be covered in the next chapter).

If there truly is no prior knowledge at all, then setting $p(\text{model}) = 1$ reduces to maximum likelihood estimation. But that's rarely the case, and even simple priors have great value. Although priors can be formally expressed as a probability distribution, they're usually just added to the cost function with a Lagrange multiplier. A common prior is the belief that the model should be locally linear unless the data forces it to do otherwise; this is measured by the integral of the square of the curvature of the model. If a least squares (Gaussian) error model is used then the quantity to be minimzed for a 1D problem is

$$I = \sum_{n=1}^{N} [y_n - f(x_n, \vec{a})]^2 + \lambda \int \left[\frac{d^2 f}{dx^2}\right]^2 dx \quad , \tag{14.47}$$

where $\vec{a}$ is the set of coefficients to be determined and the bounds of integration are over the interval being fit. The left hand term gives the agreement between the data and the model, and the term on the right depends on the model alone. Making the sum of both terms extremal by solving $\partial I/\vec{a} = 0$ (Chapter 5) for a particular value of λ gives a set of equations to be solved to find the corresponding $\vec{a}$. The best value for λ can then be found iteratively by a procedure such as cross-validation. Problem 13.2 gives an example of the use of this regularizer.

Other common regularizers include the entropy of a distribution, expressing the belief that the distribution should be flat, and the total magnitude of a model's coefficients, seeking the most parsimonious model. Regularization plays a central role in the theory of *inverse problems* [Parker, 1977; Engl *et al*., 1996], the essential task that arises throughout science and engineering of deducing the state of a system from a set of measurements that underdetermine it.

Flexible models such as neural networks do have an impressive ability to find a useful model from a huge space of possibilities, but they aren't magic, and they don't replace the need to think [Weigend & Gershenfeld, 1993]. Clever training alone is not a cure for bad data. In practical applications much of the progress comes from collecting good training data sets, finding appropriate features to use for the model inputs and outputs, and imposing domain-specific constraints on the model to improve its ability to generalize from limited observations. Salvation lies in these details. Given unlimited data and unlimited resolution almost any reasonable model should work with data in whatever form you present it, but in the real world of imperfect measurements, insight into model architectures must be enhaced by a combination of advance analysis of a system and experimentation with the data.

14.8 SELECTED REFERENCES

[Hertz *et al*., 1991] Hertz, John A., Krogh, Anders S., & Palmer, Richard G. (1991). *Introduction to the Theory of Neural Computation.* Redwood City, CA: Addison-Wesley.

[Bishop, 1996] Bishop, Christopher M. (1996). *Neural Networks for Pattern Recognition.* Oxford: Oxford University Press.

Excellent introductions to neural networks.

14.9 PROBLEMS

(13.1) Find the first five *diagonal* Padé approximants $[1/1], \ldots, [5/5]$ to e^x around the origin. Remember that the numerator and denominator can be multiplied by a constant to make the numbers as convenient as possible. Evaluate the approximations at $x = 1$ and compare with the correct value of $e = 2.718281828459045$. How is the error improving with the order?

(13.2) Take as a data set $x = \{-10, -9, \ldots, 9, 10\}$, and $y(x) = 0$ if $x \leq 0$ and $y(x) = 1$ if $x > 0$.

- (*a*) Fit the data with a polynomial with 5, 10, and 15 terms, using a pseudo-inverse of the Vandermonde matrix (such as Matlab's *pinv* function).
- (*b*) Fit the data with 5, 10, and 15 r^3 RBFs uniformly distributed between $x = -10$ and $x = 10$.
- (*c*) Using the coefficients found for these six fits, evaluate the total out-of-sample error at $x = \{-10.5, -9.5, \ldots, 9.5, 10.5\}$.
- (*d*) Using a 10th-order polynomial, fit the data with the curvature regularizer in equation (14.47), and plot the fits for $\lambda = 0, 0.01, 0.1, 1$ (this part is harder than the others).