5 Automatic feature design for regression

As discussed in the end of Section 3.2, rarely can we design perfect or even strongly performing features for the general regression problem by completely relying on our understanding of a given dataset. In this chapter we describe tools for automatically designing proper features for the general regression problem, without the explicit incorporation of human knowledge gained from e.g., visualization of the data, philosophical reflection, or domain expertise.

We begin by introducing the tools used to perform regression in the ideal but extremely unrealistic scenario where we have complete and noiseless access to all possible input feature/output pairs of a regression phenomenon, i.e., a continuous function (as first discussed in Section 3.2). Here we will see how, in the case where we have such unfettered access to regression data, perfect features can be designed automatically by combining elements from a set of basic feature transformations. We then see how this process for building features translates, albeit imperfectly, to the general instance of regression where we have access to only noisy samples of a regression relationship. Following this we describe *cross-validation*, a crucial procedure to employing automatic feature design in practice. Finally we discuss several issues pertaining to the best choice of primary features for automatic feature design in practice.

5.1 Automatic feature design for the ideal regression scenario

In Fig. 5.1 we illustrate a prototypical dataset on which we perform regression, where our input feature and output have some sort of clear nonlinear relationship. Recall from Section 3.2 that at the heart of feature design for regression is the tacit assumption that the data we receive are in fact noisy samples of some underlying continuous function (shown in dashed black in Fig. 5.1). Our goal in solving the general regression problem is then, using the data at our disposal (which we may think of as noisy glimpses of the underlying function), to approximate this data-generating function as well as we can.

In this section we will assume the impossible: that we have complete access to a clean version of every input feature/output pair of a regression phenomenon, or in other words that our data completely traces out a continuous function $y(\mathbf{x})$. We do not assume that we know a functional form for $y(\mathbf{x})$, but in such an ideal scenario we will see how perfect features may be designed automatically to fit such data (regardless of the complexity or ambient dimension of $y(\mathbf{x})$) by combining different elements from a basis of primary features.

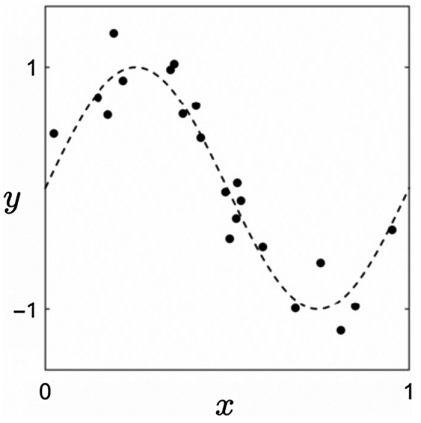

Fig. 5.1 A realistic dataset for regression made by taking noisy samples from the data generating function $y(x) = \sin(2\pi x)$ (shown in dashed black) over the unit interval.

The entire mathematical framework for the automatic design of features in this perfect regression scenario comes from the classic study of *continuous function approximation*,[1] which has been developed by mathematicians, physical scientists, and engineers over the past several centuries. Because of this we will use phrases like "function approximation" and "automatic feature design" synonymously in the description that follows.

5.1.1 Vector approximation

Recall the linear algebra fact that any vector $\mathbf{y}$ in $\mathbb{R}^P$, that is the set of all column vectors of length P with real entries, can be represented perfectly over any given basis of P linearly independent vectors. In other words, given a set of P linearly independent vectors $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$ in $\mathbb{R}^P$, we can always express $\mathbf{y}$ precisely (i.e., without any error) as a linear combination of its elements,

$$\sum_{p=1}^{P}\mathbf{x}_p w_p = \mathbf{y}. \tag{5.1}$$

Now let us suppose that we only have access to a subset $\{\mathbf{x}_m\}_{m=1}^{M}$ of the full basis $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$ in order to represent $\mathbf{y}$, where $M \leq P$. Although in this case there is no guarantee that the vector $\mathbf{y}$ lies completely in the span of the partial basis $\{\mathbf{x}_m\}_{m=1}^{M}$, we can still approximate $\mathbf{y}$ via a linear combination of its elements,

$$\sum_{m=1}^{M}\mathbf{x}_m w_m \approx \mathbf{y}. \tag{5.2}$$

[1] Throughout the remainder of this section we will be fairly loose in our discussion of function approximation, which is by nature a highly technical subject. The interested reader can see Section 5.7 for a short discussion and a list of more technical treatments of the subject.

Note that the approximation in (5.2) can be made to hold to any desired level of tolerance by making M larger.

The ideal set of weights $\{w_m\}_{m=1}^{M}$ to make the *partial basis approximation* in (5.2) hold as well as possible can then be determined by solving the related Least Squares problem:

$$\underset{w_1 \ldots w_M}{\text{minimize}} \left\| \sum_{m=1}^{M} \mathbf{x}_m w_m - \mathbf{y} \right\|_2^2, \tag{5.3}$$

which has a closed form solution as detailed in Section 3.1.3.

5.1.2 From vectors to continuous functions

Any vector $\mathbf{y}$ in $\mathbb{R}^P$ can be viewed as a "discrete function" on the unit interval [0, 1] after plotting its entries at equidistant points $\left\{x_p = p/P\right\}_{p=1}^{P}$ on the x-axis, as pairs $\left\{\left(x_p, y_p\right)\right\}_{p=1}^{P}$. We illustrate this idea in the top left panel of Fig. 5.2 using a $P = 4$ dimensional vector $\mathbf{y}$ defined entry-wise as $y_p = \sin\left(2\pi x_p\right)$, where $x_p = p/P$ and $p = 1 \ldots P$. Also shown in the top row of this figure is the vector $\mathbf{y}$, constructed in precisely the same manner, only this time with $P = 40$ (middle panel) and $P = 400$ (right panel). As can be seen in this figure, for larger values of P the collection of points $\left(x_p, y_p\right)$ closely resembles the continuous function $y(x) = \sin(2\pi x)$.

In other words, as $P \longrightarrow \infty$ the set of points $\left\{\left(x_p, y_p\right) = \left(p/P, \sin\left(2\pi x_p\right)\right)\right\}_{p=1}^{P}$ for all intents and purposes, precisely describes the continuous function $y(x) = \sin(2\pi x)$. Hence we can think of a continuous function (defined on the interval [0, 1]) as, *roughly*, an infinite dimensional vector.

This same intuition applies to functions $y(x)$ defined over an arbitrary interval $[a, b]$ as well, since we can make the same argument given above when $a = 0$ and $b = 1$ and approximate y as finely as desired using a discrete set of sampled points. Furthermore, we can employ a natural extension of this argument to say the same thing about general functions $y(\mathbf{x})$ where $\mathbf{x}$ is an N-dimensional vector defined over a hyper-rectangle, that is where each entry of $\mathbf{x}$ lies in some interval $x_n \in [a_n, b_n]$. We do this by evaluating y over a finer and finer grid of evenly spaced points $\mathbf{x}_p$ covering the hyper-rectangle of its input domain (illustrated with a particular example for two dimensional input in the bottom row of Fig. 5.2). Therefore in general we can roughly think about any continuous function $y(\mathbf{x})$, with bounded input $\mathbf{x}$ of length N, as an infinite length vector.

This perspective on continuous functions is especially helpful in framing the notion of function approximation, since the key concepts broadly follow the same shape as (finite length) vector approximation described in Section 5.1.1. As we discuss next, many of the defining ideas with vector approximation in $\mathbb{R}^P$, e.g., the notions of bases and Least Squares weight fitting, have direct analogs in the case of continuous function approximation.

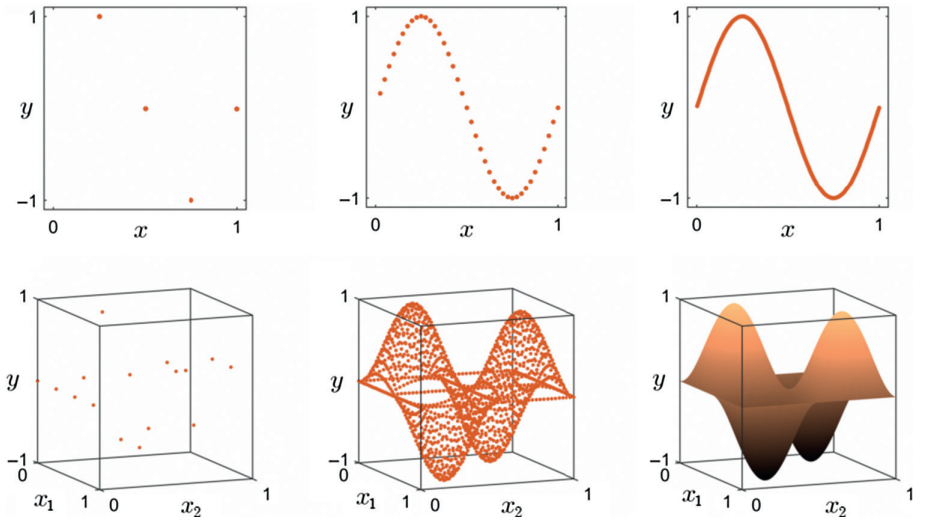

Fig. 5.2 (top row) The P-dimensional vector $\mathbf{y}$ with entries $y_p = \sin\left(2\pi x_p\right)$ where $x_p = p/P$ and $p = 1 \ldots P$, plotted as points $\left(x_p,\, y_p\right)$ with (left) $P = 4$, (middle) $P = 40$, and (right) $P = 400$. The vector $\mathbf{y}$, as a discrete function, closely resembles the continuous function $y\,(x) = \sin\left(2\pi x\right)$, especially for larger values of P. (bottom row) An analogous example in three dimensions using the function $y\,(\mathbf{x}) = \sin\left(2\pi x_1\right)\sin\left(2\pi x_2\right)$ evaluated over a grid of (left) $P = 16$, (middle) $P = 1600$, and (right) $P = 160\,000$ evenly spaced points over the unit square. Note that the number of samples P required to maintain a certain resolution of the function grows exponentially with the input dimension. This unwanted phenomenon is often called "the curse of dimensionality."

5.1.3 Continuous function approximation

Like a discrete vector $\mathbf{y}$ in $\mathbb{R}^P$, any continuous function $y\,(\mathbf{x})$ with bounded N-dimensional input $\mathbf{x}$ can be completely decomposed over a variety of bases. For clarity and convenience we will suppose $\mathbf{x}$ lies in the N-dimensional unit hypercube $[0,\, 1]^N$ for the remainder of this chapter, that is each entry $x_n \in [0,\, 1]$ (however the discussion here holds for y with more general input as well). In any case, as with finite length vectors, we may write such a function $y\,(\mathbf{x})$ as a linear combination of basis elements which are themselves continuous functions. In terms of regression, the elements of a basis are basic features that we may combine in order to perfectly (or near-perfectly) approximate our continuous function input/output data $(\mathbf{x},\, y\,(\mathbf{x}))$, for which we have for all $\mathbf{x} \in [0,\, 1]^N$.

As can be intuited by the rough description given previously of such a continuous function as an "infinite length vector," we must correspondingly use an infinite number of basis elements to represent any such desired function completely. Formally a basis in this instance is a set of basic feature transformations $\{f_m\,(\mathbf{x})\}_{m=1}^{\infty}$ such that at all points $\mathbf{x}$ in the unit hypercube we can express $y\,(\mathbf{x})$ perfectly as

$$\sum_{m=0}^{\infty} f_m\,(\mathbf{x})\, w_m = y\,(\mathbf{x})\,. \tag{5.4}$$

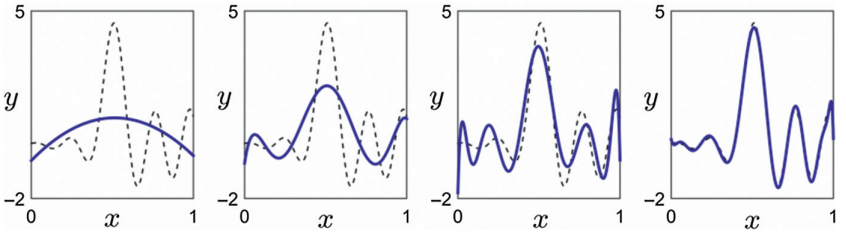

Fig. 5.3 Partial basis approximation (in blue) of an example function $y(x)$ (in dashed black), where (from left to right) $M = 2, 6, 10$, and 20, respectively. As the number of basis features is increased the approximation more closely resembles the underlying function.

Here for each m the weight w_m must be tuned properly for a given y so that the above equality will indeed hold. Take a moment to appreciate just how similar in both concept and shape this is to the analogous vector formula given in (5.1) which describes the decomposition of a (finite length) vector over a corresponding vector basis. Structurally, the set of vectors $\mathbb{R}^P$ and the set of continuous functions defined on the N-dimensional unit hypercube $[0, 1]^N$, which we shall denote by $\mathcal{C}^N$, have much in common.

As with vectors, for large enough M we can approximate y over its input domain as

$$\sum_{m=0}^{M} f_m(\mathbf{x})\, w_m \approx y(\mathbf{x}). \tag{5.5}$$

Again, this approximation can be made as finely as desired[2] by increasing the number of basis features M used and tuning the associated parameters appropriately (as detailed in Section 5.1.5). In other words, by increasing M we can design automatically (near) perfect features to represent $y(\mathbf{x})$. This is illustrated with a particular example in Fig. 5.3, where an increasing number of basis elements (in this instance polynomials) are used to approximate a given function.[3]

5.1.4 Common bases for continuous function approximation

Bases for continuous function approximation, sometimes referred to as *universal approximators*, can be distinguished by those whose elements are functions of the input $\mathbf{x}$ alone, and those whose elements are also functions of further internal parameters. The former variety, referred to as *fixed bases* due to their fixed shape and sole dependence on $\mathbf{x}$, include the polynomial and sinusoidal feature bases. For $N = 1$ elements of the polynomial basis consist of a constant term $f_0(x) = 1$ and the set of simple monomial features of the form

[2] Depending on both the function and the employed basis, the approximation in (5.5) may not improve at each and every point $\mathbf{x} \in [0, 1]^N$ as we increase the number of basis elements. However, this technicality does not concern us as the approximation can be shown to generally improve over the entire domain of the function by increasing M, which is sufficient for our purposes.

[3] The function approximated here is defined as $y(x) = e^{3x} \frac{\sin\left(3\pi^2(x-0.5)\right)}{3\pi^2(x-0.5)}$.

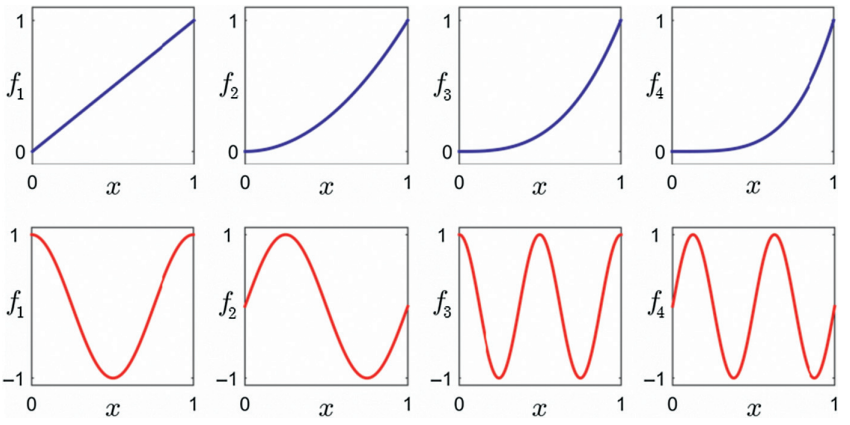

Fig. 5.4 (top row, from left to right) The first four non-constant elements of the polynomial basis. (bottom row, from left to right) The first four non-constant elements of the Fourier basis.

$$f_m(x) = x^m \quad \text{for all } m \geq 1. \tag{5.6}$$

One often encounters the polynomial basis when learning calculus, where it is employed in the form of a Taylor series approximation of a (many times) differentiable function $y(x)$.

Likewise for $N = 1$ the sinusoidal or *Fourier* basis, so named after its inventor Joseph Fourier who first used these functions in the early 1800s to study heat diffusion, consists of the constant term $f_0(x) = 1$ and the set of cosine and sine waves with ever increasing frequency of the form[4]

$$\begin{cases} f_{2m-1}(x) = \cos(2\pi mx) & \text{for all } m \geq 1 \\ f_{2m}(x) = \sin(2\pi mx) & \text{for all } m \geq 1. \end{cases} \tag{5.7}$$

The first four non-constant elements of both the polynomial and Fourier bases for $N = 1$ are shown in Fig. 5.4. With slightly more cumbersome notation both the polynomial and Fourier bases are likewise defined[5] for general N-dimensional input $\mathbf{x}$.

The second class of bases we refer to as *adjustable*, due to the fact that their elements have tunable internal parameters, and are more straightforward to define for general N-dimensional input. The simplest adjustable basis is what is commonly referred to as a *single hidden layer feed forward neural network*, coined by neuroscientists in the late 1940s who first created this sort of basis as a way to roughly model how the human brain processes information. With the exception of the constant term $f_0(\mathbf{x}) = 1$, this basis uses

[4] It is also common to write the Fourier basis approximation using classic *complex exponential* definitions of both cosine and sine, i.e., $\cos(\alpha) = \frac{1}{2}\left(e^{i\alpha} + e^{-i\alpha}\right)$ and $\sin(\alpha) = \frac{1}{2i}\left(e^{i\alpha} - e^{-i\alpha}\right)$. With these complex exponential definitions, one can show (see Exercise 5.5) that with a scalar input x, Fourier basis elements can be written in complex exponential form $f_m(x) = e^{2\pi imx}$.

[5] For a general N-dimensional input each polynomial feature takes the analogous form $f_m(\mathbf{x}) = x_1^{m_1} x_2^{m_2} \cdots x_N^{m_N}$. Likewise, using the complex exponential notation (see previous footnote) each multidimensional Fourier basis element takes the form $f_m(\mathbf{x}) = e^{2\pi im_1x_1} e^{2\pi im_2x_2} \cdots e^{2\pi im_Nx_N}$.

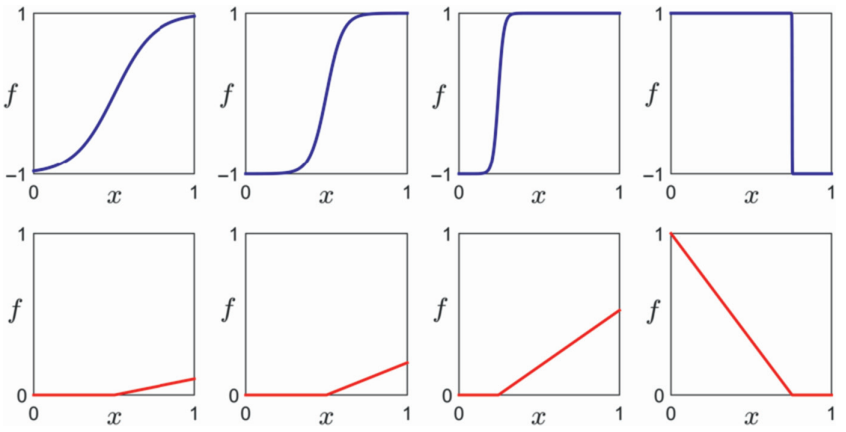

Fig. 5.5 Unlike a fixed basis, elements of an adjustable basis are free to change form by adjusting internal parameters. Here we show four instances of a single (top row) hyperbolic tangent and (bottom row) hinge basis function, with each instance corresponding to a different setting of its internal parameters.

a single type of parameterized function (often referred to as an *activation function*) for each basis feature. Common examples include the hyperbolic tangent function

$$f_m(\mathbf{x}) = \tanh\left(c_m + \mathbf{x}^T\mathbf{v}_m\right) \quad \text{for all } m \geq 1, \tag{5.8}$$

and the max or hinge function (also referred to as a "rectified linear unit" in this context)

$$f_m(\mathbf{x}) = \max\left(0,\, c_m + \mathbf{x}^T\mathbf{v}_m\right) \quad \text{for all } m \geq 1. \tag{5.9}$$

Note that the scalar parameter c_m as well as the N-dimensional vector parameter $\mathbf{v}_m$ are unique to each basis element f_m and are adjustable, giving each basis element a range of possible shapes to take depending on how the parameters are set.

For example, illustrated in Fig. 5.5 are four instances[6] of basis features in Equations (5.8) and (5.9) with scalar input (i.e., $N = 1$), where the $\tanh(\cdot)$ and $\max(0, \cdot)$ elements are shown in the top and bottom rows, respectively. For each type of basis the four panels show the form taken by a single basis element with four different settings for c_m and v_m.

Generally speaking the flexibility of each basis feature, gained by introduction of adjustable internal parameters, typically enables effective approximation using fewer neural network basis elements than a standard fixed basis. For example, in Fig. 5.6 we show the results of using a polynomial, Fourier, and single hidden layer neural network with $\tanh(\cdot)$ activation function respectively to approximate a particular function[7] $y(x)$ over $x \in [0, 1]$. In each row from left to right we use $M = 2$ and $M = 6$ basis elements of each type to approximate the function, and as expected the approximations improve for all basis types as we increase M. However, comparing the evolution of approximations

[6] Note that unlike the fixed basis functions shown in Fig. 5.4, which can be arranged and counted in a sequence of low to high "degree" elements, there is no such ordering within a set of adjustable basis functions.

[7] The function approximated here is $y(x) = e^x \cos(2\pi \sin(\pi x))$.

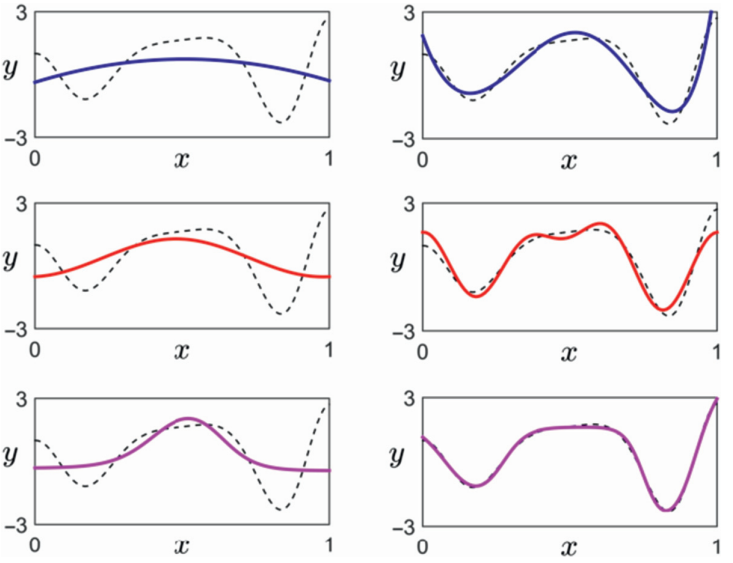

Fig. 5.6 From left to right, approximation of a continuous function (shown by the dashed black curve) over [0, 1], using $M = 2$ and $M = 6$ elements of (top row) polynomial, (middle row) Fourier, and (bottom row) single hidden layer neural network bases, respectively. While all three bases could approximate this function as finely as desired by increasing M, the neural network basis (with its adjustable internal parameters) approximates the underlying function more closely using the same number of basis elements compared to both fixed bases.

in each row one can see that the neural network basis (with the added flexibility of its internal parameters) better approximates y than either fixed bases using the same number of basis elements.

Even more flexible adjustable basis features are commonly constructed via *summation* and *composition* of activation functions.[8] For instance, to create a single basis element of a feed forward neural network with *two hidden layers* we take a weighted sum of single hidden layer basis features and pass the result through an activation function (of the same kind as used in the single layer basis). Doing this with the tanh basis in Equation (5.8) gives

$$f_m(\mathbf{x}) = \tanh\left(c_m^{(1)} + \sum_{m_2=1}^{M_2} \tanh\left(c_{m_2}^{(2)} + \mathbf{x}^T \mathbf{v}_{m_2}^{(2)}\right) v_{m_2,m}^{(1)}\right), \tag{5.10}$$

for all $m \geq 1$. Note that here for organizational purposes we have used superscripts on each internal parameter to indicate the layer it belongs to. Likewise, we can compose the hinge function with itself to create a two hidden layer basis function of the form

[8] Although one could think of designing more flexible basis elements than those of a single layer neural network in Equation (5.8) in a variety of ways, this is the most common approach (i.e., summation and composition using a single type of activation function).

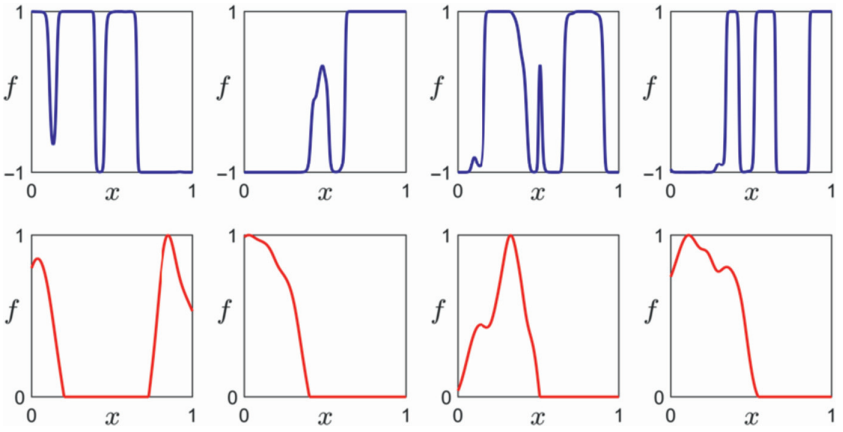

Fig. 5.7 Four instances of a two layer neural network basis function made by composing (top row) hyperbolic tangent and (bottom row) hinge functions. In each instance the internal parameters are set randomly. Note how the two layer basis elements are far more diverse in shape than those of a single layer basis shown in Fig. 5.5.

$$f_m(\mathbf{x}) = \max\left(0,\ c_m^{(1)} + \sum_{m_2=1}^{M_2} \max\left(0,\ c_{m_2}^{(2)} + \mathbf{x}^T\mathbf{v}_{m_2}^{(2)}\right) v_{m_2,m}^{(1)}\right), \tag{5.11}$$

for all $m \geq 1$. Note that with a two layer neural network basis, we have increased the number of internal parameters which are nonlinearly related layer by layer. Specifically, in addition to the first layer bias parameter $c_m^{(1)}$ and $M_2 \times 1$ weight vector $\mathbf{v}_m^{(1)}$, we also have in the second layer M_2 bias parameters and M_2 weight vectors each of length N.

In Fig. 5.7 we show four instances for each of the two basis function types[9] in Equations (5.10) and (5.11), where $N = 1$ and $M_2 = 1000$. Note that the two layer basis functions in Fig. 5.7 are clearly more diverse in shape than the single layer basis functions shown in Fig. 5.5.

To achieve even greater flexibility for individual basis features we can create a *neural network with three hidden layers* (or more) by simply repeating the procedure used to create the two layer basis from the single layer version. That is, we take a weighted sum of two hidden layer basis features and pass the result through an activation function (of the same kind as used in the two layer basis). For example, performing this procedure for the two layer hinge basis function in Equation (5.11) gives a three layer network basis function of the form

$$f_m(\mathbf{x}) = \max\left(0,\ c_m^{(1)} + \sum_{m_2=1}^{M_2} \max\left(0,\ c_{m_2}^{(2)} + \sum_{m_3=1}^{M_3} \max\left(0,\ c_{m_3}^{(3)} + \mathbf{x}^T\mathbf{v}_{m_3}^{(3)}\right) v_{m_3,m_2}^{(2)}\right) v_{m_2,m}^{(1)}\right), \tag{5.12}$$

[9] Although not a common choice, one can mix and match different activation functions for different layers, as in $f_m(\mathbf{x}) = \tanh\left(c_m^{(1)} + \sum_{m_2=1}^{M_2} \max\left(0,\ c_{m_2}^{(2)} + \mathbf{x}^T\mathbf{v}_{m_2}^{(2)}\right) v_{m_2,m}^{(1)}\right)$.

for all $m \geq 1$. This procedure can be repeated to produce a neural network basis with an arbitrary number of hidden layers. Currently the convention is to refer to a neural network basis with three or more hidden layers as a *deep network* [6–10].

5.1.5 Recovering weights

As with vector approximation, a common way to tune the weights $\{w_m\}_{m=0}^{M}$, as well as the possible internal parameters of the basis features themselves when neural networks are used, is to minimize a Least Squares cost function. In this instance we seek to minimize the difference between y and its partial basis approximation in Equation (5.5), over all points in the unit hypercube denoted by $[0, 1]^N$. Stated formally, the minimization of this Least Squares cost is written as

$$\underset{w_0,\ldots,w_M,\Theta}{\text{minimize}} \int_{\mathbf{x}\in[0,1]^N} \left(\sum_{m=0}^{M} f_m(\mathbf{x})\, w_m - y(\mathbf{x}) \right)^2 d\mathbf{x}, \tag{5.13}$$

where the set Θ contains possible parameters of the basis elements themselves, which is empty if a fixed basis is used. Note that Equations (5.3) and (5.13) are entirely analogous Least Squares problems for learning the weights associated to a partial basis approximation, the former stated for vectors in $\mathbb{R}^P$ and the latter for continuous functions in $\mathcal{C}^N$.

Unlike its vector counterpart, however, the Least Squares problem in (5.13) cannot typically[10] be solved in closed form due to intractability of the integrals involved. Instead, one typically solves an approximate form of the problem where each function is first discretized, as will be described in Section 5.2.1. This makes a tractable problem which, as we will soon see, naturally leads to a model for the general problem of regression.

5.1.6 Graphical representation of a neural network

It is common to represent the weighted sum of M neural network basis features,

$$r = b + \sum_{m=1}^{M} f_m(\mathbf{x})\, w_m, \tag{5.14}$$

graphically to visualize the compositional structure of each element. Here $f_m(\mathbf{x})$ can be a general multilayer neural network feature as described previously.

As a simple example, in Fig. 5.8 we represent graphically the mathematical expression $a(x_1 v_1 + x_2 v_2 + x_3 v_3)$ where $a(\cdot)$ is any activation function. This graphical representation consists of: 1) weighted edges that represent the individual multiplications (i.e., of x_1 by v_1, x_2 by v_2, and x_3 by v_3); 2) a summation unit representing the sum $x_1 v_1 + x_2 v_2 + x_3 v_3$ (shown as a small hollow circle); and finally 3) an activation unit representing the sum evaluated by the activation function (shown as a larger blue circle).

[10] While finding weights associated to a fixed basis is simple in theory (see chapter exercises), discretization is still usually required.

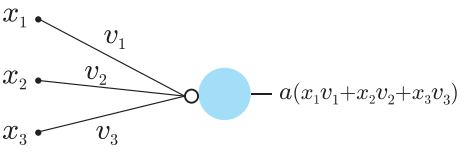

Fig. 5.8 Representation of a single activation function with $N = 3$-dimensional input. Each input is connected to the summation unit via a weighted edge. The summation unit (shown by a small hollow circle) takes in the weighted inputs, and outputs their sum to the activation unit (shown by a large blue circle). See text for further details.

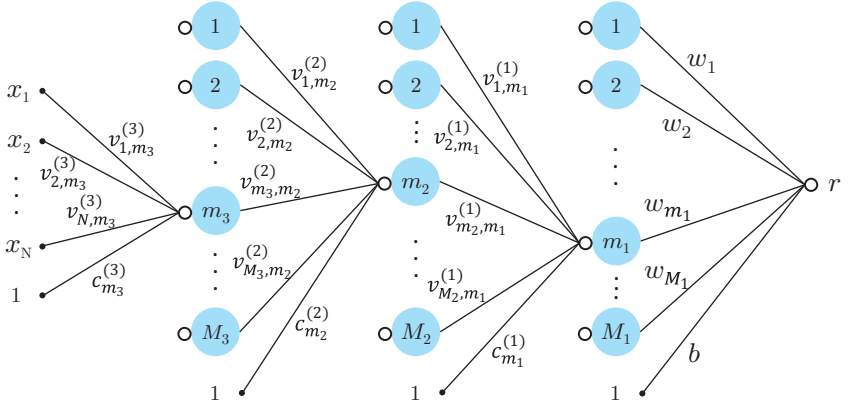

Fig. 5.9 A three hidden layer neural network. Note that for visualization purposes, we just show the edges connecting the units in one layer to only one of the units in the previous layer. See text for further details.

By cascading this manner of representing the input/output relationship of a single activation function we may similarly represent sums of multilayer neural network basis features. For example, in Fig. 5.9 we show a graphical representation of (5.14) with $M = M_1$ three hidden layer network basis features taking in general N dimensional input (like the one shown in Equation (5.12)). As in Fig. 5.8, the input $\mathbf{x}$ is shown entry-wise on the left and the output r on the right. In between are each of the three hidden layers, from right to left each consisting of M_1, M_2, and M_3 hidden units respectively. Some texts number these hidden layers in ascending order from left to right. While this slightly changes the notation we use in this book, all results and conclusions remain the same.

5.2 Automatic feature design for the real regression scenario

Here we describe how fixed and adjustable bases of features, introduced in the previous section, are applied to the automatic design of features in the real regression scenario. Although they lose their power as perfect feature design tools (which they had in the case of the ideal regression scenario), strong features can often be built by combining elements of bases for real instances of regression.

5.2.1 Approximation of discretized continuous functions

The Least Squares problem in Equation (5.13), for tuning the weights of a sum of basis features in the ideal regression scenario discussed in Section 5.1 is highly intractable. However, by finely discretizing all of the continuous functions involved we can employ standard optimization tools to solve a discrete version of the problem. Recall from Section 5.1.2 that given any function y defined over the unit hypercube, we may sample with a finely spaced grid over unit hypercube $[0, 1]^N$ a potentially large (but finite) number of P points so that the collection of pairs $\left\{\left(\mathbf{x}_p, y\left(\mathbf{x}_p\right)\right)\right\}_{p=1}^{P}$ resembles the function $y\left(\mathbf{x}\right)$ as well as desired. Using such a discretization scheme we can closely approximate the function $y\left(\mathbf{x}\right)$, as well as the constant basis term $f_0\left(\mathbf{x}\right) = 1$ and any set of M non-constant basis features $f_m\left(\mathbf{x}\right)$ for $m = 1 \ldots M$, so that a discretized form of Equation (5.5) holds at each $\mathbf{x}_p$, as

$$\sum_{m=0}^{M} f_m\left(\mathbf{x}_p\right) w_m \approx y\left(\mathbf{x}_p\right). \tag{5.15}$$

Denoting by $b = w_0$ the weight on the constant basis element, $y_p = y\left(\mathbf{x}_p\right)$, as well as the compact *feature vector* notation $\mathbf{f}_p = \begin{bmatrix} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots & f_M\left(\mathbf{x}_p\right) \end{bmatrix}^T$ and weight vector $\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \cdots & w_M \end{bmatrix}^T$, we may write Equation (5.15) more conveniently as

$$b + \mathbf{f}_p^T \mathbf{w} \approx y_p. \tag{5.16}$$

In order for this approximation to hold we can then consider minimizing the squared difference between both sides over all P, giving the associated Least Squares problem

$$\underset{b, \mathbf{w}, \Theta}{\text{minimize}} \sum_{p=1}^{P} \left(b + \mathbf{f}_p^T \mathbf{w} - y_p\right)^2. \tag{5.17}$$

Note that once again we denote by Θ the set of internal parameters of all basis features, which is empty in the case of fixed bases. This is precisely a discretized form of the continuous Least Squares problem shown originally in Equation (5.13). Also note that this is only a slight generalization of the Least Squares problem for regression discussed throughout Chapter 3.

We illustrate the idea of approximating a continuous function from a discretized version of it, using a particular example in Fig. 5.10, where the continuous function $y\left(x\right) = \sin\left(2\pi x\right)$ is shown in the left panel along with its discretized version in the middle panel. Employing a polynomial basis, and using weights provided by solving the Least Squares problem in Equation (5.17), we have an excellent approximation of the true function in the right panel.

5.2.2 The real regression scenario

The approximation of a finely discretized continuous function provides an almost ideal scenario for regression where the data, the sampled points, gives a clear (noiseless)

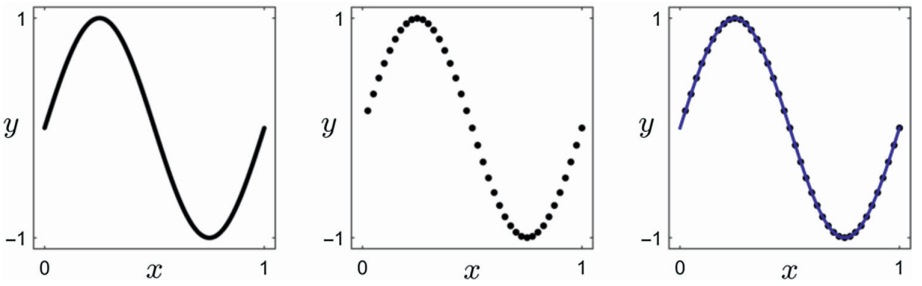

Fig. 5.10 (left panel) The continuous function $y(x) = \sin(2\pi x)$ defined on the unit interval and (middle panel) its discretized version made by evaluating $y(x)$ over a sequence of $P = 40$ evenly spaced points on the horizontal axis. (right panel) Fitting a degree $M = 12$ polynomial curve (in blue) to the discretized function via solving (5.17) provides an excellent continuous approximation to the original function.

picture of the underlying function over its entire domain. However, rarely in practice do we have access to such large quantities of noiseless data which span the entire input space of a phenomenon. Conversely, a dataset seen in practice may consist of only a small number of samples, these samples may not be distributed evenly in space, and they may be corrupted by measurement error or some other sort of "noise." Indeed most datasets for regression are akin to noisy samples of some unknown continuous function making the machine learning task of regression, in general, a function approximation problem based only on noisy samples of the underlying function.

> The general instance of regression is a function approximation problem based on noisy samples of the underlying function.

To illustrate this idea, in the right panel of Fig. 5.11 we show a simulated example of a realistic regression dataset consisting of $P = 21$ data points $\left\{\left(x_p, y_p\right)\right\}_{p=1}^{P}$. This dataset is made by taking samples of the function $y(x) = \sin(2\pi x)$, where each input x_p is chosen randomly on the interval $[0, 1]$ and evaluated by the function with the addition of noise ϵ_p as $y_p = y\left(x_p\right) + \epsilon_p$. Here this noise simulates common small errors made, for instance, in the collection of regression data. Also shown in this figure for comparison is the ideal dataset previously shown in the middle panel of Fig. 5.10.

With a realistic regression dataset, we use the same learning framework to find optimal parameters as we did with ideal discretized data, i.e., by solving the discrete Least Squares problem in (5.17). Note again that depending on the basis type (fixed or adjustable) used in (5.17), the design of the feature vector $\mathbf{f}_p$ changes.

In Fig. 5.12 we show various fits to the toy sinusoidal dataset in the right panel of Fig. 5.11, using a polynomial (where $M = 3$), Fourier (where $M = 1$), and single hidden layer neural network basis with $\tanh(\cdot)$ activation function (where $M = 4$). Details on

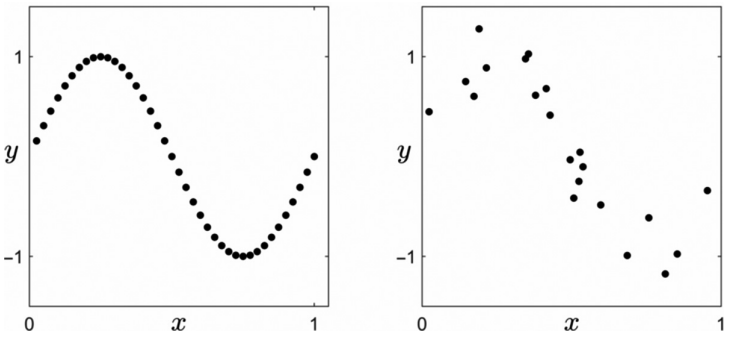

Fig. 5.11 (left panel) An ideal dataset for regression made by finely and evenly sampling the continuous function $y(x) = \sin(2\pi x)$ over the unit interval. (right panel) A more realistic simulated dataset for regression made by evaluating the same function at a smaller number of random input points and corrupting the result by adding noise to simulate e.g., errors made in data collection.

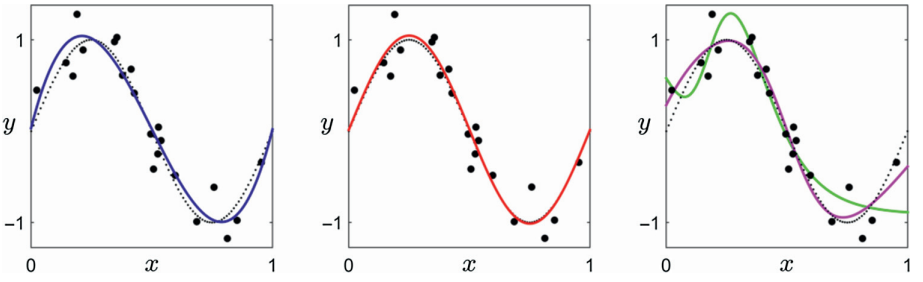

Fig. 5.12 A comparison of (left panel) polynomial, (middle panel) Fourier, and (right panel) single layer neural network fits to the regression dataset shown in the right panel of Fig. 5.11. The optimal weights in each case are found by minimizing the Least Squares cost function as described in Examples 5.1 (for polynomial and Fourier features) and 5.2 (for neural network features). Two solutions shown in the right panel correspond to the finding of poor (in green) and good (in magenta) stationary points when minimizing the non-convex Least Squares cost in the case of neural network features.

how these weights were learned in each instance are discussed in separate examples following the figure.

Example 5.1 Regression with fixed bases of features

To perform regression using a fixed basis of features (e.g., polynomials or Fourier) it is natural to choose a degree D and transform the input data using the associated basis functions. For example, employing a degree D polynomial or Fourier basis for a scalar input, we transform each input x_p to form an associated feature vector $\mathbf{f}_p = \begin{bmatrix} x_p & x_p^2 & \cdots & x_p^D \end{bmatrix}^T$ or $\mathbf{f}_p = \begin{bmatrix} \cos(2\pi x_p) & \sin(2\pi x_p) & \cdots & \cos(2\pi D x_p) & \sin(2\pi D x_p) \end{bmatrix}^T$ respectively. For higher dimensions of input N fixed basis features can be similarly used; however, the sheer number of elements involved (the length of each $\mathbf{f}_p$)

explodes[11] for even moderate values of N and D (as we will see in Section 7.1, this issue can be ameliorated via the notion of a "kernal," however, this introduces a serious numerical optimization problem as the size of the data-set grows).

In any case, once feature vectors $\mathbf{f}_p$ have been constructed using the data we can then determine proper weights b and $\mathbf{w}$ by minimizing the Least Squares cost function as

$$\underset{b,\mathbf{w}}{\text{minimize}} \sum_{p=1}^{P} \left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2. \tag{5.18}$$

Using the compact notation $\tilde{\mathbf{w}} = \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}$ and $\tilde{\mathbf{f}}_p = \begin{bmatrix} 1 \\ \mathbf{f}_p \end{bmatrix}$ for each p we may rewrite the cost as $g\left(\tilde{\mathbf{w}}\right) = \sum_{p=1}^{P} \left(\tilde{\mathbf{f}}_p^T\tilde{\mathbf{w}} - y_p\right)^2$, and checking the first order condition then gives the linear system of equations

$$\left(\sum_{p=1}^{P} \tilde{\mathbf{f}}_p\tilde{\mathbf{f}}_p^T\right)\tilde{\mathbf{w}} = \sum_{p=1}^{P} \tilde{\mathbf{f}}_p y_p, \tag{5.19}$$

that when solved recovers an optimal set of parameters $\tilde{\mathbf{w}}$.

Example 5.2 Regression with a basis of single hidden layer neural network features

The feature vector of the input $\mathbf{x}_p$ made by using a basis of single hidden layer neural network features takes the form

$$\mathbf{f}_p = \begin{bmatrix} a\left(c_1 + \mathbf{x}_p^T\mathbf{v}_1\right) & a\left(c_2 + \mathbf{x}_p^T\mathbf{v}_2\right) & \cdots & a\left(c_M + \mathbf{x}_p^T\mathbf{v}_M\right) \end{bmatrix}^T, \tag{5.20}$$

where $a(\cdot)$ is any activation function as detailed in Section 5.1.4. However, unlike the case with fixed feature bases, the corresponding Least Squares problem

$$\underset{b,\mathbf{w},\Theta}{\text{minimize}} \sum_{p=1}^{P} \left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2, \tag{5.21}$$

cannot be solved in closed form due to the internal parameters (denoted all together in the set Θ) that are related in a nonlinear fashion with the basis weights b and $\mathbf{w}$. Moreover, this problem is almost always non-convex, and so several runs of gradient descent are typically made in order to ensure convergence to a good local minimum.

[11] For a general N-dimensional input $\mathbf{x}$, a degree D polynomial basis-feature transformation includes all monomials of the form $f_m(\mathbf{x}) = x_1^{m_1}x_2^{m_2}\cdots x_N^{m_N}$ where $0 \leq m_1 + m_2 + \cdots + m_N \leq D$. Similarly, a degree D Fourier expansion contains basis elements of the form $f_m(\mathbf{x}) = e^{2\pi i m_1 x_1}e^{2\pi i m_2 x_2}\cdots e^{2\pi i m_N x_N}$ where $-D \leq m_1, m_2, \cdots, m_N \leq D$. Containing all non-constant terms, one can easily show that the associated polynomial and Fourier feature vectors have length $M = \frac{(N+D)!}{N!D!} - 1$ and $M = (2D+1)^N - 1$, respectively. Note that in both cases the feature vector dimension grows extremely rapidly in N and D, which can lead to serious practical problems even with moderate amounts of N and D.

Calculating the full gradient of the cost g, we have a vector of length $Q = M(N+2)+1$ containing the derivatives of the cost with respect to each variable,

$$\nabla g = \begin{bmatrix} \frac{\partial}{\partial b} g & \frac{\partial}{\partial w_1} g & \cdots & \frac{\partial}{\partial w_M} g & \frac{\partial}{\partial c_1} g \cdots & \frac{\partial}{\partial c_M} g & \nabla_{\mathbf{v}_1}^T g & \cdots & \nabla_{\mathbf{v}_M}^T g \end{bmatrix}^T, \quad (5.22)$$

where the derivatives are easily calculated using the chain rule (see Exercise 5.9).

Example 5.3 Regression with a basis of multiple hidden layer neural network features

As discussed in Section 5.1.4, by increasing the number of layers in a neural network each basis function gains more flexibility. In their use with machine learning, this added flexibility comes at the practical expense of making the corresponding cost function more challenging to minimize. In terms of numerical optimization the primary challenge with deep net features is that the associated cost function can become highly non-convex (i.e., having many local minima and/or saddle points). Proper choice of activation function can help ameliorate this problem, e.g., the hinge or rectified linear function $a(x) = \max(0, x)$ has been shown to work well for deep nets (see e.g., [36]). Furthermore, as discussed in Section 3.3.2, regularization can also be used to improve this problem.

A practical implementation issue with using deep net features is that computing the gradient, for use with gradient descent (often referred to in the machine learning community as *the backpropagation algorithm*) of associated cost functions becomes more cumbersome as additional layers are added, requiring several applications of the chain rule and careful book-keeping to ensure no errors are made. For the interested reader we provide an organized derivation of the gradient for a cost function that employs deep net features in Section 7.2. To avoid potential errors in computing the derivatives of a deep net cost function by hand, computational techniques like automatic differentiation [59] are often utilized when using deep nets in practice.

5.3 Cross-validation for regression

In the ideal instance of regression, where we look to approximate a continuous function using a fixed or adjustable (neural network) basis of features, we saw in Section 5.1 that using more elements of a basis results in a better approximation (see e.g., Fig. 5.3). In short, in the context of continuous function approximation more (basis elements) is always better. Does the same principle apply in the real instance of regression, i.e., in the case of a noisily sampled function approximation problem? Unfortunately, *no*.

Take for example the semi-ideal and realistic sinusoidal datasets shown in Fig. 5.13, along with polynomial fits of degrees three (in blue) and ten (in purple). In the left panel of this figure, which shows the discrete sinusoidal dataset with evenly spaced points, by increasing the number of basis features M from 3 to 10 the corresponding polynomial model fits the data *and* the underlying function better. Conversely, in the right panel while the model fits the data better as we increase the number of polynomial features

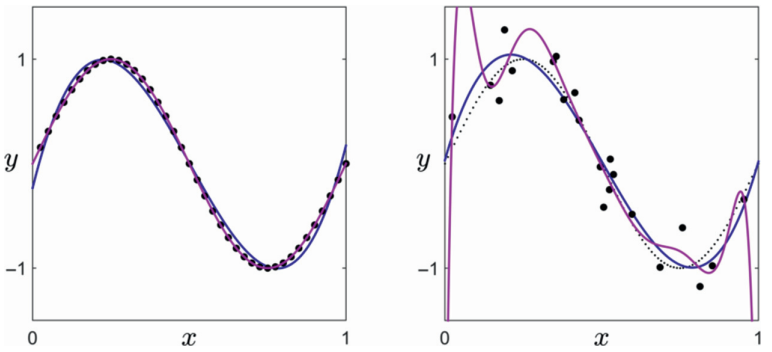

Fig. 5.13 Plots of (left panel) discretized and (right panel) noisy samples of the data-generating function $y(x) = \sin(2\pi x)$, along with its degree three and degree ten polynomial approximations in blue and purple, respectively. While the higher degree polynomial does a better job at modeling both the discretized data and underlying function, it only fits the noisy sample data better, providing a worse approximation of the underlying data-generating function than the lower degree polynomial. Using cross-validation we can determine the more appropriate model, in this case the degree three polynomial, for such a dataset.

from $M = 3$ to 10, the representation of the underlying data-generating function actually gets *worse*. Since the underlying function is the object we truly wish to understand, this is a problem.

The phenomenon illustrated through this simple example is in fact true more generally: by increasing the number M of any type of basis features (fixed or neural network) we can indeed produce better fitting models of a dataset, but at the potential cost of creating poorer representations of the data-generating function we care foremost about. Stated formally, given any dataset we can drive the value of the Least Squares cost to zero via solving the minimization problem

$$\underset{b,\mathbf{w},\Theta}{\text{minimize}} \sum_{p=1}^{P} \left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2, \tag{5.23}$$

by increasing M where $\mathbf{f}_p = \begin{bmatrix} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots & f_M\left(\mathbf{x}_p\right) \end{bmatrix}^T$. Therefore, choosing M correctly is extremely important. Note in the language of machine learning, a model corresponding to too large a choice of M is said to *overfit* the data. Likewise when choosing M too small[12] the model is said to *underfit* the data.

In this section we describe *cross-validation*, an effective framework for choosing the proper value for M automatically and intelligently so as to prevent the problem of underfitting/overfitting. For example, in the case of the data shown in the right panel of Fig. 5.13 cross-validation will determine $M = 3$ the better model, as opposed to $M = 12$. This discussion will culminate in the description of a specific procedure known as *k-fold cross-validation* which is commonly used in practice.

[12] For instance, using a degree $M = 1$ polynomial feature we can only find the best linear fit to the data in Fig. 5.13, which would be not only a poor fit to the observed data but also a poor representation of the underlying sinusoidal pattern.

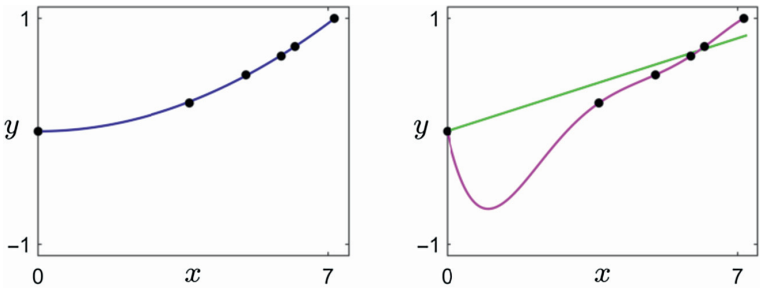

Fig. 5.14 Data from Galileo's simple ramp experiment from Example 3.3, exploring the relationship between time and the distance an object falls due to gravity. (left panel) Galileo fit a simple quadratic to the data. (right panel) A linear model (shown in green) is not flexible enough and as a result, underfits the data. A degree twelve polynomial (shown in magenta) overfits the data, being too complicated and unnatural (between the start and 0.25 of the way down the ramp the ball travels a negative distance!) to be a model of a simple natural phenomenon.

Example 5.4 Overfitting and underfitting Galileo's ramp data

In the left panel of Fig. 5.14 we show the data from Galileo's classic ramp experiment, initially described in Example 1.7, performed in order to understand the relationship between time and the acceleration of an object due to (the force we today know as) gravity. Also shown in this figure is (left panel) the kind of quadratic fit Galileo used to describe the underlying relationship traced out by the data, along with two other possible model choices (right panel): a linear fit in green, as well as a degree 12 polynomial fit in magenta. Of course the linear model is inappropriate, as with this data any line would have large squared error (see e.g., Fig. 3.3) and would thus be a poor representation of the data. On the other hand, while the degree 12 polynomial fits the data-set perfectly, with corresponding squared error value of zero, the model itself just "looks wrong."

Examining the right panel of this figure why, for example, when traveling between the beginning and a quarter of the way down the ramp, does the distance the ball travels become negative! This kind of behavior does not at all match our intuition or expectation about how gravity should operate on an object. This is why Galileo chose a quadratic, rather than a higher order degree polynomial, to fit such a data-set: because he *expected* that the rules which govern our universe are explanatory yet simple.

This principle, that the rules we use to describe our universe should be flexible yet simple, is often called *Occam's Razor* and lies at the heart of essentially all scientific inquiry past and present. Since machine learning can be thought of as a set of tools for making sense of arbitrary kinds of data, i.e., not only data relating to a physical system or law, we want the relationship learned in solving a regression (or classification) problem to also satisfy this basic Occam's Razor principle. In the context of machine learning, Occam's Razor manifests itself geometrically, i.e., we expect the model (or function) underlying our data to be simple yet flexible enough to explain the data we have. The linear model in Fig. 5.14, being too rigid and inflexible to establish the relationship between time and the distance an object falls due to gravity, fits very poorly. As previously mentioned, in machine learning such a model is said to underfit the data we have. On

the other hand, the degree 12 polynomial model is needlessly complicated, resulting in a very close fit to the data we have, but is far too oscillatory to be representative of the underlying phenomenon and is said to overfit the data.

5.3.1 Diagnosing the problem of overfitting/underfitting

A reasonable diagnosis of the overfitting/underfitting problems is that both fail at representing *new* data, generated via the same process by which the current data was made, that we can potentially receive in the future. For example, the overfitting degree ten polynomial shown in the right panel of Fig. 5.13 would poorly model any future data generated by the same process since it poorly represents the underlying data-generating function (a sinusoid). This data-centric perspective provokes a practical criterion for determining an ideal choice of M for a given dataset: the number M of basis features used should be such that the corresponding model fits well to both the current dataset as well as to new data we will receive in the future.

5.3.2 Hold out cross-validation

While we of course do not have access to any "new data we will receive in the future," we can *simulate* such a scenario by splitting our data into two subsets: a larger *training set* of data we already have, and a smaller *testing set* of data that we "will receive in the future." Then, we can try a range of values for M by fitting each to the training set of known data, and pick the one that performs the best on our testing set of unknown data. By keeping a larger portion of the original data as the training set we can safely assume that the learned model which best represents the testing data will also fit the training set fairly well. In short, by employing this sort of procedure for comparing a set of models, referred to as *hold out cross-validation*, we can determine a candidate that approximately satisfies our criterion for an ideal well-fitting model.

What portion of our dataset should we save for testing? *There is no hard rule*, and in practice typically between $1/10$ to $1/3$ of the data is assigned to the testing set. One general rule of thumb is that the larger the dataset (given that it is relatively clean and well distributed) the bigger the portion of the original data may be assigned to the testing set (e.g., $1/3$ may be placed in the testing set) since the data is plentiful enough for the training data to still accurately represent the underlying phenomenon. Conversely, in general with smaller or less rich (i.e., more noisy or poorly distributed) datasets we should assign a smaller portion to the testing set (e.g., $1/10$ may be placed in the testing set) so that the relatively larger training set retains what little information of the underlying phenomenon was captured by the original data.

> In general the larger/smaller the original dataset the larger/smaller the portion of the original data that should be assigned to the testing set.

Fig. 5.15 Hold out cross-validation. The original data (left panel) shown here as the entire circular mass is split randomly (middle panel) into k non-overlapping sets (here $k = 3$). (right panel) One piece, or $1/k$ of the original dataset, is then taken randomly as the testing set with the remaining pieces, or $k-1/k$ of the original data, taken as the training set.

As illustrated in Fig. 5.15, to form the training and testing sets we split the original data randomly into k non-overlapping parts and assign 1 portion for testing ($1/k$ of the original data) and $k - 1$ portions to the training set ($k-1/k$ of the original data).

Regardless of the value we choose for k, we train our model on the training set using a range of different values of M. We then evaluate how well each model (or in other words, each value of M) fits to both the training and testing sets, via measuring the model's *training error* and *testing error*, respectively. The best-fitting model is chosen as the one providing the lowest testing error or the best fit to the "unseen" testing data. Finally, in order to leverage the full power of our data we use the optimal number of basis features M to train our model, this time using the entire data (both training and testing sets).

Example 5.5 Hold out for regression using Fourier features

To solidify these details, in Fig. 5.16 we show an example of applying hold out cross-validation using a dataset of $P = 30$ points generated via the function $y(x)$ shown in Fig. 5.3. To perform hold out cross-validation on this dataset we randomly partition it into $k = 3$ equal-sized (ten points each) non-overlapping subsets, using two partitions together as the training set and the final part as testing set, as illustrated in the left panel of Fig. 5.16. The points in this panel are colored blue and yellow indicating that they belong to the training and testing sets respectively. We then train our model on the training set (blue points) by solving several instances of the Least Squares problem in (5.18). In particular we use a range of even values for M Fourier features $M = 2, 4, 6, \ldots, 16$ (since Fourier elements naturally come in pairs of two as shown in Equation (5.7)) which corresponds to the range of degrees $D = 1, 2, 3, \ldots, 8$ (note that for clarity panels in the figure are indexed by D).

Based on the models learned for each value of M (see the middle set of eight panels of the figure) we plot training and testing errors (in the panel second from the right), measuring how well each model fits the training and testing data respectively, over the entire range of values. Note that unlike the testing error, the training error always decreases as we increase M (which occurs more generally regardless of the dataset/feature basis used). The model that provides the smallest testing error ($M^\star = 10$ or equivalently

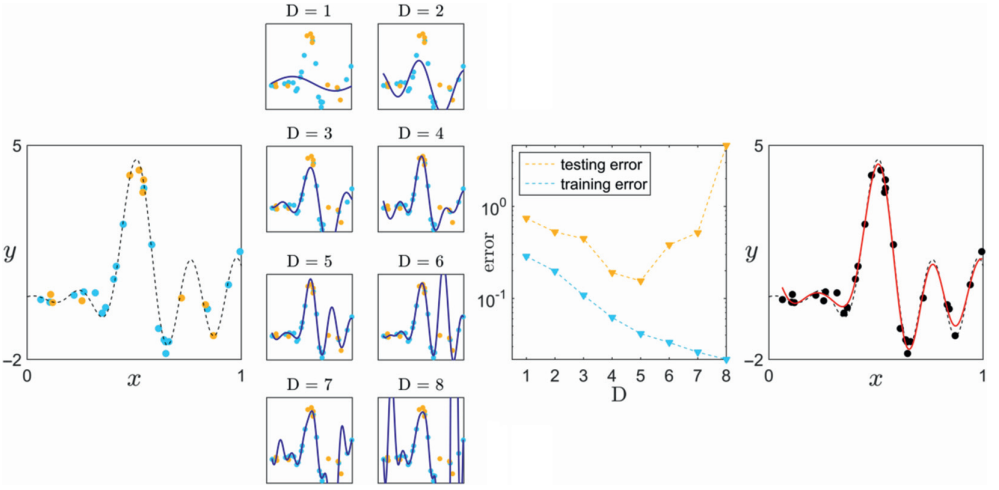

Fig. 5.16 An example of hold out cross-validation applied to a simple dataset using Fourier features. (left panel) The original data split into training and testing sets, with the points belonging to each set colored blue and yellow respectively. (middle eight panels) The fit resulting from each set of degree D Fourier features in the range $D = 1, 2, \ldots, 8$ is shown in blue in each panel. Note how the lower degree fits underfit the data, while the higher degree fits overfit the data. (second from right panel) The training and testing errors, in blue and yellow respectively, of each fit over the range of degrees tested. From this we see that $D^\star = 5$ (or $M^\star = 10$) provides the best fit. Also note how the training error always decreases as we increase the degree/number of basis elements, which will always occur regardless of the dataset/feature basis type used. (right panel) The final model using $M^\star = 10$ trained on the entire dataset (shown in red) fits the data well and closely matches the underlying data generating function (shown in dashed black).

$D^\star = 5$) is then trained again on the entire dataset, giving the final regression model shown in red in the rightmost panel of Fig. 5.16.

5.3.3 Hold out calculations

Here we give a complete set of hold out cross-validation calculations in a general setting. We denote the collection of points belonging to the training and testing sets respectively by their indices as

$$\begin{aligned} \Omega_{\text{train}} &= \left\{p \,|\, \left(\mathbf{x}_p,\, y_p\right) \text{ belongs to the training set}\right\} \\ \Omega_{\text{test}} &= \left\{p \,|\, \left(\mathbf{x}_p,\, y_p\right) \text{ belongs to the testing set}\right\}. \end{aligned} \tag{5.24}$$

We then choose a basis type (e.g., polynomial, Fourier, neural network) and choose a range for the number of basis features over which we search for an ideal value for M. To determine the training and testing error of each value of M tested we first form the corresponding feature vector $\mathbf{f}_p = \left[\, f_1\left(\mathbf{x}_p\right) \quad f_2\left(\mathbf{x}_p\right) \quad \cdots \quad f_M\left(\mathbf{x}_p\right) \,\right]^T$ and fit a corresponding model to the training set by solving the corresponding[13] Least Squares problem

[13] Once again, for a fixed basis this problem may be solved in closed form since Θ is empty, while for neural networks it must be solved via gradient descent (see e.g., Example 5.2).

$$\underset{b,\,\mathbf{w},\Theta}{\text{minimize}} \sum_{p\in\Omega_{\text{train}}} \left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2. \tag{5.25}$$

Denoting a solution to the problem above as $\left(b_M^\star,\, \mathbf{w}_M^\star,\, \Theta_M^\star\right)$ we find the training and testing errors for the current value of M by simply computing the mean squared error using these parameters over the training and testing sets, respectively

$$\begin{aligned} \text{Training error} &= \frac{1}{|\Omega_{\text{train}}|} \sum_{p\in\Omega_{\text{train}}} \left(b_M^\star + \mathbf{f}_p^T\mathbf{w}_M^\star - y_p\right)^2 \\ \text{Testing error} &= \frac{1}{|\Omega_{\text{test}}|} \sum_{p\in\Omega_{\text{test}}} \left(b_M^\star + \mathbf{f}_p^T\mathbf{w}_M^\star - y_p\right)^2, \end{aligned} \tag{5.26}$$

where the notation $|\Omega_{\text{train}}|$ and $|\Omega_{\text{test}}|$ denotes the cardinality or number of points in the training and testing sets, respectively. Once we have performed these calculations for all values of M we wish to test, we choose the one that provides the lowest testing error, denoted by $M^\star$.

Finally we form the feature vector $\mathbf{f}_p = \left[\begin{array}{ccc} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots \quad f_{M^\star}\left(\mathbf{x}_p\right) \end{array}\right]^T$ for all the points in the entire dataset, and solve the Least Squares problem over the entire dataset to form the final model

$$\underset{b,\,\mathbf{w},\Theta}{\text{minimize}} \sum_{p=1}^{P} \left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2. \tag{5.27}$$

5.3.4 k-fold cross-validation

While the hold out method previously described is an intuitive approach to determining proper fitting models, it suffers from an obvious flaw: having been chosen at random, the points assigned to the training set may not adequately describe the original data. However, we can easily extend and robustify the hold out method as we now describe.

As illustrated in Fig. 5.17 for $k = 3$, with *k-fold cross-validation* we once again randomly split our data into k non-overlapping parts. By combining $k - 1$ parts we can, as with the hold out method, create a large training set and use the remaining single fold as a test set. With k-fold cross-validation we will repeat this procedure k times (each instance being referred to as a *fold*), in each instance using a different single portion of the split as testing set and the remaining $k - 1$ parts as the corresponding training set, and computing the training and testing errors of all values of M as described in the previous section. We then choose the value of M that has the lowest *average testing error*, a more robust choice than the hold out method provides, that can average out a scenario where one particular choice of training set inadequately describes the original data.

Note, however, that this advantage comes at a cost: k-fold cross-validation is (approximately) k times more computationally costly than its hold out counterpart. In fact performing k-fold cross-validation is often the most computationally expensive process performed to solve a regression problem.

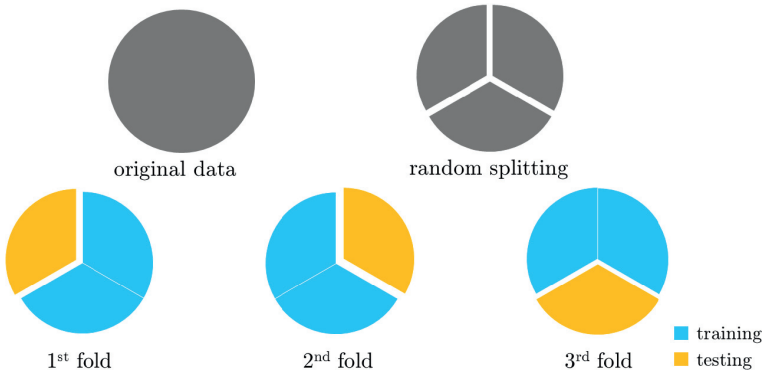

Fig. 5.17 k-fold cross-validation for $k = 3$. The original data shown here as the entire circular mass (top left) is split into k non-overlapping sets (top right) just as with the hold out method. However with k-fold cross-validation we repeat the hold out calculations k times (bottom), once per "fold," in each instance, keeping a different portion of the split data as the testing set while merging the remaining $k - 1$ pieces as the training set.

Performing k-fold cross-validation is often the most computationally expensive component in solving a general regression problem.

There is again no universal rule for the number k of non-overlapping partitions (or the number of folds) to break the original data into. However, the same intuition previously described for choosing k with the hold out method also applies here, as well as the same convention with popular values of k ranging from $k = 3 \ldots 10$ in practice.

For convenience we provide a pseudo-code for applying k-fold cross-validation in Algorithm 5.1.

Algorithm 5.1 k-fold cross-validation pseudo-code

Input: Data-set $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, k (number of folds), a range of values for M to try, and a type of basis feature
Split the data into k equal (as possible) sized folds
for $s = 1 \ldots k$
    for each M (in the range of values to try)
        1) Train a model with M basis features on sth fold's training set
        2) Compute corresponding testing error on this fold
Return: value $M^\star$ with lowest *average* testing error over all k folds

Example 5.6 k-fold cross-validation for regression using Fourier features

In Fig. 5.18 we illustrate the result of applying k-fold cross-validation to choose the ideal number M of Fourier features for the dataset shown in Example 5.5, where it was

originally used to illustrate the hold out method. As in the previous example, here we set $k = 3$ and try M in the range $M = 2, 4, 6, \ldots, 16$, which corresponds to the range of degrees $D = 1, 2, 3, \ldots, 8$ (note that for clarity, panels in the figure are indexed by D).

In the top three rows of Fig. 5.18 we show the result of applying hold out on each fold. In each row we show a fold's training and testing data colored blue and yellow respectively in the left panel, the training/testing errors for each M on the fold (as computed in

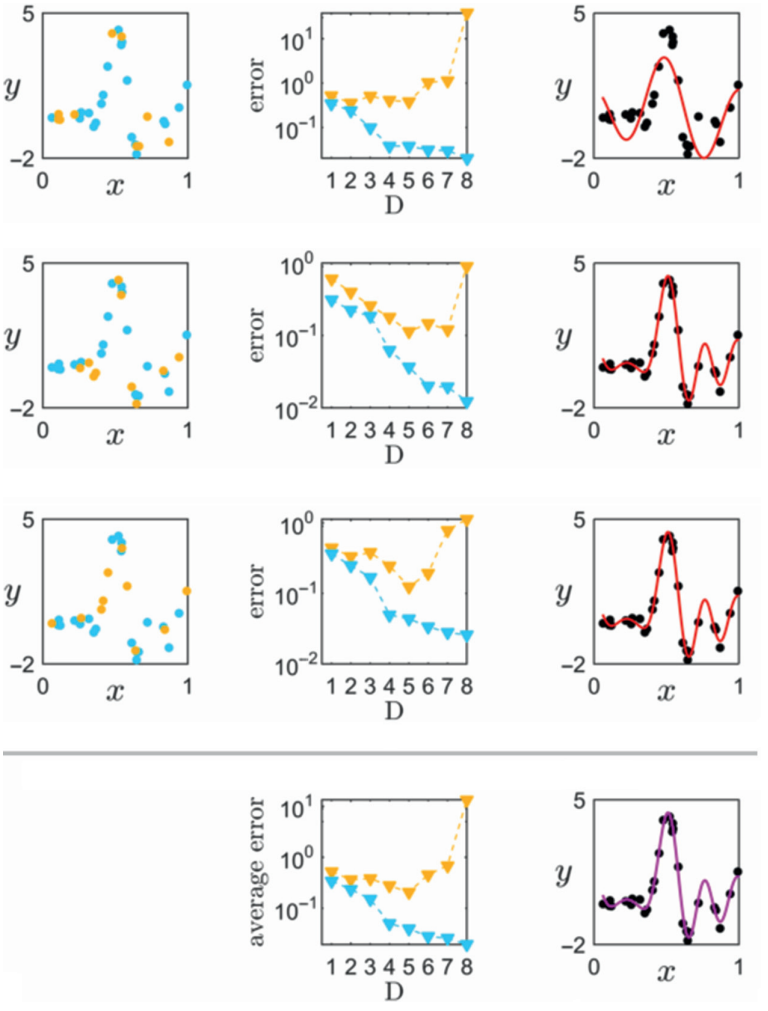

Fig. 5.18 Result of performing k-fold cross-validation with $k = 3$ (see text for further details). The top three rows display the result of performing the hold out method on each fold. The left, middle, and right columns show each fold's training/testing sets (colored blue and yellow respectively) training and testing errors over the range of M tried, and the final model (fit to the entire dataset) chosen by picking the value of M providing the lowest testing error. Due to the split of the data, performing hold out on the first fold (top row) results in a poor underfitting model for the data. However, as illustrated in the final row, by averaging the testing errors (bottom middle panel) and choosing the model with minimum associated average test error, we average out this problem (finding that $D^\star = 5$ or $M^\star = 10$) and determine an excellent model for the phenomenon (as shown in the bottom right panel).

Equation (5.26)) in the middle panel, and the final model (learned to the entire dataset) provided by the choice of M with lowest testing error. As can be seen in the top row, the particular split of the first fold leads to too low a value of M being chosen, and thus an underfitting model. In the middle panel of the final row we show the result of averaging the training/testing errors over all $k = 3$ folds, and in the right panel the result of choosing the overall best $M^\star = 10$ (or equivalently $D^\star = 5$) providing the lowest average testing error. By taking this value we average out the poor choice determined on the first fold, and end up with a model that fits both the data and underlying function quite well.

Example 5.7 Leave-one-out cross-validation for Galileo's ramp data

In Fig. 5.19 we show how using $k = P$ fold cross-validation (since we have only $P = 6$ data points, intuition suggests, see Section 5.3.2, that we use a large value for k), sometimes referred to as *leave-one-out cross-validation*, allows us to recover precisely the quadratic fit Galileo made by eye. Note that by choosing $k = P$ this means that every data point will take a turn being the testing set. Here we search over the polynomial basis features of degree $M = 1 \ldots 6$. While not all of the hold out models over the six folds fit the data well, the average k-fold result is indeed the $M^\star = 2$ quadratic polynomial fit originally proposed by Galileo!

5.4 Which basis works best?

While some guidance can be given in certain situations regarding the best basis to employ, no general rule exists for which basis one should use in all instances

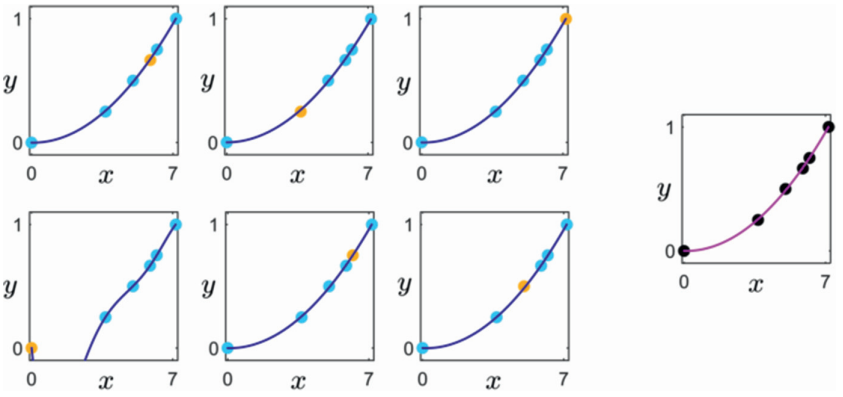

Fig. 5.19 (six panels on the left) Each fold of training/testing sets shown in blue/yellow respectively of a k-fold run on the Galileo's ramp data, along with their individual hold out model (shown in blue). Only the model learned on the fourth fold overfits the data. By choosing the model with minimum average testing error over the $k = 6$ folds we recover the desired quadratic $M^\star = 2$ fit originally proposed by Galileo (shown in magenta in the right panel).

of regression. Indeed for an arbitrary dataset it may very well be the case that no basis/feature map is especially better than any other. However, in some instances understanding of the phenomenon underlying the data and practical considerations can lead one to a particular choice of basis or at least eliminate potential candidates.

5.4.1 Understanding of the phenomenon underlying the data

Gathering some understanding of a phenomenon, while not always easy, is generally the most effective way of deducing the particular effectiveness of a specific basis. For example, the gravitational phenomenon underlying Galileo's ramp dataset, shown in e.g., Fig. 5.19, is extremely well understood as quadratic in nature, implying the appropriateness of a polynomial basis [58]. Fourier basis/feature map are intuitively appropriate if dealing with data generated from a known periodic phenomenon. Often referred to as "time series" data (due to the input variable being time), periodic behavior arises in a variety of disciplines including speech processing and financial modeling (see e.g., [37, 40, 67]). Fourier and neural network bases/features are often employed with image and audio data, in the latter case due to a belief in the correspondence of neural network bases and the way such data is processed by the human brain or the compositional structure of certain problems (see e.g., [17, 19, 36] and references therein). Importantly, note that the information used to favor a particular basis need not come from a regression (or more broadly a machine learning) problem itself but rather from scientific understanding of a phenomenon more broadly.

5.4.2 Practical considerations

Practical considerations can also guide the choice of basis/feature type. For example if, given the nature of a dataset's input variable, it does not make sense to normalize its values to lie in the range $[0, 1]$, then polynomials can be a very poor choice of basis/features. This is due to the fact that polynomials grow rapidly for values outside this range, e.g., $x^{20} = 3\,486\,784\,401$ when $x = 3$, making the corresponding (closed form) calculations for solving the Least Squares problem difficult if not impossible to perform on a modern computer. The amount of engineering involved in effectively employing a kernelized form of a fixed basis (see Chapter 7 for further details) or a deep neural network (e.g., the number of layers as well as the number of activation functions within each layer, type of activation function, etc.) can also guide the choice of basis in practice.

5.4.3 When the choice of basis is arbitrary

In the (rare) scenario where data is plentiful, relatively noise free, and nicely distributed throughout the input space, the choice of basis/feature map is fairly arbitrary, since in such a case the data carves out the entire underlying function fairly well (and so we

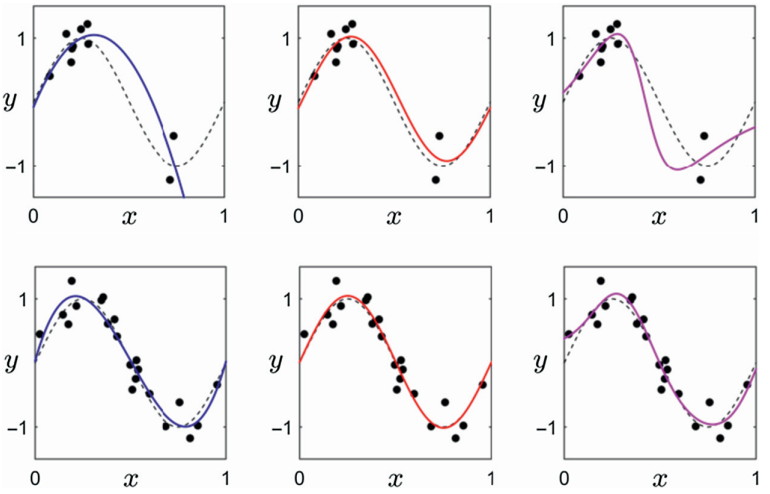

Fig. 5.20 A comparison of the k-fold found best polynomial (left column), Fourier (middle column), and single hidden layer neural network (right column) fit to two datasets (see text for further details). Since the data generating function belongs to the Fourier basis itself, the Fourier cross-validated model fits the underlying function better on the smaller dataset (top row). On the larger dataset the choice of basis is less crucial to finding a good cross-validated model, as the relatively large amount of data carves out the entire underlying function fairly well.

essentially "revert" to the problem of continuous function approximation described in Section 5.1 where all bases are equally effective).

Example 5.8 A simulated dataset where one basis is more appropriate than others

In Fig. 5.20 we illustrate (using simulated datasets) the scenario where one basis is more effective than others, as well as when the data is large/clean/well distributed enough to make the choice of basis a moot point. In this figure we show two datasets of $P = 10$ (top row) and $P = 21$ (bottom row) points generated using the underlying function $y(x) = \sin(2\pi x)$, and the result of k-fold cross validated polynomial (left column), Fourier (middle column), and single hidden layer neural network with $\tanh(\cdot)$ activation (right column) bases.[14] Because the data generating function itself belongs to the Fourier basis, the Fourier cross-validated model fits the underlying model better than the other two bases on the smaller dataset. However, on the larger dataset the choice is less important, with all three cross-validated models performing well.

[14] For each dataset the same training/testing sets were used in performing k-fold cross-validation for each basis feature type. For the smaller dataset we have used $k = 5$, while for the larger dataset we set $k = 3$. For all three basis types the range of degrees/number of hidden units was in the range $M = 1, 2, 3, 4, 5$.

5.5 Summary

In this chapter we have described how features may be designed automatically for the general problem of regression, by viewing it as a noisily sampled function approximation problem. Beginning in Section 5.1.3 with the perfect but unrealistic scenario where we have the data generating function itself, we introduced both fixed and adjustable feed forward neural network bases of fundamental features whose elements can be combined in order to automatically generate features to approximate any such function.

The corresponding Least Squares problem, given in Equation (5.13), for learning proper weights of these bases being intractable, we then saw in Section 5.2 how discretization of both the bases and desired function leads to a corresponding discretized Least Squares problem that closely approximates the original and can be solved using numerical optimization. In this same section we next saw how the realistic case of nonlinear regression (where the data is assumed to be noisy samples of some underlying function) is described mathematically using the same discretized framework, and thus how the same discretized Least Squares problem can be used for regression.

However, we saw in the third section that while increasing the number of basis features creates a better fitting model for the data we currently have, this will overfit the data giving a model that poorly represents data we might receive in the future/the underlying data generating function. This motivated the technique of cross-validation, culminating in the highly useful but computationally costly k-fold cross-validation method, for choosing the proper number of basis features in order to prevent this.

In the final section we discussed the proper choice of basis/feature map. While generally speaking we can say little about which basis will work best in all circumstances, understanding of the phenomenon underlying the data as well as practical considerations can be used to choose a proper basis (or at least narrow down potential candidates) in many important instances.

5.6 Exercises

Section 5.1 exercises

Exercises 5.1 The convenience of orthogonality

A basis $\{\mathbf{x}_m\}_{m=1}^{M}$ is often chosen to be *orthogonal* (typically to simplify calculations) meaning that

$$\mathbf{x}_k^T\mathbf{x}_j = \begin{cases} S & \text{if } k = j \\ 0 & \text{else,} \end{cases} \tag{5.28}$$

for some $S > 0$. For example, the "standard basis" defined for each $m = 1 \dots M$ as

$$e_{m,j} = \begin{cases} 1 & \text{if } m = j \\ 0 & \text{else.} \end{cases} \tag{5.29}$$

is clearly orthogonal. Another example is the Discrete Cosine Transform basis described in Example 9.3. An orthogonal basis provides much more easily calculable weights for representing a given vector $\mathbf{y}$ than otherwise, as you will show in this exercise.

Show in this instance that the ideal weights are given simply as $w_j = \frac{1}{S}\mathbf{x}_j^T\mathbf{y}$ for all j by solving the Least Squares problem

$$\underset{w_1 \ldots w_M}{\text{minimize}} \left\| \sum_{m=1}^{M} \mathbf{x}_m w_m - \mathbf{y} \right\|_2^2 \tag{5.30}$$

for the jth weight w_j. Briefly describe how this compares to the Least Squares solution.

Exercises 5.2 Orthogonal basis functions

In analogy to orthogonal bases in the case of N-dimensional vectors, a set of basis functions $\{f_m\}_{m=0}^{\infty}$, used in approximating a function $y(x)$ over the interval $[0, 1]$ is *orthogonal* if

$$\langle f_m, f_j \rangle = \int_0^1 f_m(x) f_j(x)\, dx = \begin{cases} S & \text{if } m = j \\ 0 & \text{else} \end{cases} \tag{5.31}$$

for some $S > 0$. The integral quantity above defines the continuous inner product between two functions, and is a generalization of the vector inner product.

Show, as in the finite dimensional case, that orthogonality provides an easily expressible set of weights of the form $w_j = \frac{1}{S}\int_0^1 f_j(x)\, y(x)\, dx$ as solutions to the corresponding Least Squares problem:

$$\underset{w_0, w_1, \ldots}{\text{minimize}} \int_0^1 \left(\sum_{m=0}^{\infty} f_m(x)\, w_m - y(x) \right)^2 dx, \tag{5.32}$$

for the optimal jth weight w_j. *Hint: you may pass each derivative* $\frac{\partial}{\partial w_j}$ *through both the integral and sum.*

Exercises 5.3 The Fourier basis is orthogonal

Using the fact that basis functions $\sin(2\pi kx)$ and $\cos(2\pi jx)$ are orthogonal functions, i.e.,

$$\begin{aligned} &\int_0^1 \sin(2\pi kx)\cos(2\pi jx)\, dx = 0 \text{ for all } k, j \\ &\int_0^1 \sin(2\pi kx)\sin(2\pi jx)\, dx = \begin{cases} 1/2 & \text{if } k = j \\ 0 & \text{else} \end{cases} \\ &\int_0^1 \cos(2\pi kx)\cos(2\pi jx)\, dx = \begin{cases} 1/2 & \text{if } k = j \\ 0 & \text{else} \end{cases}, \end{aligned} \tag{5.33}$$

find the ideal Least Squares coefficients, that is the solution to

$$\underset{w_0,w_1,\ldots,w_{2K}}{\text{minimize}} \int_0^1 \left(w_0 + \sum_{k=1}^{K} \left(w_{2k-1} \sin\left(2\pi kx\right) + w_{2k} \cos\left(2\pi kx\right) \right) - y\left(x\right) \right)^2 dx. \quad (5.34)$$

These weights will be expressed as simple integrals.

Exercises 5.4 Least Squares weights for polynomial approximation

The Least Squares weights for a degree D polynomial basis of a scalar input function $y\left(x\right)$ over $[0, 1]$ (note there are $M = D + 1$ terms in this case) are determined by solving the Least Squares problem

$$\underset{w_0,w_1,\ldots,w_D}{\text{minimize}} \int_0^1 \left(\sum_{m=0}^{D} x^m w_m - y\left(x\right) \right)^2 dx. \quad (5.35)$$

Show that solving the above by setting the derivatives of the cost function in each w_j equal to zero results in a linear system of the form

$$\mathbf{P}\mathbf{w} = \mathbf{d}. \quad (5.36)$$

In particular show that $\mathbf{P}$ (often referred to as a Hilbert matrix) takes the explicit form

$$\mathbf{P} = \begin{bmatrix} 1 & 1/2 & 1/3 & \cdots & 1/D+1 \\ 1/2 & 1/3 & 1/4 & \cdots & 1/D+2 \\ 1/3 & 1/4 & 1/5 & \cdots & 1/D+3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/D+1 & 1/D+2 & 1/D+3 & \cdots & 1/2D+1 \end{bmatrix}. \quad (5.37)$$

Hint: $\int_0^1 x^j dx = \frac{1}{j+1}$.

Exercises 5.5 Complex Fourier representation

Verify that using complex exponential definitions of cosine and sine functions, i.e., $\cos\left(\alpha\right) = \frac{1}{2}\left(e^{i\alpha} + e^{-i\alpha}\right)$ and $\sin\left(\alpha\right) = \frac{1}{2i}\left(e^{i\alpha} - e^{-i\alpha}\right)$, we can write the partial Fourier expansion

$$w_0 + \sum_{m=1}^{M} \cos\left(2\pi mx\right) w_{2m-1} + \sin\left(2\pi mx\right) w_{2m} \quad (5.38)$$

equivalently as

$$\sum_{m=-M}^{M} e^{2\pi imx} w'_m, \quad (5.39)$$

where the complex weights $\left\{w'_m\right\}_{m=-M}^{M}$ are given in terms of the real weights $\{w_m\}_{m=0}^{2M}$ as

$$w'_m = \begin{cases} \frac{1}{2}\left(w_{2m-1} - iw_{2m}\right) & \text{if } m > 0 \\ w_0 & \text{if } m = 0 \\ \frac{1}{2}\left(w_{1-2m} + iw_{-2m}\right) & \text{if } m < 0. \end{cases} \quad (5.40)$$

Exercises 5.6 Graphical representation of a neural network

a) Use Fig. 5.9 or Equation (5.12) to count the total number of parameters Q (including both internal parameters and feature weights) in a three hidden layer neural network basis approximation. Can you generalize this to find a formula for Q in a neural network with L hidden layers? *Hint: you may find it convenient to define* $M_0 = 1$, $M_1 = M$, *and* $M_{L+1} = N$.

b) Based on your answer in part a), how well does a neural network basis scale to large datasets? More specifically, how does the input dimension N contribute to the number of parameters Q (a.k.a. the dimension of the optimization problem)? How does the number of parameters change with the number of data points P?

Section 5.2 exercises

Exercises 5.7 Polynomial basis feature regression for scalar valued input

In this exercise you will explore how various degree D polynomial basis features fit the sinusoidal dataset shown in Fig. 5.12. You will need the wrapper *poly_regression_hw* and the data located in *noisy_sin_samples.csv*.

a) Use the description of polynomial basis features given in Example 5.1 to transform the input using a general degree D polynomial. Write this feature transformation in the module

$$\mathbf{F} = \text{poly\_features}\,(\mathbf{x},\, D) \tag{5.41}$$

located in the wrapper. Here $\mathbf{x}$ is the input data, D the degree of the polynomial features, and $\mathbf{F}$ the corresponding degree D feature transformation of the input (note your code should be able to transform the input to any degree D desired).

b) With your module complete you may run the wrapper. Two figures will be generated: the first shows the data along with various degree D polynomial fits, and the second shows the mean squared error (MSE) of each fit to the dataset. Discuss the results shown in these two figures. In particular describe the relationship between a model's MSE and how well it seems to represent the phenomenon generating the data as D increases over the range shown.

Exercises 5.8 Fourier basis feature regression for scalar valued input

In this exercise you will explore how various degree D Fourier basis features fit the sinusoidal dataset shown in Fig. 5.12. For this exercise you will need the wrapper *fourier_regression_hw* and the data located in *noisy_sin_samples.csv*.

a) Use the description of Fourier basis features given in Example 5.1 to transform the input using a general degree D Fourier basis (remember that there are $M = 2D$ basis

elements in this case, a $\cos(2\pi mx)$ and $\sin(2\pi mx)$ pair for each $m = 1, \ldots, D$). Write this feature transformation in the module

$$\mathbf{F} = \text{fourier\_features}(\mathbf{x}, D) \tag{5.42}$$

located in the wrapper. Here $\mathbf{x}$ is the input data, D the degree of the Fourier features, and $\mathbf{F}$ the corresponding degree D feature transformation of the input (note your code should be able to transform the input to any degree D desired).

b) With your module complete you may run the wrapper. Two figures will be generated: the first shows the data along with various degree D Fourier fits, and the second shows the MSE of each fit to the data-set. Discuss the results shown in these two figures. In particular describe the relationship between a model's MSE and how well it seems to represent the phenomenon generating the data as D increases over the range shown.

Exercises 5.9 Single hidden layer network regression with scalar valued input

In this exercise you will explore how various initializations affect the result of an $M = 4$ neural network basis features fit to the sinusoidal dataset shown in Fig. 5.12. For this exercise you will need the wrapper *tanh_regression_hw* and the data located in *noisy_sin_samples.csv*.

a) Using the chain rule, verify that the gradient of the Least Squares problem shown in Equation (5.21) with general activation function $a(\cdot)$ is given as

$$\begin{aligned}
\frac{\partial}{\partial b}g &= 2\sum_{p=1}^{P}\left(b + \sum_{m=1}^{M} a\left(c_m + \mathbf{x}_p^T\mathbf{v}_m\right)w_m - y_p\right) \\
\frac{\partial}{\partial w_n}g &= 2\sum_{p=1}^{P}\left(b + \sum_{m=1}^{M} a\left(c_m + \mathbf{x}_p^T\mathbf{v}_m\right)w_m - y_p\right)a\left(c_n + \mathbf{x}_p^T\mathbf{v}_n\right) \\
\frac{\partial}{\partial c_n}g &= 2\sum_{p=1}^{P}\left(b + \sum_{m=1}^{M} a\left(c_m + \mathbf{x}_p^T\mathbf{v}_m\right)w_m - y_p\right)a'\left(c_n + \mathbf{x}_p^T\mathbf{v}_n\right)w_n \\
\nabla_{\mathbf{v}_n}g &= 2\sum_{p=1}^{P}\left(b + \sum_{m=1}^{M} a\left(c_m + \mathbf{x}_p^T\mathbf{v}_m\right)w_m - y_p\right)a'\left(c_n + \mathbf{x}_p^T\mathbf{v}_n\right)w_n\mathbf{x}_p,
\end{aligned} \tag{5.43}$$

where $a'(\cdot)$ is the derivative of the activation with respect to its input.

b) This gradient can be written more efficiently for programming languages like Python and MATLAB/OCTAVE that have especially good implementations of matrix/vector operations by writing it more compactly. Supposing that $a = \tanh(\cdot)$ is the activation function (meaning $a' = \text{sech}^2(\cdot)$ is the hyperbolic secant function squared) verify that the derivatives from part a) can be written more compactly for a scalar input as

$$\begin{aligned}\frac{\partial}{\partial b}g &= 2\cdot\mathbf{1}_{P\times 1}^{T}\mathbf{q}\\ \frac{\partial}{\partial w_n}g &= 2\cdot\mathbf{1}_{P\times 1}^{T}\left(\mathbf{q}\odot\mathbf{t}_n\right)\\ \frac{\partial}{\partial c_n}g &= 2\cdot\mathbf{1}_{P\times 1}^{T}\left(\mathbf{q}\odot\mathbf{s}_n\right)w_n\\ \frac{\partial}{\partial v_n}g &= 2\cdot\mathbf{1}_{P\times 1}^{T}\left(\mathbf{q}\odot\mathbf{x}\odot\mathbf{s}_n\right)w_n,\end{aligned} \tag{5.44}$$

where $q_p = \left(b+\sum_{m=1}^{M} w_m \tanh\left(c_m + x_p v_m\right) - y_p\right)$, $t_{np} = \tanh\left(c_n + x_p v_n\right)$, and $s_{np} = \operatorname{sech}^2\left(c_n + x_p v_n\right)$, and $\mathbf{q}$, $\mathbf{t}_n$, and $\mathbf{s}_n$ are the P length vectors containing these entries. Note that $\mathbf{a}\odot\mathbf{b}$ denotes the entry-wise product of vectors $\mathbf{a}$ and $\mathbf{b}$.

c) Plug in the form of the gradient from part b) into the gradient descent module called

$$\left[b, \mathbf{w}, \mathbf{c}, \mathbf{v}, \text{obj\_val}\right] = \text{tanh\_grad\_descent}\left(\mathbf{x}, \mathbf{y}, i\right) \tag{5.45}$$

located in the wrapper. Here $\mathbf{x}$ and $\mathbf{y}$ are the input and output data respectively, i is a counter that will load an initialization for all variables, and b, $\mathbf{w}$, $\mathbf{c}$, and $\mathbf{v}$ are the optimal variables learned via gradient descent. Use the maximum iteration stopping condition with 15 000 iterations, and a fixed step length $\alpha = 10^{-3}$. Initializations are already given in the wrapper.

With this module completed the wrapper will execute gradient descent three times using three different initializations, displaying for each run the corresponding fit to the data achieved at the last step of gradient descent, as well as the objective value calculated at each iteration. Briefly discuss the results shown in this figure.

Section 5.3 exercises

Exercises 5.10 Four guys and four error plots

Eric, Stanley, Kyle, and Kenneth used hold out cross-validation to find the best degree polynomial fit to their respective datasets. Based on the error plots they have made (Fig. 5.21), what advice would you give to each of them as to what their next step should be.

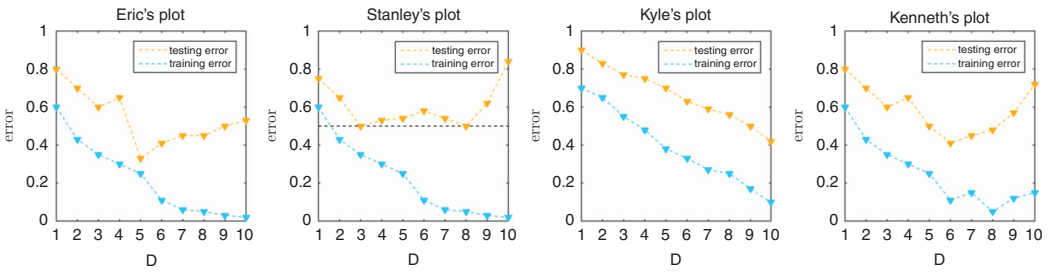

Fig. 5.21 Hold out cross-validation error plots for four different datasets.

Exercises 5.11 Practice the hold out method

In this exercise you will perform hold out cross-validation on the dataset shown in Fig. 5.16 as described in Example 5.5 of the text. Start by randomly splitting the dataset, located in *wavy_data.csv*, into $k = 3$ equal sized folds (keeping 2 folds as training, and 1 fold as testing data). Use the Fourier basis features M in the range $M = 2, 4, 6, \ldots, 16$ (or likewise D in the range $D = 1, 2, \ldots, 8$) and produce a graph showing the training and testing error for each D like the one shown in Fig. 5.16, as well as the best (i.e., the lowest test error) model fit to the data.

Note: your results may appear slightly different than those of the figure given that you will likely use a different random partition of the data. Note: you may find it very useful here to re-use code from previous exercises e.g., functions that compute Fourier features, plot curves, etc.

Exercises 5.12 Code up k-fold cross-validation

In this exercise you will perform k-fold cross-validation on the dataset shown in Fig. 5.19 as described in Example 5.7 of the text. Start by randomly splitting the dataset of $P = 6$ points, located in *galileo_ramp_data.csv*, into $k = 6$ equal sized folds (keeping 5 folds as training, and 1 fold as testing data during each round of cross-validation). The value of $k = P$ has been chosen in this instance due to the small size of the dataset (this is sometimes called "leave one out" cross-validation since each training set consists of all but one point from the original dataset).

Use the polynomial basis features and M in the range $M = 1, 2, \ldots, 6$ and produce a graph showing the average training and testing error for each M, as well as the best (i.e., the lowest average test error) model fit to the data.

Note: you may find it very useful here to re-use code from previous exercises, e.g., functions that split a dataset into k random parts, that compute training/testing errors, polynomial features, plot curves, etc.

Section 5.4 exercises

Exercises 5.13 Comparing all bases

In this exercise you will reproduce the k-fold cross-validation result shown in Fig. 5.12 using the wrapper *compare_maps_regression_hw* and the corresponding datasets shown in the figure. This wrapper performs k-fold cross-validation using the polynomial, Fourier, and single hidden layer tanh feature maps and produces the figure. It is virtually complete, i.e., the code necessary to generate the associated plot is already provided in the wrapper, save four modules you will need to include. Insert the data splitting module described in Exercise 5.11, as well as the polynomial, Fourier, and single hidden layer tanh modules for solving their corresponding Least Squares problems described in Exercises 5.7, 5.8, and 5.9 respectively. With these modules installed you should be able to run the wrapper.

Note that some of your results should appear different than those of the figure given that you will use a different random partition of the data in each instance.

5.7 Notes on continuous function approximation

Polynomials were the first provable universal approximators, this having been shown in 1885 via the so-called (Stone–) Weierstrass approximation theorem (see e.g., [71]). The Fourier basis (and its discrete derivatives) is an extremely popular function approximation tool, used particularly in physics, signal processing, and engineering fields. The convergence behavior of Fourier series has been studied for centuries, and much can be said about its convergence on larger classes of functions beyond $\mathcal{C}^N$ (see e.g., [61, 74] for a sample of results). The universal approximation properties of popular adjustable bases like the single-layer and multilayer neural networks were shown in the late 1980s and early 1990s [28, 38, 63]. Interestingly, an evolutionary step between fixed and adjustable bases, a random fixed basis where internal parameters of a given adjustable basis type are randomized, leaving only the external linear weights to be learned, has been shown to be a universal approximator more recently than deep architectures (see e.g., [69]).