7 Kernels, backpropagation, and regularized cross-validation

This chapter is essentially an appendix of technical material critically relevant to the ideas described in the previous two chapters, consisting of three sections each of which may be read independently of the other two. The first describes *fixed feature kernels*, which is a method of representing fixed basis features so that they scale more gracefully when applied to vector valued input. In the second we provide an organized set of computations of the gradient for any cost function employing multilayer neural network basis features for performing gradient descent, commonly referred to as the *backpropagation algorithm* when such deep network basis features are used. Finally, in Section 7.3 we describe a slight variation of the cross-validation technique discussed in previous chapters, called *regularized cross-validation*, that is more appropriate for fixed feature kernels, multilayer network features, as well as the softmax multiclass classification (using either fixed or neural network basis features).

7.1 Fixed feature kernels

A serious practical issue presents itself when applying fixed basis features to vector valued input: even with a moderate sized input dimension N, the corresponding dimension M of the transformed features grows rapidly with N and quickly becomes prohibitively large in terms of storage and computation. For example, the precise number M of non-bias features/feature weights of a degree D polynomial of an input with dimension N is $M = \binom{N+D}{D} - 1 = \frac{(N+D)!}{D!N!} - 1$. Even if the input dimension is of reasonably small size, for instance $N = 100$ or $N = 500$, then just the associated degree $D = 5$ polynomial feature map of these input dimensions has dimension $M = 96560\,645$ and $M = 268\,318\,178\,226$ respectively! In the latter case we cannot even hold the feature vectors in memory on a modern computer.[1]

This crucial issue, of not being able to effectively store high dimensional fixed basis feature transformations, motivates the search for more efficient representations of fixed bases. Here we introduce *kernels* or *kernelized representations* of fixed feature transformations, which are clever ways of constructing them that do not require explicit construction of the fixed features themselves. Kernels allow us to avoid this

[1] The corresponding number of transformed features with a Fourier basis/map is even more gargantuan: the degree D Fourier feature map of arbitrary input dimension N has $(2D+1)^N$ associated/feature weights. When $D = 5$ and $N = 80$ this is 11^{80}, a number larger than current estimates of the number of atoms in the visible universe (around 10^{80} atoms)!

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

combinatorial storage problem and use fixed features with vector input (at the cost, as we will see, of scaling poorly with the size of a data-set). Additionally they provide a way of generating new fixed feature maps defined solely through such a kernelized representation.

7.1.1 The fundamental theorem of linear algebra

Before discussing the concept of kernelization, it will be helpful to first recall a useful fact, generally referred to as the *fundamental theorem of linear algebra*. This is a simple statement about how to deconstruct an M length vector $\mathbf{w} \in \mathbb{R}^M$ over the columns of a given matrix.

Recall that a set of M-dimensional vectors $\left\{\mathbf{f}_p\right\}_{p=1}^{P}$ spans a subspace of dimension P, where $P \leq M$, and that any vector $\mathbf{w}$ in this subspace can be written as some linear combination of the vectors as

$$\mathbf{w} = \sum_{p=1}^{P} \mathbf{f}_p z_p, \tag{7.1}$$

where z_p are weights associated with $\mathbf{w}$. By stacking the vectors $\mathbf{f}_p$ column-wise into an $M \times P$ matrix $\mathbf{F}$ and the z_p together into a $P \times 1$ vector $\mathbf{z}$ this relationship can be written more compactly as

$$\mathbf{w} = \mathbf{F}\mathbf{z}. \tag{7.2}$$

As illustrated in Fig. 7.1, any vector $\mathbf{w} \in \mathbb{R}^M$ can then be decomposed into two pieces: the portion of $\mathbf{w}$ belonging to the subspace spanned by the columns of $\mathbf{F}$ and an orthogonal component $\mathbf{r}$. Formally this decomposition is written as

$$\mathbf{w} = \mathbf{F}\mathbf{z} + \mathbf{r}. \tag{7.3}$$

Note that $\mathbf{r}$ being orthogonal to the span of $\mathbf{F}$'s columns means formally that $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$.

As we will now see this simple statement is the key to representing fixed basis features more effectively (when used to transform vector valued input for use) with every cost function discussed in this book.

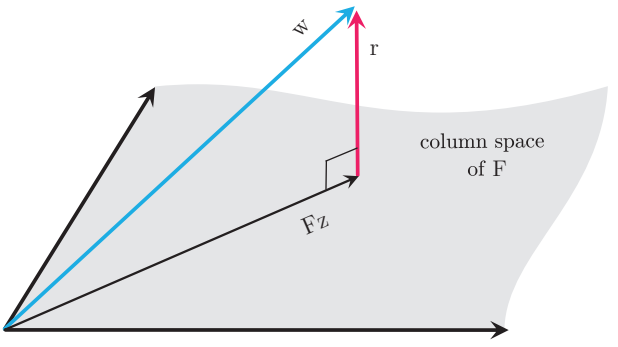

Fig. 7.1 An illustration of the fundamental theorem of linear algebra which states that any vector $\mathbf{w}$ in an M-dimensional space can be decomposed as $\mathbf{w} = \mathbf{F}\mathbf{z} + \mathbf{r}$. Here the vector $\mathbf{F}\mathbf{z}$ belongs in the subspace determined by the columns of the matrix $\mathbf{F}$ and $\mathbf{r}$ is orthogonal to this subspace.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

7.1.2 Kernelizing cost functions

Suppose that we have a dataset of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ where each input $\mathbf{x}_p$ has dimension N. Recall from Section 5.2 that when employing any fixed feature basis we learn proper parameters by minimizing the Least Squares regression cost,

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2, \tag{7.4}$$

where we have used the vector notation $\mathbf{f}_p = \left[\begin{array}{cccc} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots & f_M\left(\mathbf{x}_p\right)\end{array}\right]^T$ to denote the M fixed basis feature transformations of the input $\mathbf{x}_p$. Denote by $\mathbf{F}$ the $M \times P$ matrix $\mathbf{F}$ formed by stacking the vectors $\mathbf{f}_p$ column-wise. Now, employing the fundamental theorem of linear algebra discussed in the previous section we may write $\mathbf{w}$ here as

$$\mathbf{w} = \mathbf{F}\mathbf{z} + \mathbf{r}, \tag{7.5}$$

where $\mathbf{r}$ satisfies $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Plugging this representation of $\mathbf{w}$ back into the cost function then gives

$$\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\left(\mathbf{F}\mathbf{z} + \mathbf{r}\right) - y_p\right)^2 = \sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\mathbf{F}\mathbf{z} - y_p\right)^2. \tag{7.6}$$

Finally, denoting the symmetric matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ (and where $\mathbf{h}_p = \mathbf{F}^T\mathbf{f}_p$ is the pth column of this matrix), referred to as a fixed basis *kernel matrix*, our original cost function becomes equivalently

$$g\left(b, \mathbf{z}\right) = \sum_{p=1}^{P}\left(b + \mathbf{h}_p^T\mathbf{z} - y_p\right)^2. \tag{7.7}$$

Note that we have changed the arguments of the cost function from $g\left(b, \mathbf{w}\right)$ to $g\left(b, \mathbf{z}\right)$ due to our substitution of $\mathbf{w}$. The original problem of minimizing the Least Squares cost may now be written equivalently in this *kernelized form* as

$$\underset{b,\mathbf{z}}{\text{minimize}} \sum_{p=1}^{P}\left(b + \mathbf{h}_p^T\mathbf{z} - y_p\right)^2. \tag{7.8}$$

Using precisely the same argument given here we may *kernelize* all of the cost functions discussed in this book including: the softmax cost/logistic regression classifier, the squared margin-perceptron/soft-margin SVMs, the multiclass softmax cost function, as well as any ℓ_2 regularized version of these models. We show both the original and kernelized forms of these formulae in Table 7.1 for easy reference.

7.1.3 The value of kernelization

The real value of kernelizing any cost function is that for many fixed feature maps, including polynomials and Fourier features, the kernel matrix $\mathbf{H}$ may be constructed *without* first building the matrix $\mathbf{F}$, that is we need not construct it explicitly as $\mathbf{H} = \mathbf{F}^T\mathbf{F}$,

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

Table 7.1 Cost functions and their kernelized versions. Note that the ℓ_2 regularizer can be added to any cost function in the middle column and the resulting kernelized form of the sum will be the sum of the kernelized cost and the kernelized regularizer. For example, the kernelized form of the regularized Least Squares problem $\sum_{p=1}^{P}\left(b+\mathbf{f}_p^T\mathbf{w}-y_p\right)^2+\lambda\left\|\mathbf{w}\right\|_2^2$ is $\sum_{p=1}^{P}\left(b+\mathbf{h}_p^T\mathbf{z}-y_p\right)^2+\lambda\mathbf{z}^T\mathbf{H}\mathbf{z}$.

Cost function	Original version	Kernelized version
Least Squares	$\sum_{p=1}^{P}\left(b+\mathbf{f}_p^T\mathbf{w}-y_p\right)^2$	$\sum_{p=1}^{P}\left(b+\mathbf{h}_p^T\mathbf{z}-y_p\right)^2$
Softmax cost/logistic regression	$\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)}\right)$	$\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{h}_p^T\mathbf{z}\right)}\right)$
Squared margin/ soft-margin SVMs	$\sum_{p=1}^{P}\max^2\left(0,1-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)\right)$	$\sum_{p=1}^{P}\max^2\left(0,1-y_p\left(b+\mathbf{h}_p^T\mathbf{z}\right)\right)$
Multiclass softmax	$\sum_{c=1}^{C}\sum_{p\in\Omega_c}\log\left(1+\sum_{\substack{j=1\\j\neq c}}^{C}e^{\left(b_j-b_c\right)+\mathbf{f}_p^T\left(\mathbf{w}_j-\mathbf{w}_c\right)}\right)$	$\sum_{c=1}^{C}\sum_{p\in\Omega_c}\log\left(1+\sum_{\substack{j=1\\j\neq c}}^{C}e^{\left(b_j-b_c\right)+\mathbf{h}_p^T\left(\mathbf{z}_j-\mathbf{z}_c\right)}\right)$
ℓ_2-regularizer	$\lambda\left\|\mathbf{w}\right\|_2^2$	$\lambda\mathbf{z}^T\mathbf{H}\mathbf{z}$

but this matrix may be constructed entry-wise via simple formulae. In fact, as we will see, thinking about constructing kernel matrices in this way leads to the construction of fixed feature bases by defining the kernel matrix first (that is, not by beginning with an explicit feature transformation). As we see in the next section this can be done for both degree D polynomial and Fourier feature bases, as well as many other fixed maps. This is highly advantageous since recall, as discussed in the introduction to this section, that even with moderate sized input dimension N the dimension of a fixed feature transformation M will likely be gargantuan, so large that we may not even be able to store the matrix $\mathbf{F}$ let alone compute with it.

However, note that the non-bias optimization variable from the original to kernelized form has changed from $\mathbf{w}$, which had dimension M in Equation (7.4), to $\mathbf{z}$, which has dimension P in the kernelized version shown in Equation (7.7). This is precisely how the dimension of the non-bias optimization variable changes with kernelized cost functions as well, like those shown in Table 7.1.

While it is true that for large datasets (that is large values of P, e.g., in the thousands or tens of thousands) the minimization of a kernelized cost function becomes more challenging, the main obstacle is storing the $P\times P$ kernel matrix itself, which for large values of P is difficult or even impossible to do completely. For example, with $P=10\,000$ the

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

corresponding kernel matrix will be of size $10\,000 \times 10\,000$, with 10^8 values to store, far more than a modern computer can store all at once. Moreover, the amount of computation required to perform, e.g. gradient descent, grows dramatically with the size of a kernel matrix due to its explosive size.

Common ways of dealing with these issues for large datasets include: 1) using advanced first order methods such as stochastic gradient descent, discussed in Chapter 8, so that only a small number of the kernelized points are dealt with at a time when optimizing; 2) reducing the dimension of data using techniques like those discussed in Chapter 9 and hence avoiding the need for kernelized versions of fixed bases; 3) using the explicit structure of certain problems (see e.g., [22, 49]); and 4) employing the tools from function approximation to avoid explicit construction of the kernel matrix [64, 68, 69].

7.1.4 Examples of kernels

Here we present a list of examples of kernels for popular fixed feature transformations that may be built without first constructing the explicit feature transformation itself. While these are the most commonly used kernels in practice, the reader can see e.g., [20, 51] for a more exhaustive list of kernels and their properties.

Example 7.1 The polynomial kernel

Consider the following second degree polynomial mapping from $N = 2$ to $M = 5$ dimensional space given by

$$\mathbf{f}\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} \sqrt{2}x_1 & \sqrt{2}x_2 & x_1^2 & \sqrt{2}x_1x_2 & x_2^2 \end{bmatrix}^T. \tag{7.9}$$

This is entirely equivalent to a standard degree 2 polynomial, as the $\sqrt{2}$ attached to several of the terms can be absorbed by their associated weights when taking the corresponding weighted sum $\sum_{m=1}^{5} f_m(\mathbf{x})\, w_m$. Denoting briefly by $\mathbf{u} = \mathbf{x}_i$ and $\mathbf{v} = \mathbf{x}_j$ the ith and jth input data points respectively, the (i,j)th element of the kernel matrix for a degree 2 polynomial $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ may be written as

$$\begin{aligned} \mathbf{H}_{ij} &= \begin{bmatrix} \sqrt{2}u_1 & \sqrt{2}u_2 & u_1^2 & \sqrt{2}u_1u_2 & u_2^2 \end{bmatrix} \begin{bmatrix} \sqrt{2}v_1 \\ \sqrt{2}v_2 \\ v_1^2 \\ \sqrt{2}v_1v_2 \\ v_2^2 \end{bmatrix} \\ &= \left(1 + 2u_1v_1 + 2u_2v_2 + u_1^2v_1^2 + 2u_1u_2v_1v_2 + u_2^2v_2^2\right) - 1 \\ &= (1 + u_1v_1 + u_2v_2)^2 - 1 = \left(1 + \mathbf{u}^T\mathbf{v}\right)^2 - 1. \end{aligned} \tag{7.10}$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

In short, the *polynomial kernel* matrix $\mathbf{H}$ may be built without first constructing the explicit features in Equation (7.9), and may be simply defined entry-wise as

$$\mathbf{H}_{ij} = \left(1 + \mathbf{x}_i^T \mathbf{x}_j\right)^2 - 1. \tag{7.11}$$

Again note that with the polynomial kernel defined above we only require access to the original input data, not the explicit polynomial features themselves.

Although the kernel construction rule in (7.11) was derived specifically for $N = 2$ and a degree two polynomial, one can show that a polynomial kernel can be defined entry-wise for general N and degree D analogously as

$$\mathbf{H}_{ij} = \left(1 + \mathbf{x}_i^T \mathbf{x}_j\right)^D - 1. \tag{7.12}$$

Example 7.2 The Fourier kernel

Recall from Example 5.1 the degree D Fourier feature transformation from $N = 1$ to $M = 2D$ dimensional space, with corresponding transformed feature vector given as

$$\mathbf{f}_p = \left[\begin{array}{cccc} \sqrt{2}\cos\left(2\pi x_p\right) & \sqrt{2}\sin\left(2\pi x_p\right) & \cdots & \sqrt{2}\cos\left(2D\pi x_p\right) \quad \sqrt{2}\sin\left(2D\pi x_p\right) \end{array}\right]^T. \tag{7.13}$$

For a dataset of P points the corresponding (i, j)th element of the corresponding kernel matrix $\mathbf{H}$ can be written as

$$\mathbf{H}_{ij} = \mathbf{f}_i^T \mathbf{f}_j = 2\sum_{m=1}^{D} \cos\left(2\pi m x_i\right)\cos\left(2\pi m x_j\right) + \sin\left(2\pi m x_i\right)\sin\left(2\pi m x_j\right). \tag{7.14}$$

Using trigonometric identities one can show (see Section 7.5.2) that this may equivalently be written as

$$\mathbf{H}_{ij} = \frac{\sin\left(\left(2D+1\right)\pi\left(x_i - x_j\right)\right)}{\sin\left(\pi\left(x_i - x_j\right)\right)} - 1. \tag{7.15}$$

Note that whenever $x_i - x_j$ is integer valued the term $\frac{\sin\left(\left(2D+1\right)\pi\left(x_i - x_j\right)\right)}{\sin\left(\pi\left(x_i - x_j\right)\right)}$ is not technically defined. In these cases it is simply replaced by its associated limit which, regardless of the integer value $x_i - x_j$, is always equal to $2D + 1$ meaning that $\mathbf{H}_{ij} = 2D$.

Moreover, for general N-dimensional input the corresponding kernel can be written similarly entry-wise as

$$\mathbf{H}_{ij} = \prod_{n=1}^{N} \frac{\sin\left(\left(2D+1\right)\pi\left(x_{in} - x_{jn}\right)\right)}{\sin\left(\pi\left(x_{in} - x_{jn}\right)\right)} - 1. \tag{7.16}$$

As with the 1-dimensional version, whenever $x_{in} - x_{jn}$ is integer valued the associated term $\frac{\sin\left(\left(2D+1\right)\pi\left(x_{in} - x_{jn}\right)\right)}{\sin\left(\pi\left(x_{in} - x_{jn}\right)\right)}$ in the product is replaced by its limit which, regardless of the value of $x_{in} - x_{jn}$, is always equal to $2D + 1$. See Section 7.5.3 for further details.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

With this formula we can compute the degree D Fourier features for arbitrary N-dimensional input vectors without calculating the enormous number (see footnote 1) of basis features explicitly.

Example 7.3 Kernel representation of radial basis function (RBF) features

Another popular choice of kernel is the *radial basis function* (RBF) kernel which is typically defined explicitly as a kernel matrix over the input data as

$$\mathbf{H}_{ij} = e^{-\beta \|\mathbf{x}_i - \mathbf{x}_j\|_2^2}. \tag{7.17}$$

Here the kernel parameter β is tuned to the data in practice via cross-validation.

While the RBF kernel is typically defined directly as above, it can be traced back to an explicit fixed feature basis as with the polynomial and Fourier kernels, i.e., we have that

$$\mathbf{H}_{ij} = \mathbf{f}_i^T \mathbf{f}_j, \tag{7.18}$$

where $\mathbf{f}_i$ is the fixed feature transformation of the input $\mathbf{x}_i$ based on a fixed basis. While the length of a feature transformation corresponding to a degree D polynomial/Fourier kernel matrix can be extremely large (as discussed in the introduction to this section), with the RBF kernel the associated feature transformation is always *infinite* dimensional. For example, when $N = 1$ the feature vector $\mathbf{f}_i$ takes the form $\mathbf{f}_i = \begin{bmatrix} f_1(x_i) & f_2(x_i) & f_3(x_i) & \cdots \end{bmatrix}^T$, where the mth fixed basis feature is defined as

$$f_m(x_i) = e^{-\beta x_i^2} \sqrt{\frac{(2\beta)^{m-1}}{(m-1)!}} x_i^{m-1} \quad \text{for all } m \geq 1. \tag{7.19}$$

When $N > 1$ the corresponding feature vector takes on an analogous form (and is also infinite in length), but regardless of the input dimension it would be impossible to even construct and store a single $\mathbf{f}_i$ let alone such transformations of the entire dataset.

7.1.5 Kernels as similarity matrices

The polynomial, Fourier, and RBF kernel matrices introduced earlier are all *similarity matrices*, essentially encoding how close or similar a collection of data points are to one another, with points in proximity to one another receiving a high value and those far apart receiving a low value. In this sense all three kernels discussed here, and hence all three corresponding fixed feature bases, define some kind of similarity between data points $\mathbf{x}_i$ and $\mathbf{x}_j$ from different geometric perspectives.

In Fig. 7.2 we compare these three kernels geometrically by fixing a point $\mathbf{x}_p = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}^T$ and plotting $\mathbf{H}(\mathbf{x}, \mathbf{x}_p)$ over the range $\mathbf{x} \in [0, 1]^2$, producing a color-coded surface showing how each kernel treats points near $\mathbf{x}_p$. Analyzing this figure we can judge more generally how the three kernels define "similarity" between points.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

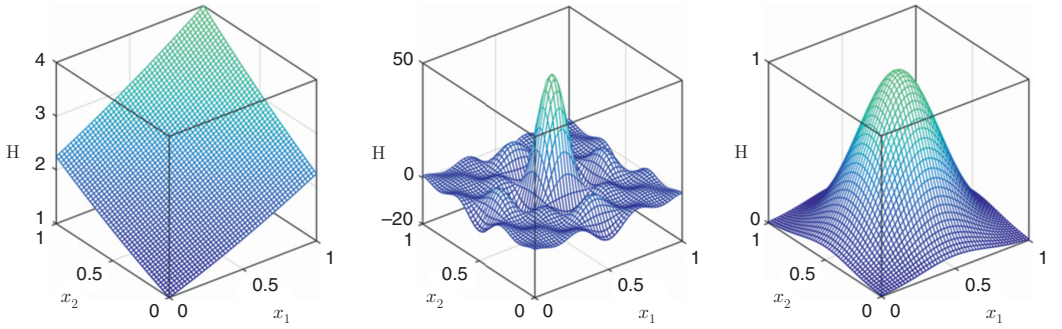

Fig. 7.2 Surfaces generated by polynomial, Fourier, and RBF kernels centered at $\mathbf{x}_p = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}^T$ with the surfaces color-coded based on their similarity to $\mathbf{x}_p$. (left panel) A degree 2 polynomial kernel, (middle panel) degree 3 Fourier kernel, and (right panel) RBF kernel with $\beta = 10$. See text for further details.

Firstly, we can see that a polynomial kernel treats data points $\mathbf{x}_i$ and $\mathbf{x}_j$ similarly if their inner product is high or, in other words, they highly correlate with each other. Likewise the points are treated as dissimilar when they are orthogonal to one another. On the other hand, the Fourier kernel treats points as similar if they lie close together, but their similarity differs like a "sinc" function as their distance from each other grows. Finally an RBF kernel provides a smooth similarity between points. If they are close to each other in a Euclidean sense they are highly similar; however, once the distance between them passes a certain threshold they are deemed rapidly dissimilar.

7.2 The backpropagation algorithm

In this section we provide details for applying gradient descent, commonly referred to as the *backpropagation algorithm*, to any cost function employing a multilayer neural network feature basis. The term "backpropagation" is often used because, as we will see, there is a natural movement or propagation of computation in calculating the gradient of such a cost function *backward* through a sum of neural network basis features. However, one should not think of this as somehow a special version of gradient descent, it is just the standard gradient descent procedure we have used throughout the text applied to a more complicated (cost) function.

While computing the gradient of such a function only requires the use of careful bookkeeping as well as repeated use of the chain rule, these calculations can easily be incorrect because of human error due to their tedious nature. Because of this we provide explicit calculations for general two and three layer hidden networks, and will write them assuming arbitrary cost and activation functions. Since there is no useful compact formula to express the derivatives associated with an arbitrary layered neural network, the reader can extend the pattern for computing two and three layer networks shown here if employing deeper networks.

Variations of gradient descent, including stochastic gradient descent as well as gradient descent with momentum (see Sections 8.3 and 7.2.3 for further details), are

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

also commonly used to minimize a cost function employing neural network features as they tend to speed up convergence in practice.

7.2.1 Computing the gradient of a two layer network cost function

Let g be any cost function for regression or classification described in this book. Note that, over a dataset of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, each can be decomposed over the P points as

$$g = \sum_{p=1}^{P} h\left(b + \mathbf{x}_p^T \mathbf{w}\right); \tag{7.20}$$

e.g., if g is the Least Squares cost for regression or the softmax cost for classification then $h\left(b + \mathbf{x}_p^T \mathbf{w}\right) = \left(b + \mathbf{x}_p^T \mathbf{w} - y_p\right)^2$ and $h\left(b + \mathbf{x}_p^T \mathbf{w}\right) = \log\left(1 + e^{-y_p\left(b + \mathbf{x}_p^T \mathbf{w}\right)}\right)$ respectively. In what follows we will compute derivatives of $h\left(b + \mathbf{x}_p^T \mathbf{w}\right)$, which may then be added up to give corresponding derivatives of g.

Substituting an M_2 two-layer neural network feature $\mathbf{f}_p$ map of $\mathbf{x}_p$, whose mth coordinate takes the form

$$f_m\left(\mathbf{x}_p\right) = a\left(c_m^{(1)} + \sum_{m_2=1}^{M_2} a\left(c_{m_2}^{(2)} + \sum_{n=1}^{N} x_{p,n} v_{n,m_2}^{(2)}\right) v_{m_2,m}^{(1)}\right), \tag{7.21}$$

or more compactly all together we can write

$$\mathbf{f}_p = a\left(\mathbf{c}^{(1)} + \mathbf{V}^{T(1)} a\left(\mathbf{c}^{(2)} + \mathbf{V}^{T(2)} \mathbf{x}_p\right)\right), \tag{7.22}$$

where we slightly abuse notation and say that the activation $a(\cdot)$ applies the function to each coordinate of its input. With this network map, our cost summand is given as $h\left(b + \mathbf{f}_p^T \mathbf{w}\right)$. Here we have stacked the parameters of the first layer into the $M_1 \times 1$ vector $\mathbf{c}^{(1)}$ and $M_2 \times M_1$ matrix $\mathbf{V}^{(1)}$, and those of the second layer into the $M_2 \times 1$ vector $\mathbf{c}^{(2)}$ and $N \times M_2$ matrix $\mathbf{V}^{(2)}$ respectively.

Because we will need to employ the chain rule many times, in order to more effectively compute the gradient of this summand it will be helpful to introduce notation for the argument or *residual* of each layer of the network, as well as the result of each layer after passing through the activation function. Firstly, we will write the arguments at each layer recursively as

$$\begin{aligned} r &= b + \mathbf{w}^T a\left(\mathbf{r}^{(1)}\right) \\ \mathbf{r}^{(1)} &= \mathbf{c}^{(1)} + \mathbf{V}^{T(1)} a\left(\mathbf{r}^{(2)}\right) \\ \mathbf{r}^{(2)} &= \mathbf{c}^{(2)} + \mathbf{V}^{T(2)} \mathbf{x}_p. \end{aligned} \tag{7.23}$$

Note that the first argument r is a scalar, while the latter two $\mathbf{r}^{(1)}$ and $\mathbf{r}^{(2)}$ are M_1 and M_2 length vectors respectively. Correspondingly, we can write $\mathbf{a}^{(1)} = a\left(\mathbf{r}^{(1)}\right)$ and $\mathbf{a}^{(2)} = a\left(\mathbf{r}^{(2)}\right)$, the result of the first layer and second layer argument passed through the activation function respectively (note these are also M_1 and M_2 length vectors).

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

With this notation we have in particular $h(r) = h\left(b + \mathbf{f}_p^T \mathbf{w}\right)$, and we can write the derivatives of the parameters $(b, \mathbf{w})$ via the chain rule as

$$\begin{aligned} \frac{\partial h}{\partial b} &= \frac{\partial h}{\partial r}\frac{\partial r}{\partial b} \\ \nabla_{\mathbf{w}} h &= \frac{\partial h}{\partial r}\nabla_{\mathbf{w}} r. \end{aligned} \tag{7.24}$$

Each derivative on the right hand side above may be calculated in closed form, e.g., $\frac{\partial r^{(0)}}{\partial b} = 1$ and $\nabla_{\mathbf{w}} r = \mathbf{a}^{(1)}$, and if $h(t) = \log\left(1 + e^{-t}\right)$ is the softmax cost summand then $\frac{\partial h}{\partial r} = h'(r) = \sigma(-r)$. Computing derivatives of the first and second hidden layers' parameters similarly yields, via applying the chain rule multiple times,

$$\begin{aligned} \frac{\partial h}{\partial c_i^{(1)}} &= \frac{\partial h}{\partial r}\frac{\partial r}{\partial a_i^{(1)}}\frac{\partial a_i^{(1)}}{\partial r_i^{(1)}}\frac{\partial r_i^{(1)}}{\partial c_i^{(1)}} \\ \nabla_{\mathbf{v}_i^{(1)}} h &= \frac{\partial h}{\partial r}\frac{\partial r}{\partial a_i^{(1)}}\frac{\partial a_i^{(1)}}{\partial r_i^{(1)}}\nabla_{\mathbf{v}_i^{(1)}} r_i^{(1)} \\ \frac{\partial h}{\partial c_i^{(2)}} &= \frac{\partial h}{\partial r}\left(\sum_{n_1=1}^{M_1}\frac{\partial r}{\partial a_{n_1}^{(1)}}\frac{\partial a_{n_1}^{(1)}}{\partial r_{n_1}^{(1)}}\frac{\partial r_{n_1}^{(1)}}{\partial a_i^{(2)}}\right)\frac{\partial a_i^{(2)}}{\partial r_i^{(2)}}\frac{\partial r_i^{(2)}}{\partial c_i^{(2)}} \\ \nabla_{\mathbf{v}_i^{(2)}} h &= \frac{\partial h}{\partial r}\left(\sum_{n_1=1}^{L_1}\frac{\partial r}{\partial a_{n_1}^{(1)}}\frac{\partial a_{n_1}^{(1)}}{\partial r_{n_1}^{(1)}}\frac{\partial r_{n_1}^{(1)}}{\partial a_i^{(2)}}\right)\frac{\partial a_i^{(2)}}{\partial r_i^{(2)}}\nabla_{\mathbf{v}_i^{(2)}} r_i^{(2)} \end{aligned} \tag{7.25}$$

Note that due to the decision to denote the residuals in each layer, these derivatives take a predictable form, consisting of repeating pairs of partial derivatives: "partial of a function with respect to its residual" and "partial of the residual with respect to a parameter or the following layer activation." Again each individual derivative on the right hand side of the equalities may be computed in closed form given a choice of cost summand h and activation function a. As we have already seen, $\frac{\partial h}{\partial r} = h'(r)$, and as for the remainder of the derivatives we have

$$\begin{aligned} &\frac{\partial r}{\partial a_i^{(1)}} = w_i \qquad \frac{\partial a_i^{(1)}}{\partial r_i^{(1)}} = a'\left(r_i^{(1)}\right) \qquad \frac{\partial r_i^{(1)}}{\partial c_i^{(1)}} = 1 \qquad \nabla_{\mathbf{v}_i^{(1)}} r_i^{(1)} = \mathbf{a}^{(2)} \\ &\frac{\partial r_{n_1}^{(1)}}{\partial a_i^{(2)}} = v_{n_1,i}^{(2)} \qquad \frac{\partial a_i^{(2)}}{\partial r_i^{(2)}} = a'\left(r_i^{(2)}\right) \qquad \frac{\partial r_i^{(2)}}{\partial c_i^{(2)}} = 1 \qquad \nabla_{\mathbf{v}_i^{(2)}} r_i^{(2)} = \mathbf{x}_p. \end{aligned} \tag{7.26}$$

Note that due to the recursive nature of the arguments, as shown in Equation (7.23), these are typically precomputed (that is, prior to computing derivatives) and, in particular, must be computed in a *forward* manner from inner (i.e., closer to the input) to outer (i.e. farther from the input) layers of the network sequentially. Conversely, as we can see above in Equation (7.25), the propagation of derivative calculations is performed *backwards.* In other words, first the outer layer derivatives in Equation (7.24) are computed, then derivatives of layer one are constructed, and then finally layer two. This pattern, of computing the residuals and derivatives in a forward and backward manner respectively, holds more generally when employing an arbitrary number of layers in a neural network.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

As mentioned in the introduction, gradient descent is often referred to as backpropagation because the residuals having been computed in a forward fashion, we can then compute the output residual r which is propagated backwards as we compute the gradients layer by layer.

7.2.2 Three layer neural network gradient calculations

To reiterate the points made in computing two-layer derivatives as shown previously, we briefly show mirrored results for a three-layer network so that the reader can be more comfortable with the notation, as well as the pattern of partial derivatives. Again to make the use of the chain rule more predictable, we define residuals at each layer of the network (in complete similarity to Equation (7.23)) as

$$\begin{aligned} r &= b + \mathbf{w}^T a\left(\mathbf{r}^{(1)}\right) \\ \mathbf{r}^{(1)} &= \mathbf{c}^{(1)} + \mathbf{V}^{(1)T} a\left(\mathbf{r}^{(2)}\right) \\ \mathbf{r}^{(2)} &= \mathbf{c}^{(2)} + \mathbf{V}^{(2)T} a\left(\mathbf{r}^{(3)}\right) \\ \mathbf{r}^{(3)} &= \mathbf{c}^{(3)} + \mathbf{V}^{(3)T} \mathbf{x}_p \end{aligned} \tag{7.27}$$

where again we slightly abuse notation and say that the activation $a(\cdot)$ applies the function to each coordinate of its input. The forms of the derivatives of the bias and feature weights are given precisely as shown in Equation (7.24), and the first two layer-wise derivatives precisely as shown in Equation (7.25). Just note that the form of the residuals and of the total summand h have changed, since we now have three layers in the network. Using the chain rule the third-layer derivatives can then be computed as

$$\begin{aligned} \frac{\partial h}{\partial c_i^{(3)}} &= \frac{\partial h}{\partial r} \sum_{n_1=1}^{M_1} \frac{\partial r}{\partial a_{n_1}^{(1)}} \frac{\partial a_{n_1}^{(1)}}{\partial r_{n_1}^{(1)}} \left(\sum_{n_2=1}^{M_2} \frac{\partial r_{n_1}^{(1)}}{\partial a_{n_2}^{(2)}} \frac{\partial a_{n_2}^{(2)}}{\partial r_{n_2}^{(2)}} \frac{\partial r_{n_2}^{(2)}}{\partial a_i^{(3)}} \right) \frac{\partial a_i^{(3)}}{\partial r_i^{(3)}} \frac{\partial r_i^{(3)}}{\partial c_i^{(3)}} \\ \nabla_{\mathbf{v}_i^{(3)}} h &= \frac{\partial h}{\partial r} \sum_{n_1=1}^{L_1} \frac{\partial r}{\partial a_{n_1}^{(1)}} \frac{\partial a_{n_1}^{(1)}}{\partial r_{n_1}^{(1)}} \left(\sum_{n_2=1}^{L_2} \frac{\partial r_{n_1}^{(1)}}{\partial a_{n_2}^{(2)}} \frac{\partial a_{n_2}^{(2)}}{\partial r_{n_2}^{(2)}} \frac{\partial r_{n_2}^{(2)}}{\partial a_i^{(3)}} \right) \frac{\partial a_i^{(3)}}{\partial r_i^{(3)}} \nabla_{\mathbf{v}_i^{(3)}} r_i^{(3)} \end{aligned} \tag{7.28}$$

where, as with the previous layers, all derivatives on the right hand side can be computed in closed form. Also, as with the previous two-layer case, again the residuals are computed first in a forward manner (from the inner to outer layer of the network), while the derivatives are naturally computed in a backward manner given their structure.

Example 7.4 Comparison of different networks on a toy classification dataset

Here we give a simple example comparing neural network feature maps with one, two, and three hidden layers on the toy classification dataset shown in the middle panel of Fig. 6.9. In particular we use $M_1 = M_2 = M_3 = 10$ units in each hidden layer of each network respectively. In Fig. 7.3 we show the objective value per iteration of gradient descent, for illustrative purposes showing iterations 10 through 50. In this figure we

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

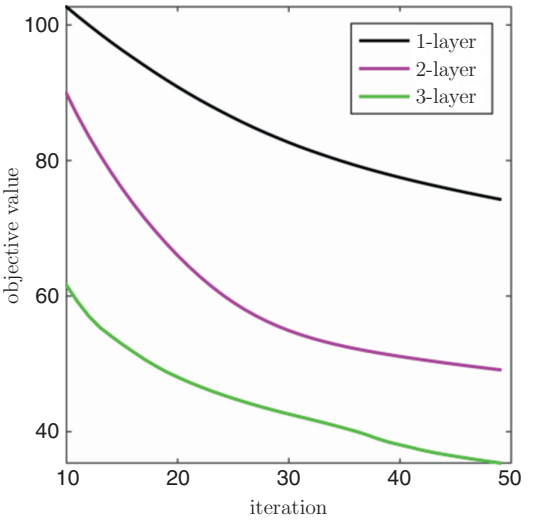

Fig. 7.3 Objective value resulting from the first 50 iterations of gradient descent applied to minimizing the softmax cost employing one/two/three hidden layer neural network features for the dataset shown in the middle panel of Fig. 6.9 (see text for further details). This plot simply reflects the fact that as we increase the number of layers of a neural network each basis feature becomes more flexible, allowing for faster fitting to nonlinearly separable data. However, as described in the context of both regression and classification, overfitting becomes more of a potential problem as we increase the flexibility of a feature map and thus cross-validation is absolutely required when using deep net features.

can see that the deeper networks provide stronger fits faster to the dataset, with the three layer network providing the best progress throughout the range of iterations shown. This is true more generally since, as described in Section 5.1.4, as we increase the number of layers in a network each basis feature becomes more flexible. However, as discussed in Sections 5.3 and 6.4, such flexibility in representation can lead deeper networks to overfit a dataset and therefore cross-validation must be employed with deep networks (typically performed via the regularization approach detailed in Section 7.3).

7.2.3 Gradient descent with momentum

A practical problem that occurs in minimizing some cost functions g, especially in higher dimensions, is that the gradient descent steps tend to *zig-zag* towards a solution as illustrated in two dimensions in Fig. 7.4. In this hypothetical example we illustrate the use of an adaptive step length rule (see Section 8.2) in order to exaggerate the real problem in higher dimensions.

This problem motivates the concept of an old and simple heuristic known as the *momentum term*, which is added to the standard gradient descent step applied to both convex and non-convex functions. The momentum term is a simple weighted difference

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

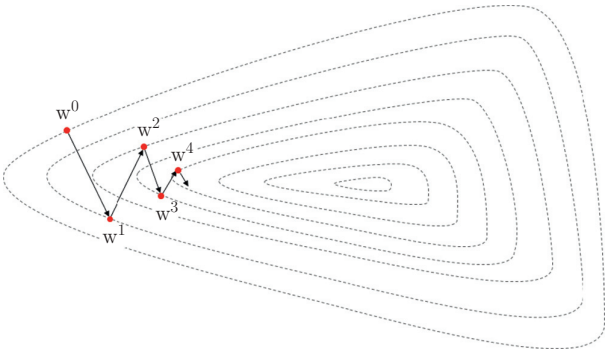

Fig. 7.4 A figurative illustration of gradient steps toward the minimum of a function in two dimensions. Note that the gradient step directions are perpendicular to the contours of the surface shown with dashed ellipses.

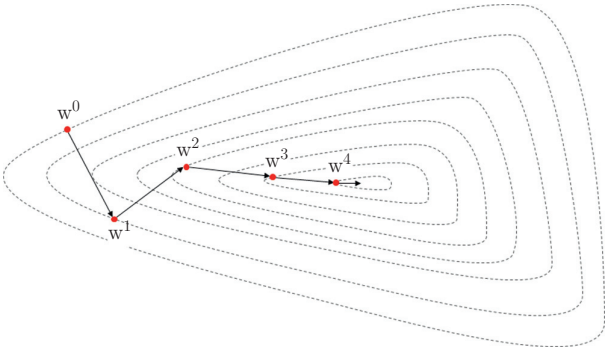

Fig. 7.5 A figurative illustration of momentum-adjusted gradient descent steps toward the minimum of the same function shown in Fig. 7.4. The addition of the momentum term averages out the zig-zagging inherent in standard gradient descent steps.

of the subsequent kth and $(k-1)$th gradient steps, i.e., $\beta\left(\mathbf{w}^k - \mathbf{w}^{k-1}\right)$ for some $\beta > 0$, and is designed to even out the zig-zagging effect of the gradient descent iterates. Hypothetical steps from a gradient descent scheme with momentum are illustrated in Fig. 7.5.

Adding this term to the $(k+1)$th gradient step gives the combined update of

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha_k \nabla g\left(\mathbf{w}^k\right) + \beta\left(\mathbf{w}^k - \mathbf{w}^{k-1}\right), \tag{7.29}$$

where β can be adjusted as well at each iteration if desired. When tuned properly the adjusted gradient descent step with momentum is known empirically to significantly improve the convergence of gradient descent (see e.g., [66]).

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

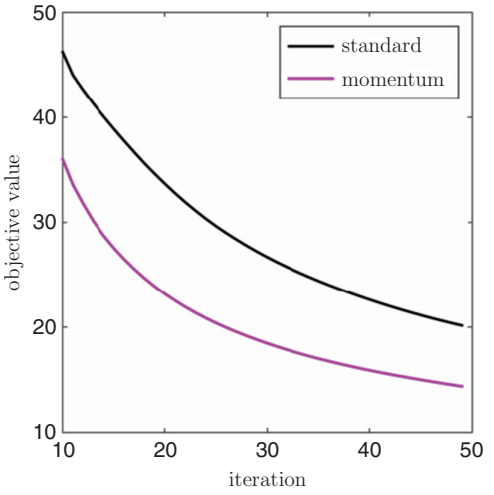

Fig. 7.6 The cost function value of the first $k = 10$–50 iterations of standard and momentum gradient descent procedures (shown in black and magenta respectively) on a classification dataset, averaged over five runs. By tuning the momentum parameter we can create faster converging gradient descent schemes. See text for further details.

Example 7.5 Multilayer neural network example

In this example we use the softmax cost with a three-layer neural network feature map, using $M_1 = M_2 = M_3 = 10$ hidden units in all three layers and the rectified linear unit activation function, and the classification dataset first shown in the left panel of Fig. 4.3. We compare standard gradient descent to the momentum version shown in Equation (7.29) by averaging the results of five runs of each algorithm (using the same random initialization for each run for both algorithms).

Shown in Fig. 7.6 is the average objective function value for the first 50 iterations of both algorithms. β has been tuned[2] such that the momentum gradient descent runs converge significantly faster than do the standard runs.

7.3 Cross-validation via ℓ_2 regularization

In this section we describe the concept of cross-validation via ℓ_2 regularization, a variation of the direct comparison method for cross-validation described in previous chapters. This method provides a much more useful framework for performing cross-validation in a variety of important circumstances. These include:

- **Fixed/kernelized feature bases of high dimensional input.** The difference in dimension M, or the number of basis elements, of fixed feature representations having

[2] For both we used a constant fixed step size of $\alpha = 10^{-3}$ and a momentum weight of $\beta = 0.5$ for all runs.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

subsequent degrees D and $D+1$ becomes extremely large as we increase the input dimension N. Therefore comparison of fixed features of various degrees becomes far too coarse: because there is an increasing number of possible feature configurations between degrees D and $D+1$ we do not try.[3] Furthermore, by kernelizing such fixed features we lose direct access to individual elements of the original basis, making it impossible to make comparisons at a finer scale. Moreover for fixed bases defined explicitly at the level of a kernel (e.g., the RBF kernel in Example 7.3), regardless of input dimension N, we cannot even practically compare different length feature vectors, as they are all infinite in dimension.

- **Multilayer neural networks.** As the number of layers/activation units of a neural network feature basis is increased each corresponding entry of the feature vector/basis element used, having many layers of internal parameters, becomes increasingly flexible. This makes simply testing the effectiveness of M versus $M+1$ dimensional neural network features increasingly coarse, meaning again that there is an increasing number of feature configurations we do not test, as the number of layers/activation units employed is increased.
- **Multiclass softmax classification.** Both of the previously described problems are exacerbated in the case of multiclass classification, particularly when using the multiclass softmax classifier.

We begin by discussing regularized cross-validation with regression followed by a mirrored section for classification. Note that throughout the remainder of this section we will work with P input/output pairs $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, where the input $\mathbf{x}_p$ is of dimension N and y_p are either continuous in the case of regression, or discrete in the case of classification. Further we will assume that either M-dimensional fixed or neural network basis features with arbitrary number of layers have been taken of each input $\mathbf{x}_p$, denoted as $\mathbf{f}_p = \begin{bmatrix} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots & f_M\left(\mathbf{x}_p\right) \end{bmatrix}^T$, and Θ is the set of possible internal parameters of the feature basis elements, which is empty in the case of fixed/kernelized features.

7.3.1 ℓ_2 regularization and cross-validation

Regularized approaches to cross-validation can be understood as following from the same simple observation that motivated the original direct approach described in Sections 5.3 and 6.4. However, instead of trying to determine the proper number M of basis features in order to avoid the pitfall of overfitting, the regularization approach first fixes M at a reasonably high value and adds a second term (the regularizer) to the associated cost function. This additional regularizer constrains the weights, prohibiting the cost function itself from achieving too small values (i.e., creating an overfitting model) over the entire dataset.

[3] Recall that, for example, the number of non-constant basis elements in a degree D polynomial is given as $M = \frac{(N+D)!}{D!N!} - 1$. This means, for example, that with $N = 500$ dimensional input there are 20 958 500 more basis features in a degree $D = 3$ than a degree $D = 2$ polynomial.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

While many choices of regularizer function are available, by far the most commonly used in practice is the squared ℓ_2 norm of the weights, which we have seen previously used to "convexify" non-convex functions (see Section 3.3.2) in order to aid in their minimization, as well as a mathematical way of encoding the margin length in the soft-margin SVM classifier (see Section 4.3.3).

7.3.2 Regularized k-fold cross-validation for regression

Formally to regularize the Least Squares cost function using the approach described previously, we add to it a weighted version of the squared ℓ_2 norm of all the weights, giving the ℓ_2 regularized Least Squares cost function

$$g\left(b, \mathbf{w}, \Theta\right) = \sum_{p=1}^{P} \left(b + \mathbf{f}_p^T \mathbf{w} - y_p\right)^2 + \lambda \left(\|\mathbf{w}\|_2^2 + \sum_{\theta \in \Theta} \theta^2\right). \tag{7.30}$$

Note that since the definition of the squared ℓ_2 norm gives $\|\mathbf{a}\|_2^2 = \sum_{m=1}^{M} a_m^2$, the final term $\sum_{\theta \in \Theta} \theta^2$ is equivalent to first forming a single long column vector containing all internal parameters in Θ, and then taking the squared ℓ_2 norm of the result. Also note that we do not regularize the bias b since we are only concerned with mitigating the impact of the features themselves on our final model. Finally note that when using a kernelized fixed map, as described in Section 7.1.2, the parameter set is empty and the above is written as

$$g\left(b, \mathbf{z}\right) = \sum_{p=1}^{P} \left(b + \mathbf{h}_p^T \mathbf{z} - y_p\right)^2 + \lambda \mathbf{z}^T \mathbf{H} \mathbf{z}. \tag{7.31}$$

In either case, the parameter $\lambda \geq 0$ controls the strength of each term, the Least Squares cost, and regularizer, in the final sum. For example, if $\lambda = 0$ we have our original cost again, but if on the other hand λ is set very high then the regularizer drowns out the cost function.

Now, the precise value of λ is determined by employing the cross-validation framework described in Section 5.3, where instead of trying various values for M we (having fixed the feature dimension at M) try a discrete set of values in some range $\lambda \in [\lambda_{\min}, \lambda_{\max}]$. In other words, we split our dataset of P points into k non-overlapping portions, for each fold by forming $k - 1$ portions of the data into a training set and leaving a single (distinct for each fold) portion as a test set.

Fitting to a single fold training set for one choice of λ, we minimize (7.30) over the training set by solving

$$\underset{b, \mathbf{w}, \Theta}{\text{minimize}} \sum_{p \in \Omega_{\text{train}}} \left(b + \mathbf{f}_p^T \mathbf{w} - y_p\right)^2 + \lambda \left(\|\mathbf{w}\|_2^2 + \sum_{\theta \in \Theta} \theta^2\right), \tag{7.32}$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

where we once again denote via index sets those points belonging to this fold's training and testing sets as

$$\begin{aligned}\Omega_{\text{train}} &= \left\{p \,|\, \left(\mathbf{x}_p,\, y_p\right) \text{ belongs to the training set}\right\} \\ \Omega_{\text{test}} &= \left\{p \,|\, \left(\mathbf{x}_p,\, y_p\right) \text{ belongs to the testing set}\right\}.\end{aligned} \tag{7.33}$$

Denoting an optimal solution to the above problem as $\left(b^\star_\lambda, \mathbf{w}^\star_\lambda, \Theta^\star_\lambda\right)$, we then compute the training and testing errors on a single fold for one choice of λ as

$$\begin{aligned}\text{Training error} &= \frac{1}{|\Omega_{\text{train}}|}\sum_{p\in\Omega_{\text{train}}}\left(b^\star_\lambda + \mathbf{f}_p^T\mathbf{w}^\star_\lambda - y_p\right)^2 \\ \text{Testing error} &= \frac{1}{|\Omega_{\text{test}}|}\sum_{p\in\Omega_{\text{test}}}\left(b^\star_\lambda + \mathbf{f}_p^T\mathbf{w}^\star_\lambda - y_p\right)^2,\end{aligned} \tag{7.34}$$

where the notation $|\Omega_{\text{train}}|$ and $|\Omega_{\text{test}}|$ denotes the cardinality or number of indices in the training and testing sets respectively.

To perform k-fold cross-validation we then execute these calculations over all k folds and average the results for each value of λ. We then pick the value $\lambda^\star$ providing the lowest average testing error, and fit the final model to the entire dataset by solving

$$\underset{b,\,\mathbf{w},\Theta}{\text{minimize}}\; \sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\mathbf{w} - y_p\right)^2 + \lambda^\star\left(\|\mathbf{w}\|_2^2 + \sum_{\theta\in\Theta}\theta^2\right) \tag{7.35}$$

7.3.3 Regularized cross-validation for classification

As with the case of regression, we may regularize any cost function like e.g., the softmax cost with the ℓ_2 norm squared of all feature weights as

$$g\left(b, \mathbf{w}, \Theta\right) = \sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)}\right) + \lambda\left(\|\mathbf{w}\|_2^2 + \sum_{\theta\in\Theta}\theta^2\right), \tag{7.36}$$

where again if a kernelized fixed feature map is used we may write the above as

$$g\left(b, \mathbf{z}\right) = \sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)}\right) + \lambda\mathbf{z}^T\mathbf{H}\mathbf{z}. \tag{7.37}$$

Following the same format as with regression, to determine a proper value of $\lambda \geq 0$ we perform k-fold cross-validation for a discrete set of values in a range of $\lambda \in [\lambda_{\min},\, \lambda_{\max}]$ and choose the value providing the lowest average testing error.

To determine the training/testing error for one value of λ on a single fold we first fit to one fold's training set, solving

$$\underset{b,\,\mathbf{w},\Theta}{\text{minimize}}\; \sum_{p\in\Omega_{\text{train}}}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)}\right) + \lambda\left(\|\mathbf{w}\|_2^2 + \sum_{\theta\in\Theta}\theta^2\right), \tag{7.38}$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

and computing the training and testing errors on this fold:

$$\begin{aligned}\text{Training error} &= \frac{1}{|\Omega_{\text{train}}|}\sum_{p\in\Omega_{\text{train}}}\max\left(0,\ \text{sign}\left(-y_p\left(b_\lambda^\star + \mathbf{x}_p^T\mathbf{w}_\lambda^\star\right)\right)\right)\\ \text{Testing error} &= \frac{1}{|\Omega_{\text{test}}|}\sum_{p\in\Omega_{\text{test}}}\max\left(0,\ \text{sign}\left(-y_p\left(b_\lambda^\star + \mathbf{x}_p^T\mathbf{w}_\lambda^\star\right)\right)\right)\end{aligned} \tag{7.39}$$

where once again Ω_{train} and Ω_{test} denote the training and testing sets on this fold. Averaging these values over all k folds we then pick the $\lambda^\star$ with lowest average testing error, and fit the corresponding model to the entire dataset as

$$\underset{b,\,\mathbf{w},\Theta}{\text{minimize}}\ \sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b+\mathbf{f}_p^T\mathbf{w}\right)}\right) + \lambda^\star\left(\|\mathbf{w}\|_2^2 + \sum_{\theta\in\Theta}\theta^2\right). \tag{7.40}$$

7.4 Summary

In the first section of this chapter we described how kernel representations are used to overcome the serious scaling issue of fixed feature bases with vector valued input. Furthermore, we have seen how new kinds of fixed bases can be defined directly through a kernelized representation. We have also showed how every machine learning cost function discussed in this book may be kernelized (permitting the use of any fixed basis kernel).

In Section 7.2 we gave careful derivations of the gradient when using multilayer network features. As we saw, this requires very careful bookkeeping, as well as repeated use of the chain rule.

In the final section we detailed a variation of cross-validation based on the ℓ_2 regularizer. This approach is founded on the same principles that led to the direct approach described in previous chapters, but here k-fold cross-validation is used to determine the proper value of the penalty parameter on the regularizer (instead of determining the best number of basis features to use). This regularized approach to cross-validation is a much more effective way of properly fitting regression/classification models employing either kernelized fixed feature or deep net feature bases.

7.5 Further kernel calculations

7.5.1 Kernelizing various cost functions

Here we derive the kernelization of the three core classification models: softmax cost/logistic regression, soft-margin SVMs, and the multiclass softmax classifier. Although we will only describe how to kernelize the ℓ_2 regularizer along with the SVM model, precisely the same argument can be made in combination with any other machine learning model shown in Table 7.1. As with the derivation for Least Squares regression shown in Section 7.1.2, here the main tool for kernelizing these models is again the fundamental theorem of linear algebra described in Section 7.1.1.

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

Throughout this section we will suppose that an arbitrary M-dimensional fixed feature vector has been taken of the input of P points $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ giving feature vectors $\mathbf{f}_p = \begin{bmatrix} f_1\left(\mathbf{x}_p\right) & f_2\left(\mathbf{x}_p\right) & \cdots & f_M\left(\mathbf{x}_p\right) \end{bmatrix}^T$ for each $\mathbf{x}_p$.

Example 7.6 Kernelizing two-class softmax classification/logistic regression

Recall that the softmax perceptron cost function used with fixed feature mapped input is given as

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)}\right). \tag{7.41}$$

Using the fundamental theorem of linear algebra for any $\mathbf{w}$ we can then write $\mathbf{w} = \mathbf{F}\mathbf{z} + \mathbf{r}$ where $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Making this substitution into the above and simplifying gives

$$g\left(b, \mathbf{z}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\mathbf{F}\mathbf{z}\right)}\right), \tag{7.42}$$

and denoting the kernel matrix $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ (where $\mathbf{h}_p = \mathbf{F}^T\mathbf{f}_p$ is the pth column of $\mathbf{H}$) we can then write the above in kernelized form as

$$g\left(b, \mathbf{z}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)}\right). \tag{7.43}$$

This is the kernelized form of logistic regression shown in Table 7.1.

Example 7.7 Kernelizing soft-margin SVM/regularized margin-perceptron

Recall the soft-margin SVM cost/regularized margin-perceptron cost:

$$g\left(b, \mathbf{w}\right) = \sum_{p=1}^{P} \max^2\left(0, 1 - y_p\left(b + \mathbf{f}_p^T\mathbf{w}\right)\right) + \lambda \left\|\mathbf{w}\right\|_2^2 \tag{7.44}$$

Applying the fundamental theorem of linear algebra we may then write $\mathbf{w}$ as $\mathbf{w} = \mathbf{F}\mathbf{z} + \mathbf{r}$ where $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Substituting into the cost and noting that $\mathbf{w}^T\mathbf{w} = \left(\mathbf{F}\mathbf{z} + \mathbf{r}\right)^T\left(\mathbf{F}\mathbf{z} + \mathbf{r}\right) = \mathbf{z}^T\mathbf{F}^T\mathbf{F}\mathbf{z} + \mathbf{r}^T\mathbf{r} = \mathbf{z}^T\mathbf{H}\mathbf{z} + \left\|\mathbf{r}\right\|_2^2$, denoting $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ as the kernel matrix we may rewrite the above equivalently as

$$g\left(b, \mathbf{z}, \mathbf{r}\right) = \sum_{p=1}^{P} \max^2\left(0, 1 - y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)\right) + \lambda\mathbf{z}^T\mathbf{H}\mathbf{z} + \lambda \left\|\mathbf{r}\right\|_2^2. \tag{7.45}$$

Note that since we are aiming to minimize the quantity above over $\left(b, \mathbf{z}, \mathbf{r}\right)$, and since the only term with $\mathbf{r}$ remaining is $\left\|\mathbf{r}\right\|_2^2$, the optimal value of $\mathbf{r}$ is zero, for otherwise the

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

value of the cost function would be larger than necessary. Therefore we can ignore $\mathbf{r}$ and write the cost function above in kernelized form as

$$g(b,\mathbf{z}) = \sum_{p=1}^{P} \max^2\left(0, 1 - y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)\right) + \lambda \mathbf{z}^T\mathbf{H}\mathbf{z}, \tag{7.46}$$

as originally shown in Table 7.1.

Example 7.8 Kernelizing the multiclass softmax loss

Recall that the multiclass softmax cost function is written as

$$g(b_1,\ldots,b_C,\mathbf{w}_1,\ldots,\mathbf{w}_C) = \sum_{c=1}^{C}\sum_{p\in\Omega_c} \log\left(1 + \sum_{\substack{j=1\\ j\neq c}}^{C} e^{\left(b_j - b_c\right) + \mathbf{f}_p^T\left(\mathbf{w}_j - \mathbf{w}_c\right)}\right). \tag{7.47}$$

Rewriting each $\mathbf{w}_j$ as $\mathbf{w}_j = \mathbf{F}\mathbf{z}_j + \mathbf{r}_j$, where $\mathbf{F}^T\mathbf{r}_j = \mathbf{0}_{P\times 1}$ for all j, we can rewrite each $\mathbf{f}_p^T\left(\mathbf{w}_j - \mathbf{w}_c\right)$ term as $\mathbf{f}_p^T\left(\mathbf{w}_j - \mathbf{w}_c\right) = \mathbf{f}_p^T\left(\mathbf{F}\left(\mathbf{z}_j - \mathbf{z}_c\right) + \left(\mathbf{r}_j - \mathbf{r}_c\right)\right) = \mathbf{f}_p^T\mathbf{F}\left(\mathbf{z}_j - \mathbf{z}_c\right)$. And denoting $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ the kernel matrix we have that $\mathbf{f}_p^T\left(\mathbf{w}_j - \mathbf{w}_c\right) = \mathbf{h}_p^T\left(\mathbf{z}_j - \mathbf{z}_c\right)$ and so the cost may be written equivalently (kernelized) as

$$g(b_1,\ldots,b_C,\mathbf{z}_1,\ldots,\mathbf{z}_C) = \sum_{c=1}^{C}\sum_{p\in\Omega_c} \log\left(1 + \sum_{\substack{j=1\\ j\neq c}}^{C} e^{\left(b_j - b_c\right) + \mathbf{h}_p^T\left(\mathbf{z}_j - \mathbf{z}_c\right)}\right), \tag{7.48}$$

as shown in Table 7.1.

7.5.2 Fourier kernel calculations – scalar input

From Example 7.2 the (i,j)th element of the kernel matrix $\mathbf{H}$ is given as

$$\mathbf{H}_{ij} = 2\sum_{m=1}^{D} \cos\left(2\pi m x_i\right)\cos\left(2\pi m x_j\right) + \sin\left(2\pi m x_i\right)\sin\left(2\pi m x_j\right). \tag{7.49}$$

Writing this using the complex exponential notation (see Exercise 5.5), we have equivalently

$$\mathbf{H}_{ij} = \sum_{m=-D}^{D} e^{2\pi i m\left(x_i - x_j\right)} - 1. \tag{7.50}$$

If $x_i - x_j$ is an integer then $e^{2\pi i m\left(x_i - x_j\right)} = 1$ and so clearly the above sums to $2D$. Supposing this is not the case, examining the summation alone we may write

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

$$\sum_{m=-D}^{D} e^{2\pi i m(x_i - x_j)} = e^{-2\pi i D(x_i - x_j)} \sum_{m=0}^{2D} e^{2\pi i m(x_i - x_j)}. \tag{7.51}$$

Now the sum on the right hand side above is a geometric series, thus we have the above is equal to

$$e^{-2\pi i D(x_i - x_j)} \frac{1 - e^{2\pi i (x_i - x_j)(2D+1)}}{1 - e^{2\pi i (x_i - x_j)}} = \frac{\sin\left((2D+1)\,\pi\,\left(x_i - x_j\right)\right)}{\sin\left(\pi\,\left(x_i - x_j\right)\right)}, \tag{7.52}$$

where final equality follows from the definition of the complex exponential. Because in the limit as t approaches any integer value $\frac{\sin((2D+1)\pi t)}{\sin(\pi t)} = 2D + 1$, which one can show using L'Hospital's rule from basic calculus, we may therefore generally write in conclusion that

$$\mathbf{H}_{ij} = \frac{\sin\left((2D+1)\,\pi\,\left(x_i - x_j\right)\right)}{\sin\left(\pi\,\left(x_i - x_j\right)\right)} - 1, \tag{7.53}$$

where at integer values of the input it is defined by the associated limit.

7.5.3 Fourier kernel calculations – vector input

Like the multidimensional polynomial basis element (see footnote 5 in the previous chapter) with the complex exponential notation for a general N-dimensional input, each Fourier basis element takes the form $f_{\mathbf{m}}(\mathbf{x}) = e^{2\pi i m_1 x_1} e^{2\pi i m_2 x_2} \cdots e^{2\pi i m_N x_N} = e^{2\pi i \mathbf{m}^T \mathbf{x}}$ where $\mathbf{m} = \begin{bmatrix} m_1 & m_2 & \cdots & m_N \end{bmatrix}^T$, a product of 1-dimensional basis elements. Further a "degree D" sum contains all such basis elements where $-D \leq m_1, m_2, \cdots, m_N \leq D$, and one may deduce that there are $M = (2D+1)^N - 1$ non-constant basis elements in this sum.

The corresponding (i,j)th entry of the kernel matrix in this instance takes the form

$$\mathbf{H}_{ij} = \mathbf{f}_i^T \overline{\mathbf{f}_j} = \left(\sum_{-D \leq m_1, m_2, \cdots, m_N \leq D} e^{2\pi i \mathbf{m}^T (\mathbf{x}_i - \mathbf{x}_j)} \right) - 1. \tag{7.54}$$

Since $e^{a+b} = e^a e^b$ we may write each summand above as $e^{2\pi i \mathbf{m}^T (\mathbf{x}_i - \mathbf{x}_j)} = \prod_{n=1}^{N} e^{2\pi i m_n (x_{in} - x_{jn})}$ and the entire summation as

$$\sum_{-D \leq m_1, m_2, \cdots, m_N \leq D} \prod_{n=1}^{N} e^{2\pi i m_n (x_{in} - x_{jn})}. \tag{7.55}$$

Finally, one can show that the above can be written simply as

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010

$$\sum_{-D\leq m_1,m_2,\cdots,m_N\leq D}\prod_{n=1}^{N}e^{2\pi i m_n(x_{in}-x_{jn})}=\prod_{n=1}^{N}\left(\sum_{m=-D}^{D}e^{2\pi i m(x_{in}-x_{jn})}\right). \tag{7.56}$$

Since we already have that $\sum_{m=-D}^{D}e^{2\pi i m(x_{in}-x_{jn})}=\frac{\sin((2D+1)\pi(x_{in}-x_{jn}))}{\sin(\pi(x_{in}-x_{jn}))}$, the (i,j)th entry of the kernel matrix can easily be calculated as

$$\mathbf{H}_{ij}=\prod_{n=1}^{N}\frac{\sin\left((2D+1)\,\pi\,\left(x_{in}-x_{jn}\right)\right)}{\sin\left(\pi\,\left(x_{in}-x_{jn}\right)\right)}-1. \tag{7.57}$$

Downloaded from https://www.cambridge.org/core. Universitaetsbibliothek Duisburg-Essen, on 13 Jul 2018 at 16:44:30, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9781316402276.010