9 Regularization

Section 1.2

We introduced the concept of regularization when discussing polynomial curve fitting as a way to reduce over-fitting by discouraging the parameters of the model from taking values with a large magnitude. This involved adding a simple penalty term to the error function to give a regularized error function in the form

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{9.1}$$

Section 4.3

where $\mathbf{w}$ is the vector of model parameters, $E(\mathbf{w})$ is the unregularized error function, and the regularization hyperparameter λ controls the strength of the regularization effect. An improvement in predictive accuracy with such a regularizer can be understood in terms of the bias–variance trade-off through the reduction in the variance of the solution at the expense of some increase in bias. In this chapter we will explore regularization in depth and will discuss several different approaches to regularization. We will also look more broadly at the important role of bias in achieving good generalization from finite training data sets.

In a practical application, it is very unlikely that the process that generates the data will correspond precisely to a particular neural network architecture, and so any given neural network will only ever represent an approximation to the true data generator. Larger networks can provide closer approximations, but this comes at the risk of over-fitting. In practice, we find that the best generalization results are almost always obtained by using a larger network combined with some form of regularization. In this chapter we explore several alternative regularization techniques including early stopping, model averaging, dropout, data augmentation, and parameter sharing. Multiple forms of regularization can be used together if desired. For example, error function regularization of the form (9.1) is often used alongside dropout.

9.1. Inductive Bias

Section 1.2.4

Section 1.2.5

Section 4.3

When we compared the predictive error of polynomials of various orders for the sinusoidal synthetic data problem, we saw that the smallest generalization error was achieved using a polynomial of intermediate complexity, being neither too simple nor too flexible. A similar result was found when we used a regularization term of the form (9.1) to control model complexity, as an intermediate value of the regularization coefficient λ gave the best predictions for new input values. Insight into this result came from the bias–variance decomposition, where we saw that an appropriate level of bias in the model was important to allow generalization from finite data sets. Simple models with high bias are unable to capture the variation in the underlying data generation process, whereas highly flexible models with low bias are prone to over-fitting leading to poor generalization. As the size of the data set grows, we can afford to use more flexible models having less bias without incurring excessive variance, thereby leading to improved generalization. Note that in a practical setting, our choice of model might also be influenced by factors such as memory usage or speed of execution. Here we ignore such ancillary considerations and focus on the core goal of achieving good predictive performance, in other words good generalization.

9.1.1 Inverse problems

This issue of model choice lies at the heart of machine learning and can be traced to the fact that most machine learning tasks are examples of *inverse problems*. Given a conditional distribution $p(t|\mathbf{x})$ along with a finite set of input points $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, it is straightforward, at least in principle, to sample corresponding values $\{t_1, \ldots, t_N\}$ from that distribution. In machine learning, however, we have to solve the inverse of this problem, namely to infer an entire distribution given only a finite number of samples. This is intrinsically *ill-posed*, as there are infinitely many distributions which could potentially have been responsible for generating the observed training data. In fact any distribution that has a non-zero probability density at the observed target values is a candidate.

For machine learning to be useful, however, we need to make predictions for new values of $\mathbf{x}$, and therefore we need a way to choose a specific distribution from amongst the infinitely many possibilities. The preference for one choice over others is called *inductive bias*, or *prior knowledge*, and plays a central role in machine learning. Prior knowledge may come from background information that helps constrain the space of solutions. For many applications, small changes in the input values should lead to small changes in the output values, and so we should bias our solutions towards those with smoothly varying functions. Regularization terms of the form (9.1) encourage the model weights to have a smaller magnitude and hence introduce a bias towards functions that vary more slowly with changes in the inputs. Likewise, when detecting objects in images, we can introduce prior knowledge that the identity of an object is generally independent of its location within the image. *Chapter 10* This is known as translation invariance, and incorporating this into our solution can greatly simplify the task of building a system with good generalization. However, care must be taken not to incorporate biases or constraints that are inconsistent with the underlying process that generates the data. For example, assuming that the relationship between outputs and inputs is linear when in fact there are significant nonlinearities can lead to a system that produces inaccurate answers.

Section 6.3.4 Techniques such as transfer learning and multi-task learning can also be viewed from the perspective of regularization. When training data for a particular task is limited, additional data from a different, but related, task can be used to help determine the learnable parameters in a neural network. The assumption of similarity between the tasks represents a more sophisticated form of inductive bias compared to simple regularization, and this explains the improved performance resulting from the use of the additional data.

9.1.2 No free lunch theorem

The core focus of this book is on the important class of machine learning models called deep neural networks. These are highly flexible models and have revolutionized many fields including computer vision, speech recognition, and natural language processing. In fact, they have become the framework of choice for the great majority of machine learning applications. It might appear, therefore, that they represent a ‘universal’ learning algorithm able to solve all tasks. However, even very flexible neural networks contain important inductive biases. For example, convolutional neural networks encode specific forms of inductive bias, including translation equivariance, that are especially useful in applications involving images. *Chapter 10*

The *no free lunch theorem* (Wolpert, 1996), named from the expression ‘There’s no such thing as a free lunch,’ states that every learning algorithm is as good as any other when averaged over all possible problems. If a particular model or algorithm is better than average on some problems, it must be worse than average on others. However, this is a rather theoretical notion as the space of possible problems here includes relationships between input and output that would be very uncharacteristic of any plausible practical application. For example, we have already noted that most examples of practical interest exhibit some degree of smoothness, in which small changes in the input values are associated, for the most part, with small changes in the target values. Models such as neural networks, and indeed most widely used machine learning techniques, exhibit this form of inductive bias, and therefore to some degree, they have broad applicability.

Although the no free lunch theorem is somewhat theoretical, it does highlight the central importance of bias in determining the performance of a machine learning algorithm. It is not possible to learn 'purely from data' in the absence of any bias. In practice, bias may be implicit. For example, every neural network has a finite number of parameters, which therefore limits the functions that it can represent. However, bias may also be encoded explicitly as a reflection of prior knowledge relating to the specific type of problem being solved.

In trying to find general-purpose learning algorithms, we are really seeking inductive biases that are appropriate to the broad classes of applications that will be encountered in practice. For any given application, however, better results can be obtained if it is possible to incorporate stronger inductive biases that are specific to that application. The perspective of *model-based machine learning* (Winn *et al.*, 2023) advocates making all the assumptions in machine learning models explicit so that the appropriate choices can be made for inductive biases.

We have seen that inductive bias can be incorporated through the form of distribution, for example by specifying that the output is a linear function of a fixed set of specific basis functions. It can also be incorporated through the addition of a regularization term to the error function used during training. Yet another way to control the complexity of a neural network is through the training process itself. We will see that deep neural networks can give good generalization even when the number of adjustable parameters exceeds the number of training data points, provided the training process is set up correctly. Part of the skill in applying deep learning to real-world problems is in the careful design of inductive bias and the incorporation of prior knowledge.

Chapter 7

9.1.3 Symmetry and invariance

In many applications of machine learning, the predictions should be unchanged, or *invariant*, under one or more transformations of the input variables. For example, when classifying an object in two-dimensional images, such as a cat or dog, a particular object should be assigned the same classification irrespective of its position within the image. This is known as *translation invariance*. Likewise changes to the size of the object within the image should also leave its classification unchanged. This is called *scale invariance*. Exploiting such symmetries to create inductive biases can greatly improve the performance of machine learning models and forms the subject of *geometric deep learning* (Bronstein *et al.*, 2021).

Transformations, such as a translation or scaling, that leave particular properties unchanged, represent *symmetries*. The set of all transformations corresponding to a particular symmetry form a mathematical structure called a *group*. A group consists of a set of elements $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ together with a binary operation for composing pairs of elements together, which we denote using the notation $\mathcal{A} \circ \mathcal{B}$. The following four axioms hold for a group:

1. **Closed**: For any two elements $\mathcal{A}, \mathcal{B}$ in the set, $\mathcal{A} \circ \mathcal{B}$ must also be in the set.
2. **Associative**: For any three elements $\mathcal{A}, \mathcal{B}, \mathcal{C}$ in the set, $(\mathcal{A} \circ \mathcal{B}) \circ \mathcal{C} = \mathcal{A} \circ (\mathcal{B} \circ \mathcal{C})$.
3. **Identity**: There exists an element $\mathcal{I}$ of the set, called the identity, with the property: $\mathcal{A} \circ \mathcal{I} = \mathcal{I} \circ \mathcal{A} = \mathcal{A}$ for every element $\mathcal{A}$ in the set.
4. **Inverse**: For each element $\mathcal{A}$ in the set, there exists another element in the set, which we denote by $\mathcal{A}^{-1}$, called the inverse, which has the property: $\mathcal{A} \circ \mathcal{A}^{-1} = \mathcal{A}^{-1} \circ \mathcal{A} = \mathcal{I}$.

Exercise 9.1

Simple examples of groups include the set of rotations of a square through multiples of 90° or the set of continuous translations of an object in a two-dimensional plane.

In principle, invariance of the predictions made by a neural network to transformations of the input space could be learned from data, without any special modifications to the network or the training procedure. In practice, however, this can prove to be extremely challenging because such transformations can produce substantial changes in the raw data. For example, relatively small translations of an object within an image, even by a few pixels, can cause pixel values to change significantly. Furthermore, multiple invariances must often hold at the same time, for example invariance to translations in two dimensions as well as scaling, rotation, changes of intensity, changes of colour balance, and many others. There are exponentially many possible *combinations* of such transformations, making the size of the required training set needed to learn all of these invariances prohibitive.

We therefore seek more efficient approaches for encouraging an adaptive model to exhibit the required invariances. These can broadly be divided into four categories:

1. **Pre-processing.** Invariances are built into a pre-processing stage by computing features of the data that are invariant under the required transformations. Any subsequent regression or classification system that uses such features as inputs will necessarily also respect these invariances.
2. **Regularized error function.** A regularization term is added to the error function to penalize changes in the model output when the input is subject to one of the invariant transformations.
3. **Data augmentation.** The training set is expanded using replicas of the training data points, transformed according to the desired invariances and carrying the same output target values as the untransformed examples.
4. **Network architecture.** The invariance properties are built into the structure of a neural network through an appropriate choice of network architecture.

Chapter 6

One challenge with approach 1 is to design features that exhibit the required invariances without also discarding information that can be useful for determining the network outputs. We have already seen that fixed, hand-crafted features have limited capabilities and have been superseded by learned representations obtained using deep neural networks.

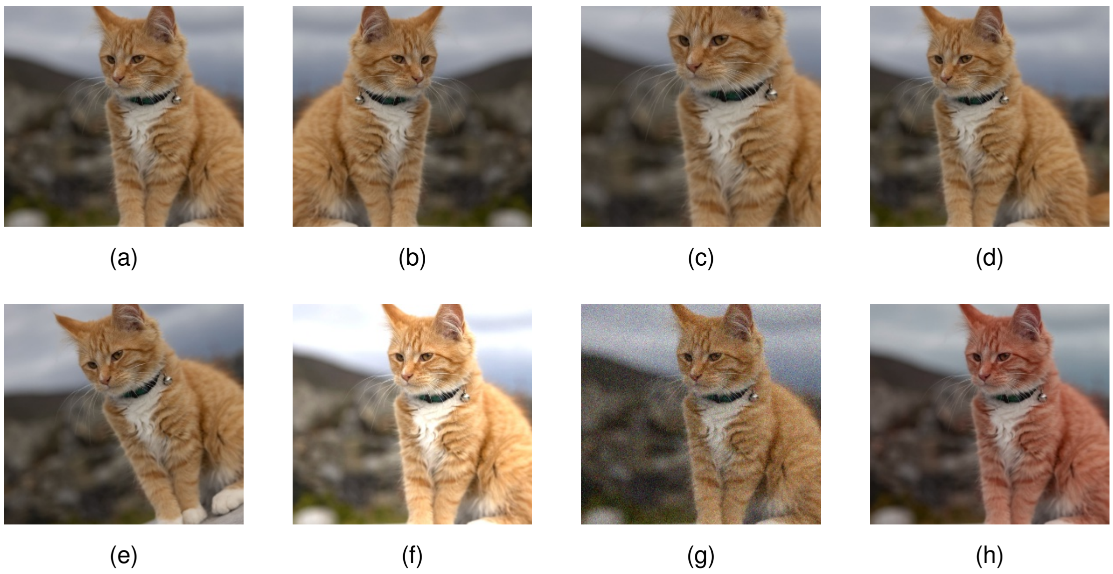

Figure 9.1 Illustration of data set augmentation, showing (a) the original image, (b) horizontal inversion, (c) scaling, (d) translation, (e) rotation, (f) brightness and contrast change, (g) additive noise, and (h) colour shift.

An example of approach 2 is the technique of *tangent propagation* (Simard *et al.*, 1992) in which a regularisation term is added to the error function during training. This term directly penalizes changes in the output resulting from changes in the input variables that correspond to one of the invariant transformations. A limitation of this technique, in addition to the extra complexity of training, is can only cope with small transformations (e.g., translations by less than a pixel).

Approach 3 is known as *data set augmentation*. It is often relatively easy to implement and can prove to be very effective in practice. It is often applied in the context of image analysis as it straightforward to create the transformed training data. Figure 9.1 shows examples of such transformations applied to an image of a cat. For medical images of soft tissue, data augmentation could also include continuous 'rubber sheet' deformations (Ronneberger, Fischer, and Brox, 2015).

For sequential training algorithms, such as stochastic gradient descent, the data set can be augmented by transforming each input data point before it is presented to the model so that, if the data points are being recycled, a different transformation (drawn from an appropriate distribution) is applied each time. For batch methods, a similar effect can be achieved by replicating each data point a number of times and transforming each copy independently.

We can analyse the effect of using augmented data by considering transformations that represent small changes to the original examples and then making a Taylor expansion of the error function in powers of the magnitude of the transformation (Bishop, 1995c; Leen, 1995; Bishop, 2006). This leads to a regularized error function in which the regularizer penalizes the gradient of the network output with respect to the input variables projected onto the direction of transformation. This is related to the technique of tangent propagation discussed above. A special case arises when the transformation of the input variables consists simply of the addition of random noise, in which case the regularizer penalizes the derivatives of the network outputs with respect to the inputs. Again, this is intuitively reasonable, since we are encouraging the network outputs to remain unchanged despite the addition of noise to the input variables.

Exercise 9.2

Finally, approach 4, in which we build invariances into the structure of the network, has proven to be very powerful and effective and leads to other key benefits. We will discuss this approach at length in the context of convolutional neural networks for computer vision.

Chapter 10

9.1.4 Equivariance

An important generalization of invariance is called *equivariance* in which the output of the network, instead of remaining constant when the input is transformed, is itself transformed in a specific way. For example, consider a network that takes an image as input and returns a segmentation of that image in which each pixel is classified as belonging either to a foreground object or to the background. In this case, if the location of the object within the image is translated, we want the corresponding segmentation of the object to be similarly translated. Suppose we denote the image by $\mathbf{I}$, and the operation of the segmentation network by $\mathcal{S}$, then for a translation operation $\mathcal{T}$ we have

$$\mathcal{S}(\mathcal{T}(\mathbf{I})) = \mathcal{T}(\mathcal{S}(\mathbf{I})), \tag{9.2}$$

which says that the segmentation of the translated image is given by the translation of the segmentation of the original image. This is illustrated in Figure 9.2

More generally, equivariance can hold if the transformation applied to the output is different to that applied to the input:

$$\mathcal{S}(\mathcal{T}(\mathbf{I})) = \widetilde{\mathcal{T}}(\mathcal{S}(\mathbf{I})). \tag{9.3}$$

For example, if the segmented image has a lower resolution than the original image, then if $\mathcal{T}$ is a translation in the original image space, $\widetilde{\mathcal{T}}$ represents the corresponding translation in the lower-dimensional segmentation space. Similarly, if $\mathcal{S}$ is an operator that measures the orientation of an object within an image, and $\mathcal{T}$ represents a rotation (which is a complex nonlinear transformation of all of the pixel values in the image) then $\widetilde{\mathcal{T}}$ will increment or decrement the scalar orientation value generated by $\mathcal{S}$.

We also see that invariance is a special case of equivariance in which the output transformation is simply the identity. For example, if $\mathcal{C}$ is a neural network that classifies an object within an image and $\mathcal{T}$ is a translation operator then

$$\mathcal{C}(\mathcal{T}(\mathbf{I})) = \mathcal{C}(\mathbf{I}), \tag{9.4}$$

which says that the class of the object does not depend on its position within the image.

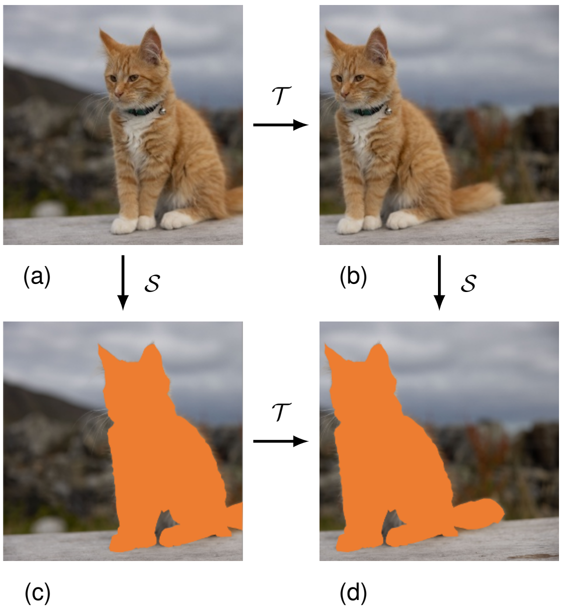

Figure 9.2 Illustration of equivariance, corresponding to (9.2). If an image (a) is first translated to give (b) and then segmented to give (d), the result is the same as if the image is first segmented to give (c) and then translated to give (d).

9.2. Weight Decay

Section 1.2.5

We introduced regularization in the context of linear regression to control model complexity, as an alternative to limiting the number of parameters in the model. The simplest regularizer comprises the sum of the squares of the model parameters to give a regularized error function of the form (9.1), which penalizes parameter values with large magnitude. The effective model complexity is then determined by the choice of the regularization coefficient λ.

Section 2.6.2

We have also seen that this additive regularization term can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector $\mathbf{w}$. This provides a probabilistic perspective on the inclusion of prior knowledge into the model training process. Unfortunately, this prior is expressed over the model parameters, whereas any domain knowledge we might possess regarding the problem to be solved is more likely to be expressed in terms of the network function from inputs to outputs. The relationship between the parameters and the network function is, however, extremely complex, and therefore only very limited kinds of prior knowledge can easily be expressed directly as priors over model parameters.

Exercise 9.3

The sum-of-squares regularizer in (9.1) is known in the machine learning literature as *weight decay* because in sequential learning algorithms, it encourages weight values to decay towards zero, unless supported by the data. One advantage of this kind of regularizer is that it is trivial to evaluate its derivatives for use in gradient descent training. Specifically for (9.1) the gradient is given by

$$\nabla \widetilde{E}(\mathbf{w}) = \nabla E(\mathbf{w}) + \lambda \mathbf{w}. \tag{9.5}$$

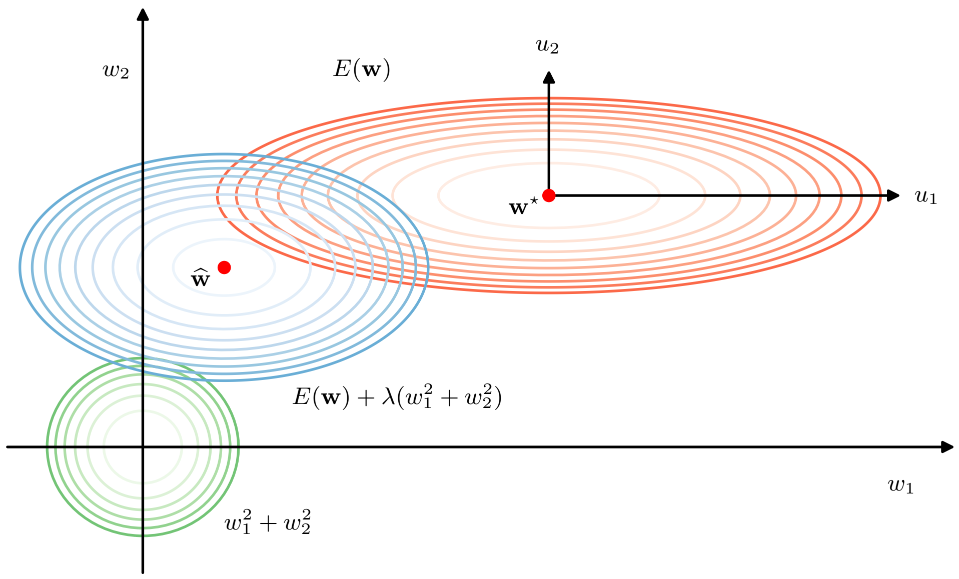

Figure 9.3 Contours of the error function (red), the regularization term (green), and a linear combination of the two (blue) for a quadratic error function and a sum-of-squares regularizer $\lambda(w_1^2 + w_2^2)$. Here the axes in parameter space have been rotated to align with the axes of the elliptical contour of the unregularized error function. For $\lambda = 0$, the minimum error is indicated by $\mathbf{w}^\star$. When $\lambda > 0$, the minimum of the regularized error function $E(\mathbf{w}) + \lambda(w_1^2 + w_2^2)$ is shifted towards the origin. This shift is greater in the direction of w_1 because the unregularized error is relatively insensitive to the parameter value, and less in direction w_2 where the error is more strongly dependent on the parameter value. The regularization term is effectively suppressing parameters that have only a small effect on the accuracy of the network predictions.

We see that the factor of $1/2$ in (9.1), which is often included by convention, disappears when we take the derivative.

The general effect of a quadratic regularizer can be seen by considering a two-dimensional parameter space along with an unregularized error function $E(\mathbf{w})$ that is a quadratic function of $\mathbf{w}$ (corresponding to a simple linear regression model with a sum-of-squares error function), as illustrated in Figure 9.3. The axes in parameter space have been rotated to align with the eigenvectors of the Hessian matrix, corresponding to the axes of the elliptical error function contours. We see that the effect of the regularization term is to shrink the magnitudes of the weight parameters. However, the effect is much larger for parameter w_1 because the unregularized error is much less sensitive to the value of w_1 compared to that of w_2. Intuitively only the parameter w_2 is 'active' because the output is relatively insensitive to w_1, and hence the regularizer pushes w_1 close to zero. The *effective number of parameters* is the number that remain active after regularization, and this concept can be formalized either from a Bayesian or from a frequentist perspective (Bishop, 2006; Hastie, Tibshirani, and Friedman, 2009). For $\lambda \to \infty$, all the parameters are driven to zero and the effective number of parameters is then zero. As λ is reduced, the number of parameters increases until for $\lambda = 0$ it equals the total number of actual param-

Section 4.1.2
Section 8.1.6

eters in the model. We see that controlling model complexity by regularization has similarities to controlling model complexity by limiting the number of parameters.

9.2.1 Consistent regularizers

One of the limitations of simple weight decay in the form (9.1) is that it breaks certain desirable transformation properties of network mappings. To illustrate this, consider a multilayer perceptron network having two layers of weights and linear output units that performs a mapping from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$. The activations of the hidden units in the first hidden layer take the form

$$z_j = h\left(\sum_i w_{ji}x_i + w_{j0}\right) \tag{9.6}$$

whereas the activations of the output units are given by

$$y_k = \sum_j w_{kj}z_j + w_{k0}. \tag{9.7}$$

Suppose we perform a linear transformation of the input data:

$$x_i \rightarrow \widetilde{x}_i = ax_i + b. \tag{9.8}$$

Then we can arrange for the mapping performed by the network to be unchanged by making a corresponding linear transformation of the weights and biases from the inputs to the units in the hidden layer:

Exercise 9.4

$$w_{ji} \rightarrow \widetilde{w}_{ji} = \frac{1}{a}w_{ji} \tag{9.9}$$

$$w_{j0} \rightarrow \widetilde{w}_{j0} = w_{j0} - \frac{b}{a}\sum_i w_{ji}. \tag{9.10}$$

Similarly, a linear transformation of the output variables of the network of the form

$$y_k \rightarrow \widetilde{y}_k = cy_k + d \tag{9.11}$$

can be achieved transforming the second-layer weights and biases using

$$w_{kj} \rightarrow \widetilde{w}_{kj} = cw_{kj} \tag{9.12}$$

$$w_{k0} \rightarrow \widetilde{w}_{k0} = cw_{k0} + d. \tag{9.13}$$

If we train one network using the original data and one network using data for which the input and/or target variables have been transformed by one of the above linear transformations, then consistency requires that we should obtain equivalent networks that differ only by the linear transformation of the weights as given. Any regularizer should be consistent with this property, otherwise it would arbitrarily favour one solution over another, equivalent one. Clearly, simple weight decay (9.1), which treats all weights and biases on an equal footing, does not satisfy this property.

We therefore look for a regularizer that is invariant under the linear transformations (9.9), (9.10), (9.12), and (9.13). These require that the regularizer should be invariant to re-scaling of the weights and to shifts of the biases. Such a regularizer is given by

$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2 \tag{9.14}$$

where $\mathcal{W}_1$ denotes the set of weights in the first layer, $\mathcal{W}_2$ denotes the set of weights in the second layer, and biases are excluded from the summations. This regularizer will remain unchanged under the weight transformations provided the regularization parameters are re-scaled using $\lambda_1 \to a^{1/2}\lambda_1$ and $\lambda_2 \to c^{-1/2}\lambda_2$.

The regularizer (9.14) corresponds to a prior distribution over the parameters of the form:

$$p(\mathbf{w}|\alpha_1,\alpha_2) \propto \exp\left(-\frac{\alpha_1}{2}\sum_{w\in\mathcal{W}_1} w^2 - \frac{\alpha_2}{2}\sum_{w\in\mathcal{W}_2} w^2\right). \tag{9.15}$$

Note that priors of this form are *improper* (they cannot be normalized) because the bias parameters are unconstrained. Using improper priors can lead to difficulties in selecting regularization coefficients and in model comparison within the Bayesian framework. It is therefore common to include separate priors for the biases (which then break the shift invariance) that have their own hyperparameters.

We can illustrate the effect of the resulting four hyperparameters by drawing samples from the prior and plotting the corresponding network functions, as shown in Figure 9.4. The priors are governed by four hyperparameters, α_1^{b}, α_1^{w}, α_2^{b}, and α_2^{w}, which represent the precisions of the Gaussian distributions of the first-layer biases, first-layer weights, second-layer biases, and second-layer weights, respectively. We see that the parameter α_2^{w} governs the vertical scale of the functions (note the different vertical axis ranges on the top two diagrams), α_1^{w} governs the horizontal scale of variations in the function values, and α_1^{b} governs the horizontal range over which variations occur. The parameter α_2^{b}, whose effect is not illustrated here, governs the range of the vertical offsets of the functions

More generally, we can consider regularizers in which the weights are divided into any number of groups $\mathcal{W}_k$ so that

$$\Omega(\mathbf{w}) = \frac{1}{2}\sum_k \alpha_k \|\mathbf{w}\|_k^2 \tag{9.16}$$

where

$$\|\mathbf{w}\|_k^2 = \sum_{j\in\mathcal{W}_k} w_j^2. \tag{9.17}$$

For example, we could use a different regularizer for each layer in a multilayer network.

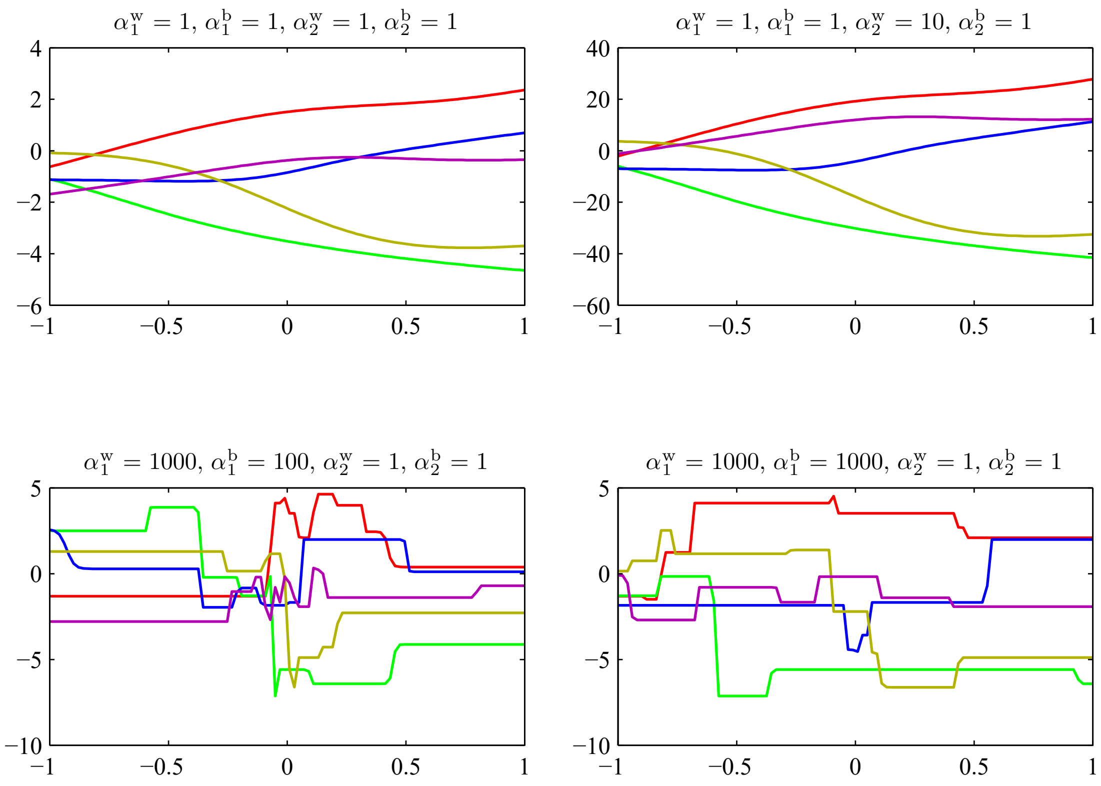

Figure 9.4 Illustration of the effect of the hyperparameters governing the prior distribution over weights and biases in a two-layer network having a single input, a single linear output, and 12 hidden units with $\tanh$ activation functions.

9.2.2 Generalized weight decay

A generalization of the simple quadratic regularizer is sometimes used:

$$\Omega(\mathbf{w}) = \frac{\lambda}{2} \sum_{j=1}^{M} |w_j|^q \tag{9.18}$$

where $q = 2$ corresponds to the quadratic regularizer in (9.1). Figure 9.5 shows contours of the regularization function for different values of q.

A regularizer of the form (9.18) with $q = 1$ is known as a *lasso* in the statistics literature (Tibshirani, 1996). For quadratic error functions, it has the property that if λ is sufficiently large, some of the coefficients w_j are driven to zero, leading to a *sparse* model in which the corresponding basis functions play no role. To see this,

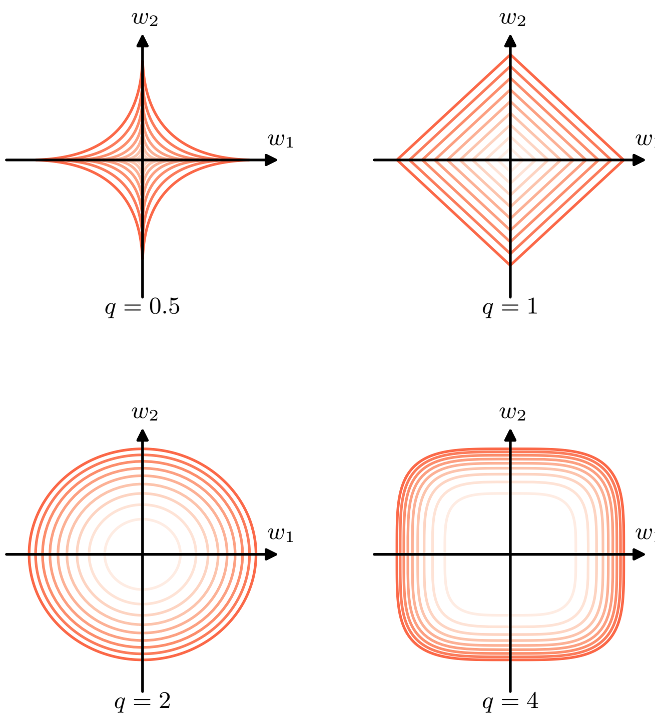

Figure 9.5 Contours of the regularization term in (9.18) for various values of the parameter q.

we first note that minimizing the regularized error function given by

$$E(\mathbf{w}) + \frac{\lambda}{2} \sum_{j=1}^{M} |w_j|^q \tag{9.19}$$

Exercise 9.5

is equivalent to minimizing the unregularized error function $E(\mathbf{w})$ subject to the constraint

$$\sum_{j=1}^{M} |w_j|^q \leqslant \eta \tag{9.20}$$

Appendix C

for an appropriate value of the parameter η, where the two approaches can be related using Lagrange multipliers. The origin of the sparsity can be seen in Figure 9.6, which shows the minimum of the error function, subject to the constraint (9.20). As λ is increased, more parameters will be driven to zero. By comparison, a quadratic regularizer leaves both weight parameters with non-zero values.

Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity. However, the problem of determining the optimal model complexity is then shifted from one of finding the appropriate number of learnable parameters to one of determining a suitable value of the regularization coefficient λ. We will discuss the issue of model complexity in the next section.

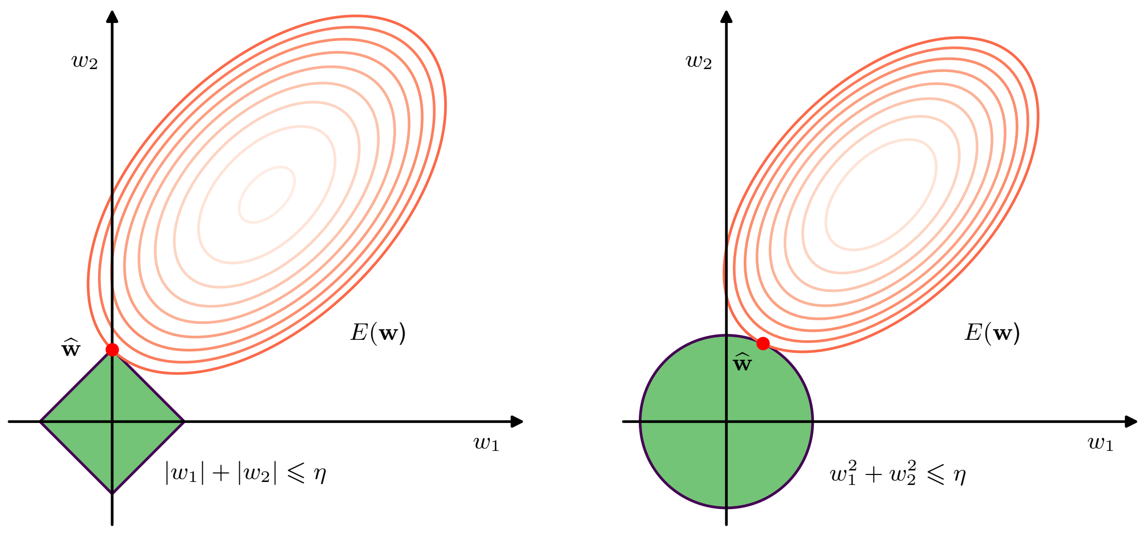

Figure 9.6 Plot of the contours of the unregularized error function (red) along with the constraint region (9.20) for the lasso regularizer $q = 1$ on the left, and the quadratic regularizer $q = 2$ on the right, in which the optimum value for the parameter vector $\mathbf{w}$ is denoted by $\widehat{\mathbf{w}}$. The lasso gives a sparse solution in which $\widehat{w}_1 = 0$, whereas the quadratic regularizer simply reduces w_1 to a smaller value.

9.3. Learning Curves

We have already explored how the generalization performance of a model varies as we change the number of parameters in the model, the size of the data set, and the coefficient of a weight-decay regularization term. Each of these allows for a trade-off between bias and variance to minimize the generalization error. Another factor that influences this trade-off is the learning process itself. During optimization of the error function through gradient descent, the training error typically decreases as the model parameters are updated, whereas the error for hold-out data may be non-monotonic. This behaviour can be visualized using *learning curves*, which plot performance measures such as training set and validation set error as a function of iteration number during an iterative learning process such as stochastic gradient descent. These curves provide insight into the progress of training and also offer a practical methodology for controlling the effective model complexity.

9.3.1 Early stopping

An alternative to regularization as a way of controlling the effective complexity of a network is *early stopping*. The training of deep learning models involves an iterative reduction of the error function defined with respect to a set of training data. Although the error function evaluated using the training set often shows a broadly monotonic decrease as a function of the iteration number, the error measured with respect to held-out data, generally called a validation set, often shows a decrease at

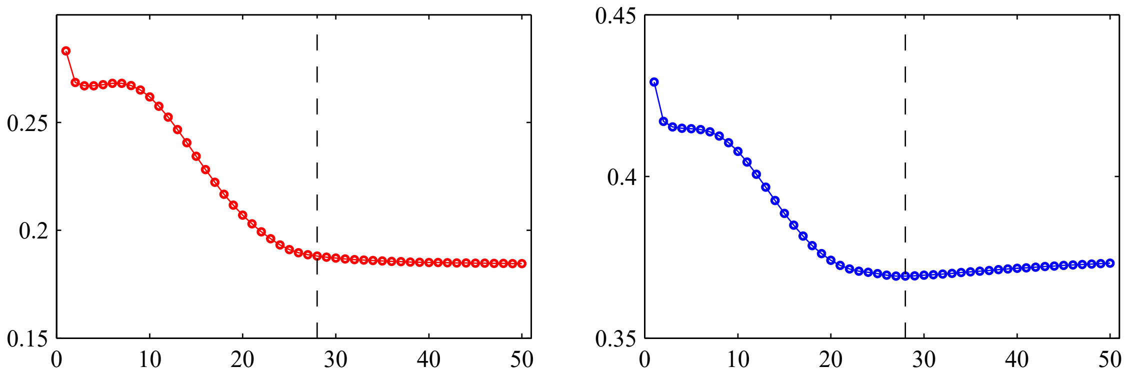

Figure 9.7 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. To achieve the best generalization performance , the training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

first, followed by an increase as the network starts to over-fit. Therefore, to obtain a network with good generalization performance, training should be stopped at the point of smallest error with respect to the validation data set, as indicated in Figure 9.7.

This behaviour of the learning curves is sometimes explained qualitatively in terms of the effective number of parameters in the network. This number starts out small and then grows during training, corresponding to a steady increase in the effective complexity of the model. Stopping training before a minimum of the training error has been reached is a way to limit the effective network complexity.

We can verify this insight for a quadratic error function and show that early stopping should exhibit similar behaviour to regularization using a simple weight-decay term (Bishop, 1995a). This can be understood from Figure 9.8, in which the axes in weight space have been rotated to be parallel to the eigenvectors of the Hessian matrix. If, in the absence of weight decay, the weight vector starts at the origin and proceeds during training along a path that follows the local negative gradient vector, then the weight vector will move initially parallel to the w_2 axis through a point corresponding roughly to $\widehat{\mathbf{w}}$ and then move towards the minimum of the error function $\mathbf{w}_{\mathrm{ML}}$. This follows from the shape of the error surface and the widely differing eigenvalues of the Hessian. Stopping at a point near $\widehat{\mathbf{w}}$ is therefore similar to weight decay. The relationship between early stopping and weight decay can be made quantitative, thereby showing that the quantity $\tau\eta$ (where τ is the iteration index and η is the learning rate parameter) acts like the reciprocal of the regularization parameter λ. The effective number of parameters in the network therefore grows during training.

Section 7.1.1

Exercise 9.6

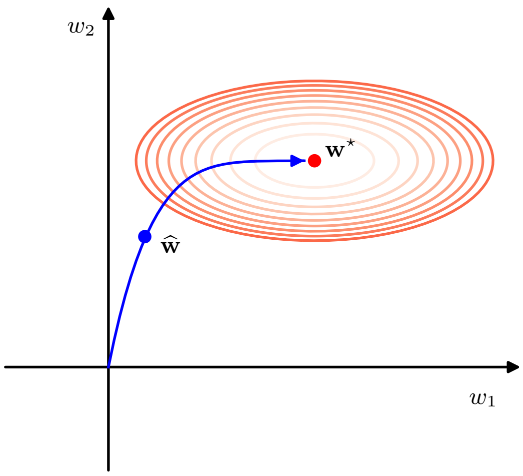

Figure 9.8 A schematic illustration of why early stopping can give similar results to weight decay for a quadratic error function. The ellipses show contours of constant error, and $\mathbf{w}^\star$ denotes the maximum likelihood solution corresponding to the minimum of the unregularized error function. If the weight vector starts at the origin and moves according to the local negative gradient direction, then it will follow the path shown by the curve. By stopping training early, a weight vector $\widehat{\mathbf{w}}$ is found that is qualitatively like that obtained with a simple weight-decay regularizer along with training to the minimum of the regularized error, as can be seen by comparing with Figure 9.3.

9.3.2 Double descent

Section 4.3

The bias–variance trade-off provides insight into the generalization performance of a learnable model as the number of parameters in the model is varied. Models with too few parameters will have a high test set error due to the limited representational capacity (high bias), and as the number of parameters increases, the test error is expected to fall. However, as the number of parameters is increased further, the test error increases again due to over-fitting (high variance). This leads to the conventional belief, widespread in classical statistics, that the number of parameters in the model needs to be limited according to the size of the data set and that for a given training data set, very large models are expected to have poor performance.

Contrary to this expectation, however, modern deep neural networks can have excellent performance even when the number of parameters far exceeds that required to achieve a perfect fit to the training data (Zhang *et al.*, 2016), and the general wisdom in the deep learning community is that bigger models are better. Although early stopping is sometimes used, models may also be trained to zero error and yet still have good performance on test data.

These seemingly contradictory perspectives can be reconciled by examining learning curves and other plots of generalization performance versus model complexity, which reveal a more subtle phenomenon called *double descent* (Belkin *et al.*, 2019). This is illustrated in Figure 9.9, which shows training set and test set errors versus model complexity, as determined by the number of learnable parameters, for a large neural network called ResNet18 (He *et al.*, 2015a), which has 18 layers of parameters trained on an image classification task. The number of weights and biases in the network is varied by changing the 'width parameter', which governs the number of hidden units in each layer. We see that the training error decreases monotonically with increasing complexity of the model, as expected. However, the

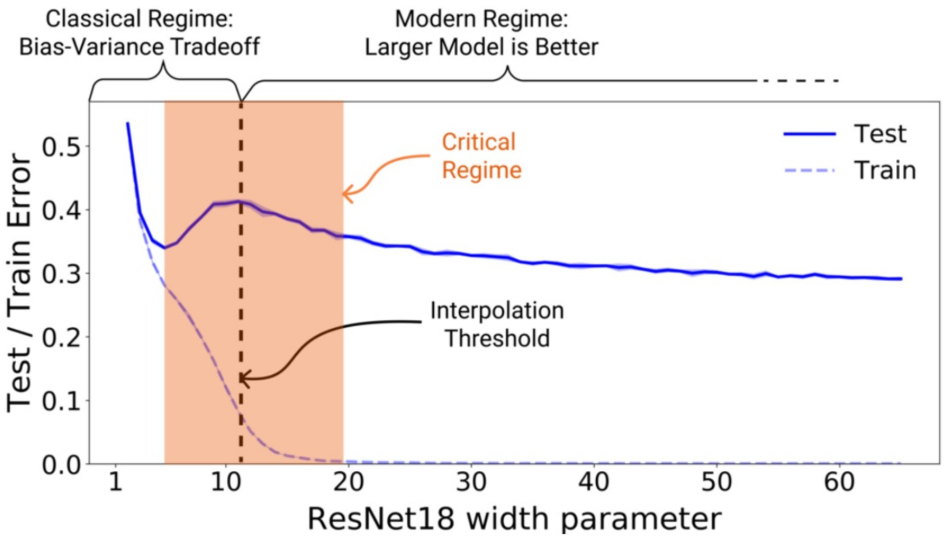

Figure 9.9 Plot of training set and test set errors for a large neural network model called ResNet18 trained on an image classification problem versus the complexity of a model. The horizontal axis represents a hyperparameter governing the number of hidden units and hence the overall number of weights and biases in the network. The vertical dashed line, labelled 'interpolation threshold' indicates the level of model complexity at which the model is capable, in principle, of achieving zero error on the training set. [From Nakkiran *et al.* (2019) with permission.]

test set error decreases at first then increases again and then finally decreases again. This reduction in test set error for very large models continues even after the training set error has reached zero.

This surprising behaviour is more complex than we would expect from the usual bias–variance discussion of classical statistics and exhibits two different regimes of model fitting, as shown schematically in Figure 9.9, corresponding to the classical bias–variance trade-off for small to medium complexity, followed by a further reduction in test set error as we enter a regime of very large models. The transition between the two regimes occurs roughly when the number of parameters in the model is sufficiently large that the model is able to fit the training data exactly (Belkin *et al.*, 2019). Nakkiran *et al.* (2019) define the *effective model complexity* to be the maximum size of training data set on which a model can achieve zero training error, and so double descent arises when the effective model complexity exceeds the number of data points in the training set.

We see similar behaviour if we control model complexity using early stopping, as seen in Figure 9.10. Increasing the number of training epochs increases the effective model complexity, and for a sufficiently large model, double descent is again observed. For such models there are many possible solutions including those that over-fit to the data. It therefore seems to be a property of stochastic gradient descent that the implicit biases that it introduces lead to good generalization performance.

Analogous results are also obtained when a regularization term in the error func-

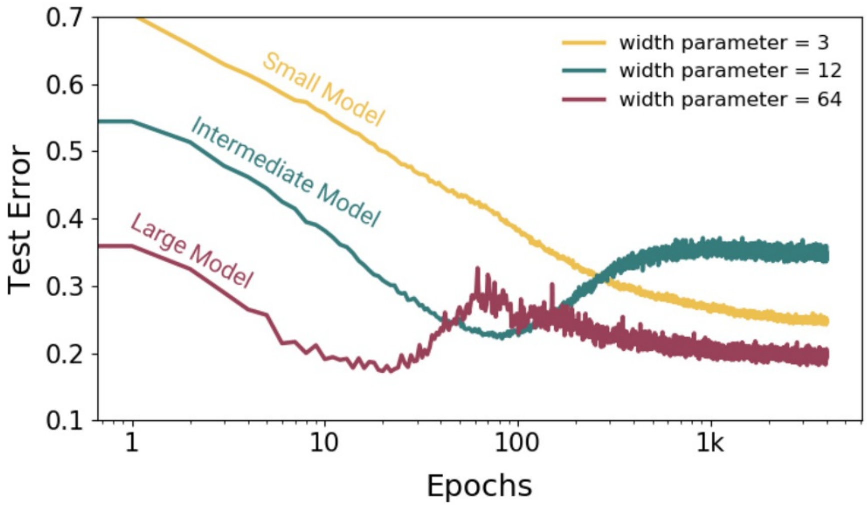

Figure 9.10 Plot of test set error versus number of epochs of gradient descent training for ResNet18 models of various sizes. The effective model complexity increases with the number of training epochs, and the double descent phenomenon is observed for a sufficiently large model. [From Nakkiran *et al.* (2019) with permission.]

tion is used to control complexity. Here the test set error of a large model trained to convergence shows double descent with respect to $1/\lambda$, the inverse regularization parameter, since high λ corresponds to low complexity (Yilmaz and Heckel, 2022).

One ironic consequence of double descent is that it possible to operate in a regime where increasing the size of the training data set could actually reduce performance, contrary to the conventional view that more data is always a good thing. For a model in the critical regime shown in Figure 9.9, an increase in the size of the training set can push the interpolation threshold to the right, leading to a higher test set error. This is confirmed in Figure 9.11, which shows the test set error for a transformer model as a function of the dimensionality of the input space, known as the embedding dimension. Increasing the embedding dimension increases the number of weights and biases in the model and hence increases the model complexity. We see that increasing the training set size from 4,000 to 18,000 data points leads to a curve that is overall much lower. However, for a range of embedding dimensions that correspond to models in the critical complexity regime, increasing the size of the data set can actually reduce generalization performance.

Chapter 12

9.4. Parameter Sharing

Regularization terms, such as the L_2 regularizer $\|\mathbf{w}\|^2$, help to reduce over-fitting by encouraging weight values to be close to zero. Another way to reduce network complexity is to impose hard constraints on the weights by forming them into groups and requiring that all weights within each group share the same value, in which the shared

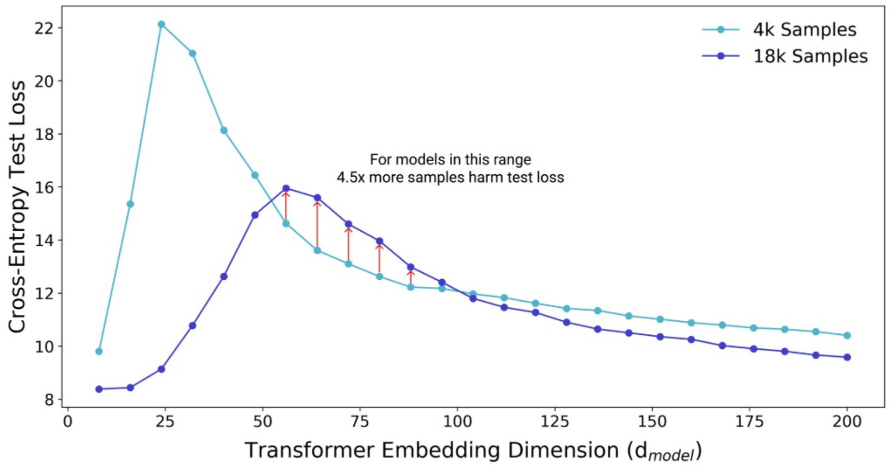

Figure 9.11 Plot of test set error for a large transformer model versus the embedding dimension, which controls the number of parameters in the model. Increasing the size of the training set from 4,000 to 18,000 samples generally leads to a lower test set error, but for some intermediate values of model complexity, there can be an increase in the error, as shown by the vertical red arrows. [From Nakkiran *et al.* (2019) with permission.]

value is learned from data. This is known as *weight sharing* or *parameter sharing* or *parameter tying*. It means that the number of degrees of freedom is smaller than the number of connections in the network. Usually this is introduced as a way to encode inductive bias into a network to express some known invariances. Evaluating the error function gradients for such networks can be done using a small modification to backpropagation although in practice this is handled implicitly through automatic differentiation. We will make extensive use of parameter sharing when we discuss convolutional neural networks. Parameter sharing is applicable, however, only to particular problems in which the form of the constraints can be specified in advance.

Exercise 9.7

Chapter 10

9.4.1 Soft weight sharing

Instead of using a hard constraint that forces sets of model parameters to be equal, Nowlan and Hinton (1992) introduced a form of *soft weight sharing* in which a regularization term encourages groups of weights to have similar values. Furthermore, the division of weights into groups, the mean weight value for each group, and the spread of values within the groups are all determined as part of the learning process.

Recall that the simple-weight decay regularizer in (9.1) can be viewed as the negative log of a Gaussian prior distribution over the weights. This encourages all the weights to converge towards a single value of zero. We can instead encourage the weight values to form several groups, rather than just one group, by considering a probability distribution that is a *mixture* of Gaussians. The means $\{\mu_j\}$ and variances $\{\sigma_j^2\}$ of the Gaussian components, as well as the mixing coefficients $\{\pi_j\}$, will be considered as adjustable parameters to be determined as part of the learning process.

Section 3.2.9

Thus, we have a probability density of the form

$$p(\mathbf{w}) = \prod_i \left\{ \sum_{j=1}^{K} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \right\} \tag{9.21}$$

where K is the number of components in the mixture. Taking the negative logarithm then leads to a regularization function of the form

$$\Omega(\mathbf{w}) = -\sum_i \ln \left(\sum_{j=1}^{K} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \right). \tag{9.22}$$

The total error function is then given by

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\Omega(\mathbf{w}) \tag{9.23}$$

where λ is the regularization coefficient.

This error is minimized jointly with respect to the weights $\{w_i\}$ and with respect to the parameters $\{\pi_j, \mu_j, \sigma_j\}$ of the mixture model. This can be done using gradient descent, which requires that we evaluate the derivatives of $\Omega(\mathbf{w})$ with respect to all the learnable parameters. To do this, it is convenient to regard the $\{\pi_j\}$ as *prior* probabilities for each component to have generated a weight value, and to introduce the corresponding posterior probabilities, which are given by Bayes' theorem:

Exercise 9.8

$$\gamma_j(w_i) = \frac{\pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2)}{\sum_k \pi_k \mathcal{N}(w_i|\mu_k, \sigma_k^2)}. \tag{9.24}$$

The derivatives of the total error function with respect to the weights are then given by

Exercise 9.9

$$\frac{\partial \widetilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda \sum_j \gamma_j(w_i) \frac{(w_i - \mu_j)}{\sigma_j^2}. \tag{9.25}$$

The effect of the regularization term is therefore to pull each weight towards the centre of the jth Gaussian, with a force proportional to the posterior probability of that Gaussian for the given weight. This is precisely the kind of effect that we are seeking.

Derivatives of the error with respect to the centres of the Gaussians are also easily computed to give

Exercise 9.10

$$\frac{\partial \widetilde{E}}{\partial \mu_j} = \lambda \sum_i \gamma_j(w_i) \frac{(\mu_j - w_i)}{\sigma_j^2} \tag{9.26}$$

which has a simple intuitive interpretation, because it pushes μ_j towards an average of the weight values, weighted by the posterior probabilities that the respective weight parameters were generated by component j.

To ensure that the variances $\{\sigma_j^2\}$ remain positive, we introduce new variables $\{\xi_j\}$ defined by

$$\sigma_j^2 = \exp(\xi_j) \tag{9.27}$$

and an unconstrained minimization is performed with respect to the $\{\xi_j\}$. The associated derivatives are then given by

Exercise 9.11

$$\frac{\partial \widetilde{E}}{\partial \xi} = \frac{\lambda}{2} \sum_i \gamma_j(w_i) \left(1 - \frac{(w_i - \mu_j)^2}{\sigma_j^2}\right). \tag{9.28}$$

This process drives σ_j towards a weighted average of the squared deviations of the weights around the corresponding centre μ_j, where the weighting coefficients are again given by the posterior probability that each weight is generated by component j.

For the derivatives with respect to the mixing coefficients π_j, we need to take account of the constraints

$$\sum_j \pi_j = 1, \qquad 0 \leqslant \pi_i \leqslant 1, \tag{9.29}$$

which follow from the interpretation of the π_j as prior probabilities. This can be done by expressing the mixing coefficients in terms of a set of auxiliary variables $\{\eta_j\}$ using the *softmax* function given by

$$\pi_j = \frac{\exp(\eta_j)}{\sum_{k=1}^K \exp(\eta_k)}. \tag{9.30}$$

The derivatives of the regularized error function with respect to the $\{\eta_j\}$ then take the form

Exercise 9.12

$$\frac{\partial \widetilde{E}}{\partial \eta_j} = \lambda \sum_i \{\pi_j - \gamma_j(w_i)\}. \tag{9.31}$$

We see that π_j is therefore driven towards the average posterior probability for mixture component j.

A different application of soft weight sharing (Lasserre, Bishop, and Minka, 2006) introduces a principled approach that combines the unsupervised training of a generative model with the supervised training of a corresponding discriminative model. This is useful in situations where we have a significant amount of unlabelled data but where labelled data is in short supply. The generative model has the advantage that all of the data can be used to determine its parameters, whereas only the labelled examples directly inform the parameters of the discriminative model. However, a discriminative model can achieve better generalization when there is model mis-specification, in other words when the model does not exactly describe the true distribution that generates the data, as is typically the case. By introducing a soft tying of the parameters of the two models, we obtain a well-defined hybrid of generative and discriminative approaches that can be robust to model mis-specification while also benefiting from being trained on unlabelled data.

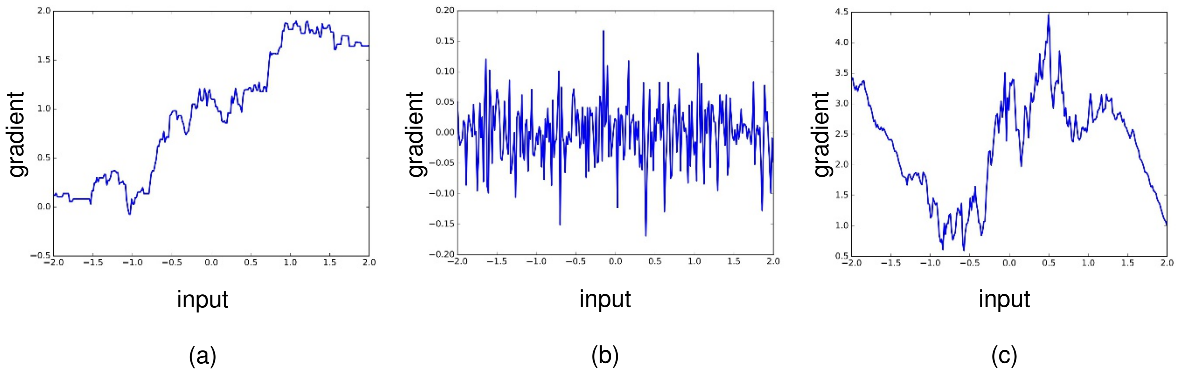

Figure 9.12 Plots of the Jacobian for networks with a single input and a single output, showing (a) a network with two layers of weights, (b) a network with 25 layers of weights, and (c) a network with 51 layers of weights together with residual connections. [From Balduzzi *et al.* (2017) with permission.]

9.5. Residual Connections

The representational power of deep neural networks stems in large part from the use of multiple layers of processing, and it has been observed that increasing the number of layers in a network can increase generalization performance significantly. We have also seen how batch normalization, along with careful initialization of the weights and biases, can help address the problem of vanishing or exploding gradients in deep networks. However, even with batch normalization, it becomes increasingly difficult to train networks with a large number of layers.

Section 7.4.2
Section 7.2.5

One explanation for this phenomenon is called *shattered gradients* (Balduzzi *et al.*, 2017). We have seen that the representational capabilities of neural networks increase exponentially with depth. With ReLU activation functions, there is an exponential increase in the number of linear regions that the network can represent. However, a consequence of this is a proliferation of discontinuities in the gradient of the error function. This is illustrated for networks with a single input variable and a single output variable in Figure 9.12. Here the derivative of the output variable with respect to the input variable (the Jacobian of the network) is plotted as a function of the input variable. From the chain rule of calculus, these derivatives determine the gradients of the error function surface. We see that for deep networks, extremely small changes in the weight parameters in the early layers of the network can produce significant changes in the gradient. Iterative gradient-based optimization algorithms assume that the gradient varies smoothly across parameter space, and hence this 'shattered gradient' effect can render training ineffective in very deep networks.

Section 6.3

An important modification to the architecture of neural networks that greatly assists in training very deep networks is that of *residual connections* (He *et al.*, 2015a), which are a particular form of *skip-layer connections*. Consider a neural network

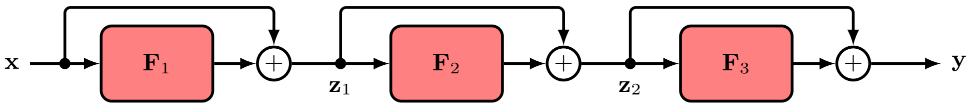

Figure 9.13 A residual network consisting of three residual blocks, corresponding to the sequence of transformations (9.35) to (9.37).

that consists of a sequence of three layers of the form

$$\mathbf{z}_1 = \mathbf{F}_1(\mathbf{x}) \tag{9.32}$$
$$\mathbf{z}_2 = \mathbf{F}_2(\mathbf{z}_1) \tag{9.33}$$
$$\mathbf{y} = \mathbf{F}_3(\mathbf{z}_2). \tag{9.34}$$

Here the functions $\mathbf{F}_l(\cdot)$ might simply consist of a linear transformation followed by a ReLU activation function or they might be more complex with multiple linear, activation function, and normalization layers. A residual connection consists simply of adding the input to each function back onto the output to give

$$\mathbf{z}_1 = \mathbf{F}_1(\mathbf{x}) + \mathbf{x} \tag{9.35}$$
$$\mathbf{z}_2 = \mathbf{F}_2(\mathbf{z}_1) + \mathbf{z}_1 \tag{9.36}$$
$$\mathbf{y} = \mathbf{F}_3(\mathbf{z}_2) + \mathbf{z}_2. \tag{9.37}$$

Each combination of a function and a residual connection, such as $\mathbf{F}_1(\mathbf{x}) + \mathbf{x}$, is called a *residual block*. A *residual network*, also known as a *ResNet*, consists of multiple layers of such blocks in sequence. A modified network with residual connections is illustrated in Figure 9.13. A residual block can easily generate the identity transformation, if the parameters in the nonlinear function are small enough for the function outputs to become close to zero.

The term 'residual' refers to the fact that in each block the function learns the residual between the identity map and the desired output, which we can see by rearranging the residual transformation:

$$\mathbf{F}_l(\mathbf{z}_{l-1}) = \mathbf{z}_l - \mathbf{z}_{l-1}. \tag{9.38}$$

The gradients in a network with residual connections are much less sensitive to input values compared to a standard deep network, as seen in Figure 9.12(c).

Li *et al.* (2017) developed a way to visualize error surfaces directly, which showed that the effect of the residual connections is to create smoother error function surfaces, as shown in Figure 9.14. It is usual to include batch normalization layers in a residual network, as together they significantly reduce the issue of vanishing and exploding gradients. He *et al.* (2015a) showed that including residual connections allows very deep networks, potentially having hundreds of layers, to be trained effectively.

Further insight into the way residual connections encourage smooth error surfaces can be obtained if we combine (9.35), (9.36), and (9.37) to give a single overall equation for the whole network:

$$\mathbf{y} = \mathbf{F}_3(\mathbf{F}_2(\mathbf{F}_1(\mathbf{x}) + \mathbf{x}) + \mathbf{z}_1) + \mathbf{z}_2. \tag{9.39}$$

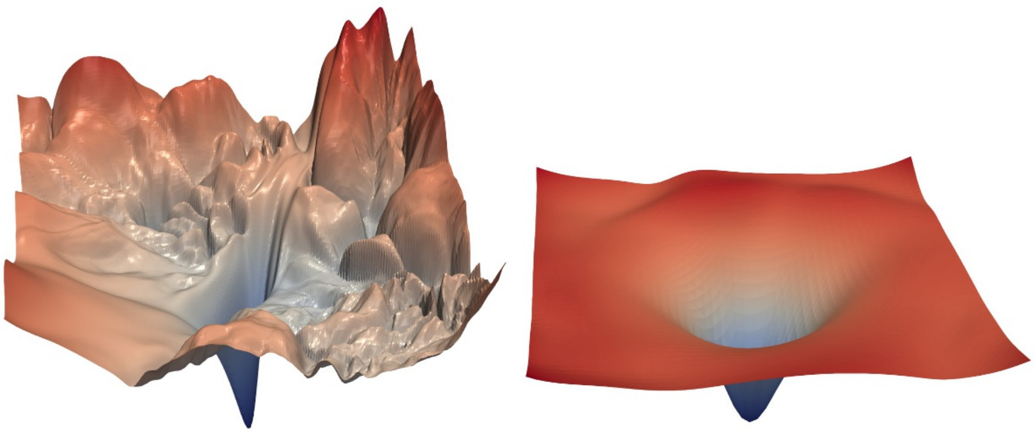

Figure 9.14 (a) A visualization of the error surface for a network with 56 layers. (b) The same network with the inclusion of residual connections, showing the smoothing effect that comes from the residual connections. [From Li *et al.* (2017) with permission.]

Exercise 9.13

We can now substitute for the intermediate variables $\mathbf{z}_1$ and $\mathbf{z}_2$ to give an expression for the network output as a function of the input $\mathbf{x}$:

$$\begin{aligned}\mathbf{y} = \mathbf{F}_3(&\mathbf{F}_2(\mathbf{F}_1(\mathbf{x}) + \mathbf{x}) + \mathbf{F}_1(\mathbf{x}) + \mathbf{x}) \\ &+ \mathbf{F}_2(\mathbf{F}_1(\mathbf{x}) + \mathbf{x})) \\ &+ \mathbf{F}_1(\mathbf{x}) + \mathbf{x}. \end{aligned} \tag{9.40}$$

This expanded form of the residual network is depicted in Figure 9.15. We see that the overall function consists of multiple networks acting in parallel and that these include networks with fewer layers. The network has the representational capability of a deep network, since it contains such a network as a special case. However, the error surface is moderated by a combination of shallow and deep sub-networks.

Note that the skip-layer connections defined by (9.40) require the input and all the intermediate variables to have the same dimensionality so that they can be added. We can change the dimensionality at some point in the network by including a non-square matrix $\mathbf{W}$ of learnable parameters in the form

$$\mathbf{z}_l = \mathbf{F}_l(\mathbf{z}_{l-1}) + \mathbf{W}\mathbf{z}_{l-1}. \tag{9.41}$$

So far we have not been specific about the form of the learnable nonlinear functions $\mathbf{F}_l(\cdot)$. The simplest choice would be a standard neural network that alternates between layers consisting of a learnable linear transformation and a fixed nonlinear activation function such as ReLU. This opens two possibilities for placing the residual connections, as shown in Figure 9.16. In version (a) the quantities being added are always non-negative since they are given by the outputs of ReLU layers, and so to allow for both positive and negative values, version (b) is more commonly used.

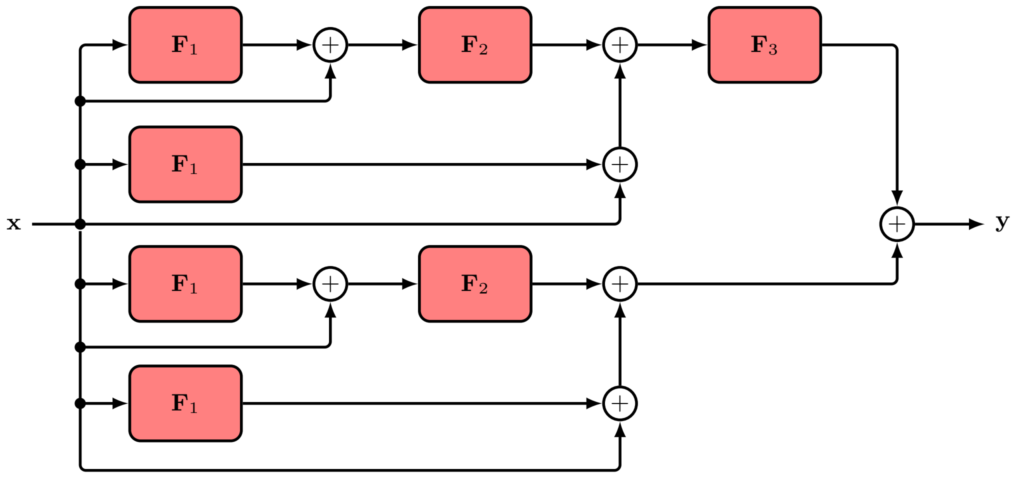

Figure 9.15 The same network as in Figure 9.13, shown here in expanded form.

9.6. Model Averaging

If we have several different models trained to solve the same problem then instead of trying to select the single best model, we can often improve generalization by averaging the predictions made by the individual models. Such combinations of models are sometimes called *committees* or *ensembles*. For models that produce probabilistic outputs, the predicted distribution is the average of the predictions from each model:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{L}\sum_{l=1}^{L} p_l(\mathbf{y}|\mathbf{x}) \tag{9.42}$$

where $p_l(\mathbf{y}|\mathbf{x})$ is the output of model l and L is the total number of models.

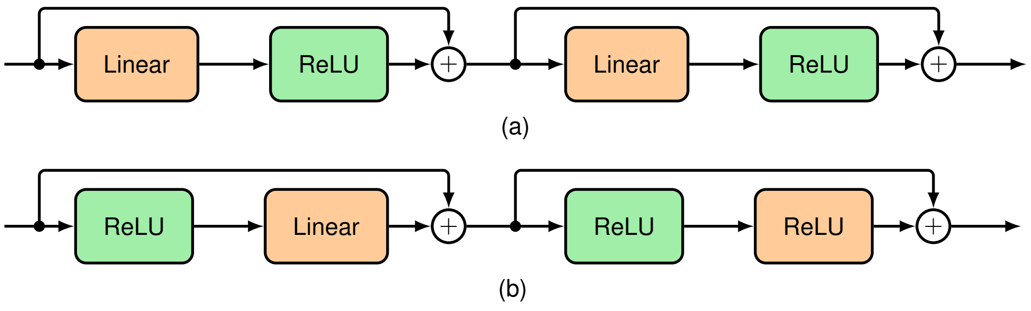

Figure 9.16 Two alternative ways to include residual network connections into a standard feed-forward network that alternates between learnable linear layers and nonlinear ReLU activation functions.

Section 4.3

This averaging process can be motivated by considering the trade-off between bias and variance. Recall from Figure 4.7 that when we trained multiple polynomials using the sinusoidal data and then averaged the resulting functions, the contribution arising from the variance term tended to cancel, leading to improved predictions.

In practice, of course, we have only a single data set, and so we have to find a way to introduce variability between the different models within the committee. One approach is to use *bootstrap* data sets, in which multiple data sets are created as follows. Suppose our original data set consists of N data points $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. We can create a new data set $\mathbf{X}_\mathrm{B}$ by drawing N points at random from $\mathbf{X}$, with replacement, so that some points in $\mathbf{X}$ may be replicated in $\mathbf{X}_\mathrm{B}$, whereas other points in $\mathbf{X}$ may be absent from $\mathbf{X}_\mathrm{B}$. This process can be repeated L times to generate L data sets each of size N and each obtained by sampling from the original data set $\mathbf{X}$. Each data set can then be used to train a model, and the predictions of the resulting models are averaged. This procedure is known as bootstrap aggregation or *bagging* (Breiman, 1996). An alternative approach to forming an ensemble is to use the original data set to train multiple different models having different architectures.

We can analyse the benefits of ensemble predictions by considering a regression problem with an input vector $\mathbf{x}$ and a single output variable y. Suppose we have a set of trained models $y_1(\mathbf{x}), \ldots, y_M(\mathbf{x})$, and we form a committee prediction given by

$$y_{\mathrm{COM}}(\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} y_m(\mathbf{x}). \tag{9.43}$$

If the true function that we are trying to predict is given by $h(\mathbf{x})$, then the output of each of the models can be written as the true value plus an error:

$$y_m(\mathbf{x}) = h(\mathbf{x}) + \epsilon_m(\mathbf{x}). \tag{9.44}$$

The average sum-of-squares error then takes the form

$$\mathbb{E}_{\mathbf{x}}\left[\{y_m(\mathbf{x}) - h(\mathbf{x})\}^2\right] = \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right] \tag{9.45}$$

where $\mathbb{E}_{\mathbf{x}}[\cdot]$ denotes a frequentist expectation with respect to the distribution of the input vector $\mathbf{x}$. The average error made by the models acting individually is therefore

$$E_{\mathrm{AV}} = \frac{1}{M}\sum_{m=1}^{M}\mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right]. \tag{9.46}$$

Similarly, the expected error from the committee (9.43) is given by

$$\begin{aligned} E_{\mathrm{COM}} &= \mathbb{E}_{\mathbf{x}}\left[\left\{\frac{1}{M}\sum_{m=1}^{M} y_m(\mathbf{x}) - h(\mathbf{x})\right\}^2\right] \\ &= \mathbb{E}_{\mathbf{x}}\left[\left\{\frac{1}{M}\sum_{m=1}^{M} \epsilon_m(\mathbf{x})\right\}^2\right]. \end{aligned} \tag{9.47}$$

If we assume that the errors have zero mean and are uncorrelated, so that

$$\begin{aligned} \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})\right] &= 0 & (9.48) \\ \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})\epsilon_l(\mathbf{x})\right] &= 0, \qquad m \neq l & (9.49) \end{aligned}$$

Exercise 9.14

then we obtain

$$E_{\mathrm{COM}} = \frac{1}{M} E_{\mathrm{AV}}. \tag{9.50}$$

This apparently dramatic result suggests that the average error of a model can be reduced by a factor of M simply by averaging M versions of the model. Unfortunately, it depends on the key assumption that the errors due to the individual models are uncorrelated. In practice, the errors are typically highly correlated, and the reduction in the overall error is generally much smaller. It can, however, be shown that the expected committee error will not exceed the expected error of the constituent models, so that $E_{\mathrm{COM}} \leqslant E_{\mathrm{AV}}$.

Exercise 9.15

A somewhat different approach to model combination, known as *boosting* (Freund and Schapire, 1996), combines multiple 'base' classifiers to produce a form of committee whose performance can be significantly better than that of any of the base classifiers. Boosting can give good results even if the base classifiers perform only slightly better than random. The principal difference between boosting and the committee methods, such as bagging as discussed above, is that the base classifiers are trained in sequence and each base classifier is trained using a weighted form of the data set in which the weighting coefficient associated with each data point depends on the performance of the previous classifiers. In particular, points that are misclassified by one of the base classifiers are given a greater weight when used to train the next classifier in the sequence. Once all the classifiers have been trained, their predictions are then combined through a weighted majority voting scheme.

In practice, the major drawback of all model combination methods is that multiple models have to be trained and then predictions have to be evaluated for all the models, thereby increasing the computational cost of both training and inference. How significant this depends on the specific application scenario.

9.6.1 Dropout

A widely used and very effective form of regularization known as *dropout* (Srivastava *et al.*, 2014) can be viewed as an implicit way to perform approximate model averaging over exponentially many models without having to train multiple models individually. It has broad applicability and is computationally cheap. Dropout is one of the most effective forms of regularization and is widely used in applications.

The central idea of dropout is to delete nodes from the network, including their connections, at random during training. Each time a data point is presented to the network, a new random choice is made for which nodes to omit. Figure 9.17 shows a simple network along with examples of pruned networks in which subsets of nodes have been omitted.

Dropout is applied to both hidden nodes and input nodes, but not outputs, and is equivalent to setting the output of a dropped node to zero. It can be implemented by defining a mask vector $R_i \in \{0, 1\}$ which multiplies the activation of the non-output

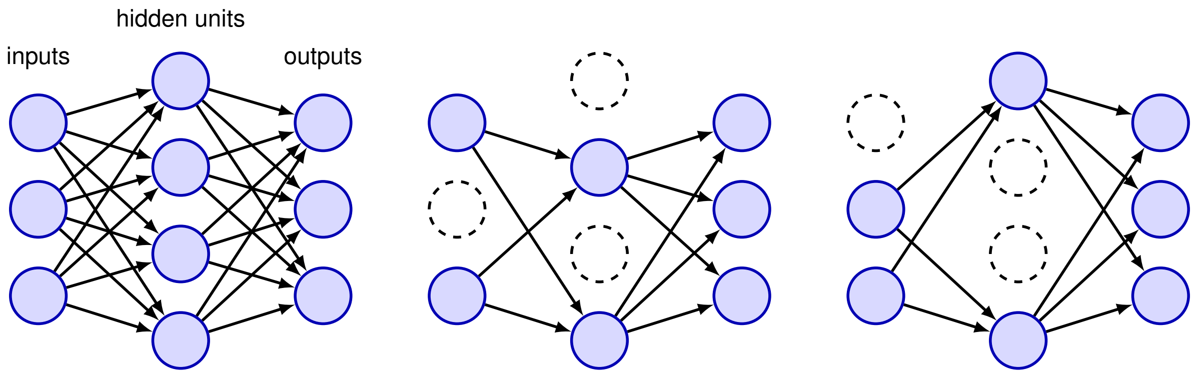

Figure 9.17 A neural network on the left along with two examples of pruned networks in which a random subset of nodes have been omitted.

node i for data point n, whose values are set to 1 with probability ρ. A value of $\rho = 0.5$ seems to work well for the hidden nodes, whereas for the inputs a value of $\rho = 0.8$ is typically used.

Section 7.2.4

During training, as each data point is presented to the network, a new mask is created, and the forward and backward propagation steps are applied on that pruned network to create error function gradients, which are then used to update the weights, for example by stochastic gradient descent. If the data points are grouped into mini-batches then the gradients are averaged over the data points in each mini-batch before applying the weight update. For a network with M non-output nodes, there are 2^M pruned networks, and so only a small fraction of these networks will ever be considered during training. This differs from conventional ensemble methods in which each of the networks in the ensemble is independently trained to convergence. Another difference is that the exponentially many networks that are implicitly being trained with dropout are not independent but share their parameter values with the full network, and hence with each other. Note that training can take longer with dropout since the individual parameter updates are very noisy. Also, because the error function is intrinsically noisy, it is harder to confirm that the optimization algorithm is working correctly just by looking for a decreasing error function during training.

Once training is complete, predictions can in principle be made by applying the ensemble rule (9.42), which in this case takes the form

$$p(\mathbf{y}|\mathbf{x}) = \sum_{\mathbf{R}} p(\mathbf{R}) p(\mathbf{y}|\mathbf{x}, \mathbf{R}) \tag{9.51}$$

where the sum is over the exponentially large space of masks, and $p(\mathbf{y}|\mathbf{x}, \mathbf{R})$ is the predictive distribution from the network with mask $\mathbf{R}$. Because this summation is intractable, it can be approximated by sampling a small number of masks, and in practice, as few as 10 or 20 masks can be sufficient to obtain good results. This procedure is known as *Monte Carlo dropout*.

An even simpler approach is to make predictions using the trained network with no nodes masked out, and to re-scale the weights in the network so that the expected input to each node is roughly the same during testing as it would be during training, compensating for the fact that in training a proportion of the nodes would be missing. Thus, if a node is present with probability ρ during training, then during testing the output weights from that node would be multiplied by ρ before using the network to make predictions.

Section 2.6

A different motivation for dropout comes from the Bayesian perspective. In a fully Bayesian treatment, we would make predictions by averaging over all possible 2^M network models, with each network weighted by its posterior probability. Computationally, this would be prohibitively expensive, both during training when evaluating the posterior probabilities and during testing when computing the weighted predictions. Dropout approximates this model averaging by giving an equal weight to each possible model.

Exercise 9.18

Further intuition behind dropout comes from its role in reducing over-fitting. In a standard network, the parameters can become tuned to noise on individual data points, with hidden nodes becoming over-specialized. Each node adjusts its weights to minimize the error, given the outputs of other nodes, leading to co-adaptation of nodes in a way that might not generalize to new data. With dropout, each node cannot rely on the presence of other specific nodes and must instead make useful contributions in a broad range of contexts, thereby reducing co-adaptation and specialization. For a simple linear regression model trained using least squares, dropout regularization is equivalent to a modified form of quadratic regularization.

Exercises

Section 9.1.3

9.1 ($\star$) By considering each of the four group axioms in turn, show that the set of all possible rotations of a square through (positive or negative) multiples of $90°$, together with the binary operation of composing rotations, forms a group. Similarly, show that the set of all continuous translations of an object in a two-dimensional plane also forms a group.

9.2 ($\star\star$) Consider a linear model of the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i \tag{9.52}$$

together with a sum-of-squares error function of the form

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 . \tag{9.53}$$

Now suppose that Gaussian noise ϵ_i with zero mean and variance σ^2 is added independently to each of the input variables x_i. By making use of $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{E}[\epsilon_i \epsilon_j] = \delta_{ij}\sigma^2$, show that minimizing E_D averaged over the noise distribution is