Chapter 9

Regularization

Chapter 8 described how to measure model performance and identified that there could be a significant performance gap between the training and test data. Possible reasons for this discrepancy include: (i) the model describes statistical peculiarities of the training data that are not representative of the true mapping from input to output (overfitting), and (ii) the model is unconstrained in areas with no training examples, leading to suboptimal predictions.

This chapter discusses *regularization* techniques. These are a family of methods that reduce the generalization gap between training and test performance. Strictly speaking, regularization involves adding explicit terms to the loss function that favor certain parameter choices. However, in machine learning, this term is commonly used to refer to any strategy that improves generalization.

We start by considering regularization in its strictest sense. Then we show how the stochastic gradient descent algorithm itself favors certain solutions. This is known as implicit regularization. Following this, we consider a set of heuristic methods that improve test performance. These include early stopping, ensembling, dropout, label smoothing, and transfer learning.

9.1 Explicit regularization

Consider fitting a model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ using a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek the parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss function $L[\boldsymbol{\phi}]$:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\big[L[\boldsymbol{\phi}]\big] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, \mathbf{y}_i]\right], \end{aligned} \tag{9.1}$$

where the individual terms $\ell_i[\mathbf{x}_i, \mathbf{y}_i]$ measure the mismatch between the network predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and output targets $\mathbf{y}_i$ for each training pair. To bias this minimization

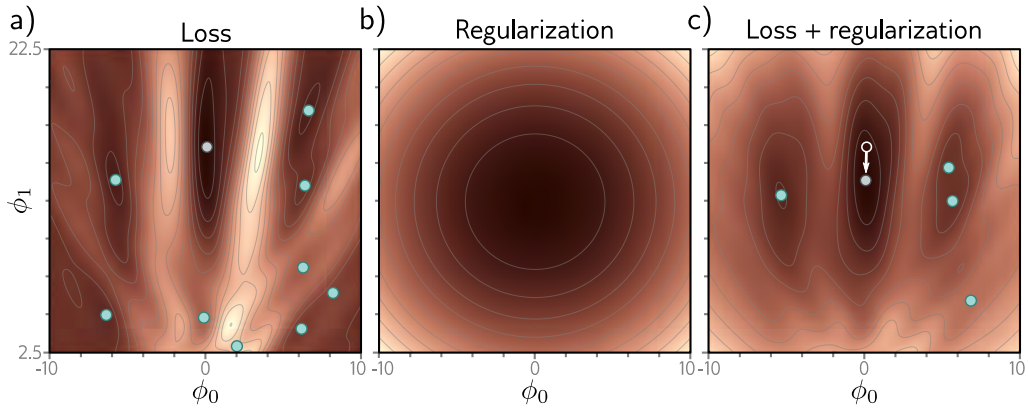

Figure 9.1 Explicit regularization. a) Loss function for Gabor model (see section 6.1.2). Cyan circles represent local minima. Gray circle represents the global minimum. b) The regularization term favors parameters close to the center of the plot by adding an increasing penalty as we move away from this point. c) The final loss function is the sum of the original loss function plus the regularization term. This surface has fewer local minima, and the global minimum has moved to a different position (arrow shows change).

toward certain solutions, we include an additional term:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, \mathbf{y}_i] + \lambda \cdot \mathrm{g}[\boldsymbol{\phi}] \right], \tag{9.2}$$

where $\mathrm{g}[\boldsymbol{\phi}]$ is a function that returns a scalar which takes larger values when the parameters are less preferred. The term λ is a positive scalar that controls the relative contribution of the original loss function and the regularization term. The minima of the regularized loss function usually differ from those in the original, so the training procedure converges to different parameter values (figure 9.1).

9.1.1 Probabilistic interpretation

Regularization can be viewed from a probabilistic perspective. Section 5.1 shows how loss functions are constructed from the maximum likelihood criterion:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) \right]. \tag{9.3}$$

The regularization term can be considered as a *prior* $Pr(\boldsymbol{\phi})$ that represents knowledge about the parameters before we observe the data and we now have the *maximum a posteriori* or *MAP* criterion:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}) \right]. \tag{9.4}$$

Moving back to the negative log-likelihood loss function by taking the log and multiplying by minus one, we see that $\lambda \cdot \mathrm{g}[\boldsymbol{\phi}] = -\log[Pr(\boldsymbol{\phi})]$.

9.1.2 L2 regularization

This discussion has sidestepped the question of *which* solutions the regularization term should penalize (or equivalently that the prior should favor). Since neural networks are used in an extremely broad range of applications, these can only be very generic preferences. The most commonly used regularization term is the *L2 norm*, which penalizes the sum of the squares of the parameter values:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, \mathbf{y}_i] + \lambda \sum_j \phi_j^2 \right], \tag{9.5}$$

Problems 9.1–9.2

where j indexes the parameters. This is also referred to as *Tikhonov regularization* or *ridge regression*, or (when applied to matrices) *Frobenius norm regularization*.

For neural networks, L2 regularization is usually applied to the weights but not the biases and is hence referred to as a *weight decay* term. The effect is to encourage smaller weights, so the output function is smoother. To see this, consider that the output prediction is a weighted sum of the activations at the last hidden layer. If the weights have a smaller magnitude, the output will vary less. The same logic applies to the computation of the pre-activations at the last hidden layer and so on, progressing backward through the network. In the limit, if we forced all the weights to be zero, the network would produce a constant output determined by the final bias parameter.

Notebook 9.1
L2 regularization

Figure 9.2 shows the effect of fitting the simplified network from figure 8.4 with weight decay and different values of the regularization coefficient λ. When λ is small, it has little effect. However, as λ increases, the fit to the data becomes less accurate, and the function becomes smoother. This might improve the test performance for two reasons:

- If the network is overfitting, then adding the regularization term means that the network must trade off slavish adherence to the data against the desire to be smooth. One way to think about this is that the error due to variance reduces (the model no longer needs to pass through every data point) at the cost of increased bias (the model can only describe smooth functions).
- When the network is over-parameterized, some of the extra model capacity describes areas with no training data. Here, the regularization term will favor functions that smoothly interpolate between the nearby points. This is reasonable behavior in the absence of knowledge about the true function.

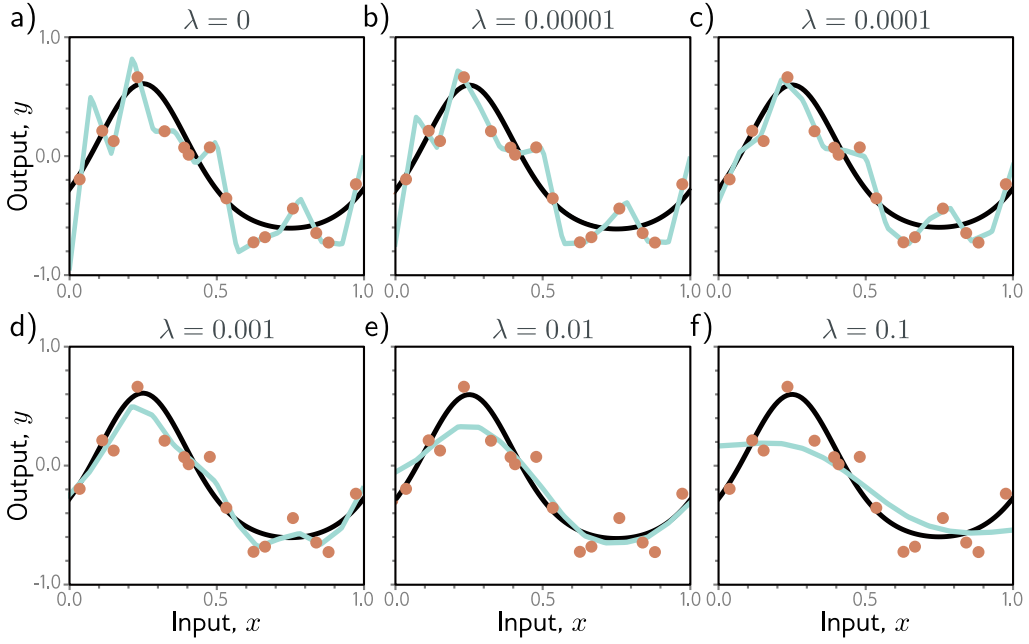

Figure 9.2 L2 regularization in simplified network with 14 hidden units (see figure 8.4). a–f) Fitted functions as we increase the regularization coefficient λ. The black curve is the true function, the orange circles are the noisy training data, and the cyan curve is the fitted model. For small λ (panels a–b), the fitted function passes exactly through the data points. For intermediate λ (panels c–d), the function is smoother and more similar to the ground truth. For large λ (panels e–f), the regularization term overpowers the likelihood term, so the fitted function is too smooth and the overall fit is worse.

9.2 Implicit regularization

An intriguing recent finding is that neither gradient descent nor stochastic gradient descent moves neutrally to the minimum of the loss function; each exhibits a preference for some solutions over others. This is known as *implicit regularization*.

9.2.1 Implicit regularization in gradient descent

Consider a continuous version of gradient descent where the step size is infinitesimal. The change in parameters $\boldsymbol{\phi}$ will be governed by the differential equation:

$$\frac{d\boldsymbol{\phi}}{dt} = -\frac{\partial L}{\partial \boldsymbol{\phi}}. \tag{9.6}$$

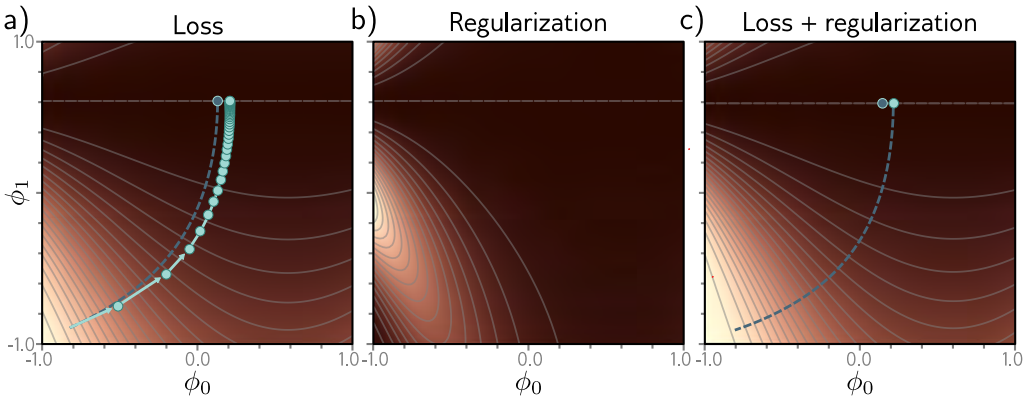

Figure 9.3 Implicit regularization in gradient descent. a) Loss function with family of global minima on horizontal line $\phi_1 = 0.61$. Dashed blue line shows continuous gradient descent path starting in bottom-left. Cyan trajectory shows discrete gradient descent with step size 0.1 (first few steps shown explicitly as arrows). The finite step size causes the paths to diverge and reach a different final position. b) This disparity can be approximated by adding a regularization term to the continuous gradient descent loss function that penalizes the squared gradient magnitude. c) After adding this term, the continuous gradient descent path converges to the same place that the discrete one did on the original function.

Gradient descent approximates this process with a series of discrete steps of size α:

$$\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}_t - \alpha \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{9.7}$$

The discretization causes a deviation from the continuous path (figure 9.3).

This deviation can be understood by deriving a modified loss term $\tilde{L}$ for the continuous case that arrives at the same place as the discretized version on the original loss L. It can be shown (see end of chapter) that this modified loss is:

$$\tilde{L}_{GD}[\boldsymbol{\phi}] = L[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\| \frac{\partial L}{\partial \boldsymbol{\phi}} \right\|^2. \tag{9.8}$$

In other words, the discrete trajectory is repelled from places where the gradient norm is large (the surface is steep). This doesn't change the position of the minima where the gradients are zero anyway. However, it changes the effective loss function elsewhere and modifies the optimization trajectory, which potentially converges to a different minimum. Implicit regularization due to gradient descent may be responsible for the observation that full batch gradient descent generalizes better with larger step sizes (figure 9.5a).

9.2.2 Implicit regularization in stochastic gradient descent

A similar analysis can be applied to stochastic gradient descent. Now we seek a modified loss function such that the continuous version reaches the same place as the average of the possible random SGD updates. This can be shown to be:

$$\begin{aligned}\tilde{L}_{SGD}[\boldsymbol{\phi}] &= \tilde{L}_{GD}[\boldsymbol{\phi}] + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 \\ &= L[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2. \end{aligned} \tag{9.9}$$

Here, L_b is the loss for the b^{th} of the B batches in an epoch, and both L and L_b now represent the means of the I individual losses in the full dataset and the $|\mathcal{B}|$ individual losses in the batch, respectively:

$$L = \frac{1}{I}\sum_{i=1}^{I}\ell_i[\mathbf{x}_i, y_i] \qquad \text{and} \qquad L_b = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_b}\ell_i[\mathbf{x}_i, y_i]. \tag{9.10}$$

Equation 9.9 reveals an extra regularization term, which corresponds to the variance of the gradients of the batch losses L_b. In other words, SGD implicitly favors places where the gradients are stable (where all the batches agree on the slope). Once more, this modifies the trajectory of the optimization process (figure 9.4) but does not necessarily change the position of the global minimum; if the model is over-parameterized, then it may fit all the training data exactly, so each of these gradient terms will be zero at the global minimum.

SGD generalizes better than gradient descent, and smaller batch sizes generally perform better than larger ones (figure 9.5b). One possible explanation is that the inherent randomness allows the algorithm to reach different parts of the loss function. However, it's also possible that some or all of this performance increase is due to implicit regularization; this encourages solutions where all the data fits well (so the batch variance is small) rather than solutions where some of the data fit extremely well and other data less well (perhaps with the same overall loss, but with larger batch variance). The former solutions are likely to generalize better.

Notebook 9.2 Implicit regularization

9.3 Heuristics to improve performance

We've seen that explicit regularization encourages the training algorithm to find a good solution by adding extra terms to the loss function. This also occurs implicitly as an unintended (but seemingly helpful) byproduct of stochastic gradient descent. This section describes other heuristic methods used to improve generalization.

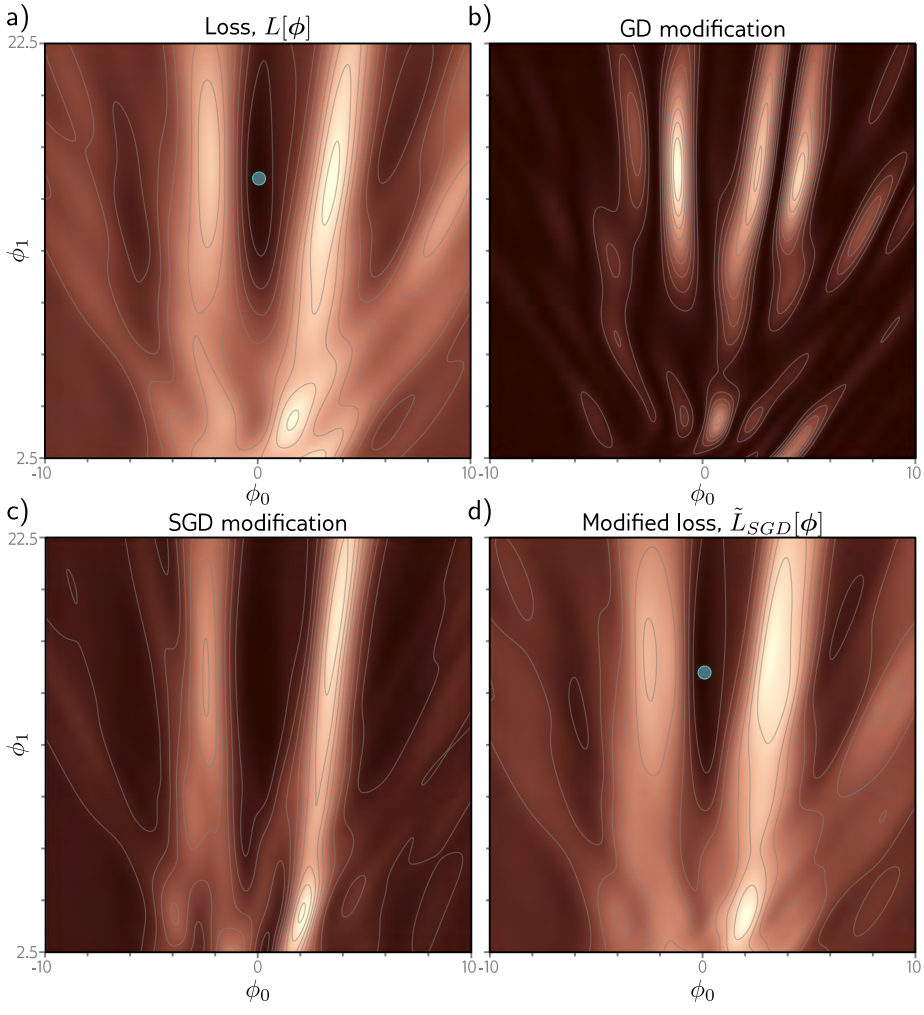

Figure 9.4 Implicit regularization for stochastic gradient descent. a) Original loss function for Gabor model (section 6.1.2). Blue point represents global minimum. b) Implicit regularization term from gradient descent penalizes the squared gradient magnitude. c) Additional implicit regularization from stochastic gradient descent penalizes the variance of the batch gradients. d) Modified loss function (sum of original loss plus two implicit regularization components). Blue point represents global minimum which may now be in a different place from panel (a).

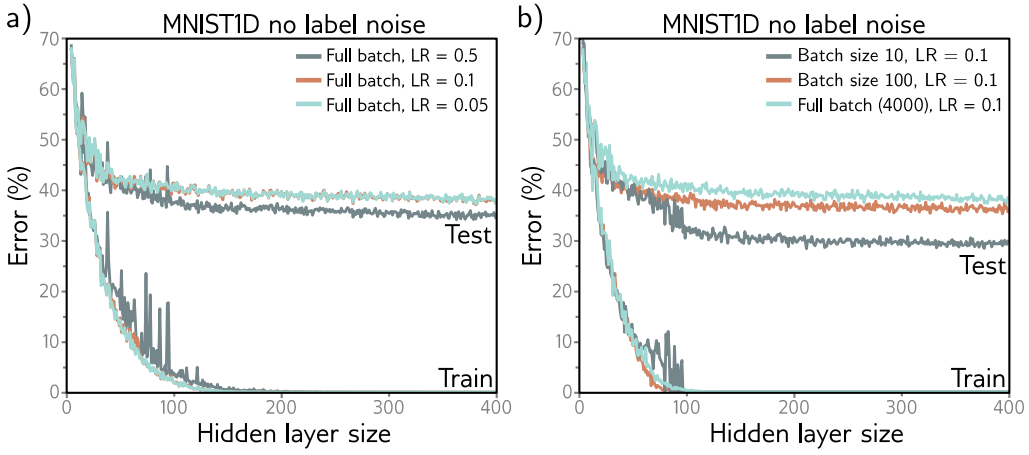

Figure 9.5 Effect of learning rate (LR) and batch size for 4000 training and 4000 test examples from MNIST-1D (see figure 8.1) for a neural network with two hidden layers. a) Performance is better for large learning rates than for intermediate or small ones. In each case, the number of iterations is 6000/LR, so each solution has the opportunity to move the same distance. b) Performance is superior for smaller batch sizes. In each case, the number of iterations was chosen so that the training data were memorized at roughly the same model capacity.

9.3.1 Early stopping

Early stopping refers to stopping the training procedure before it has fully converged. This can reduce overfitting if the model has already captured the coarse shape of the underlying function but has not yet had time to overfit to the noise (figure 9.6). One way of thinking about this is that since the weights are initialized to small values (see section 7.5), they simply don't have time to become large, so early stopping has a similar effect to explicit L2 regularization. A different view is that early stopping reduces the effective model complexity. Hence, we move back down the bias/variance trade-off curve from the critical region, and performance improves (see figures 8.9 and 8.10).

Early stopping has a single hyperparameter, the number of steps after which learning is terminated. As usual, this is chosen empirically using a validation set (section 8.5). However, for early stopping, the hyperparameter can be selected without the need to train multiple models. The model is trained once, the performance on the validation set is monitored every T iterations, and the associated parameters are stored. The stored parameters where the validation performance was best are selected.

9.3.2 Ensembling

Another approach to reducing the generalization gap between training and test data is to build several models and average their predictions. A group of such models is known

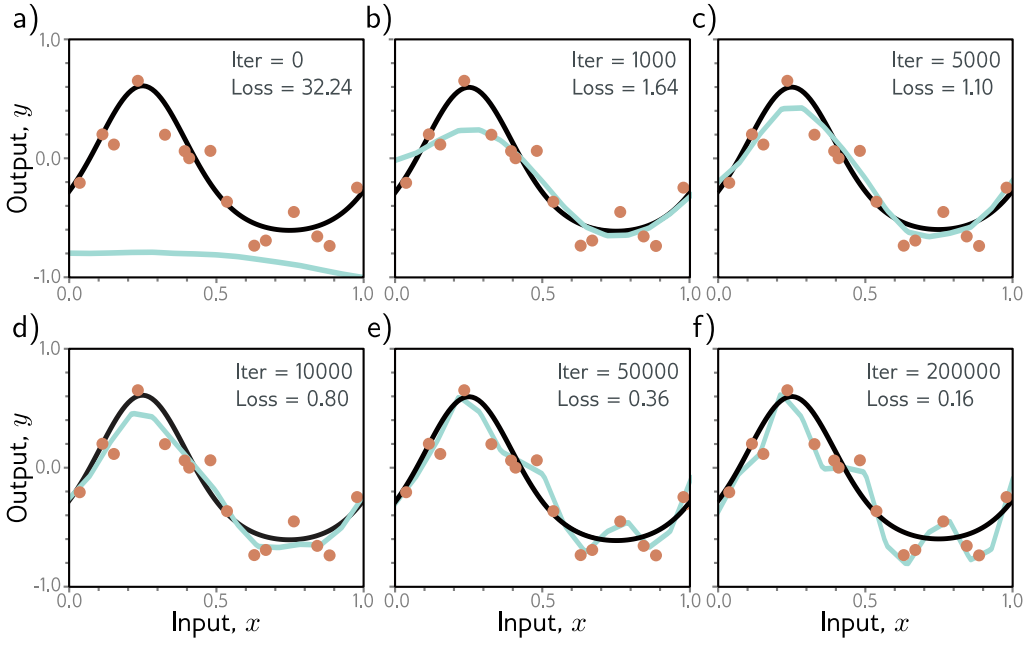

Figure 9.6 Early stopping. a) Simplified shallow network model with 14 linear regions (figure 8.4) is initialized randomly (cyan curve) and trained with SGD using a batch size of five and a learning rate of 0.05. b–d) As training proceeds, the function first captures the coarse structure of the true function (black curve) before e–f) overfitting to the noisy training data (orange points). Although the training loss continues to decrease throughout this process, the learned models in panels (c) and (d) are closest to the true underlying function. They will generalize better on average to test data than those in panels (e) or (f).

as an *ensemble*. This technique reliably improves test performance at the cost of training and storing multiple models and performing inference multiple times.

The models can be combined by taking the mean of the outputs (for regression problems) or the mean of the pre-softmax activations (for classification problems). The assumption is that model errors are independent and will cancel out. Alternatively, we can take the median of the outputs (for regression problems) or the most frequent predicted class (for classification problems) to make the predictions more robust.

Notebook 9.3
Ensembling

One way to train different models is just to use different random initializations. This may help in regions of input space far from the training data. Here, the fitted function is relatively unconstrained, and different models may produce different predictions, so the average of several models may generalize better than any single model.

A second approach is to generate several different datasets by re-sampling the training data with replacement and training a different model from each. This is known as *bootstrap aggregating* or *bagging* for short (figure 9.7). It has the effect of smoothing out the data; if a data point is not present in one training set, the model will interpo-

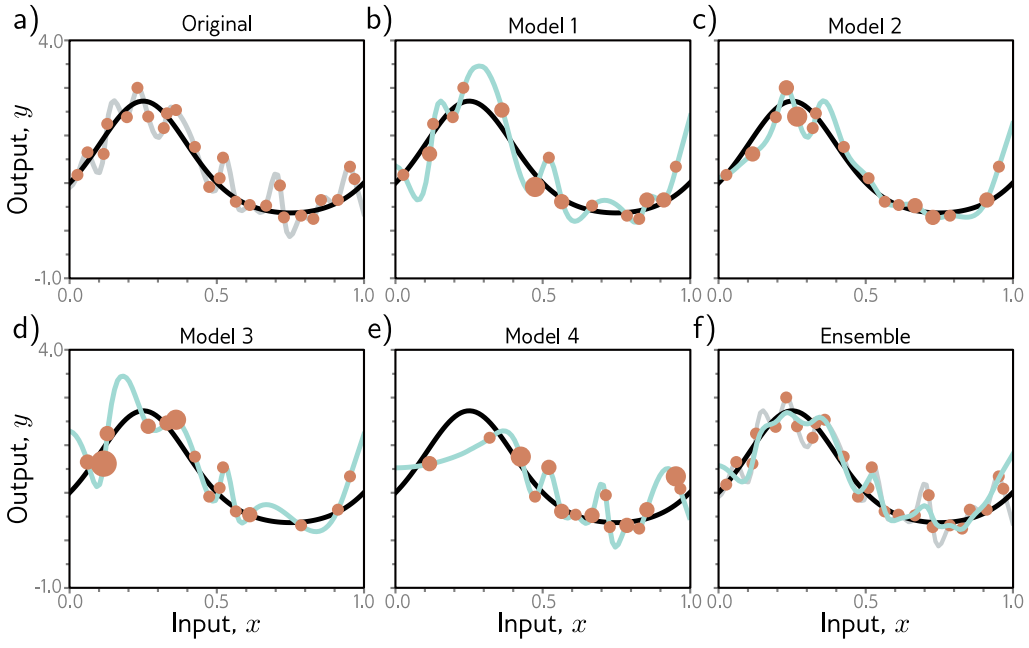

Figure 9.7 Ensemble methods. a) Fitting a single model (gray curve) to the entire dataset (orange points). b–e) Four models created by re-sampling the data with replacement (bagging) four times (size of orange point indicates number of times the data point was re-sampled). f) When we average the predictions of this ensemble, the result (cyan curve) is smoother than the result from panel (a) for the full dataset (gray curve) and will probably generalize better.

late from nearby points; hence, if that point was an outlier, the fitted function will be more moderate in this region. Other approaches include training models with different hyperparameters or training completely different families of models.

9.3.3 Dropout

Dropout clamps a random subset (typically 50%) of hidden units to zero at each iteration of SGD (figure 9.8). This makes the network less dependent on any given hidden unit and encourages the weights to have smaller magnitudes so that the change in the function due to the presence or absence of any specific hidden unit is reduced.

This technique has the positive benefit that it can eliminate undesirable "kinks" in the function that are far from the training data and don't affect the loss. For example, consider three hidden units that become active sequentially as we move along the curve (figure 9.9a). The first hidden unit causes a large increase in the slope. A second hidden

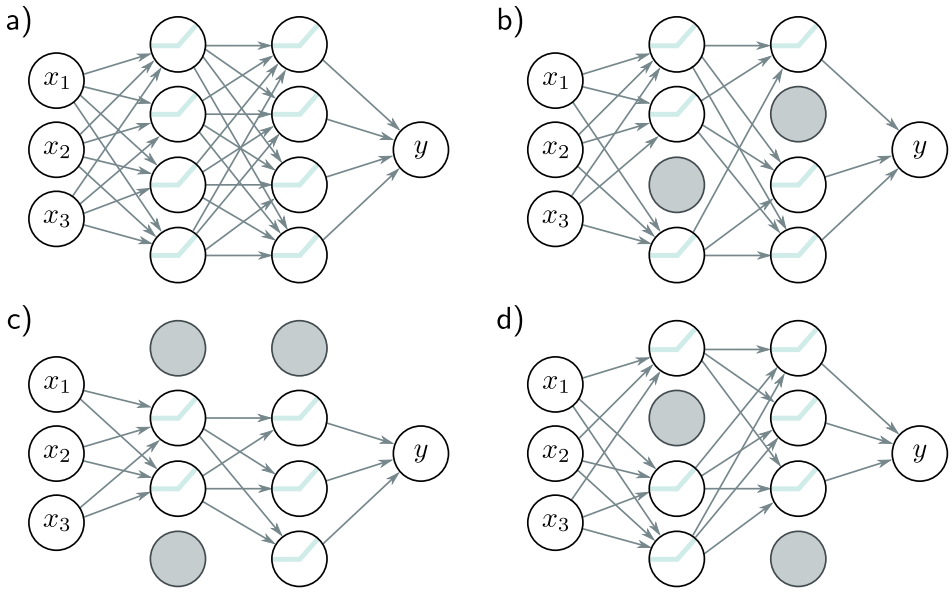

Figure 9.8 Dropout. a) Original network. b–d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes). The result is that the incoming and outgoing weights from these units have no effect, so we are training with a slightly different network each time.

unit decreases the slope, so the function goes back down. Finally, the third unit cancels out this decrease and returns the curve to its original trajectory. These three units conspire to make an undesirable local change in the function. This will not change the training loss but is unlikely to generalize well.

When several units conspire in this way, eliminating one (as would happen in dropout) causes a considerable change to the output function in the half-space where that unit was active (figure 9.9b). A subsequent gradient descent step will attempt to compensate for the change that this induces, and such dependencies will be eliminated over time. The overall effect is that large unnecessary changes between training data points are gradually removed even though they contribute nothing to the loss (figure 9.9).

At test time, we can run the network as usual with all the hidden units active; however, the network now has more hidden units than it was trained with at any given iteration, so we multiply the weights by one minus the dropout probability to compensate. This is known as the *weight scaling inference rule*. A different approach to inference is to use *Monte Carlo dropout*, in which we run the network multiple times with different random subsets of units clamped to zero (as in training) and combine the results. This is closely related to ensembling in that every random version of the network is a different model; however, we do not have to train or store multiple networks here.

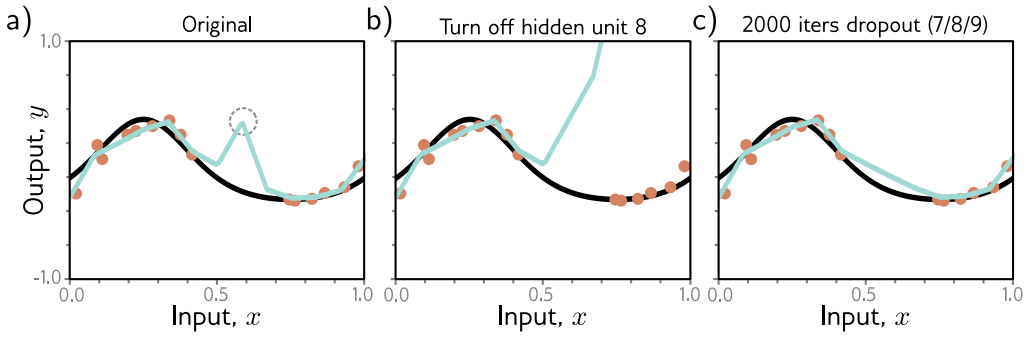

Figure 9.9 Dropout mechanism. a) An undesirable kink in the curve is caused by a sequential increase in the slope, decrease in the slope (at circled joint), and then another increase to return the curve to its original trajectory. Here we are using full-batch gradient descent, and the model (from figure 8.4) fits the data as well as possible, so further training won't remove the kink. b) Consider what happens if we remove the eighth hidden unit that produced the circled joint in panel (a), as might happen using dropout. Without the decrease in the slope, the right-hand side of the function takes an upwards trajectory, and a subsequent gradient descent step will aim to compensate for this change. c) Curve after 2000 iterations of (i) randomly removing one of the three hidden units that cause the kink and (ii) performing a gradient descent step. The kink does not affect the loss but is nonetheless removed by this approximation of the dropout mechanism.

9.3.4 Applying noise

Dropout can be interpreted as applying multiplicative Bernoulli noise to the network activations. This leads to the idea of applying noise to other parts of the network during training to make the final model more robust.

One option is to add noise to the input data; this smooths out the learned function (figure 9.10). For regression problems, it can be shown to be equivalent to adding a regularizing term that penalizes the derivatives of the network's output with respect to its input. An extreme variant is *adversarial training*, in which the optimization algorithm actively searches for small perturbations of the input that cause large changes to the output. These can be thought of as worst-case additive noise vectors.

Problem 9.3

A second possibility is to add noise to the weights. This encourages the network to make sensible predictions even for small perturbations of the weights. The result is that the training converges to local minima in the middle of wide, flat regions, where changing the individual weights does not matter much.

Finally, we can perturb the labels. The maximum-likelihood criterion for multiclass classification aims to predict the correct class with absolute certainty (equation 5.24). To this end, the final network activations (i.e., before the softmax function) are pushed to very large values for the correct class and very small values for the wrong classes.

We could discourage this overconfident behavior by assuming that a proportion ρ of

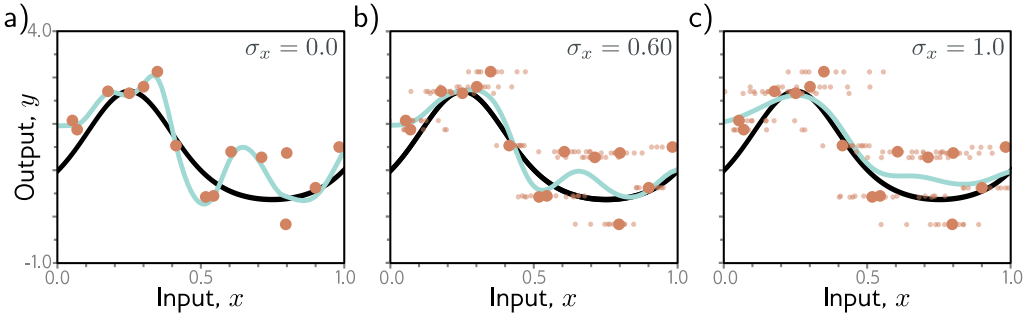

Figure 9.10 Adding noise to inputs. At each step of SGD, random noise with variance σ_x^2 is added to the batch data. a–c) Fitted model with different noise levels (small dots represent ten samples). Adding more noise smooths out the fitted function (cyan line).

the training labels are incorrect and belong with equal probability to the other classes. This could be done by randomly changing the labels at each training iteration. However, the same end can be achieved by changing the loss function to minimize the cross-entropy between the predicted distribution and a distribution where the true label has probability $1-\rho$, and the other classes have equal probability. This is known as *label smoothing* and improves generalization in diverse scenarios.

Problem 9.4

9.3.5 Bayesian inference

The maximum likelihood approach is generally overconfident; it selects the most likely parameters during training and uses these to make predictions. However, many parameter values may be broadly compatible with the data and only slightly less likely. The Bayesian approach treats the parameters as unknown variables and computes a distribution $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ over these parameters $\boldsymbol{\phi}$ conditioned on the training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using Bayes' rule:

Appendix C.1.4 Bayes' rule

$$Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) = \frac{\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi})}{\int \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}) d\boldsymbol{\phi}}, \tag{9.11}$$

where $Pr(\boldsymbol{\phi})$ is the prior probability of the parameters, and the denominator is a normalizing term. Hence, every parameter choice is assigned a probability (figure 9.11).

The prediction $\mathbf{y}$ for new input $\mathbf{x}$ is an infinite weighted sum (i.e., an integral) of the predictions for each parameter set, where the weights are the associated probabilities:

$$Pr(\mathbf{y}|\mathbf{x}, \{\mathbf{x}_i, \mathbf{y}_i\}) = \int Pr(\mathbf{y}|\mathbf{x}, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) d\boldsymbol{\phi}. \tag{9.12}$$

This is effectively an infinite weighted ensemble, where the weight depends on (i) the prior probability of the parameters and (ii) their agreement with the data.

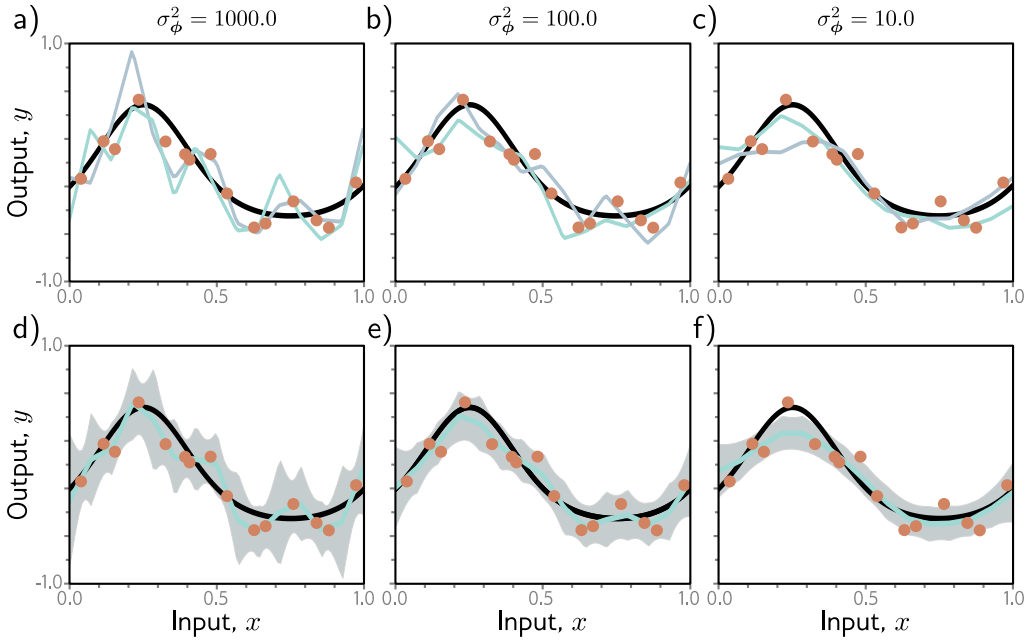

Figure 9.11 Bayesian approach for simplified network model (see figure 8.4). The parameters are treated as uncertain. The posterior probability $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ for a set of parameters is determined by their compatibility with the data $\{\mathbf{x}_i, \mathbf{y}_i\}$ and a prior distribution $Pr(\boldsymbol{\phi})$. a–c) Two sets of parameters (cyan and gray curves) sampled from the posterior using normally distributed priors with mean zero and three variances. When the prior variance σ_ϕ^2 is small, the parameters also tend to be small, and the functions smoother. d–f) Inference proceeds by taking a weighted sum over all possible parameter values where the weights are the posterior probabilities. This produces both a prediction of the mean (cyan curves) and the associated uncertainty (gray region is two standard deviations).

The Bayesian approach is elegant and can provide more robust predictions than those that derive from maximum likelihood. Unfortunately, for complex models like neural networks, there is no practical way to represent the full probability distribution over the parameters or to integrate over it during the inference phase. Consequently, all current methods of this type make approximations of some kind, and typically these add considerable complexity to learning and inference.

Notebook 9.4 Bayesian approach

9.3.6 Transfer learning and multi-task learning

When training data are limited, other datasets can be exploited to improve performance. In *transfer learning* (figure 9.12a), the network is *pre-trained* to perform a related sec-

ondary task for which data are more plentiful. The resulting model is then adapted to the original task. This is typically done by removing the last layer and adding one or more layers that produce a suitable output. The main model may be fixed, and the new layers trained for the original task, or we may *fine-tune* the entire model.

The principle is that the network will build a good internal representation of the data from the secondary task, which can subsequently be exploited for the original task. Equivalently, transfer learning can be viewed as initializing most of the parameters of the final network in a sensible part of the space that is likely to produce a good solution.

Multi-task learning (figure 9.12b) is a related technique in which the network is trained to solve several problems concurrently. For example, the network might take an image and simultaneously learn to segment the scene, estimate the pixel-wise depth, and predict a caption describing the image. All of these tasks require some understanding of the image and, when learned simultaneously, the model performance for each may improve.

9.3.7 Self-supervised learning

The above discussion assumes that we have plentiful data for a secondary task or data for multiple tasks to be learned concurrently. If not, we can create large amounts of "free" labeled data using *self-supervised* learning and use this for transfer learning. There are two families of methods for self-supervised learning: *generative* and *contrastive*.

In *generative self-supervised learning*, part of each data example is masked, and the secondary task is to predict the missing part (figure 9.12c). For example, we might use a corpus of unlabeled images and a secondary task that aims to *inpaint* (fill in) missing parts of the image (figure 9.12c). Similarly, we might use a large corpus of text and mask some words. We train the network to predict the missing words and then fine-tune it for the actual language task we are interested in (see chapter 12).

In *contrastive self-supervised learning*, pairs of examples with commonalities are compared to unrelated pairs. For images, the secondary task might be to identify whether a pair of images are transformed versions of one another or are unconnected. For text, the secondary task might be to determine whether two sentences followed one another in the original document. Sometimes, the precise relationship between a connected pair must be identified (e.g., finding the relative position of two patches from the same image).

9.3.8 Augmentation

Transfer learning improves performance by exploiting a different dataset. Multi-task learning improves performance using additional labels. A third option is to expand the dataset. We can often transform each input data example in such a way that the label stays the same. For example, we might aim to determine if there is a bird in an image (figure 9.13). Here, we could rotate, flip, blur, or manipulate the color balance of the image, and the label "bird" remains valid. Similarly, for tasks where the input is text, we can substitute synonyms or translate to another language and back again. For tasks where the input is audio, we can amplify or attenuate different frequency bands.

Notebook 9.5
Augmentation

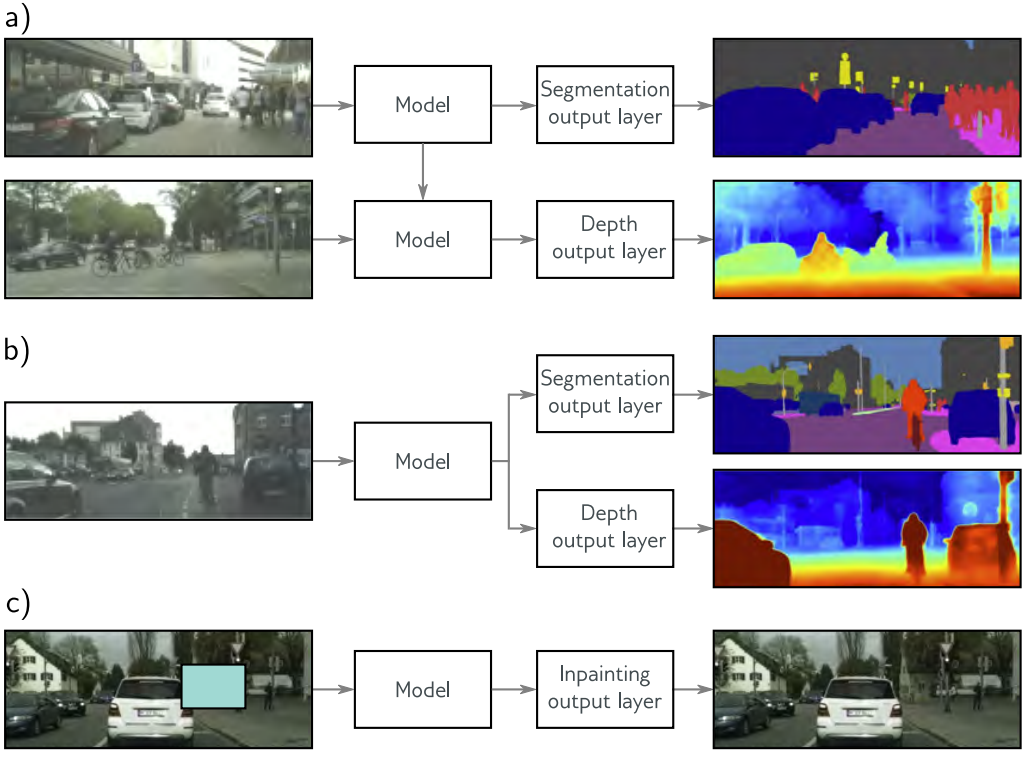

Figure 9.12 Transfer, multi-task, and self-supervised learning. a) Transfer learning is used when we have limited labeled data for the primary task (here depth estimation) but plentiful data for a secondary task (here segmentation). We train a model for the secondary task, remove the final layers, and replace them with new layers appropriate to the primary task. We then train only the new layers or fine-tune the entire network for the primary task. The network learns a good internal representation from the secondary task that is then exploited for the primary task. b) In multi-task learning, we train a model to perform multiple tasks simultaneously, hoping that performance on each will improve. c) In generative self-supervised learning, we remove part of the data and train the network to complete the missing information. Here, the task is to fill in (inpaint) a masked portion of the image. This permits transfer learning when no labels are available. Images from Cordts et al. (2016).

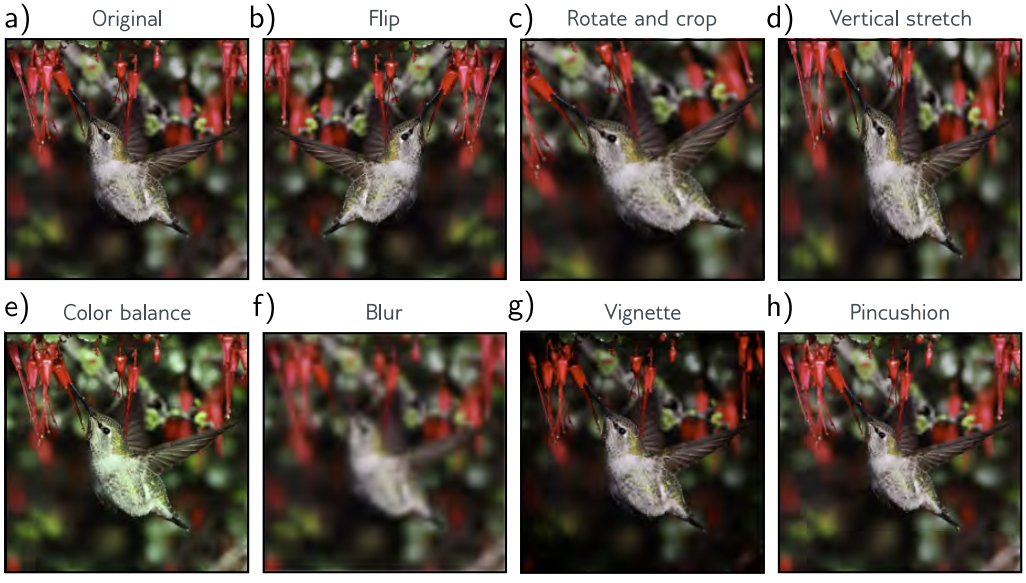

Figure 9.13 Data augmentation. For some problems, each data example can be transformed to augment the dataset. a) Original image. b–h) Various geometric and photometric transformations of this image. For image classification, all these images still have the same label, "bird." Adapted from Wu et al. (2015a).

Generating extra training data in this way is known as *data augmentation*. The aim is to teach the model to be indifferent to these irrelevant data transformations.

9.4 Summary

Explicit regularization involves adding an extra term to the loss function that changes the position of the minimum. The term can be interpreted as a prior probability over the parameters. Stochastic gradient descent with a finite step size does not neutrally descend to the minimum of the loss function. This bias can be interpreted as adding additional terms to the loss function, and this is known as implicit regularization.

There are also many heuristics for improving generalization, including early stopping, dropout, ensembling, the Bayesian approach, adding noise, transfer learning, multi-task learning, and data augmentation. There are four main principles behind these methods (figure 9.14). We can (i) encourage the function to be smoother (e.g., L2 regularization), (ii) increase the amount of data (e.g., data augmentation), (iii) combine models (e.g., ensembling), or (iv) search for wider minima (e.g., applying noise to network weights).

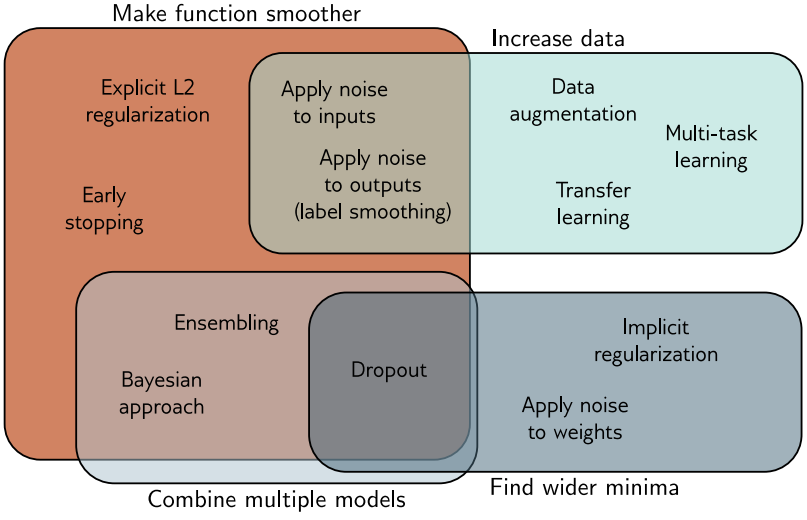

Figure 9.14 Regularization methods. The regularization methods discussed in this chapter aim to improve generalization by one of four mechanisms. Some methods aim to make the modeled function smoother. Other methods increase the effective amount of data. The third group of methods combine multiple models and hence mitigate against uncertainty in the fitting process. Finally, the fourth group of methods encourages the training process to converge to a wide minimum where small errors in the estimated parameters are less important (see also figure 20.11).

Another way to improve generalization is to choose the model architecture to suit the task. For example, in image segmentation, we can share parameters within the model, so we don't need to independently learn what a tree looks like at every image location. Chapters 10–13 consider architectural variations designed for different tasks.

Notes

An overview and taxonomy of regularization techniques in deep learning can be found in Kukačka et al. (2017). Notably missing from the discussion in this chapter is BatchNorm (Szegedy et al., 2016) and its variants, which are described in chapter 11.

Regularization: L2 regularization penalizes the sum of squares of the network weights. This encourages the output function to change slowly (i.e., become smoother) and is the most used regularization term. It is sometimes referred to as Frobenius norm regularization as it penalizes the Frobenius norms of the weight matrices. It is often also mistakenly referred to as "weight decay," although this is a separate technique devised by Hanson & Pratt (1988) in which the parameters $\boldsymbol{\phi}$ are updated as:

$$\boldsymbol{\phi} \longleftarrow (1-\lambda')\boldsymbol{\phi} - \alpha\frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.13}$$

where, as usual, α is the learning rate, and L is the loss. This is identical to gradient descent, except that the weights are reduced by a factor of $1-\lambda'$ before the gradient update. For standard SGD, weight decay is equivalent to L2 regularization (equation 9.5) with coefficient $\lambda = \lambda'/2\alpha$. However, for Adam, the learning rate α is different for each parameter, so L2 regularization and weight decay differ. Loshchilov & Hutter (2019) present AdamW, which modifies Adam to implement weight decay correctly and show that this improves performance.

Problem 9.5

Other choices of vector norm encourage sparsity in the weights. The L0 regularization term applies a fixed penalty for every non-zero weight. The effect is to "prune" the network. L0 regularization can also be used to encourage group sparsity; this might apply a fixed penalty if any of the weights contributing to a given hidden unit are non-zero. If they are all zero, we can remove the unit, decreasing the model size and making inference faster.

Appendix B.3.2 Vector norms

Unfortunately, L0 regularization is challenging to implement since the derivative of the regularization term is not smooth, and more sophisticated fitting methods are required (see Louizos et al., 2018). Somewhere between L2 and L0 regularization is L1 regularization or *LASSO* (least absolute shrinkage and selection operator), which imposes a penalty on the absolute values of the weights. L2 regularization somewhat discourages sparsity in that the derivative of the squared penalty decreases as the weight becomes smaller, lowering the pressure to make it smaller still. L1 regularization does not have this disadvantage, as the derivative of the penalty is constant. This can produce sparser solutions than L2 regularization but is much easier to optimize than L0 regularization. Sometimes both L1 and L2 regularization terms are used, which is termed an *elastic net* penalty (Zou & Hastie, 2005).

Problem 9.6

A different approach to regularization is to modify the gradients of the learning algorithm without ever explicitly formulating a new loss function (e.g., equation 9.13). This approach has been used to promote sparsity during backpropagation (Schwarz et al., 2021).

The evidence on the effectiveness of explicit regularization is mixed. Zhang et al. (2017a) showed that L2 regularization contributes little to generalization. It has been proven that the Lipschitz constant of the network (how fast the function can change as we modify the input) bounds the generalization error (Bartlett et al., 2017; Neyshabur et al., 2018). However, the Lipschitz constant depends on the product of the spectral norms of the weight matrices $\boldsymbol{\Omega}_k$, which are only indirectly dependent on the magnitudes of the individual weights. Bartlett et al. (2017), Neyshabur et al. (2018), and Yoshida & Miyato (2017) all add terms that indirectly encourage the spectral norms to be smaller. Gouk et al. (2021) take a different approach and develop an algorithm that constrains the Lipschitz constant of the network to be below a particular value.

Appendix B.1.1 Lipschitz constant

Appendix B.3.7 Spectral norm

Implicit regularization in gradient descent: The gradient descent step is:

$$\boldsymbol{\phi}_1 = \boldsymbol{\phi}_0 + \alpha \cdot \mathbf{g}[\boldsymbol{\phi}_0], \tag{9.14}$$

where $\mathbf{g}[\boldsymbol{\phi}_0]$ is the negative of the gradient of the loss function, and α is the step size. As $\alpha \to 0$, the gradient descent process can be described by a differential equation:

$$\frac{d\boldsymbol{\phi}}{dt} = \mathbf{g}[\boldsymbol{\phi}]. \tag{9.15}$$

For typical step sizes α, the discrete and continuous versions converge to different solutions. We can use *backward error analysis* to find a correction $\mathbf{g}_1[\boldsymbol{\phi}]$ to the continuous version:

$$\frac{d\boldsymbol{\phi}}{dt} \approx \mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}] + \ldots, \tag{9.16}$$

so that it gives the same result as the discrete version.

Consider the first two terms of a Taylor expansion of the modified continuous solution $\boldsymbol{\phi}$ around initial position $\boldsymbol{\phi}_0$:

$$\begin{aligned}
\boldsymbol{\phi}[\alpha] &\approx \boldsymbol{\phi} + \alpha\frac{d\boldsymbol{\phi}}{dt} + \frac{\alpha^2}{2}\frac{d^2\boldsymbol{\phi}}{dt^2}\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha\left(\mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2}\left(\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\frac{d\boldsymbol{\phi}}{dt} + \alpha\frac{\partial\mathbf{g}_1[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\frac{d\boldsymbol{\phi}}{dt}\right)\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&= \boldsymbol{\phi} + \alpha\left(\mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2}\left(\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}] + \alpha\frac{\partial\mathbf{g}_1[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]\right)\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha\mathbf{g}[\boldsymbol{\phi}] + \alpha^2\left(\mathbf{g}_1[\boldsymbol{\phi}] + \frac{1}{2}\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]\right)\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0}, \qquad (9.17)
\end{aligned}$$

where in the second line, we have introduced the correction term (equation 9.16), and in the final line, we have removed terms of greater order than α^2.

Note that the first two terms on the right-hand side $\boldsymbol{\phi}_0 + \alpha\mathbf{g}[\boldsymbol{\phi}_0]$ are the same as the discrete update (equation 9.14). Hence, to make the continuous and discrete versions arrive at the same place, the third term on the right-hand side must equal zero, allowing us to solve for $\mathbf{g}_1[\boldsymbol{\phi}]$:

$$\mathbf{g}_1[\boldsymbol{\phi}] = -\frac{1}{2}\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]. \qquad (9.18)$$

During training, the evolution function $\mathbf{g}[\boldsymbol{\phi}]$ is the negative of the gradient of the loss:

$$\begin{aligned}
\frac{d\boldsymbol{\phi}}{dt} &\approx \mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}] \\
&= -\frac{\partial L}{\partial\boldsymbol{\phi}} - \frac{\alpha}{2}\left(\frac{\partial^2 L}{\partial\boldsymbol{\phi}^2}\right)\frac{\partial L}{\partial\boldsymbol{\phi}}. \qquad (9.19)
\end{aligned}$$

This is equivalent to performing continuous gradient descent on the loss function:

$$L_{GD}[\boldsymbol{\phi}] = L[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial\boldsymbol{\phi}}\right\|^2, \qquad (9.20)$$

because the right-hand side of equation 9.19 is the derivative of that in equation 9.20.

This formulation of implicit regularization was developed by Barrett & Dherin (2021) and extended to stochastic gradient descent by Smith et al. (2021). Smith et al. (2020) and others have shown that stochastic gradient descent with small or moderate batch sizes outperforms full batch gradient descent on the test set, and this may in part be due to implicit regularization.

Relatedly, Jastrzębski et al. (2021) and Cohen et al. (2021) both show that using a large learning rate reduces the tendency of typical optimization trajectories to move to "sharper" parts of the loss function (i.e., where at least one direction has high curvature). This implicit regularization effect of large learning rates can be approximated by penalizing the trace of the Fisher Information Matrix, which is closely related to penalizing the gradient norm in equation 9.20 (Jastrzębski et al., 2021).

Early stopping: Bishop (1995) and Sjöberg & Ljung (1995) argued that early stopping limits the effective solution space that the training procedure can explore; given that the weights are initialized to small values, this leads to the idea that early stopping helps prevent the weights from getting too large. Goodfellow et al. (2016) show that under a quadratic approximation of the loss function with parameters initialized to zero, early stopping is equivalent to L2 regularization in gradient descent. The effective regularization weight λ is approximately $1/(\tau\alpha)$ where α is the learning rate, and τ is the early stopping time.

Ensembling: Ensembles can be trained using different random seeds (Lakshminarayanan et al., 2017), hyperparameters (Wenzel et al., 2020b), or even entirely different families of models. The models can be combined by averaging their predictions, weighting the predictions, or *stacking* (Wolpert, 1992), in which the results are combined using another machine learning model. Lakshminarayanan et al. (2017) showed that averaging the output of independently trained networks can improve accuracy, calibration, and robustness. Conversely, Frankle et al. (2020) showed that if we average together the weights to make one model, the network fails. Fort et al. (2019) compared ensembling solutions that resulted from different initializations with ensembling solutions that were generated from the same original model. For example, in the latter case, they consider exploring around the solution in a limited subspace to find other good nearby points. They found that both techniques provide complementary benefits but that genuine ensembling from different random starting points provides a bigger improvement.

Appendix B.3.6 Subspaces

An efficient way of ensembling is to combine models from intermediate stages of training. To this end, Izmailov et al. (2018) introduce *stochastic weight averaging*, in which the model weights are sampled at different time steps and averaged together. As the name suggests, *snapshot ensembles* (Huang et al., 2017a) also store the models from different time steps and average their predictions. The diversity of these models can be improved by cyclically increasing and decreasing the learning rate. Garipov et al. (2018) observed that different minima of the loss function are often connected by a low-energy path (i.e., a path with a low loss everywhere along it). Motivated by this observation, they developed a method that explores low-energy regions around an initial solution to provide diverse models without retraining. This is known as *fast geometric ensembling*. A review of ensembling methods can be found in Ganaie et al. (2022).

Dropout: Dropout was first introduced by Hinton et al. (2012b) and Srivastava et al. (2014). Dropout is applied at the level of hidden units. Dropping a hidden unit has the same effect as temporarily setting all the incoming and outgoing weights and the bias to zero. Wan et al. (2013) generalized dropout by randomly setting individual weights to zero. Gal & Ghahramani (2016) and Kendall & Gal (2017) proposed Monte Carlo dropout, in which inference is computed with several dropout patterns, and the results are averaged together. Gal & Ghahramani (2016) argued that this could be interpreted as approximating Bayesian inference.

Dropout is equivalent to applying multiplicative Bernoulli noise to the hidden units. Similar benefits derive from using other distributions, including the normal (Srivastava et al., 2014; Shen et al., 2017), uniform (Shen et al., 2017), and beta distributions (Liu et al., 2019b).

Adding noise: Bishop (1995) and An (1996) added Gaussian noise to the network inputs to improve performance. Bishop (1995) showed that this is equivalent to weight decay. An (1996) also investigated adding noise to the weights. DeVries & Taylor (2017a) added Gaussian noise to the hidden units. The *randomized ReLU* (Xu et al., 2015) applies noise in a different way by making the activation functions stochastic.

Label smoothing: Label smoothing was introduced by Szegedy et al. (2016) for image classification but has since been shown to be helpful in speech recognition (Chorowski & Jaitly, 2017), machine translation (Vaswani et al., 2017), and language modeling (Pereyra et al., 2017). The precise mechanism by which label smoothing improves test performance isn't well understood, although Müller et al. (2019a) show that it improves the calibration of the predicted output probabilities. A closely related technique is *DisturbLabel* (Xie et al., 2016), in which a certain percentage of the labels in each batch are randomly switched at each training iteration.

Finding wider minima: It is thought that wider minima generalize better (see figure 20.11). Here, the exact values of the weights are less important, so performance should be robust to errors in their estimates. One of the reasons that applying noise to parts of the network during training is effective is that it encourages the network to be indifferent to their exact values.

Chaudhari et al. (2019) developed a variant of SGD that biases the optimization toward flat minima, which they call *entropy SGD*. The idea is to incorporate local entropy as a term in the loss function. In practice, this takes the form of one SGD-like update within another. Keskar

et al. (2017) showed that SGD finds wider minima as the batch size is reduced. This may be because of the batch variance term that results from implicit regularization by SGD.

Ishida et al. (2020) use a technique named *flooding*, in which they intentionally prevent the training loss from becoming zero. This encourages the solution to perform a random walk over the loss landscape and drift into a flatter area with better generalization.

Bayesian approaches: For some models, including the simplified neural network model in figure 9.11, the Bayesian predictive distribution can be computed in closed form (see Bishop, 2006; Prince, 2012). For neural networks, the posterior distribution over the parameters cannot be represented in closed form and must be approximated. The two main approaches are variational Bayes (Hinton & van Camp, 1993; MacKay, 1995; Barber & Bishop, 1997; Blundell et al., 2015), in which the posterior is approximated by a simpler tractable distribution, and Markov Chain Monte Carlo (MCMC) methods, which approximate the distribution by drawing a set of samples (Neal, 1995; Welling & Teh, 2011; Chen et al., 2014; Ma et al., 2015; Li et al., 2016a). The generation of samples can be integrated into SGD, and this is known as stochastic gradient MCMC (see Ma et al., 2015). It has recently been discovered that "cooling" the posterior distribution over the parameters (making it sharper) improves predictions from these models (Wenzel et al., 2020a), but this is not currently fully understood (see Noci et al., 2021).

Transfer learning: Transfer learning for visual tasks works extremely well (Sharif Razavian et al., 2014) and has supported rapid progress in computer vision, including the original AlexNet results (Krizhevsky et al., 2012). Transfer learning has also impacted natural language processing (NLP), where many models are based on pre-trained features from the BERT model (Devlin et al., 2019). More information can be found in Zhuang et al. (2020) and Yang et al. (2020b).

Self-supervised learning: Self-supervised learning techniques for images have included inpainting masked image regions (Pathak et al., 2016), predicting the relative position of patches in an image (Doersch et al., 2015), re-arranging permuted image tiles back into their original configuration (Noroozi & Favaro, 2016), colorizing grayscale images (Zhang et al., 2016b), and transforming rotated images back to their original orientation (Gidaris et al., 2018). In SimCLR (Chen et al., 2020c), a network is learned that maps versions of the same image that have been photometrically and geometrically transformed to the same representation while repelling versions of different images, with the goal of becoming indifferent to irrelevant image transformations. Jing & Tian (2020) present a survey of self-supervised learning in images.

Self-supervised learning in NLP can be based on predicting masked words(Devlin et al., 2019), predicting the next word in a sentence (Radford et al., 2019; Brown et al., 2020), or predicting whether two sentences follow one another (Devlin et al., 2019). In automatic speech recognition, the Wav2Vec model (Schneider et al., 2019) aims to distinguish an original audio sample from one where 10ms of audio has been swapped out from elsewhere in the clip. Self-supervision has also been applied to graph neural networks (chapter 13). Tasks include recovering masked features (You et al., 2020) and recovering the adjacency structure of the graph (Kipf & Welling, 2016). Liu et al. (2023a) review self-supervised learning for graph models.

Data augmentation: Data augmentation for images dates back to at least LeCun et al. (1998) and contributed to the success of AlexNet (Krizhevsky et al., 2012), in which the dataset was increased by a factor of 2048. Image augmentation approaches include geometric transformations, changing or manipulating the color space, noise injection, and applying spatial filters. More elaborate techniques include randomly mixing images (Inoue, 2018; Summers & Dinneen, 2019), randomly erasing parts of the image (Zhong et al., 2020), style transfer (Jackson et al., 2019), and randomly swapping image patches (Kang et al., 2017). In addition, many studies have used generative adversarial networks or GANs (see chapter 15) to produce novel but plausible data examples (e.g., Calimeri et al., 2017). In other cases, the data have been augmented with adversarial examples (Goodfellow et al., 2015a), which are minor perturbations of the training data that cause the example to be misclassified. A review of data augmentation for images can be found in Shorten & Khoshgoftaar (2019).

Augmentation methods for acoustic data include pitch shifting, time stretching, dynamic range compression, and adding random noise (e.g., Abeßer et al., 2017; Salamon & Bello, 2017; Xu et al., 2015; Lasseck, 2018), as well as mixing data pairs (Zhang et al., 2017c; Yun et al., 2019), masking features (Park et al., 2019), and using GANs to generate new data (Mun et al., 2017). Augmentation for speech data includes vocal tract length perturbation (Jaitly & Hinton, 2013; Kanda et al., 2013), style transfer (Gales, 1998; Ye & Young, 2004), adding noise (Hannun et al., 2014), and synthesizing speech (Gales et al., 2009).

Augmentation methods for text include adding noise at a character level by switching, deleting, and inserting letters (Belinkov & Bisk, 2018; Feng et al., 2020), or by generating adversarial examples (Ebrahimi et al., 2018), using common spelling mistakes (Coulombe, 2018), randomly swapping or deleting words (Wei & Zou, 2019), using synonyms (Kolomiyets et al., 2011), altering adjectives (Li et al., 2017c), passivization (Min et al., 2020), using generative models to create new data (Qiu et al., 2020), and round-trip translation to another language and back (Aiken & Park, 2010). Augmentation methods for text are reviewed by Bayer et al. (2022).

Problems

Problem 9.1 Consider a model where the prior distribution over the parameters is a normal distribution with mean zero and variance $\sigma_{\boldsymbol{\phi}}^2$ so that

$$Pr(\boldsymbol{\phi}) = \prod_{j=1}^{J} \text{Norm}_{\phi_j}[0, \sigma_{\boldsymbol{\phi}}^2], \tag{9.21}$$

where j indexes the model parameters. We now maximize $\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi})Pr(\boldsymbol{\phi})$. Show that the associated loss function of this model is equivalent to L2 regularization.

Problem 9.2 How do the gradients of the loss function change when L2 regularization (equation 9.5) is added?

Problem 9.3* Consider a linear regression model $y = \phi_0 + \phi_1 x$ with input x, output y, and parameters ϕ_0 and ϕ_1. Assume we have I training examples $\{x_i, y_i\}$ and use a least squares loss. Consider adding Gaussian noise with mean zero and variance σ_x^2 to the inputs x_i at each training iteration. What is the expected gradient update?

Problem 9.4* Derive the loss function for multiclass classification when we use label smoothing so that the target probability distribution has 0.9 at the correct class and the remaining probability mass of 0.1 is divided between the remaining $D_o - 1$ classes.

Problem 9.5 Show that the weight decay parameter update with decay rate λ:

$$\boldsymbol{\phi} \longleftarrow (1-\lambda)\boldsymbol{\phi} - \alpha\frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.22}$$

on the original loss function $L[\boldsymbol{\phi}]$ is equivalent to a standard gradient update using L2 regularization so that the modified loss function $\tilde{L}[\boldsymbol{\phi}]$ is:

$$\tilde{L}[\boldsymbol{\phi}] = L[\boldsymbol{\phi}] + \frac{\lambda}{2\alpha}\sum_{k}\phi_k^2, \tag{9.23}$$

where $\boldsymbol{\phi}$ are the parameters, and α is the learning rate.

Problem 9.6 Consider a model with parameters $\boldsymbol{\phi} = [\boldsymbol{\phi}_0, \boldsymbol{\phi}_1]^T$. Draw the L0, $L\frac{1}{2}$, and L1 regularization terms in a similar form to figure 9.1b. The LP regularization term is $\sum_{d=1}^{D}|\phi_d|^P$.