Chapter 11

Regularization

Building machine learning models is all about finding a balance between simplicity and complexity. We want models that are powerful enough to fit the complex trends in our data, but we also do not want these models to have excessive variance and overfit. In an ideal world, we could always collect larger and larger training sets, which would allow us to combat overfitting and reliably optimize complex models. In practice, however, we are often stuck with fixed-size training sets and must reduce overfitting by simplifying our models.

So far, when discussing model complexity, we have primarily discussed the idea of adding or removing features from a model. Adding more features generally makes a model more complex. Similarly, removing features is a good way to simplify a model that is overfitting. However, this approach is a relatively blunt instrument. Ideally, we would like an approach that can tune the complexity of our models in a more fine-grained way, without throwing away potentially useful feature information. This is the motivation behind the method of *regularization*.

11.1 Penalizing Complexity

Regularizing a machine learning model is equivalent to penalizing its complexity. As a consequence, the most straightforward way to regularize a model is to add a penalty term to the optimization problem, which penalizes complex models. The idea is that we want to replace our standard loss term $L(y, f(\mathbf{x}))$ with a penalized version

$$L_{\text{reg}}(y, f(\mathbf{x})) = L(y, f(\mathbf{x})) + \lambda\Omega(f), \tag{11.1}$$

where Ω is some measure of the model's complexity and $\lambda \in \mathbb{R}$ is a hyperparameter controlling the strength of the regularization. Thus, rather than simply penalizing models for making incorrect predictions, we also want to penalize models if they are overly complex, with the motivation that simpler models will tend to generalize better.

The primary challenge in implementing Equation 11.1 is how we quantify model complexity. In Chapter 10 we mentioned the idea of quantifying a model's

complexity by measuring the size of its model class. In practice, we cannot use this idea to penalize model complexity—however—because (a) we want a measure of complexity specific to a particular model and not a whole model class and (b) most model classes in the real world are infinite.

11.1.1 Weight decay and L2 regularization

The most popular and ubiquitous approach to penalizing model complexity goes by many names. Its often known as L2 regularization, Tikhonov regularization, or weight decay. The idea is quite simple. We penalize the Euclidean norm (also known as the L2 norm) of the model's parameter vector:

$$L_{\text{reg}}(y, f_{\mathbf{w}}(\mathbf{x})) = L(y, f(\mathbf{x})) + \lambda \|\mathbf{w}\|^2. \tag{11.2}$$

By penalizing the norm of the parameter vector, we force the model to find solutions that involve smaller parameter coefficients. This effectively reduces the space of possible models. Moreover, reducing the magnitude of the parameter coefficients intuitively reduces the variance of our model, since we are effectively reducing the dynamic range of our prediction values.

L2 regularization is the go-to approach for any machine learning model that uses continuous parameters. (Note that even if our model has parameter matrices or tensors—rather than vectors—we can simply flatten them to vectors when computing the regularization term.) It is also convex and easily differentiable, which makes it naturally suitable for any application involving gradient descent. In practice, L2 regularization is included in *most* machine learning models by default.

11.1.2 Enforcing sparsity and L1 regularization

L2 regularization penalizes models based on the Euclidean norm of the parameter vector, which is also known as the L2 norm. Could other norms also be useful as penalty terms? In fact, the second most popular regularization approach—commonly known as L1 regularization or lasso regularization—takes exactly this approach. Instead of using the Euclidean or L2 norm, we use the L2 norm, which generalizes the absolute value to vectors:

$$|\mathbf{x}| = \sum_{j=1}^{m} |\mathbf{x}[j]|. \tag{11.3}$$

Using this norm, we can regularize our model in a manner analogous to L2 regularization

$$L_{\text{reg}}(y, f_{\mathbf{w}}(\mathbf{x})) = L(y, f(\mathbf{x})) + \lambda |\mathbf{w}|. \tag{11.4}$$

Like L2 regularization, L1 regularization penalizes the magnitude of the model parameters. However, the behavior of L1 regularization is quite different. In practice, L1 regularization tends to lead to *sparse* solutions, where many of the parameter weights are set to zero. The geometric intuition for this

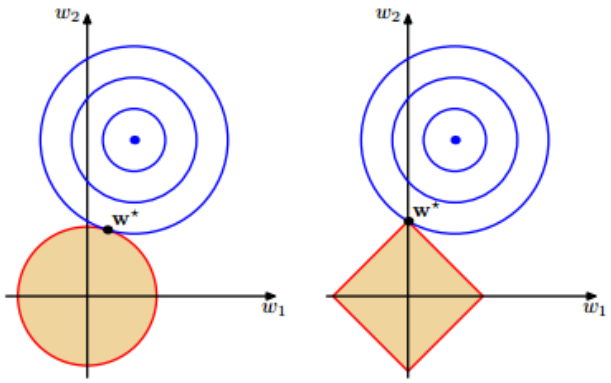

Figure 11.1: Illustration of L2 regularization (left), compared to L1 regularization (right) in a two dimensional parameter space. The red circles show regions of space with constrained magnitudes, according to either the L1 or L2 norm. The blue contours correspond to the loss function.[1]

phenomenon in two-dimensions is shown in Figure 11.1. We can think of L2 regularization as constraining model parameters to lie within a sphere of bounded radius, since we are penalizing the Euclidean norm. In other words, if we consider all weight vectors $\mathbf{w}$ that have equal norm $\|\mathbf{w}\|$ then we will form a circle or spherical shape. On the other hand, L1 regularization constrains the model parameters to lie within a diamond-shaped region, as shown in Figure 11.1. As a result, the set of constrained L1-regularized parameters tends to intersect the contours of the loss function along the axes of the parameter space (i.e., along points where some dimensions are equal to zero).

Optimizing L1 regularized models The attentive reader will have noticed that the L1 penalty violates a key property that we want in optimization: it is not smooth. This is a major drawback of L1 regularization, and there are two strategies to avoid it, and we cannot naively apply gradient descent to L1-regularized models. One strategy to get around this is to apply specially designed optimization algorithms, which do not need the smoothness requirement. An alternative approach relies on the notion of *subgradient descent*, which generalizes gradient descent to convex but not-necessarily-differentiable functions. The theory of subgradients is covered in more advanced machine learning courses. However, the consequence in the case of L1 regularization is that we can use

$$\frac{\partial |w|}{\partial w} = \text{sign}(w) \tag{11.5}$$

as a stand-in for the derivative of the L1 norm (i.e., the absolute value) when performing gradient descent.

11.1.3 Controlling the strength of regularization

Effectively employing either L1 or L2 regularization requires choosing the strength of the regularization, i.e., the λ value. This value is one of the most frequent and important *hyperparameters* in machine learning. There is no *a priori* way to know what the right value is, and we must use a validation set (or cross validation) to choose it. (Recall from Chapter 2 that validation sets data used to estimate model performance when choosing hyperparameters; we never train our models on a validation set!) It is essential to note that the best λ value might not give the highest training accuracy. We want to choose a λ that will give the best accuracy when generalizing to unseen data, and the only way to measure this is by using a validation set.

11.2 Early Stopping and Gradient Descent

L1 and L2 regularization are effective and ubiquitous approaches to reduce overfitting. When in doubt, L2 regularization is almost always the right starting point if you find that your model is overfitting. However, there is another strategy to reduce overfitting that is becoming increasingly popular in machine learning: *early stopping*.

The early stopping approach is only applicable in cases where gradient descent is being used, and it is generally only employed with stochastic minibatch gradient descent. The basic idea in early stopping is that we intentionally stop gradient descent before the model parameters have converged. The intuition is that models are generally increasing in complexity as gradient descent progresses. Thus, if we can stop the training early, we can prevent our model from overfitting to the training data.

An effective strategy for choosing when to stop gradient descent involves a held-out validation set. After every k iterations of gradient descent, we evaluate our model's performance on the validation set. If at any point we see the model's performance on the validation set has *decreased* compared to the last time we evaluated, then we know that we should stop training.