Stochastic Gradient Descent

The simplest gradient descent algorithm involves calculating $\nabla R(\boldsymbol{\theta})$ each time we take a step and just setting $\boldsymbol{\theta}_{(1)} = \boldsymbol{\theta}_{(0)} - \nabla R\left(\boldsymbol{\theta}_{(0)}\right)$.

This is a bad idea for at least two reasons:

1. If the gradient is steep, this may involve a big step, and we risk going too far. If so, $R(\boldsymbol{\theta})$ may actually increase.
2. When n is large, calculating the gradient can be expensive.

If we take steps that are too big, we can either converge very slowly or fail to converge.

This problem can be solved by using $\boldsymbol{\theta}_{(1)} = \boldsymbol{\theta}_{(0)} - \rho \nabla R\left(\boldsymbol{\theta}_{(0)}\right)$, where ρ is the **learning rate**, and it is small so that every step is a small one.

In principle, we might want to change ρ as the algorithm progresses.

We could also try several values of ρ for each gradient and pick the one that leads to the smallest value of $R(\boldsymbol{\theta})$.

Figure 20.1 Gradient descent in two dimenions

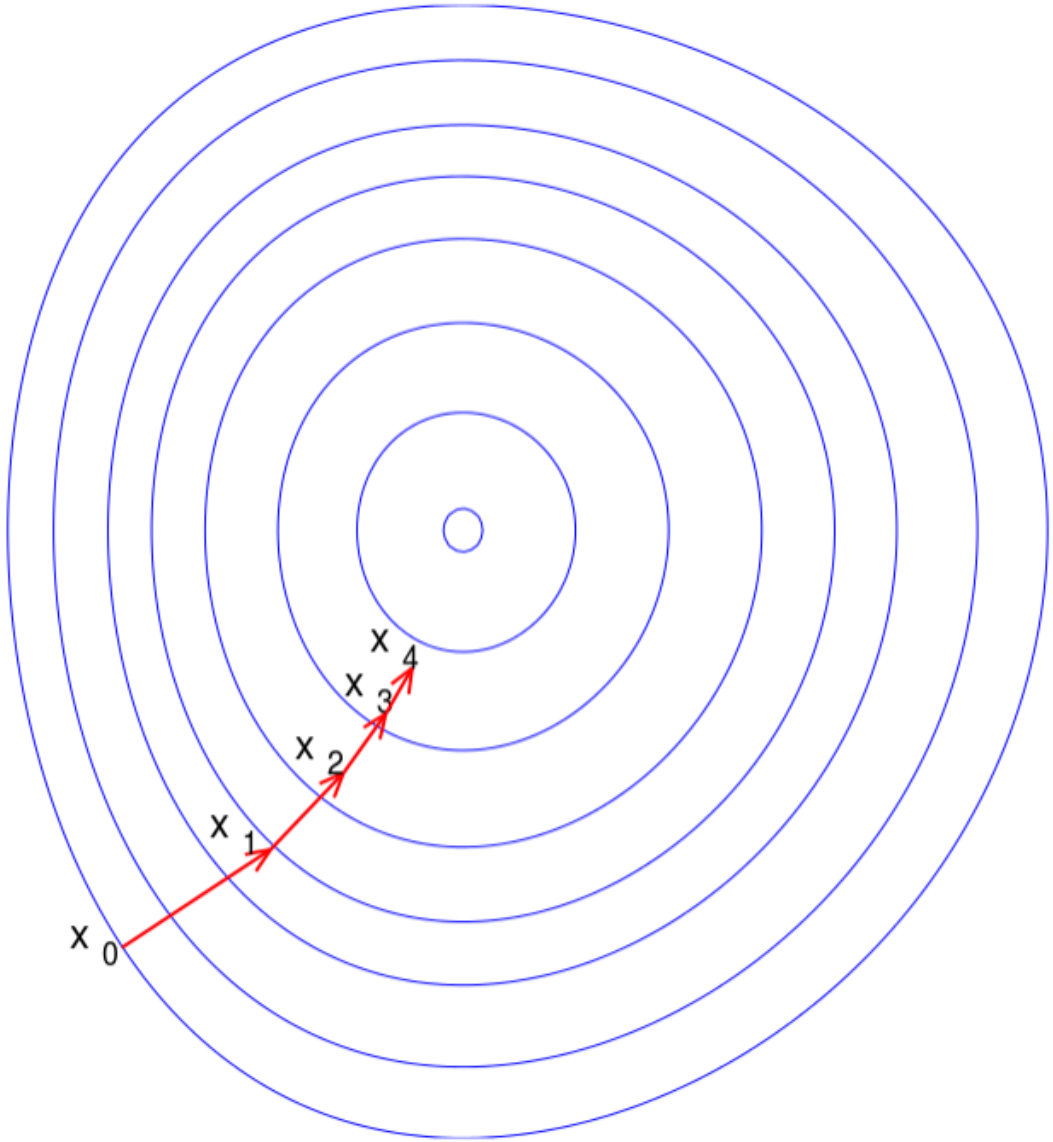


Figure 20.1 illustrates gradient descent in two dimensions.

- The steps get smaller as we approach the minimum because the gradient becomes less steep.
- Here there is just one local minimum, which is also the global minimum.

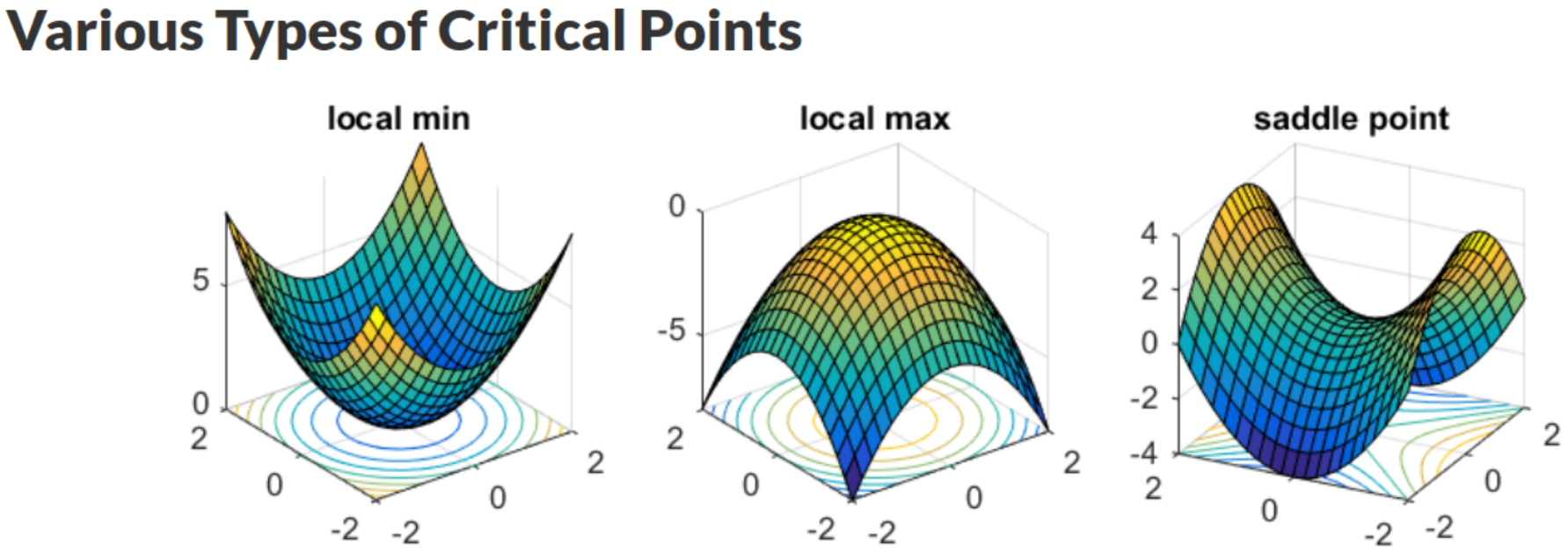
If there are two or more local minima, gradient descent will only find one of them, and it may be the wrong one.

However, there must be many equally-good local minima. Imagine just switching the labels of two hidden units.

We could try multiple random starting places, or we could occasionally try random large steps that are not gradient descent steps, just to see what happens.

Most such steps will make $R(\boldsymbol{\theta})$ larger, so we reject them. But the odd one may help, and so we take it and then start descending again.

Figure 20.2 Three cases where the gradient is zero



For high-dimensional problems, saddle points are commonly encountered. See Figure 20.2.

We don't want to stop at a local maximum or a saddle point.

A steepest descent algorithm is unlikely to find a local maximum, but it might find a saddle point.

Using what ISLR/ISLP calls **stochastic gradient descent**, but other authors call **mini-batch gradient descent**, solves the saddle-point problem and has other advantages.

Instead of computing $\nabla R(\boldsymbol{\theta})$ for the entire sample, we just compute it for a small fraction of the observations, or a mini-batch.

For example, each mini-batch might consist of 32, 64, 128, or 256 randomly-chosen observations. Powers of 2 are used because cache sizes are usually powers of 2.

If a mini-batch contains 128 observations, processing $n/128$ mini-batches means going through the entire sample once. This is called an **epoch**.

This approach involves updating $\boldsymbol{\theta}$ a great many times, but evaluating only part of $R(\boldsymbol{\theta})$ and $\nabla R(\boldsymbol{\theta})$ each time.

- The mini-batches may be chosen randomly (without replacement) each time they are used.
- For large samples, it makes sense to randomize the order of the observations at the beginning of each epoch, then take mini-batches sequentially.

Unlike with many other ML methods, it is common to use a separate validation sample when fitting neural networks.

Then it is customary to keep track of how the fit evolves for both the training and validation sets as the number of epochs increases.

See Figure 20.3, which shows this for the `MNIST` dataset of handwritten digits. There are 48,000 training observations and 12,000 validation ones. A different random training/validation split would lead to different results.

Figure 20.3 Evolution of training and validation errors

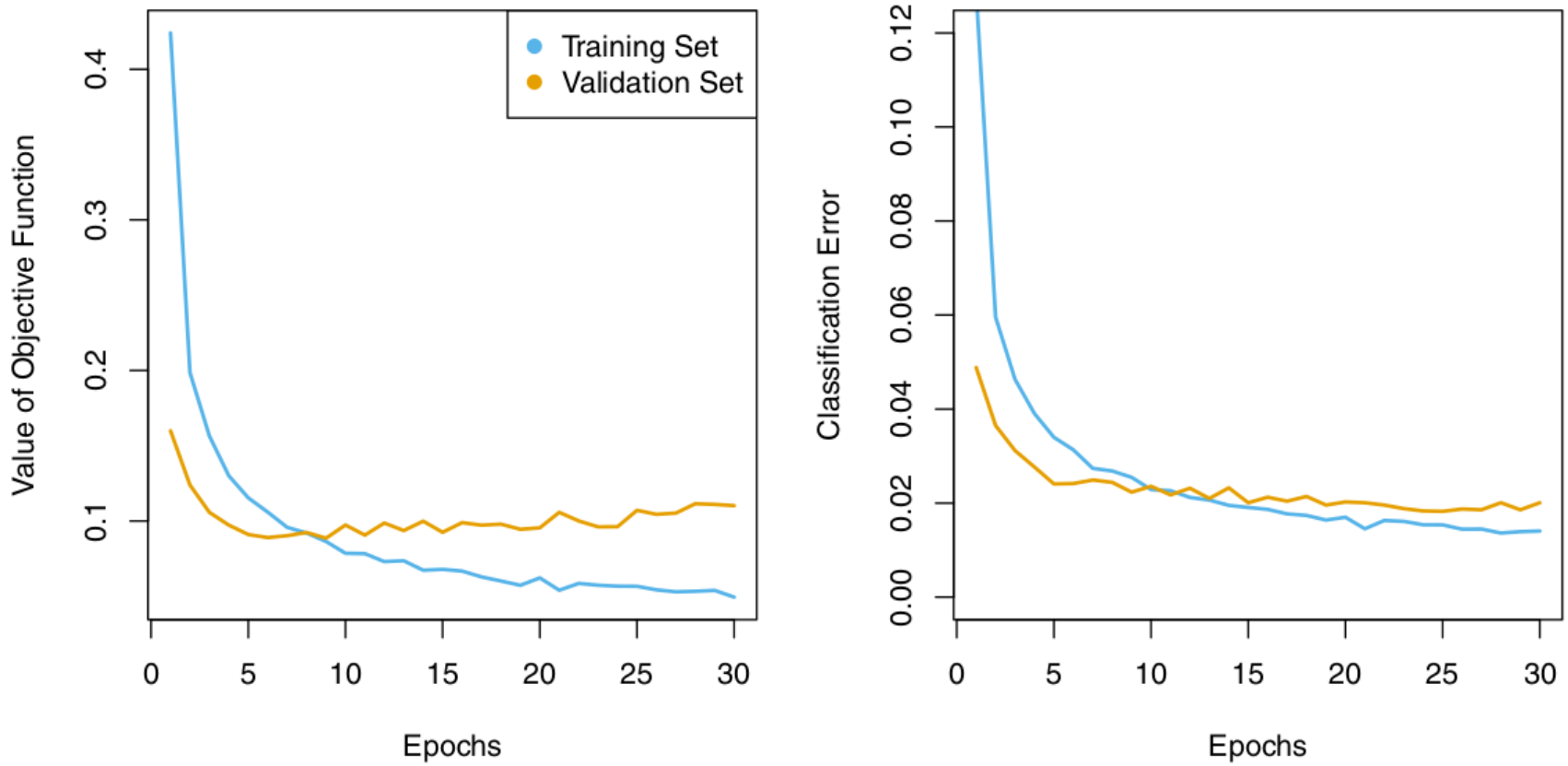


FIGURE 10.18. *Evolution of training and validation errors for the* `MNIST` *neural network depicted in Figure 10.4, as a function of training epochs. The objective refers to the log-likelihood (10.14).*

As expected, performance on the training dataset improves steadily as the number of epochs increases.

However, the objective function for the validation dataset seems to be minimized after 5 epochs and increases after that, although the classification error rate continues to improve.

- One way to avoid over-fitting of neural networks is **early stopping**, which in this case would mean stopping after 5 epochs.
- We monitor performance on the validation sample and stop just before it starts to get worse.

For the `MNIST` example, ISLR/ISLP used ridge regularization. They only penalized the parameters in $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$, not the relatively small number in $\boldsymbol{B}$. They used a small, preset value of λ.

Lasso regularization could also be used, either instead of ridge or in addition to it, as in elastic net.

Dropout Learning

This is a recent innovation (Hinton et al., 2014). We randomly remove some of the units in one or more layers when training the model.

If the fraction removed is ϕ, then the remaining weights are scaled up by a factor of $1/(1-\phi)$.

At the start of each epoch, the activations of the dropped-out units are randomly set to zero.

Like regularization, this helps to prevent over-fitting.

Conceptually, it is similar to random forests. But estimating a bunch of neural nets and averaging them would be very expensive.

Dropout may be implemented on any or all hidden layers as well as the input layer. It is not used on the output layer.

It often results in a sparser network. The value of ϕ may change across the layers, being low (say 0.2) for the input layer and higher (say 0.5) for the topmost hidden layer.

Figure 20.4 Dropout learning

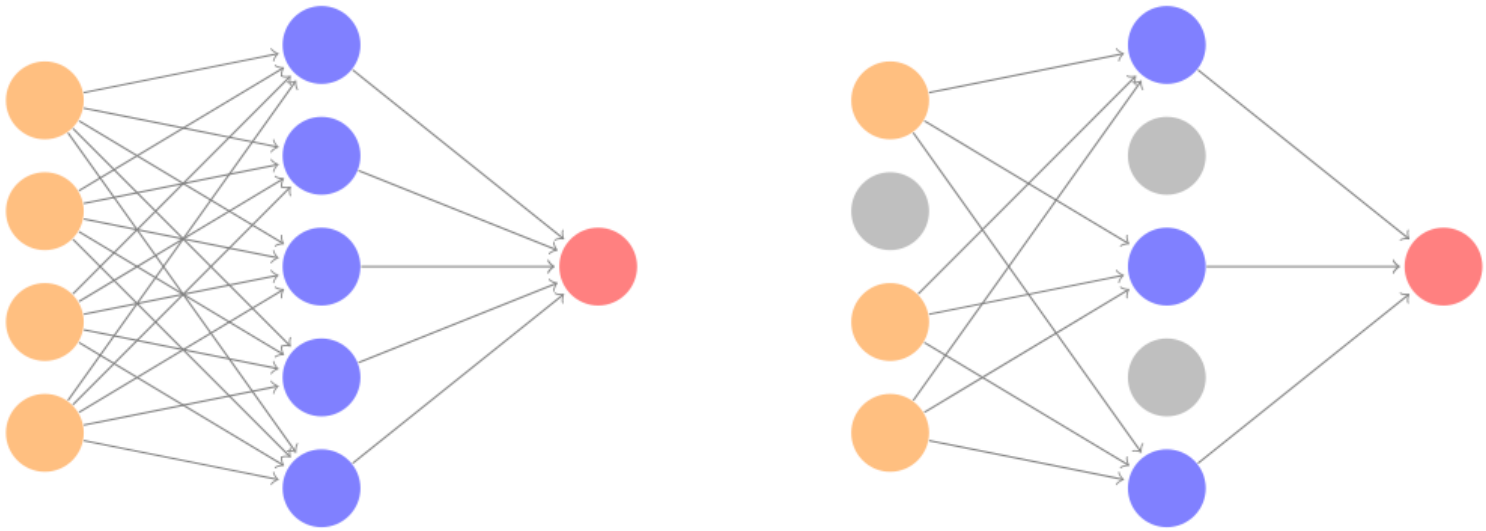


FIGURE 10.19. *Dropout Learning. Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random, and ignored in an instance of training.*

Network Tuning

Estimating a neural network requires making a lot of decisions, all of which can affect its performance. Even for a simple feed-forward network, we need to decide:

- Number of hidden layers, and number of units per layer.
- How to regularize each layer, and the tuning parameter(s) to use. These include the dropout rate(s) ϕ and values of λ for ridge and/or lasso regularization.
- Details of stochastic gradient descent, including batch sizes, the number of epochs, and possibly data augmentation.
- The starting point(s) for gradient descent. Starting with all parameters zero seems natural but is a bad choice. Using several random starting points is probably a good idea.

There may be infinitely many sets of "good" estimates.

Double Descent

Neural networks can have an odd feature that seems to contradict what we know about bias-variance tradeoffs.

For most statistical learning methods, increasing flexibility by adding parameters or relaxing restrictions reduces bias but increases variance.

If we plot both training and test errors against flexibility, we find that:

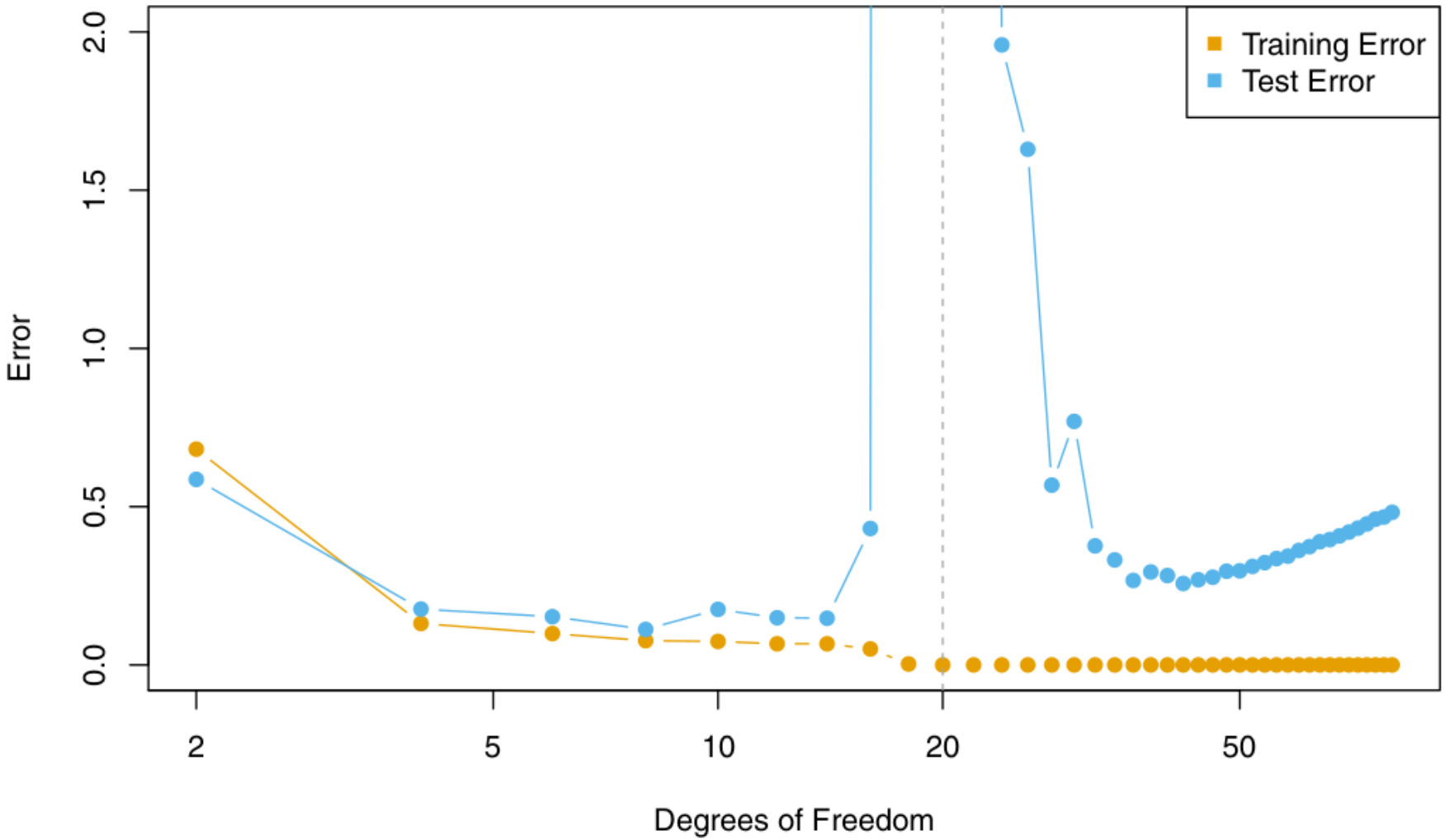
- Training error decreases monotonically as flexibility increases.
- Test error initially decreases and then increases.

If a model is so flexible that it can fit every point in the training set perfectly, it is said to **interpolate** the data.

For most methods, this is a bad thing. A model that interpolates the data normally will have very high variance.

But consider Figure 20.5.

Figure 20.5 An example of double descent



The data here come from the very simple DGP

$$y = \sin(x) + \epsilon, \quad \epsilon \sim \mathrm{N}(0, 0.09), \tag{1}$$

where $n = 20$ and the $x_i \sim \mathrm{U}(-5, 5)$.

The model is a natural cubic spline with d knots at the d quantiles of the x_i. If $d = 4$, these are the 0.2, 0.4, 0.6, and 0.8 quantiles.

For $d \geq 20$, the spline model fits perfectly, and the training error is therefore zero.

As d increases towards 20, the test error initially falls, achieves a minimum at $d = 8$, and then goes crazy for $d > 16$.

But suppose we keep going beyond $d = 20$. There is now an infinite number of ways to make the spline model fit perfectly.

For each d, we choose the **minimum-norm** solution that makes $\sum_{j=1}^{d} \hat{\beta}_j^2$ as small as possible.

As d increases beyond 20, the fit to the training data does not change, but the fit to the test data initially improves!

It improves very fast initially, achieving a local minimum at $d = 42$.

After that, it gradually gets worse.

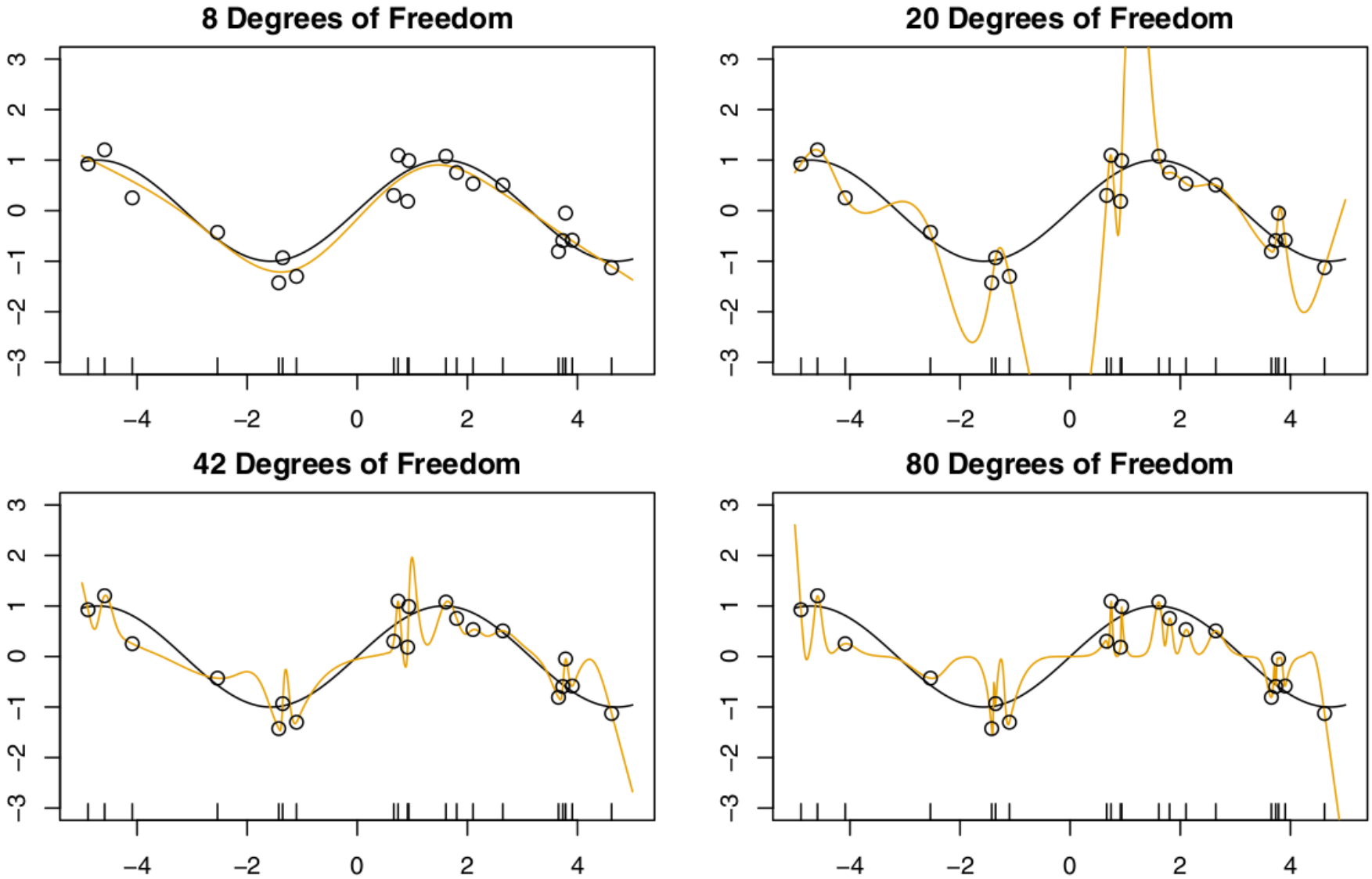
The lowest test errors for $d < 20$ are noticeably better than the lowest ones for $d > 20$.

Figure 20.6 shows why the test error improves with d, at least up to a point, as d increases beyond 20.

There are infinitely many ways to interpolate the training data, and the minimum-norm solution becomes smoother as d increases.

The signal-to-noise ratio is quite high for this model, so that interpolating more smoothly gets the model closer to the true regression function.

Figure 20.6 Why we see double descent



The double-descent phenomenon does not contradict the existence of a bias-variance tradeoff.

In the example, the number of knots does not properly capture the flexibility of the spline model.

Most learning methods never interpolate the data (fit perfectly). In this case, using ridge regularization with the natural cubic spline would have avoided interpolation and performed better.

- Allowing neural nets to interpolate is most likely to work well when the signal-to-noise ratio is high.
- For neural networks, the number of parameters can be enormous, making it likely that they will fit perfectly.
- Ridge regularization and early stopping are two ways to prevent this from happening.

Support vector machines can sometimes interpolate the data but still work well, because they also involve minimum-norm solutions.

Section 10.9 shows how to estimate the neural nets that are illustrated in the chapter.

It uses an R package called `keras`, which in turn uses a Python package called `tensorflow`.

After some effort, I found this page

`https://hastie.su.domains/ISLR2/keras-instructions.html`

which suggests that getting `keras`, Python, and `tensorflow` to work together is very difficult.

Since ISLR2 was published, a new package called `torch` has become available. It does not depend on Python packages and is therefore much easier to install.

A new version of the lab in Section 10.9 for use with torch may be found here:
`https://hastie.su.domains/ISLR2/Labs/R_Labs/Ch10-deeplearning-lab-torch.R`