Document Classification

We will classify critiques of movies into positive or negative categories (`sentiment`).

Each review can be a different length, include slang or non-words, have spelling errors, etc. We need to find a way to **featurize** such a document.

The simplest and most common featurization is the **bag-of-words model**.
每个文档被转换为一个与词典等长的向量，用1/0代表相应位置的单词是否出现。词典可以取训练文本中最常出现的10,000个词。

> <\START> this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert <\UNK> is an amazing actor and now the same being director <\UNK> father came from the same scottish island as myself so i loved . . .

<\UNK>: unknown words.

Only 1.3% of the binary entries are nonzero. We call such a matrix **sparse**; it can be stored efficiently in sparse matrix format (by only storing the locations of nonzero entries).

One could instead record the relative frequency of words.

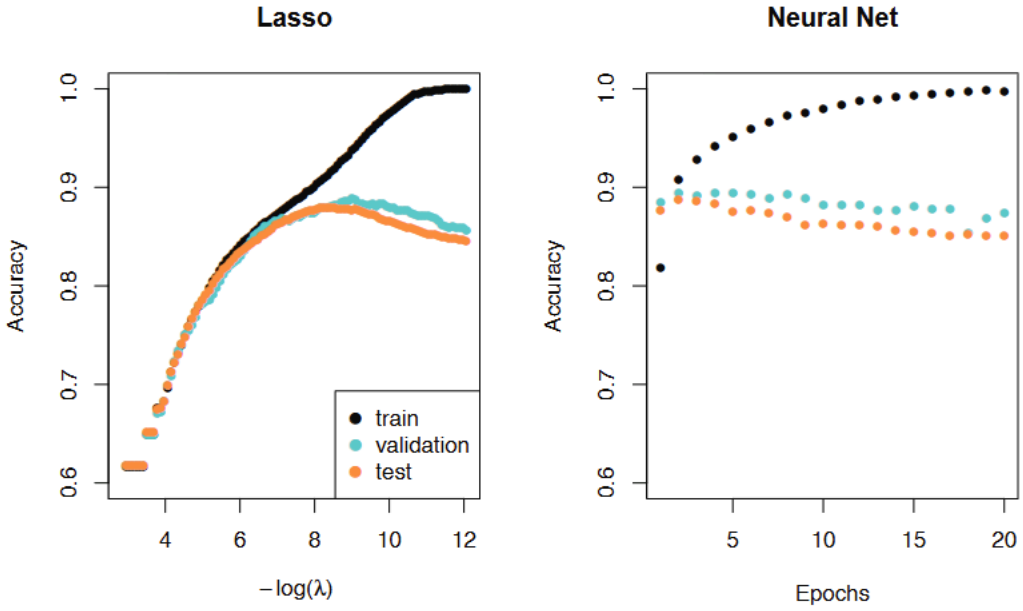

FIGURE 10.11. *Accuracy of the lasso and a two-hidden-layer neural network on the* IMDb *data. For the lasso, the x-axis displays $-\log(\lambda)$, while for the neural network it displays epochs (number of times the fitting algorithm passes through the training set). Both show a tendency to overfit, and achieve approximately the same test accuracy.*

Note that a two-class neural network amounts to a nonlinear logistic regression model.

$$\begin{aligned}\log\left(\frac{\Pr(Y=1\mid X)}{\Pr(Y=0\mid X)}\right) &= Z_1 - Z_0 \\ &= (\beta_{10}-\beta_{00}) + \sum_{\ell=1}^{K_2}(\beta_{1\ell}-\beta_{0\ell})\,A_\ell^{(2)}\end{aligned}$$

The bag-of-words model summarizes a document by the words present, and ignores their context. There are at least two popular ways to take the context into account:

- The bag-of-n-grams model. For example, a bag of 2-grams records
  the consecutive co-occurrence of every distinct pair of words. "Blissfully long" can be seen as a positive phrase in a movie review, while "blissfully short" a negative.
- Treat the document as a sequence, taking account of all the words in the context of those that preceded and those that follow.

Recurrent Neural Networks

Many data sources are sequential in nature, and call for special treatment.

- Documents. (topic classification, sentiment analysis, and language translation)
- Time series.
- Sound recordings.
- Handwriting. (optical character recognition)

The order of the words, and closeness of certain words in a sentence, convey semantic meaning. RNNs are designed to accommodate and take advantage of the sequential nature of such input objects.

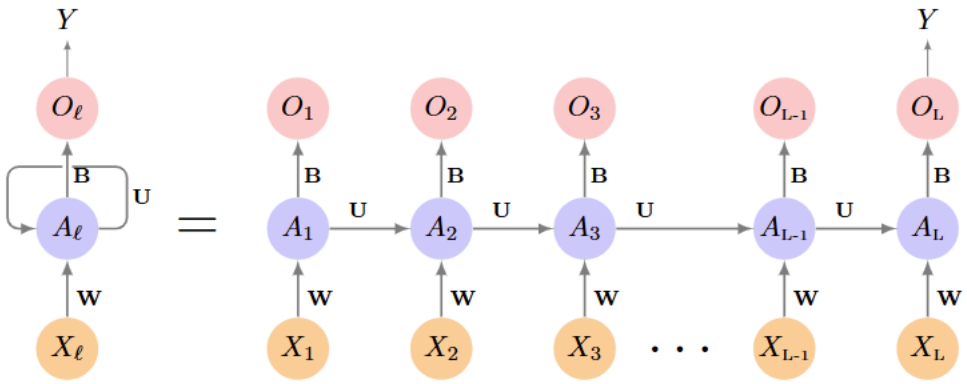

FIGURE 10.12. *Schematic of a simple recurrent neural network. The input is a sequence of vectors* $\{X_\ell\}_1^L$*, and here the target is a single response. The network processes the input sequence* X *sequentially; each* X_ℓ *feeds into the hidden layer, which also has as input the activation vector* $A_{\ell-1}$ *from the previous element in the sequence, and produces the current activation vector* A_ℓ*. The same collections of weights* $\mathbf{W}$*,* $\mathbf{U}$ *and* $\mathbf{B}$ *are used as each element of the sequence is processed. The output layer produces a sequence of predictions* O_ℓ *from the current activation* A_ℓ*, but typically only the last of these,* O_L*, is of relevance. To the left of the equal sign is a concise representation of the network, which is* unrolled *into a more explicit version on the right.*

Each X_ℓ is a vector representing a word. As the sequence is processed one vector X_ℓ at a time, the network updates the activations A_ℓ in the hidden layer, taking as input the vector X_ℓ and the activation vector $A_{\ell-1}$ from the previous step in the sequence.
Each A_ℓ feeds into the output layer and produces a prediction O_ℓ for Y. O_L, the last of these, is the most relevant.

Suppose each X_ℓ has p components, and the hidden layer consists of K units.
$\mathbf{W}$ is a $K \times (p+1)$ weight matrix for the input layer;
$\mathbf{U}$ is a $K \times K$ weight matrix for the hidden-to-hidden layers;
$\mathbf{B}$ is a $K+1$ vector of weights for the output layer.

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right)$$

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}$$

Here $g(\cdot)$ is an activation function such as ReLU.

Notice that the same weights $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ are used. This is a form of **weight sharing** used by RNNs, and similar to the use of filters in convolutional neural network.

For regression problems the loss function for an observation (X, Y) is

$$(Y - O_L)^2$$

which only references the final output O_L. The intermediate outputs O_ℓ are not used.
Each element X_ℓ of the input sequence X contributes to O_L via the chain, and hence contributes indirectly to learning the shared parameters $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ via the loss.

O_L provide an evolving prediction for the output. For some learning tasks the response is also a sequence, and the output sequence $\{O_1, O_2, \ldots, O_L\}$ is explicitly needed.

Sequential Models for Document Classification

We have a dimensionality problem using one-hot-encoded vector to represent a word.
An approach that has become popular is to represent each word in a m-dimensional embedding space. Then we need a matrix $\mathbf{E}$ of dimension $m \times 10,000$,

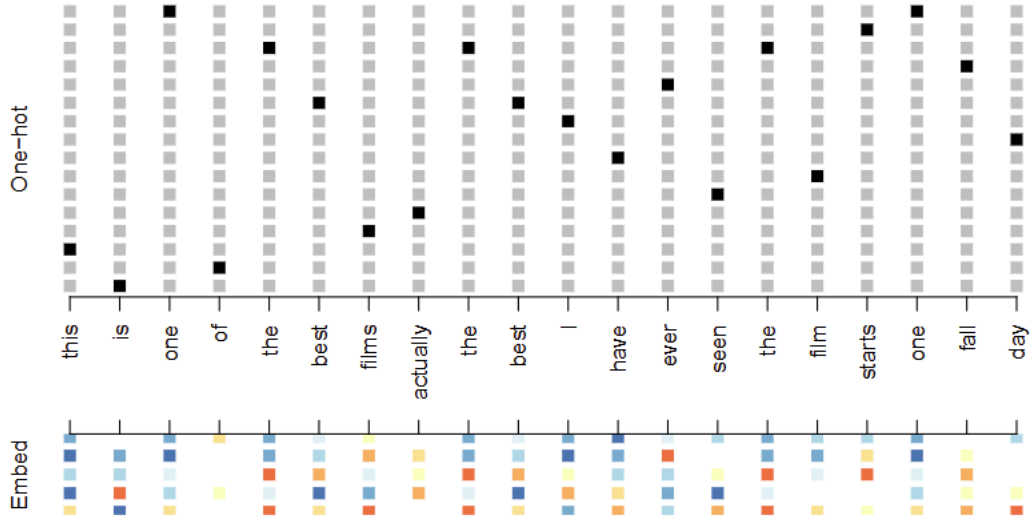

FIGURE 10.13. *Depiction of a sequence of* 20 *words representing a single document: one-hot encoded using a dictionary of* 16 *words (top panel) and embedded in an* m*-dimensional space with* $m = 5$ *(bottom panel).*

If we have a large corpus of labeled documents, we can have the neural network learn $\mathbf{E}$ as part of the optimization. Otherwise we can insert a precomputed matrix $\mathbf{E}$ in the embedding layer, a process known as **weight freezing**.
Two pretrained embeddings, word2vec and GloVe, are widely used. These are built from a very large corpus of documents by a variant of principal components analysis.

The idea is that the positions of words in the embedding space preserve semantic meaning; e.g. synonyms should appear near each other.

Now each document is now represented as a sequence of m vectors that represents the sequence of words. Then we limit each document to L words by truncating the sequence or padding them with zeros upfront. So now each document is represented by a series consisting of L vectors $X = \{X_1, X_2, \ldots, X_L\}$, and each X_ℓ in the sequence has m components.

We now use the RNN structure. A parallel series of hidden activation vectors $A_\ell, \ell = 1, \ldots, L$ is created for each document.

More elaborate versions of RNN use long term and short term memory (LSTM). Two tracks of hidden-layer activations are maintained, so that when the activation A_ℓ is computed, it gets input from hidden units both further back in time, and closer in time. With long sequences, this overcomes the problem of early signals being washed out by the time they get propagated through the chain to the final activation vector A_L.

LSTM models take a long time to train, which makes exploring many architectures and parameter optimization tedious.

Summary of RNNs

There are many variations and enhancements of the simple RNN we used for sequence modeling.

- One can use a one-dimensional convolutional neural network, treating the sequence of vectors as an image. The convolution filter slides along the sequence in a one-dimensional fashion, with the potential to learn particular phrases or short subsequences relevant to the learning task.
- One can also have additional hidden layers.
- bidirectional RNNs scan the sequences in both directions.

In language translation the target is also a sequence of words. In this so-called Seq2Seq learning, the hidden units are thought to capture the semantic meaning of the sentences.

Algorithms used to fit RNNs can be complex and computationally costly.

When to Use Deep Learning

Linear models are much easier to present and understand than the neural network, which is essentially a black box. We are much better off following the Occam's razor principle: when faced with several methods that give roughly equivalent performance, pick the simplest.

Wherever possible, it makes sense to try the simpler models as well, and then make a choice based on the performance/complexity tradeoff.

Typically we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority.

Fitting a Neural Network

- $\beta = (\beta_0, \beta_1, \ldots, \beta_K)$
- $w_k = (w_{k0}, w_{k1}, \ldots, w_{kp})$

We could fit the model by solving a nonlinear least squares problem

$$\underset{\{w_k\}_1^K, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} \left(y_i - f\left(x_i\right)\right)^2$$

where

$$f\left(x_i\right) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\right)$$

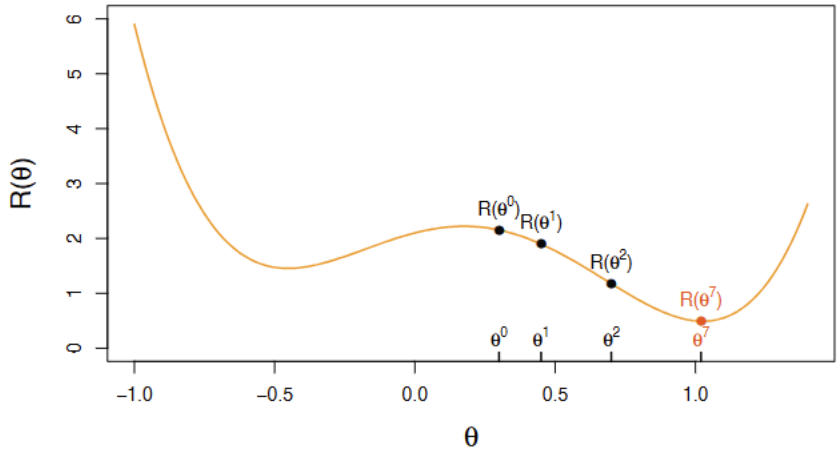

FIGURE 10.17. *Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill — against the gradient — until it cannot go down any further. Here gradient descent reached the global minimum in 7 steps.*

There are two solutions: one is a local minimum and the other is a global minimum.

Two general strategies are employed when fitting neural networks.

- Slow Learning: the model is fit in a somewhat slow iterative fashion, using gradient descent. The fitting process is then stopped when overfitting is detected.
- Regularization: penalties are imposed on the parameters, usually lasso or ridge.

Suppose we represent all the parameters in one long vector θ. Then we can rewrite the objective as

$$R(\theta) = \frac{1}{2} \sum_{i=1}^{n} \left(y_i - f_\theta\left(x_i\right)\right)^2$$

where we make explicit the dependence of f on the parameters.

The idea of gradient descent is:

1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.

2. Iterate until the objective fails to decrease:
   (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective.
   (b) Set $t \leftarrow t + 1$.

In general we can hope to end up at a (good) local minimum.

Backpropagation

How do we find the directions to move θ ?

The gradient of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of partial derivatives at that point:

$$\nabla R\left(\theta^m\right) = \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta^m}$$

The subscript $\theta = \theta^m$ means that after computing the vector of derivatives, we evaluate it at the current guess, θ^m.
This gives the direction in θ-space in which $R(\theta)$ increases most rapidly. The idea of gradient descent is to move θ a little in the opposite direction (since we wish to go downhill):

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R\left(\theta^m\right)$$

For a small enough value of the learning rate ρ, this step will decrease the objective $R(\theta)$. If the gradient vector is zero, then we may have arrived at a minimum of the objective.

We draw upon the power of the chain rule of differentiation.

$$R_i(\theta) = \frac{1}{2}\left(y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\right)\right)^2$$

To simplify the expressions to follow, we write $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$.

The derivative with respect to β_k :

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial \beta_k} &= \frac{\partial R_i(\theta)}{\partial f_\theta\left(x_i\right)} \cdot \frac{\partial f_\theta\left(x_i\right)}{\partial \beta_k} \\ &= -\left(y_i - f_\theta\left(x_i\right)\right) \cdot g\left(z_{ik}\right) \end{aligned} \tag{10.29}$$

The derivative with respect to w_{kj} :

$$\begin{aligned} \frac{\partial R_i(\theta)}{\partial w_{kj}} &= \frac{\partial R_i(\theta)}{\partial f_\theta\left(x_i\right)} \cdot \frac{\partial f_\theta\left(x_i\right)}{\partial g\left(z_{ik}\right)} \cdot \frac{\partial g\left(z_{ik}\right)}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} \\ &= -\left(y_i - f_\theta\left(x_i\right)\right) \cdot \beta_k \cdot g'\left(z_{ik}\right) \cdot x_{ij} \end{aligned} \tag{10.30}$$

Notice that both these expressions contain the residual $y_i - f_\theta\left(x_i\right)$.

In (10.29) we see that a fraction of that residual gets attributed to each of the hidden units according to the value of $g\left(z_{ik}\right)$.
Then in (10.30) we see a similar attribution to input j via hidden unit k.
So the act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule - a process known as **backpropagation** in the neural network literature.

When n is large, instead of summing (10.29)-(10.30) over all n observations, we can sample a small fraction or minibatch of them each time we compute a gradient step. This process is known as **stochastic gradient descent** (SGD) and is the state of the art for learning deep neural networks.

Regularization and Stochastic Gradient Descent

Gradient descent usually takes many steps to reach a local minimum. In practice, there are a number of approaches for accelerating the process.

We now turn to the multilayer network used in the digit recognition problem. The network has over 235,000 weights, which is around four times the number of training examples.

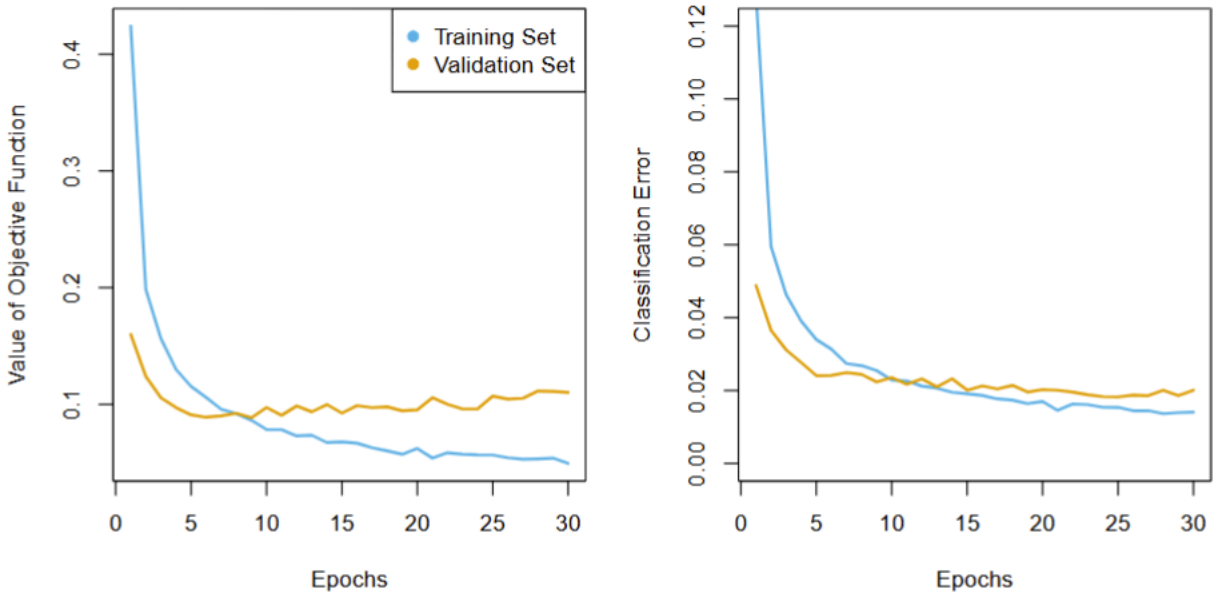

FIGURE 10.18. *Evolution of training and validation errors for the* `MNIST` *neural network depicted in Figure 10.4, as a function of training epochs. The objective refers to the log-likelihood (10.14).*

Regularization is essential here to avoid overfitting.

$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log\left(f_m\left(x_i\right)\right) + \lambda\sum_{j}\theta_j^2$$

We can use different values of λ for the groups of weights from different layers; in this case $\mathbf{W}_1$ and $\mathbf{W}_2$ were penalized, while the relatively few weights $\mathbf{B}$ of the output layer were not penalized at all. Lasso regularization is also popular as an additional form of regularization.

It turns out that SGD naturally enforces its own form of approximately quadratic regularization.

Here the minibatch size was 128 observations per gradient update. The term epochs counts **the number of times an equivalent of the full training set has been processed**. For this network, 20% of the 60,000 training observations were used as a validation set in order to determine when training should stop. So in fact 48,000 observations were used for training, and hence there are $48,000/128 \approx 375$ minibatch gradient updates per epoch.

We see that the value of the validation objective actually starts to increase by 30 epochs, so **early stopping** can also be used as an additional form of regularization.

Dropout Learning

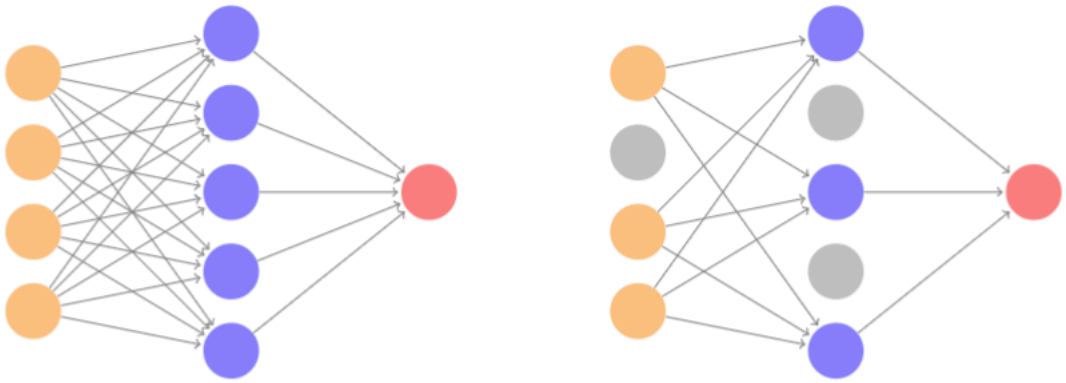

FIGURE 10.19. *Dropout Learning. Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random, and ignored in an instance of training.*

Inspired by random forests (Section 8.2), the idea is to randomly remove a fraction ϕ of the units in a layer when fitting the model. This is done separately each time a

training observation is processed.

The surviving units stand in for those missing, and their weights are scaled up by a factor of $1/(1-\phi)$ to compensate. This prevents nodes from becoming over-specialized, and can be seen as a form of regularization.

In practice dropout is achieved by randomly setting the activations for the "dropped out" units to zero, while keeping the architecture intact.

Network Tuning

We have to make a number of choices:

- The number of hidden layers, and the number of units per layer. Modern thinking is that the number of units per hidden layer can be large, and overfitting can be controlled via the various forms of regularization.
- Regularization tuning parameters. These include the dropout rate ϕ and the strength λ of lasso and ridge regularization, and are typically set separately at each layer.
- Details of stochastic gradient descent. These include the batch size, the number of epochs, and if used, details of data augmentation.

Interpolation and Double Descent

One implication of the bias-variance trade-off is that it is generally not a good idea to interpolate the training data - that is, to get zero training error - since that will often result in very high test error.

However, in certain specific settings it can be possible for
a statistical learning method that interpolates the training data to perform well. This phenomenon is known as **double descent**.

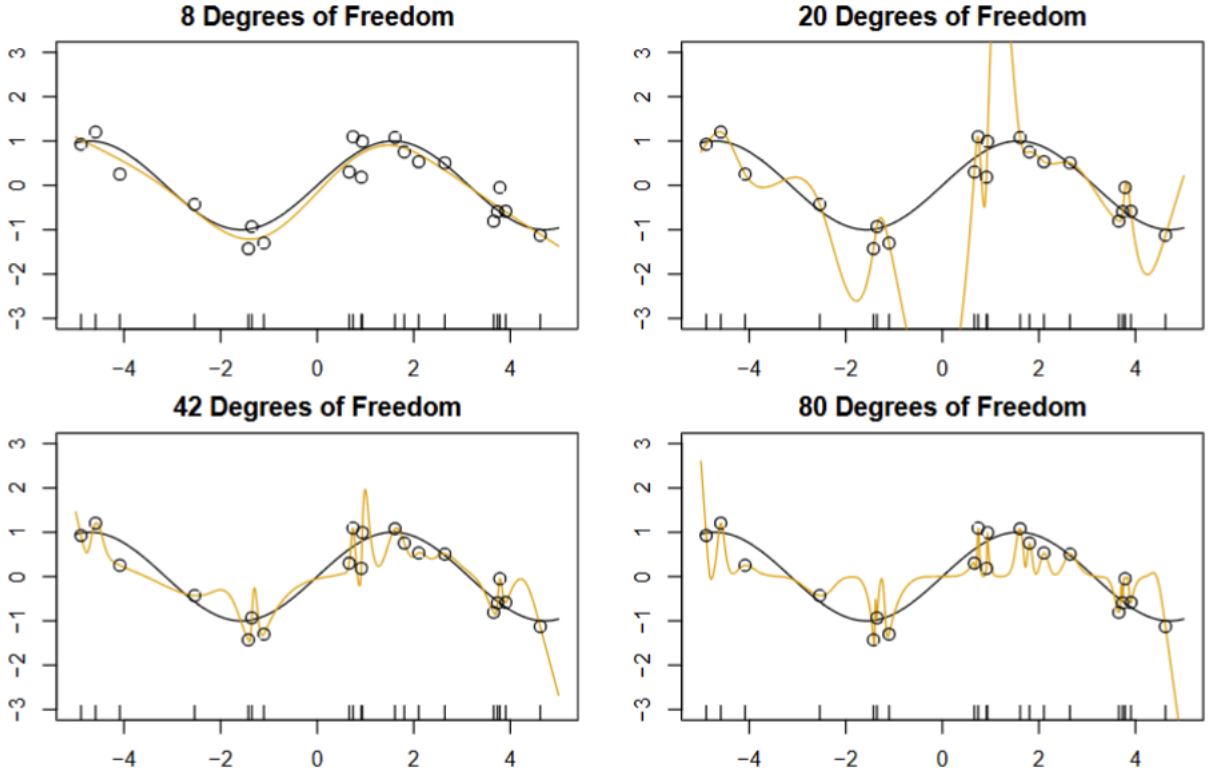

FIGURE 10.21. *Fitted functions $\hat{f}_d(X)$ (orange), true function $f(X)$ (black) and the observed 20 training data points. A different value of d (degrees of freedom) is used in each panel. For $d \geq 20$ the orange curves all interpolate the training points, and hence the training error is zero.*

"Double descent" gets its name from the fact that the test error has a U-shape before the interpolation threshold is reached, and then it descends again (for a while, at least) as an increasingly flexible model is fit.

Essentially, $\hat{f}_{20}(X)$ is very wild because there is just a single way to interpolate $n = 20$ observations using $d = 20$ basis functions, and that single way results in a somewhat extreme fitted function.
By contrast, there are an infinite number of ways to interpolate $n = 20$ observations using $d = 42$ or $d = 80$ basis functions, and the smoothest of them - that is, the **minimum norm solution** (with the smallest sum of squared coefficients) - is much less wild than $\hat{f}_{20}(X)$!

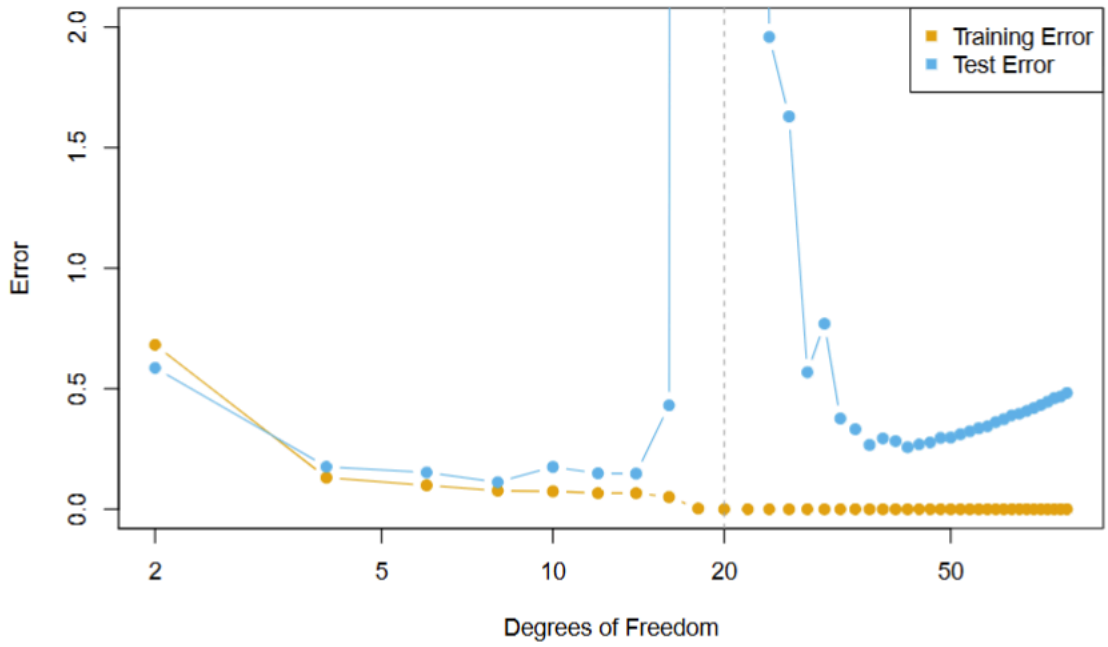

FIGURE 10.20. *Double descent phenomenon, illustrated using error plots for a one-dimensional natural spline example. The horizontal axis refers to the number of spline basis functions on the log scale. The training error hits zero when the degrees of freedom coincides with the sample size $n = 20$, the "interpolation threshold", and remains zero thereafter. The test error increases dramatically at this threshold, but then descends again to a reasonable value before finally increasing again.*

For this example the **signal-to-noise ratio** - $\mathrm{Var}(f(X))/\sigma^2$ - is 5.9 , which is quite high (the data points are close to the true curve). So an estimate that interpolates the data and does not wander too far inbetween the observed data points will likely do well.

The same phenomenon can arise for deep learning. Basically, when we fit neural networks with a huge number of parameters, we are sometimes able to get good results with zero training error. This is particularly true in problems with high signal-to-noise ratio, such as natural image recognition and language translation, for example. This is because the techniques used to fit neural networks, including stochastic gradient descent, naturally lend themselves to selecting a "smooth" interpolating model that has good test-set performance on these kinds of problems.

Some points are worth emphasizing:

- The double-descent phenomenon does not contradict the bias-variance trade-off. The number of spline basis functions used does not properly capture the true "flexibility" of models that interpolate the training data. Stated another way, in this example, the minimum-norm natural spline with $d = 42$ has lower variance than the natural spline with $d = 20$.
- Regularization approaches can give great results without interpolating the data!
- In Chapter 9, we saw that maximal margin classifiers and SVMs that have zero training error nonetheless often achieve very good test error. This is in part because those methods seek smooth minimum norm solutions.

However, fitting to zero error is not always optimal, and whether it is advisable depends on the signal-to-noise ratio.

Early stopping during stochastic gradient descent can also serve as a form of regularization that prevents us from interpolating the training data, while still getting very good results on test data.

To summarize: though double descent can sometimes occur in neural networks, we typically do not want to rely on this behavior. Moreover, it is important to remember that the bias-variance trade-off always holds (though it is possible that test error as a function of flexibility may not exhibit a U-shape, depending on how we have parametrized the notion of "flexibility" on the x-axis).