10
Deep Learning

This chapter covers the important topic of *deep learning*. At the time of writing (2020), deep learning is a very active area of research in the machine learning and artificial intelligence communities. The cornerstone of deep learning is the *neural network*.

deep learning

neural network

Neural networks rose to fame in the late 1980s. There was a lot of excitement and a certain amount of hype associated with this approach, and they were the impetus for the popular *Neural Information Processing Systems* meetings (NeurIPS, formerly NIPS) held every year, typically in exotic places like ski resorts. This was followed by a synthesis stage, where the properties of neural networks were analyzed by machine learners, mathematicians and statisticians; algorithms were improved, and the methodology stabilized. Then along came SVMs, boosting, and random forests, and neural networks fell somewhat from favor. Part of the reason was that neural networks required a lot of tinkering, while the new methods were more automatic. Also, on many problems the new methods outperformed poorly-trained neural networks. This was the *status quo* for the first decade in the new millennium.

All the while, though, a core group of neural-network enthusiasts were pushing their technology harder on ever-larger computing architectures and data sets. Neural networks resurfaced after 2010 with the new name *deep learning*, with new architectures, additional bells and whistles, and a string of success stories on some niche problems such as image and video classification, speech and text modeling. Many in the field believe that the major reason for these successes is the availability of ever-larger training datasets, made possible by the wide-scale use of digitization in science and industry.

In this chapter we discuss the basics of neural networks and deep learning, and then go into some of the specializations for specific problems, such as convolutional neural networks (CNNs) for image classification, and recurrent neural networks (RNNs) for time series and other sequences. We will also demonstrate these models using the `Python` package `keras`, which interfaces with the `tensorflow` deep-learning software developed at Google.[1]

The material in this chapter is slightly more challenging than elsewhere in this book.

10.1 Single Layer Neural Networks

A neural network takes an input vector of p variables $X = (X_1, X_2, \ldots, X_p)$ and builds a nonlinear function $f(X)$ to predict the response Y. We have built nonlinear prediction models in earlier chapters, using trees, boosting and generalized additive models. What distinguishes neural networks from these methods is the particular *structure* of the model. Figure 10.1 shows a simple *feed-forward neural network* for modeling a quantitative response using $p = 4$ predictors. In the terminology of neural networks, the four features $X_1, \ldots, X_4$ make up the units in the *input layer*. The arrows indicate that each of the inputs from the input layer feeds into each of the K *hidden units* (we get to pick K; here we chose 5). The neural network model has the form

feed-forward neural network

input layer

hidden units

$$\begin{aligned} f(X) &= \beta_0 + \textstyle\sum_{k=1}^{K} \beta_k h_k(X) \\ &= \beta_0 + \textstyle\sum_{k=1}^{K} \beta_k g(w_{k0} + \textstyle\sum_{j=1}^{p} w_{kj} X_j). \end{aligned} \tag{10.1}$$

It is built up here in two steps. First the K *activations* A_k, $k = 1, \ldots, K$, in the hidden layer are computed as functions of the input features $X_1, \ldots, X_p$,

activations

$$A_k = h_k(X) = g(w_{k0} + \textstyle\sum_{j=1}^{p} w_{kj} X_j), \tag{10.2}$$

where $g(z)$ is a nonlinear *activation function* that is specified in advance. We can think of each A_k as a different transformation $h_k(X)$ of the original features, much like the basis functions of Chapter 7. These K activations from the hidden layer then feed into the output layer, resulting in

activation function

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k, \tag{10.3}$$

a linear regression model in the $K = 5$ activations. All the parameters $\beta_0, \ldots, \beta_K$ and $w_{10}, \ldots, w_{Kp}$ need to be estimated from data. In the early

[1] For more information about `keras`, see Chollet et al. (2015) "Keras", available at https://keras.io. For more information about `tensorflow`, see Abadi et al. (2015) "TensorFlow: Large-scale machine learning on heterogeneous distributed systems", available at https://www.tensorflow.org/.

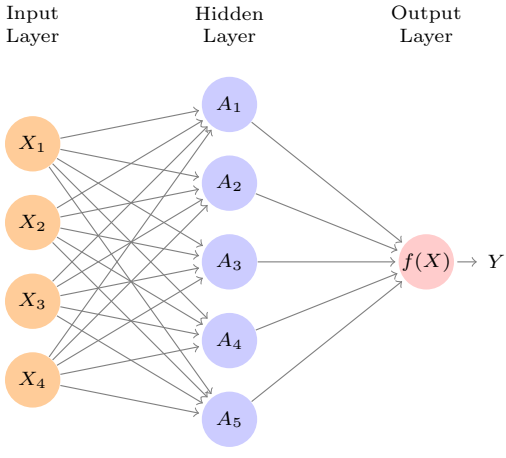

FIGURE 10.1. *Neural network with a single hidden layer. The hidden layer computes activations $A_k = h_k(X)$ that are nonlinear transformations of linear combinations of the inputs $X_1, X_2, \ldots, X_p$. Hence these A_k are not directly observed. The functions $h_k(\cdot)$ are not fixed in advance, but are learned during the training of the network. The output layer is a linear model that uses these activations A_k as inputs, resulting in a function $f(X)$.*

instances of neural networks, the *sigmoid* activation function was favored,

sigmoid

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}, \tag{10.4}$$

which is the same function used in logistic regression to convert a linear function into probabilities between zero and one (see Figure 10.2). The preferred choice in modern neural networks is the *ReLU* (*rectified linear unit*) activation function, which takes the form

ReLU
rectified linear unit

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise.} \end{cases} \tag{10.5}$$

A ReLU activation can be computed and stored more efficiently than a sigmoid activation. Although it thresholds at zero, because we apply it to a linear function (10.2) the constant term w_{k0} will shift this inflection point.

So in words, the model depicted in Figure 10.1 derives five new features by computing five different linear combinations of X, and then squashes each through an activation function $g(\cdot)$ to transform it. The final model is linear in these derived variables.

The name *neural network* originally derived from thinking of these hidden units as analogous to neurons in the brain — values of the activations $A_k = h_k(X)$ close to one are *firing*, while those close to zero are *silent* (using the sigmoid activation function).

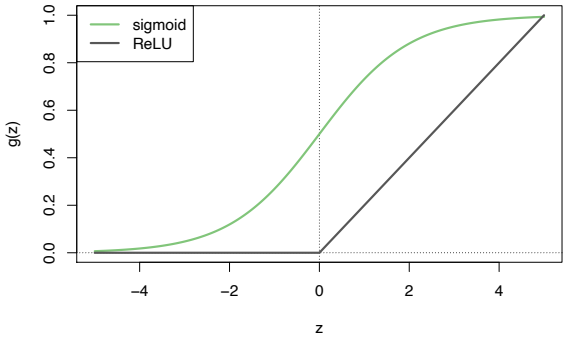

FIGURE 10.2. *Activation functions. The piecewise-linear* `ReLU` *function is popular for its efficiency and computability. We have scaled it down by a factor of five for ease of comparison.*

The nonlinearity in the activation function $g(\cdot)$ is essential, since without it the model $f(X)$ in (10.1) would collapse into a simple linear model in $X_1, \ldots, X_p$. Moreover, having a nonlinear activation function allows the model to capture complex nonlinearities and interaction effects. Consider a very simple example with $p = 2$ input variables $X = (X_1, X_2)$, and $K = 2$ hidden units $h_1(X)$ and $h_2(X)$ with $g(z) = z^2$. We specify the other parameters as

$$\begin{array}{lll} \beta_0 = 0, & \beta_1 = \frac{1}{4}, & \beta_2 = -\frac{1}{4}, \\ w_{10} = 0, & w_{11} = 1, & w_{12} = \ \ 1, \\ w_{20} = 0, & w_{21} = 1, & w_{22} = -1. \end{array} \tag{10.6}$$

From (10.2), this means that

$$\begin{array}{rcl} h_1(X) & = & (0 + X_1 + X_2)^2, \\ h_2(X) & = & (0 + X_1 - X_2)^2. \end{array} \tag{10.7}$$

Then plugging (10.7) into (10.1), we get

$$\begin{array}{rcl} f(X) & = & 0 + \frac{1}{4} \cdot (0 + X_1 + X_2)^2 - \frac{1}{4} \cdot (0 + X_1 - X_2)^2 \\ & = & \frac{1}{4}\left[(X_1 + X_2)^2 - (X_1 - X_2)^2\right] \\ & = & X_1 X_2. \end{array} \tag{10.8}$$

So the sum of two nonlinear transformations of linear functions can give us an interaction! In practice we would not use a quadratic function for $g(z)$, since we would always get a second-degree polynomial in the original coordinates $X_1, \ldots, X_p$. The sigmoid or ReLU activations do not have such a limitation.

Fitting a neural network requires estimating the unknown parameters in (10.1). For a quantitative response, typically squared-error loss is used, so that the parameters are chosen to minimize

$$\sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2. \tag{10.9}$$

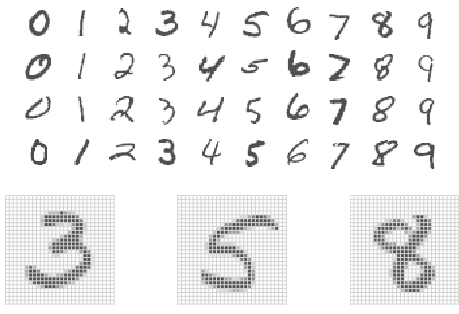

FIGURE 10.3. *Examples of handwritten digits from the* MNIST *corpus. Each grayscale image has* 28×28 *pixels, each of which is an eight-bit number (0–255) which represents how dark that pixel is. The first 3, 5, and 8 are enlarged to show their 784 individual pixel values.*

Details about how to perform this minimization are provided in Section 10.7.

10.2 Multilayer Neural Networks

Modern neural networks typically have more than one hidden layer, and often many units per layer. In theory a single hidden layer with a large number of units has the ability to approximate most functions. However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size.

We will illustrate a large dense network on the famous and publicly available MNIST handwritten digit dataset.[2] Figure 10.3 shows examples of these digits. The idea is to build a model to classify the images into their correct digit class 0–9. Every image has $p = 28 \times 28 = 784$ pixels, each of which is an eight-bit grayscale value between 0 and 255 representing the relative amount of the written digit in that tiny square.[3] These pixels are stored in the input vector X (in, say, column order). The output is the class label, represented by a vector $Y = (Y_0, Y_1, \ldots, Y_9)$ of 10 dummy variables, with a one in the position corresponding to the label, and zeros elsewhere. In the machine learning community, this is known as *one-hot encoding*. There are 60,000 training images, and 10,000 test images.

one-hot encoding

On a historical note, digit recognition problems were the catalyst that accelerated the development of neural network technology in the late 1980s at AT&T Bell Laboratories and elsewhere. Pattern recognition tasks of this

[2]See LeCun, Cortes, and Burges (2010) "The MNIST database of handwritten digits", available at http://yann.lecun.com/exdb/mnist.

[3]In the analog-to-digital conversion process, only part of the written numeral may fall in the square representing a particular pixel.

kind are relatively simple for humans. Our visual system occupies a large fraction of our brains, and good recognition is an evolutionary force for survival. These tasks are not so simple for machines, and it has taken more than 30 years to refine the neural-network architectures to match human performance.

Figure 10.4 shows a multilayer network architecture that works well for solving the digit-classification task. It differs from Figure 10.1 in several ways:

- It has two hidden layers L_1 (256 units) and L_2 (128 units) rather than one. Later we will see a network with seven hidden layers.
- It has ten output variables, rather than one. In this case the ten variables really represent a single qualitative variable and so are quite dependent. (We have indexed them by the digit class 0–9 rather than 1–10, for clarity.) More generally, in *multi-task learning* one can predict different responses simultaneously with a single network; they all have a say in the formation of the hidden layers.

multi-task learning

- The loss function used for training the network is tailored for the multiclass classification task.

The first hidden layer is as in (10.2), with

$$\begin{aligned} A_k^{(1)} &= h_k^{(1)}(X) \\ &= g(w_{k0}^{(1)} + \textstyle\sum_{j=1}^{p} w_{kj}^{(1)} X_j) \end{aligned} \tag{10.10}$$

for $k = 1, \ldots, K_1$. The second hidden layer treats the activations $A_k^{(1)}$ of the first hidden layer as inputs and computes new activations

$$\begin{aligned} A_\ell^{(2)} &= h_\ell^{(2)}(X) \\ &= g(w_{\ell 0}^{(2)} + \textstyle\sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)}) \end{aligned} \tag{10.11}$$

for $\ell = 1, \ldots, K_2$. Notice that each of the activations in the second layer $A_\ell^{(2)} = h_\ell^{(2)}(X)$ is a function of the input vector X. This is the case because while they are explicitly a function of the activations $A_k^{(1)}$ from layer L_1, these in turn are functions of X. This would also be the case with more hidden layers. Thus, through a chain of transformations, the network is able to build up fairly complex transformations of X that ultimately feed into the output layer as features.

We have introduced additional superscript notation such as $h_\ell^{(2)}(X)$ and $w_{\ell j}^{(2)}$ in (10.10) and (10.11) to indicate to which layer the activations and *weights* (coefficients) belong, in this case layer 2. The notation $\mathbf{W}_1$ in Figure 10.4 represents the entire matrix of weights that feed from the input layer to the first hidden layer L_1. This matrix will have $785 \times 256 = 200{,}960$

weights

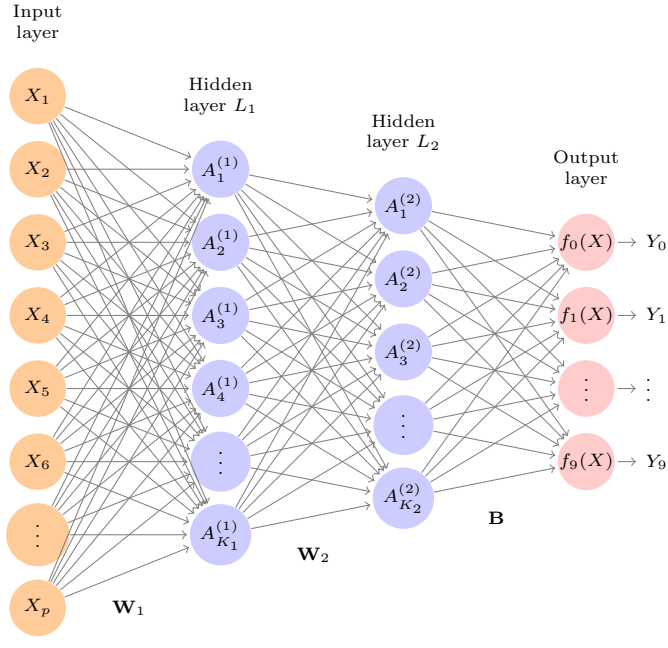

FIGURE 10.4. *Neural network diagram with two hidden layers and multiple outputs, suitable for the* MNIST *handwritten-digit problem. The input layer has* $p = 784$ *units, the two hidden layers* $K_1 = 256$ *and* $K_2 = 128$ *units respectively, and the output layer* 10 *units. Along with intercepts (referred to as* biases *in the deep-learning community) this network has 235,146 parameters (referred to as weights).*

elements; there are 785 rather than 784 because we must account for the intercept or *bias* term.[4]

bias

Each element $A_k^{(1)}$ feeds to the second hidden layer L_2 via the matrix of weights $\mathbf{W}_2$ of dimension $257 \times 128 = 32{,}896$.

We now get to the output layer, where we now have ten responses rather than one. The first step is to compute ten different linear models similar to our single model (10.1),

$$\begin{aligned} Z_m &= \beta_{m0} + \textstyle\sum_{\ell=1}^{K_2} \beta_{m\ell} h_\ell^{(2)}(X) \\ &= \beta_{m0} + \textstyle\sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}, \end{aligned} \tag{10.12}$$

for $m = 0, 1, \ldots, 9$. The matrix $\mathbf{B}$ stores all $129 \times 10 = 1{,}290$ of these weights.

[4]The use of "weights" for coefficients and "bias" for the intercepts w_{k0} in (10.2) is popular in the machine learning community; this use of bias is not to be confused with the "bias-variance" usage elsewhere in this book.

Method	Test Error
Neural Network + Ridge Regularization	2.3%
Neural Network + Dropout Regularization	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

TABLE 10.1. *Test error rate on the* MNIST *data, for neural networks with two forms of regularization, as well as multinomial logistic regression and linear discriminant analysis. In this example, the extra complexity of the neural network leads to a marked improvement in test error.*

If these were all separate quantitative responses, we would simply set each $f_m(X) = Z_m$ and be done. However, we would like our estimates to represent class probabilities $f_m(X) = \Pr(Y = m|X)$, just like in multinomial logistic regression in Section 4.3.5. So we use the special *softmax* activation function (see (4.13) on page 141),

softmax

$$f_m(X) = \Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{\ell=0}^{9} e^{Z_\ell}}, \tag{10.13}$$

for $m = 0, 1, \ldots, 9$. This ensures that the 10 numbers behave like probabilities (non-negative and sum to one). Even though the goal is to build a classifier, our model actually estimates a probability for each of the 10 classes. The classifier then assigns the image to the class with the highest probability.

To train this network, since the response is qualitative, we look for coefficient estimates that minimize the negative multinomial log-likelihood

$$-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i)), \tag{10.14}$$

also known as the *cross-entropy*. This is a generalization of the criterion (4.5) for two-class logistic regression. Details on how to minimize this objective are given in Section 10.7. If the response were quantitative, we would instead minimize squared-error loss as in (10.9).

cross-entropy

Table 10.1 compares the test performance of the neural network with two simple models presented in Chapter 4 that make use of linear decision boundaries: multinomial logistic regression and linear discriminant analysis. The improvement of neural networks over both of these linear methods is dramatic: the network with dropout regularization achieves a test error rate below 2% on the 10,000 test images. (We describe dropout regularization in Section 10.7.3.) In Section 10.9.2 of the lab, we present the code for fitting this model, which runs in just over two minutes on a laptop computer.

Adding the number of coefficients in $\mathbf{W}_1$, $\mathbf{W}_2$ and $\mathbf{B}$, we get 235,146 in all, more than 33 times the number $785 \times 9 = 7{,}065$ needed for multinomial logistic regression. Recall that there are 60,000 images in the training set.

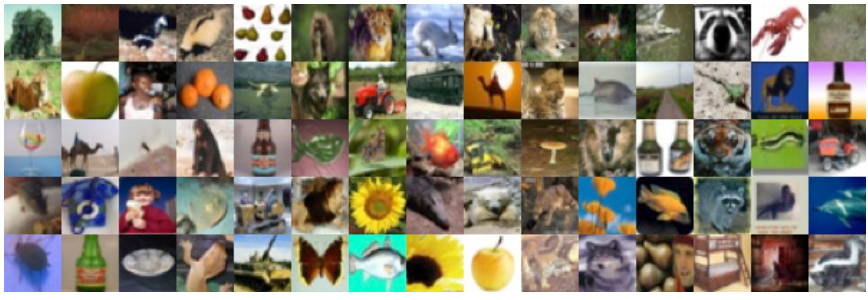

FIGURE 10.5. *A sample of images from the* CIFAR100 *database: a collection of natural images from everyday life, with 100 different classes represented.*

While this might seem like a large training set, there are almost four times as many coefficients in the neural network model as there are observations in the training set! To avoid overfitting, some regularization is needed. In this example, we used two forms of regularization: ridge regularization, which is similar to ridge regression from Chapter 6, and *dropout* regularization. We discuss both forms of regularization in Section 10.7.

dropout

10.3 Convolutional Neural Networks

Neural networks rebounded around 2010 with big successes in image classification. Around that time, massive databases of labeled images were being accumulated, with ever-increasing numbers of classes. Figure 10.5 shows 75 images drawn from the CIFAR100 database.[5] This database consists of 60,000 images labeled according to 20 superclasses (e.g. aquatic mammals), with five classes per superclass (beaver, dolphin, otter, seal, whale). Each image has a resolution of 32×32 pixels, with three eight-bit numbers per pixel representing red, green and blue. The numbers for each image are organized in a three-dimensional array called a *feature map*. The first two axes are spatial (both are 32-dimensional), and the third is the *channel* axis,[6] representing the three colors. There is a designated training set of 50,000 images, and a test set of 10,000.

feature map
channel

A special family of *convolutional neural networks* (CNNs) has evolved for classifying images such as these, and has shown spectacular success on a wide range of problems. CNNs mimic to some degree how humans classify images, by recognizing specific features or patterns anywhere in the image

convolutional neural networks

[5]See Chapter 3 of Krizhevsky (2009) "Learning multiple layers of features from tiny images", available at https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[6]The term *channel* is taken from the signal-processing literature. Each channel is a distinct source of information.

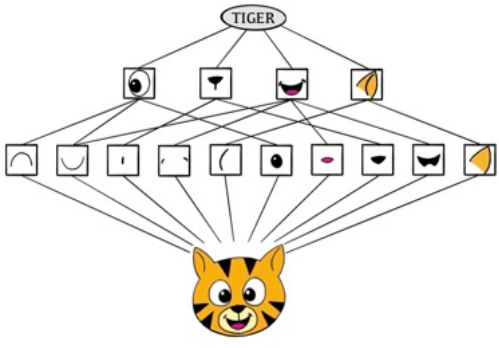

FIGURE 10.6. *Schematic showing how a convolutional neural network classifies an image of a tiger. The network takes in the image and identifies local features. It then combines the local features in order to create compound features, which in this example include eyes and ears. These compound features are used to output the label "tiger".*

that distinguish each particular object class. In this section we give a brief overview of how they work.

Figure 10.6 illustrates the idea behind a convolutional neural network on a cartoon image of a tiger.[7]

The network first identifies low-level features in the input image, such as small edges, patches of color, and the like. These low-level features are then combined to form higher-level features, such as parts of ears, eyes, and so on. Eventually, the presence or absence of these higher-level features contributes to the probability of any given output class.

How does a convolutional neural network build up this hierarchy? It combines two specialized types of hidden layers, called *convolution* layers and *pooling* layers. Convolution layers search for instances of small patterns in the image, whereas pooling layers downsample these to select a prominent subset. In order to achieve state-of-the-art results, contemporary neural-network architectures make use of many convolution and pooling layers. We describe convolution and pooling layers next.

10.3.1 Convolution Layers

A *convolution layer* is made up of a large number of *convolution filters*, each of which is a template that determines whether a particular local feature is present in an image. A convolution filter relies on a very simple operation, called a *convolution*, which basically amounts to repeatedly multiplying matrix elements and then adding the results.

convolution layer

convolution filter

[7]Thanks to Elena Tuzhilina for producing the diagram and https://www.cartooning4kids.com/ for permission to use the cartoon tiger.

To understand how a convolution filter works, consider a very simple example of a 4×3 image:

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}.$$

Now consider a 2×2 filter of the form

$$\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

When we *convolve* the image with the filter, we get the result[8]

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}.$$

For instance, the top-left element comes from multiplying each element in the 2×2 filter by the corresponding element in the top left 2×2 portion of the image, and adding the results. The other elements are obtained in a similar way: the convolution filter is applied to every 2×2 submatrix of the original image in order to obtain the convolved image. If a 2×2 submatrix of the original image resembles the convolution filter, then it will have a *large* value in the convolved image; otherwise, it will have a *small* value. Thus, *the convolved image highlights regions of the original image that resemble the convolution filter.* We have used 2×2 as an example; in general convolution filters are small $\ell_1 \times \ell_2$ arrays, with ℓ_1 and ℓ_2 small positive integers that are not necessarily equal.

Figure 10.7 illustrates the application of two convolution filters to a 192×179 image of a tiger, shown on the left-hand side.[9] Each convolution filter is a 15×15 image containing mostly zeros (black), with a narrow strip of ones (white) oriented either vertically or horizontally within the image. When each filter is convolved with the image of the tiger, areas of the tiger that resemble the filter (i.e. that have either horizontal or vertical stripes or edges) are given large values, and areas of the tiger that do not resemble the feature are given small values. The convolved images are displayed on the right-hand side. We see that the horizontal stripe filter picks out horizontal stripes and edges in the original image, whereas the vertical stripe filter picks out vertical stripes and edges in the original image.

---

[8] The convolved image is smaller than the original image because its dimension is given by the number of 2×2 submatrices in the original image. Note that 2×2 is the dimension of the convolution filter. If we want the convolved image to have the same dimension as the original image, then padding can be applied.

[9] The tiger image used in Figures 10.7–10.9 was obtained from the public domain image resource https://www.needpix.com/.

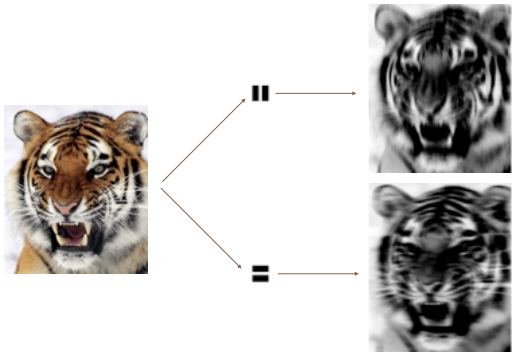

FIGURE 10.7. *Convolution filters find local features in an image, such as edges and small shapes. We begin with the image of the tiger shown on the left, and apply the two small convolution filters in the middle. The convolved images highlight areas in the original image where details similar to the filters are found. Specifically, the top convolved image highlights the tiger's vertical stripes, whereas the bottom convolved image highlights the tiger's horizontal stripes. We can think of the original image as the input layer in a convolutional neural network, and the convolved images as the units in the first hidden layer.*

We have used a large image and two large filters in Figure 10.7 for illustration. For the `CIFAR100` database there are 32×32 color pixels per image, and we use 3×3 convolution filters.

In a convolution layer, we use a whole bank of filters to pick out a variety of differently-oriented edges and shapes in the image. Using predefined filters in this way is standard practice in image processing. By contrast, with CNNs the filters are *learned* for the specific classification task. We can think of the filter weights as the parameters going from an input layer to a hidden layer, with one hidden unit for each pixel in the convolved image. This is in fact the case, though the parameters are highly structured and constrained (see Exercise 4 for more details). They operate on localized patches in the input image (so there are many structural zeros), and the same weights in a given filter are reused for all possible patches in the image (so the weights are constrained).[10]

We now give some additional details.

- Since the input image is in color, it has three channels represented by a three-dimensional feature map (array). Each channel is a two-dimensional (32×32) feature map — one for red, one for green, and one for blue. A single convolution filter will also have three channels, one per color, each of dimension 3×3, with potentially different filter weights. The results of the three convolutions are summed to form

[10]This used to be called *weight sharing* in the early years of neural networks.

a two-dimensional output feature map. Note that at this point the color information has been used, and is not passed on to subsequent layers except through its role in the convolution.

- If we use K different convolution filters at this first hidden layer, we get K two-dimensional output feature maps, which together are treated as a single three-dimensional feature map. We view each of the K output feature maps as a separate channel of information, so now we have K channels in contrast to the three color channels of the original input feature map. The three-dimensional feature map is just like the activations in a hidden layer of a simple neural network, except organized and produced in a spatially structured way.
- We typically apply the ReLU activation function (10.5) to the convolved image. This step is sometimes viewed as a separate layer in the convolutional neural network, in which case it is referred to as a *detector layer*.

detector layer

10.3.2 Pooling Layers

A *pooling* layer provides a way to condense a large image into a smaller summary image. While there are a number of possible ways to perform pooling, the *max pooling* operation summarizes each non-overlapping 2×2 block of pixels in an image using the maximum value in the block. This reduces the size of the image by a factor of two in each direction, and it also provides some *location invariance*: i.e. as long as there is a large value in one of the four pixels in the block, the whole block registers as a large value in the reduced image.

pooling

Here is a simple example of max pooling:

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}.$$

10.3.3 Architecture of a Convolutional Neural Network

So far we have defined a single convolution layer — each filter produces a new two-dimensional feature map. The number of convolution filters in a convolution layer is akin to the number of units at a particular hidden layer in a fully-connected neural network of the type we saw in Section 10.2. This number also defines the number of channels in the resulting three-dimensional feature map. We have also described a pooling layer, which reduces the first two dimensions of each three-dimensional feature map. Deep CNNs have many such layers. Figure 10.8 shows a typical architecture for a CNN for the `CIFAR100` image classification task.

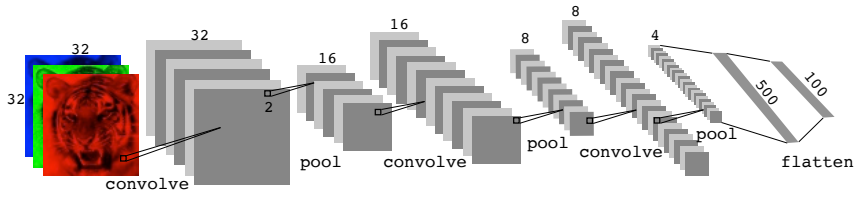

FIGURE 10.8. *Architecture of a deep CNN for the* `CIFAR100` *classification task. Convolution layers are interspersed with* 2×2 *max-pool layers, which reduce the size by a factor of 2 in both dimensions.*

At the input layer, we see the three-dimensional feature map of a color image, where the channel axis represents each color by a 32×32 two-dimensional feature map of pixels. Each convolution filter produces a new channel at the first hidden layer, each of which is a 32×32 feature map (after some padding at the edges). After this first round of convolutions, we now have a new "image"; a feature map with considerably more channels than the three color input channels (six in the figure, since we used six convolution filters).

This is followed by a max-pool layer, which reduces the size of the feature map in each channel by a factor of four: two in each dimension.

This convolve-then-pool sequence is now repeated for the next two layers. Some details are as follows:

- Each subsequent convolve layer is similar to the first. It takes as input the three-dimensional feature map from the previous layer and treats it like a single multi-channel image. Each convolution filter learned has as many channels as this feature map.
- Since the channel feature maps are reduced in size after each pool layer, we usually increase the number of filters in the next convolve layer to compensate.
- Sometimes we repeat several convolve layers before a pool layer. This effectively increases the dimension of the filter.

These operations are repeated until the pooling has reduced each channel feature map down to just a few pixels in each dimension. At this point the three-dimensional feature maps are *flattened* — the pixels are treated as separate units — and fed into one or more fully-connected layers before reaching the output layer, which is a *softmax activation* for the 100 classes (as in (10.13)).

There are many tuning parameters to be selected in constructing such a network, apart from the number, nature, and sizes of each layer. Dropout learning can be used at each layer, as well as lasso or ridge regularization (see Section 10.7). The details of constructing a convolutional neural network can seem daunting. Fortunately, terrific software is available, with

FIGURE 10.9. *Data augmentation. The original image (leftmost) is distorted in natural ways to produce different images with the same class label. These distortions do not fool humans, and act as a form of regularization when fitting the CNN.*

extensive examples and vignettes that provide guidance on sensible choices for the parameters. For the `CIFAR100` official test set, the best accuracy as of this writing is just above 75%, but undoubtedly this performance will continue to improve.

10.3.4 Data Augmentation

An additional important trick used with image modeling is *data augmentation*. Essentially, each training image is replicated many times, with each replicate randomly distorted in a natural way such that human recognition is unaffected. Figure 10.9 shows some examples. Typical distortions are zoom, horizontal and vertical shift, shear, small rotations, and in this case horizontal flips. At face value this is a way of increasing the training set considerably with somewhat different examples, and thus protects against overfitting. In fact we can see this as a form of regularization: we build a cloud of images around each original image, all with the same label. This kind of fattening of the data is similar in spirit to ridge regularization.

data augmentation

We will see in Section 10.7.2 that the stochastic gradient descent algorithms for fitting deep learning models repeatedly process randomly-selected batches of, say, 128 training images at a time. This works hand-in-glove with augmentation, because we can distort each image in the batch on the fly, and hence do not have to store all the new images.

10.3.5 Results Using a Pretrained Classifier

Here we use an industry-level pretrained classifier to predict the class of some new images. The `resnet50` classifier is a convolutional neural network that was trained using the `imagenet` data set, which consists of millions of images that belong to an ever-growing number of categories.[11] Figure 10.10

[11] For more information about `resnet50`, see He, Zhang, Ren, and Sun (2015) "Deep residual learning for image recognition", https://arxiv.org/abs/1512.03385. For details about `imagenet`, see Russakovsky, Deng, et al. (2015) "ImageNet Large Scale Visual Recognition Challenge", in *International Journal of Computer Vision*.

flamingo		Cooper's hawk		Cooper's hawk	
flamingo	0.83	kite	0.60	fountain	0.35
spoonbill	0.17	great grey owl	0.09	nail	0.12
white stork	0.00	robin	0.06	hook	0.07
Lhasa Apso		cat		Cape weaver	
Tibetan terrier	0.56	Old English sheepdog	0.82	jacamar	0.28
Lhasa	0.32	Shih-Tzu	0.04	macaw	0.12
cocker spaniel	0.03	Persian cat	0.04	robin	0.12

FIGURE 10.10. *Classification of six photographs using the* `resnet50` *CNN trained on the* `imagenet` *corpus. The table below the images displays the true (intended) label at the top of each panel, and the top three choices of the classifier (out of 100). The numbers are the estimated probabilities for each choice. (A kite is a raptor, but not a hawk.)*

demonstrates the performance of `resnet50` on six photographs (private collection of one of the authors).[12] The CNN does a reasonable job classifying the hawk in the second image. If we zoom out as in the third image, it gets confused and chooses the fountain rather than the hawk. In the final image a "jacamar" is a tropical bird from South and Central America with similar coloring to the South African Cape Weaver. We give more details on this example in Section 10.9.4.

Much of the work in fitting a CNN is in learning the convolution filters at the hidden layers; these are the coefficients of a CNN. For models fit to massive corpora such as `imagenet` with many classes, the output of these

[12]These `resnet` results can change with time, since the publicly-trained model gets updated periodically.

filters can serve as features for general natural-image classification problems. One can use these pretrained hidden layers for new problems with much smaller training sets (a process referred to as *weight freezing*), and just train the last few layers of the network, which requires much less data. The vignettes and book[13] that accompany the `keras` package give more details on such applications.

weight freezing

10.4 Document Classification

In this section we introduce a new type of example that has important applications in industry and science: predicting attributes of documents. Examples of documents include articles in medical journals, Reuters news feeds, emails, tweets, and so on. Our example will be `IMDb` (Internet Movie Database) ratings — short documents where viewers have written critiques of movies.[14] The response in this case is the `sentiment` of the review, which will be *positive* or *negative*.

Here is the beginning of a rather amusing negative review:

> *This has to be one of the worst films of the 1990s. When my friends & I were watching this film (being the target audience it was aimed at) we just sat & watched the first half an hour with our jaws touching the floor at how bad it really was. The rest of the time, everyone else in the theater just started talking to each other, leaving or generally crying into their popcorn . . .*

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. This is modern parlance for defining a set of predictors.

featurize

The simplest and most common featurization is the *bag-of-words* model. We score each document for the presence or absence of each of the words in a language dictionary — in this case an English dictionary. If the dictionary contains M words, that means for each document we create a binary feature vector of length M, and score a 1 for every word present, and 0 otherwise. That can be a very wide feature vector, so we limit the dictionary — in this case to the 10,000 most frequently occurring words in the training corpus of 25,000 reviews. Fortunately there are nice tools for doing this automatically. Here is the beginning of a positive review that has been redacted in this way:

bag-of-words

> *⟨START⟩ this film was just brilliant casting location scenery story direction everyone's really suited the part they played and*

[13] *Deep Learning with R* by F. Chollet and J.J. Allaire, 2018, Manning Publications.

[14] For details, see Maas et al. (2011) "Learning word vectors for sentiment analysis", in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150.

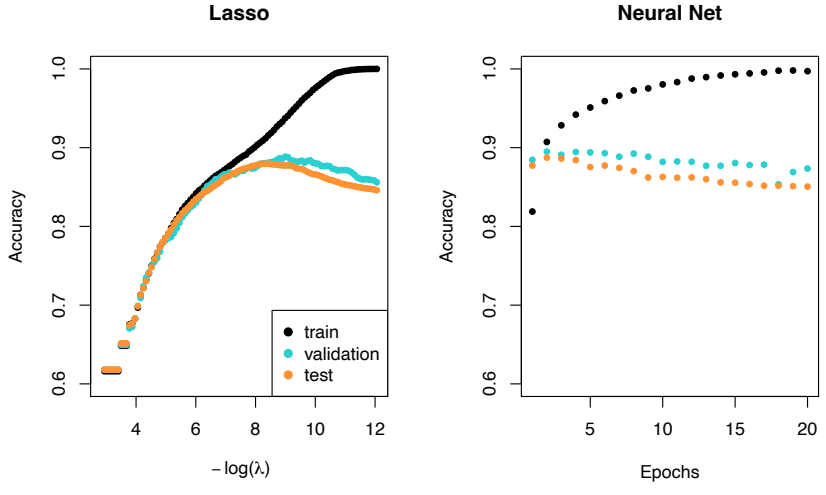

FIGURE 10.11. *Accuracy of the lasso and a two-hidden-layer neural network on the* `IMDb` *data. For the lasso, the x-axis displays* $-\log(\lambda)$*, while for the neural network it displays epochs (number of times the fitting algorithm passes through the training set). Both show a tendency to overfit,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

> *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 . . .*

Here we can see many words have been omitted, and some unknown words (UNK) have been marked as such. With this reduction the binary feature vector has length 10,000, and consists mostly of 0's and a smattering of 1's in the positions corresponding to words that are present in the document. We have a training set and test set, each with 25,000 examples, and each balanced with regard to `sentiment`. The resulting training feature matrix $\mathbf{X}$ has dimension $25{,}000 \times 10{,}000$, but only 1.3% of the binary entries are non-zero. We call such a matrix sparse, because most of the values are the same (zero in this case); it can be stored efficiently in *sparse matrix format*.[15] There are a variety of ways to account for the document length; here we only score a word as in or out of the document, but for example one could instead record the relative frequency of words. We split off a validation set of size 2,000 from the 25,000 training observations (for model tuning), and fit two model sequences:

sparse matrix format

[15]Rather than store the whole matrix, we can store instead the location and values for the nonzero entries. In this case, since the nonzero entries are all 1, just the locations are stored.

- A lasso logistic regression using the `glmnet` package;
- A two-class neural network with two hidden layers, each with 16 ReLU units.

Both methods produce a sequence of solutions. The lasso sequence is indexed by the regularization parameter λ. The neural-net sequence is indexed by the number of gradient-descent iterations used in the fitting, as measured by training epochs or passes through the training set (Section 10.7). Notice that the training accuracy in Figure 10.11 (black points) increases monotonically in both cases. We can use the validation error to pick a good solution from each sequence (blue points in the plots), which would then be used to make predictions on the test data set.

Note that a two-class neural network amounts to a nonlinear logistic regression model. From (10.12) and (10.13) we can see that

$$\begin{aligned} \log\left(\frac{\Pr(Y=1|X)}{\Pr(Y=0|X)}\right) &= Z_1 - Z_0 \qquad (10.15) \\ &= (\beta_{10}-\beta_{00}) + \sum_{\ell=1}^{K_2}(\beta_{1\ell}-\beta_{0\ell})A_\ell^{(2)}. \end{aligned}$$

(This shows the redundancy in the softmax function; for K classes we really only need to estimate $K-1$ sets of coefficients. See Section 4.3.5.) In Figure 10.11 we show *accuracy* (fraction correct) rather than classification error (fraction incorrect), the former being more popular in the machine learning community. Both models achieve a test-set accuracy of about 88%.

accuracy

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.

bag-of-*n*-grams

- Treat the document as a sequence, taking account of all the words in the context of those that preceded and those that follow.

In the next section we explore models for sequences of data, which have applications in weather forecasting, speech recognition, language translation, and time-series prediction, to name a few. We continue with this `IMDb` example there.

10.5 Recurrent Neural Networks

Many data sources are sequential in nature, and call for special treatment when building predictive models. Examples include:

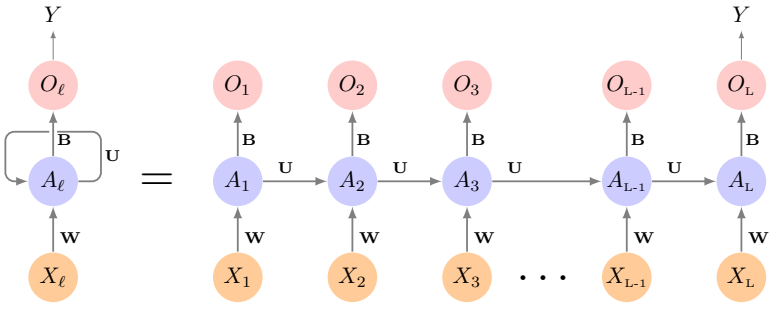

FIGURE 10.12. *Schematic of a simple recurrent neural network. The input is a sequence of vectors* $\{X_\ell\}_1^L$*, and here the target is a single response. The network processes the input sequence* X *sequentially; each* X_ℓ *feeds into the hidden layer, which also has as input the activation vector* $A_{\ell-1}$ *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.*

- Documents such as book and movie reviews, newspaper articles, and tweets. The sequence and relative positions of words in a document capture the narrative, theme and tone, and can be exploited in tasks such as topic classification, sentiment analysis, and language translation.
- Time series of temperature, rainfall, wind speed, air quality, and so on. We may want to forecast the weather several days ahead, or climate several decades ahead.
- Financial time series, where we track market indices, trading volumes, stock and bond prices, and exchange rates. Here prediction is often difficult, but as we will see, certain indices can be predicted with reasonable accuracy.
- Recorded speech, musical recordings, and other sound recordings. We may want to give a text transcription of a speech, or perhaps a language translation. We may want to assess the quality of a piece of music, or assign certain attributes.
- Handwriting, such as doctor's notes, and handwritten digits such as zip codes. Here we want to turn the handwriting into digital text, or read the digits (optical character recognition).

In a *recurrent neural network* (RNN), the input object X is a *sequence*.

recurrent neural network

Consider a corpus of documents, such as the collection of IMDb movie reviews. Each document can be represented as a sequence of L words, so $X = \{X_1, X_2, \ldots, X_L\}$, where each X_ℓ represents a word. 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, much like convolutional neural networks accommodate the spatial structure of image inputs. The output Y can also be a sequence (such as in language translation), but often is a scalar, like the binary sentiment label of a movie review document.

Figure 10.12 illustrates the structure of a very basic RNN with a sequence $X = \{X_1, X_2, \ldots, X_L\}$ as input, a simple output Y, and a hidden-layer sequence $\{A_\ell\}_1^L = \{A_1, A_2, \ldots, A_L\}$. Each X_ℓ is a vector; in the document example X_ℓ could represent a one-hot encoding for the ℓth word based on the language dictionary for the corpus (see the top panel in Figure 10.13 for a simple example). 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.

In detail, suppose each vector X_ℓ of the input sequence has p components $X_\ell^T = (X_{\ell 1}, X_{\ell 2}, \ldots, X_{\ell p})$, and the hidden layer consists of K units $A_\ell^T = (A_{\ell 1}, A_{\ell 2}, \ldots, A_{\ell K})$. As in Figure 10.4, we represent the collection of $K \times (p+1)$ shared weights w_{kj} for the input layer by a matrix $\mathbf{W}$, and similarly $\mathbf{U}$ is a $K \times K$ matrix of the weights u_{ks} for the hidden-to-hidden layers, and $\mathbf{B}$ is a $K+1$ vector of weights β_k for the output layer. Then

$$A_{\ell k} = g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\Big), \tag{10.16}$$

and the output O_ℓ is computed as

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k} \tag{10.17}$$

for a quantitative response, or with an additional sigmoid activation function for a binary response, for example. Here $g(\cdot)$ is an activation function such as ReLU. Notice that the same weights $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ are used as we process each element in the sequence, i.e. they are not functions of ℓ. This is a form of *weight sharing* used by RNNs, and similar to the use of filters in convolutional neural networks (Section 10.3.1.) As we proceed from beginning to end, the activations A_ℓ accumulate a history of what has been seen before, so that the learned context can be used for prediction.

weight sharing

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

$$(Y - O_L)^2, \tag{10.18}$$

which only references the final output $O_L = \beta_0 + \sum_{k=1}^{K} \beta_k A_{Lk}$. Thus $O_1, O_2, \ldots, O_{L-1}$ are not used. When we fit the model, each element X_ℓ of the input sequence X contributes to O_L via the chain (10.16), and hence contributes indirectly to learning the shared parameters $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ via the loss (10.18). With n input sequence/response pairs (x_i, y_i), the parameters are found by minimizing the sum of squares

$$\sum_{i=1}^{n}(y_i - o_{iL})^2 = \sum_{i=1}^{n}\Big(y_i - \big(\beta_0 + \sum_{k=1}^{K}\beta_k g\big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{iLj} + \sum_{s=1}^{K} u_{ks}a_{i,L-1,s}\big)\big)\Big)^2. \tag{10.19}$$

Here we use lowercase letters for the observed y_i and vector sequences $x_i = \{x_{i1}, x_{i2}, \ldots, x_{iL}\}$,[16] as well as the derived activations.

Since the intermediate outputs O_ℓ are not used, one may well ask why they are there at all. First of all, they come for free, since they use the same output weights $\mathbf{B}$ needed to produce O_L, and provide an evolving prediction for the output. Furthermore, for some learning tasks the response is also a sequence, and so the output sequence $\{O_1, O_2, \ldots, O_L\}$ is explicitly needed.

When used at full strength, recurrent neural networks can be quite complex. We illustrate their use in two simple applications. In the first, we continue with the IMDb sentiment analysis of the previous section, where we process the words in the reviews sequentially. In the second application, we illustrate their use in a financial time series forecasting problem.

10.5.1 Sequential Models for Document Classification

Here we return to our classification task with the IMDb reviews. Our approach in Section 10.4 was to use the bag-of-words model. Here the plan is to use instead the sequence of words occurring in a document to make predictions about the label for the entire document.

We have, however, a dimensionality problem: each word in our document is represented by a one-hot-encoded vector (dummy variable) with 10,000 elements (one per word in the dictionary)! An approach that has become popular is to represent each word in a much lower-dimensional *embedding* space. This means that rather than representing each word by a binary vector with 9,999 zeros and a single one in some position, we will represent it instead by a set of m real numbers, none of which are typically zero. Here m is the embedding dimension, and can be in the low 100s, or even less. This means (in our case) that we need a matrix $\mathbf{E}$ of dimension $m \times 10{,}000$, where each column is indexed by one of the 10,000 words in our dictionary, and the values in that column give the m coordinates for that word in the embedding space.

embedding

[16] This is a sequence of vectors; each element $x_{i\ell}$ is a p-vector.

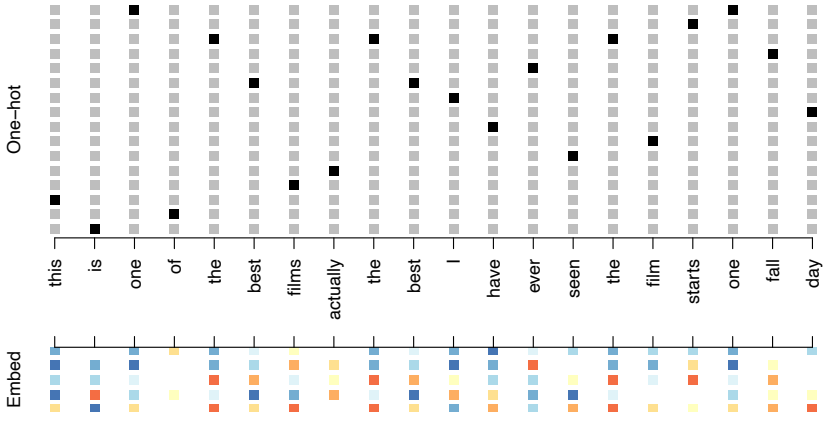

FIGURE 10.13. *Depiction of a sequence of* 20 *words representing a single document: one-hot encoded using a dictionary of* 16 *words (top panel) and embedded in an* m*-dimensional space with* $m = 5$ *(bottom panel).*

Figure 10.13 illustrates the idea (with a dictionary of 16 rather than 10,000, and $m = 5$). Where does $\mathbf{E}$ come from? If we have a large corpus of labeled documents, we can have the neural network *learn* $\mathbf{E}$ as part of the optimization. In this case $\mathbf{E}$ is referred to as an *embedding layer*, and a specialized $\mathbf{E}$ is learned for the task at hand. 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.[17] These are built from a very large corpus of documents by a variant of principal components analysis (Section 12.2). The idea is that the positions of words in the embedding space preserve semantic meaning; e.g. synonyms should appear near each other.

embedding layer

weight freezing

`word2vec`

`GloVe`

So far, so good. Each document is now represented as a sequence of m-vectors that represents the sequence of words. The next step is to limit each document to the last L words. Documents that are shorter than L get padded 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 in Figure 10.12. The training corpus consists of n separate series (documents) of length L, each of which gets processed sequentially from left to right. In the process, a parallel series of hidden activation vectors A_ℓ, $\ell = 1, \ldots, L$ is created as in (10.16) for each document. A_ℓ feeds into the output layer to produce the evolving prediction

[17] `word2vec` is described in Mikolov, Chen, Corrado, and Dean (2013), available at https://code.google.com/archive/p/word2vec. `GloVe` is described in Pennington, Socher, and Manning (2014), available at https://nlp.stanford.edu/projects/glove.

O_ℓ. We use the final value O_L to predict the response: the sentiment of the review.

This is a simple RNN, and has relatively few parameters. If there are K hidden units, the common weight matrix $\mathbf{W}$ has $K \times (m+1)$ parameters, the matrix $\mathbf{U}$ has $K \times K$ parameters, and $\mathbf{B}$ has $2(K+1)$ for the two-class logistic regression as in (10.15). These are used repeatedly as we process the sequence $X = \{X_\ell\}_1^L$ from left to right, much like we use a single convolution filter to process each patch in an image (Section 10.3.1). If the embedding layer $\mathbf{E}$ is learned, that adds an additional $m \times D$ parameters ($D = 10{,}000$ here), and is by far the biggest cost.

We fit the RNN as described in Figure 10.12 and the accompaying text to the IMDb data. The model had an embedding matrix $\mathbf{E}$ with $m = 32$ (which was learned in training as opposed to precomputed), followed by a single recurrent layer with $K = 32$ hidden units. The model was trained with dropout regularization on the 25,000 reviews in the designated training set, and achieved a disappointing 76% accuracy on the IMDb test data. A network using the GloVe pretrained embedding matrix $\mathbf{E}$ performed slightly worse.

For ease of exposition we have presented a very simple RNN. More elaborate versions 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 — a so-called *LSTM RNN*. 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 RNN

When we refit our model using the LSTM architecture for the hidden layer, the performance improved to 87% on the IMDb test data. This is comparable with the 88% achieved by the bag-of-words model in Section 10.4. We give details on fitting these models in Section 10.9.6.

Despite this added LSTM complexity, our RNN is still somewhat "entry level". We could probably achieve slightly better results by changing the size of the model, changing the regularization, and including additional hidden layers. However, LSTM models take a long time to train, which makes exploring many architectures and parameter optimization tedious.

RNNs provide a rich framework for modeling data sequences, and they continue to evolve. There have been many advances in the development of RNNs — in architecture, data augmentation, and in the learning algorithms. At the time of this writing (early 2020) the leading RNN configurations report accuracy above 95% on the IMDb data. The details are beyond the scope of this book.[18]

[18] An IMDb leaderboard can be found at https://paperswithcode.com/sota/sentiment-analysis-on-imdb.

10.5.2 Time Series Forecasting

Figure 10.14 shows historical trading statistics from the New York Stock Exchange. Shown are three daily time series covering the period December 3, 1962 to December 31, 1986:[19]

- `Log trading volume`. This is the fraction of all outstanding shares that are traded on that day, relative to a 100-day moving average of past turnover, on the log scale.
- `Dow Jones return`. This is the difference between the log of the Dow Jones Industrial Index on consecutive trading days.
- `Log volatility`. This is based on the absolute values of daily price movements.

Predicting stock prices is a notoriously hard problem, but it turns out that predicting trading volume based on recent past history is more manageable (and is useful for planning trading strategies).

An observation here consists of the measurements (v_t, r_t, z_t) on day t, in this case the values for `log_volume`, `DJ_return` and `log_volatility`. There are a total of $T = 6{,}051$ such triples, each of which is plotted as a time series in Figure 10.14. One feature that strikes us immediately is that the day-to-day observations are not independent of each other. The series exhibit *auto-correlation* — in this case values nearby in time tend to be similar to each other. This distinguishes time series from other data sets we have encountered, in which observations can be assumed to be independent of each other. To be clear, consider pairs of observations $(v_t, v_{t-\ell})$, a *lag* of ℓ days apart. If we take all such pairs in the v_t series and compute their correlation coefficient, this gives the autocorrelation at lag ℓ. Figure 10.15 shows the autocorrelation function for all lags up to 37, and we see considerable correlation.

auto-correlation

lag

Another interesting characteristic of this forecasting problem is that the response variable v_t — `log_volume` — is also a predictor! In particular, we will use the past values of `log_volume` to predict values in the future.

RNN forecaster

We wish to predict a value v_t from past values $v_{t-1}, v_{t-2}, \ldots$, and also to make use of past values of the other series $r_{t-1}, r_{t-2}, \ldots$ and $z_{t-1}, z_{t-2}, \ldots$. Although our combined data is quite a long series with 6,051 trading days, the structure of the problem is different from the previous document-classification example.

- We only have one series of data, not 25,000.

[19] These data were assembled by LeBaron and Weigend (1998) *IEEE Transactions on Neural Networks*, 9(1): 213–220.

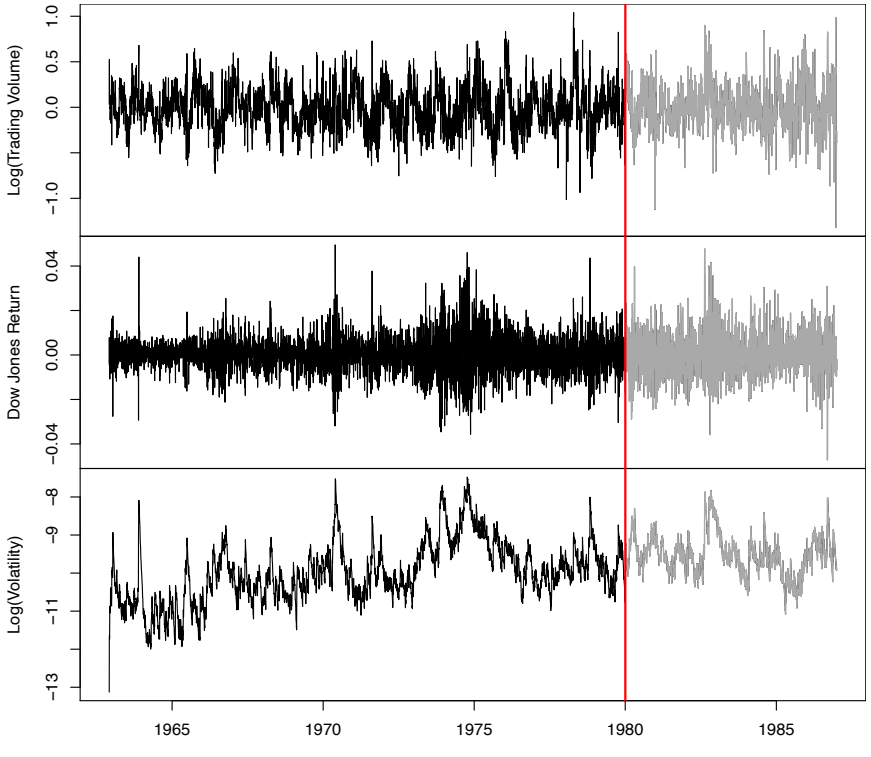

FIGURE 10.14. *Historical trading statistics from the New York Stock Exchange. Daily values of the normalized log trading volume, DJIA return, and log volatility are shown for a 24-year period from 1962–1986. We wish to predict trading volume on any day, given the history on all earlier days. To the left of the red bar (January 2, 1980) is training data, and to the right test data.*

- We have an entire *series* of targets v_t, and the inputs include past values of this series.

How do we represent this problem in terms of the structure displayed in Figure 10.12? The idea is to extract many short mini-series of input sequences $X = \{X_1, X_2, \ldots, X_L\}$ with a predefined length L (called the *lag* in this context), and a corresponding target Y. They have the form

lag

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, \; X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \cdots, X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \quad \text{and } Y = v_t. \tag{10.20}$$

So here the target Y is the value of `log_volume` v_t at a single timepoint t, and the input sequence X is the series of 3-vectors $\{X_\ell\}_1^L$ each consisting of the three measurements `log_volume`, `DJ_return` and `log_volatility` from day $t-L$, $t-L+1$, up to $t-1$. Each value of t makes a separate (X, Y)

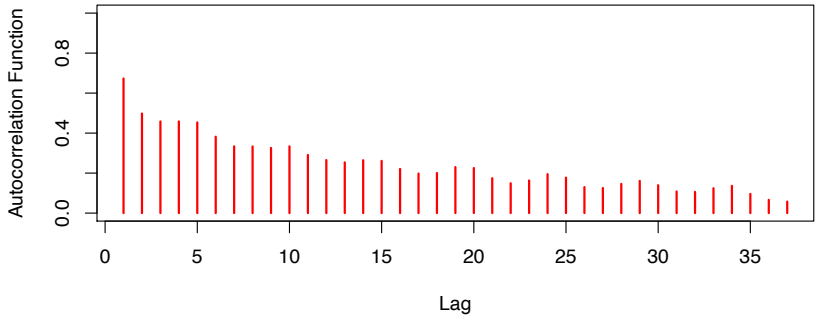

FIGURE 10.15. *The autocorrelation function for* `log_volume`*. We see that nearby values are fairly strongly correlated, with correlations above* 0.2 *as far as 20 days apart.*

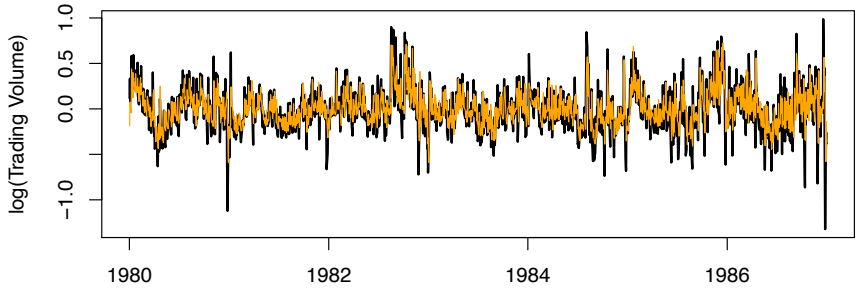

FIGURE 10.16. *RNN forecast of* `log_volume` *on the* `NYSE` *test data. The black lines are the true volumes, and the superimposed orange the forecasts. The forecasted series accounts for 42% of the variance of* `log_volume`*.*

pair, for t running from $L+1$ to T. For the `NYSE` data we will use the past five trading days to predict the next day's trading volume. Hence, we use $L = 5$. Since $T = 6{,}051$, we can create 6,046 such (X, Y) pairs. Clearly L is a parameter that should be chosen with care, perhaps using validation data.

We fit this model with $K = 12$ hidden units using the 4,281 training sequences derived from the data before January 2, 1980 (see Figure 10.14), and then used it to forecast the 1,770 values of `log_volume` after this date. We achieve an $R^2 = 0.42$ on the test data. Details are given in Section 10.9.6. As a *straw man*,[20] using yesterday's value for `log_volume` as the prediction for today has $R^2 = 0.18$. Figure 10.16 shows the forecast results. We have plotted the observed values of the daily `log_volume` for the

[20] A straw man here refers to a simple and sensible prediction that can be used as a baseline for comparison.

test period 1980–1986 in black, and superimposed the predicted series in orange. The correspondence seems rather good.

In forecasting the value of `log_volume` in the test period, we have to use the test data itself in forming the input sequences X. This may feel like cheating, but in fact it is not; we are always using past data to predict the future.

Autoregression

The RNN we just fit has much in common with a traditional *autoregression* (AR) linear model, which we present now for comparison. We first consider the response sequence v_t alone, and construct a response vector $\mathbf{y}$ and a matrix $\mathbf{M}$ of predictors for least squares regression as follows:

autoregression

$$\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ v_{L+3} \\ \vdots \\ v_T \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L} \end{bmatrix}. \tag{10.21}$$

$\mathbf{M}$ and $\mathbf{y}$ each have $T - L$ rows, one per observation. We see that the predictors for any given response v_t on day t are the previous L values of the same series. Fitting a regression of $\mathbf{y}$ on $\mathbf{M}$ amounts to fitting the model

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \hat{\beta}_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}, \tag{10.22}$$

and is called an order-L autoregressive model, or simply AR(L). For the `NYSE` data we can include lagged versions of `DJ_return` and `log_volatility`, r_t and z_t, in the predictor matrix $\mathbf{M}$, resulting in $3L + 1$ columns. An AR model with $L = 5$ achieves a test R^2 of 0.41, slightly inferior to the 0.42 achieved by the RNN.

Of course the RNN and AR models are very similar. They both use the same response Y and input sequences X of length $L = 5$ and dimension $p = 3$ in this case. The RNN processes this sequence from left to right with the same weights $\mathbf{W}$ (for the input layer), while the AR model simply treats all L elements of the sequence equally as a vector of $L \times p$ predictors — a process called *flattening* in the neural network literature. Of course the RNN also includes the hidden layer activations A_ℓ which transfer information along the sequence, and introduces additional nonlinearity. From (10.19) with $K = 12$ hidden units, we see that the RNN has $13 + 12 \times (1 + 3 + 12) = 205$ parameters, compared to the 16 for the AR(5) model.

flattening

An obvious extension of the AR model is to use the set of lagged predictors as the input vector to an ordinary feedforward neural network (10.1), and hence add more flexibility. This achieved a test $R^2 = 0.42$, slightly better than the linear AR, and the same as the RNN.

All the models can be improved by including the variable `day_of_week` corresponding to the day t of the target v_t (which can be learned from the calendar dates supplied with the data); trading volume is often higher on Mondays and Fridays. Since there are five trading days, this one-hot encodes to five binary variables. The performance of the AR model improved to $R^2 = 0.46$ as did the RNN, and the nonlinear AR model improved to $R^2 = 0.47$.

We used the most simple version of the RNN in our examples here. Additional experiments with the LSTM extension of the RNN yielded small improvements, typically of up to 1% in R^2 in these examples.

We give details of how we fit all three models in Section 10.9.6.

10.5.3 Summary of RNNs

We have illustrated RNNs through two simple use cases, and have only scratched the surface.

There are many variations and enhancements of the simple RNN we used for sequence modeling. One approach we did not discuss uses a one-dimensional convolutional neural network, treating the sequence of vectors (say words, as represented in the embedding space) 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 in an RNN. For example, with two hidden layers, the sequence A_ℓ is treated as an input sequence to the next hidden layer in an obvious fashion.

The RNN we used scanned the document from beginning to end; alternative *bidirectional* RNNs scan the sequences in both directions.

bidirectional

In language translation the target is also a sequence of words, in a language different from that of the input sequence. Both the input sequence and the target sequence are represented by a structure similar to Figure 10.12, and they share the hidden units. In this so-called *Seq2Seq* learning, the hidden units are thought to capture the semantic meaning of the sentences. Some of the big breakthroughs in language modeling and translation resulted from the relatively recent improvements in such RNNs.

Seq2Seq

Algorithms used to fit RNNs can be complex and computationally costly. Fortunately, good software protects users somewhat from these complexities, and makes specifying and fitting these models relatively painless. Many of the models that we enjoy in daily life (like *Google Translate*) use state-of-the-art architectures developed by teams of highly skilled engineers, and have been trained using massive computational and data resources.

10.6 When to Use Deep Learning

The performance of deep learning in this chapter has been rather impressive. It nailed the digit classification problem, and deep CNNs have really revolutionized image classification. We see daily reports of new success stories for deep learning. Many of these are related to image classification tasks, such as machine diagnosis of mammograms or digital X-ray images, ophthalmology eye scans, annotations of MRI scans, and so on. Likewise there are numerous successes of RNNs in speech and language translation, forecasting, and document modeling. The question that then begs an answer is: *should we discard all our older tools, and use deep learning on every problem with data?* To address this question, we revisit our `Hitters` dataset from Chapter 6.

This is a regression problem, where the goal is to predict the `Salary` of a baseball player in 1987 using his performance statistics from 1986. After removing players with missing responses, we are left with 263 players and 19 variables. We randomly split the data into a training set of 176 players (two thirds), and a test set of 87 players (one third). We used three methods for fitting a regression model to these data.

- A linear model was used to fit the training data, and make predictions on the test data. The model has 20 parameters.
- The same linear model was fit with lasso regularization. The tuning parameter was selected by 10-fold cross-validation on the training data. It selected a model with 12 variables having nonzero coefficients.
- A neural network with one hidden layer consisting of 64 `ReLU` units was fit to the data. This model has 1,345 parameters.[21]

Table 10.2 compares the results. We see similar performance for all three models. We report the mean absolute error on the test data, as well as the test R^2 for each method, which are all respectable (see Exercise 5). We spent a fair bit of time fiddling with the configuration parameters of the neural network to achieve these results. It is possible that if we were to spend more time, and got the form and amount of regularization just right, that we might be able to match or even outperform linear regression and the lasso. But with great ease we obtained linear models that work well. Linear models are much easier to present and understand than the neural network, which is essentially a black box. The lasso selected 12 of the 19 variables in making its prediction. So in cases like this we are much better off following the *Occam's razor* principle: when faced with several methods

Occam's razor

[21]The model was fit by stochastic gradient descent with a batch size of 32 for 1,000 epochs, and 10% dropout regularization. The test error performance flattened out and started to slowly increase after 1,000 epochs. These fitting details are discussed in Section 10.7.

Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1345	257.4	0.54

TABLE 10.2. *Prediction results on the* `Hitters` *test data for linear models fit by ordinary least squares and lasso, compared to a neural network fit by stochastic gradient descent with dropout regularization.*

	Coefficient	Std. error	t-statistic	p-value
`Intercept`	-226.67	86.26	-2.63	0.0103
`Hits`	3.06	1.02	3.00	0.0036
`Walks`	0.181	2.04	0.09	0.9294
`CRuns`	0.859	0.12	7.09	< 0.0001
`PutOuts`	0.465	0.13	3.60	0.0005

TABLE 10.3. *Least squares coefficient estimates associated with the regression of* `Salary` *on four variables chosen by lasso on the* `Hitters` *data set. This model achieved the best performance on the test data, with a mean absolute error of 224.8. The results reported here were obtained from a regression on the test data, which was not used in fitting the lasso model.*

that give roughly equivalent performance, pick the simplest.

After a bit more exploration with the lasso model, we identified an even simpler model with four variables. We then refit the linear model with these four variables to the training data (the so-called *relaxed lasso*), and achieved a test mean absolute error of 224.8, the overall winner! It is tempting to present the summary table from this fit, so we can see coefficients and p-values; however, since the model was selected on the training data, there would be *selection bias*. Instead, we refit the model on the test data, which was not used in the selection. Table 10.3 shows the results.

We have a number of very powerful tools at our disposal, including neural networks, random forests and boosting, support vector machines and generalized additive models, to name a few. And then we have linear models, and simple variants of these. When faced with new data modeling and prediction problems, it's tempting to always go for the trendy new methods. Often they give extremely impressive results, especially when the datasets are very large and can support the fitting of high-dimensional nonlinear models. However, *if* we can produce models with the simpler tools that perform as well, they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches. 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.

10.7 Fitting a Neural Network

Fitting neural networks is somewhat complex, and we give a brief overview here. The ideas generalize to much more complex networks. Readers who find this material challenging can safely skip it. Fortunately, as we see in the lab at the end of this chapter, good software is available to fit neural network models in a relatively automated way, without worrying about the technical details of the model-fitting procedure.

We start with the simple network depicted in Figure 10.1 in Section 10.1. In model (10.1) the parameters are $\beta = (\beta_0, \beta_1, \ldots, \beta_K)$, as well as each of the $w_k = (w_{k0}, w_{k1}, \ldots, w_{kp})$, $k = 1, \ldots, K$. Given observations (x_i, y_i), $i = 1, \ldots, n$, we could fit the model by solving a nonlinear least squares problem

$$\underset{\{w_k\}_1^K,\, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2, \tag{10.23}$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\Big). \tag{10.24}$$

The objective in (10.23) looks simple enough, but because of the nested arrangement of the parameters and the symmetry of the hidden units, it is not straightforward to minimize. The problem is nonconvex in the parameters, and hence there are multiple solutions. As an example, Figure 10.17 shows a simple nonconvex function of a single variable θ; there are two solutions: one is a *local minimum* and the other is a *global minimum*. Furthermore, (10.1) is the very simplest of neural networks; in this chapter we have presented much more complex ones where these problems are compounded. To overcome some of these issues and to protect from overfitting, two general strategies are employed when fitting neural networks.

local minimum

global minimum

- *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.

gradient descent

- *Regularization:* penalties are imposed on the parameters, usually lasso or ridge as discussed in Section 6.2.

Suppose we represent all the parameters in one long vector θ. Then we can rewrite the objective in (10.23) as

$$R(\theta) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f_\theta(x_i))^2, \tag{10.25}$$

where we make explicit the dependence of f on the parameters. The idea of gradient descent is very simple.

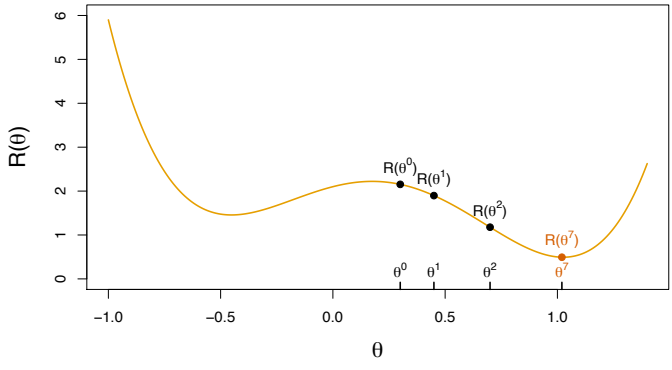

FIGURE 10.17. *Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill —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

1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.
2. Iterate until the objective (10.25) 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; i.e. such that $R(\theta^{t+1}) < R(\theta^t)$.
   (b) Set $t \leftarrow t + 1$.

One can visualize (Figure 10.17) standing in a mountainous terrain, and the goal is to get to the bottom through a series of steps. As long as each step goes downhill, we must eventually get to the bottom. In this case we were lucky, because with our starting guess θ^0 we end up at the global minimum. In general we can hope to end up at a (good) local minimum.

10.7.1 Backpropagation

How do we find the directions to move θ so as to decrease the objective $R(\theta)$ in (10.25)? The *gradient* of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of partial derivatives at that point: gradient

$$\nabla R(\theta^m) = \frac{\partial R(\theta)}{\partial \theta}\Big|_{\theta=\theta^m}. \tag{10.26}$$

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):

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m). \tag{10.27}$$

For a small enough value of the *learning rate* ρ, this step will decrease the objective $R(\theta)$; i.e. $R(\theta^{m+1}) \leq R(\theta^m)$. If the gradient vector is zero, then we may have arrived at a minimum of the objective.

learning rate

How complicated is the calculation (10.26)? It turns out that it is quite simple here, and remains simple even for much more complex networks, because of the *chain rule* of differentiation.

chain rule

Since $R(\theta) = \sum_{i=1}^n R_i(\theta) = \frac{1}{2}\sum_{i=1}^n (y_i - f_\theta(x_i))^2$ is a sum, its gradient is also a sum over the n observations, so we will just examine one of these terms,

$$R_i(\theta) = \frac{1}{2}\Big(y_i - \beta_0 - \sum_{k=1}^K \beta_k g\big(w_{k0} + \sum_{j=1}^p w_{kj} x_{ij}\big)\Big)^2. \tag{10.28}$$

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

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

And now we take the derivative with respect to w_{kj}:

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

Notice that both these expressions contain the residual $y_i - f_\theta(x_i)$. In (10.29) we see that a fraction of that residual gets attributed to each of the hidden units according to the value of $g(z_{ik})$. 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. Although these calculations are straightforward, it takes careful bookkeeping to keep track of all the pieces.

backpropagation

10.7.2 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. Also, 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

minibatch

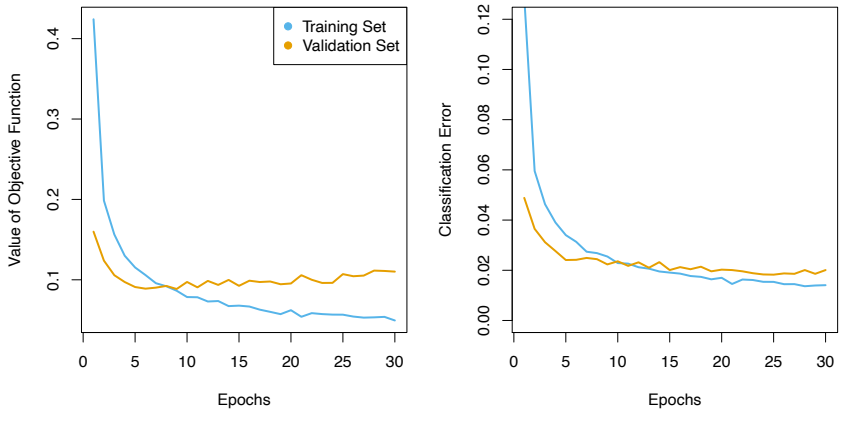

FIGURE 10.18. *Evolution of training and validation errors for the* MNIST *neural network depicted in Figure 10.4, as a function of training epochs. The objective refers to the log-likelihood (10.14).*

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. Fortunately, there is very good software for setting up deep learning models, and for fitting them to data, so most of the technicalities are hidden from the user.

stochastic gradient descent

We now turn to the multilayer network (Figure 10.4) used in the digit recognition problem. The network has over 235,000 weights, which is around four times the number of training examples. Regularization is essential here to avoid overfitting. The first row in Table 10.1 uses ridge regularization on the weights. This is achieved by augmenting the objective function (10.14) with a penalty term:

$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log(f_m(x_i)) + \lambda\sum_{j}\theta_j^2. \tag{10.31}$$

The parameter λ is often preset at a small value, or else it is found using the validation-set approach of Section 5.3.1. We can also 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, or as an alternative to ridge.

Figure 10.18 shows some metrics that evolve during the training of the network on the MNIST data. It turns out that SGD naturally enforces its own form of approximately quadratic regularization.[22] Here the minibatch

[22]This and other properties of SGD for deep learning are the subject of much research in the machine learning literature at the time of writing.

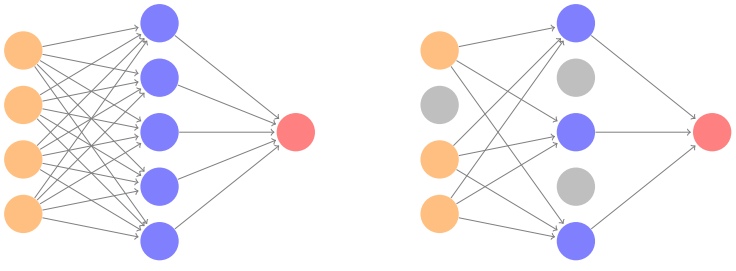

FIGURE 10.19. *Dropout Learning. Left: a fully connected network. Right: network with dropout in the input and hidden layer. The nodes in grey are selected at random, and ignored in an instance of training.*

size was 128 observations per gradient update. The term *epochs* labeling the horizontal axis in Figure 10.18 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.

epochs

early stopping

10.7.3 Dropout Learning

The second row in Table 10.1 is labeled *dropout*. This is a relatively new and efficient form of regularization, similar in some respects to ridge regularization. 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. Figure 10.19 illustrates this. 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.

dropout

10.7.4 Network Tuning

The network in Figure 10.4 is considered to be relatively straightforward; it nevertheless requires a number of choices that all have an effect on the performance: