13 Fully Connected Neural Networks

13.1 Introduction

As we first saw in Section 11.2.3, artificial neural networks, unlike polynomials and other fixed-shape approximators, have internal parameters that allow each of their units to take on a variety of shapes. In this chapter we expand on that introduction extensively, discussing general multi-layer neural networks, also referred to as *fully connected networks*, *multi-layer perceptrons*, and *deep feed-forward neural networks*.

13.2 Fully Connected Neural Networks

In this section we describe general fully connected neural networks, which are recursively built generalizations of the sort of units we first saw throughout Chapter 11. As this is an often confused subject we describe fully connected networks progressively (and with some redundancy that will hopefully benefit the reader), layer by layer, beginning with *single-hidden-layer* units first described in Section 11.2.3, providing algebraic, graphical, and computational perspectives on their construction. Afterwards, we briefly touch on the biological plausibility of fully connected networks, and end this section with an in-depth description of how to efficiently implement them in `Python`.

13.2.1 Single-hidden-layer units

The general algebraic representation (i.e., the formula) of a single-hidden-layer unit, also called a single-layer unit for short, is something we first saw in Equation (11.12) and is quite simple: a linear combination of input passed through a nonlinear *activation* function, which is typically an elementary mathematical function (e.g., tanh). Here we will denote such units in general as

$$f^{(1)}(\mathbf{x}) = a\left(w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n\right) \tag{13.1}$$

where $a(\cdot)$ denotes an activation function, and the superscripts on f and w_0

through w_N indicate they represent a single- (i.e., one-) layer unit and its internal weights, respectively.

Because we will want to extend the single-layer idea to create multi-layer networks, it is helpful to pull apart the sequence of two operations used to construct a single-layer unit: the linear combination of input, and the passage through a nonlinear activation function. We refer to this manner of writing out the unit as the *recursive recipe* for creating single-layer neural network units, and summarize it below.

Recursive recipe for single-layer units

1. Choose an activation function $a(\cdot)$
2. Compute the linear combination $v = w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n$
3. Pass the result through activation and form $a(v)$

Example 13.1 Illustrating the capacity of single-layer units

In the top panels of Figure 13.1 we plot four instances of a single-layer unit using tanh as our nonlinear activation function. These four nonlinear units take the form

$$f^{(1)}(x) = \tanh\left(w_0^{(1)} + w_1^{(1)} x\right) \tag{13.2}$$

where in each instance the internal parameters of the unit (i.e., $w_0^{(1)}$ and $w_1^{(1)}$) have been set randomly, giving each instance a distinct shape. This roughly illustrates the *capacity* of each single-layer unit. Capacity (a concept first introduced in Section 11.2.2) refers to the range of shapes a function like this can take, given all the different settings of its internal parameters.

In the bottom row of Figure 13.1 we swap out tanh for the ReLU[1] activation function, forming a single-layer unit of the form

$$f^{(1)}(x) = \max\left(0, w_0^{(1)} + w_1^{(1)} x\right). \tag{13.3}$$

Once again the internal parameters of this unit allow it to take on a variety of shapes (distinct from those created by tanh activation), four instances of which are illustrated in the bottom panels of the figure.

---

[1] The Rectified Linear Unit or ReLU function was first introduced in the context of two-class classification and the Perceptron cost in Section 6.4.2.

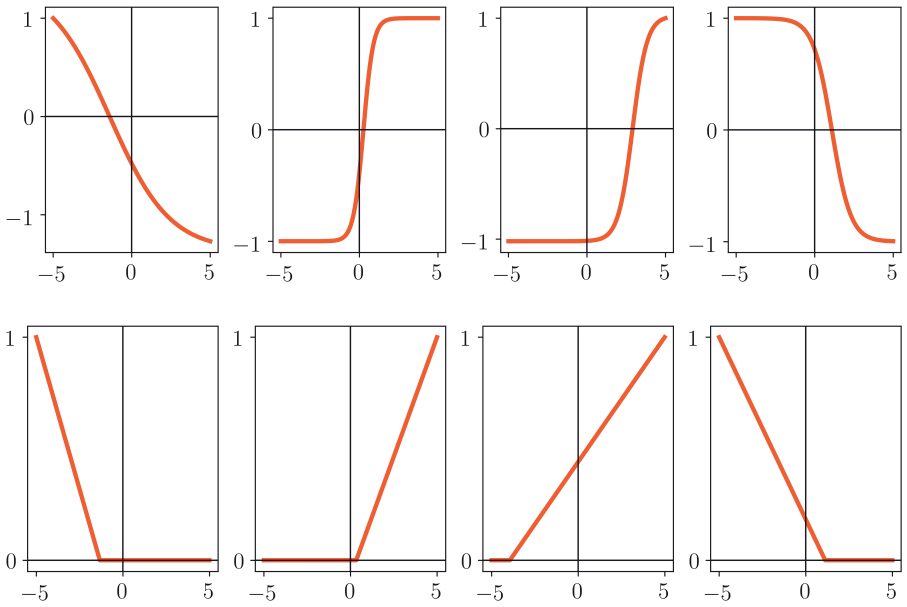

Figure 13.1 Figure associated with Example 13.1. Four instances of a single-layer neural network unit with tanh (top row) and ReLU activation (bottom row). See text for further details.

If we form a general nonlinear model using $B = U_1$ such single-layer units as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(1)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_1}^{(1)}\left(\mathbf{x}\right) w_{U_1} \tag{13.4}$$

whose jth unit takes the form

$$f_j^{(1)}\left(\mathbf{x}\right) = a\left(w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n\right) \tag{13.5}$$

then the parameter set Θ contains not only the weights of the final linear combination w_0 through w_{U_1}, but all parameters internal to each $f_j^{(1)}$ as well. This is precisely the sort of model we used in the neural network examples throughout Chapter 11.

The left panel of Figure 13.2 shows a common graphical representation of the single-layer model in Equation (13.4), and visually unravels the individual algebraic operations performed by such a model. A visual representation like this is often referred to as a *neural network architecture* or just an *architecture*. Here the bias and input of each single-layer unit composing the model is shown as a sequence of dots all the way on the left of the diagram. This layer is "visible" to us since this is where we inject the input data to our network which we ourselves can "see," and is also often referred to as the *first* or *input* layer of the

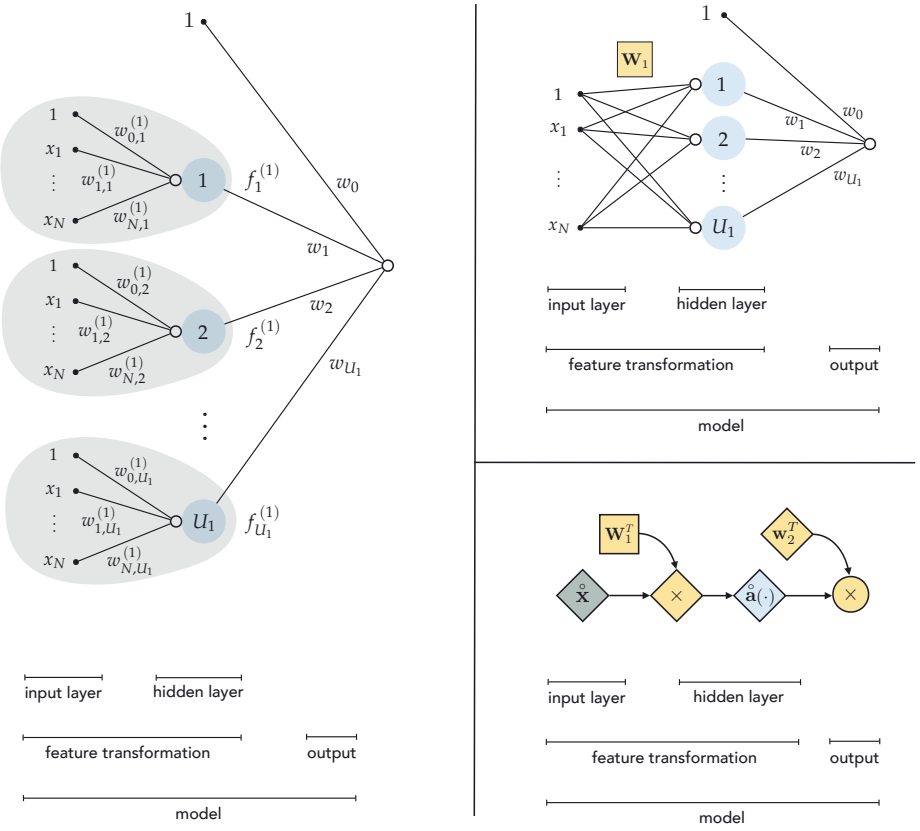

Figure 13.2 (left panel) Graphical representation of a single-layer neural network model, given in Equation (13.4), which is composed of U_1 single-layer units. (top-right panel) A condensed graphical representation of a single-layer neural network. (bottom-right panel) This network can be represented even more compactly, illustrating in a simple diagram all of the computation performed by a single-layer neural network model. See text for further details.

network. The linear combination of input leading to each unit is then shown visually by edges connecting the input to a hollow circle (summation unit), with the nonlinear activation then shown as a larger blue circle (activation unit). In the middle of this visual depiction (where the blue circles representing all U_1 activations align) is the *hidden* layer of this architecture. This layer is called "hidden" because it contains internally processed versions of our input that we do not "see." While the name hidden is not entirely accurate (as we can visualize the internal state of these units if we so desire) it is a commonly used convention, hence the name single-*hidden*-layer unit. The output of these U_1 units is then collected in a linear combination, and once again visualized by edges connecting each unit to a final summation shown as a hollow circle. This is the final output

of the model, and is often called the *final* or *output* layer of the network, which is again "visible" to us (not hidden).

Compact representation of single-layer neural networks

Because we will soon wish to add more hidden layers to our rudimentary model in detailing multi-layer networks, the sort of visual depiction in the left panel of Figure 13.2 quickly becomes unwieldy. Thus, in order to keep ourselves organized and better prepared to understand *deeper* neural network units, it is quite helpful to compactify this visualization. We can do so by first using a more compact notation to represent our model algebraically, beginning by more concisely representing our input, placing a 1 at the top of our input vector $\mathbf{x}$, which we denote by placing a hollow ring symbol over $\mathbf{x}$ as[2]

$$\mathring{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{bmatrix}. \tag{13.6}$$

Next we collect all of the *internal parameters* of our U_1 single-layer units. Examining the algebraic form for the jth unit in Equation (13.5) we can see that it has $N+1$ such internal parameters. Taking these parameters we form a column vector, starting with the bias $w_{0,j}^{(1)}$ and then input-touching weights $w_{1,j}^{(1)}$ through $w_{N,j}^{(1)}$, and place them into the jth column of an $(N+1) \times U_1$ matrix

$$\mathbf{W}_1 = \begin{bmatrix} w_{0,1}^{(1)} & w_{0,2}^{(1)} & \cdots & w_{0,U_1}^{(1)} \\ w_{1,1}^{(1)} & w_{1,2}^{(1)} & \cdots & w_{1,U_1}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N,1}^{(1)} & w_{N,2}^{(1)} & \cdots & w_{N,U_1}^{(1)} \end{bmatrix}. \tag{13.7}$$

With this notation note how the matrix-vector product $\mathbf{W}_1^T \mathring{\mathbf{x}}$ contains every linear combination *internal* to our U_1 nonlinear units. In other words, $\mathbf{W}_1^T \mathring{\mathbf{x}}$ has dimension $U_1 \times 1$, and its jth entry is precisely the linear combination of the input data internal to the jth unit

$$\left[\mathbf{W}_1^T \mathring{\mathbf{x}}\right]_j = w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n \qquad j = 1, 2, ..., U_1. \tag{13.8}$$

Next, we extend our notation for the arbitrary activation function $a(\cdot)$ to

[2] This notation was introduced and employed previously in Section 5.2.

handle such a vector. More specifically, we define $\mathbf{a}(\cdot)$ as the vector function that takes in a general $d \times 1$ vector $\mathbf{v}$ and returns – as output – a vector of the same dimension containing activation of each of its input's entries, as

$$\mathbf{a}(\mathbf{v}) = \begin{bmatrix} a(v_1) \\ \vdots \\ a(v_d) \end{bmatrix}. \tag{13.9}$$

Notice how with this notation, the vector activation $\mathbf{a}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)$ becomes a $U_1 \times 1$ vector containing all U_1 single-layer units, the jth of which is given as

$$\left[\mathbf{a}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right]_j = a\left(w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n\right) \qquad j = 1, 2, ..., U_1. \tag{13.10}$$

Using another compact notation to denote the weights of the final linear combination as

$$\mathbf{w}_2 = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_1} \end{bmatrix} \tag{13.11}$$

and extending our vector $\mathbf{a}$ by tacking a 1 on top of it – denoting the resulting $(U_1 + 1) \times 1$ vector $\mathring{\mathbf{a}}$ – we can finally write out the model in Equation (13.4) quite compactly as

$$\text{model}(\mathbf{x}, \Theta) = \mathbf{w}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right). \tag{13.12}$$

This more compact algebraic formulation lends itself to much more easily digestible visual depictions. In the top-right panel of Figure 13.2 we show a slightly condensed version of our original graph in the left panel, where the linear weights attached to each input are now shown more compactly as the set of crisscrossing line segments connecting the input to each unit, with the matrix $\mathbf{W}_1$ jointly representing all U_1 of the weighted combinations. In the bottom-right panel of Figure 13.2 we compactify our original visual representation even further. In this more compact representation we can more easily visualize the computation performed by the general single-layer neural network model in Equation (13.12), where we depict the scalars, vectors, and matrices in this formula symbolically as circles, diamonds, and squares, respectively.

13.2.2 Two-hidden-layer units

To create a *two-hidden-layer* neural network unit, or a two-layer unit for short, we *recurse* on the idea of the single-layer unit detailed in the previous section.

We do this by first constructing a set of U_1 single-layer units and treat them as input to another nonlinear unit. That is, we take their linear combination and pass the result through a nonlinear activation.

The algebraic form of a general two-layer unit is given as

$$f^{(2)}(\mathbf{x}) = a\left(w_0^{(2)} + \sum_{i=1}^{U_1} w_i^{(2)} f_i^{(1)}(\mathbf{x})\right) \tag{13.13}$$

which reflects the recursive nature of constructing two-layer units using single-layer ones. This recursive nature can be also seen in the *recursive recipe* given below for building two-layer units.

Recursive recipe for two-layer units

1. Choose an activation function $a(\cdot)$
2. Construct U_1 single-layer units $f_i^{(1)}(\mathbf{x})$ for $i = 1, 2, ..., U_1$
3. Compute the linear combination $v = w_0^{(2)} + \sum_{i=1}^{U_1} w_i^{(2)} f_i^{(1)}(\mathbf{x})$
4. Pass the result through activation and form $a(v)$

Example 13.2 Illustrating the capacity of two-layer units

In the top row of Figure 13.3 we plot four instances of a two-layer neural network unit – using tanh activation – of the form

$$f^{(2)}(x) = \tanh\left(w_0^{(2)} + w_1^{(2)} f^{(1)}(x)\right) \tag{13.14}$$

where

$$f^{(1)}(x) = \tanh\left(w_0^{(1)} + w_1^{(1)} x\right). \tag{13.15}$$

The wider variety of shapes taken on by instances of this unit, as shown in the figure, reflects the increased capacity of two-layer units over their single-layer analogs shown in Figure 13.1.

In the bottom row of the figure we show four exemplars of the same sort of unit, only now we use ReLU activation instead of tanh in each layer.

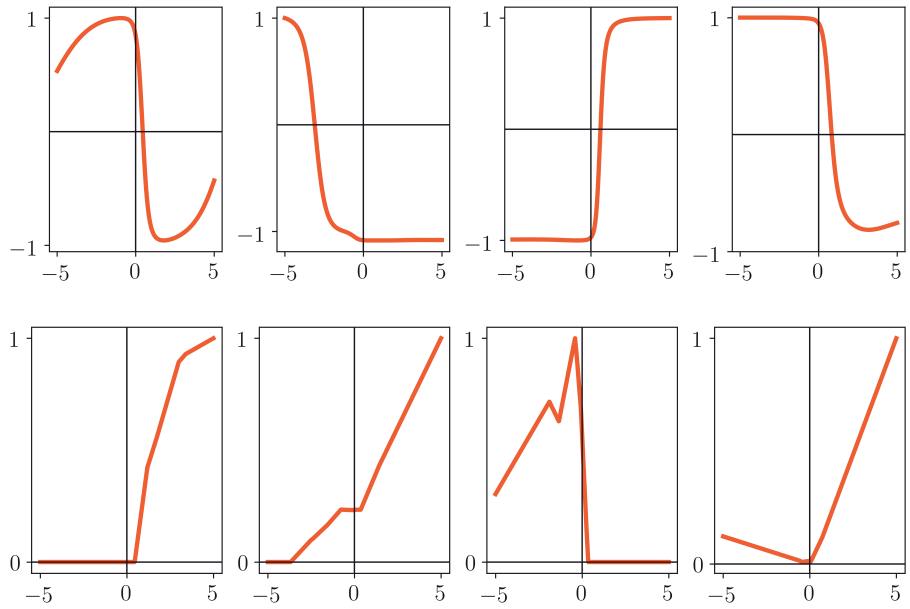

Figure 13.3 Figure associated with Example 13.2. Four instances of a two-layer neural network unit with tanh (top row) and ReLU activation (bottom row). See text for further details.

In general, if we wish to create a model using $B = U_2$ two-layer neural network units we write

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(2)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_2}^{(2)}\left(\mathbf{x}\right) w_{U_2} \tag{13.16}$$

where

$$f_j^{(2)}\left(\mathbf{x}\right) = a\left(w_{0,j}^{(2)} + \sum_{i=1}^{U_1} w_{i,j}^{(2)} f_i^{(1)}\left(\mathbf{x}\right)\right) \qquad j = 1, 2, ..., U_2 \tag{13.17}$$

and where the parameter set Θ, as always, contains those (superscripted) weights internal to the neural network units as well as the final linear combination weights. Importantly note that while each two-layer unit $f_j^{(2)}$ in Equation (13.17) has unique internal parameters – denoted by $w_{i,j}^{(2)}$ where i ranges from 0 to U_1 – the weights internal to each single-layer unit $f_i^{(1)}$ are the same across all the two-layer units themselves.

Figure 13.4 shows a graphical representation (or architecture) of a generic two-layer neural network model whose algebraic form is given in Equation (13.16). In the left panel we illustrate each input single-layer unit precisely as shown previously in the top-right panel of Figure 13.2. The input layer, all the way to the left, is first fed into each of our U_1 single-layer units (which still constitutes

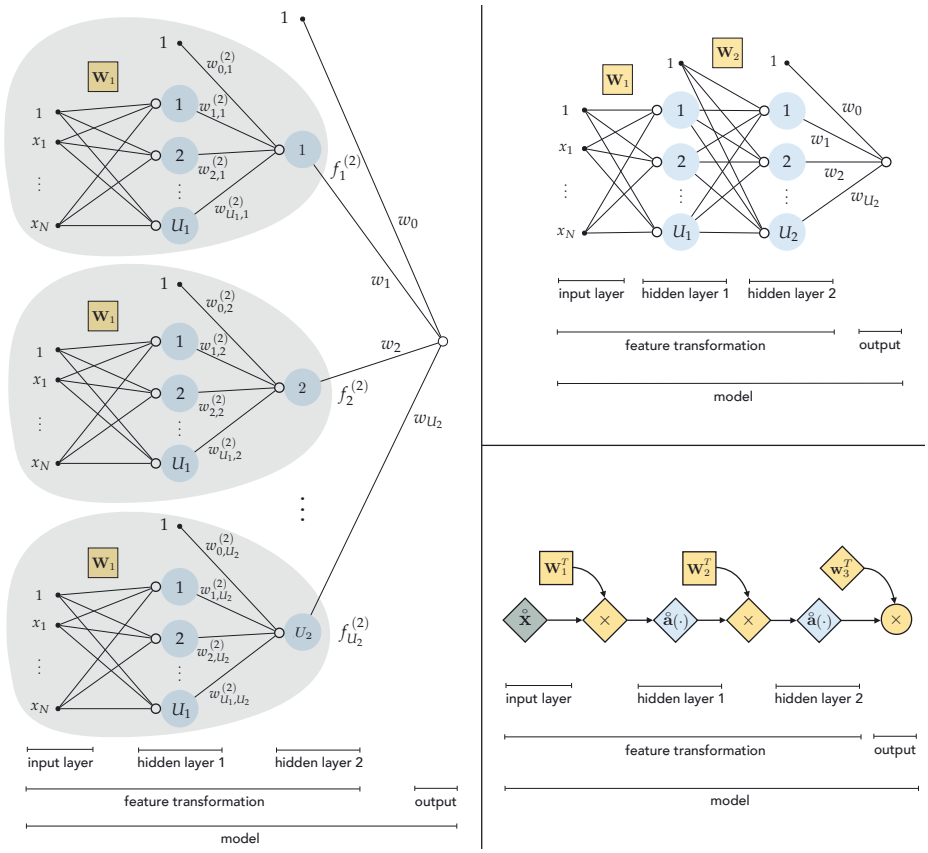

Figure 13.4 (left panel) Graphical representation of a two-layer neural network model, given in Equation (13.16), which is composed of U_2 two-layer units. (top-right panel) A condensed graphical representation of a two-layer neural network. (bottom-right panel) This network can be represented more compactly, providing a simpler depiction of the computation performed by a two-layer neural network model. See text for further details.

the first *hidden* layer of the network). A linear combination of these single-layer units is then fed into each of the U_2 two-layer units, referred to by convention as the second *hidden* layer, since its computation is also not immediately "visible" to us. Here we can also see why this sort of architecture is referred to as *fully connected*: every dimension of input is connected to every unit in the first hidden layer, and each unit of the first hidden layer is connected to every unit of the second hidden layer. At last, all the way to the right of this panel, we see a linear combination of the U_2 two-layer units which produces the final (visible) layer of the network: the output of our two-layer model.

Compact representation of two-layer neural networks

As with single-layer models, here it is also helpful to compactify both our notation and the corresponding visualization of a two-layer model in order to simplify our understanding and make the concept easier to wield. Using the same notation introduced in Section 13.2.1, we can compactly designate the output of our U_1 single-layer units as

$$\text{output of first hidden layer:} \quad \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right). \tag{13.18}$$

Following the same pattern as before we can then condense all internal weights of the U_2 units in the second layer column-wise into a $(U_1 + 1) \times U_2$ matrix of the form

$$\mathbf{W}_2 = \begin{bmatrix} w_{0,1}^{(2)} & w_{0,2}^{(2)} & \cdots & w_{0,U_2}^{(2)} \\ w_{1,1}^{(2)} & w_{1,2}^{(2)} & \cdots & w_{1,U_2}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{U_1,1}^{(2)} & w_{U_1,2}^{(2)} & \cdots & w_{U_1,U_2}^{(2)} \end{bmatrix} \tag{13.19}$$

which mirrors precisely how we defined the $(N + 1) \times U_1$ internal weight matrix $\mathbf{W}_1$ for our single-layer units in Equation (13.7). This allows us to likewise express the output of our U_2 two-layer units compactly as

$$\text{output of second hidden layer:} \quad \mathring{\mathbf{a}}\left(\mathbf{W}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right). \tag{13.20}$$

The recursive nature of two-layer units is on full display here. Remember that we use the notation $\mathring{\mathbf{a}}(\cdot)$ somewhat loosely as a vector-valued function in the sense that it simply represents taking the nonlinear activation $a(\cdot)$, *element-wise*, of whatever vector is input into it as shown in Equation (13.9), with a 1 appended to the top of the result.

Concatenating the final linear combination weights into a single vector as

$$\mathbf{w}_3 = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_2} \end{bmatrix} \tag{13.21}$$

allows us to write the full two-layer neural network model as

$$\text{model}(\mathbf{x}, \Theta) = \mathbf{w}_3^T \mathring{\mathbf{a}}\left(\mathbf{W}_2^T \mathring{\mathbf{a}}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right)\right). \tag{13.22}$$

As with its single-layer analog, this compact algebraic formulation of a two-layer

neural network lends itself to much more easily digestible visual depictions. In the top-right panel of Figure 13.4 we show a slightly condensed version of the original graph in the left panel, where the redundancy of showing every single-layer unit has been reduced to a single visual representation. In doing so we remove all the weights assigned to crisscrossing edges connecting the first and second hidden layers, and place them in the matrix $\mathbf{W}_2$ defined in Equation (13.19) to avoid cluttering the visualization. In the bottom-right panel of Figure 13.4 we condense this two-layer depiction even further where scalars, vectors, and matrices are depicted symbolically as circles, diamonds, and squares, respectively. This greatly compacted graph provides a simplified visual representation of the total computation performed by a general two-layer neural network model.

13.2.3 General multi-hidden-layer units

Following the same pattern we have seen thus far in describing single- and two-layer units we can construct general fully connected neural network units with an arbitrary number of hidden layers. With each hidden layer added we increase the capacity of a neural network unit, as we have seen previously in graduating from single-layer units to two-layer units, as well as a model built using such units.

To construct a general L-hidden-layer neural network unit, or L-layer unit for short, we simply recurse on the pattern we have established previously $L-1$ times, with the resulting L-layer unit taking as input a number U_{L-1} of $(L-1)$-layer units, as

$$f^{(L)}(\mathbf{x}) = a\left(w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})\right). \tag{13.23}$$

As was the case with single- and two-layer units, this formula is perhaps easier to digest if we think about it in terms of the *recursive recipe* given below.

Recursive recipe for L-layer units

1. Choose an activation function $a(\cdot)$
2. Construct U_{L-1} number of $(L-1)$-layer units $f_i^{(L-1)}(\mathbf{x})$ for $i = 1, 2, ..., U_{L-1}$
3. Compute the linear combination $v = w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})$
4. Pass the result through activation and form $a(v)$

Note that while in principle the same activation function need not be used for all hidden layers of an L-layer unit, for the sake of simplicity, a single kind of activation is almost always used.

Example 13.3 Illustrating the capacity of three-layer units

In the top row of Figure 13.5 we show four instances of a three-layer unit with tanh activation. The greater variety of shapes shown here, as compared to single- and two-layer analogs in Examples 13.1 and 13.2, reflects the increased capacity of these units. In the bottom row we repeat the same experiment, only using the ReLU activation function instead of tanh.

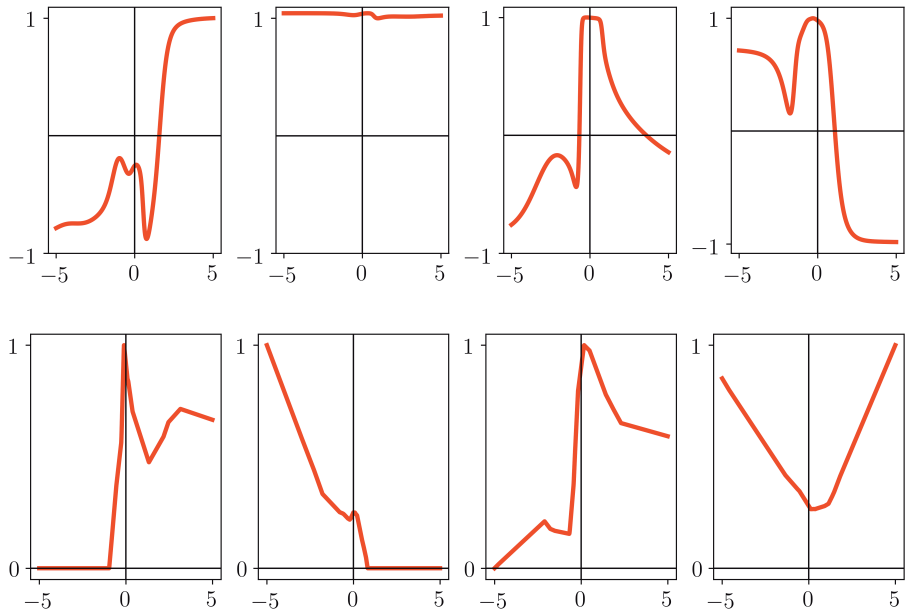

Figure 13.5 Figure associated with Example 13.3. Four instances of a three-layer network unit with tanh (top row) and ReLU activation (bottom row). See text for further details.

In general we can produce a model consisting of $B = U_L$ such L-layer units as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(L)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_L}^{(L)}\left(\mathbf{x}\right) w_{U_L} \tag{13.24}$$

where

$$f_j^{(L)}(\mathbf{x}) = a\left(w_{0,j}^{(L)} + \sum_{i=1}^{U_{L-1}} w_{i,j}^{(L)} f_i^{(L-1)}(\mathbf{x})\right) \qquad j = 1, 2, ..., U_L \tag{13.25}$$

and where the parameter set Θ contains both those weights internal to the neural network units as well as the final linear combination weights.

Figure 13.6 shows an unraveled graphical representation of this model, which is a direct generalization of the kinds of visualizations we have seen previously with single- and two-layer networks. From left to right we can see the input layer to the network, its L hidden layers, and the output. Often models built using three or more hidden layers are referred to as *deep* neural networks in the jargon of machine learning.

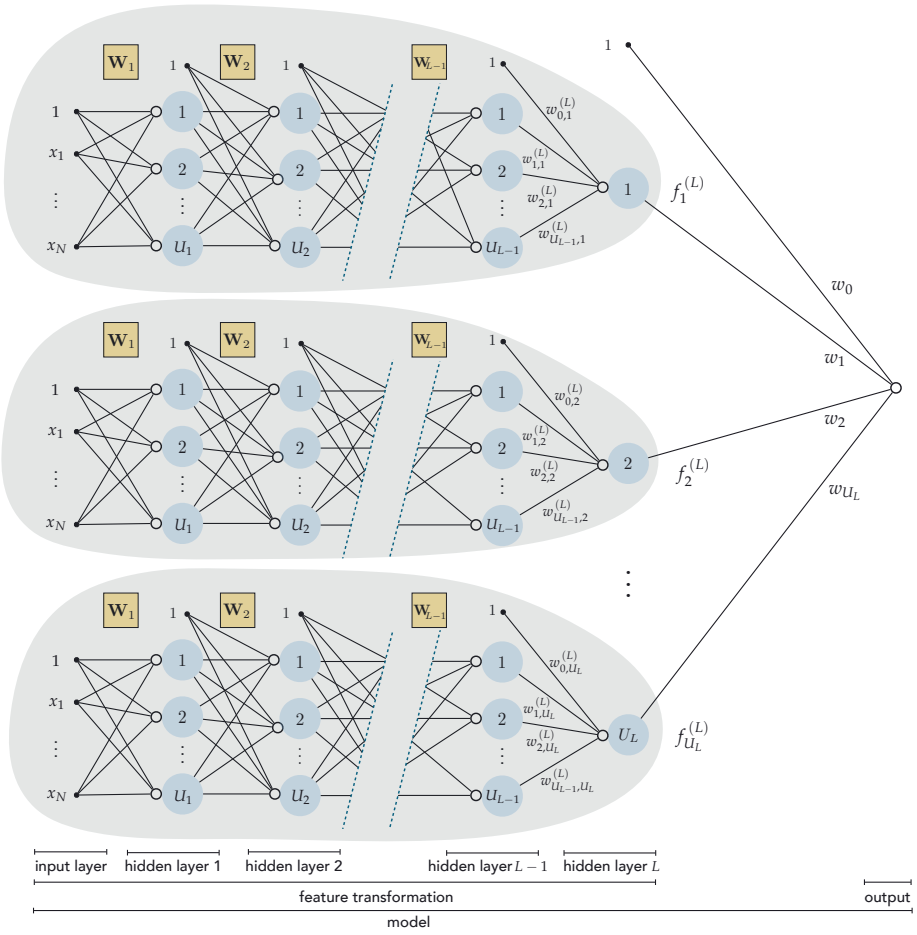

Figure 13.6 Graphical representation of an L-layer neural network model, given in Equation (13.24), which is composed of U_L L-layer units.

Compact representation of multi-layer neural networks

To simplify our understanding of a general multi-layer neural network architecture we can use precisely the same compact notation and visualizations we have introduced in the simpler contexts of single- and two-layer neural networks. In complete analogy to the way we compactly represented two-layer neural networks, we denote the output of the Lth hidden layer compactly as

$$\text{output of } L\text{th hidden layer:} \quad \mathring{\mathbf{a}}\left(\mathbf{W}_L^T\,\mathring{\mathbf{a}}\left(\mathbf{W}_{L-1}^T\mathring{\mathbf{a}}\left(\cdots\mathring{\mathbf{a}}\left(\mathbf{W}_1^T\mathring{\mathbf{x}}\right)\right)\right)\right). \tag{13.26}$$

Denoting the weights of the final linear combination as

$$\mathbf{w}_{L+1} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_L} \end{bmatrix} \tag{13.27}$$

we can express the L-layer neural network model compactly as

$$\text{model}\left(\mathbf{x},\Theta\right) = \mathbf{w}_{L+1}^T\mathring{\mathbf{a}}\left(\mathbf{W}_L^T\,\mathring{\mathbf{a}}\left(\mathbf{W}_{L-1}^T\mathring{\mathbf{a}}\left(\cdots\mathring{\mathbf{a}}\left(\mathbf{W}_1^T\mathring{\mathbf{x}}\right)\right)\right)\right). \tag{13.28}$$

Once again this compact algebraic formulation of an L-layer neural network lends itself to much more easily digestible visual depictions. The top panel of Figure 13.7 shows a condensed version of the original graph in Figure 13.6, where the redundancy of showing every $(L-1)$-layer unit has been reduced to a single visualization. The succinct visual depiction shown in the bottom panel of Figure 13.7 represents this network architecture even more compactly, with scalars, vectors, and matrices are shown symbolically as circles, diamonds, and squares, respectively.

13.2.4 Selecting the right network architecture

We have now seen a general and recursive method of constructing arbitrarily "deep" neural networks, but many curiosities and technical issues remain to be addressed that are the subject of subsequent sections in this chapter. These include the choice of activation function, popular cross-validation methods for models employing neural network units, as well as a variety of optimization-related issues such as the notions of *backpropagation* and *batch normalization*.

However, one fundamental question can (at least in general) be addressed now, which is how we choose the "right" number of units and layers for a neural network architecture. As with the choice of proper universal approximator in general (as detailed in Section 11.8), typically we do not know *a priori* what sort of architecture will work best for a given dataset (in terms of number of hidden layers and number of units per hidden layer). In principle, to determine the best architecture for use with a given dataset we must cross-validate an array

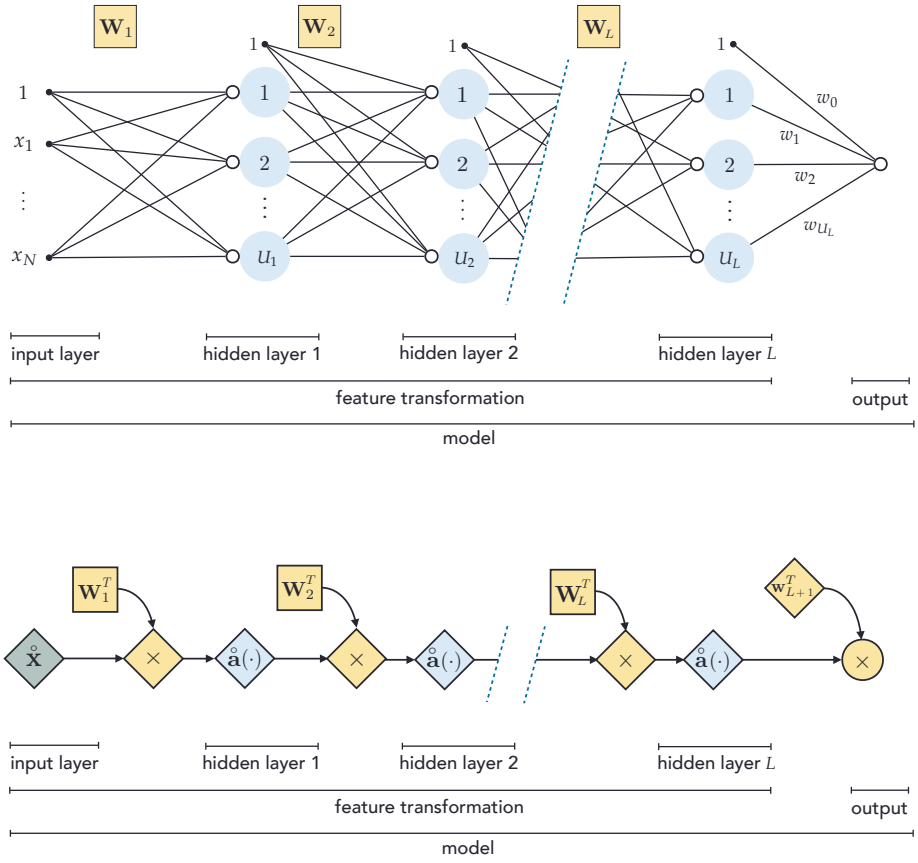

Figure 13.7 (top panel) A condensed graphical representation of an L-layer neural network model shown in Figure 13.6. (bottom panel) A more compact version, succinctly describing the computation performed by a general L-layer neural network model. See text for further details.

of choices. In doing so we note that, generally speaking, the capacity gained by adding new individual units to a neural network model is typically much smaller relative to the capacity gained by the addition of new hidden layers. This is because appending an additional *layer* to an architecture grafts an additional recursion to the computation involved in each unit, which significantly increases their capacity and that of any corresponding model, as we have seen in Examples 13.1, 13.2, and 13.3. In practice, performing model search across a variety of neural network architectures can be expensive, hence compromises must be made that aim at determining a high-quality model using minimal computation. To this end, early stopping based regularization (see Sections 11.6.2 and 13.7) is commonly employed with neural network models.

13.2.5 Neural networks: the biological perspective

The human brain contains roughly 10^{11} biological neurons which work together in concert when we perform cognitive tasks. Even when performing relatively minor tasks we employ a sizable series of interconnected neurons – called *biological neural networks* – to perform the task properly. For example, somewhere between 10^5 to 10^6 neurons are required to render a realistic image of our visual surroundings. Most of the basic jargon and modeling principles of neural network universal approximators we have seen thus far originated as a (very) rough mathematical model of such biological neural networks.

An individual biological neuron (shown in the top-left panel of Figure 13.8) consists of three main parts: *dendrites* (the neuron's receivers), *soma* (the cell body), and *axon* (the neuron's transmitter). Starting around the 1940s psychologists and neuroscientists, with a shared desire to understand the human brain better, became interested in modeling neurons mathematically. These early models, later dubbed as artificial neurons (a basic exemplar of which is shown in the top-right panel of Figure 13.8), culminated in the introduction of the *Perceptron* model in 1957 [59]. Closely mimicking a biological neuron's structure, an artificial neuron comprises a set of dendrite-like edges that connect it to other neurons, each taking an input and multiplying it by a (synaptic) weight associated with that edge. These weighted inputs are summed up after going through a *summation unit* (shown by a small hollow circle). The result is subsequently fed to an *activation unit* (shown by a large blue circle) whose output is then transmitted to the outside via an axon-like projection. From a biological perspective, neurons are believed to remain inactive until the net input to the cell body (soma) reaches a certain threshold, at which point the neuron gets *activated* and fires an electro-chemical signal, hence the name *activation function*.

Stringing together large sets of such artificial neurons in layers creates a more mathematically complex model of a biological neural network, which still remains a *very* simple approximation to what goes on in the brain. The bottom panel of Figure 13.8 shows the kind of graphical representation (here of a two-layer network) used when thinking about neural networks from this biological perspective. In this rather complex visual depiction every multiplicative operation of the architecture is shown as an edge, creating a mesh of intersecting lines connecting each layer. In describing fully connected neural networks in Sections 13.2.1 through 13.2.3 we preferred simpler, more compact visualizations that avoid this sort of complex visual mesh.

13.2.6 `Python` implementation

In this section we show how to implement a generic model consisting of L-layer neural network units in `Python` using `NumPy`. Because such models can have a large number of parameters, and because associated cost functions employing them can be highly nonconvex (as further discussed in Section 13.5), first-order

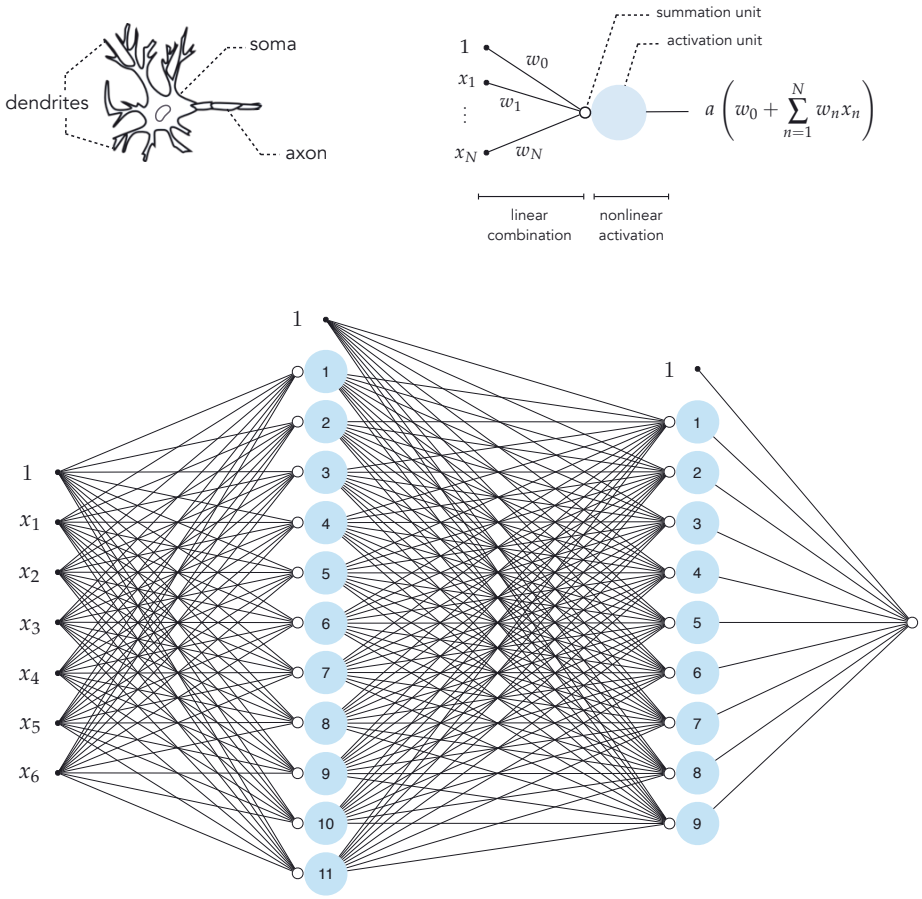

Figure 13.8 (top-left panel) A typical biological neuron. (top-right panel) An artificial neuron, that is a simplistic mathematical model of the biological neuron, consisting of: (i) weighted edges that represent the individual multiplications (of 1 by w_0, x_1 by w_1, etc.), (ii) a summation unit shown as a small hollow circle representing the sum $w_0 + w_1 x_1 + \cdots + w_N x_N$, and (iii) an activation unit shown as a larger blue circle representing the sum evaluated by the nonlinear activation function $a(\cdot)$. (bottom panel) An example of a fully connected two-layer neural network as commonly illustrated when detailing neural networks from a biological perspective.

local optimization methods (e.g., gradient descent and its variants detailed in Chapter 3 and Appendix A) are the most common schemes used to tune the parameters of general neural network models. Moreover, because computing derivatives of a model employing neural network units "by hand" is extremely tedious (see Section 13.4) the use of automatic differentiation is not only helpful but in most cases necessary. Thus in the `Python` implementation detailed here we strongly recommend using `autograd` – the automatic differentiator used

throughout this text (see Section B.10.2) – to make computing derivatives clean and simple.

Below we provide an implementation of a neural network feature transformation called `feature_transforms` (a notation we first introduced in Chapter 10).

```
1  # neural network feature transformation
2  def feature_transforms(a, w):
3
4      # loop through each layer
5      for W in w:
6
7          # compute inner-product with current layer weights
8          a = W[0] + np.dot(a.T, W[1:])
9
10         # pass through activation
11         a = activation(a).T
12
13     return a
```

This `Python` function takes in the input `x`, written as the variable `a`, and the entire set of weight matrices $\mathbf{W}_1$ through $\mathbf{W}_L$ internal to our neural network architecture, written as the variable `w` where `w = [W_1, W_2, ..., W_L]`. The output of our feature transformation function is the output of the final layer of the network, expressed algebraically in Equation (13.26). We compute this output recursively, looping *forward* through the network architecture, starting with the first hidden layer using matrix $\mathbf{W}_1$ and ending with the computation of the final hidden layer using $\mathbf{W}_L$. This results in a `Python` function consisting of a simple `for` loop over the weight matrices of the hidden layers. Notice, in line 11 of the implementation above, that `activation` can refer to any elementary function built using `NumPy` operations. For example, the tanh activation can be written as `np.tanh(a)`, and the ReLU as `np.maximum(0,a)`.

With our feature transformation function complete we can now implement our `model`, which is a simple variation on the implementations we have seen in previous chapters. Here the inputs to this `Python` function are `x` – our input data – and a `Python` list `theta` of length two whose first entry contains our list of internal weight matrices, and whose second entry contains the weights of the final linear combination.

```
1  # neural network model
2  def model(x, theta):
3
4      # compute feature transformation
5      f = feature_transforms(x, theta[0])
6
7      # compute final linear combination
8      a = theta[1][0] + np.dot(f.T, theta[1][1:])
```

```
    return a.T
```

This implementation of a fully connected neural network model can be easily paired with Pythonic implementation of the generic machine learning cost functions detailed in previous chapters.

Finally, we provide a `Python` function called `network_initializer` that creates initial weights for a general neural network model, and also provides a simple interface for creating general architectures: it is precisely this initializer that determines the shape of our implemented network. To create a desired network we simply input a comma-separated list called `layer_sizes` of the following form

```
layer_sizes = [N, U_1, ..., U_L, C]
```

where `N` is the input dimension, `U_1` through `U_L` are the number of desired units in the hidden layers 1 through L, respectively, and `C` is the output dimension.

The initializer will then automatically create initial weight matrices (of the proper dimensions) as well as the final weights for the linear combination, packaged together as the output `theta_init`.

```
# create initial weights for a neural network model
def network_initializer(layer_sizes, scale):

    # container for all tunable weights
    weights = []

    # create appropriately-sized initial
    # weight matrix for each layer of network
    for k in range(len(layer_sizes)-1):

        # get layer sizes for current weight matrix
        U_k = layer_sizes[k]
        U_k_plus_1 = layer_sizes[k+1]

        # make weight matrix
        weight = scale*np.random.randn(U_k+1, U_k_plus_1)
        weights.append(weight)

    # repackage weights so that theta_init[0] contains all
    # weight matrices internal to the network, and theta_init[1]
    # contains final linear combination weights
    theta_init = [weights[:-1], weights[-1]]

    return theta_init
```

Next, we provide several examples using this implementation.

Example 13.4 Nonlinear classification using multi-layer neural networks
In this example we use a multi-layer architecture to perform nonlinear classification, first on the two-class dataset shown previously in Example 11.9.2. Here, we arbitrarily choose the network to have four hidden layers with ten units in each layer, and the tanh activation. We then tune the parameters of this model by minimizing an associated two-class Softmax cost (see Equation (10.31)) via gradient descent, visualizing the nonlinear decision boundary learned in the top row of Figure 13.9 along with the dataset itself.

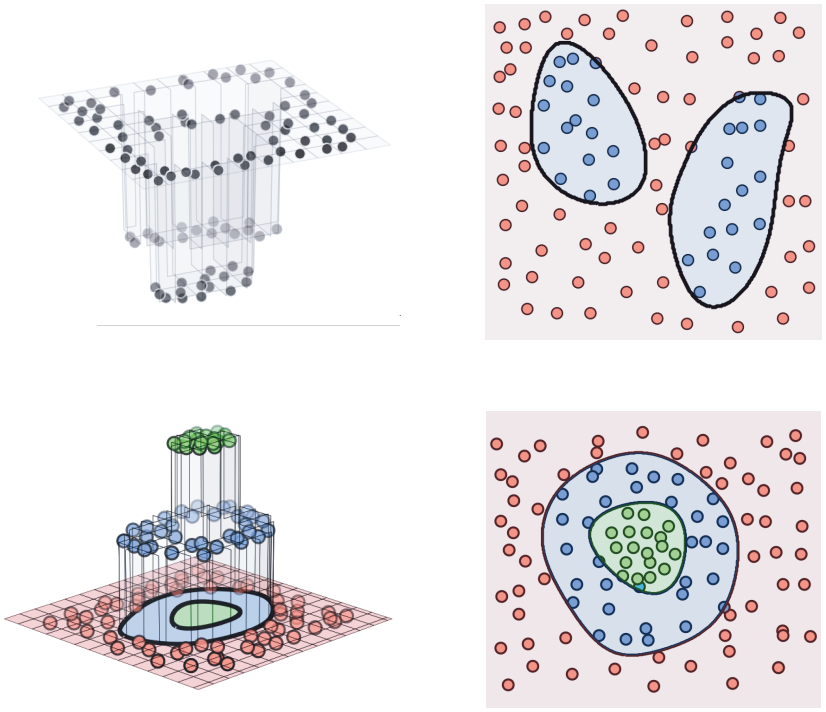

Figure 13.9 Figure associated with Example 13.4. The resulting decision boundary learned by a fully connected neural network on a two-class dataset (top row) and multi-class dataset (bottom row), from both the regression perspective (left column) and perceptron perspective (right column). See text for further details.

Next, we perform multi-class classification on the multi-class dataset first shown in Example 10.6 ($C = 3$), using a model consisting of two hidden layers, choosing the number of units in each layer arbitrarily as $U_1 = 12$ and $U_2 = 5$, respectively, the tanh activation, and using a shared scheme (that is, the network architecture is shared by each classifier, as detailed in Section 10.5 for general feature transformations). We then tune the parameters of this model by minimizing the corresponding multi-class Softmax cost (as shown in Equation

(10.42)), and show the resulting learned decision boundary in the bottom row of Figure 13.9.

Example 13.5 Random Autoencoder manifolds

In Section 10.6.1 we introduced the general nonlinear Autoencoder, which consists of two nonlinear functions: an encoder $\mathbf{f}_e$ and decoder $\mathbf{f}_d$, whose parameters we tune so that (ideally) the composition $\mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}\right)\right)$ forms the best *nonlinear manifold* on which an input dataset rests. In other words, given a set of input points $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$ we aim to tune the parameters of our encoder/decoder pair so that

$$\mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}_p\right)\right) \approx \mathbf{x}_p \qquad p = 1, 2, ..., P. \tag{13.29}$$

In Figure 13.10 we visualize nine instances of the function $\mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}\right)\right)$, each of which is technically called a manifold, where both $\mathbf{f}_d$ and $\mathbf{f}_e$ are five-hidden-layer neural networks with ten units in each layer, and the sinc function as activation. All weights are set randomly in each instance to show the kind of nonlinear manifolds we could potentially discover using such encoding/decoding functions. Because of the exceedingly high capacity of this nonlinear Autoencoder model the instances shown in the figure are quite diverse in shape.

Figure 13.10 Figure associated with Example 13.5. Nine random manifolds generated by a multi-layer neural network Autoencoder. See text for further details.

Example 13.6 Nonlinear Autoencoder using multi-layer neural networks

In this example we illustrate the use of a multi-layer neural network Autoencoder for learning a nonlinear manifold over the dataset shown in the top-left

panel of Figure 13.11. Here for both the encoder and decoder functions we arbitrarily use a three-hidden-layer fully connected network with ten units in each layer, and the tanh activation. We then tune the parameters of both functions together by minimizing the Least Squares cost given in Equation (10.53), and uncover the proper nonlinear manifold on which the dataset rests. In Figure 13.11 we show the learned manifold (top-right panel), the decoded version of the original dataset, i.e., the original dataset projected onto our learned manifold (bottom-left panel), and a *projection map* visualizing how all of the data in this space is projected onto the learned manifold via a vector-field plot (bottom-right panel).

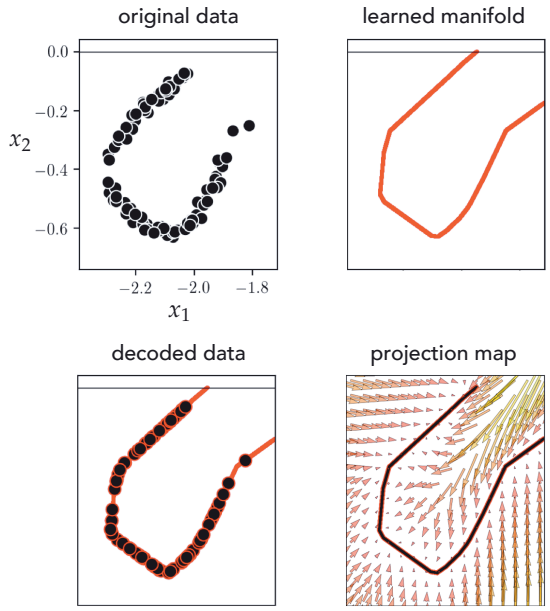

Figure 13.11 Figure associated with Example 13.6. See text for details.

13.3 Activation Functions

In principle one can use any (nonlinear) function as an activation for a fully connected neural network. For some time after their invention activations were chosen largely based on their biological plausibility, since this is the perspective in which neural networks themselves were at first largely regarded (as detailed in Section 13.2.5). Today activation functions are chosen based on practical considerations including our ability to properly optimize models which employ them as well as (of course) the general performance they provide. In this section

we briefly review popular historical and modern activation functions through several examples.

Example 13.7 The step and sigmoid activations

As broadly discussed in Section 13.2.5 the concept of neural networks was first introduced from a biological perspective where each unit of an architecture mimics a biological neuron in the human brain. Such neurons were thought to act somewhat like digital switches, being either completely "on" or "off" to transmitting information to connected cells. This belief naturally led to the use of a *step function* taking on just two values: 0 (off) and 1 (on). However, this kind of step function (which we discuss in the context of logistic regression in Section 6.2) leads to piece-wise flat cost functions (see, e.g., the left panel of Figure 6.3), which are extremely difficult to optimize using any local optimization technique. In the context of logistic regression this sort of problem is what led to the *logistic sigmoid*, and for the same practical reason the sigmoid function was one of the first popularly used activation functions. As a smooth approximation to the step function, the logistic sigmoid was viewed as a reasonable compromise between the desired neuronal model and the practical need to tune parameters properly.

While the logistic sigmoid performs very well in the comparatively simpler context of linear classification, when used as an activation function it often leads to a technical issue known as the *vanishing gradient* problem. Note how the logistic sigmoid function (shown in the top-left panel of Figure 13.12) maps almost all negative input values (except those near the origin) to output values very close to zero, and its derivative (shown in the bottom-left panel of the figure) maps input values away from the origin to output values very close to zero. These characteristics can cause the gradient of a neural network model employing sigmoid activations to shrink undesirably, preventing proper parameter tuning – a problem that balloons as more hidden layers are added.

In practice, neural network models employing the hyperbolic tangent function (tanh) typically perform better than the same network employing logistic sigmoid activations, because the function itself centers its output about the origin. However, since the derivative of tanh likewise maps input values away from the origin to output values very close to zero, neural networks employing the tanh activation can also suffer from the vanishing gradient problem.

Example 13.8 The Rectified Linear Unit (ReLU) activation

For decades after fully connected neural networks were first introduced, researchers employed almost exclusively logistic sigmoid activation functions, based on their biologically plausible nature (outlined in Section 13.2.5). Only in the early 2000s did some researchers begin to break away from this tradition, entertaining and testing alternative activation functions [60]. The simple ReLU function (see Example 13.1) was the first such activation function to be popularized.

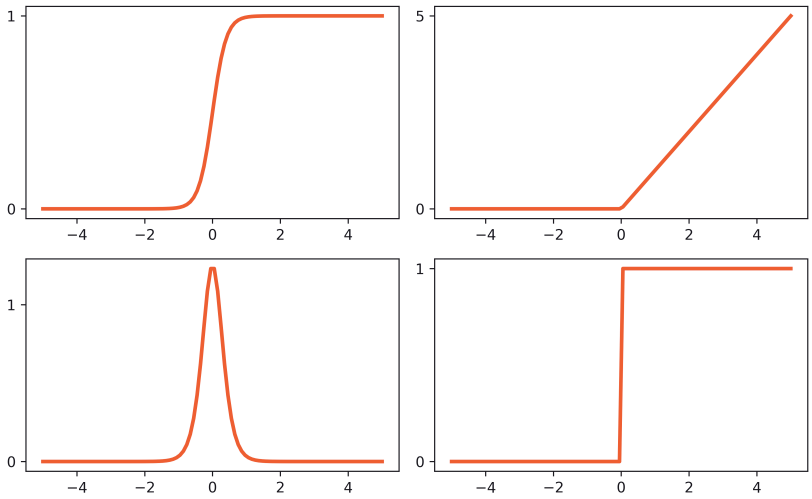

Figure 13.12 Figure associated with Examples 13.7 and 13.8. The logistic sigmoid function (top-left panel) and its derivative (bottom-left panel). The derivative here maps most input values away from the origin to output values very close to zero, which can cause the gradient of a network employing sigmoid activation to vanish during optimization, and thus prevent adequate parameter tuning. The ReLU function (top-right panel) and its derivative (bottom-right panel). While not as susceptible to vanishing gradient problems, neural networks employing the ReLU need to be initialized away from zero to prevent units from disappearing. See text for further details.

A computationally simpler function in comparison to the logistic sigmoid (which involves the use of both a log and exponential function), the ReLU has quickly become the most popular activation function in use today. Because the derivative of the ReLU function (plotted in the bottom-right panel of Figure 13.12) only maps negative input values to zero, networks employing this activation tend not to suffer (as severely) from the sort of vanishing gradient problem commonly found with the logistic sigmoid activation (detailed in the previous example). However, because of the shape of the ReLU itself (shown in the top-right panel of the figure) care still must be taken when initializing and training a network employing ReLU activations, as ReLU units themselves vanish as well at nonpositive inputs. For example, akin to the need to initialize the ReLU cost function detailed in Section 6.4 away from the origin where the optimization process will stall, a fully connected neural network employing ReLU activations should be initialized away from the origin to avoid too many of the units (and their gradients) from disappearing.

Example 13.9 The maxout activation
The *maxout* activation, defined as

$$a(x) = \max(v_0 + v_1 x,\ w_0 + w_1 x) \tag{13.30}$$

is a relative of the ReLU that takes the maximum of two linear combinations of the input (instead of one linear combination and zero, as is the case with ReLU). Four instances of such a maxout unit are plotted in Figure 13.13, where in each instance the parameters v_0, v_1, w_0, and w_1 are set at random. While this change seems algebraically rather minor, multi-layer neural network architectures employing the maxout activation tend to have certain advantages over those employing tanh and ReLU activations, including (i) fewer issues with problematic initializations, (ii) fewer issues with gradients vanishing, and (iii) empirically faster convergence with far fewer gradient descent steps [61]. These advantages come with a simple price: the maxout activation has twice as many internal parameters as either the ReLU or tanh, hence architectures built using them have roughly twice as many parameters to tune.

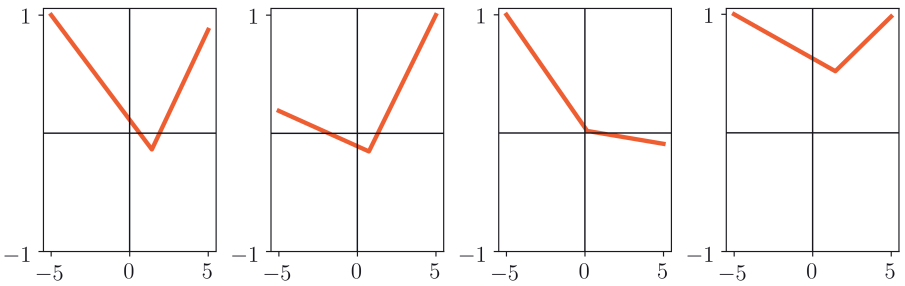

Figure 13.13 Figure associated with Example 13.9. Four instances of a maxout unit in Equation (13.30), with parameters randomly set in each case. See text for further details.

13.4 The Backpropagation Algorithm

The backpropagation algorithm, often referred to simply as backpropagation or backprop for short, is the jargon phrase used in machine learning to describe an approach to computing gradients algorithmically via a computer program that is especially effective for models employing multi-layer neural networks. Backprop is a special case of a more general scheme generally referred to as the *reverse mode of automatic differentiation*. The interested reader is encouraged to see Appendix Sections B.6 and B.7 for more on automatic differentiation.

Automatic differentiation allows us to turn over the tedious burden of computing gradients "by hand" to the tireless laborer that is the modern computer.

In other words, an automatic differentiator is an effective *calculator* that makes computing gradients of virtually any cost function a simple chore. Like many technical advances throughout history, automatic differentiation was discovered and rediscovered by different researchers in different areas of science and engineering at different times. This is precisely why this universally applicable concept (automatic differentiation) is referred to as *backpropagation* in the machine learning community, as this was the name given to it by its discoverers in this field.

13.5 Optimization of Neural Network Models

Typically cost functions associated with fully connected neural network models are highly nonconvex. In this section we address this issue, highlighting the local optimization techniques discussed in Appendix A of the text that are especially useful at properly tuning such models.

13.5.1 Nonconvexity

Models employing multi-layer neural networks are virtually always nonconvex. However, they often exhibit a variety of nonconvexity that we can fairly easily deal with using advanced optimization methods such as those detailed in Appendix A of the text. For example, in Section 6.2.4 we studied the shape of a Least Squares cost for logistic regression that employs a sigmoid based model. With one-dimensional input this model, written as

$$\text{model}\left(x, \Theta\right) = \sigma\left(w_0 + w_1 x\right) \tag{13.31}$$

can be interpreted, through the lens of neural networks, as a single unit of a single-hidden-layer neural network with scalar input and logistic sigmoid activation, where the weights of the final linear combination are fixed (with the bias set to zero and the weight touching $\sigma\left(\cdot\right)$ set to one). In the middle panel of Figure 6.3 we showed the surface of the Least Squares cost of this model over the simple dataset shown in Figure 6.2.

Examining the general shape of this cost function we can clearly see that it is nonconvex. Several portions of the cost surface, on either side of the *long narrow valley* containing the cost's global minimum, are almost completely *flat*. In general, cost functions employing neural network models have nonconvex shapes that share the kinds of basic characteristics seen in this figure: long narrow valleys, flat areas, and many saddle points and local minima.

These sorts of nonconvexities are problematic for *basic* first- and second-order optimization schemes. However, both frameworks can be extended to better deal with such eccentricities. This is especially true for gradient-based methods, as powerful modifications to the standard gradient descent method allow

it to easily deal with *long narrow valleys* and *flat areas* of nonconvex functions. These modifications include *momentum-based* (see Appendix Section A.2) and *normalized* gradient methods (see Appendix Section A.3). Combining these two modifications (see Appendix Section A.4), in addition to mini-batch optimization (see Section 7.8 and Appendix Section A.5), can further enhance gradient-based methods so that they can more easily minimize cost functions like the one shown in the middle panel of Figure 6.3, and neural network models in general. In short, the sort of nonconvexity encountered with neural network models is often manageable using advanced optimization tools.

Even when neural network cost functions have many *local minima*, these tend to lie at a depth close to that of their global minima, and thus tend to provide similar performance if found via local optimization. An abstract depiction of this sort of prototypical cost function, the kind often encountered with neural network models, is shown in the top panel of Figure 13.14.

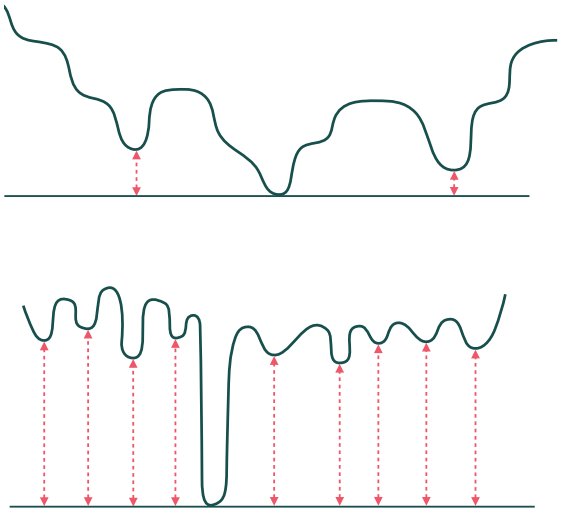

Figure 13.14 (top panel) An abstract depiction of the sort of prototypical cost function seen when employing neural network models. Such a cost may contain many saddle points, long narrow valleys, and local minima whose depth closely match its global minima. With advanced local optimization techniques we can easily traverse saddle points and long narrow valleys, determining such local minima. (bottom panel) An abstract depiction of the sort of worst-case scenario nonconvex function rarely encountered when using neural network models. Here the difference in depth between the cost's local and global minima is substantial, and therefore models employing parameters from each area will vary substantially in quality. Given the vast number of poor local minima, such a cost function is very difficult to minimize properly using local optimization.

This is starkly different than a kind of hypothetical, worst-case scenario nonconvex cost (not often encountered with neural networks) whose global minima

lie considerably lower than its local minima. An abstract depiction of such a function is shown in the bottom panel of Figure 13.14. There are no enhancements one can make to any local optimization method to enable it to effectively minimize such a cost function, with the only practical salve being to make many runs starting from random initial points to see if a global minimum can be reached by some individual run.

Example 13.10 Comparing first-order optimizers on a multi-layer neural network model

In this example we use $P = 50,000$ randomly selected data points from the MNIST dataset (see Example 7.10) to perform multi-class classification ($C = 10$) using a four-hidden-layer neural network with ten units per layer, and a tanh activation (this particular architecture was chosen arbitrarily for the purposes of this example). Here we compare the efficacy of three first-order optimizers: the standard gradient descent scheme (see Section 3.5), its component-normalized version (see Section A.3.2), and RMSProp (see Section A.4).

Each optimizer is used in both batch and mini-batch (using batch size of 200) regimes (see Section 7.8) to minimize the multi-class Softmax cost over this data. For all runs we initialize at the same starting point, and in each instance use the largest fixed steplength value of the form $\alpha = 10^{\gamma}$ (for integer γ) that produced convergence.

In Figure 13.15 we show the results of the batch (top row) and mini-batch (bottom row) versions, showing both cost and accuracy histories measuring the efficacy of each optimizer during the first ten epochs (or full sweeps through the dataset). In both the batch and mini-batch runs we can see how the component-normalized version of gradient descent and RMSProp significantly outperform the standard algorithm in terms of speed of convergence.

13.6 Batch Normalization

Previously, in Section 9.3, we saw how normalizing each input feature of a dataset significantly aids in speeding up parameter tuning, particularly with first-order optimization methods, by improving the shape of a cost function's contours (making them more "circular").

With our generic linear model

$$\text{model}\left(\mathbf{x}, \mathbf{w}\right) = w_0 + x_1 w_1 + \cdots + x_N w_N \tag{13.32}$$

standard normalization involves normalizing the distribution of each input dimension of a dataset $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$ by making the substitution

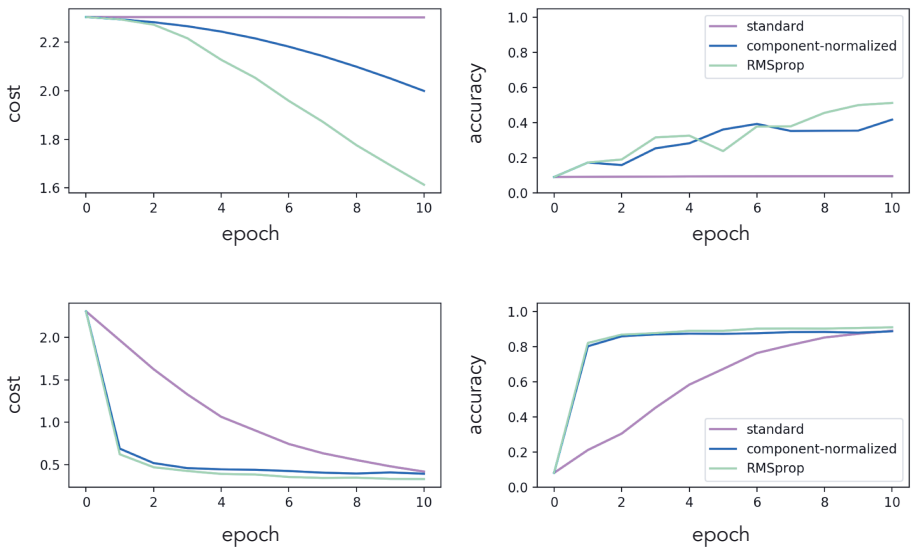

Figure 13.15 Figure associated with Example 13.10. Cost function (left column) and accuracy (right column) history plots comparing the efficacy of three first order optimizers in tuning the parameters of a four-hidden-layer neural network model over the MNIST dataset, using the batch (top row) and mini-batch (bottom row) versions of each optimizer. See text for further details.

$$x_{p,n} \longleftarrow \frac{x_{p,n} - \mu_n}{\sigma_n} \tag{13.33}$$

for the nth input dimension, where μ_n and σ_n are the mean and standard deviation along this dimension, respectively.

In this section we will learn how grafting a standard normalization step on to *each* hidden layer of an L-layer neural network model

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(L)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_L}^{(L)}\left(\mathbf{x}\right) w_{U_L} \tag{13.34}$$

where $f_1^{(L)}$ through $f_{U_L}^{(L)}$ are L-layer units as described in Section 13.2, similarly makes tuning the parameters of such a model significantly easier.

With this extended standard normalization technique, called *batch normalization* [62], we normalize not just the input to our fully connected network but the distribution of every unit in every hidden layer of the network as well. As we will see, doing this provides even greater optimization speed-up (than that provided by standard normalization of the input alone) for fully connected neural network models.

13.6.1 Batch normalization of single-hidden-layer units

Suppose first, for simplicity, that we are dealing with a single-layer neural network model. Setting $L = 1$ in Equation (13.34) gives this model as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1^{(1)}\left(\mathbf{x}\right) w_1 + \cdots + f_{U_1}^{(1)}\left(\mathbf{x}\right) w_{U_1}. \tag{13.35}$$

We now extend the basic standard normalization scheme, applying the same normalization concept to every "weight-touching" distribution of this model. Of course here the *input features* no longer touch the weights of the final linear combination (i.e., $w_1, w_2, ..., w_{U_1}$). They instead touch the weights internal to the single-layer units themselves. We can see this more easily by analyzing the jth single-layer unit in this network

$$f_j^{(1)}\left(\mathbf{x}\right) = a\left(w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n\right) \tag{13.36}$$

wherein the nth dimension of the input x_n only touches the internal weight $w_{n,j}^{(1)}$. Thus in standard normalizing the input we directly affect the contours of a cost function only along the weights internal to the single-layer units. To affect the contours of a cost function with respect to weights external to the first hidden layer (here the weights of the final linear combination) we must naturally normalize the *output* of the first hidden layer.

Putting these output values in a set – and denoting it by $\left\{f_j^{(1)}\left(\mathbf{x}_p\right)\right\}_{p=1}^{P}$ – we would naturally want to standard normalize *its* distribution as

$$f_j^{(1)}\left(\mathbf{x}\right) \longleftarrow \frac{f_j^{(1)}\left(\mathbf{x}\right) - \mu_{f_j^{(1)}}}{\sigma_{f_j^{(1)}}} \tag{13.37}$$

where the mean $\mu_{f_j^{(1)}}$ and standard deviation $\sigma_{f_j^{(1)}}$ are given as

$$\mu_{f_j^{(1)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(1)}\left(\mathbf{x}_p\right) \qquad \text{and} \qquad \sigma_{f_j^{(1)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(1)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(1)}}\right)^2}. \tag{13.38}$$

Note importantly that, unlike our input features, the output of the single-layer units (and hence their distributions) change every time the internal parameters of our model are changed, e.g., during each step of gradient descent. The constant alteration of these distributions is referred to as *internal covariate shift* in the jargon of machine learning, or just *covariate shift* for short, and implies that if we are to carry over the principle of standard normalization completely we will need to normalize the output of the first hidden layer at *every* step of parameter tuning. In other words, we need to graft standard normalization directly into the hidden layer of our architecture itself.

Below we show a generic recipe for doing just this, a simple extension of the recursive recipe for single-layer units given in Section 13.2.1.

Recursive recipe for batch-normalized single-layer units

1. Choose an activation function $a(\cdot)$
2. Compute the linear combination $v = w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n$
3. Pass the result through activation and form $f^{(1)}(\mathbf{x}) = a(v)$
4. Standard normalize $f^{(1)}$ as $f^{(1)}(\mathbf{x}) \longleftarrow \frac{f^{(1)}(\mathbf{x}) - \mu_{f^{(1)}}}{\sigma_{f^{(1)}}}$

Example 13.11 Visualizing internal covariate shift in a single-layer network
In this example we illustrate the internal covariate shift in a single-layer neural network model using two ReLU units $f_1^{(1)}$ and $f_2^{(1)}$, applied to performing two-class classification of the toy dataset introduced in Example 11.7. We run 5000 steps of gradient descent to minimize the two-class Softmax cost using this single-layer network, where we standard normalize the input data.

In Figure 13.16 we show the progression of this gradient descent run, plotting the tuples $\left\{\left(f_1^{(1)}\left(\mathbf{x}_p\right), f_2^{(1)}\left(\mathbf{x}_p\right)\right)\right\}_{p=1}^{P}$ at three of the steps taken during the run. The top and bottom panels respectively show the covariate shift and the complete cost function history curve, where the current step of the optimization is marked on the curve with a red dot.

As can be seen in the top row of Figure 13.16, the distribution of these tuples change dramatically as the gradient descent algorithm progresses. We can intuit (from our previous discussions on input normalization) that this sort of shifting distribution negatively affects the speed at which gradient descent can properly minimize our cost function.

Next, we repeat this experiment using the same gradient descent settings but now with batch-normalized single-layer units, and plot the results in a similar manner in Figure 13.17. Notice how the distribution of activation outputs stays considerably more stable as gradient descent progresses.

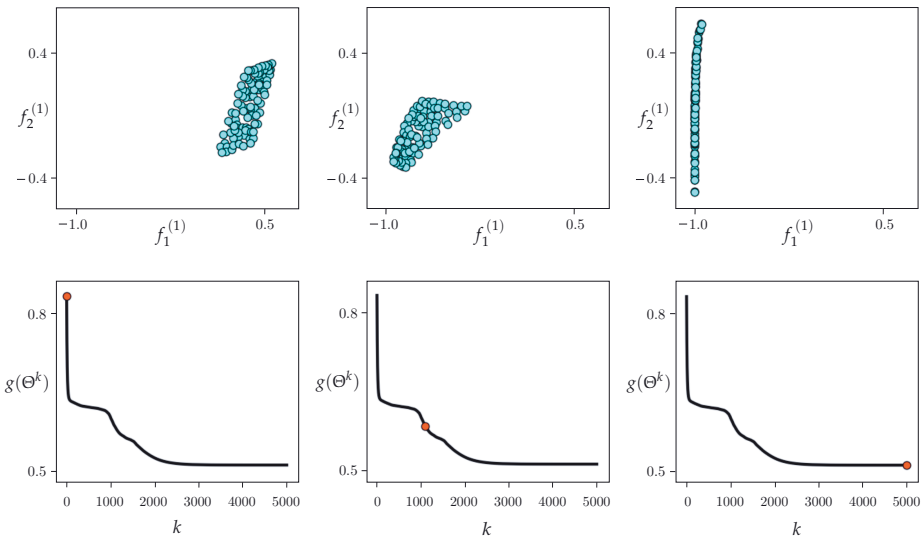

Figure 13.16 Figure associated with Example 13.11. See text for details.

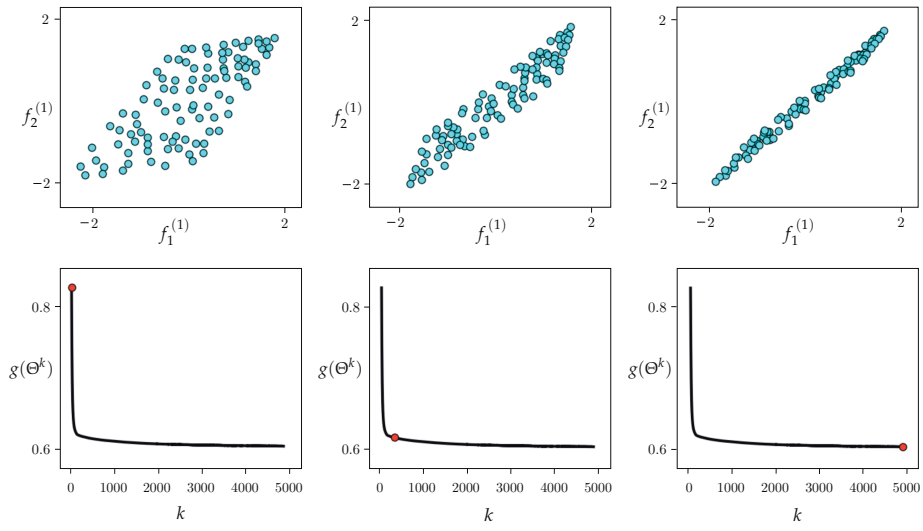

Figure 13.17 Figure associated with Example 13.11. See text for details.

13.6.2 Batch normalization of multi-hidden-layer units

Suppose for a moment that our fully connected neural network has just two hidden layers (i.e., $L = 2$ in Equation (13.34)), and that we have grafted a standard normalization step onto the first hidden layer of our network as described in the previous section, so that now our two-hidden-layer units touch the final linear

combination weights (i.e., $w_1, w_2, ..., w_{U_2}$). To temper an associated cost function with respect to w_j we normalize the associated distribution of our jth unit via

$$f_j^{(2)}(\mathbf{x}) \longleftarrow \frac{f_j^{(2)}(\mathbf{x}) - \mu_{f_j^{(2)}}}{\sigma_{f_j^{(2)}}} \tag{13.39}$$

where the mean $\mu_{f_j^{(2)}}$ and standard deviation $\sigma_{f_j^{(2)}}$ are defined as

$$\mu_{f_j^{(2)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(2)}\left(\mathbf{x}_p\right) \qquad \text{and} \qquad \sigma_{f_j^{(2)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(2)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(2)}}\right)^2}. \tag{13.40}$$

As was the case in studying the single-layer case in the previous section, here too we will need to graft this step on to the second hidden layer of our network so that whenever its parameters change (e.g., during each step of a local optimizer) the distribution of this unit remains normalized.

Extending this concept to a general L-hidden-layer neural network model in Equation (13.34) we will normalize the output of every hidden layer of the network. Thus in general, for L-layer units once we have standard normalized the output of every layer preceding it we standard normalize the jth unit of the Lth hidden layer

$$f_j^{(L)}(\mathbf{x}) = a\left(w_{0,j}^{(L)} + \sum_{i=1}^{U_{L-1}} w_{i,j}^{(L)} f_i^{(L-1)}(\mathbf{x})\right) \tag{13.41}$$

via the substitution

$$f_j^{(L)}(\mathbf{x}) \longleftarrow \frac{f_j^{(L)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(L)}}}{\sigma_{f_j^{(L)}}} \tag{13.42}$$

where the mean $\mu_{f_j^{(L)}}$ and standard deviation $\sigma_{f_j^{(L)}}$ are defined as

$$\mu_{f_j^{(L)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(L)}\left(\mathbf{x}_p\right) \qquad \text{and} \qquad \sigma_{f_j^{(L)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(L)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(L)}}\right)^2}. \tag{13.43}$$

As with the single-layer case, we can still construct each batch-normalized unit recursively since all we must do is insert a standard normalization step into the end of each layer as summarized in the following recipe (akin to the recipe given for a general L-layer unit in Section 13.2.3).

Recursive recipe for batch-normalized L-layer units

1. Choose an activation function $a(\cdot)$
2. Construct U_{L-1} batch-normalized $(L-1)$-layer units $f_i^{(L-1)}(\mathbf{x})$ for $i = 1, 2, ..., U_{L-1}$
3. Compute the linear combination $v = w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})$
4. Pass the result through activation and form $f^{(L)}(\mathbf{x}) = a(v)$
5. Standard normalize $f^{(L)}$ via $f^{(L)}(\mathbf{x}) \longleftarrow \frac{f^{(L)}(\mathbf{x}) - \mu_{f^{(L)}}}{\sigma_{f^{(L)}}}$

When employing a stochastic or mini-batch first-order method for optimization (see Section 7.8), normalization of the architecture is performed precisely as detailed here, on each individual mini-batch. Also note that, in practice, the batch normalization formula in Equation (13.42) is often parameterized as

$$f^{(L)}(\mathbf{x}) \longleftarrow \alpha \frac{f^{(L)}(\mathbf{x}) - \mu_{f^{(L)}}}{\sigma_{f^{(L)}}} + \beta \tag{13.44}$$

where inclusion of the tunable parameters α and β (which are tuned along with the other parameters of a batch-normalized network) allows for greater flexibility. However, even without these extra parameters we can achieve significant improvement in optimization speed when tuning fully connected neural network models with the employment of (unparameterized) batch normalization.

Example 13.12 Visualizing internal covariate shift in a multi-layer network
In this example we illustrate the covariate shift in a four-hidden-layer network with two units per layer, using the ReLU activation and the same dataset employed in Example 13.11. We then compare this to the covariate shift present in the batch-normalized version of the same network. We use just two units per layer so that we can visualize the distribution of activation outputs of each layer.

Beginning with the unnormalized version of the network we can see that – as with the single-layer case in Example 13.11 – the covariate shift of this network (shown in the top row of Figure 13.18) is considerable. The distribution of each hidden layer's units is shown here, with the output tuples of the ℓth hidden layer $\left\{\left(f_1^{(\ell)}\left(\mathbf{x}_p\right), f_2^{(\ell)}\left(\mathbf{x}_p\right)\right)\right\}_{p=1}^{P}$ colored in cyan for $\ell = 1$, in magenta for $\ell = 2$, in lime green for $\ell = 3$, and in orange for $\ell = 4$.

Performing batch normalization on each layer of this network helps considerably in taming this covariate shift. In Figure 13.19 we show the result of running the same experiment, using the same initialization, activation, and dataset, but

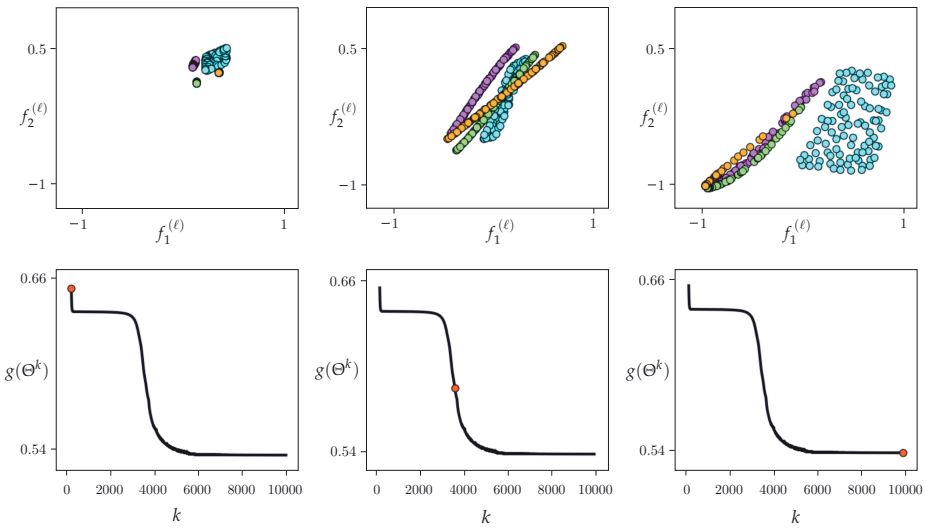

Figure 13.18 Figure associated with Example 13.12. See text for details.

this time using the batch-normalized version of the network. Studying the figure from left to right, as gradient descent progresses, we can see once again that the distribution of each layer's activation outputs remains much more stable than previously.

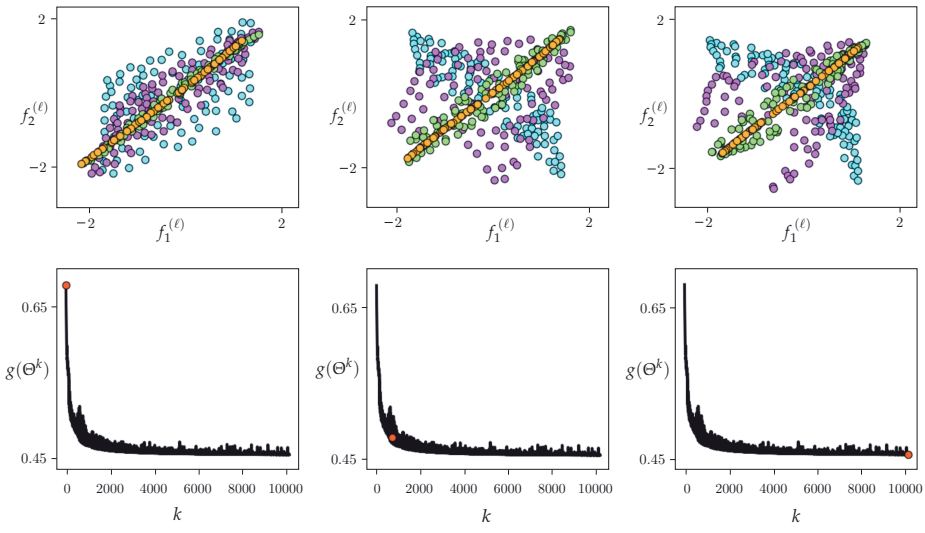

Figure 13.19 Figure associated with Example 13.12. See text for details.

Example 13.13 Standard versus batch normalization on MNIST

In this example we illustrate the benefit of batch normalization in terms of speeding up optimization via gradient descent on a dataset of $P = 50,000$ randomly chosen handwritten digits from the MNIST dataset (introduced in Example 7.11). In Figure 13.20 we show cost (left panel) and classification accuracy (right panel) histories of ten epochs of gradient descent, using the largest steplength of the form 10^{γ} (for integer γ) we found that produced adequate convergence. We compare the standard and batch-normalized version of a four-hidden-layer neural network with ten units per layer and ReLU activation. Here we can see, both in terms of cost function value and number of misclassifications (accuracy), that the batch-normalized version allows for much more rapid minimization via gradient descent.

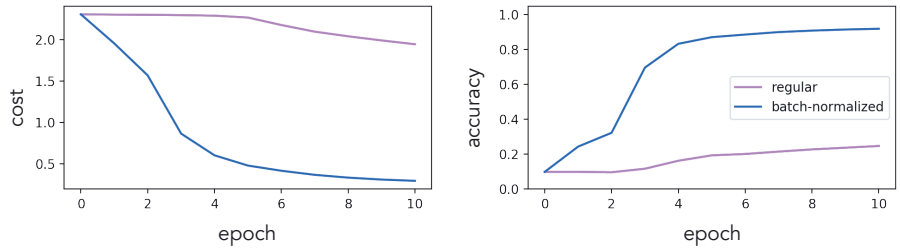

Figure 13.20 Figure associated with Example 13.13. See text for details.

13.6.3 Evaluation of new data points in batch-normalized networks

An important point to remember when employing a batch-normalized neural network is that we must treat new data points (not used in training) precisely as we treat training data. This means that the final normalization constants determined during training (i.e., the various means and standard deviations of the input as well as those for each hidden layer output) must be saved and reused in order to properly evaluate new data points. More specifically, all normalization constants in a batch-normalized network should be *fixed* to the values computed at the final step of training (e.g., at the best step of gradient descent) when evaluating new data points.

13.7 Cross-Validation via Early Stopping

Being highly parameterized, the optimization of cost functions associated with fully connected neural networks, particularly those employing many hidden layers, can require significant computation. Because of this *early stopping based*

regularization (as described in Section 11.6.2), which involves learning parameters to minimize validation error during a single run of optimization, is a popular cross-validation technique when employing fully connected multi-layer networks. The notion of early stopping is also the basis for specialized ensembling techniques which aim at producing a set of neural network models for bagging (introduced in Section 11.9) at minimal computational cost (see, e.g., [63, 64]).

Example 13.14 Early stopping and regression

In this example we illustrate the early stopping procedure using a simple nonlinear regression dataset (split into $\frac{2}{3}$ training and $\frac{1}{3}$ validation), and a three-hidden-layer neural network with ten units per layer, and with tanh activation. Three different steps from a single run of gradient descent (for a total of $10,000$ steps) is illustrated in Figure 13.21, one per each column, with the resulting fit at each step shown over the original (first row), training (second row), and validation data (third row). Stopping the gradient descent early after taking (around) 2000 steps provides, for this training-validation split of the original data, a fine nonlinear model for the entire dataset.

Example 13.15 Early stopping and handwritten digit classification

In this example we use early stopping based regularization to determine the optimal settings of a two-hidden-layer neural network, with 100 units per layer and ReLU activation, over the MNIST dataset of handwritten digits first described in Example 7.10. This multi-class dataset ($C = 10$) consists of $P = 50,000$ points in the training and $10,000$ points in the validation set. With a batch size of 500 we run 100 epochs of the standard mini-batch gradient descent scheme, resulting in the training (blue) and validation (yellow) cost function (left panel) and accuracy (right panel) history curves shown in Figure 13.22. Employing the multi-class Softmax cost, we found the optimal epoch with this setup achieved around 99 percent accuracy on the training set, and around 96 percent accuracy on the validation set. One can introduce enhancements like those discussed in the previous sections of this chapter to improve these accuracies further. For comparison, a linear classifier – trained/validated on the same data – achieved 94 and 92 percent training and validation accuracies, respectively.

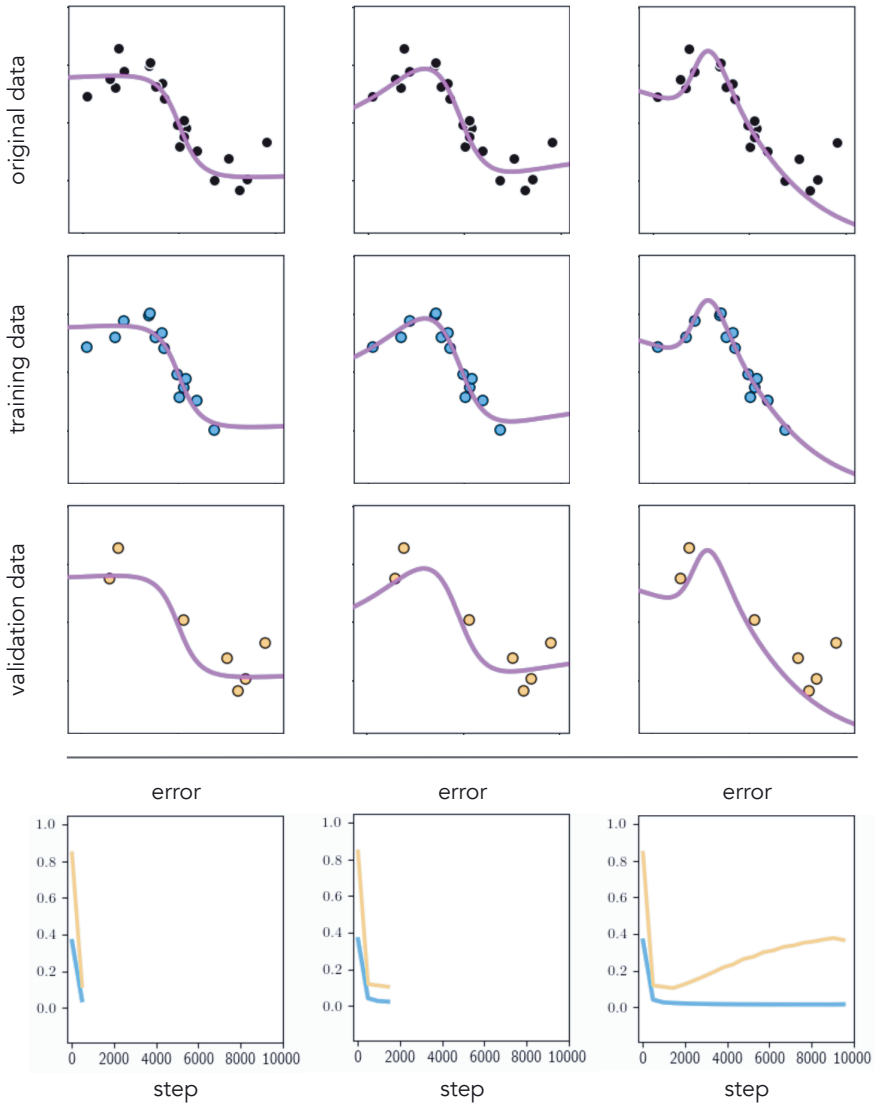

Figure 13.21 Figure associated with Example 13.14. See text for details.

13.8 Conclusion

In this chapter we described a range of technical matters associated with fully connected neural networks, which were first introduced in Section 11.2.3.

We began by carefully describing single- and multi-layer neural network architectures in Section 13.2, followed by a discussion of activation functions in Section 13.3, backpropagation in Section 13.4, and the nonconvexity of cost

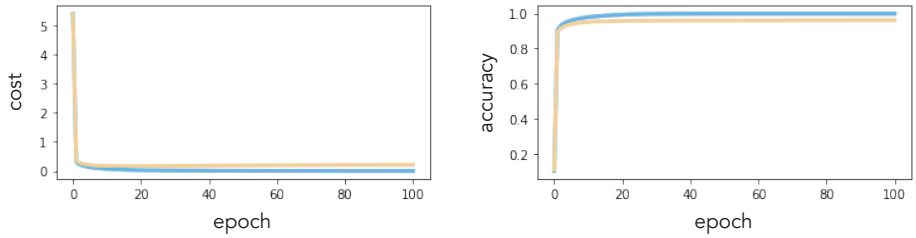

Figure 13.22 Figure associated with Example 13.15. See text for details.

functions over neural network models in Section 13.5. Batch normalization – a natural extension of the standard normalization procedure described in Section 9.4 – was then explored in Section 13.6. Finally, in Section 13.7 we discussed the use of regularization (and in particular, early stopping) based cross-validation – first described in great detail in Section 11.6 – with fully connected neural network models.

13.9 Exercises

† The data required to complete the following exercises can be downloaded from the text's github repository at `github.com/jermwatt/machine_learning_refined`

13.1 Two-class classification with neural networks

Repeat the two-class classification experiment described in Example 13.4 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the result shown in the top row of Figure 13.9, but can verify your result via checking that you can achieve perfect classification of the data.

13.2 Multi-class classification with neural networks

Repeat the multi-class classification experiment described in Example 13.4 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the result shown in the bottom row of Figure 13.9, but can verify your result via checking that you can achieve perfect classification of the data.

13.3 Number of weights to learn in a neural network

(a) Find the total number Q of tunable parameters in a general L-hidden-layer neural network, in terms of variables expressed in the `layer_sizes` list in Section 13.2.6.

(b) Based on your answer in part (a), explain how the input dimension N and number of data points P each contributes to Q. How is this different from what you saw with kernel methods in the previous chapter?

13.4 Nonlinear Autoencoder using neural networks

Repeat the Autoencoder experiment described in Example 13.6 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the projection map shown in the bottom-right panel of Figure 13.11.

13.5 The maxout activation function

Repeat Exercise 13.4 using the maxout activation (detailed in Example 13.9).

13.6 Comparing advanced first-order optimizers I

Repeat the first set of experiments described in Example 13.10, and produce plots like those shown in the top row of Figure 13.15. Your plots may not look precisely like those shown in this figure (but they should look similar).

13.7 Comparing advanced first-order optimizers II

Repeat the second set of experiments described in Example 13.10, and produce plots like those shown in the bottom row of Figure 13.15. Your plots may not look precisely like those shown in this figure (but they should look similar).

13.8 Batch normalization

Repeat the experiment described in Example 13.13, and produce plots like those shown in Figure 13.20. Your plots may not look precisely like those shown in this figure (but they should look similar).

13.9 Early stopping cross-validation

Repeat the experiment described in Example 13.14. You need not reproduce all the panels shown in Figure 13.21. However, you should plot the fit provided by the weights associated with the minimum validation error on top of the entire dataset.

13.10 Handwritten digit recognition using neural networks

Repeat the experiment described in Example 13.15, and produce cost/accuracy history plots like the ones shown in Figure 13.22. You may not reproduce exactly what is reported based on your particular implementation. However, you should be able to achieve similar results as reported in Example 13.15.